

# HANDBOOK OF MAGMA FUNCTIONS

## Volume 6

### **Finitely-presented Groups**

John Cannon      Wieb Bosma

Claus Fieker      Allan Steel

Editors

Version 2.19

**Sydney**

December 17, 2012



# HANDBOOK OF MAGMA FUNCTIONS

Editors:

*John Cannon      Wieb Bosma      Claus Fieker      Allan Steel*

Handbook Contributors:

*Geoff Bailey, Wieb Bosma, Gavin Brown, Nils Bruin, John Cannon, Jon Carlson, Scott Contini, Bruce Cox, Brendan Creutz, Steve Donnelly, Tim Dokchitser, Willem de Graaf, Andreas-Stephan Elsenhans, Claus Fieker, Damien Fisher, Volker Gebhardt, Sergei Haller, Michael Harrison, Florian Hess, Derek Holt, David Howden, Al Kasprzyk, Markus Kirschmer, David Kohel, Axel Kohnert, Dimitri Leemans, Paulette Lieby, Graham Matthews, Scott Murray, Eamonn O'Brien, Dan Roozmond, Ben Smith, Bernd Souvignier, William Stein, Allan Steel, Damien Stehlé, Nicole Sutherland, Don Taylor, Bill Unger, Alexa van der Waall, Paul van Wamelen, Helena Verrill, John Voight, Mark Watkins, Greg White*

Production Editors:

*Wieb Bosma      Claus Fieker      Allan Steel      Nicole Sutherland*

HTML Production:

*Claus Fieker      Allan Steel*



## VOLUME 6: OVERVIEW

<b>X</b>	<b>FINITELY-PRESENTED GROUPS . . . . .</b>	<b>2037</b>
69	ABELIAN GROUPS	2039
70	FINITELY PRESENTED GROUPS	2075
71	FINITELY PRESENTED GROUPS: ADVANCED	2201
72	POLYCYCLIC GROUPS	2247
73	BRAID GROUPS	2287
74	GROUPS DEFINED BY REWRITE SYSTEMS	2337
75	AUTOMATIC GROUPS	2355
76	GROUPS OF STRAIGHT-LINE PROGRAMS	2375
77	FINITELY PRESENTED SEMIGROUPS	2385
78	MONOIDS GIVEN BY REWRITE SYSTEMS	2397

# VOLUME 6: CONTENTS

<b>X</b>	<b>FINITELY-PRESENTED GROUPS</b>	<b>2037</b>
69	ABELIAN GROUPS . . . . .	2039
69.1	<i>Introduction</i>	2041
69.2	<i>Construction of a Finitely Presented Abelian Group and its Elements</i>	2041
69.2.1	The Free Abelian Group	2041
69.2.2	Relations	2042
69.2.3	Specification of a Presentation	2043
69.2.4	Accessing the Defining Generators and Relations	2044
69.3	<i>Construction of a Generic Abelian Group</i>	2045
69.3.1	Specification of a Generic Abelian Group	2045
69.3.2	Accessing Generators	2048
69.3.3	Computing Abelian Group Structure	2048
69.4	<i>Elements</i>	2050
69.4.1	Construction of Elements	2050
69.4.2	Representation of an Element	2051
69.4.3	Arithmetic with Elements	2052
69.5	<i>Construction of Subgroups and Quotient Groups</i>	2053
69.5.1	Construction of Subgroups	2053
69.5.2	Construction of Quotient Groups	2055
69.6	<i>Standard Constructions and Conversions</i>	2055
69.7	<i>Operations on Elements</i>	2057
69.7.1	Order of an Element	2057
69.7.2	Discrete Logarithm	2058
69.7.3	Equality and Comparison	2059
69.8	<i>Invariants of an Abelian Group</i>	2060
69.9	<i>Canonical Decomposition</i>	2060
69.10	<i>Set-Theoretic Operations</i>	2061
69.10.1	Functions Relating to Group Order	2061
69.10.2	Membership and Equality	2061
69.10.3	Set Operations	2062
69.11	<i>Coset Spaces</i>	2063
69.11.1	Coercions Between Groups and Subgroups	2063
69.12	<i>Subgroup Constructions</i>	2064
69.13	<i>Subgroup Chains</i>	2065
69.14	<i>General Group Properties</i>	2065
69.14.1	Properties of Subgroups	2066
69.14.2	Enumeration of Subgroups	2066
69.15	<i>Representation Theory</i>	2068
69.16	<i>The Hom Functor</i>	2068
69.17	<i>Automorphism Groups</i>	2070
69.18	<i>Cohomology</i>	2070
69.19	<i>Homomorphisms</i>	2070
69.20	<i>Bibliography</i>	2073

70	FINITELY PRESENTED GROUPS . . . . .	2075
70.1	<i>Introduction</i>	2079
70.1.1	Overview of Facilities	2079
70.1.2	The Construction of Finitely Presented Groups	2079
70.2	<i>Free Groups and Words</i>	2080
70.2.1	Construction of a Free Group	2080
70.2.2	Construction of Words	2081
70.2.3	Access Functions for Words	2081
70.2.4	Arithmetic Operators for Words	2083
70.2.5	Comparison of Words	2084
70.2.6	Relations	2085
70.3	<i>Construction of an FP-Group</i>	2087
70.3.1	The Quotient Group Constructor	2087
70.3.2	The FP-Group Constructor	2089
70.3.3	Construction from a Finite Permutation or Matrix Group	2090
70.3.4	Construction of the Standard Presentation for a Coxeter Group	2092
70.3.5	Conversion from a Special Form of FP-Group	2093
70.3.6	Construction of a Standard Group	2094
70.3.7	Construction of Extensions	2096
70.3.8	Accessing the Defining Generators and Relations	2098
70.4	<i>Homomorphisms</i>	2098
70.4.1	General Remarks	2098
70.4.2	Construction of Homomorphisms	2099
70.4.3	Accessing Homomorphisms	2099
70.4.4	Computing Homomorphisms to Finite Groups	2102
70.4.5	The $L_2$ -Quotient Algorithm	2110
70.4.6	Infinite $L_2$ quotients	2117
70.4.7	Searching for Isomorphisms	2121
70.5	<i>Abelian, Nilpotent and Soluble Quotient</i>	2123
70.5.1	Abelian Quotient	2123
70.5.2	$p$ -Quotient	2126
70.5.3	The Construction of a $p$ -Quotient	2127
70.5.4	Nilpotent Quotient	2129
70.5.5	Soluble Quotient	2135
70.6	<i>Subgroups</i>	2138
70.6.1	Specification of a Subgroup	2138
70.6.2	Index of a Subgroup: The Todd-Coxeter Algorithm	2140
70.6.3	Implicit Invocation of the Todd-Coxeter Algorithm	2145
70.6.4	Constructing a Presentation for a Subgroup	2146
70.7	<i>Subgroups of Finite Index</i>	2150
70.7.1	Low Index Subgroups	2150
70.7.2	Subgroup Constructions	2159
70.7.3	Properties of Subgroups	2164
70.8	<i>Coset Spaces and Tables</i>	2168
70.8.1	Coset Tables	2168
70.8.2	Coset Spaces: Construction	2170
70.8.3	Coset Spaces: Elementary Operations	2171
70.8.4	Accessing Information	2172
70.8.5	Double Coset Spaces: Construction	2176
70.8.6	Coset Spaces: Selection of Cosets	2177
70.8.7	Coset Spaces: Induced Homomorphism	2178
70.9	<i>Simplification</i>	2181
70.9.1	Reducing Generating Sets	2181
70.9.2	Tietze Transformations	2181
70.10	<i>Representation Theory</i>	2192
70.11	<i>Small Group Identification</i>	2196

70.11.1	Concrete Representations of Small Groups	2198
70.12	<i>Bibliography</i>	2198
71	FINITELY PRESENTED GROUPS: ADVANCED . . . . .	2201
71.1	<i>Introduction</i>	2203
71.2	<i>Low Level Operations on Presentations and Words</i>	2203
71.2.1	Modifying Presentations	2204
71.2.2	Low Level Operations on Words	2206
71.3	<i>Interactive Coset Enumeration</i>	2208
71.3.1	Introduction	2208
71.3.2	Constructing and Modifying a Coset Enumeration Process	2209
71.3.3	Starting and Restarting an Enumeration	2214
71.3.4	Accessing Information	2216
71.3.5	Induced Permutation Representations	2225
71.3.6	Coset Spaces and Transversals	2226
71.4	<i>p-Quotients (Process Version)</i>	2229
71.4.1	The $p$ -Quotient Process	2229
71.4.2	Using $p$ -Quotient Interactively	2230
71.5	<i>Soluble Quotients</i>	2239
71.5.1	Introduction	2239
71.5.2	Construction	2239
71.5.3	Calculating the Relevant Primes	2241
71.5.4	The Functions	2241
71.6	<i>Bibliography</i>	2245
72	POLYCYCLIC GROUPS . . . . .	2247
72.1	<i>Introduction</i>	2249
72.2	<i>Polycyclic Groups and Polycyclic Presentations</i>	2249
72.2.1	Introduction	2249
72.2.2	Specification of Elements	2250
72.2.3	Access Functions for Elements	2250
72.2.4	Arithmetic Operations on Elements	2251
72.2.5	Operators for Elements	2252
72.2.6	Comparison Operators for Elements	2252
72.2.7	Specification of a Polycyclic Presentation	2253
72.2.8	Properties of a Polycyclic Presentation	2257
72.3	<i>Subgroups, Quotient Groups, Homomorphisms and Extensions</i>	2257
72.3.1	Construction of Subgroups	2257
72.3.2	Coercions Between Groups and Subgroups	2258
72.3.3	Construction of Quotient Groups	2259
72.3.4	Homomorphisms	2259
72.3.5	Construction of Extensions	2260
72.3.6	Construction of Standard Groups	2260
72.4	<i>Conversion between Categories</i>	2263
72.5	<i>Access Functions for Groups</i>	2264
72.6	<i>Set-Theoretic Operations in a Group</i>	2265
72.6.1	Functions Relating to Group Order	2265
72.6.2	Membership and Equality	2265
72.6.3	Set Operations	2266
72.7	<i>Coset Spaces</i>	2267
72.8	<i>The Subgroup Structure</i>	2270
72.8.1	General Subgroup Constructions	2270
72.8.2	Subgroup Constructions Requiring a Nilpotent Covering Group	2270
72.9	<i>General Group Properties</i>	2271

72.9.1	General Properties of Subgroups	2272
72.9.2	Properties of Subgroups Requiring a Nilpotent Covering Group	2272
72.10	<i>Normal Structure and Characteristic Subgroups</i>	2274
72.10.1	Characteristic Subgroups and Subgroup Series	2274
72.10.2	The Abelian Quotient Structure of a Group	2278
72.11	<i>Conjugacy</i>	2278
72.12	<i>Representation Theory</i>	2279
72.13	<i>Power Groups</i>	2285
72.14	<i>Bibliography</i>	2286
73	<b>BRAID GROUPS . . . . .</b>	<b>2287</b>
73.1	<i>Introduction</i>	2289
73.1.1	Lattice Structure and Simple Elements	2290
73.1.2	Representing Elements of a Braid Group	2291
73.1.3	Normal Form for Elements of a Braid Group	2292
73.1.4	Mixed Canonical Form and Lattice Operations	2293
73.1.5	Conjugacy Testing and Conjugacy Search	2294
73.2	<i>Constructing and Accessing Braid Groups</i>	2296
73.3	<i>Creating Elements of a Braid Group</i>	2297
73.4	<i>Working with Elements of a Braid Group</i>	2303
73.4.1	Accessing Information	2303
73.4.2	Computing Normal Forms of Elements	2306
73.4.3	Arithmetic Operators and Functions for Elements	2309
73.4.4	Boolean Predicates for Elements	2313
73.4.5	Lattice Operations	2317
73.4.6	Invariants of Conjugacy Classes	2321
73.5	<i>Homomorphisms</i>	2330
73.5.1	General Remarks	2330
73.5.2	Constructing Homomorphisms	2330
73.5.3	Accessing Homomorphisms	2331
73.5.4	Representations of Braid Groups	2334
73.6	<i>Bibliography</i>	2336
74	<b>GROUPS DEFINED BY REWRITE SYSTEMS . . . . .</b>	<b>2337</b>
74.1	<i>Introduction</i>	2339
74.1.1	Terminology	2339
74.1.2	The Category of Rewrite Groups	2339
74.1.3	The Construction of a Rewrite Group	2339
74.2	<i>Constructing Confluent Presentations</i>	2340
74.2.1	The Knuth-Bendix Procedure	2340
74.2.2	Defining Orderings	2341
74.2.3	Setting Limits	2343
74.2.4	Accessing Group Information	2345
74.3	<i>Properties of a Rewrite Group</i>	2347
74.4	<i>Arithmetic with Words</i>	2348
74.4.1	Construction of a Word	2348
74.4.2	Element Operations	2349
74.5	<i>Operations on the Set of Group Elements</i>	2351
74.6	<i>Homomorphisms</i>	2353
74.6.1	General Remarks	2353
74.6.2	Construction of Homomorphisms	2353
74.7	<i>Conversion to a Finitely Presented Group</i>	2354
74.8	<i>Bibliography</i>	2354

75	AUTOMATIC GROUPS . . . . .	2355
75.1	<i>Introduction</i>	2357
75.1.1	Terminology	2357
75.1.2	The Category of Automatic Groups	2357
75.1.3	The Construction of an Automatic Group	2357
75.2	<i>Creation of Automatic Groups</i>	2358
75.2.1	Construction of an Automatic Group	2358
75.2.2	Modifying Limits	2359
75.2.3	Accessing Group Information	2363
75.3	<i>Properties of an Automatic Group</i>	2364
75.4	<i>Arithmetic with Words</i>	2366
75.4.1	Construction of a Word	2366
75.4.2	Operations on Elements	2367
75.5	<i>Homomorphisms</i>	2369
75.5.1	General Remarks	2369
75.5.2	Construction of Homomorphisms	2370
75.6	<i>Set Operations</i>	2370
75.7	<i>The Growth Function</i>	2372
75.8	<i>Bibliography</i>	2373
76	GROUPS OF STRAIGHT-LINE PROGRAMS . . . . .	2375
76.1	<i>Introduction</i>	2377
76.2	<i>Construction of an SLP-Group and its Elements</i>	2377
76.2.1	Structure Constructors	2377
76.2.2	Construction of an Element	2378
76.3	<i>Arithmetic with Elements</i>	2378
76.3.1	Accessing the Defining Generators and Relations	2378
76.4	<i>Addition of Extra Generators</i>	2379
76.5	<i>Creating Homomorphisms</i>	2379
76.6	<i>Operations on Elements</i>	2381
76.6.1	Equality and Comparison	2381
76.7	<i>Set-Theoretic Operations</i>	2381
76.7.1	Membership and Equality	2381
76.7.2	Set Operations	2382
76.7.3	Coercions Between Related Groups	2383
76.8	<i>Bibliography</i>	2383
77	FINITELY PRESENTED SEMIGROUPS . . . . .	2385
77.1	<i>Introduction</i>	2387
77.2	<i>The Construction of Free Semigroups and their Elements</i>	2387
77.2.1	Structure Constructors	2387
77.2.2	Element Constructors	2388
77.3	<i>Elementary Operators for Words</i>	2388
77.3.1	Multiplication and Exponentiation	2388
77.3.2	The Length of a Word	2388
77.3.3	Equality and Comparison	2389
77.4	<i>Specification of a Presentation</i>	2390
77.4.1	Relations	2390
77.4.2	Presentations	2390
77.4.3	Accessing the Defining Generators and Relations	2391
77.5	<i>Subsemigroups, Ideals and Quotients</i>	2392
77.5.1	Subsemigroups and Ideals	2392
77.5.2	Quotients	2393

77.6	<i>Extensions</i>	2393
77.7	<i>Elementary Tietze Transformations</i>	2393
77.8	<i>String Operations on Words</i>	2395
78	MONOIDS GIVEN BY REWRITE SYSTEMS . . . . .	2397
78.1	<i>Introduction</i>	2399
78.1.1	Terminology	2399
78.1.2	The Category of Rewrite Monoids	2399
78.1.3	The Construction of a Rewrite Monoid	2399
78.2	<i>Construction of a Rewrite Monoid</i>	2400
78.3	<i>Basic Operations</i>	2405
78.3.1	Accessing Monoid Information	2405
78.3.2	Properties of a Rewrite Monoid	2406
78.3.3	Construction of a Word	2408
78.3.4	Arithmetic with Words	2408
78.4	<i>Homomorphisms</i>	2410
78.4.1	General Remarks	2410
78.4.2	Construction of Homomorphisms	2410
78.5	<i>Set Operations</i>	2410
78.6	<i>Conversion to a Finitely Presented Monoid</i>	2412
78.7	<i>Bibliography</i>	2413



# PART X

## FINITELY-PRESENTED GROUPS

69	ABELIAN GROUPS	2039
70	FINITELY PRESENTED GROUPS	2075
71	FINITELY PRESENTED GROUPS: ADVANCED	2201
72	POLYCYCLIC GROUPS	2247
73	BRAID GROUPS	2287
74	GROUPS DEFINED BY REWRITE SYSTEMS	2337
75	AUTOMATIC GROUPS	2355
76	GROUPS OF STRAIGHT-LINE PROGRAMS	2375
77	FINITELY PRESENTED SEMIGROUPS	2385
78	MONOIDS GIVEN BY REWRITE SYSTEMS	2397



# 69 ABELIAN GROUPS

<b>69.1 Introduction . . . . .</b>	<b>2041</b>		
<b>69.2 Construction of a Finitely Presented Abelian Group and its Elements . . . . .</b>	<b>2041</b>		
69.2.1 <i>The Free Abelian Group . . . . .</i>	<i>2041</i>		
FreeAbelianGroup(n)	2041		
69.2.2 <i>Relations . . . . .</i>	<i>2042</i>		
=	2042		
r[1]	2042		
LHS(r)	2042		
r[2]	2042		
RHS(r)	2042		
Parent(r)	2042		
69.2.3 <i>Specification of a Presentation . . . . .</i>	<i>2043</i>		
AbelianGroup< >	2043		
AbelianGroup([n <sub>1</sub> , . . . , n <sub>r</sub> ])	2044		
69.2.4 <i>Accessing the Defining Generators and Relations . . . . .</i>	<i>2044</i>		
.	2044		
Generators(A)	2044		
NumberOfGenerators(A)	2044		
Ngens(A)	2044		
Parent(u)	2044		
Relations(A)	2044		
RelationMatrix(A)	2044		
<b>69.3 Construction of a Generic Abelian Group . . . . .</b>	<b>2045</b>		
69.3.1 <i>Specification of a Generic Abelian Group . . . . .</i>	<i>2045</i>		
GenericAbelianGroup(U: -)	2045		
69.3.2 <i>Accessing Generators . . . . .</i>	<i>2048</i>		
Universe(A)	2048		
.	2048		
Generators(A)	2048		
UserGenerators(A)	2048		
NumberOfGenerators(A)	2048		
Ngens(A)	2048		
69.3.3 <i>Computing Abelian Group Structure</i>	<i>2048</i>		
AbelianGroup(A: -)	2049		
<b>69.4 Elements . . . . .</b>	<b>2050</b>		
69.4.1 <i>Construction of Elements . . . . .</i>	<i>2050</i>		
!	2050		
!	2050		
!	2050		
!	2050		
Random(A)	2050		
Identity(A)	2050		
Id(A)	2050		
		!	2050
		69.4.2 <i>Representation of an Element . . . . .</i>	<i>2051</i>
		Representation(g)	2051
		ElementToSequence(g)	2051
		Eltseq(g)	2051
		UserRepresentation(g)	2051
		Representation(S, g)	2051
		69.4.3 <i>Arithmetic with Elements . . . . .</i>	<i>2052</i>
		+	2052
		-	2052
		-	2052
		*	2052
		*	2052
		<b>69.5 Construction of Subgroups and Quotient Groups . . . . .</b>	<b>2053</b>
		69.5.1 <i>Construction of Subgroups . . . . .</i>	<i>2053</i>
		sub< >	2053
		sub< >	2054
		69.5.2 <i>Construction of Quotient Groups . . . . .</i>	<i>2055</i>
		quo< >	2055
		/	2055
		<b>69.6 Standard Constructions and Conversions . . . . .</b>	<b>2055</b>
		AbelianGroup(GrpAb, Q)	2055
		AbelianGroup(Q)	2055
		AbelianGroup(G)	2055
		AbelianQuotient(G)	2055
		DirectSum(A, B)	2056
		PCGroup(A)	2056
		PermutationGroup(A)	2056
		FPGroup(A)	2056
		CommutatorSubgroup(G)	2056
		DerivedSubgroup(G)	2056
		CommutatorSubgroup(H, K)	2056
		CommutatorSubgroup(G, H, K)	2056
		Centralizer(G, a)	2056
		Centraliser(G, a)	2056
		Core(G, H)	2056
		Centre(G)	2056
		Center(G)	2056
		<b>69.7 Operations on Elements . . . . .</b>	<b>2057</b>
		69.7.1 <i>Order of an Element . . . . .</i>	<i>2057</i>
		Order(x)	2057
		Order(g: -)	2057
		Order(g, l, u: -)	2058
		Order(g, l, u, n, m: -)	2058
		69.7.2 <i>Discrete Logarithm . . . . .</i>	<i>2058</i>
		Log(g, d: -)	2058
		69.7.3 <i>Equality and Comparison . . . . .</i>	<i>2059</i>

eq	2059	meet	2064
ne	2059	meet:=	2064
IsIdentity(u)	2059	+	2064
IsId(u)	2059	*	2064
<b>69.8 Invariants of an Abelian Group</b>	<b>2060</b>	FrattiniSubgroup(G)	2064
ElementaryAbelianQuotient(G, p)	2060	SylowSubgroup(G, p : -)	2064
FreeAbelianQuotient(G)	2060	Sylow(G, p : -)	2064
Invariants(A)	2060	<b>69.13 Subgroup Chains . . . . .</b>	<b>2065</b>
TorsionFreeRank(A)	2060	CompositionSeries(G)	2065
TorsionInvariants(A)	2060	Agemo(G, i)	2065
PrimaryInvariants(A)	2060	Omega(G, i)	2065
pPrimaryInvariants(A, p)	2060	<b>69.14 General Group Properties . . . . .</b>	<b>2065</b>
<b>69.9 Canonical Decomposition . . . . .</b>	<b>2060</b>	IsCyclic(G)	2065
TorsionFreeSubgroup(A)	2060	IsElementaryAbelian(G)	2065
TorsionSubgroup(A)	2060	IsFree(G)	2065
pPrimaryComponent(A, p)	2060	IsMixed(G)	2065
<b>69.10 Set-Theoretic Operations . . . . .</b>	<b>2061</b>	IspGroup(G)	2065
<i>69.10.1 Functions Relating to Group Order</i>	<i>2061</i>	<i>69.14.1 Properties of Subgroups . . . . .</i>	<i>2066</i>
Order(G)	2061	IsMaximal(G, H)	2066
#	2061	Index(G, H)	2066
FactoredOrder(G)	2061	FactoredIndex(G, H)	2066
Exponent(G)	2061	IsPure(G, H)	2066
IsFinite(G)	2061	IsNeat(G, H)	2066
IsInfinite(G)	2061	<i>69.14.2 Enumeration of Subgroups . . . . .</i>	<i>2066</i>
<i>69.10.2 Membership and Equality . . . . .</i>	<i>2061</i>	MaximalSubgroups(G)	2066
in	2061	Subgroups(G:-)	2066
notin	2061	NumberOfSubgroupsAbelianPGroup (A)	2067
subset	2061	HasComplement(G, U)	2067
notsubset	2061	<b>69.15 Representation Theory . . . . .</b>	<b>2068</b>
subset	2062	CharacterTable(G)	2068
notsubset	2062	<b>69.16 The Hom Functor . . . . .</b>	<b>2068</b>
eq	2062	Hom(G, H)	2068
ne	2062	HomGenerators(G, H)	2068
<i>69.10.3 Set Operations . . . . .</i>	<i>2062</i>	Homomorphisms(G, H)	2068
NumberingMap(G)	2062	<b>69.17 Automorphism Groups . . . . .</b>	<b>2070</b>
RandomProcess(G)	2062	<b>69.18 Cohomology . . . . .</b>	<b>2070</b>
Random(P)	2062	Dual(G)	2070
Random(G)	2063	H2_G_QmodZ(G)	2070
Rep(G)	2063	Res_H2_G_QmodZ(U, H2)	2070
<b>69.11 Coset Spaces . . . . .</b>	<b>2063</b>	<b>69.19 Homomorphisms . . . . .</b>	<b>2070</b>
Transversal(G, H)	2063	hom< >	2070
RightTransversal(G, H)	2063	Homomorphism(A, B, X, Y)	2071
<i>69.11.1 Coercions Between Groups and</i>	<i>2063</i>	iso< >	2071
<i>Subgroups . . . . .</i>	<i>2063</i>	Isomorphism(A, B, X, Y)	2071
!	2063	<b>69.20 Bibliography . . . . .</b>	<b>2073</b>
!	2063		
!	2063		
Morphism(H, G)	2063		
<b>69.12 Subgroup Constructions . . . . .</b>	<b>2064</b>		

# Chapter 69

## ABELIAN GROUPS

### 69.1 Introduction

This Chapter contains a description of the MAGMA machinery provided for computing with abstract abelian groups. Such a group may be described either in terms of defining relations (category `GrpAb`) or by defining the group operations on some finite set (category `GrpAbGen`). In the case of finitely presented groups, the abelian groups may be finite and infinite, the only restriction being that the group be finitely generated.

### 69.2 Construction of a Finitely Presented Abelian Group and its Elements

#### 69.2.1 The Free Abelian Group

<code>FreeAbelianGroup(n)</code>
----------------------------------

Construct the free abelian group  $F$  on  $n$  generators, where  $n$  is a positive integer. The  $i$ -th generator may be referenced by the expression `F.i`,  $i = 1, \dots, n$ . Note that a special form of the assignment statement is provided which enables the user to assign names to the generators of  $F$ . In this form of assignment, the list of generator names is enclosed within angle brackets and appended to the variable name on the *left hand side* of the assignment statement.

#### Example H69E1

---

The statement

```
> F := FreeAbelianGroup(2);
```

creates the free abelian group on two generators. Here the generators may be referenced using the standard names,  $F.1$  and  $F.2$ .

The statement

```
> F<x, y> := FreeAbelianGroup(2);
```

defines  $F$  to be the free abelian group on two generators and assigns the names  $x$  and  $y$  to the generators.

---

### 69.2.2 Relations

$w_1 = w_2$

Given words  $w_1$  and  $w_2$  over the generators of an abelian group  $A$ , create the relation  $w_1 = w_2$ . Note that this relation is not automatically added to the existing set of defining relations  $R$  for  $S$ . It may be added to  $R$ , for example, through use of the `quo`-constructor (see below).

$r[1]$

$\text{LHS}(r)$

Given a relation  $r$  over the generators of  $A$ , return the left hand side of the relation  $r$ . The object returned is a word over the generators of  $A$ .

$r[2]$

$\text{RHS}(r)$

Given a relation  $r$  over the generators of  $A$ , return the right hand side of the relation  $r$ . The object returned is a word over the generators of  $A$ .

$\text{Parent}(r)$

Group over which the relation  $r$  is taken.

#### Example H69E2

---

We may define a group and a set of relations as follows:

```
> F<x, y> := FreeAbelianGroup(2);
> rels := { 2*x = 3*y, 4*x + 4*y = Id(F) } ;
```

To replace one side of a relation, the easiest way is to reassign the relation. So for example, to replace the relation  $2x = 3y$  by  $2x = 4y$ :

```
> r := 2*x = 3*y;
> r := LHS(r) = 4*y;
```

---

### 69.2.3 Specification of a Presentation

An abelian group with non-trivial relations is constructed as a quotient of an existing abelian group, possibly a free abelian group.

AbelianGroup< X   R >
-----------------------

Given a list  $X$  of variables  $x_1, \dots, x_r$ , and a list of relations  $R$  over these generators, let  $F$  be the free abelian group on the generators  $x_1, \dots, x_r$ . Then construct (a) an abelian group  $A$  isomorphic to the quotient of  $F$  by the subgroup of  $F$  defined by  $R$ , and (b) the natural homomorphism  $\phi : F \rightarrow A$ .

Each term of the list  $R$  is either a *word*, a *relation*, a *relation list* or a *subgroup* of  $F$ .

- A *relation* consists of a pair of words, separated by ‘=’.
- A *word*  $w$  is interpreted as a *relator*, that is, it is equivalent to the relation  $w = 0$ . (See above).
- A *relation list* consists of a list of words, where each pair of adjacent words is separated by ‘=’:  $w_1 = w_2 = \dots = w_r$ . This is interpreted as the set of relations  $w_1 = w_r, \dots, w_{r-1} = w_r$ . Note that the relation list construct is only meaningful in the context of the `quo`-constructor.

A *subgroup*  $H$  appearing in the list  $R$  contributes its generators to the relation set for  $A$ , i.e., each generator of  $H$  is interpreted as a relator for  $A$ .

The group  $F$  may be referred to by the special symbol  $\$$  in any word appearing to the right of the ‘|’ symbol in the `quo`-constructor. Also, in the context of the `quo`-constructor, the identity element (empty word) may be represented by the digit 0. The function returns:

- (a) The quotient group  $A$ ;
- (b) The natural homomorphism  $\phi : F \rightarrow A$ .

---

#### Example H69E3

We create the abelian group defined by the presentation  $\langle a, b, c \mid 7a + 4b + c, 8a + 5b + 2c, 9a + 6b + 3c \rangle$ .

```
> F<a, b, c> := FreeAbelianGroup(3);
> A := quo< F | 7*a + 4*b + c, 8*a + 5*b + 2*c, 9*a + 6*b + 3*c >;
> A;
AbelianGroup isomorphic to Z_3 + Z
Defined on 2 generators
Relations:
  3*A.1 = 0
```

---

A simple way of specifying an abelian group is as a product of cyclic groups.

`AbelianGroup([n1, ..., nr])`

Construct the abelian group defined by the sequence  $[n_1, \dots, n_r]$  of non-negative integers as an abelian group. The function returns the direct product of cyclic groups  $C_{n_1} \times C_{n_2} \times \dots \times C_{n_r}$ , where  $C_0$  is interpreted as an infinite cyclic group.

---

#### Example H69E4

We create the abelian group  $Z_2 \times Z_3 \times Z_4 \times Z_5 \times Z_6 \times Z \times Z$ .

```
> G<[x]> := AbelianGroup([2,3,4,5,6,0,0]);
> G;
Abelian Group isomorphic to Z/2 + Z/6 + Z/60 + Z + Z
Defined on 7 generators
Relations:
  2*G.1 = 0
  3*G.2 = 0
  4*G.3 = 0
  5*G.4 = 0
  6*G.5 = 0
```

---

### 69.2.4 Accessing the Defining Generators and Relations

The functions described here provide access to basic information stored for an abelian group  $A$ .

`A . i`

The  $i$ -th defining generator for  $A$ .

`Generators(A)`

A set containing the generators for  $A$ .

`NumberOfGenerators(A)`

`Ngens(A)`

The number of generators for  $A$ .

`Parent(u)`

The parent group  $A$  of the word  $u$ .

`Relations(A)`

A sequence containing the defining relations for  $A$ .

`RelationMatrix(A)`

A matrix where each row corresponds to one of the defining relations of  $A$ .

### 69.3 Construction of a Generic Abelian Group

The term generic abelian group refers to the situation in which a group  $A$  is described by giving a set  $X$  together with functions acting on  $X$  that perform the fundamental group operations for  $A$ . Specifically, the user must provide functions which compute the product, the inverse and the identity. For efficiency, the user may also optionally specify the order and a generating set for the group. This is made explicit below.

Going from such a definition of an abelian group  $A$  to a presentation for  $A$  will often be extremely expensive and so a small number of operations are implemented so as to not require this information. The two key operations are computing the order of an element and computing the discrete logarithm of an element to a given base. For many abelian group operations, knowledge of a presentation is required and if such an operation is invoked, the structure of  $A$  (and hence a presentation) will be automatically constructed.

There are two possible ways of computing the structure of the group. If the order of the group is known (or can be computed) then it is relatively easy to construct each of the  $p$ -Sylow subgroups. If the order is not available, the structure is computed from a set of generators supplied by the user.

#### 69.3.1 Specification of a Generic Abelian Group

GenericAbelianGroup( $U$ : <i>parameters</i> )		
IdIntrinsic	MONSTG	Default :
AddIntrinsic	MONSTG	Default :
InverseIntrinsic	MONSTG	Default :
UseRepresentation	BOOL	Default : false
Order	RNGINTELT	Default :
UserGenerators	SETENUM	Default :
ProperSubset	BOOL	Default : false
RandomIntrinsic	MONSTG	Default :
ComputeStructure	BOOL	Default : false

Construct the generic abelian group  $A$  over the domain  $U$ . The domain  $U$  can be an aggregate (set, indexed set or sequence) of elements or it can be any structure, for example, an elliptic curve, a jacobian, a magma of quadratic forms.

If the parameters `IdIntrinsic`, `AddIntrinsic` and `InverseIntrinsic` are not set, then the identity, the composition and the inverse function of  $A$  are taken to be the identity, the composition and the inverse function of  $U$  or of `Universe( $U$ )` if  $U$  is an aggregate. If the parameters `IdIntrinsic`, `AddIntrinsic` and `InverseIntrinsic` are set, they define the identity, the composition and the inverse function of  $A$ .

The parameter `IdIntrinsic` must be a function name whose sole argument is  $U$  or `Universe( $U$ )` if  $U$  is an aggregate. `AddIntrinsic` must be a function name whose only two arguments are elements of  $U$  or of `Universe( $U$ )` if  $U$  is

an aggregate. `InverseIntrinsic` must be a function name whose only two arguments are elements of  $U$  or of  $\text{Universe}(U)$  if  $U$  is an aggregate. That is, it is required that `InverseIntrinsic` be a binary operation (see the example below where `InverseIntrinsic := "/"` is indeed binary). Defining any of the three above parameters implies that the other two must be defined as well.

Setting the parameter `UseRepresentation` to true implies that all elements of  $A$  will be internally represented as linear combinations of the generators of  $A$  obtained while computing the structure of  $A$ . This can be a costly procedure, since such a representation is akin to solving the discrete logarithm problem. The advantage of such a representation is that arithmetic for elements of  $A$  as well as computing the order of elements of  $A$  are then essentially trivial operations.

The parameter `Order` can be set to the order of the group, if known. Knowledge of the order may save a considerable amount of time for operations such as computing a Sylow subgroup, determining the group structure or solve a discrete logarithm problem. More importantly, if  $A$  is a proper subset of  $U$ , or of  $\text{Universe}(U)$  if  $U$  is an aggregate, then one of `Order` or `UserGenerators` must be set.

One can set `UserGenerators` to some set of elements of  $U$ , or of  $\text{Universe}(U)$  if  $U$  is an aggregate, which generate the group  $A$ . This can be useful when  $A$  is a subset of  $U$  ( $\text{Universe}(U)$ ), or more generally when the computation of the group structure of  $A$  is made from a set of generators. Finally, setting `UserGenerators` may be an alternative to setting `RandomIntrinsic`.

The parameter `ProperSubset` must be set when  $A$  is a proper subset of  $U$  ( $\text{Universe}(U)$ ).

The parameter `RandomIntrinsic` indicates the random function to use. If it is not set, the random function (if it is required) is taken to be the random function applying to  $U$  ( $\text{Universe}(U)$ ).

The parameter `RandomIntrinsic` must be the name of a function taking as its sole argument  $U$  ( $\text{Universe}(U)$ ) and returning an element of  $U$  ( $\text{Universe}(U)$ ) which is also an element of the group  $A$ , which is important when  $A$  is a proper subset of  $U$  ( $\text{Universe}(U)$ ). Therefore, if  $A$  is a proper subset of  $U$  ( $\text{Universe}(U)$ ), then `RandomIntrinsic` must be set, unless the parameter `UserGenerators` is set.

The parameter `Structure` indicates that the group structure should be determined at the time of creation. If this parameter is set then the user may also want to set the following parameters:

<code>UseUserGenerators</code>	BOOL	<i>Default : false</i>
<code>PollardRhoRParam</code>	RNGINT	<i>Default : 20</i>
<code>PollardRhoTParam</code>	RNGINT	<i>Default : 8</i>
<code>PollardRhoVParam</code>	RNGINT	<i>Default : 3</i>

Their meaning is detailed in Section 69.3.3.

---

### Example H69E5

In our first example we create the subset  $U$  of  $\mathbf{Z}/34384\mathbf{Z}$  corresponding to its unit group as a

generic abelian group  $G$ . Note that  $U$  is an indexed set.

```
> m := 34384;
> Zm := Integers(m);
> U := {@ x : x in Zm | GCD(x, m) eq 1 @} ;
> G := GenericAbelianGroup(U :
>   IdIntrinsic := "Id",
>   AddIntrinsic := "*",
>   InverseIntrinsic := "/");
> G;
Generic Abelian Group over
Residue class ring of integers modulo 34384
```

In our next example we construct unique representatives for the classes of binary quadratic forms corresponding to elements of a subgroup of the class group of the imaginary quadratic field of discriminant  $-4000004$ .

```
> n := 6;
> Dk := -4*(10^n + 1);
> Q := QuadraticForms(Dk);
>
> p := 2;
> S := { };
> while #S lt 10 do
>   if KroneckerSymbol(Dk,p) eq 1 then
>     I := Reduction(PrimeForm(Q,p));
>     Include(~S, I);
>   end if;
>   p := NextPrime(p);
> end while;
>
> QF := GenericAbelianGroup(Q : UserGenerators := S,
>   ComputeStructure := true,
>   UseUserGenerators := true);
> QF;
Generic Abelian Group over
Binary quadratic forms of discriminant -4000004
Abelian Group isomorphic to Z/2 + Z/516
Defined on 2 generators
Relations:
  2*$.1 = 0
  516*$.2 = 0
```

So the structure of the subgroup is  $Z_2 \times Z_{516}$

### 69.3.2 Accessing Generators

These functions give access to generating sets for a generic group  $A$ . If a generating set is requested and none has been supplied by the user then this will require the determination of the group structure which could be very expensive. Note the distinction between `Generators` and `UserGenerators`. From now on, unless otherwise specified, whenever mention is made of the generators of  $A$ , we refer to the generators as given by the `Generators` function.

`Universe(A)`

The universe over which the generic abelian group  $A$  is defined.

`A . i`

The  $i$ -th generator for the generic abelian group  $A$ .

`Generators(A)`

A sequence containing a generating set for the generic abelian group  $A$  as elements of  $A$ . The set of generators returned for  $A$  is a reduced set of generators as constructed during the structure computation. Therefore, if the group structure of  $A$  has been computed from a user-supplied set of generators, it is unlikely that `Generators(A)` and `UserGenerators(A)` will be the same.

`UserGenerators(A)`

A sequence containing the user-supplied generators for the generic abelian group  $A$  as elements of  $A$ .

`NumberOfGenerators(A)`

`Ngens(A)`

The number of generators for the generic abelian group  $A$ .

### 69.3.3 Computing Abelian Group Structure

If the order of a generic abelian group  $A$  can be obtained then the structure of  $A$  is found by constructing each  $p$ -Sylow subgroup of  $A$ . The  $p$ -Sylow subgroups are built from random elements of the underlying set  $X$  of  $A$ . Recall that  $U$  (`Universe(U)`) is the domain of  $A$ . Random elements are obtained using either a random function attached to  $X$  or using a user-supplied function (the `RandomIntrinsic` parameter), or using user-supplied generators (the `UserGenerators` parameter). It is therefore vital that user-supplied generators truly generate the group one wishes to construct. A drawback of this method of obtaining the structure of  $A$  is the memory needed to store all the elements of a specific  $p$ -Sylow subgroup while under construction. This algorithm is mostly based on work by Michael Stoll.

The second approach computes the group structure from a set of generators as supplied by the user, removing the need to compute the order of  $A$ . This can be particularly useful when computing this order is expensive. Note that computing the structure of a group from a set of generators is akin to solving a number of the discrete logarithm problems.

This second algorithm uses a variant of the Pollard–Rho algorithm and is due to Edlyn Teske [Tes98].

If  $A$  is a subgroup of a generic abelian group,  $G$  say, then the structure of  $G$  is computed first. The rationale is that once the structure of  $G$  is known, computing the structure of  $A$  is almost always cheap.

AbelianGroup(A: <i>parameters</i> )		
-------------------------------------	--	--

UseUserGenerators	BOOL	<i>Default</i> : false
PollardRhoRParam	RNGINT	<i>Default</i> : 20
PollardRhoTParam	RNGINT	<i>Default</i> : 8
PollardRhoVParam	RNGINT	<i>Default</i> : 3

Compute the group structure of the generic abelian group  $A$ , which may be a subgroup as created by the subgroup constructor or the Sylow function. The two values returned are the abstract abelian group and the invertible map from the latter into  $A$ .

If `UseUserGenerators` is false, then the group structure computation is made via the construction of each  $p$ -Sylow subgroup, using the factorization of the order of  $A$ .

If `UseUserGenerators` is set to true, the group structure computation uses the user-supplied set of generators for  $A$ . In this case, the additional parameters `PollardRhoRParam`, `PollardRhoTParam`, and `PollardRhoVParam` can be supplied. Their values will be passed to the Pollard–Rho algorithm used in the group structure computation: `PollardRhoRParam` sets the size of the  $r$ -adding walks, `PollardRhoTParam` sets the size of the internal storage of elements, and `PollardRhoVParam` is used for an efficient finding of the periodic segment. It is conjectured that the default values as given above are ‘best’ (see [Tes98b]), therefore there should be no need to set these parameters in general.

---

### Example H69E6

The following statement computes the structure of the unit group of  $\mathbf{Z}_{34384}$ .

```
> G := AbelianGroup(G); G;
Generic Abelian Group over
Residue class ring of integers modulo 34384
Abelian Group isomorphic to Z/2 + Z/2 + Z/6 + Z/612
Defined on 4 generators
Relations:
  2*G.1 = 0
  2*G.2 = 0
  6*G.3 = 0
  612*G.4 = 0
```

---

## 69.4 Elements

### 69.4.1 Construction of Elements

Unless otherwise stated, the operations in this section apply to fp-abelian groups and generic abelian groups.

**A ! [a<sub>1</sub>, . . . , a<sub>n</sub>]**

Given an abelian group  $A$  with generators  $e_1, \dots, e_r$  and a sequence  $Q = [a_1, \dots, a_r]$  of integers, construct the element  $a_1e_1 + \dots + a_re_r$  of  $A$ .

**A ! e**

Given a generic abelian group  $A$  and an element  $e$  of the domain over which it is defined, return  $e$  as an element of  $A$ . If  $A$  is a proper subset of its underlying domain, then  $e$  must be a linear combination of the generators (which may be user-supplied) of  $A$ .

**A ! g**

Given a generic abelian group  $A$  and an element  $g$  of the underlying set  $X$  of  $A$ , return  $g$  as an element of  $A$ .

**A ! n**

Given an abelian group  $A$  with exactly one generator  $x$ , construct the element  $nx$ .

**Random(A)**

Given either a finite fp-abelian group or a generic abelian group  $A$ , return a random element of  $A$ .

**Identity(A)**

**Id(A)**

**A ! 0**

Construct the identity element (empty word) for the abelian group  $A$ .

Let  $A$  be a generic abelian group defined in the universe  $U$  of  $A$ . If  $g$  is an element of  $A$ , then  $U!g$  is an element of  $U$ .

### 69.4.2 Representation of an Element

An element  $g$  of an abelian group  $A$  can be represented as a linear combination with respect to a given generating sequence. The coefficients appearing in this linear combination provide an alternative representation for  $g$ . If  $A$  is a fp-group, the generating set will be the one on which the group was defined. In the case of a generic group, the generating set can either be that obtained when constructing a presentation for  $A$  or a user-supplied generating set.

**Representation(g)**

**ElementToSequence(g)**

**Eltseq(g)**

Let  $A$  be an abelian group with generating set  $e_1, \dots, e_n$  and suppose  $g$  is an element of  $A$ , where  $g = a_1e_1 + \dots + a_n e_n$ . These functions return the sequence  $Q$  of  $n$  integers defined by  $Q[i] = a_i$ , for  $i = 1, \dots, n$ . Moreover, each  $a_i$ ,  $i = 1, \dots, n$ , is the integer residue modulus the order of the  $i$ th generator.

**UserRepresentation(g)**

Let  $A$  be a generic abelian group with a user-supplied set of generators  $u_1, \dots, u_n$  and suppose  $g$  is an element of  $A$ , where  $g = a_1u_1 + \dots + a_n u_n$ . This function returns the sequence  $Q$  of  $n$  integers defined by  $Q[i] = a_i$ , for  $i = 1, \dots, n$ . Moreover, each  $a_i$ ,  $i = 1, \dots, n$ , is the integer residue modulus the order of the  $i$ th generator.

**Representation(S, g)**

Let  $A$  be a generic abelian group and let  $S = [s_1, \dots, s_m]$  be any sequence of elements of  $A$ . Assume  $g$  is an element of  $A$  such that  $bg = a_1s_1 + \dots + a_m s_m$ . This function returns as its first value the sequence  $Q$  of  $m$  integers defined by  $Q[i] = a_i$ , for  $i = 1, \dots, m$ . The second value returned is the coefficient  $b$  of  $g$ . Note that  $b$  might not be 1.

#### Example H69E7

---

We use the quadratic forms example considered above to illustrate these functions.

```
> Generators(QF);
[ <2,2,500001>, <206,-102,4867> ]
> g := QF ! [5, 6];
> g;
<837,-766,1370>
> Representation(g);
[ 1, 6 ]
>
> g := Random(QF);
> Representation(g);
[ 1, 270 ]
>
```

```

> UserRepresentation(g);
[ 377, 0, 515, 0, 0, 0, 0, 0, 0, 0 ]
>
> S := [];
> for i in [1..3] do
>   d := Random(QF);
>   Include(~S, d);
> end for;
> seq, coeff := Representation(S, g);
> seq; coeff;
[ -170, -3, 0 ]
1

```

---

### 69.4.3 Arithmetic with Elements

If the generic abelian group  $A$  has been constructed with the flag `UserRepresentation` set true, then arithmetic with elements of  $A$  is trivial.

$u + v$

Given elements  $u$  and  $v$  belonging to the same abelian group  $A$ , return the sum of  $u$  and  $v$ .

$-u$

The inverse of element  $u$ .

$u - v$

Given elements  $u$  and  $v$  belonging to the same abelian group  $A$ , return the sum of  $u$  and the inverse of  $v$ .

$m * u$

$u * m$

Given an integer  $m$ , return the element  $w + w + \cdots + w$  ( $|m|$  summands), where  $w = u$ , if  $m$  is positive and  $w = -u$  if  $m$  is negative.

## 69.5 Construction of Subgroups and Quotient Groups

The operations in this section apply to both, free abelian groups and arbitrary abelian groups.

### 69.5.1 Construction of Subgroups

sub< A | L >

Construct the subgroup  $B$  of the abelian group  $A$  generated by the elements specified by the terms of the *generator list*  $L$ . A term  $L[i]$  of the generator list may consist of any of the following objects:

- (a) An element liftable to  $A$ ;
- (b) A sequence of integers representing an element of  $A$ ;
- (c) A subgroup of  $A$ ;
- (d) A set or sequence of type (a), (b), or (c).

The collection of words and groups specified by the list must all belong to the group  $A$  and the group  $B$  will be constructed as a subgroup of  $A$ .

#### Example H69E8

---

We create a subgroup of the group  $A = Z_2 + Z_3 + Z_4 + Z_5 + Z_6 + Z + Z$ .

```
> A<[x]> := AbelianGroup([2,3,4,5,6,0,0]);
> A;
Abelian Group isomorphic to Z/2 + Z/6 + Z/60 + Z + Z
Defined on 7 generators
Relations:
  2*x[1] = 0
  3*x[2] = 0
  4*x[3] = 0
  5*x[4] = 0
  6*x[5] = 0
> B<[y]> := sub< A | x[1], x[3], x[5], x[7] >;
> B;
Abelian Group isomorphic to Z/2 + Z/2 + Z/12 + Z
Defined on 4 generators in supergroup A:
  y[1] = 2*x[3] + 3*x[5]
  y[2] = x[1]
  y[3] = 3*x[3] + x[5]
  y[4] = x[7]
Relations:
  2*y[1] = 0
  2*y[2] = 0
  12*y[3] = 0
```

---

In the case of subgroups of generic groups, a number of parameters are provided.

sub< A   L: <i>parameters</i> >
---------------------------------

Construct the subgroup of the generic abelian group  $A$  generated by the elements specified by the terms of the *generator list*  $L$ . A term  $L[i]$  of the generator list may consist of any of the following objects:

- (a) An element liftable into  $A$ ;
- (b) A sequence of integers representing an element of  $A$ ;
- (c) A set or sequence whose elements may be of either of the above types.

An element liftable into  $A$  may be an element of  $A$  itself, or it may be an element of  $U$  ( $\text{Universe}(U)$ ),  $U$  being as usual the domain over which  $A$  is defined. For consistency with the construction, the values of the following parameters may also be passed to the subgroup constructor:

Order	RNGINT	Default :
RandomIntrinsic	MONSTG	Default :
ComputeStructure	BOOL	Default : false
UseUserGenerators	BOOL	Default : false
PollardRhoRParam	RNGINT	Default : 20
PollardRhoTParam	RNGINT	Default : 8
PollardRhoVParam	RNGINT	Default : 3

In particular, it is possible to construct a subgroup by giving its order and a random function generating elements of the *subgroup*. In this case, the list  $L$  would be empty since calculation of the subgroup structure would result in the construction of the  $p$ -Sylow subgroups from random elements of the subgroup. Further, when the structure of  $A$  is already known and if the subgroup is defined in terms of a set of generators in  $L$  then the subgroup structure is computed at the time of creation.

### Example H69E9

---

We create a subgroup of the quadratic forms group considered above.

```
> S := [];
> for j in [1..2] do
>   P := Random(QF);
>   Include(~S, P);
> end for;
> S;
[ <45,26,22226>, <937,-930,1298> ]
> QF1 := sub< QF | S>;
> QF1;
Generic Abelian Group over
Binary quadratic forms of discriminant -4000004
Abelian Group isomorphic to Z/2 + Z/258
Defined on 2 generators in supergroup A:
```

```

QF1.1 = QF.1
QF1.2 = 2*QF.2
Relations:
2*QF1.1 = 0
258*QF1.2 = 0
>

```

---

### 69.5.2 Construction of Quotient Groups

`quo< F | R >`

Given an abelian group  $F$ , and a set of relations  $R$  in the generators of  $F$ , construct (a) an abelian group  $A$  isomorphic to the quotient of  $F$  by the subgroup of  $F$  defined by  $R$ , and (b) the natural homomorphism  $\phi : F \rightarrow A$ .

The expression defining  $F$  may be either simply the name of a previously constructed group, or an expression defining an abelian group. The possibilities for the relation list  $R$  are the same as for the `AbelianGroup` construction.

The function returns:

- (a) The quotient group  $A$ ;
- (b) The natural homomorphism  $\phi : F \rightarrow A$ .

`A / B`

Given a subgroup  $B$  of the abelian group  $A$ , construct the quotient of  $A$  by  $B$ .

### 69.6 Standard Constructions and Conversions

`AbelianGroup(GrpAb, Q)`

`AbelianGroup(Q)`

Let  $Q = [a_1, \dots, a_r]$  be a sequence of non-negative integers. This function creates the abelian group  $\mathbf{Z}_1 + \dots + \mathbf{Z}_r$ , where  $Z_i$  is the cyclic group of order  $|a_i|$  if  $a_i \neq 0$  or the infinite cyclic group  $\mathbf{Z}$  otherwise,  $i = 1, \dots, r$ .

`AbelianGroup(G)`

Given an abelian permutation, matrix or polycyclic group  $G$ , represent it as an abelian group  $A$ . The function also returns the isomorphism  $\phi : G \rightarrow A$  as its second value.

`AbelianQuotient(G)`

Given a finitely presented, permutation, matrix or polycyclic group  $G$ , return the maximal abelian quotient  $A$  of  $G$ . The function returns the natural homomorphism  $\phi : G \rightarrow A$  as its second value.

`DirectSum(A, B)`

The direct sum of abelian groups  $A$  and  $B$ .

`PCGroup(A)`

A pc-group representation  $G$  of  $A$ . The isomorphism  $\phi : A \rightarrow G$  is also returned.

`PermutationGroup(A)`

A permutation group representation of  $A$ . The particular group  $G$  is generated by disjoint cycles whose lengths are the abelian invariants of  $A$ . The isomorphism  $\phi : G \rightarrow A$  is also returned.

`FPGroup(A)`

A fp-group representation of  $A$ . The particular group  $G$  is generated by commuting generators whose orders are the abelian invariants of  $A$ . The isomorphism  $\phi : G \rightarrow A$  is also returned.

`CommutatorSubgroup(G)`

`DerivedSubgroup(G)`

The derived subgroup of  $G$ , that is the trivial group, since  $G$  is abelian.

`CommutatorSubgroup(H, K)`

`CommutatorSubgroup(G, H, K)`

The commutator subgroup of groups  $H$  and  $K$  in their common overgroup  $G$ .

`Centralizer(G, a)`

`Centraliser(G, a)`

The centraliser of  $a$  in  $G$ .

`Core(G, H)`

The maximal normal subgroup of  $G$  that is contained in the subgroup  $H$  of  $G$ . Since  $G$  is abelian, this is  $H$  itself.

`Centre(G)`

`Center(G)`

The center of  $G$ , ie.  $G$  itself.

## 69.7 Operations on Elements

### 69.7.1 Order of an Element

Order( $x$ )
--------------

Order of the element  $x$  belonging to an fp-abelian group. If the element has infinite order, the value zero is returned.

---

#### Example H69E10

We compute the orders of some elements in the group  $Z_2 \times Z_3 \times Z_4 \times Z_5 \times Z$ .

```
> G<[x]> := AbelianGroup([2,3,4,5,0]);
> Order( x[1] + 2*x[2] + 3*x[4] );
30
> Order( x[1] + x[5] );
0
```

---

It is possible to determine the order of an element of a generic group with first calculating the structure of the group. The order function involves the use of one of several algorithms:

- an improved baby-step giant-step algorithm, due to J. Buchmann, M.J. Jacobson, E. Teske [BJT97],
- the Pollard–Rho method based algorithm, described above [Tes98].

When computing the order of an element whose lower and upper bounds are known, or where lower and upper bounds for the group order are known, the following two algorithms have been shown to be significantly faster than the two algorithms mentioned above.

- the standard baby-step giant-step Shanks algorithm,
- another variant of the Pollard–Rho method which is due to P. Gaudry and R. Harley [GH00].

To avoid confusion we will distinguish the algorithms due to Teske *et al* and name them the T baby-step giant-step algorithm and the T Pollard–Rho algorithm respectively. The Pollard–Rho algorithm has smaller space requirements than the baby-step giant-step algorithm, so it is recommended for use in very large groups.

In all cases, if the group order is known beforehand, the element order is computed using this knowledge. This is a trivial operation.

Order( $g$ : <i>parameters</i> )
----------------------------------

ComputeGroupOrder	BOOL	<i>Default : true</i>
BSGSLowerBound	RNGINTELT	<i>Default : 0</i>
BSGSStepWidth	RNGINTELT	<i>Default : 0</i>

Assume that  $g$  is an element of the generic abelian group  $A$ . This function returns the order of the element  $g$ . Note that if `UseRepresentation` is set to true for  $A$ , then this is a trivial operation.

If the parameter `ComputeGroupOrder` is true, the order of  $A$  is computed first (unless it is already known). The order of  $g$  is then computed using this knowledge; this last computation is trivial.

If `ComputeGroupOrder` is false, the order of  $g$  is computed using the T baby-step giant-step algorithm.

`BSGSLowerBound` and `BSGSStepWidth` are parameters which can be passed to the baby-step giant-step algorithm.

`BSGSLowerBound` sets a lowerbound on the order of the element  $g$ , and `BSGSStepWidth` sets the step-width in the algorithm.

<code>Order(g, l, u: parameters)</code>
---

<code>Alg</code>	MONSTG	<i>Default : "Shanks"</i>
<code>UseInversion</code>	BOOLELT	<i>Default : false</i>

Assume that  $g$  is an element of the generic abelian group  $A$  such that the order of  $g$  or the order of  $A$  is bounded by  $u$  and  $l$ . This function returns the order of the element  $g$ .

If the parameter `Alg` is set to "Shanks" then the generic Shanks algorithm is used, and when `Alg` is set to "PollardRho", the Gaudry–Harley Pollard–Rho variant is used. Setting `UseInversion` halves the search space. To be effective element inversion must be fast.

<code>Order(g, l, u, n, m: parameters)</code>
---

<code>Alg</code>	MONSTG	<i>Default : "Shanks"</i>
<code>UseInversion</code>	BOOLELT	<i>Default : false</i>

Assume that  $g$  is an element of the generic abelian group  $A$  such that the order of  $g$  or the order of  $A$  is bounded by  $u$  and  $l$ . Assume also that  $\text{Order}(g) \equiv n \pmod{m}$  or that  $\#A \equiv \pmod{m}$ . This function returns the order of the element  $g$ . The two parameters `Alg` and `UseInversion` have the same meaning as for the previous `Order` function.

## 69.7.2 Discrete Logarithm

<code>Log(g, d: parameters)</code>
------------------------------------

<code>ComputeGroupOrder</code>	BOOL	<i>Default : true</i>
<code>AllInPohligHellmanLoop</code>	MONSTG	<i>Default : "BSGS"</i>
<code>BSGSStepWidth</code>	RNGINTELT	<i>Default : 0</i>
<code>PollardRhoRParam</code>	RNGINT	<i>Default : 20</i>
<code>PollardRhoTParam</code>	RNGINT	<i>Default : 8</i>
<code>PollardRhoVParam</code>	RNGINT	<i>Default : 3</i>

Assume that  $g$  and  $d$  are elements of the generic abelian group  $A$ . This function returns the discrete logarithm of  $d$  to the base  $g$ . If `ComputeGroupOrder` is true, then the group order is computed first (if not already known) and from this the order of the base  $g$  is computed. The discrete logarithm problem is then solved using a Pohlig–Hellman loop calling either the T baby-step giant-step algorithm (`AlInPohligHellmanLoop := "BSGS"`) or the T Pollard–Rho algorithm (`AlInPohligHellmanLoop := "PollardRho"`).

The parameter `BSGSStepWidth` has the same meaning as for the `Order` function. Parameters `PollardRhoRParam`, `PollardRhoTParam`, and `PollardRhoVParam` have the same meaning as they do for the determination of structure (function `AbelianGroup`). If `ComputeGroupOrder` is false then the discrete logarithm problem is solved using the T baby-step giant-step algorithm. Here again the parameter `BSGSStepWidth` applies.

---

### Example H69E11

It is assumed that the structure of the groups  $QF$  has already been computed. We illustrate the computation of the discrete logarithm relative to a given base.

```
> n := 38716;
> Ip := Reduction(PrimeForm(Q,11));
> g := GA_qf!Ip;
> d := n * g;
>
> l1 := Log(g, d : BSGSStepWidth := Floor((-Dk)^(1/4)/2));
> l2 := Log(g, d : AlInPohligHellmanLoop := "PollardRho");
> l3 := Log(g, d : ComputeGroupOrder := false);
> l4 := Log(g, d: ComputeGroupOrder := false,
>           BSGSStepWidth := Floor((-Dk)^(1/4)/2));
> assert l1 eq l2 and l2 eq l3 and l3 eq l4;
> assert IsDivisibleBy(n - l1, Order(g));
```

---

### 69.7.3 Equality and Comparison

`u eq v`

Returns `true` if the elements  $u$  and  $v$  are identical (as elements of the appropriate free abelian group), `false` otherwise.

`u ne v`

Returns `true` if the elements  $u$  and  $v$  are not identical (as elements of the appropriate free abelian group), `false` otherwise.

`IsIdentity(u)`

`IsId(u)`

Returns `true` if the element  $u$ , belonging to the abelian group  $A$ , is the identity element (zero), `false` otherwise.

## 69.8 Invariants of an Abelian Group

`ElementaryAbelianQuotient(G, p)`

The maximal  $p$ -elementary abelian quotient of the group  $G$  as `GrpAb`. The natural epimorphism is returned as second value.

`FreeAbelianQuotient(G)`

The maximal free abelian quotient of the group  $G$  as `GrpAb`. The natural epimorphism is returned as second value.

`Invariants(A)`

The invariants of the abelian group  $G$ . Each infinite cyclic factor is represented by zero.

`TorsionFreeRank(A)`

The torsion-free rank of the abelian group  $G$ .

`TorsionInvariants(A)`

The torsion invariants of the abelian group  $G$ .

`PrimaryInvariants(A)`

The primary invariants of the abelian group  $G$ .

`pPrimaryInvariants(A, p)`

The  $p$ -primary invariants of the abelian group  $G$ .

## 69.9 Canonical Decomposition

`TorsionFreeSubgroup(A)`

The torsion-free subgroup of the abelian group  $G$ .

`TorsionSubgroup(A)`

The torsion subgroup of the abelian group  $G$ .

`pPrimaryComponent(A, p)`

The  $p$ -primary component of the abelian group  $G$ .

## 69.10 Set-Theoretic Operations

### 69.10.1 Functions Relating to Group Order

`Order(G)`

`#G`

The order of the group  $G$ , returned as an ordinary integer. If  $G$  is an infinite group, the value zero is returned. Note that if  $G$  is a generic group then determining the order will require the structure of  $G$  to be determined.

`FactoredOrder(G)`

The factored order of the group  $G$ , returned as a sequence of prime-exponent pairs. If  $G$  is an infinite group, the empty sequence is returned. Note that if  $G$  is a generic group then determining the order will require the structure of  $G$  to be determined.

`Exponent(G)`

The exponent of the group  $G$ . If the group is infinite, the value zero is returned. Note that if  $G$  is a generic group then determining the exponent will require the structure of  $G$  to be determined.

`IsFinite(G)`

Return `true` if the group  $G$  is finite.

`IsInfinite(G)`

Return `true` if  $G$ , `false` otherwise.

### 69.10.2 Membership and Equality

`g in G`

Given an element  $g$  and a group  $G$ , return `true` if  $g$  is an element of  $G$ , `false` otherwise.

`g notin G`

Given an element  $g$  and a group  $G$ , return `true` if  $g$  is not an element of  $G$ , `false` otherwise.

`S subset G`

Given a group  $G$  and a set  $S$  of elements belonging to a group  $H$ , where  $G$  and  $H$  have some covering group, return `true` if  $S$  is a subset of  $G$ , `false` otherwise.

`S notsubset G`

Given a group  $G$  and a set  $S$  of elements belonging to a group  $H$ , where  $G$  and  $H$  have some covering group, return `true` if  $S$  is not a subset of  $G$ , `false` otherwise.

**H subset G**

Given groups  $G$  and  $H$ , subgroups of some common overgroup, return **true** if  $H$  is a subgroup of  $G$ , and **false** otherwise.

**H notsubset G**

Given groups  $G$  and  $H$ , subgroups of some common overgroup, return **true** if  $H$  is not a subgroup of  $G$ , and **false** otherwise.

**G eq H**

Given groups  $G$  and  $H$ , subgroups of some common overgroup, return **true** if  $G$  and  $H$  are identical, and **false** otherwise.

**G ne H**

Given groups  $G$  and  $H$ , subgroups of some common overgroup, return **true** if  $G$  and  $H$  are distinct groups, and **false** otherwise.

### 69.10.3 Set Operations

**NumberingMap(G)**

A bijective mapping from the finite group  $G$  onto the set of integers  $\{1 \dots |G|\}$ . The actual mapping depends upon choice of standard generators for  $G$ .

**RandomProcess(G)**

<b>Slots</b>	RNGINTELT	<i>Default : 10</i>
<b>Scramble</b>	RNGINTELT	<i>Default : 100</i>

Create a process to generate randomly chosen elements from the finite group  $G$ . The process is based on the product-replacement algorithm of [CLGM<sup>+</sup>95], modified by the use of an accumulator. At all times,  $N$  elements are stored where  $N$  is the maximum of the specified value for **Slots** and  $\text{Ngens}(G) + 1$ . Initially, these are just the generators of  $G$ . As well, one extra group element is stored, the accumulator. Initially, this is the identity. Random elements are now produced by successive calls to **Random(P)**, where  $P$  is the process created by this function. Each such call chooses one of the elements in the slots and adds it into the accumulator. The element in that slot is replaced by the sum of it and another randomly chosen slot. The random value returned is the new accumulator value. Setting **Scramble := m** causes  $m$  such sum-replacement operations to be performed before the process is returned. Note that this algorithm cannot produce well-distributed random elements of an infinite group.

**Random(P)**

Given a random element process  $P$  created by the function **RandomProcess(G)** for the finite abelian group  $G$ , return the next random element of  $G$  defined by the process.

Random( $G$ )
---------------

An element chosen at random from the finite group  $G$ .

Rep( $G$ )
------------

A representative element of  $G$ .

## 69.11 Coset Spaces

Transversal( $G, H$ )
-----------------------

RightTransversal( $G, H$ )
----------------------------

Given a group  $G$  and a subgroup  $H$  of  $G$ , this function returns:

- (a) An indexed set of elements  $T$  of  $G$  forming a right transversal for  $G$  over  $H$ ; and,
- (b) The corresponding transversal mapping  $\phi : G \rightarrow T$ . If  $T = \{t_1, \dots, t_r\}$  and  $g$  in  $G$ ,  $\phi$  is defined by  $\phi(g) = t_i$ , where  $g \in Ht_i$ .

### 69.11.1 Coercions Between Groups and Subgroups

$G ! g$
---------

Given an element  $g$  belonging to the subgroup  $H$  of the group  $G$ , rewrite  $g$  as an element of  $G$ .

$H ! g$
---------

Given an element  $g$  belonging to the group  $G$ , and given a subgroup  $H$  of  $G$  containing  $g$ , rewrite  $g$  as an element of  $H$ .

$K ! g$
---------

Given an element  $g$  belonging to the group  $H$ , and a group  $K$ , such that  $H$  and  $K$  are subgroups of  $G$ , and both  $H$  and  $K$  contain  $g$ , rewrite  $g$  as an element of  $K$ .

Morphism( $H, G$ )
--------------------

The integer matrix defining the inclusion monomorphism from the subgroup  $H$  of  $G$  into  $G$ .

## 69.12 Subgroup Constructions

Although, in the case of an abelian group, many of the standard subgroup constructors are trivial, they are all implemented for the sake of uniformity. Here we document only those which are meaningful in the context of abelian groups.

**H meet K**

Given subgroups  $H$  and  $K$  of some group  $G$ , construct their intersection.

**H meet:= K**

Replace  $H$  with the intersection of groups  $H$  and  $K$ .

**H + K**

Given subgroups  $H$  and  $K$  of some group  $G$ , construct the smallest subgroup containing both.

**n \* G**

For an integer  $n$  and some abelian group  $G$ , construct the subgroup  $nG$ . The second return value is the map  $G \rightarrow G$  sending  $g \rightarrow ng$ .

**FrattiniSubgroup(G)**

The Frattini subgroup of the finite abelian group  $G$ .

**SylowSubgroup(G, p : parameters)**

**Sylow(G, p : parameters)**

Structure

BOOL

Default : false

The Sylow  $p$ -subgroup for the group  $G$ . If  $G$  is a generic group and the parameter **Structure** is true, or if the group structure of  $A$  is known, then the group structure of the Sylow subgroup is computed.

### Example H69E12

---

In the following example, we construct the Sylow 2-subgroup of  $G = Z_{34384}$ .

```
> m := 34384;
> Zm := Integers(m);
> U := {@ x : x in Zm | GCD(x, m) eq 1 @};
> G := GenericAbelianGroup(U : IdIntrinsic := "Id",
>   AddIntrinsic := "*", InverseIntrinsic := "/");
> _ := AbelianGroup(G);
> Factorization(#G);
> Sylow(G, 2);
2-Sylow subgroup: Generic Abelian Group over
Residue class ring of integers modulo 34384
Abelian Group isomorphic to Z/2 + Z/2 + Z/2 + Z/4
Defined on 4 generators in supergroup G:
  GAp.1 = G.1
  GAp.2 = G.2
```

GAp.3 = 3\*G.3  
 GAp.4 = 153\*G.4

Relations:

2\*GAp.1 = 0  
 2\*GAp.2 = 0  
 2\*GAp.3 = 0  
 4\*GAp.4 = 0

---

### 69.13 Subgroup Chains

**CompositionSeries(G)**

A composition series for the finite abelian group  $G$  returned as a sequence of subgroups.

**Agemo(G, i)**

Given a finite  $p$ -group  $G$ , return the characteristic subgroup of  $G$  generated by the elements  $x^{p^i}$ ,  $x \in G$ , where  $i$  is a positive integer.

**Omega(G, i)**

Given a finite  $p$ -group  $G$ , return the characteristic subgroup of  $G$  generated by the elements of order dividing  $p^i$ , where  $i$  is a positive integer.

### 69.14 General Group Properties

**IsCyclic(G)**

Returns **true** if the group  $G$  is cyclic, **false** otherwise.

**IsElementaryAbelian(G)**

Returns **true** if the group  $G$  is elementary abelian, **false** otherwise.

**IsFree(G)**

Returns **true** if  $G$  is free, **false** otherwise.

**IsMixed(G)**

Returns **true** if  $G$  is a mixed group, **false** otherwise. An abelian group is mixed if it is neither a torsion group nor free.

**IspGroup(G)**

Returns **true** if the finite group  $G$  is a  $p$ -group, ie. if all elements have order a power of  $p$ .

### 69.14.1 Properties of Subgroups

`IsMaximal(G, H)`

Returns `true` if the subgroup  $H$  of the finite group  $G$  is a maximal subgroup of  $G$ , `false` otherwise.

`Index(G, H)`

The index of the subgroup  $H$  in the group  $G$ , returned as an ordinary integer. If  $H$  has infinite index in  $G$ , the value zero is returned.

`FactoredIndex(G, H)`

The factored index of the subgroup  $H$  in the group  $G$ , returned as a sequence of prime-exponent pairs. If  $H$  has infinite index in  $G$ , the empty sequence is returned.

`IsPure(G, H)`

Returns `true` if the subgroup  $H$  of the finite group  $G$  is pure, i.e. if for all  $n$  we have  $nG \cap H = nH$ .

`IsNeat(G, H)`

Returns `true` if the subgroup  $H$  of the finite group  $G$  is neat, i.e., if for all primes  $p$  we have  $pG \cap H = pH$ .

### 69.14.2 Enumeration of Subgroups

`MaximalSubgroups(G)`

The maximal subgroups of the finite group  $G$  returned as a sequence of subgroups.

`Subgroups(G: parameters)`

The subgroups of the finite group  $G$  are returned as a sequence of records. The record fields are `subgroup`, storing the actual group; `order`, storing the group order; and `length`, storing the length of the conjugacy class, which is always 1 for abelian groups.

`Sub` [RNGINTELT] *Default* : []

If parameter `Sub` is set, only subgroups with invariants equal to the given sequence are found. The given sequence should contain positive integers, such that each divides the following.

`Quot` [RNGINTELT] *Default* : []

If parameter `Quot` is set, only subgroups such that the quotient group has invariants equal to the given sequence are found. The given sequence should contain positive integers, such that each divides the following.

NumberOfSubgroupsAbelianPGroup (A)
------------------------------------

Return the number of subgroups of each non-trivial order in the abelian  $p$ -group  $G$  where  $A = [a_1, a_2, \dots]$  and  $G = C_{a_1} \times C_{a_2} \times \dots$ . The  $m$ -th entry in the sequence returned is the number of subgroups of order  $p^m$ .

HasComplement(G, U)
---------------------

For a finite abelian group  $G$  and a subgroup  $U$  decide if there exist some other subgroup  $V$  such that  $G = U + V$  and  $U \cap V = \{0\}$ . In case such a  $V$  exists, it is returned as the second value.

---

**Example H69E13**

We look at subgroups of an abelian group of order 12.

```
> G := AbelianGroup([2,6]);
> s := Subgroups(G); #s;
10
> s[7];
rec<recformat<order, length, subgroup, presentation> |
  order := 3, length := 1,
  subgroup := Abelian Group isomorphic to Z/3
Defined on 1 generator in supergroup G:
$.1 = 2*G.2
Relations:
3*$.1 = 0>
> [x'order:x in s];
[ 12, 6, 4, 2, 6, 6, 3, 2, 2, 1 ]
```

Now we find the elementary abelian subgroup of order 4.

```
> s22 := Subgroups(G:Sub := [2,2]); #s22;
1
> s22;
Conjugacy classes of subgroups
-----
[1]      Order 4          Length 1
      Abelian Group isomorphic to Z/2 + Z/2
      Defined on 2 generators in supergroup G:
          $.1 = G.1
          $.2 = 3*G.2
      Relations:
          2*$.1 = 0
          2*$.2 = 0
```

There is more than one subgroup of index 2 in  $G$ .

```
> q2 := Subgroups(G:Quot := [2]); #q2;
3
> q2[3] 'subgroup;
```

Abelian Group isomorphic to  $Z/6$

Defined on 1 generator in supergroup  $G$ :

$$$.1 = G.1 + G.2$$

Relations:

$$6*$.1 = 0$$


---

## 69.15 Representation Theory

`CharacterTable(G)`

The table of irreducible characters for the group  $G$ .

## 69.16 The Hom Functor

`Hom(G, H)`

Given finite abelian groups  $G$  and  $H$ , return an abelian group  $A$  isomorphic to  $\text{Hom}(G, H)$ , and a transfer map  $t$  such that, given an element  $a$  of  $A$ ,  $t(a)$  yields the corresponding (MAGMA Map type) homomorphism from  $G$  to  $H$ . The structure of  $\text{Hom}(G, H)$  may thus be analyzed by examining  $A$ .

`HomGenerators(G, H)`

Given finite abelian groups  $G$  and  $H$ , return a sequence of ( $Z$ -module) generators of  $\text{Hom}(G, H)$ . The generators are returned as actual (MAGMA Map type) homomorphisms. Note that  $\text{Hom}(G, H)$  is usually not free, so it is difficult to generate all homomorphisms uniquely using the generators alone (use `Hom` or `Homomorphisms` if that is desired).

`Homomorphisms(G, H)`

Given finite abelian groups  $G$  and  $H$ , return a sequence containing all elements of  $\text{Hom}(G, H)$ . The elements are returned as actual (MAGMA Map type) homomorphisms. Note that this function simply uses `Hom`, transferring each element of the returned group to the actual MAGMA Map type homomorphism.

**Example H69E14**

---

We examine  $A = \text{Hom}(G, H)$ , for certain abelian groups  $G$  and  $H$ .

```
> G := AbelianGroup([2, 3]);
> H := AbelianGroup([4, 6]);
> A, t := Hom(G, H);
> #A;
12
> A;
Abelian Group isomorphic to Z/2 + Z/6
Defined on 2 generators
Relations:
    2*A.1 = 0
    6*A.2 = 0
> h := t(A.1);
> h;
Mapping from: GrpAb: G to GrpAb: H
> h(G.1);
3*H.2
> h(G.2);
0
```

We now enumerate all elements of  $A$  and examine the images of each generator of  $G$  under each homomorphism. We note that each possible list of images occurs only once.

```
> I := [<h(G.1), h(G.2)> where h is t(x): x in A];
> I;
[
    <0, 0>,
    <3*H.2, 0>,
    <2*H.1, 2*H.2>,
    <2*H.1 + 3*H.2, 2*H.2>,
    <0, 4*H.2>,
    <3*H.2, 4*H.2>,
    <2*H.1, 0>,
    <2*H.1 + 3*H.2, 0>,
    <0, 2*H.2>,
    <3*H.2, 2*H.2>,
    <2*H.1, 4*H.2>,
    <2*H.1 + 3*H.2, 4*H.2>
]
> #I;
12
> #Set(I);
12
```

---

## 69.17 Automorphism Groups

The full automorphism group of the abelian group  $G$ .

## 69.18 Cohomology

`Dual(G)`

Computes the dual group  $G^*$  of  $G$  and a map  $M$  from  $G \times G^* \rightarrow \mathbf{Z}/m\mathbf{Z}$  for  $m$  the exponent of  $G$  that allows  $G^*$  to act on  $G$ .  $G$  must be finite.

`H2_G_QmodZ(G)`

Computes  $H := H^2(G, \mathbf{Q}/\mathbf{Z})$  and a map  $f : H \rightarrow (G \times G \rightarrow \mathbf{Z}/m\mathbf{Z})$  that will give the cocycles as maps from  $G \times G \rightarrow \mathbf{Z}/m\mathbf{Z}$ .  $m := \#G$ .

`Res_H2_G_QmodZ(U, H2)`

For a subgroup  $U$  of  $G$  and  $H2 = H^2(G, \mathbf{Q}/\mathbf{Z})$  computes  $H^2(U, \mathbf{Q}/\mathbf{Z})$  in a compatible way together with the restriction map into  $H2$ .

$H2$  must be the result of `H2_G_QmodZ` as this function relies on the attributes stored in there.

## 69.19 Homomorphisms

Two functions are provided to construct homomorphisms or isomorphisms from one group into another, where either of the groups, or both, may be generic abelian groups.

`hom< A -> B | L >`

Given groups  $A$  and  $B$ , construct a homomorphism from  $A$  to  $B$  as defined by the extension  $L$ . If one or both of  $A$  and  $B$  are generic abelian groups this works as usual, with one minor difference as explained below. Suppose that the generators of  $A$  are  $g_1, \dots, g_n$ , and that  $\phi(g_i) = h_i$  for each  $i$ , where  $\phi$  is the homomorphism one wishes to construct. The list  $L$  as required by the constructors must be one of the following:

- (a) a list of the  $n$  2-tuples  $\langle g_i, h_i \rangle$  (order not important);
- (b) a list of the  $n$  arrow-pairs  $g_i \rightarrow h_i$  (order not important);
- (c)  $h_1, \dots, h_n$  (order is important).

If  $A$  is a generic abelian group this rule is relaxed somewhat in the following sense: If  $L$  is a list of  $n$  2-tuples or of  $n$  arrow-pairs, the elements  $g_i$  need not be the defining generators of  $A$ . The only requirement is that the set  $\{g_1, \dots, g_n\}$  does actually generate the whole of  $A$ .

Homomorphism(A, B, X, Y)
--------------------------

Creates a homomorphism from  $A$  into  $B$  as given by the mapping of  $X$  into  $Y$ . The arguments  $A$  and  $B$  may be any type of group, including of course, generic abelian groups.

The function `Homomorphism` does not require the elements of the argument  $X$  to be generators of  $A$  as given by `Generators(A)`, so it allows more freedom when creating a homomorphism. If, however, these elements fail to generate the whole of  $A$  then the subsequent map application will fail.

iso< A -> B   L >
-------------------

Given groups  $A$  and  $B$ , construct an isomorphism from  $A$  to  $B$  as defined by the extension  $L$ . If one or both of  $A$  and  $B$  are generic abelian groups this works as usual, with one minor difference as explained below. Suppose that the generators of  $A$  are  $g_1, \dots, g_n$ , and that  $\phi(g_i) = h_i$  for each  $i$ , where  $\phi$  is the isomorphism one wishes to construct. The list  $L$  as required by the constructors must be one of the following:

- (a) a list of the  $n$  2-tuples  $\langle g_i, h_i \rangle$  (order not important);
- (b) a list of the  $n$  arrow-pairs  $g_i \rightarrow h_i$  (order not important);
- (c)  $h_1, \dots, h_n$  (order is important).

If  $A$  is a generic abelian group this rule is relaxed somewhat in the following sense: If  $L$  is a list of  $n$  2-tuples or of  $n$  arrow-pairs, the elements  $g_i$  may not necessarily be generators of  $A$  as given by the function `Generators(A)`. The only requirement is that the set  $\{h_1, \dots, h_n\}$  does actually generate the whole of  $B$ .

Isomorphism(A, B, X, Y)
-------------------------

Creates a isomorphism from  $A$  into  $B$  as given by the mapping of  $X$  into  $Y$ . The arguments  $A$  and  $B$  can be any type of group, including of course, generic abelian groups.

The function `Isomorphism` does not require the elements of the argument  $X$  to be generators of  $A$  as given by `Generators(A)`, so it allows more freedom when creating an isomorphism. If, however, these elements fail to generate the whole of  $A$  then the subsequent map application will fail.

### Example H69E15

---

Recall that we defined the subgroups  $GH1_{Z_m}$  and  $GH2_{Z_m}$  of  $G$  as:

```
> GH1_Zm;
Generic Abelian Group over
Residue class ring of integers modulo 34384
Abelian Group isomorphic to Z/6 + Z/612
Defined on 2 generators in supergroup G:
  GH1_Zm.1 = G.2 + G.3
  GH1_Zm.2 = G.4
```

Relations:

$$6 * \text{GH1\_Zm}.1 = 0$$

$$612 * \text{GH1\_Zm}.2 = 0$$

> GH2\_Zm;

Generic Abelian Group over

Residue class ring of integers modulo 34384

Abelian Group isomorphic to  $\mathbb{Z}/2 + \mathbb{Z}/2 + \mathbb{Z}/6 + \mathbb{Z}/612$

Defined on 4 generators in supergroup G:

$$\text{GH2\_Zm}.1 = \text{G}.1$$

$$\text{GH2\_Zm}.2 = \text{G}.2$$

$$\text{GH2\_Zm}.3 = \text{G}.3$$

$$\text{GH2\_Zm}.4 = \text{G}$$

Relations:

$$2 * \text{GH2\_Zm}.1 = 0$$

$$2 * \text{GH2\_Zm}.2 = 0$$

$$6 * \text{GH2\_Zm}.3 = 0$$

$$612 * \text{GH2\_Zm}.4 = 0$$

We construct the homomorphism

> h := hom<GH1\_Zm -> GH2\_Zm | GH2\_Zm.1, GH2\_Zm.2 >;

> h(GH1\_Zm);

Generic Abelian Group over

Residue class ring of integers modulo 34384

Abelian Group isomorphic to  $\mathbb{Z}/2 + \mathbb{Z}/2$

Defined on 2 generators in supergroup GH2\_Zm:

$$$.1 = \text{GH2\_Zm}.2$$

$$$.2 = \text{GH2\_Zm}.1$$

Relations:

$$2 * $.1 = 0$$

$$2 * $.2 = 0$$

but we cannot construct the isomorphism

> i := iso<GH1\_Zm -> GH2\_Zm | GH2\_Zm.1, GH2\_Zm.2 >;

>> i := iso<GH1\_Zm -> GH2\_Zm | GH2\_Zm.1, GH2\_Zm.2 >;

^

Runtime error in map< ... >: Images do not generate the (whole) codomain

An alternative way of creating the homomorphism  $h$  would be

> h := Homomorphism(GH1\_Zm, GH2\_Zm, Generators(GH1\_Zm), [GH2\_Zm.1, GH2\_Zm.2]);

## 69.20 Bibliography

- [**BJT97**] J. Buchmann, M. J. Jacobson, Jr., and E. Teske. On Some Computational Problems in Finite Abelian Groups. *Mathematics of Computation*, 66:1663–1687, 1997.
- [**Bos00**] Wieb Bosma, editor. *ANTS IV*, volume 1838 of *LNCS*. Springer-Verlag, 2000.
- [**CLGM<sup>+</sup>95**] Frank Celler, Charles R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and E. A. O’Brien. Generating random elements of a finite group. *Comm. Algebra*, 23(13):4931–4948, 1995.
- [**GH00**] P. Gaudry and R. Harley. Counting Points on Hyperelliptic Curves over Finite Fields. In Bosma [Bos00], pages 313–332.
- [**Tes98a**] E. Teske. A Space Efficient Algorithm for Group Structure Computation. *Mathematics of Computation*, 67:1637–1663, 1998.
- [**Tes98b**] E. Teske. Better Random Walks for Pollard’s Rho Method. 1998.



# 70 FINITELY PRESENTED GROUPS

<b>70.1 Introduction . . . . .</b>	<b>2079</b>	/	2088
70.1.1 Overview of Facilities . . . . .	2079	70.3.2 The FP-Group Constructor . . . . .	2089
70.1.2 The Construction of Finitely Presented Groups . . . . .	2079	Group< >	2089
<b>70.2 Free Groups and Words . . . . .</b>	<b>2080</b>	70.3.3 Construction from a Finite Permutation or Matrix Group . . . . .	2090
70.2.1 Construction of a Free Group . . . . .	2080	FPGroup(G)	2090
FreeGroup(n)	2080	FPGroupStrong(G)	2091
70.2.2 Construction of Words . . . . .	2081	FPGroupStrong(G, N)	2091
!	2081	70.3.4 Construction of the Standard Presentation for a Coxeter Group . . . . .	2092
Identity(G)	2081	CoxeterGroup(GrpFP, W)	2092
Id(G)	2081	70.3.5 Conversion from a Special Form of FP-Group . . . . .	2093
!	2081	FPGroup(G)	2093
Random(G, m, n)	2081	70.3.6 Construction of a Standard Group . . . . .	2094
70.2.3 Access Functions for Words . . . . .	2081	AbelianGroup(GrpFP, [n <sub>1</sub> , . . . , n <sub>r</sub> ])	2094
#	2081	AlternatingGroup(GrpFP, n)	2094
ElementToSequence(w)	2081	Alt(GrpFP, n)	2094
Eltseq(w)	2081	BraidGroup(GrpFP, n)	2094
ExponentSum(w, x)	2081	CoxeterGroup(GrpFP, t)	2094
Weight(w, x)	2081	CyclicGroup(GrpFP, n)	2095
GeneratorNumber(w)	2082	DihedralGroup(GrpFP, n)	2095
LeadingGenerator(w)	2082	ExtraSpecialGroup(GrpFP, p, n : -)	2095
Parent(w)	2082	SymmetricGroup(GrpFP, n)	2095
70.2.4 Arithmetic Operators for Words . . . . .	2083	Sym(GrpFP, n)	2095
*	2083	70.3.7 Construction of Extensions . . . . .	2096
~	2083	Darstellungsgruppe(G)	2096
~	2083	DirectProduct(G, H)	2096
(u, v)	2083	DirectProduct(Q)	2096
(u <sub>1</sub> , . . . , u <sub>n</sub> )	2083	FreeProduct(G, H)	2096
70.2.5 Comparison of Words . . . . .	2084	FreeProduct(Q)	2096
eq	2084	70.3.8 Accessing the Defining Generators and Relations . . . . .	2098
ne	2084	.	2098
lt	2084	Generators(G)	2098
le	2084	NumberOfGenerators(G)	2098
ge	2084	Ngens(G)	2098
gt	2084	PresentationLength(G)	2098
70.2.6 Relations . . . . .	2085	Relations(G)	2098
=	2085	<b>70.4 Homomorphisms . . . . .</b>	<b>2098</b>
r[1]	2085	70.4.1 General Remarks . . . . .	2098
LHS(r)	2085	70.4.2 Construction of Homomorphisms . . . . .	2099
r[2]	2086	hom< >	2099
RHS(r)	2086	IsSatisfied(U, E)	2099
r[1] := w	2086	70.4.3 Accessing Homomorphisms . . . . .	2099
r[2] := w	2086	w @ f	2099
f(r)	2086	f(w)	2099
Parent(r)	2086	H @ f	2099
<b>70.3 Construction of an FP-Group . . . . .</b>	<b>2087</b>	f(H)	2099
70.3.1 The Quotient Group Constructor . . . . .	2087		
quo< >	2087		

<code>g @@ f</code>	2100	<code>AbelianQuotientInvariants(G, T, n)</code>	2124
<code>H @@ f</code>	2100	<code>AQInvariants(G, T, n)</code>	2124
<code>Domain(f)</code>	2100	<code>HasComputableAbelianQuotient(G)</code>	2124
<code>Codomain(f)</code>	2100	<code>HasInfiniteComputableAbelian</code>	
<code>Image(f)</code>	2100	<code>Quotient(G)</code>	2125
<code>Kernel(f)</code>	2100	<code>IsPerfect(G)</code>	2125
<i>70.4.4 Computing Homomorphisms to Finite Groups.</i>	<i>2102</i>	<code>TorsionFreeRank(G)</code>	2125
<code>Homomorphisms(F, G, A : -)</code>	2102	<i>70.5.2 p-Quotient</i>	<i>2126</i>
<code>Homomorphisms(F, G : -)</code>	2102	<i>70.5.3 The Construction of a p-Quotient</i>	<i>2127</i>
<code>Homomorphisms(F, G, A : -)</code>	2103	<code>pQuotient(F, p, c: -)</code>	2127
<code>Homomorphisms(F, G : -)</code>	2103	<i>70.5.4 Nilpotent Quotient</i>	<i>2129</i>
<code>HomomorphismsProcess(F, G, A : -)</code>	2104	<code>NilpotentQuotient(G, c: -)</code>	2130
<code>HomomorphismsProcess(F, G : -)</code>	2104	<code>SetVerbose("NilpotentQuotient", n)</code>	2134
<code>NextElement(~P)</code>	2104	<i>70.5.5 Soluble Quotient</i>	<i>2135</i>
<code>Complete(~P)</code>	2105	<code>SolvableQuotient(G : -)</code>	2135
<code>IsEmpty(P)</code>	2105	<code>SolubleQuotient(G : -)</code>	2135
<code>IsValid(P)</code>	2105	<code>SolvableQuotient(F, n : -)</code>	2136
<code>DefinesHomomorphism(P)</code>	2105	<code>SolubleQuotient(F, n : -)</code>	2136
<code>Homomorphism(P)</code>	2105	<code>SolvableQuotient(F, P : -)</code>	2136
<code>#</code>	2105	<code>SolubleQuotient(F, P : -)</code>	2136
<code>Homomorphisms(P)</code>	2105	<b>70.6 Subgroups</b>	<b>2138</b>
<code>SimpleQuotients(F, deg1, deg2, ord1, ord2: -)</code>	2107	<i>70.6.1 Specification of a Subgroup</i>	<i>2138</i>
<code>SimpleQuotients(F, ord1, ord2: -)</code>	2107	<code>sub&lt; &gt;</code>	2138
<code>SimpleQuotients(F, ord2: -)</code>	2107	<code>sub&lt; &gt;</code>	2138
<code>SimpleQuotientProcess(F, deg1, deg2, ord1, ord2: -)</code>	2108	<code>ncl&lt; &gt;</code>	2138
<code>NextSimpleQuotient(~P)</code>	2108	<code>ncl&lt; &gt;</code>	2139
<code>IsEmptySimpleQuotientProcess(P)</code>	2108	<code>CommutatorSubgroup(G)</code>	2139
<code>SimpleEpimorphisms(P)</code>	2108	<code>DerivedSubgroup(G)</code>	2139
<i>70.4.5 The L<sub>2</sub>-Quotient Algorithm.</i>	<i>2110</i>	<code>DerivedGroup(G)</code>	2139
<code>L2Quotients(G)</code>	2110	<i>70.6.2 Index of a Subgroup: The Todd-Coxeter Algorithm.</i>	<i>2140</i>
<code>L2Type(P)</code>	2110	<code>ToddCoxeter(G, H: -)</code>	2141
<code>L2Generators(P)</code>	2110	<code>Index(G, H: -)</code>	2141
<code>L2Ideals(I)</code>	2110	<code>FactoredIndex(G, H: -)</code>	2141
<i>70.4.6 Infinite L<sub>2</sub> quotients</i>	<i>2117</i>	<code>Order(G: -)</code>	2142
<code>HasInfinitePSL2Quotient(G)</code>	2118	<code>FactoredOrder(G: -)</code>	2142
<i>70.4.7 Searching for Isomorphisms</i>	<i>2121</i>	<i>70.6.3 Implicit Invocation of the Todd-Coxeter Algorithm.</i>	<i>2145</i>
<code>SearchForIsomorphism(F, G, m : -)</code>	2121	<code>SetGlobalTCPParameters(: -)</code>	2145
<b>70.5 Abelian, Nilpotent and Soluble Quotient</b>	<b>2123</b>	<code>UnsetGlobalTCPParameters()</code>	2145
<i>70.5.1 Abelian Quotient</i>	<i>2123</i>	<i>70.6.4 Constructing a Presentation for a Subgroup.</i>	<i>2146</i>
<code>AbelianQuotient(G)</code>	2123	<code>Rewrite(G, H : -)</code>	2147
<code>ElementaryAbelianQuotient(G, p)</code>	2123	<code>Rewrite(G, ~H : -)</code>	2148
<code>AbelianQuotientInvariants(G)</code>	2123	<b>70.7 Subgroups of Finite Index</b>	<b>2150</b>
<code>AQInvariants(G)</code>	2123	<i>70.7.1 Low Index Subgroups</i>	<i>2150</i>
<code>AbelianQuotientInvariants(H)</code>	2124	<code>LowIndexSubgroups(G, R : -)</code>	2150
<code>AQInvariants(H)</code>	2124	<code>LowIndexProcess(G, R : -)</code>	2154
<code>AbelianQuotientInvariants(G, T)</code>	2124	<code>NextSubgroup(~P)</code>	2155
<code>AQInvariants(G, T)</code>	2124	<code>NextSubgroup(~P, ~G)</code>	2155
<code>AbelianQuotientInvariants(G, n)</code>	2124	<code>ExtractGroup(P)</code>	2155
<code>AQInvariants(G, n)</code>	2124	<code>ExtractGenerators(P)</code>	2155
<code>AbelianQuotientInvariants(H, n)</code>	2124	<code>IsEmpty(P)</code>	2155
<code>AQInvariants(H, n)</code>	2124		

IsValid(P)	2155	IndexedCoset(V, C)	2172
LowIndexNormalSubgroups(G, n: -)	2158	Group(V)	2172
70.7.2 Subgroup Constructions . . . . .	2159	Subgroup(V)	2172
~	2159	IsComplete(V)	2172
Conjugate(H, u)	2159	ExcludedConjugates(V)	2173
meet	2159	ExcludedConjugates(T)	2173
Core(G, H)	2159	Transversal(G, H)	2173
GeneratingWords(G, H)	2159	RightTransversal(G, H)	2173
MaximalOvergroup(G, H)	2160	70.8.5 Double Coset Spaces: Construction	2176
MinimalOvergroup(G, H)	2160	DoubleCoset(G, H, g, K)	2176
~	2160	DoubleCosets(G, H, K)	2176
NormalClosure(G, H)	2160	70.8.6 Coset Spaces: Selection of Cosets .	2177
Normaliser(G, H)	2160	CosetsSatisfying(T, S: -)	2177
Normalizer(G, H)	2160	CosetSatisfying(T, S: -)	2177
SchreierGenerators(G, H: -)	2160	CosetsSatisfying(V, S: -)	2177
SchreierSystem(G, H)	2160	CosetSatisfying(V, S: -)	2177
Transversal(G, H)	2160	70.8.7 Coset Spaces: Induced	
Transversal(G, H, K)	2161	Homomorphism . . . . .	2178
70.7.3 Properties of Subgroups . . . . .	2164	CosetAction(G, H)	2178
in	2164	CosetAction(V)	2178
notin	2164	CosetImage(G, H)	2178
eq	2164	CosetImage(V)	2179
ne	2164	CosetKernel(G, H)	2179
subset	2165	CosetKernel(V)	2179
notsubset	2165	<b>70.9 Simplification . . . . .</b>	<b>2181</b>
IsConjugate(G, H, K)	2165	70.9.1 Reducing Generating Sets . . . . .	2181
IsNormal(G, H)	2165	ReduceGenerators(G)	2181
IsMaximal(G, H)	2165	70.9.2 Tietze Transformations . . . . .	2181
IsSelfNormalizing(G, H)	2165	Simplify(G: -)	2181
<b>70.8 Coset Spaces and Tables . . .</b>	<b>2168</b>	SimplifyLength(G: -)	2183
70.8.1 Coset Tables . . . . .	2168	TietzeProcess(G: -)	2183
CosetTable(G, H: -)	2168	ShowOptions(~P: -)	2184
CosetTableToRepresentation(G, T)	2169	SetOptions(~P: -)	2184
CosetTableToPermutationGroup(G, T)	2169	Simplify(~P: -)	2184
70.8.2 Coset Spaces: Construction . . . . .	2170	SimplifyPresentation(~P: -)	2184
CosetSpace(G, H: -)	2171	SimplifyLength(~P: -)	2184
RightCosetSpace(G, H: -)	2171	Eliminate(~P: -)	2184
LeftCosetSpace(G, H: -)	2171	EliminateGenerators(~P: -)	2184
70.8.3 Coset Spaces: Elementary Opera-		Search(~P: -)	2185
tions . . . . .	2171	SearchEqual(~P: -)	2185
*	2171	Group(P)	2185
*	2171	NumberOfGenerators(P)	2185
*	2171	Ngens(P)	2185
in	2171	NumberOfRelations(P)	2185
notin	2171	Nrels(P)	2185
eq	2172	PresentationLength(P)	2185
ne	2172	<b>70.10 Representation Theory . . .</b>	<b>2192</b>
70.8.4 Accessing Information . . . . .	2172	GModulePrimes(G, A)	2192
#	2172	GModulePrimes(G, A, B)	2193
Action(V)	2172	GModule(G, A, p)	2193
<i, w> @ T	2172	GModule(G, A, B, p)	2193
T(i, w)	2172	GModule(G, A, B)	2193
ExplicitCoset(V, i)	2172	Pullback(f, N)	2193
IndexedCoset(V, w)	2172	<b>70.11 Small Group Identification .</b>	<b>2196</b>

IdentifyGroup(G)	2196	PermutationGroup(G)	2198
70.11.1 Concrete Representations of Small Groups . . . . .	2198	PGGroup(G)	2198
		<b>70.12 Bibliography . . . . .</b>	<b>2198</b>

# Chapter 70

## FINITELY PRESENTED GROUPS

### 70.1 Introduction

This Chapter presents the functions designed for computing with finitely-presented groups (fp-groups for short). The name of the corresponding MAGMA category is `GrpFP`. The functions considered here are designed for doing what is sometimes referred to as *combinatorial group theory*.

#### 70.1.1 Overview of Facilities

The facilities provided for fp-groups fall into a number of natural groupings:

- The construction of fp-groups in terms of generators and relations;
- The construction of particular types of quotient groups: abelian quotient,  $p$ -quotient, nilpotent quotient and soluble quotient;
- Index determination and subgroup building based on the Todd-Coxeter procedure;
- Calculations with subgroups having finite index in a group, where the subgroups are represented by coset tables;
- The construction of all subgroups having index less than some (small) specified bound;
- The construction of representations of an fp-group corresponding to actions on coset spaces and elementary abelian sections;
- The use of a rewriting process for constructing presentations of subgroups;
- The simplification of words with respect to a given set of relations.

For a description of fundamental algorithms for finitely presented groups, we refer the reader to [Sim94].

#### 70.1.2 The Construction of Finitely Presented Groups

The construction of fp-groups utilises the fact that every group is a quotient of some free group. Thus, two general fp-group constructors are provided: `FreeGroup(n)` which constructs a free group of rank  $n$ , and `quo< F | R >` which constructs the quotient of group  $F$  by the normal subgroup defined by the relations  $R$ .

The naming of generators presents special difficulties since they are not always used in a consistent manner in the mathematical literature. A generator name is used in two distinct ways. Firstly, it plays the role of a *variable* having as its value a designated generator of  $G$ . Secondly, it appears as the *symbol* designating the specified generator whenever elements of the group are output. These two uses are separated in the MAGMA semantics.

In MAGMA, a standard indexing notation is provided for referencing the generators of any fp-group  $G$ . Thus, `G.i` denotes the  $i$ -th generator of  $G$ . However, users may give

individual names to the generators by means of the *generator-assignment*. Suppose that the group  $G$  is defined on  $r$  generators. Then if the right hand side of the following statement creates a group, the special assignment

```
> G< v_1, ..., v_r> := construction;
```

is equivalent to the statements

```
> G := construction;
>   v_1 := G.1;
>   ...
>   v_r := G.r;
```

It should be noted that when the fp-group  $G$  is created as the quotient of the group  $F$ , any names that the user may have associated with the generators of  $F$  will not be associated with the corresponding generators of  $G$ . If this were allowed, then it would violate the fundamental principle that every object is viewed as belonging to a *unique* structure.

## 70.2 Free Groups and Words

### 70.2.1 Construction of a Free Group

FreeGroup(n)
--------------

Construct the free group  $F$  of rank  $n$ , where  $n$  is a positive integer.

The  $i$ -th generator of  $F$  may be referenced by the expression  $F.i$ ,  $i = 1, \dots, n$ . Note that a special form of the assignment statement is provided which enables the user to assign names to the generators of  $F$ . In this form of assignment, the list of generator names is enclosed within angle brackets and appended to the variable name on the *left hand side* of the assignment statement:  $F< v_1, \dots, v_n > := \text{FreeGroup}(n)$ ;

---

#### Example H70E1

The statement

```
> F := FreeGroup(2);
```

creates the free group of rank 2. Here the generators may be referenced using the standard names,  $F.1$  and  $F.2$ .

The statement

```
> F<x, y> := FreeGroup(2);
```

defines  $F$  to be the free group of rank 2 and assigns the names  $x$  and  $y$  to the two generators.

---

### 70.2.2 Construction of Words

The operations in this section apply to both, free groups and arbitrary fp-groups.

**G ! [ i<sub>1</sub>, ..., i<sub>s</sub> ]**

Given a group  $G$  defined on  $r$  generators and a sequence  $[i_1, \dots, i_s]$  of integers lying in the range  $[-r, r]$ , excluding 0, construct the word

$$G.|i_1|^{\epsilon_1} * G.|i_2|^{\epsilon_2} * \dots * G.|i_s|^{\epsilon_s}$$

where  $\epsilon_j$  is +1 if  $i_j$  is positive, and -1 if  $i_j$  is negative.

**Identity(G)**

**Id(G)**

**G ! 1**

Construct the identity element, represented as the empty word, for the fp-group  $G$ . For a sample application of this function, see Example H70E3.

**Random(G, m, n)**

A random word of length  $l$  in the generators of the group  $G$ , where  $m \leq l \leq n$ . For a sample application of this function, see Example H70E3.

### 70.2.3 Access Functions for Words

This section describes some basic access functions for words. These operations apply to both, free groups and arbitrary fp-groups.

**#w**

The length of the word  $w$ .

**ElementToSequence(w)**

**Eltseq(w)**

The sequence  $Q$  obtained by decomposing the word  $w$  into its constituent generators and generator inverses. Suppose  $w$  is a word in the group  $G$ . Then, if  $w = G.i_1^{e_1} \dots G.i_m^{e_m}$ , with each  $e_i = \pm 1$ , then  $Q[j] = i_j$  if  $e_j = +1$  and  $Q[j] = -i_j$  if  $e_j = -1$ , for  $j = 1, \dots, m$ .

**ExponentSum(w, x)**

**Weight(w, x)**

Given a word  $w$ , and the name of a generator  $x$  of a group  $G$ , compute the sum of the exponents of the generator  $x$  in the word  $w$ . For a sample application of this function, see Example H70E3.

**GeneratorNumber(w)**

Suppose  $w$  is a word belonging to a group  $G$ . Assume  $x$  is the name of the  $i$ -th generator of  $G$ . Then

- (i) if  $w = \text{Identity}(G)$ ,  $\text{GeneratorNumber}(w)$  is 0;
- (ii) if  $w = x * w'$ ,  $w'$  a word in  $G$ ,  $\text{GeneratorNumber}(w)$  is  $i$ ;
- (iii) if  $w = x^{-1} * w'$ ,  $w'$  a word in  $G$ ,  $\text{GeneratorNumber}(w)$  is  $-i$ .

**LeadingGenerator(w)**

Suppose  $w$  is a word belonging to a group  $G$ . If  $w = x^\epsilon * w'$ ,  $w'$  a word in  $G$ ,  $x$  a generator of  $G$  and  $\epsilon \in \{-1, +1\}$ , the function returns  $x^\epsilon$ . If  $w = \text{Identity}(G)$ , it returns  $\text{Identity}(G)$ . For a sample application of this function, see Example H70E3.

**Parent(w)**

The parent group  $G$  of the word  $w$ .

**Example H70E2**

---

Consider the free group  $F$  on 6 generators

```
> F<u,v,w,x,y,z> := FreeGroup(6);
```

and the sequence of words

```
> rels := [ (u*v)^42, (v,x), (x*z^2)^4,
>           v^2*y^3, (v*z)^3, y^4, (x*z)^3 ];
```

The abelianised relation matrix of the quotient

$$\langle u, v, w, x, y, z \mid (uv)^{42}, (v, x), (xz^2)^4, v^2y^3, (vz)^3, y^4, (xz)^3 \rangle$$

can be obtained using the following construction

```
> R := Matrix(Integers(),
>             [ [ Weight(r, F.j) : j in [1..6] ] : r in rels ]);
> R;
[42 42 0 0 0 0]
[ 0 0 0 0 0 0]
[ 0 0 0 4 0 8]
[ 0 2 0 0 3 0]
[ 0 3 0 0 0 3]
[ 0 0 0 0 4 0]
[ 0 0 0 3 0 3]
```

(The function `Matrix` constructs a matrix from a sequence of row vectors.)

---

### 70.2.4 Arithmetic Operators for Words

Suppose  $G$  is an fp-group for which generators have already been defined. This subsection defines the elementary arithmetic operations on words that are derived from the multiplication and inversion operators. The availability of the operators defined here enables the user to construct an element (*word*) of  $G$  in terms of the generators as follows:

- (i) A generator is a word;
- (ii) The expression  $(u)$  is a word, where  $u$  is a word;
- (iii) The product  $u * v$  of the words  $u$  and  $v$  is a word;
- (iv) The conjugate  $u^v$  of the word  $u$  by the word  $v$ , is a word ( $u^v$  expands into the word  $v^{-1} * u * v$ );
- (v) The power of a word,  $u^n$ , where  $u$  is a word and  $n$  is an integer, is a word;
- (vi) The commutator  $(u, v)$  of the words  $u$  and  $v$  is a word ( $(u, v)$  expands into the word  $u^{-1} * v^{-1} * u * v$ ). Note that  $(u, v, w)$  is equivalent to  $((u, v), w)$ , i.e. commutators are *left-normed*.

The word operations defined here may be applied either to the words of a free group or the words of a group with non-trivial relations. If such an operator is applied to a group possessing non-trivial relations, only free reduction will be applied to the resulting words.

$u * v$

Given words  $u$  and  $v$  belonging to the same fp-group  $G$ , return the product of  $u$  and  $v$ .

$u \hat{=} n$

The  $n$ -th power of the word  $u$ , where  $n$  is an integer. When invoked with  $n = -1$ , the function computes the inverse of  $u$ . When invoked with  $n = 0$ , the function returns the identity element.

$u \hat{=} v$

Given words  $u$  and  $v$  belonging to the same fp-group  $G$ , return the conjugate  $v^{-1} * u * v$  of the word  $u$  by the word  $v$ .

$(u, v)$

Given words  $u$  and  $v$  belonging to the same fp-group  $G$ , return the commutator  $u^{-1}v^{-1}uv$  of the words  $u$  and  $v$ .

$(u_1, \dots, u_n)$

Given the  $n$  words  $u_1, \dots, u_n$  belonging to the same fp-group  $G$ , return their commutator. Commutators are *left-normed*, so that they are evaluated from left to right.

### 70.2.5 Comparison of Words

Words in an fp-group may be compared both for equality and for their relationship with respect to a natural lexicographic ordering. It should be noted that even when a pair of words belong to a group defined by non-trivial relations, only the free reductions of the words are compared. Thus, a pair of words belonging to a group  $G$  may be declared to be distinct even though they may represent the same element of  $G$ .

The words of an fp-group  $G$  are ordered first by length and then lexicographically. The lexicographic ordering is determined by the following ordering on the generators and their inverses:

$$G.1 < G.1^{-1} < G.2 < G.2^{-1} < \dots$$

Here,  $u$  and  $v$  are words belonging to some common fp-group.

`u eq v`

Return **true** if the free reductions of the words  $u$  and  $v$  are identical.

`u ne v`

Return **true** if the free reductions of the words  $u$  and  $v$  are not identical.

`u lt v`

Return **true** if the word  $u$  precedes the word  $v$ , with respect to the ordering defined above for elements of an fp-group. The words  $u$  and  $v$  are freely reduced before the comparison is made.

`u le v`

Return **true** if the word  $u$  either precedes, or is equal to, the word  $v$ , with respect to the ordering defined above for elements of an fp-group. The words  $u$  and  $v$  are freely reduced before the comparison is made.

`u ge v`

Return **true** if the word  $u$  either follows, or is equal to, the word  $v$ , with respect to the ordering defined above for elements of an fp-group. The words  $u$  and  $v$  are freely reduced before the comparison is made.

`u gt v`

Return **true** if the word  $u$  follows the word  $v$ , with respect to the ordering defined above for elements of an fp-group. The words  $u$  and  $v$  are freely reduced before the comparison is made.

**Example H70E3**

---

We construct the free group on three generators and generate a random word  $w$  of length between 4 and 6.

```
> F<a,b,c> := FreeGroup(3);
>
> w := Random(F, 4, 6);
> w;
b^-1 * a^-1 * b^2 * a^-1
```

We print the length of  $w$  and the weight (the exponent sum) of generator  $a$  in  $w$ .

```
> #w;
5
>
> Weight(w, a);
-2
```

We now strip the generators from  $w$  one by one, using arithmetic and comparison operators for words and the access function `LeadingGenerator`.

```
> while w ne Identity(F) do
>   g := LeadingGenerator(w);
>   print g;
>   w := g^-1 * w;
> end while;
b^-1
a^-1
b
b
a^-1
```

---

**70.2.6 Relations**

A *relation* is an equality between two words in a fp-group. To facilitate working with relations, a *relation type* is provided.

$w_1 = w_2$

Given words  $w_1$  and  $w_2$  over the generators of an fp-group  $G$ , create the relation  $w_1 = w_2$ . Note that this relation is not automatically added to the existing set of defining relations  $R$  for  $G$ . It may be added to  $R$ , for example, through use of the quo-constructor (see below).

$r[1]$

LHS( $r$ )

Given a relation  $r$  over the generators of  $G$ , return the left hand side of the relation  $r$ . The object returned is a word over the generators of  $G$ .

`r[2]``RHS(r)`

Given a relation  $r$  over the generators of  $G$ , return the right hand side of the relation  $r$ . The object returned is a word over the generators of  $G$ .

`r[1] := w`

Redefine the left hand side of the relation  $r$  to be the word  $w$ .

`r[2] := w`

Redefine the right hand side of the relation  $r$  to be the word  $w$ .

`f(r)`

Given a homomorphism of the group  $G$  for which  $r$  is a relation, return the image of  $r$  under  $f$ .

`Parent(r)`

Group over which the relation  $r$  is taken.

#### Example H70E4

---

We may define a group and a set of relations as follows:

```
> F<x, y> := FreeGroup(2);
> rels := { x^2 = y^3, (x*y)^4 = Id(F) } ;
```

To replace one side of a relation, the easiest way is to reassign the relation. So for example, to replace the relation  $x^2 = y^3$  by  $x^2 = y^4$ , we go:

```
> r := x^2 = y^3;
> r := LHS(r) = y^4;
```

---

### 70.3 Construction of an FP-Group

An fp-group is normally constructed either by giving a presentation in terms of generators and relations or it is defined to be a subgroup or quotient group of an existing fp-group. However, fp-groups may be also created from finite permutation and matrix groups. Finally, Magma has separate types for a number of families of fp-groups that satisfy a condition such as being polycyclic. Again functions are provided to convert a member of one of these families into a general fp-group.

#### 70.3.1 The Quotient Group Constructor

A group with non-trivial relations is constructed as a quotient of an existing group, usually a free group. For convenience, the necessary free group may be constructed in-line.

quo< F   R >
--------------

Given an fp-group  $F$ , and a set of relations  $R$  in the generators of  $F$ , construct the quotient  $G$  of  $F$  by the normal subgroup of  $F$  defined by  $R$ . The group  $G$  is defined by means of a presentation which consists of the relations for  $F$  (if any), together with the additional relations defined by the list  $R$ .

The expression defining  $F$  may be either simply the name of a previously constructed group, or an expression defining an fp-group.

If  $R$  is a list then each term of the list is either a *word*, a *relation*, a *relation list* or a *subgroup* of  $F$ .

A *word* is interpreted as a relator.

A *relation* consists of a pair of words, separated by '='. (See above.)

A *relation list* consists of a list of words, where each pair of adjacent words is separated by '=':  $w_1 = w_2 = \dots = w_r$ . This is interpreted as the set of relations  $w_1 = w_r, \dots, w_{r-1} = w_r$ . Note that the relation list construct is only meaningful in the context of the quo-constructor.

A *subgroup*  $H$  appearing in the list  $R$  contributes its generators to the relation set for  $G$ , i.e. each generator of  $H$  is interpreted as a relator for  $G$ .

The group  $F$  may be referred to by the special symbol \$ in any word appearing to the right of the '|' symbol in the quo-constructor. Also, in the context of the quo-constructor, the identity element (empty word) may be represented by the digit 1.

The function returns:

- (a) The quotient group  $G$ ;
- (b) The natural homomorphism  $\phi : F \rightarrow G$ .

This function may require the computation of a coset table. Experienced users can control the behaviour of a possibly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function `SetGlobalTCPParameters`. For a detailed description of the available parameters and their meanings, we refer to Chapter 71.

## G / H

Given a subgroup  $H$  of the group  $G$ , construct the quotient of  $G$  by the normal closure  $N$  of  $H$ . The quotient is formed by taking the presentation for  $G$  and including the generating words of  $H$  as additional relators.

**Example H70E5**

---

The symmetric group of degree 4 may be represented as a two generator group with presentation  $\langle a, b \mid a^2, b^3, (ab)^4 \rangle$ . Giving the relations as a list of relators, the presentation would be specified as:

```
> F<a, b> := FreeGroup(2);
> G<x, y>, phi := quo< F | a^2, b^3, (a*b)^4 >;
```

Alternatively, giving the relations as a relations list, the presentation would be specified as:

```
> F<a, b> := FreeGroup(2);
> G<x, y>, phi := quo< F | a^2 = b^3 = (a*b)^4 = 1 >;
```

Finally, giving the relations in the form of a set of relations, this presentation would be specified as:

```
> F<a, b> := FreeGroup(2);
> rels := { a^2, b^3, (a*b)^4 };
> G<x, y>, phi := quo< F | rels >;
```

**Example H70E6**

---

A group may be defined using the quo-constructor without first assigning a free group. The \$ symbol is used to reference the group whose quotient is being formed.

```
> S4<x, y> := quo< FreeGroup(2) | $.1^2, $.2^3, ($.1*$.2)^4 >;
> S4;
```

Finitely presented group S4 on 2 generators

Relations

```
x^2 = Id(S4)
y^3 = Id(S4)
(x * y)^4 = Id(S4)
```

**Example H70E7**

---

We illustrate the use of the quo-constructor in defining the quotient of a group other than a free group.

```
> F<x, y> := Group< x, y | x^2 = y^3 = (x*y)^7 = 1 >;
> F;
```

Finitely presented group F on 2 generators

Relations

```
x^2 = Id(F)
y^3 = Id(F)
```

```

      (x * y)^7 = Id(F)
> G<a, b> := quo< F | (x, y)^8 >;
> G;
Finitely presented group G on 2 generators
Relations
      a^2 = Id(G)
      b^3 = Id(G)
      (a * b)^7 = Id(G)
      (a, b)^8 = Id(G)
> Order(G);
10752

```

---

### 70.3.2 The FP-Group Constructor

For convenience, a constructor is provided which allows the user to define an fp-group in a single step.

Group< X   R >
----------------

Given a list  $X$  of variables  $x_1, \dots, x_r$ , and a list of relations  $R$  over these generators, first construct the free group  $F$  on the generators  $x_1, \dots, x_r$  and then construct the quotient of  $F$  corresponding to the normal subgroup of  $F$  defined by the relations  $R$ .

The syntax for the relations  $R$  is the same as for the quo-constructor. The function returns:

- (a) The quotient group  $G$ ;
- (b) The natural homomorphism  $\phi : F \rightarrow G$ .

#### Example H70E8

---

We illustrate the Group-constructor by defining the binary tetrahedral group in terms of the presentation  $\langle r, s \mid r^3 = s^3 = (rs)^2 \rangle$ :

```
> G<r, s> := Group< r, s | r^3 = s^3 = (r*s)^2 >;
```

#### Example H70E9

---

Again, using the Group-constructor, the group  $\langle r, s, t \mid r^2, s^2, t^2, rst = str = trs \rangle$  would be specified as:

```
> G<r, s, t> := Group<r, s, t | r^2, s^2, t^2, r*s*t = s*t*r = t*r*s>;
```

**Example H70E10**

---

In our final example we illustrate the use of functions to represent parametrised families of groups. In the notation of Coxeter, the symbol  $(l, m | n, k)$  denotes the family of groups having presentation

$$\langle a, b \mid a^l, b^m, (a * b)^n, (a * b^{-1})^k \rangle .$$

```
> Glmnk := func< l, m, n, k | Group< a, b | a^l, b^m, (a*b)^n, (a*b^-1)^k > >;
> G<a, b> := Glmnk(3, 3, 4, 4);
> G;
Finitely presented group G on 2 generators
Relations
  a^3 = Id(G)
  b^3 = Id(G)
  (a * b)^4 = Id(G)
  (a * b^-1)^4 = Id(G)
> Order(G);
168
> G<a, b> := Glmnk(2, 3, 4, 5);
> G;
Finitely presented group G on 2 generators
Relations
  a^2 = Id(G)
  b^3 = Id(G)
  (a * b)^4 = Id(G)
  (a * b^-1)^5 = Id(G)
> Order(G);
1
```

Thus  $(2, 3 \mid 4, 5)$  is the trivial group.

---

**70.3.3 Construction from a Finite Permutation or Matrix Group****FPGroup(G)**

Given a finite group  $G$  in category GrpPerm or GrpMat, this function returns a finitely presented group  $F$ , isomorphic to  $G$ , together with the isomorphism  $\phi : F \rightarrow G$ . The generators of  $F$  correspond to the generators of  $G$ , so this function can be used to obtain a set of defining relations for the given generating set of  $G$ .

It should be noted that this function is only practical for groups of order at most a few million. In the case of much larger permutation groups, an isomorphic fp-group can be constructed using the function FPGroupStrong.

**FPGroupStrong(G)**

Given a finite group  $G$  in category `GrpPerm` or `GrpMat`, this function returns a finitely presented group  $F$ , isomorphic to  $G$ , together with the isomorphism  $\phi : F \rightarrow G$ . The generators of  $F$  correspond to a set of strong generators of  $G$ . If no strong generating set is known for  $G$ , one will be constructed.

For a detailed description of this function, in particular for a list of available parameters, we refer to Chapter 58 and Chapter 59, respectively.

**FPGroupStrong(G, N)**

Given a permutation group  $G$  and a normal subgroup  $N$  of  $G$ , this function returns a finitely presented group  $F$ , isomorphic to  $G/N$ , together with a homomorphism  $\phi : G \rightarrow F$ .

For a detailed description of this function, we refer to Chapter 58.

**Example H70E11**

We start with defining the alternating group  $G \simeq A_5$  as a permutation group.

```
> G := AlternatingGroup(5);
> G;
Permutation group G acting on a set of cardinality 5
Order = 60 = 2^2 * 3 * 5
(3, 4, 5)
(1, 2, 3)
```

Now we create an fp-group  $F$  isomorphic to  $G$ , using the function `FPGroup`. The presentation is constructed by computing a set of defining relations for the generators of  $G$ , i.e. the generators of the returned fp-group correspond to the generators of  $G$ . This defines a homomorphism from  $F$  to  $G$ , which the function `FPGroup` returns as second return value.

```
> F<x,y>, f := FPGroup(G);
> F;
Finitely presented group F on 2 generators
Relations
  x^3 = Id(F)
  y^3 = Id(F)
  (x^-1 * y * x * y)^2 = Id(F)
  (x * y^-1 * x * y)^2 = Id(F)
> f;
Mapping from: GrpFP: F to GrpPerm: G
> f(x);
(3, 4, 5)
> f(y);
(1, 2, 3)
```

Using the function `FPGroupStrong`, we now create another fp-group  $F_s$ , isomorphic to  $G$ , whose generators correspond to a set of strong generators of  $G$ .

```
> Fs<[z]>, fs := FPGroupStrong(G);
```

```

> Fs;
Finitely presented group Fs on 3 generators
Relations
  z[1]^3 = Id(Fs)
  (z[1]^-1 * z[3]^-1)^2 = Id(Fs)
  z[3]^3 = Id(Fs)
  z[1]^-1 * z[2]^-1 * z[1] * z[3]^-1 * z[2] = Id(Fs)
  z[2]^3 = Id(Fs)
  (z[3]^-1 * z[2]^-1)^2 = Id(Fs)
> fs;
Mapping from: GrpFP: Fs to GrpPerm: G
Applying the isomorphism fs, we have a look at the strong generating set constructed for G.
> [ fs(z[i]) : i in [1..#z] ];
[
  (3, 4, 5),
  (1, 2, 3),
  (2, 3, 4)
]

```

### 70.3.4 Construction of the Standard Presentation for a Coxeter Group

There is a special MAGMA category `GrpPermCox`, a subcategory of `GrpPerm`, for finite Coxeter groups. Here, we describe a function to create from a Coxeter group  $W$  a finitely presented group  $F$ , isomorphic to  $W$ , which is given by the standard Coxeter group presentation. We refer to Chapter 98.

**CoxeterGroup(GrpFP, W)**

Given a finite Coxeter group  $W$  in the category `GrpFPCox` or `GrpPermCox`, construct a finitely presented group  $F$  isomorphic to  $W$ , given by a standard Coxeter presentation. The isomorphism from  $W$  to  $F$  is returned as second return value. The first argument to this function must be the category `GrpFP`.

**Local**

**BOOLELT**

**Default : false**

If the parameter `Local` is set to `true`,  $F$  is the appropriate subgroup of the FP version of the overgroup of  $W$ .

#### Example H70E12

We construct a Coxeter group  $W$  of Cartan type  $C5$  and create an isomorphic fp-group  $F$ . We can use the isomorphism from  $W$  to  $F$  to map words in the generators of  $F$  to permutation group elements and vice versa.

```

> W := CoxeterGroup("C5");
> F<[s]>, h := CoxeterGroup(GrpFP, W);
> F;

```

Finitely presented group F on 5 generators

Relations

```

s[3]^2 = Id(F)
s[2] * s[4] = s[4] * s[2]
s[1]^2 = Id(F)
s[1] * s[2] * s[1] = s[2] * s[1] * s[2]
s[2] * s[5] = s[5] * s[2]
s[4]^2 = Id(F)
s[1] * s[3] = s[3] * s[1]
s[3] * s[4] * s[3] = s[4] * s[3] * s[4]
s[2]^2 = Id(F)
s[1] * s[4] = s[4] * s[1]
s[5]^2 = Id(F)
(s[4] * s[5])^2 = (s[5] * s[4])^2
s[3] * s[5] = s[5] * s[3]
s[1] * s[5] = s[5] * s[1]
s[2] * s[3] * s[2] = s[3] * s[2] * s[3]

```

> h;

Mapping from: GrpCox: W to GrpFP: F given by a rule

> h(W.1\*W.2);

s[1] \* s[2]

> (s[1]\*s[2]\*s[3]\*s[4]) @@ h;

```

(1, 39, 4, 3, 2)(5, 13, 19, 23, 25)(6, 36, 35, 8, 7)(9, 16, 21,
  24, 17)(10, 33, 32, 31, 11)(12, 18, 22, 15, 20)(14, 29, 28,
  27, 26)(30, 38, 44, 48, 50)(34, 41, 46, 49, 42)(37, 43, 47,
  40, 45)

```

### 70.3.5 Conversion from a Special Form of FP-Group

Groups that satisfy certain properties, such as being abelian or polycyclic, are known to possess presentations with respect to which the word problem is soluble. Specialised categories have been constructed in Magma for several of these, e.g. the categories **GrpGPC**, **GrpPC** and **GrpAb**. The functions described in this section allow a group created in one of the special presentation categories to be recast as an fp-group.

<b>FPGroup(G)</b>
-------------------

Given a group  $G$ , defined either by a polycyclic group presentation (types **GrpPC** and **GrpGPC**) or an abelian group presentation (type **GrpAb**), return a group  $H$  isomorphic to  $G$ , together with the isomorphism  $\phi : G \rightarrow H$ . The generators for  $H$  will correspond to the generators of  $G$ . The effect of this function is to convert a presentation in a special form into a general fp-group presentation.

**Example H70E13**

---

We illustrate the cast from special forms of fp-groups to the category **GrpFP** by converting a polycyclic group.

```
> G := DihedralGroup(GrpGPC, 0);
> G;
GrpGPC : G of infinite order on 2 PC-generators
PC-Relations:
  G.1^2 = Id(G),
  G.2^G.1 = G.-2
> F := FPGroup(G);
> F;
Finitely presented group F on 2 generators
Relations
  F.1^2 = Id(F)
  F.2^F.1 = F.2^-1
  F.1^-1 * F.2^-1 * F.1 = F.2
```

---

**70.3.6 Construction of a Standard Group**

A number of functions are provided which construct presentations for various standard groups.

AbelianGroup(GrpFP, [n<sub>1</sub>, ..., n<sub>r</sub>])

Construct the abelian group defined by the sequence  $[n_1, \dots, n_r]$  of non-negative integers as an fp-group. The function returns the direct product of cyclic groups  $C_{n_1} \times C_{n_2} \times \dots \times C_{n_r}$ , where  $C_0$  is interpreted as an infinite cyclic group.

AlternatingGroup(GrpFP, n)

Alt(GrpFP, n)

Construct the alternating group of degree  $n$  as an fp-group, where the generators correspond to the permutations  $(3, 4, \dots, n)$  and  $(1, 2, 3)$ , for  $n$  odd, or  $(1, 2)(3, 4, \dots, n)$  and  $(1, 2, 3)$ , for  $n$  even.

BraidGroup(GrpFP, n)

Construct the braid group on  $n$  strings ( $n - 1$  Artin generators) as an fp-group.

CoxeterGroup(GrpFP, t)

Construct the Coxeter group of Cartan type  $t$  as a finitely presented group, given by the standard Coxeter presentation. The Cartan type  $t$  is passed to this function as a string; we refer to Chapter 97 for details.



Finitely presented group  $F$  on 4 generators

Relations

$$(F.2 * F.3)^2 = (F.3 * F.2)^2$$

$$F.1^2 = \text{Id}(F)$$

$$F.1 * F.3 = F.3 * F.1$$

$$F.2 * F.4 = F.4 * F.2$$

$$F.1 * F.2 * F.1 = F.2 * F.1 * F.2$$

$$F.2^2 = \text{Id}(F)$$

$$F.3^2 = \text{Id}(F)$$

$$F.3 * F.4 * F.3 = F.4 * F.3 * F.4$$

$$F.4^2 = \text{Id}(F)$$

$$F.1 * F.4 = F.4 * F.1$$

### 70.3.7 Construction of Extensions

Darstellungsgruppe( $G$ )

Given an fp-group  $G$ , construct a maximal central extension  $\tilde{G}$  of  $G$ . The group  $\tilde{G}$  is created as an fp-group.

DirectProduct( $G, H$ )

Given two fp-groups  $G$  and  $H$ , construct the direct product of  $G$  and  $H$ .

DirectProduct( $Q$ )

Given a sequence  $Q$  of  $r$  fp-groups, construct the direct product  $Q[1] \times \dots \times Q[r]$ .

FreeProduct( $G, H$ )

Given two fp-groups  $G$  and  $H$ , construct the free product of  $G$  and  $H$ .

FreeProduct( $Q$ )

Given a sequence  $Q$  of  $r$  fp-groups, construct the free product of the groups  $Q[1], \dots, Q[r]$ .

**Example H70E15**

---

We construct a maximal central extension of the following group of order 36.

```
> G<x1, x2> := Group<x1, x2 | x1^4, (x1*x2^-1)^2, x2^4, (x1*x2)^3>;
> G;
```

Finitely presented group G on 2 generators

Relations

```
x1^4 = Id(G)
(x1 * x2^-1)^2 = Id(G)
x2^4 = Id(G)
(x1 * x2)^3 = Id(G)
```

```
> D := Darstellungsgruppe(G);
```

```
> D;
```

Finitely presented group D on 4 generators

Relations

```
D.1^4 * D.3^-1 * D.4^2 = Id(D)
D.1 * D.2^-1 * D.1 * D.2^-1 * D.4 = Id(D)
D.2^4 = Id(D)
D.1 * D.2 * D.1 * D.2 * D.1 * D.2 * D.4 = Id(D)
(D.1, D.3) = Id(D)
(D.2, D.3) = Id(D)
(D.1, D.4) = Id(D)
(D.2, D.4) = Id(D)
(D.3, D.4) = Id(D)
```

```
> Index(D, sub< D | >);
```

```
108
```

Thus, a maximal central extension of  $G$  has order 108

**Example H70E16**

---

We create the direct product of the alternating group of degree 5 and the cyclic group of order 2.

```
> A5 := Group<a, b | a^2, b^3, (a*b)^5 >;
```

```
> Z2 := quo< FreeGroup(1) | $.1^2 >;
```

```
> G := DirectProduct(A5, Z2);
```

```
> G;
```

Finitely presented group G on 3 generators

Relations

```
G.1^2 = Id(G)
G.2^3 = Id(G)
(G.1 * G.2)^5 = Id(G)
G.3^2 = Id(G)
G.1 * G.3 = G.3 * G.1
G.2 * G.3 = G.3 * G.2
```

---

### 70.3.8 Accessing the Defining Generators and Relations

The functions in this group provide access to basic information stored for a finitely-presented group  $G$ .

`G . i`

The  $i$ -th defining generator for  $G$ . A negative subscript indicates that the inverse of the generator is to be created.  $G.0$  is `Identity(G)`, the empty word in  $G$ .

`Generators(G)`

A set containing the generators for the group  $G$ .

`NumberOfGenerators(G)`

`Ngens(G)`

The number of generators for the group  $G$ .

`PresentationLength(G)`

The total length of the relators for  $G$ .

`Relations(G)`

A sequence containing the defining relations for  $G$ .

## 70.4 Homomorphisms

For a general description of homomorphisms, we refer to Chapter 16. This section describes some special aspects of homomorphisms the domain of which is a finitely presented group.

### 70.4.1 General Remarks

The kernel of a homomorphism with a domain of type `GrpFP` can be computed using the function `Kernel`, if the codomain is of one of the types `GrpGPC`, `GrpPC` (cf. Chapter 63), `GrpAb` (cf. Chapter 69), `GrpPerm` (cf. Chapter 58), `GrpMat` (cf. Chapter 59), `ModAlg` or `ModGrp` (cf. Chapter 89), if the image is finite and its order sufficiently small. In this case, a regular permutation representation of the image is constructed and the kernel is created as a subgroup of the domain, defined by a coset table.

The kernel may also be computable, if the codomain is of the type `GrpFP`, the image is sufficiently small and a presentation for the image is known.

If the kernel of a map can be computed successfully, forming preimages of substructures is possible. An attempt to compute the kernel of a map will be made automatically, if the preimage of a substructure of the codomain is to be computed.

Note, that trying to compute the kernel may be very time and memory consuming; use this feature with care.

### 70.4.2 Construction of Homomorphisms

`hom< P -> G | S >`

Returns the homomorphism from the fp-group  $P$  to the group  $G$  defined by the assignment  $S$ .  $S$  can be the one of the following:

- (i) A list, sequence or indexed set containing the images of the  $n$  generators  $P.1, \dots, P.n$  of  $P$ . Here, the  $i$ -th element of  $S$  is interpreted as the image of  $P.i$ , i.e. the order of the elements in  $S$  is important.
- (ii) A list, sequence, enumerated set or indexed set, containing  $n$  tuples  $\langle x_i, y_i \rangle$  or arrow pairs  $x_i \rightarrow y_i$ , where  $x_i$  is a generator of  $P$  and  $y_i \in G$  ( $i = 1, \dots, n$ ) and the set  $\{x_1, \dots, x_n\}$  is the full set of generators of  $P$ . In this case,  $y_i$  is assigned as the image of  $x_i$ , hence the order of the elements in  $S$  is not important.

Note, that it is currently not possible to define a homomorphism by assigning images to the elements of an arbitrary generating set of  $P$ . It is the user's responsibility to ensure that the arguments passed to the `hom`-constructor actually yield a well-defined homomorphism. For certain codomain categories, this may be checked using the function `IsSatisfied` described below.

`IsSatisfied(U, E)`

$U$  is a set or sequence of either words belonging to an  $n$ -generator fp-group  $H$  or relations over  $H$ .  $E$  is a sequence of  $n$  elements  $[e_1, \dots, e_n]$  belonging to a group  $G$  for which both, multiplication and comparison of elements are possible. Using the mapping  $H.i \rightarrow e_i$  ( $i = 1, \dots, n$ ), we evaluate the relations given by  $U$ . If  $U$  is a set or sequence of relations, the left and right hand sides of each relation are evaluated and compared for equality. Otherwise, each word in  $U$  is evaluated and compared to the identity. If all relations are satisfied, `IsSatisfied` returns the Boolean value `true`. On the other hand, if any relation is not satisfied, `IsSatisfied` returns the value `false`.

This function may be used to verify the correctness of the definition of a homomorphism from an fp-group to a group in a category for which both, multiplication and comparison of elements are possible.

### 70.4.3 Accessing Homomorphisms

`w @ f`

`f(w)`

Given a homomorphism whose domain is an fp-group  $G$  and an element  $w$  of  $G$ , return the image of  $w$  under  $f$  as an element of the codomain of  $f$ .

`H @ f`

`f(H)`

Given a homomorphism whose domain is an fp-group  $G$  and a subgroup  $H$  of  $G$ , return the image of  $H$  under  $f$  as a subgroup of the codomain of  $f$ .

Some maps do not support images of subgroups.

`g @@ f`

Given a homomorphism whose domain is an fp-group  $G$  and an element  $g$  of the image of  $f$ , return the preimage of  $g$  under  $f$  as an element of  $G$ .

Some maps do not support inverse images.

`H @@ f`

Given a homomorphism whose domain is an fp-group  $G$  and a subgroup  $H$  of the image of  $f$ , return the preimage of  $H$  under  $f$  as a subgroup of  $G$ .

Some maps do not support inverse images. The inverse image of a subgroup of the codomain can only be computed if the kernel of the homomorphism can be computed, i.e. if the kernel has moderate index in the domain.

`Domain(f)`

The domain of the homomorphism  $f$ .

`Codomain(f)`

The codomain of the homomorphism  $f$ .

`Image(f)`

The image or range of the homomorphism  $f$  as a subgroup of the codomain of  $f$ .

Some maps do not support this function.

`Kernel(f)`

The kernel of the homomorphism  $f$  as a (normal) subgroup of the domain of  $f$ , represented by a coset table.

Some maps do not support this function. The kernel of a homomorphism can only be computed, if it has moderate index in the domain.

### Example H70E17

---

For arbitrary  $n > 0$ , the symmetric group of degree  $n + 1$  is an epimorphic image of the braid group on  $n$  generators. In this example, we exhibit this relationship for  $n = 4$ .

We start with creating the braid group  $B$  on 5 strings, i.e. 4 Artin generators.

```
> B := BraidGroup(GrpFP, 5);
```

```
> B;
```

```
Finitely presented group B on 4 generators
```

```
Relations
```

```
B.1 * B.2 * B.1 = B.2 * B.1 * B.2
```

```
B.1 * B.3 = B.3 * B.1
```

```
B.1 * B.4 = B.4 * B.1
```

```
B.2 * B.3 * B.2 = B.3 * B.2 * B.3
```

```
B.2 * B.4 = B.4 * B.2
```

$$B.3 * B.4 * B.3 = B.4 * B.3 * B.4$$

In the symmetric group of degree 5, we define 4 transpositions which will be the images of the generators of  $B$ .

```
> S := SymmetricGroup(5);
> imgs := [ S!(1,2), S!(2,3), S!(3,4), S!(4,5) ];
```

In order to verify that this assignment actually gives rise to a well defined homomorphism, we check whether the potential images satisfy the defining relations of  $B$ .

```
> rels := Relations(B);
> rels;
[ B.1 * B.2 * B.1 = B.2 * B.1 * B.2, B.1 * B.3 = B.3 * B.1,
  B.1 * B.4 = B.4 * B.1, B.2 * B.3 * B.2 = B.3 * B.2 * B.3,
  B.2 * B.4 = B.4 * B.2, B.3 * B.4 * B.3 = B.4 * B.3 * B.4 ]
> IsSatisfied(rels, imgs);
true
```

They do. So we can define the homomorphism from  $B$  to  $S$ .

```
> f := hom< B->S | imgs >;
```

We see that  $f$  is surjective, i.e.  $S$  is an epimorphic image of  $B$  as claimed above.

```
> f(B) eq S;
true
```

We now check the kernel of  $f$ .

```
> Kernel(f);
Finitely presented group
Index in group B is 120 = 2^3 * 3 * 5
Subgroup of group B defined by coset table
```

Using the function `GeneratingWords` described later, we can obtain a set of generators of  $\ker(f)$  as a subgroup of  $B$ .

```
> GeneratingWords(B, Kernel(f));
{ B.2^-2, (B.1 * B.2 * B.3^-1 * B.2^-1 * B.1^-1)^2, B.1^-2,
  (B.3 * B.4^-1 * B.3^-1)^2, (B.2 * B.3^-1 * B.2^-1)^2,
  (B.2 * B.3 * B.4^-1 * B.3^-1 * B.2^-1)^2, B.4^-2,
  (B.1 * B.2^-1 * B.1^-1)^2, B.3^-2,
  (B.1 * B.2 * B.3 * B.4^-1 * B.3^-1 * B.2^-1 * B.1^-1)^2 }
```

It is easy to see that all generators of  $\ker(f)$  are conjugates of words of the form  $g^{\pm 2}$ , where  $g$  is a generator of  $B$ . We check this, using the normal closure constructor `ncl` described later.

```
> Kernel(f) eq ncl< B | B.1^2, B.2^2, B.3^2, B.4^2 >;
true
```

Thus, the braid relations together with the relations  $B.1^2, B.2^2, B.3^2, B.4^2$  are a set of defining relations for  $S$ .

---



groups with many conjugate classes, this parameter can be set to **false** in order to reduce memory requirements at the expense of increased computing time.

---

**Example H70E18**


---

Consider the finitely presented group

$$F := \langle a, b, c \mid ac^{-1}bc^{-1}aba^{-1}b, abab^{-1}c^2b^{-1}, a^2b^{-1}(ca)^4cb^{-1} \rangle.$$

```
> F := Group< a,b,c | a*c^-1*b*c^-1*a*b*a^-1*b,
>               a*b*a*b^-1*c^2*b^-1,
>               a^2*b^-1*c*a*c*a*c*a*c*a*c*b^-1 >;
```

We use the function `Homomorphisms` to prove that  $F$  maps onto  $A_5$ .

```
> G := Alt(5);
> homs := Homomorphisms(F, G : Limit := 1);
> #homs gt 0;
true
```

---

Homomorphisms(F, G, A : <i>parameters</i> )
---

Homomorphisms(F, G : <i>parameters</i> )
--

Given a finitely presented group  $F$  and two finite polycyclic groups  $G$  and  $A$  with  $G \triangleleft A$ , return a sequence containing representatives of the classes of homomorphisms from  $F$  to  $G$  modulo automorphisms of  $G$  induced by elements of  $A$ . (That is, two homomorphisms  $f_1, f_2 : F \rightarrow G$  are considered equivalent if there exists an element  $a \in A$  such that  $f_1(x) = f_2(x)^a$  for all  $x \in F$ .) The call `Homomorphisms(F, G)` is equivalent to `Homomorphisms(F, G, G)`.

The following parameters are available for this function.

<b>Surjective</b>	BOOLELT	<i>Default</i> : true
-------------------	---------	-----------------------

If this parameter is set to **true** (default), only epimorphisms are considered.

<b>Limit</b>	RNGINTELT	<i>Default</i> : 0 (no limit)
--------------	-----------	-------------------------------

If this parameter is set to  $n$ , the function terminates after  $n$  classes of homomorphisms satisfying the specified conditions have been found. A value of 0 (default) means no limit.

### 70.4.4.1 Computing Homomorphisms to Permutation Groups Interactively

A process version of the algorithm used by `Homomorphisms` is available for computing homomorphisms one at a time. The functions relevant for this interactive version are described in this section.

<code>HomomorphismsProcess(F, G, A : parameters)</code>
---

<code>HomomorphismsProcess(F, G : parameters)</code>
--

Given a finitely presented group  $F$  and two permutation groups  $G$  and  $A$  with  $G \triangleleft A$ , return a process  $P$  for computing representatives of the classes of homomorphisms from  $F$  to  $G$  modulo automorphisms of  $G$  induced by elements of  $A$ . (That is, two homomorphisms  $f_1, f_2 : F \rightarrow G$  are considered equivalent if there exists an element  $a \in A$  such that  $f_1(x) = f_2(x)^a$  for all  $x \in F$ .) `HomomorphismsProcess(F, G)` is equivalent to `HomomorphismsProcess(F, G, G)`.

After constructing the process, the search for homomorphisms is started and runs until the first homomorphism is found, the time limit is reached (in which case  $P$  is marked as *invalid*) or the search is completed without finding a homomorphism (in which case  $P$  is marked as *empty*).

The parameters have the same meaning as for the function `Homomorphisms`.

<code>Surjective</code>	BOOLELT	<i>Default : true</i>
<code>Limit</code>	RNGINTELT	<i>Default : 0 (no limit)</i>
<code>TimeLimit</code>	RNGINTELT	<i>Default : 0 (no limit)</i>
<code>CosetEnumeration</code>	BOOLELT	<i>Default : true</i>
<code>CacheCosetAction</code>	BOOLELT	<i>Default : true</i>

Setting a time limit for a process  $P$  limits the total amount of time spent in the backtrack search for homomorphisms during the construction of  $P$  and in subsequent calls to `NextElement` and `Complete`. The time spent in initial coset enumerations is *not* counted towards this limit.

If the time limit or the limit on the number of homomorphisms set for a process  $P$  is reached,  $P$  becomes *invalid*. Calling `NextElement` or `Complete` for an invalid process or for a process which is *empty*, that is, which has found all possible classes of homomorphisms, will cause a runtime error. The functions `IsValid` and `IsEmpty` can be used to check whether a process is valid or empty, respectively. The use of these functions is recommended to avoid runtime errors in loops or user written functions.

<code>NextElement(<math>\sim P</math>)</code>
---

Given a valid and non-empty process  $P$ , continue the backtrack search until a new class of homomorphisms is found. If the search completes without a new representative being found,  $P$  is marked as *empty*. If a limit set for  $P$  is reached,  $P$  is marked as *invalid*.

**Complete( $\sim P$ )**

Given a valid and non-empty process  $P$ , continue the search for homomorphisms until all classes of homomorphisms have been found or a limit set for  $P$  is reached. If the search for homomorphisms completes,  $P$  is marked as empty. If a limit set for  $P$  is reached,  $P$  is marked as invalid.

**IsEmpty( $P$ )**

Returns whether  $P$  is empty, that is, whether all possible classes of homomorphisms have been found.

**IsValid( $P$ )**

Returns **false** if a limit set for  $P$  has been reached and **true** otherwise. Note that the return value of **IsValid** is not related to whether or not  $P$  currently defines a homomorphism.

**DefinesHomomorphism( $P$ )**

Returns whether  $P$  currently defines a homomorphism which can be extracted using the function **Homomorphism**.

**Homomorphism( $P$ )**

Given a process  $P$  which defines a homomorphism, return this homomorphism, that is, the homomorphism most recently found by  $P$ . If  $P$  does not define a homomorphism, a runtime error will result. The function **DefinesHomomorphism** can be used to test whether a call to **Homomorphism** is legal for a process.

**# $P$** 

Return the number of homomorphisms that have been found by the process  $P$ .

**Homomorphisms( $P$ )**

Return a sequence containing all homomorphisms that have been found by the process  $P$ . Note that the sequence will only contain a complete set of representatives of the classes of homomorphisms if  $P$  is empty, that is, if the backtrack search for  $P$  has been completed.

**Example H70E19**

---

Consider the braid group  $B$  on 4 strings. We show how the interactive computation of homomorphisms can be used to determine a homomorphism  $f : B \rightarrow \text{PSL}(2, 16)$  whose image is a maximal subgroup of  $\text{PSL}(2, 16)$ .

Note how the functions **IsValid**, **IsEmpty** and **DefinesHomomorphism** are used to avoid runtime errors in the **while** loop.

```
> B := BraidGroup(GrpFP, 4);
> G := PSL(2,16);
> P := HomomorphismsProcess(B, G : Surjective := false,
```

```

>                                     TimeLimit := 10);
> while not IsEmpty(P) do
>   if DefinesHomomorphism(P) then
>     f := Homomorphism(P);
>     img := Image(f);
>     if IsMaximal(G,img) then
>       print "found image which is maximal subgroup";
>       break;
>     end if;
>   end if;
>   if IsValid(P) then
>     NextElement(~P);
>   else
>     print "Limit has been reached";
>     break;
>   end if;
> end while;
found image which is maximal subgroup
>
> f;
Homomorphism of GrpFP: B into GrpPerm: G, induced by
  B.1 |--> (1, 4, 3, 6, 12)(2, 11, 9, 16, 7)(5, 17, 8, 15, 13)
  B.2 |--> (1, 4, 2, 10, 16)(3, 11, 9, 12, 14)(5, 15, 8, 13, 17)
  B.3 |--> (1, 4, 3, 6, 12)(2, 11, 9, 16, 7)(5, 17, 8, 15, 13)

```

### Example H70E20

---

This example sketches how the interactive version of the homomorphism algorithm could be used as part of a function trying to prove that a group is infinite.

Note again how the functions `IsValid`, `IsEmpty` and `DefinesHomomorphism` are used to avoid runtime errors.

```

> function MyIsInfinite(F)
>
> // ...
>
> // quotient approach: check whether an obviously infinite
> //   normal subgroup can be found in reasonable time.
> S := [ Alt(5), PSL(2,7), PSL(2,9), PSL(2,11) ];
> for G in S do
>   P := HomomorphismsProcess(F, G : Surjective := false,
>                               TimeLimit := 5);
>   while IsValid(P) and not IsEmpty(P) do
>     if DefinesHomomorphism(P) then
>       f := Homomorphism(P);
>       if 0 in AQInvariants(Kernel(f)) then
>         print "found infinite normal subgroup";

```

```

>         print "Hence group is infinite";
>         return true;
>     end if;
> end if;
>     if IsValid(P) then
>         NextElement(~P);
>     end if;
> end while;
> end for;
> print "quotient approach failed; trying other strategies";
>
> // ...
>
> end function;

```

We try the code fragment on the group

$$\langle a, b \mid ab^{-1}a^{-1}ba^{-1}b^{-1}abb, ab^{-1}a^{-1}baaba^{-1}b^{-1}ab^{-1}a^{-1}baba^{-1}b^{-1} \rangle .$$

```

> F := Group< a,b |
>     a*b^-1*a^-1*b*a^-1*b^-1*a*b*b,
>     a*b^-1*a^-1*b*a*a*b*a^-1*b^-1*a*b^-1*a^-1*b*a*b*a^-1*b^-1 >;
> MyIsInfinite(F);
found infinite normal subgroup
Hence group is infinite
true

```

#### 70.4.4.2 Finding Homomorphisms onto Simple Groups

We describe utilities for finding homomorphisms onto simple groups. As in the previous example, this may be useful when the presentation defines a perfect group. The methods used are similar to the example, with a list of simple groups to try, and using the function `Homomorphisms`.

The list of simple groups supplied in V2.10 contains all non-abelian simple groups with order  $\leq 10^9$ . Such a list is dominated by  $PSL(2, q)$ 's with  $q$  odd. In this implementation these  $PSL(2, q)$ 's are treated as an infinite family rather than stored individually, and so continue beyond the above limit.

```
SimpleQuotients(F, deg1, deg2, ord1, ord2: parameters)
```

```
SimpleQuotients(F, ord1, ord2: parameters)
```

```
SimpleQuotients(F, ord2: parameters)
```

Family	ANY	Default : "All"
Limit	RNGINTELT	Default : 1
HomLimit	RNGINTELT	Default : 0

Uses `Homomorphisms` to find epimorphisms from  $F$  onto simple groups in a fixed list. The arguments  $deg1$  and  $deg2$  are respectively lower and upper bounds for the degree of the image group. If the degree arguments are not present then bounds of 5 and  $10^7$  are used. The arguments  $ord1$  and  $ord2$  are respectively lower and upper bounds for the orders of the image group. (Setting  $ord2$  low enough is particularly important if a quick search is wanted.) If  $ord1$  is not given then it defaults to 1.

The return value is a list of sequences of epimorphisms found. Each sequence contains epimorphisms onto one simple group. The parameter `Limit` limits the number of successful searches to be carried out by `Homomorphisms`. The default value is 1, so by default the search terminates with the first simple group found to be a homomorphic image of  $F$ .

The parameter `HomLimit` limits the number of homomorphisms that will be searched for by any particular call to `Homomorphisms`. It defaults to zero, so that all homomorphisms for any group found will be returned.

The parameter `Family` selects sublists of the main list to search. Possible values of this parameter are "All", "PSL", "PSL2", "Mathieu", "Alt", "PSp", "PSU", "Other", and "notPSL2"; sets of these strings are also allowed, which searches on the union of the appropriate sublists.

`SimpleQuotientProcess(F, deg1, deg2, ord1, ord2: parameters)`

Family

ANY

Default : "All"

Produce a record that defines a process for searching for simple quotients of  $F$  as `SimpleQuotients` does. Calling this function sets up the record and conducts the initial search until a quotient is found. Continuing the search for another quotient is done by calling `NextSimpleQuotient`. Extracting the epimorphisms found is achieved using `SimpleEpimorphisms`, and testing if the process has expired is the task of `IsEmptySimpleQuotientProcess`.

`NextSimpleQuotient(~P)`

When  $P$  is a record returned by `SimpleQuotientProcess`, advance the search to the next simple group which is a homomorphic image of the finitely presented group. Does nothing if the process has expired.

`IsEmptySimpleQuotientProcess(P)`

When  $P$  is a record returned by `SimpleQuotientProcess`, test whether or not  $P$  has expired.

`SimpleEpimorphisms(P)`

When  $P$  is a record returned by `SimpleQuotientProcess`, extract the most recently found epimorphisms onto a simple group, plus a tuple describing the image group. This is a valid operation when `IsEmptySimpleQuotientProcess` returns `false`.

**Example H70E21**

---

We take a perfect finitely presented group and search for simple quotients.

```
> F := Group<a,b,c|a^13,b^3,c^2,a = b*c>;
> IsPerfect(F);
true
> L := SimpleQuotients(F,1, 100, 2, 10^5:Limit := 2);
> #L;
2
> for x in L do CompositionFactors(Image(x[1])); end for;
  G
  | A(1, 13)           = L(2, 13)
  1
  G
  | A(2, 3)           = L(3, 3)
  1
> L[2,1];
Homomorphism of GrpFP: F into GrpPerm: $, Degree 13, Order
2^4 * 3^3 * 13 induced by
F.1 |--> (1, 10, 4, 5, 11, 8, 3, 6, 7, 12, 9, 13, 2)
F.2 |--> (2, 10, 4)(3, 6, 7)(5, 11, 13)(8, 12, 9)
F.3 |--> (1, 10)(2, 5)(3, 12)(8, 13)
> #L[2];
2
```

We've found  $L(2,13)$  and  $L(3,3)$  as images, with 2 inequivalent homomorphisms onto the second. We'll try a process looking through a smaller family.

```
> P := SimpleQuotientProcess(F,1, 100, 2, 10^6:Family:="PSU");
> IsEmptySimpleQuotientProcess(P);
false
> eps, info := SimpleEpimorphisms(P);
> info;
<65, 62400, PSU(3, 4)>
```

We've found  $PSU(3,4)$  of order 62400 and degree 65 as an image. We continue with this process.

```
> NextSimpleQuotient(~P);
> IsEmptySimpleQuotientProcess(P);
true
```

No, there are no more within the limits given.

---

### 70.4.5 The $L_2$ -Quotient Algorithm

Given a finitely presented group  $G$  on two generators, the  $L_2$ -quotient algorithm of Plesken and Fabianska [PF09] computes all quotients of  $G$  which are isomorphic to some  $\mathrm{PSL}(2, q) = L_2(q)$ , simultaneously for any prime power  $q$ . It can handle the case of infinitely many quotients, and also works for very large prime powers.

Note that, at the moment, the algorithm does not return images onto the groups  $\mathrm{PSL}(2, 2)$ ,  $\mathrm{PSL}(2, 3)$ , and  $\mathrm{PSL}(2, 4) = \mathrm{PSL}(2, 5)$ .

#### `L2Quotients(G)`

This is the main method. It takes as parameter a finitely presented group on two generators, and returns a list of prime ideals of  $Z[x_1, x_2, x_{12}]$ . These prime ideals contain all information about the  $L_2$ -quotients.

#### `L2Type(P)`

For a prime ideal  $P$  in  $Z[x_1, x_2, x_{12}]$ , this method returns a string describing the  $L_2$ -type which the ideal encodes. The possible types are "reducible", "dihedral", "Alt(4)", "Sym(4)", "Alt(5)", "infinite (characteristic zero)", "infinite (characteristic p)", "PGL(2, q)", and "PSL(2, q)". Note that for prime ideals which are returned by `L2Quotients` or `L2Ideals`, only the last four types can occur; the other types are eliminated in these methods.

#### `L2Generators(P)`

Given a maximal ideal  $M$  in  $Z[x_1, x_2, x_{12}]$ , compute two matrices with entries in  $Z[x_1, x_2, x_{12}]/M$  corresponding to  $M$ .

#### `L2Ideals(I)`

Given an ideal  $I$  in  $Z[x_1, x_2, x_{12}]$ , compute the minimal associated primes of  $I$  which give rise to  $L_2$ -quotients. This is mainly used for ideals returned by `L2Quotients` which encode infinitely many  $L_2$  images.

### Example H70E22

---

The first few examples are taken from Conder, Havas and Newman [CHN11].

```
> Gamma< x, y > := Group< x, y | x^2, y^3 >;
> u := x*y; v := x*y^-1;
> G := quo< Gamma | u^10*v^2*u*v*u*v^2 >;
> quot := L2Quotients(G); quot;
[
Ideal of Polynomial ring of rank 3 over Integer Ring
Order: Lexicographical
Variables: x1, x2, x12
Inhomogeneous, Dimension 0
Groebner basis:
[
x1,
```

```

    x2 + 1,
    x12^2 + 4*x12 + 2,
    5
]
]

```

L2Quotients returns only one ideal, so there is only one  $L_2$  quotient. We can use L2Type to check what group this is.

```

> L2Type(quot[1]);
PSL( 2, 5^2 )

```

Finally, L2Generators returns matrices in  $SL(2, 25)$  which map onto generators in  $PSL(2, 25)$ .

```

> L2Generators(quot[1]);
MatrixGroup(2, GF(5^2))
Generators:
[ 3 $.1^21]
[ 0 2]
[ 0 4]
[ 1 4]

```

```

Mapping from: MatrixGroup(2, GF(5^2)) to GL(2, ext<GF(5) | Polynomial(GF(5),
\[3, 3, 1])>)

```

There are other quotients of the modular group with only finitely many  $L_2$  quotients:

```

> G := quo< Gamma | u^3*v*u^3*v*u^3*v^2*u*v^2 >;
> quot := L2Quotients(G);
> [L2Type(P) : P in quot];
[ PGL( 2, 13 ) ]
>
> G := quo< Gamma | u^3*v*u^3*v^2*u*v^3*u*v^2 >;
> quot := L2Quotients(G);
> [L2Type(P) : P in quot];
[]

```

This tells us that the first group has only one  $L_2$  quotient, isomorphic to  $PGL(2, 13)$ , and the second group has no  $L_2$  quotients at all.

### Example H70E23

---

Next, we look at Coxeter presentations of the form

$$(l, m|n, k) = \langle x, y|x^l, y^m, (xy)^n, (x^{-1}y)^k \rangle$$

and

$$(l, m, n; q) = \langle x, y|x^l, y^m, (xy)^n, [x, y]^k \rangle$$

for various values of  $l, m, n, k, q$ .

```

> G := Group< x,y | x^8, y^9, (x*y)^5, (x^-1*y)^7 >;
> [L2Type(P) : P in L2Quotients(G)];
[ PSL( 2, 71 ), PSL( 2, 71 ), PSL( 2, 71 ), PSL( 2, 2521 ), PSL( 2, 239 ) ,

```

```
PSL( 2, 449 ), PSL( 2, 3876207679 ), PSL( 2, 1009 ), PSL( 2, 29113631 ),
PSL( 2, 41 ), PSL( 2, 3056201 ) ]
```

The group  $\text{PSL}(2,71)$  occurs three times. This means that there are three essentially different epimorphisms of  $G$  onto  $\text{PSL}(2,71)$ , i.e., there is no automorphism of  $\text{PSL}(2,71)$  which transforms one epimorphism into another.

Here is another example, this time for the other family.

```
> G := Group< x, y | x^4, y^3, (x*y)^5, (x,y)^6 >;
> [L2Type(P) : P in L2Quotients(G)];
[ PSL( 2, 79 ), PSL( 2, 3^2 ), PSL( 2, 5^2 ) ]
```

There are three groups in the second family for which it is not known whether they are finite or infinite (Havas & Holt (2010) [HH10]). They are  $(3,4,9;2)$ ,  $(3,4,11;2)$ , and  $(3,5,6;2)$ . Each of them has only one  $L_2$  image.

```
> G := Group< x, y | x^3, y^4, (x*y)^9, (x,y)^2 >;
> [L2Type(P) : P in L2Quotients(G)];
[ PSL( 2, 89 ) ]
> G := Group< x, y | x^3, y^4, (x*y)^11, (x,y)^2 >;
> [L2Type(P) : P in L2Quotients(G)];
[ PSL( 2, 769 ) ]
> G := Group< x, y | x^3, y^5, (x*y)^6, (x,y)^2 >;
> [L2Type(P) : P in L2Quotients(G)];
[ PSL( 2, 61 ) ]
```

The algorithm currently only works for finitely presented groups on two generators. However, it is sometimes possible to reduce the number of generators for finitely presented groups on more than two generators.

The following examples are taken from Cavicchioli, O'Brien and Spaggiari [COS08], with code supplied by Eamonn O'Brien.

```
> CHR := function (X)
>   n := X[1]; m := X[2]; k := X[3];
>   F := FreeGroup (n);
>   R := [];
>   for i in [1..n] do
>     a := i + m;
>     if a gt n then repeat a := a - n; until a le n; end if;
>     b := i + k;
>     if b gt n then repeat b := b - n; until b le n; end if;
>     Append (~R, F.i * F.a = F.b);
>   end for;
>   Q := quo < F | R >;
>   return Q;
> end function;
> G := CHR([9, 1, 3]); G;
```

Finitely presented group  $G$  on 9 generators

Relations

$$G.1 * G.2 = G.4$$

$$G.2 * G.3 = G.5$$

```

G.3 * G.4 = G.6
G.4 * G.5 = G.7
G.5 * G.6 = G.8
G.6 * G.7 = G.9
G.7 * G.8 = G.1
G.8 * G.9 = G.2
G.9 * G.1 = G.3
> H := ReduceGenerators(G); H;
Finitely presented group H on 2 generators
Generators as words in group G
  H.1 = G.2
  H.2 = G.5
Relations
  H.2^-2 * H.1^-2 * H.2^-1 * H.1^2 * H.2 * H.1 * H.2^-1 * H.1 = Id(H)
  H.2^-1 * H.1^-1 * H.2^-2 * H.1^-1 * H.2^-1 * H.1 * H.2^-2 * H.1^-1 * H.2
  * H.1^-1 = Id(H)
> L2Quotients(H);
[
  Ideal of Polynomial ring of rank 3 over Integer Ring
  Order: Lexicographical
  Variables: x1, x2, x12
  Inhomogeneous, Dimension 0
  Groebner basis:
  [
    x1 + x12,
    x2 + x12,
    x12^3 + x12 + 1,
    2
  ]
]
> [L2Type(Q) : Q in $1];
[ PSL( 2, 2^3 ) ]
>
> G := CHR([9, 1, 4]);
> H := ReduceGenerators(G); H;
Finitely presented group H on 2 generators
Generators as words in group G
  H.1 = G.1
  H.2 = G.5
Relations
  H.2 * H.1 * H.2^-1 * H.1 * H.2^-2 * H.1^2 * H.2^2 * H.1^-1 * H.2 = Id(H)
  H.1^-1 * H.2 * H.1^-1 * H.2^2 * H.1^-1 * H.2^3 * H.1^-1 * H.2 * H.1^-1 * H.2^2
  * H.1^-1 * H.2 * H.1^-1 = Id(H)
> [L2Type(Q) : Q in L2Quotients(H)];
[]

```

**Example H70E24**

---

The algorithm also handles the case in which there are infinitely many  $L_2$  quotients. This is the case if one of the prime ideals returned by `L2Quotients` is not maximal. They can be handled theoretically to get a precise list of all images. The functions in Magma can be used to get information for a specified ideal containing the prime ideal.

We start again with a quotient of the modular group.

```
> Gamma< x, y > := Group< x, y | x^2, y^3 >;
> u := x*y; v := x*y^-1;
> G := quo< Gamma | u^5*v*u*v*u*v^5*u*v*u*v >;
> quot := L2Quotients(G);
> [L2Type(P) : P in quot];
[ infinite (characteristic zero) ]
```

This means that there are infinitely many  $L_2$  quotients of  $G$  (which already proves that  $G$  is infinite). Let's look at the prime ideal.

```
> P := quot[1]; P;
Ideal of Polynomial ring of rank 3 over Integer Ring
Order: Lexicographical
Variables: x1, x2, x12
Inhomogeneous, Dimension 0
Groebner basis:
[
  x1,
  x2 + 1,
  x12^8 - 5*x12^6 + 6*x12^4 - 1
]
```

The residue class ring of  $P$  is a finite algebraic extension  $S$  of  $Z$ , and the algorithm found a homomorphism of  $G$  into  $\mathrm{PSL}(2, S)$ . Since  $S$  has epimorphisms onto finite fields in every characteristic, this gives homomorphisms of  $G$  into  $\mathrm{PSL}(2, p^k)$  for every prime  $p$  and suitable  $k$ . To check what images there are in characteristic  $p$ , we construct the ideal  $I = \langle p \rangle + P$ . This is no longer prime in general. We can use `L2Ideals` to compute the associated primes of  $I$ ; this method also removes all ideals which don't give rise to epimorphisms.

```
> R := Generic(P);
> I := ideal< R | 2 > + P;
> quot := L2Ideals(I);
> [L2Type(Q) : Q in quot];
[ PSL( 2, 2^4 ) ]
```

So  $G$  has one quotient isomorphic to  $\mathrm{PSL}(2, 16)$ , and we can construct images of the generators of  $G$ .

```
> L2Generators(quot[1]);
MatrixGroup(2, GF(2^4))
Generators:
[ 1 $.1^6 ]
[ 0 1 ]
```

$$\begin{bmatrix} & 0 & 1 \\ & 1 & 1 \end{bmatrix}$$

Mapping from: MatrixGroup(2, GF(2<sup>4</sup>)) to GL(2, ext<GF(2) | Polynomial(GF(2), \[1, 0, 0, 1, 1])>)

We can check the images in various other characteristics:

```
> for p in PrimesInInterval(3, 19) do
>   [L2Type(Q) : Q in L2Ideals(ideal< R | p > + P)];
> end for;
[ PGL( 2, 3^4 ) ]
[ PGL( 2, 5^2 ) ]
[ PSL( 2, 7^2 ), PSL( 2, 7^2 ), PGL( 2, 7^2 ) ]
[ PGL( 2, 11 ), PGL( 2, 11 ), PGL( 2, 11^2 ) ]
[ PSL( 2, 13^2 ), PSL( 2, 13^2 ), PSL( 2, 13^2 ), PSL( 2, 13^2 ) ]
[ PSL( 2, 17^4 ), PSL( 2, 17^4 ) ]
[ PSL( 2, 19 ), PSL( 2, 19 ), PSL( 2, 19^2 ), PSL( 2, 19^2 ), PGL( 2, 19 ) ]
```

There is virtually no limit on the size of the prime:

```
> p := RandomPrime(200);
> p;
307941320171307176726971038693755343299358400663182805777637
> I := ideal< R | p > + P;
> [L2Type(Q) : Q in L2Ideals(I)];
[ PGL( 2, 307941320171307176726971038693755343299358400663182805777637^2 ),
PGL( 2, 307941320171307176726971038693755343299358400663182805777637^2 ) ]
> p := NextPrime(p);
> I := ideal< R | p > + P;
> quot := L2Ideals(I);
> [L2Type(Q) : Q in quot];
[ PSL( 2, 307941320171307176726971038693755343299358400663182805777879 ),
PSL( 2, 307941320171307176726971038693755343299358400663182805777879 ),
PSL( 2, 307941320171307176726971038693755343299358400663182805777879^2 ),
PSL( 2, 307941320171307176726971038693755343299358400663182805777879^2 ),
PGL( 2, 307941320171307176726971038693755343299358400663182805777879 ) ]
> L2Generators(quot[1]);
MatrixGroup(2, GF(307941320171307176726971038693755343299358400663182805777879))
Generators:
[0 1]
[307941320171307176726971038693755343299358400663182805777878 0]
[260117736370596621578446660105615721012604377822664417621373
39170644893202404516985879121051845196952956796906799404293]
[124461947209331225807153039245860372991649922432590162143941
47823583800710555148524378588139622286754022840518388156505]
Mapping from: MatrixGroup(2, GF(3079413201713071767269710386937553432993584006631\
82805777879)) to GL(2, GF(307941320171307176726971038693755343299358400\
663182805777879, 1))
```

**Example H70E25**

In the previous example, the group had infinitely many  $L_2$  quotients, finitely many in every characteristic. The next example presents a group which has infinitely many  $L_2$  quotients, but only in a single characteristic.

```
> G := Group< x, y | x*y*x^-1*y*x*y^4*x^-1*y^-4 >;
> quot := L2Quotients(G);
> [L2Type(Q) : Q in quot];
[ infinite (characteristic 41) ]
> P := quot[1]; P;
Ideal of Polynomial ring of rank 3 over Integer Ring
Order: Lexicographical
Variables: x1, x2, x12
Inhomogeneous, Dimension >0
Groebner basis:
[
  x1 + 30*x2*x12,
  x2^2 + 22,
  41
]
```

The residue class ring of  $P$  is  $S = F_{41^2}[x_{12}]$ , and the algorithm found a homomorphism into  $\mathrm{PSL}(2, S)$ . Since  $S$  has epimorphisms onto every finite extension of  $F_{41^2}$ , this gives homomorphisms of  $G$  into  $\mathrm{PSL}(2, 41^{(2 * k)})$  for every  $k$ .

We specialize  $x_{12}$  to different elements of extensions of  $F_{41^2}$ . For example, for different specializations in  $F_{41^2}$  we get an image onto  $\mathrm{PGL}(2, 41)$  or  $\mathrm{PSL}(2, 41^2)$ :

```
> R< x1, x2, x12 > := Generic(P);
>
> I := ideal< R | x12 - 1 > + P;
> [L2Type(Q) : Q in L2Ideals(I)];
[ PGL( 2, 41 ) ]
>
> I := ideal< R | x12 - (x2 + 1) > + P;
> [L2Type(Q) : Q in L2Ideals(I)];
[ PSL( 2, 41^2 ) ]
```

We can check the images for a random specialization:

```
> pol := &+[Random([0..40])*x12^i : i in [0..30]]; pol;
19*x12^30 + x12^29 + 16*x12^28 + 14*x12^27 + 38*x12^26 + 27*x12^25 + 20*x12^24 +
  31*x12^23 + 15*x12^22 + 22*x12^21 + 32*x12^20 + 26*x12^19 + 26*x12^18 +
  18*x12^17 + 16*x12^16 + 29*x12^15 + 18*x12^14 + 11*x12^13 + 3*x12^12 +
  11*x12^11 + 36*x12^10 + 13*x12^9 + 15*x12^8 + 33*x12^7 + 30*x12^6 + 8*x12^5 +
  12*x12^4 + 40*x12^3 + 12*x12^2 + 25*x12 + 8;
> I := ideal< R | pol > + P;
> [L2Type(Q) : Q in L2Ideals(I)];
[ PGL( 2, 41 ), PGL( 2, 41 ), PGL( 2, 41 ), PSL( 2, 41^2 ), PSL( 2, 41^2 ),
  PSL( 2, 41^2 ), PSL( 2, 41^2 ), PSL( 2, 41^14 ), PSL( 2, 41^14 ),
```

```
PGL( 2, 41^9 ) ]
```

And again, this works for very large fields:

```
> pol := IrreduciblePolynomial(GF(41), 301);
> pol;
$.1^301 + $.1^2 + 6*$.1 + 30
> I := ideal< R | x12^301 + x12^2 + 6*x12 + 30 > + P;
> [L2Type(Q) : Q in L2Ideals(I)];
[ PGL( 2, 41^301 ) ]
```

### 70.4.6 Infinite L2 quotients

This section contains functions that use the L2-quotient algorithm of Plesken and Fabianska ([PF09]) to establish the existence or not of an infinite quotient of a finitely-presented group in  $PSL_2(K)$  for  $K$  a field of characteristic zero. The algorithm is often used to find surjective quotients over finite fields but as a test for non-finiteness it is easier to work directly in characteristic zero. We make some comments on the relation to finite fields below.

The algorithm first constructs an affine scheme  $X$  over  $\mathbf{Q}$  from the “trace ideal” of the group  $G$ , for which the algebraic points over a field  $K$  are in 1-1 correspondence with equivalence classes of representations of  $G$  into  $SL_2(K)$ . If  $g_1, \dots, g_n$  are the generators of  $G$ , then the affine coordinates of  $X$  correspond to the trace under the representation of various products of the  $G_i$ . For example, when  $n = 2$ ,  $X$  lies in 3-dimensional affine space and the coordinates correspond to the traces of  $g_1, g_2$  and  $g_1g_2$ . The ideal is actually generated by polynomials with coefficients in  $\mathbf{Z}$  so  $X$  is naturally defined as a scheme over  $Spec(\mathbf{Z})$ , whose reduction mod  $p$  just gives the scheme corresponding to characteristic  $p$  representations of  $G$  for  $p > 2$ .

Homomorphisms into  $PSL_2$  rather than  $SL_2$  are dealt with by considering a number of  $X$  schemes for a particular  $G$ . Each one represents the homomorphisms from the free cover of  $G$  to  $SL_2$  that takes each word in the set of relations defining  $G$  to either  $I$  or  $-I$ . Each of the  $2^r$  choices of sign for the  $r$  defining relations gives an  $X$  (by the same procedure) and the totality cover all possibilities for maps to  $PSL_2$ .

The set of points of  $X$  corresponding to geometrically reducible, dihedral  $A_4, S_4$  or  $A_5$  images in  $PSL_2$  is a closed subscheme  $Y$ . In fact the first two possibilities give closed subschemes with equations defined over  $\mathbf{Z}$  and the last three give a dimension zero scheme over  $\mathbf{Q}$ . The overall equations defining  $Y$  reduce mod  $p$  to those defining the corresponding subscheme in characteristic  $p$  for almost all primes  $p$ . The explicit equations defining  $Y$  are determined by the algorithm as explained for  $n = 2$  or  $3$  in the above reference or in more detail in Fabianska’s MSc thesis ([Fab09]). Let  $U$  be the open complement of  $Y$  in  $X$ .

There are two possibilities.

- 1 A  $U$  is non-empty, so an algebraic point gives  $\phi : G \rightarrow PSL_2(K)$ ,  $K$  a number field, with infinite image, so  $G$  is infinite. Further, for all but finitely many primes  $p$ ,  $\phi$  can be reduced mod  $v$  for any place  $v$  of characteristic  $p$  AND the reduction  $\phi_v$  corresponds to

a point in the analogue of  $U$  over the finite field. Thus the reductions give surjections of  $G$  onto a  $PSL_2(k)$  [or maybe a  $PGL_2(k)$ ] for finite fields of any characteristic outside of a finite set.

- 2 All  $U$ s are empty. Here, all the homomorphisms of  $G$  into  $PSL_2(K)$  in characteristic zero have geometrically reducible, dihedral,  $A_4$ ,  $S_4$  or  $A_5$  images and the same holds for  $K$  a field of characteristic  $p$  for all  $p$  outside of a finite set of primes.  $G$  may or may not be infinite.

**NB:** In case 2), there may still be images of  $G$  which ARE infinite, but lie in a Cartan or Borel subgroup, so are geometrically reducible, or lie in the normaliser of a Cartan and have an infinite dihedral image. These infinite reducible/dihedral possibilities are not currently checked for.

HasInfinitePSL2Quotient(G)		
----------------------------	--	--

<b>signs</b>	SEQENUM	Default : 0
<b>full</b>	BOOLELT	Default : false
<b>Verbose</b>	IsInfGrp	Maximum : 1

Function to return whether the two-generator finitely-presented group  $G$  has an infinite quotient lying in a  $PSL_2(K)$  with  $K$  a field of characteristic 0 as described above.

For the convenience of the user, we have provided some parameters to output more detailed information coming from the analysis of the  $X$  representation schemes as well as to control which sign variations are considered.

Let  $ws$  be the sequence of words giving the defining relations of  $G$  (if a defining relation is of the form  $w = v$  where  $v$  is not the identity word, then the corresponding word is  $w * v^{-1}$ ). Let  $ws = [w_1, \dots, w_r]$ .

As described in the introduction, for each of the  $2^r$  combinations of signs attached to the  $w_i$  - let  $s$  be one - the program will consider the scheme  $X$  of homomorphisms of  $G$  into  $PSL_2$  such that, when lifted to a homomorphism of the two-generator free cover  $F$  of  $G$  into  $SL_2$ ,  $w_i$  maps to  $s_i * I$ .

In some cases, the user may realise that there is no point in considering certain choices of signs. For example, if  $g_1^2$  is a relation in  $ws$ ,  $g_1^2 \mapsto I$  in  $SL_2(K)$  means that  $g_1 \mapsto 1$  in  $PSL_2(K)$ , so the  $PSL_2(K)$  image would be (maybe infinitely) cyclic, and it is a waste of time to consider this possibility. Similarly, if  $g_1^r$ ,  $r$  odd, is a relation then, without loss of generality, in a  $PSL_2$  to  $SL_2$  lift,  $g_1^r \mapsto I$ , which could be specified.

The parameter **signs** allows the user to specify a restricted set of sign options to analyse. **signs** should be an 0, 1, -1 or a sequence of length  $\#ws$  of such integers. A single value is converted into the sequence containing that value  $\#ws$  times. An entry  $e$  of 1 or -1 in position  $i$  means that the function will only consider sign sequences  $s$  with  $s[i] = e$ , ie homomorphisms where  $w_i$  map to  $eI$  in the lift to  $SL_2$ . If  $e = 0$ , then there is no condition at place  $i$  of the sign sequences considered. The default value for **signs** is the single value 0.

For each representation space  $X$  (corresponding to an allowable choice of signs  $s$ ), after removing positive dimensional components of  $Y$  corresponding to geometrically reducible or dihedral type representations, there is often a zero-dimensional subscheme left. The existence of an infinite (non-cyclic or dihedral) image homomorphism to  $PSL_2$  then comes down to examining the finite set of closed points remaining and finding one that is not geometrically reducible, dihedral,  $A_4$ ,  $S_4$  or  $A_5$  type. Clearly, once one such is found the procedure can stop. However, it might be of interest for the user to see the types of ALL of the representations corresponding to closed points if the zero-dimensional analysis stage is performed.

If parameter `full` is set to `true` (the default is `false`), the program will continue analysing all of the representations in the 0-dimensional locus, even after one corresponding to an infinite image is found. Furthermore a sequence of (signs,types) pairs is also returned which gives, for each sign combination  $s$  of maps considered, the sequence of types corresponding to the 0-dimensional locus (or empty if we don't reduce to dimension 0). These types are given as strings: "infinite", "reducible", "dihedral", "A4", "S4", and "A5".

Setting the verbose flag `IsInfGrp` to `true` or 1 gives output of information on the various stages as the function progresses, including the analysis of dimension zero loci.

#### Example H70E26

---

We look at two examples of quotients of the two-generator free group with relations  $g_1^2$ ,  $g_2^3$  and one further word. The first is infinite, the second is not. As noted above, we may as well specify that  $g_1^2$  maps to  $-I$  and  $g_2^3$  to  $I$  in  $SL_2$  and this can be done with the `signs` parameter.

```
> F := FreeGroup(2);
> rel := (F.1 * F.2 * F.1 * F.2 * F.1 * F.2 * F.1 * F.2 * F.1 * F.2 *
> F.1 * F.2 * F.1 * F.2 * F.1 * F.2^-1 * F.1 * F.2 * F.1 * F.2^-1)^2;
> G := quo<F | [F.1^2 ,F.2^3, rel]>;
> HasInfinitePSL2Quotient(G : full := true);
true
[ [*
  [ 1, 1, 1 ],
  []
*], [*
  [ 1, 1, -1 ],
  []
*], [*
  [ 1, -1, 1 ],
  []
*], [*
  [ 1, -1, -1 ],
  []
*], [*
  [ -1, 1, 1 ],
```

```

    [ A4, A4 ]
  *], [*
    [ -1, 1, -1 ],
    [ A5, A5, infinite ]
  *], [*
    [ -1, -1, 1 ],
    [ A4, A4 ]
  *], [*
    [ -1, -1, -1 ],
    [ A5, A5, infinite ]
  *] ]

```

Now try it with the sign restriction.

```

> HasInfinitePSL2Quotient(G : full := true, signs := [-1,1,0]);
true
[ [*
  [ -1, 1, 1 ],
  [ A4, A4 ]
  *], [*
  [ -1, 1, -1 ],
  [ A5, A5, infinite ]
  *] ]

```

The second example is just  $A_5$ .

```

> G := quo<F | [F.1^2 ,F.2^3, (F.1*F.2)^5]>;
> HasInfinitePSL2Quotient(G);
false
> HasInfinitePSL2Quotient(G : full := true, signs := [-1,1,0]);
false
[ [*
  [ -1, 1, 1 ],
  [ A5 ]
  *], [*
  [ -1, 1, -1 ],
  [ A5 ]
  *] ]

```

---

### 70.4.7 Searching for Isomorphisms

This section describes a function for searching for isomorphisms between two finitely presented groups.

<code>SearchForIsomorphism(F, G, m : parameters)</code>
---

Attempt to find an isomorphism from the finitely presented group  $F$  to the finitely presented group  $G$ . The search will be restricted to those homomorphisms for which the sum of the word-lengths of the images of the generators of  $F$  in  $G$  is at most  $m$ .

If an isomorphism  $\phi$  is found, then the values `true`,  $\phi$ ,  $\phi^{-1}$  are returned. Otherwise, the values `false`, `-`, `-` are returned; of course, that does not necessarily mean that the groups are not isomorphic.

An error will result if any of the generators of  $F$  turn out to be trivial.

By setting the verbose flag "IsoSearch" to 1, information about the progress of the search will be printed.

The parameters available for the function `SearchForIsomorphism` are:

<code>All</code>	BOOLELT	<i>Default : false</i>
------------------	---------	------------------------

If `All` is `false` (default), then the function halts and returns as soon as a single isomorphism is found. If `All` is `true`, then the search continues through all possible images that satisfy the image length condition, and a list of pairs  $\langle \phi, \phi^{-1} \rangle$  for all isomorphisms  $\phi : F \rightarrow G$  found is returned as the second return value.

<code>IsomsOnly</code>	BOOLELT	<i>Default : true</i>
------------------------	---------	-----------------------

If `IsomsOnly` is set to `false`, then all homomorphisms  $F \rightarrow G$  will be returned if `All` is `true`, and the first nontrivial homomorphism found will be returned if `All` is `false`.

<code>MaxRels</code>	RNGINTELT	<i>Default : 250 * m</i>
----------------------	-----------	--------------------------

The value of the `MaxRelations` parameter used in runs of `RWSGroup`. It is hard to find a sensible default, because if the value is unnecessarily large then time can be wasted unnecessarily. This may need to be increased if the function is used with finite groups, for example (although it is usually much more efficient to use permutation or matrix representations when testing isomorphism of finite groups).

<code>CycConjTest</code>	BOOLELT	<i>Default : true</i>
--------------------------	---------	-----------------------

When `CycConjTest` is `true` and `All` is `false`, then images of the first generator which have a cyclic conjugate that comes earlier in the lexicographical order are rejected, because there would be a conjugate isomorphism in which the image was the cyclic conjugate. This nearly always results in faster run-times, but occasionally it can happen that the conjugate isomorphism has a larger sum of lengths of generator images, which is clearly bad. So the user has the option of not rejecting such images.

#### Example H70E27

---

John Hillman asked whether the following two groups are isomorphic.

```
> G1<s,t,u> := Group <s,t,u | s*u*s^-1=u^-1, t^2=u^2, t*s^2*t^-1=s^-2,
```

```

>
>                                     u*(s*t)^2=(s*t)^2*u >;
> G2<x,y,z> := Group<x,y,z | x*y^2*x^-1=y^-2, y*x^2*y^-1=x^-2, x^2=z^2*(x*y)^2,
>                                     y^2=(z^-1*x)^2, z*(x*y)^2=(x*y)^2*z >;
> isiso, f1, f2 := SearchForIsomorphism(G1,G2,7);
> isiso;
true
> f1;
Homomorphism of GrpFP: G1 into GrpFP: G2 induced by
  s |--> x * z^-1
  t |--> y * z
  u |--> x * y^-1 * z
> f2;
Homomorphism of GrpFP: G2 into GrpFP: G1 induced by
  x |--> s^2 * u * t^-1
  y |--> s^-1 * u^-1
  z |--> s * u * t^-1

```

The search for an isomorphism succeeded and returns the isomorphisms explicitly.

### Example H70E28

---

Walter Neumann asked whether the next two groups are isomorphic.

```

> G1<x,y,z> := Group< x,y,z | x^2*y^5, x^14*z^23, (x^2,y), (x^2,z), x*y*z>;
> G2<a,b> := Group<a,b | a*b^16*a*b^-7, a^4*b^7*a^-1*b^7>;

```

It is a good idea to minimize the number of generators of  $G_1$  - since there is clearly a redundant generator here.

```

> G1s := Simplify(G1);
> Ngens(G1s);
2
> G1!G1s.1, G1!G1s.2;
x z

```

Now a direct call `SearchForIsomorphism(G1, G2, 15)` will succeed, but will take many hours, and also use a lot of memory.

In contrast to  $G_1$ , it can sometimes help to introduce a new generator in  $G_2$  if there are common substrings in relators.

```

> G2b<a,b,c> := Group< a,b,c | a*b^16*a*b^-7, a^4*b^7*a^-1*b^7, c=b^7>;
> isiso, f1, f2 := SearchForIsomorphism(G1s,G2b,4);
> isiso;
true
> f1;
Homomorphism of GrpFP: G1s into GrpFP: G2b induced by
  G1s.1 |--> a * c^-1
  G1s.2 |--> c
> f2;
Homomorphism of GrpFP: G2b into GrpFP: G1s induced by

```

```

a |--> G1s.1 * G1s.2
b |--> G1s.1^6 * G1s.2^10
c |--> G1s.2

```

---

## 70.5 Abelian, Nilpotent and Soluble Quotient

### 70.5.1 Abelian Quotient

The functions in this section compute information about the abelian quotient of an fp-group  $G$ . Some functions may require the computation of a coset table. Experienced users can control the behaviour of an implicit coset enumeration with a set of global parameters. These global parameters can be changed using the function `SetGlobalTParameters`. For a detailed description of the available parameters and their meanings, we refer to Chapter 71.

The functions returning the abelian quotient or the abelian quotient invariants report an error if the abelian quotient cannot be computed, for example, because the relation matrix is too large. To avoid problems in user written programs or loops, the functions `HasComputableAbelianQuotient` and `HasInfiniteComputableAbelianQuotient` can be used.

`AbelianQuotient(G)`

The maximal abelian quotient  $G/G'$  of the group  $G$  as `GrpAb` (cf. Chapter 69). The natural epimorphism  $\pi : G \rightarrow G/G'$  is returned as second value.

`ElementaryAbelianQuotient(G, p)`

The maximal  $p$ -elementary abelian quotient  $Q$  of the group  $G$  as `GrpAb` (cf. Chapter 69). The natural epimorphism  $\pi : G \rightarrow Q$  is returned as second value.

`AbelianQuotientInvariants(G)`

`AQInvariants(G)`

Given a finitely presented group  $G$ , this function computes the elementary divisors of the derived quotient group  $G/G'$ , by constructing the relation matrix for  $G$  and transforming it into Smith normal form. The algorithm used is an algorithm of Havas which does the reduction entirely over the ring of integers  $\mathbf{Z}$  using clever heuristics to minimize the growth of coefficients.

The divisors are returned as a sequence of integers.

AbelianQuotientInvariants(H)
------------------------------

AQInvariants(H)
-----------------

AbelianQuotientInvariants(G, T)
---------------------------------

AQInvariants(G, T)
--------------------

Given a subgroup  $H$  of the finitely presented group  $G$ , this function computes the elementary divisors of the derived quotient group of  $H$ . (The coset table  $T$  may be used to define  $H$ .) This is done by abelianising the Reidemeister-Schreier presentation for  $H$  and then proceeding as above. The divisors are returned as a sequence of integers.

AbelianQuotientInvariants(G, n)
---------------------------------

AQInvariants(G, n)
--------------------

Given a finitely presented group  $G$ , this function computes the elementary divisors of the quotient group  $G/N$ , where  $N$  is the normal subgroup of  $G$  generated by the derived group  $G'$  together with all  $n$ -th powers of elements of  $G$ . The algorithm constructs the relation matrix corresponding to the presentation of  $G$  and computes its Smith normal form over the ring  $Z/nZ$ . The calculation is particularly efficient when  $n$  is a small prime. The divisors are returned as a sequence of integers.

AbelianQuotientInvariants(H, n)
---------------------------------

AQInvariants(H, n)
--------------------

AbelianQuotientInvariants(G, T, n)
------------------------------------

AQInvariants(G, T, n)
-----------------------

Given a subgroup  $H$  of the finitely presented group  $G$ , this function computes the elementary divisors of the quotient group  $H/N$ , where  $N$  is the normal subgroup of  $H$  generated by  $H'$  together with all  $n$ -th powers of elements of  $H$ . (The coset table  $T$  may be used to define  $H$ .) This is done by abelianising the Reidemeister-Schreier presentation for  $H$  and then proceeding as above. The divisors are returned as a sequence of integers.

HasComputableAbelianQuotient(G)
---------------------------------

Given an fp-group  $G$ , this function tests whether the abelian quotient of  $G$  can be computed. If so, it returns the value `true`, the abelian quotient  $A$  of  $G$  and the natural epimorphism  $\pi : G \rightarrow A$ . If the abelian quotient of  $G$  cannot be computed, the value `false` is returned.

This function is especially useful to avoid runtime errors in user written loops or functions.

**HasInfiniteComputableAbelianQuotient(G)**

Given an fp-group  $G$ , this function tests whether the abelian quotient of  $G$  can be computed and is infinite. If so, it returns the value `true`, the abelian quotient  $A$  of  $G$  and the natural epimorphism  $\pi : G \rightarrow A$ . If the abelian quotient of  $G$  cannot be computed or if it is finite, the value `false` is returned.

The function first checks the modular abelian invariants for a set of small primes. If for one of these primes, the modular abelian quotient is trivial,  $A$  must be finite and the function returns without actually computing the abelian quotient. If one is interested only in infinite quotients, this heuristics may save time.

**IsPerfect(G)**

Given an fp-group  $G$ , this function tries to decide whether  $G$  is perfect by checking whether the abelian quotient of  $G$  is trivial.

**TorsionFreeRank(G)**

Given the finitely presented group  $G$ , return the torsion-free rank of the derived quotient group of  $G$ .

**Example H70E29**

The Fibonacci group  $F(7)$  has the following 2-generator presentation:

$$\langle a, b \mid a^2 b^{-2} a^{-1} b^{-2} (a^{-1} b^{-1})^2, abab^2 abab^{-1} (ab^2)^2 \rangle .$$

We proceed to investigate the structure of this group.

```
> F<a, b> := FreeGroup(2);
> F7<a, b> := quo< F | a^2*b^-2*a^-1*b^-2*(a^-1*b^-1)^2,
>      a*b*a*b^2*a*b*a*b^-1*(a*b^2)^2 >;
> F7;
```

Finitely presented group F7 on 2 generators

Relations

$$\begin{aligned} a^2 * b^{-2} * a^{-1} * b^{-2} * a^{-1} * b^{-1} * a^{-1} * b^{-1} &= \text{Id}(F7) \\ a * b * a * b^2 * a * b * a * b^{-1} * a * b^2 * a * b^2 &= \text{Id}()7 \end{aligned}$$

We begin by determining the structure of the maximal abelian quotient of  $F(7)$ .

```
> AbelianQuotientInvariants(F7);
[ 29 ]
```

The maximal abelian quotient of  $F(7)$  is cyclic of order 29. At this point there is no obvious way to proceed, so we attempt to determine the index of some subgroups.

```
> Index( F7, sub< F7 | a > );
1
```

We are in luck:  $F(7)$  is generated by  $a$  and so must be cyclic. This fact coupled with the knowledge that its abelian quotient has order 29 tells us that the group is cyclic of order 29.

**Example H70E30**

---

The group  $G = (8, 7 \mid 2, 3)$  is defined by the presentation

$$\langle a, b \mid a^8, b^7, (ab)^2, (a^{-1}b)^3 \rangle.$$

We consider the subgroup  $H$  of  $G$ , generated by the words  $a^2$  and  $a^{-1}b$ :

```
> G<a, b> := Group<a, b | a^8, b^7, (a * b)^2, (a^-1 * b)^3>;
> H<x, y> := sub< G | a^2, a^-1 * b >;
```

The fastest way to determine the order of the maximal 2-elementary abelian quotient of  $H$  is to use the function `AbelianQuotientInvariants`:

```
> #AbelianQuotientInvariants(H,2);
1
```

We see that the maximal 2-elementary abelian quotient of  $H$  has order  $2^1$ .

---

**70.5.2  $p$ -Quotient**

Let  $F$  be a finitely presented group,  $p$  a prime and  $c$  a positive integer. A  $p$ -quotient algorithm constructs a consistent power-conjugate presentation for the largest  $p$ -quotient of  $F$  having lower exponent- $p$  class at most  $c$ . The  $p$ -quotient algorithm used by MAGMA is part of the ANU  $p$ -Quotient program. For details of the algorithm, see [NO96]. In MAGMA the result is returned as a group of type `GrpPC` (cf. Chapter 63).

Assume that the  $p$ -quotient has order  $p^n$ , Frattini rank  $d$ , and that its generators are  $a_1, \dots, a_n$ . Then the power-conjugate presentation constructed has the following additional structure. The set  $\{a_1, \dots, a_d\}$  is a generating set for  $G$ . For each  $a_k$  in  $\{a_{d+1}, \dots, a_n\}$ , there is at least one relation whose right hand side is  $a_k$ . One of these relations is taken as the *definition* of  $a_k$ . (The list of definitions is also returned by `pQuotient`.) The power-conjugate generators also have a *weight* associated with them: a generator is assigned a weight corresponding to the stage at which it is added and this weight is extended to all normal words in a natural way.

The  $p$ -quotient function and its associated commands allows the user to construct a power-conjugate presentation (pcp) for a  $p$ -group. Note that there is also a process version of the  $p$ -quotient algorithm, which gives the user complete control over its execution. For a description, we refer to Chapter 71.

### 70.5.3 The Construction of a $p$ -Quotient

`pQuotient(F, p, c: parameters)`

Given an fp-group  $F$ , a prime  $p$  and a positive integer  $c$ , construct a pcp for the largest  $p$ -quotient  $G$  of  $F$  having lower exponent- $p$  class at most  $c$ . If  $c$  is given as 0, then the limit 127 is placed on the class. The function also returns the natural homomorphism  $\pi$  from  $F$  to  $G$ , a sequence  $S$  describing the definitions of the pc-generators of  $G$  and a flag indicating whether  $G$  is the maximal  $p$ -quotient of  $F$ .

The  $k$ -th element of  $S$  is a sequence of two integers, describing the definition of the  $k$ -th pc-generator  $G.k$  of  $G$  as follows.

- If  $S[k] = [0, r]$ , then  $G.k$  is defined via the image of  $F.r$  under  $\pi$ .
- If  $S[k] = [r, 0]$ , then  $G.k$  is defined via the power relation for  $G.r$ .
- If  $S[k] = [r, s]$ , then  $G.k$  is defined via the conjugate relation involving  $G.r^{G.s}$ .

There exist a number of parameters for controlling the behaviour of this function, which are described below.

---

#### Example H70E31

We construct the largest 2-quotient of class 6 for a two-generator, two-relator group.

```
> F<a,b> := FreeGroup(2);
> G := quo< F | (b, a, a) = 1, (a * b * a)^4 = 1 >;
> Q, fQ := pQuotient(G, 2, 6);
> Order(Q);
524288
> fQ;
Mapping from: GrpFP: G to GrpPC: Q
```

---

The parameters available for the function `pQuotient` are:

<b>Exponent</b>	RNGINTELT	<i>Default : 0</i>
If <b>Exponent</b> := $m$ , enforce the exponent law, $x^m = 1$ , on the group.		
<b>Metabelian</b>	BOOLELT	<i>Default : false</i>

If **Metabelian** := `true`, then a consistent pcp is constructed for the largest metabelian  $p$ -quotient of  $F$  having lower exponent- $p$  class at most  $c$ .

<b>Print</b>	RNGINTELT	<i>Default : 0</i>
--------------	-----------	--------------------

This parameter controls the volume of printing. By default its value is that returned by `GetVerbose("pQuotient")`, which is 0 unless it has been changed through use of `SetVerbose`. The effect is the following:

**Print** := 0: No output.

**Print** := 1: Report order of  $p$ -quotient at each class.

`Print := 2`: Report statistics and redundancy information about tails, consistency, collection of relations and exponent enforcement components of calculation.

`Print := 3`: Report in detail on the construction of each class.

Note that the presentation displayed is a *power-commutator* presentation (since this is the version stored by the  $p$ -quotient).

`Workspace`                      `RNGINTELT`                      `Default : 5000000`

The amount of space requested for the  $p$ -quotient computation.

---

### Example H70E32

We construct the largest 3-quotient of class 6 for a two-generator group of exponent 9.

```
> F<a,b> := FreeGroup(2);
> G := quo< F | a^3 = b^3 = 1 >;
> q := pQuotient(G, 3, 6: Print := 1, Exponent := 9);
Lower exponent-3 central series for G
Group: G to lower exponent-3 central class 1 has order 3^2
Group: G to lower exponent-3 central class 2 has order 3^3
Group: G to lower exponent-3 central class 3 has order 3^5
Group: G to lower exponent-3 central class 4 has order 3^7
Group: G to lower exponent-3 central class 5 has order 3^9
Group: G to lower exponent-3 central class 6 has order 3^11
```

---

### Example H70E33

We use the metabelian parameter to construct a metabelian 5-quotient of the group

$$\langle a, b \mid a^{625} = b^{625} = 1, (b, a, b) = 1, (b, a, a, a, a) = (b, a)^5 \rangle.$$

```
> F<a, b> := FreeGroup(2);
> G := quo< F | a^625 = b^625 = 1, (b, a, b) = 1,
>      (b, a, a, a, a) = (b, a)^5 >;
> q := pQuotient(G, 5, 20: Print := 1, Metabelian := true);
Lower exponent-5 central series for G
Group: G to lower exponent-5 central class 1 has order 5^2
Group: G to lower exponent-5 central class 2 has order 5^5
Group: G to lower exponent-5 central class 3 has order 5^8
Group: G to lower exponent-5 central class 4 has order 5^11
Group: G to lower exponent-5 central class 5 has order 5^12
Group: G to lower exponent-5 central class 6 has order 5^13
Group: G to lower exponent-5 central class 7 has order 5^14
Group: G to lower exponent-5 central class 8 has order 5^15
Group: G to lower exponent-5 central class 9 has order 5^16
Group: G to lower exponent-5 central class 10 has order 5^17
Group: G to lower exponent-5 central class 11 has order 5^18
Group: G to lower exponent-5 central class 12 has order 5^19
Group: G to lower exponent-5 central class 13 has order 5^20
Group completed. Lower exponent-5 central class = 13, order = 5^20
```

**Example H70E34**

---

In the final example, we construct the largest finite 2-generator group having exponent 5.

```
> F := FreeGroup(2);
> q := pQuotient (F, 5, 14: Print := 1, Exponent := 5);
Lower exponent-5 central series for F
Group: F to lower exponent-5 central class 1 has order 5^2
Group: F to lower exponent-5 central class 2 has order 5^3
Group: F to lower exponent-5 central class 3 has order 5^5
Group: F to lower exponent-5 central class 4 has order 5^8
Group: F to lower exponent-5 central class 5 has order 5^10
Group: F to lower exponent-5 central class 6 has order 5^14
Group: F to lower exponent-5 central class 7 has order 5^18
Group: F to lower exponent-5 central class 8 has order 5^22
Group: F to lower exponent-5 central class 9 has order 5^28
Group: F to lower exponent-5 central class 10 has order 5^31
Group: F to lower exponent-5 central class 11 has order 5^33
Group: F to lower exponent-5 central class 12 has order 5^34
Group completed. Lower exponent-5 central class = 12, order = 5^34
```

---

**70.5.4 Nilpotent Quotient**

A nilpotent quotient algorithm constructs, from a finite presentation of a group, a polycyclic presentation of a nilpotent quotient of the finitely presented group. The nilpotent quotient algorithm used by MAGMA is the Australian National University Nilpotent Quotient program, as described in [Nic96]. The version included in MAGMA is Version 2.2 of January 2007.

The lower central series  $G_0, G_1, \dots$  of a group  $G$  can be defined inductively as  $G_0 = G$ ,  $G_i = [G_{i-1}, G]$ .  $G$  is said to have nilpotency class  $c$  if  $c$  is the smallest non-zero integer such that  $G_c = 1$ . If  $N$  is a normal subgroup of  $G$  and  $G/N$  is nilpotent, then  $N$  contains  $G_i$  for some non-negative integer  $i$ .  $G$  has infinite nilpotent quotients if and only if  $G/G_1$  (the maximal abelian quotient of  $G$ ) is infinite and a prime  $p$  divides a finite factor of a nilpotent quotient if and only if  $p$  divides a cyclic factor of  $G/G_1$ . The  $i$ -th ( $i > 1$ ) factor  $G_{i-1}/G_i$  of the lower central series is generated by the elements  $[g, h]G_i$ , where  $g$  runs through a set of representatives of  $G/G_1$  and  $h$  runs through a set of representatives of  $G_{i-2}/G_{i-1}$ .

Any finitely generated nilpotent group is polycyclic and, therefore, has a subnormal series with cyclic factors. Such a subnormal series can be used to represent the group in terms of a polycyclic presentation. The ANU NQ computes successively the factor groups modulo the terms of the lower central series. Each factor group is represented by a special form of polycyclic presentation, a nilpotent presentation, that makes use of the nilpotent structure of the factor group.

The algorithm has highly efficient code for enforcing the  $n$ -Engel identity in nilpotent groups. When appropriate parameters are set, the algorithm computes the largest  $n$ -Engel quotient of  $G/G_c$ .

More generally, the algorithm has code for enforcing arbitrary identical relations. You can even enforce relations which combine generators of  $G$  with “free variables”. (Werner Nickel calls these “identical generators”.) For instance, the relation  $(a, x) = 1$ , where  $a$  is a group generator and  $x$  is a free variable, will force the construction of quotients where the image of  $a$  is central.

Mike Vaughan-Lee offers advice on when to use the nilpotent quotient algorithm. If you know nothing about the finitely presented group, it is probably a good idea to look at the abelian quotient first. If the abelian quotient is trivial then all nilpotent quotients will be trivial. Similarly, if the abelian quotient is cyclic, then all nilpotent quotients will be cyclic. Less trivially, if the abelian quotient is finite then all nilpotent quotients will be finite and so will be the direct product of finite  $p$ -groups. Moreover, the relevant primes  $p$  will all occur in the abelian quotient. Usually it will be more effective to use the  $p$ -quotient algorithm to study the direct factors. However, if you want to study 3-Engel quotients (say), then you are better using the nilpotent quotient even when the abelian quotient is finite.

NilpotentQuotient( $G$ ,  $c$ : parameters)

This function returns the class  $c$  nilpotent quotient of  $G$  as a group in category GrpGPC, together with the epimorphism  $\pi$  from  $G$  onto this quotient. When  $c$  is set to zero, the function attempts to compute the maximal nilpotent quotient of  $G$ . Using the parameters described below, the user can enforce certain conditions on the quotient found.

### Example H70E35

---

Here is a finitely presented group. The abelian quotient is infinite, so we look at the class 2 nilpotent quotient.

```
> G := Group<x,y,z|(x*y*z^-1)^2, (x^-1*y^2*z)^2, (x*y^-2*x^-1)^2 >;
> AbelianQuotient(G);
Abelian Group isomorphic to Z/2 + Z/2 + Z
Defined on 3 generators
Relations:
  2*$.1 = 0
  2*$.2 = 0
> N := NilpotentQuotient(G,2); N;
GrpGPC : N of infinite order on 6 PC-generators
PC-Relations:
  N.1^2 = N.3^2 * N.5,
  N.2^2 = N.4 * N.6,
  N.4^2 = Id(N),
  N.5^2 = Id(N),
  N.6^2 = Id(N),
  N.2^N.1 = N.2 * N.4,
  N.3^N.1 = N.3 * N.5,
  N.3^N.2 = N.3 * N.6
```

**Example H70E36**

---

The free nilpotent group of rank  $r$  and class  $e$  is defined as  $F/\gamma_{e+1}(F)$ , where  $F$  is a free group of rank  $r$  and  $\gamma_{e+1}(F)$  denotes the  $(e+1)$ st term of the lower central series of  $F$ .

We construct the free nilpotent group  $N$  of rank 2 and class 3 as quotient of the free group  $F$  of rank 2 and the natural epimorphism from  $F$  onto  $N$ .

```
> F<a,b> := FreeGroup(2);
> N<[x]>, pi := NilpotentQuotient(F, 3);
> N;
GrpGPC : N of infinite order on 5 PC-generators
PC-Relations:
  x[2]^x[1] = x[2] * x[3],
  x[2]^(x[1]^-1) = x[2] * x[3]^-1 * x[4],
  x[3]^x[1] = x[3] * x[4],
  x[3]^(x[1]^-1) = x[3] * x[4]^-1,
  x[3]^x[2] = x[3] * x[5],
  x[3]^(x[2]^-1) = x[3] * x[5]^-1
```

Using the function `NilpotencyClass` described in Chapter 72, we check the nilpotency class of the quotient.

```
> NilpotencyClass(N);
3
```

**Example H70E37**

---

The Baumslag-Solitar groups

$$BS(p, q) = \langle a, b | ab^p a^{-1} = b^q \rangle$$

form a fascinating class of 1-relator groups. We compute the nilpotent of class 4 quotient of two of them, and print out the resulting presentations, and the structure of the generators.

```
> G<a,b> := Group<a,b|a*b*a^-1=b^4>;
> N,f := NilpotentQuotient(G,4);
> N;
GrpGPC : N of infinite order on 5 PC-generators
PC-Relations:
  N.2^3 = N.3^2 * N.4^2 * N.5,
  N.3^3 = N.4^2 * N.5^2,
  N.4^3 = N.5^2,
  N.5^3 = Id(N),
  N.2^N.1 = N.2 * N.3,
  N.2^(N.1^-1) = N.2 * N.3^2 * N.4^2 * N.5,
  N.3^N.1 = N.3 * N.4,
  N.3^(N.1^-1) = N.3 * N.4^2 * N.5^2,
  N.4^N.1 = N.4 * N.5,
  N.4^(N.1^-1) = N.4 * N.5^2
> for i := 1 to Ngens(N) do
```

```

> N.i @@ f;
> end for;
a
b
(b, a)
(b, a, a)
(b, a, a, a)
> G<a,b> := Group<a,b|a*b^2*a^-1=b^4>;
> N,f := NilpotentQuotient(G,4);
> N;
GrpGPC : N of infinite order on 8 PC-generators
PC-Relations:
  N.2^2 = Id(N),
  N.3^2 = N.5 * N.8,
  N.4^2 = N.7,
  N.5^2 = N.8,
  N.6^2 = Id(N),
  N.7^2 = Id(N),
  N.8^2 = Id(N),
  N.2^N.1 = N.2 * N.3,
  N.2^(N.1^-1) = N.2 * N.3 * N.4 * N.5 * N.6,
  N.3^N.1 = N.3 * N.4,
  N.3^(N.1^-1) = N.3 * N.4 * N.6 * N.7,
  N.3^N.2 = N.3 * N.5,
  N.4^N.1 = N.4 * N.6,
  N.4^(N.1^-1) = N.4 * N.6,
  N.4^N.2 = N.4 * N.7,
  N.5^N.1 = N.5 * N.7,
  N.5^(N.1^-1) = N.5 * N.7,
  N.5^N.2 = N.5 * N.8
> for i := 1 to Ngens(N) do
> N.i @@ f;
> end for;
a
b
(b, a)
(b, a, a)
(b, a, b)
(b, a, a, a)
(b, a, b, a)
(b, a, b, b)

```

---

The parameters available for the function NilpotentQuotient are:

NumberOfEngelGenerators

RNGINTELT

Default : 1

Setting this parameter to  $k$  forces the first  $k$  generators to be left or right Engel elements, provided one (or both) of the parameters `LeftEngel` or `RightEngel` is positive. Otherwise it is ignored.

`LeftEngel`                      `RNGINTELT`                      *Default : 0*

Setting this parameter to  $n$  forces the first  $k$  generators  $g_1, \dots, g_k$  of the quotient  $Q$  to be left  $n$ -Engel elements. That is, they satisfy  $[x, \dots, x, g_i] = 1$  ( $x$  appearing  $n$  times) for all  $x$  in  $Q$ . The value of  $k$  is determined by the parameter `NumberOfEngelGenerators`.

`RightEngel`                      `RNGINTELT`                      *Default : 0*

This is the same as for `LeftEngel`, but here the generators are right  $n$ -Engel elements, so  $[g_i, x, \dots, x] = 1$ .

`Engel`                              `RNGINTELT`                      *Default : 0*

Setting this parameter to  $n$  enforces the  $n$ th Engel law on the quotient  $Q$ . That is,  $[x, y, \dots, y] = 1$  ( $y$  appearing  $n$  times) for all  $x, y$  in  $Q$ .

`SemigroupOnly`                      `BOOLELT`                      *Default : true*

This option causes the program to check only semigroup words in the generating set of the nilpotent quotient when an Engel condition is enforced. If none of the Engel parameters are set, then it is ignored.

`SemigroupFirst`                      `BOOLELT`                      *Default : false*

This option causes the program to check semigroup words in the generating set of the quotient first and then all other words, when an Engel condition is enforced. If `SemigroupOnly` is set, or no Engel condition is enforced, then it is ignored.

`ReverseOrder`                      `BOOLELT`                      *Default : false*

In checking Engel identities, instances are processed in order of increasing weight. This flag reverses the order.

`ReverseEngel`                      `BOOLELT`                      *Default : false*

This flag changes the Engel conditions from the first  $k$  generators, to the *last*  $k$  generators.

`CheckFewInstances`                      `BOOLELT`                      *Default : false*

This option stops checking the Engel law at each class if all the checks of a certain weight did not yield any non-trivial instances of the law.

`Nickel`                              `BOOLELT`                      *Default : false*

Enforce the identities  $x^8$  and  $[[x_1, x_2, x_3], [x_4, x_5, x_6]]$  on the nilpotent quotient.

`NumberOfFreeVariables`                      `RNGINTELT`                      *Default : 0*

If this parameter is set to  $n > 0$  then the last  $n$  variables of the group are treated as variables rather than generators of the group. In any relation they are treated as standing for all group elements, enabling the user to enforce identical relations on the quotient being computed. The value of  $n$  must be less than the number of generators

of the input group. When this facility is used, the domain of the epimorphism returned is the subgroup of  $G$  generated by the (first) non-free generators.

**PrintResult**                      **BOOLEAN**                      *Default : false*

If set to true, this parameter switches on the printing of results given by the stand alone version of NQ.

**SetVerbose("NilpotentQuotient", n)**

Turn on and off the printing of information as the nilpotent quotient algorithm proceeds.  $n$  may be set to any of 0, 1, 2 or 3. Setting  $n$  to be 0 turns printing off, while successively higher values for  $n$  print more and more information as the algorithm progresses. The default value of  $n$  is 0.

### Example H70E38

---

We compute the maximal nilpotent quotient  $N1$  of the group  $G$  given by the following presentation

$$\langle a, b, c, d, e \mid (b, a), (c, a), (d, a) = (c, b), (e, a) = (d, b), (e, b) = (d, c), (e, c), (e, d) \rangle.$$

```
> F<a,b,c,d,e> := FreeGroup(5);
> G<a,b,c,d,e> :=
>     quo<F | (b,a), (c,a),
>     (d,a)=(c,b), (e,a)=(d,b), (e,b)=(d,c),
>     (e,c), (e,d)>;
> N1<x>, pi1 := NilpotentQuotient(G, 0);
```

Using the function `NilpotencyClass` described in Chapter 72, we check the nilpotency class of the quotient. It turns out to have nilpotency class 6.

```
> NilpotencyClass(N1);
6
```

Next we compute a metabelian quotient  $N2$  of  $G$ , construct the natural epimorphism from  $N1$  onto  $N2$  and check its kernel. (The functions applied to the nilpotent quotients are described in Chapter 72.) To get a metabelian quotient we adjoin 4 variables and a relation to the presentation of  $G$  and use the “free variables” facility.

```
> M := Group<w,x,y,z|((w,x),(y,z))>; // metabelian identity
> D := FreeProduct(G,M); // adjoin to G
> N2, pi2 := NilpotentQuotient(D, 0: NumberOfFreeVariables := 4);
> NilpotencyClass(N2);
4
> DerivedLength(N2);
2
> f := hom< N1->N2 | [ pi1(G.i)->pi2(D.i) : i in [1..Ngens(G)] ] >;
> PCGenerators(Kernel(f), N1);
{@ x[14], x[15] * x[17], x[16], x[17]^2, x[18], x[19], x[20],
x[21], x[22], x[23], x[24], x[25], x[26], x[27], x[28], x[29],
```

```
x[30], x[31] @}
```

We compute a quotient  $N3$  of  $G$  satisfying the 4th Engel law, construct the natural epimorphism from  $N1$  onto  $N3$  and again check the kernel. (The functions applied to the nilpotent quotients are described in Chapter 72.)

```
> N3, pi3 := NilpotentQuotient(G, 0 : Engel := 4);
> NilpotencyClass(N3);
4
> DerivedLength(N3);
3
> h := hom< N1->N3 | [ pi1(g)->pi3(g) : g in Generators(G) ] >;
> PCGenerators(Kernel(h), N1);
{@ x[19], x[20], x[21], x[22], x[23], x[24], x[25], x[26], x[27],
x[28], x[29], x[30], x[31] @}
```

### 70.5.5 Soluble Quotient

A soluble quotient algorithm computes a consistent power-conjugate presentation of the largest finite soluble quotient of a finitely presented group, subject to certain algorithmic and user supplied restrictions. In this section we describe only the simplest use of such an algorithm within MAGMA. For more information the user is referred to Chapter 71.

SolvableQuotient(G : parameters)
----------------------------------

SolubleQuotient(G : parameters)
---------------------------------

Let  $G$  be a finitely presented group. The function constructs the largest finite soluble quotient of  $G$ .

#### Example H70E39

We compute the soluble quotient of the group

$$\langle a, b \mid a^2, b^4, ab^{-1}ab(abab^{-1})^5 ab^2 ab^{-2} \rangle.$$

```
> G<a,b> := Group< a, b | a^2, b^4,
>           a*b^-1*a*b*(a*b*a*b^-1)^5*a*b^2*a*b^-2 >;
> Q := SolubleQuotient(G);
> Q;
GrpPC : Q of order 1920 = 2^7 * 3 * 5
PC-Relations:
  Q.1^2 = Q.4,
  Q.2^2 = Id(Q),
  Q.3^2 = Q.6,
  Q.4^2 = Id(Q),
  Q.5^2 = Q.7,
  Q.6^2 = Id(Q),
```

$$\begin{aligned}
Q.7^2 &= \text{Id}(Q), \\
Q.8^3 &= \text{Id}(Q), \\
Q.9^5 &= \text{Id}(Q), \\
Q.2^Q.1 &= Q.2 * Q.3, \\
Q.3^Q.1 &= Q.3 * Q.5, \\
Q.3^Q.2 &= Q.3 * Q.6, \\
Q.4^Q.2 &= Q.4 * Q.5 * Q.6 * Q.7, \\
Q.4^Q.3 &= Q.4 * Q.6 * Q.7, \\
Q.5^Q.1 &= Q.5 * Q.6, \\
Q.5^Q.2 &= Q.5 * Q.7, \\
Q.5^Q.4 &= Q.5 * Q.7, \\
Q.6^Q.1 &= Q.6 * Q.7, \\
Q.8^Q.1 &= Q.8^2, \\
Q.8^Q.2 &= Q.8^2, \\
Q.9^Q.1 &= Q.9^3, \\
Q.9^Q.2 &= Q.9^4, \\
Q.9^Q.4 &= Q.9^4
\end{aligned}$$

SolvableQuotient(F, n : parameters)
-------------------------------------

SolubleQuotient(F, n : parameters)
------------------------------------

SolvableQuotient(F, P : parameters)
-------------------------------------

SolubleQuotient(F, P : parameters)
------------------------------------

Find a soluble quotient  $G$  and the epimorphism  $\pi : F \twoheadrightarrow G$  with a specified order.  $n$  must be a nonnegative integer.  $P$  must be a set of primes.

The three forms reflect possible information about the order of an expected soluble quotient. In the first form the order of  $G$  is given by  $n$ , if  $n$  is greater than zero. If  $n$  equals zero, nothing about the order is known and the relevant primes will be calculated completely.

The second form, with no  $n$  argument, is equivalent to the first with  $n = 0$ . This is a standard argument, and usually it is the most efficient way to calculate soluble quotients.

Note that, if  $n > 0$  is not the order of the maximal finite soluble quotient, it may happen that no group of order  $n$  can be found, since an epimorphic image of size  $n$  may not be exhibited by the chosen series.

In the third form a set  $P$  of relevant primes is given. The algorithm calculates the biggest quotient such that the order has prime divisors only in  $P$ .  $P$  may have a zero as element, this is just for consistency reasons. It is equivalent to the first form with  $n$  equal zero.

For a description of the algorithm used and of the set of parameters available for this function, see Chapter 71. Other more specialised functions for computing soluble quotients and some examples can be found there as well.

**Example H70E40**

---

Consider the group  $G$  defined by the presentation

$$\langle x, y \mid x^3, y^8, [x, y^4], x^{-1}yx^{-1}y^{-1}xyxy^{-1}, (xy^{-2})^2(x^{-1}y^{-2})^2(xy^2)^2(x^{-1}y^2)^2, (x^{-1}y^{-2})^6(x^{-1}y^2)^6 \rangle.$$

```
> G<x, y> := Group< x, y |
>   x^3, y^8, (x,y^4), x^-1*y*x^-1*y^-1*x*y*x*y^-1,
>   (x*y^-2)^2*(x^-1*y^-2)^2*(x*y^2)^2*(x^-1*y^2)^2,
>   (x^-1*y^-2)^6*(x^-1*y^2)^6 >;
```

We apply the soluble quotient algorithm to  $G$  and compute the order of the soluble quotient  $Q$ .

```
> time Q := SolubleQuotient(G);
Time: 116.920
> Order(Q);
165888
```

Note that  $165888 = 2^{11} \cdot 3^4$ . If we knew the possible primes in advance the soluble quotient can be computed much more quickly by using this knowledge.

```
> time Q := SolubleQuotient(G, {2, 3});
Time: 39.400
```

Note that this reduces the execution time by a factor of almost three.

We now assume that  $G$  is finite and try to compute its order by means of the Todd-Coxeter algorithm.

```
> Order(G);
165888
```

Hence the group is finite and soluble and  $G$  is isomorphic to  $Q$ . We may use the tools for finite soluble groups to investigate the structure of  $G$ . For example we can easily find the number of involutions in  $G$ .

```
> cls := ConjugacyClasses(Q);
> &+ [ cl[2] : cl in cls | cl[1] eq 2 ];
511
```

---

## 70.6 Subgroups

### 70.6.1 Specification of a Subgroup

`sub< G | L >`

Construct the subgroup  $H$  of the fp-group  $G$  generated by the words specified by the terms of the *generator list*  $L = L_1, \dots, L_r$ .

A term  $L_i$  of the generator list may consist of any of the following objects:

- (a) A word;
- (b) A set or sequence of words;
- (c) A sequence of integers representing a word;
- (d) A set or sequence of sequences of integers representing words;
- (e) A subgroup of an fp-group;
- (f) A set or sequence of subgroups.

The collection of words and groups specified by the list must all belong to the group  $G$  and  $H$  will be constructed as a subgroup of  $G$ .

The generators of  $H$  consist of the words specified directly by terms  $L_i$  together with the stored generating words for any groups specified by terms of  $L_i$ . Repetitions of an element and occurrences of the identity element are removed (unless  $H$  is trivial).

If the `sub`-constructor is invoked with an empty list  $L$ , the trivial subgroup will be constructed.

`sub< G | f >`

Given a homomorphism  $f$  from  $G$  onto a transitive subgroup of  $\text{Sym}(n)$ , construct the subgroup of  $G$  which affords this permutation representation.

`ncl< G | L >`

Construct the subgroup  $N$  of the fp-group  $G$  as the normal closure of the subgroup  $H$  generated by the words specified by the terms of the *generator list*  $L$ .

The possible forms of a term of the generator list are the same as for the `sub`-constructor.

This constructor may be applied even when  $H$  has infinite index in  $G$ , provided that its normal closure  $N$  has finite index. The subgroup  $N$  is obtained by computing the coset table of the trivial subgroup in the group defined by the relations of  $G$  together with relators corresponding to the words generating  $H$ . For a sample application of this function, see Example H70E17.

This function may require the computation of a coset table. Experienced users can control the behaviour of a possibly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function `SetGlobalTCPParameters`. For a detailed description of the available parameters and their meanings, we refer to Chapter 71.

ncl< G   f >
--------------

Given a homomorphism  $f$  from  $G$  onto a transitive subgroup of  $\text{Sym}(n)$ , construct the subgroup of  $G$  that is the normal closure of the subgroup  $K$  of  $G$  which affords this permutation representation.

CommutatorSubgroup(G)
-----------------------

DerivedSubgroup(G)
--------------------

DerivedGroup(G)
-----------------

Given an fp-group  $G$ , try to construct the derived subgroup  $G'$  of  $G$  as finite index subgroup of  $G$ . The construction fails if no presentation for  $G$  is known or can be constructed, or if the index of  $G'$  in  $G$  is too large or infinite.

This function may require the computation of a coset table. Experienced users can control the behaviour of a possibly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function `SetGlobalTCPParameters`. For a detailed description of the available parameters and their meanings, we refer to Chapter 71.

---

**Example H70E41**

The group  $(8, 7 | 2, 3)$  is defined by the presentation

$$\langle a, b \mid a^8, b^7, (ab)^2, (a^{-1}b)^3 \rangle,$$

and has a subgroup of index 448 generated by the words  $a^2$  and  $a^{-1}b$ :

```
> G<a, b> := Group<a, b | a^8, b^7, (a * b)^2, (a^-1 * b)^3>;
> G;
```

Finitely presented group G on 2 generators

Relations

$$a^8 = \text{Id}(G)$$

$$b^7 = \text{Id}(G)$$

$$(a * b)^2 = \text{Id}(G)$$

```
> H<x, y> := sub< G | a^2, a^-1 * b >;
```

```
> H;
```

Finitely presented group H on 2 generators

Generators as words in group G

$$x = a^2$$

$$y = a^{-1} * b$$

---

**Example H70E42**

Given the group  $G$  defined by the presentation

$$\langle a, b \mid a^8, b^7, (ab)^2, (a, b)^9 \rangle,$$

there is a homomorphism into  $\text{Sym}(9)$  defined by

$$\begin{aligned} a &\rightarrow (2,4)(3,5)(6,7)(8,9) \\ b &\rightarrow (1,2,3)(4,6,7)(5,8,9) \end{aligned}$$

We construct the subgroup  $H$  of  $G$  that is the preimage of the stabiliser of the point 1 in  $G$ .

```
> G<a, b> := Group< a, b | a^2, b^3, (a*b)^7, (a, b)^9>;
> T := PermutationGroup< 9 | (2, 4)(3, 5)(6, 7)(8, 9),
>   (1, 2, 3)(4, 6, 7)(5, 8, 9) >;
> f := hom< G -> T | a -> T.1, b ->T.2 >;
> H := sub< G | f >;
> H;
Finitely presented group H
Subgroup of group G defined by coset table
> Index(G, H);
9
```

Using the function `GeneratingWords`, we obtain a set of generators for  $H$ .

```
> print GeneratingWords(G, H);
{ a, b^-1 * a * b^3 * a * b, b * a * b * a * b * a * b^-1,
  b^3, b^-1 * a * b * a * b * a * b, b * a * b^3 * a * b^-1 }
```

### 70.6.2 Index of a Subgroup: The Todd-Coxeter Algorithm

This section describes the simplest use of coset enumeration techniques in MAGMA. MAGMA also provides interactive facilities for coset enumeration. For information on these more advanced uses, the user is referred to Chapter 71.

The Todd-Coxeter implementation installed in MAGMA is based on the stand alone coset enumeration programme `ACE3` developed by George Havas and Colin Ramsay at the University of Queensland. The reader should consult [CDHW73] and [Hav91] for an explanation of the terminology and a general description of the algorithm. A manual for `ACE3` as well as the sources of `ACE3` can be found online [Ram].

Experienced users can control the Todd-Coxeter procedures invoked by the functions described in this section with a wide range of parameters. For a complete description of these parameters and their meanings we refer to the manual entry for the function `CosetEnumerationProcess` in Chapter 71. We just mention briefly the most important ones:

<code>CosetLimit</code>	<code>RNGINTELT</code>	<i>Default : 0</i>
-------------------------	------------------------	--------------------

If `CosetLimit` is set to  $n$ , where  $n$  is a positive integer, then the coset table may have at most  $n$  rows. In other words, a maximum of  $n$  cosets can be defined at any instant during the enumeration. It is ensured in this case, that enough memory is allocated to store the requested number of cosets, regardless of the value of the parameter `Workspace`.



**Example H70E43**

---

The classical test example for Todd-Coxeter programmes is the enumeration of the 448 cosets of the subgroup  $H = \langle a^2, a^{-1}b \rangle$  in the group  $G = \langle a, b \mid a^8, b^7, (ab)^2, (a^{-1}b)^3 \rangle$ .

```
> F<x, y> := FreeGroup(2);
> G<a, b> := quo<F | x^8, y^7, (x*y)^2, (x^-1*y)^3>;
> H := sub<G | a^2, a^-1*b>;
> Index(G, H);
448
```

---

Order(G: parameters)
----------------------

FactoredOrder(G: parameters)
------------------------------

Given an fp-group  $G$ , this function attempts to determine the order of  $G$  or to prove that  $G$  is infinite. If a finite order can be computed, the function `Order` returns the order as a positive integer, whereas the function `FactoredOrder` returns a sequence of prime power factors. The function `FactoredOrder` reports an error in all other cases, whereas the function `Order` returns the object `Infinity`, if  $G$  can be shown to be infinite and returns a value of 0 if neither a finite value for the group order nor a proof for the infinity of  $G$  can be obtained. No conclusions can be drawn from a return value 0 of `Order`.

In addition to the parameters controlling possibly invoked coset enumerations, there exist some other parameters controlling the strategy used by the functions `Order` and `FactoredOrder`. These parameters are described below.

**Example H70E44**

---

We use the function `Order` without any parameters to compute the order of the group

$$G = \langle a, b \mid a^8, b^7, (ab)^2, (a^{-1}b)^3 \rangle .$$

```
> G<x, y> := Group<x,y | x^8, y^7, (x*y)^2, (x^-1*y)^3>;
> G;
Finitely presented group G on 2 generators
Relations
  x^8 = Id(G)
  y^7 = Id(G)
  (x * y)^2 = Id(G)
  (x^-1 * y)^3 = Id(G)
> Order(G);
10752
```

---

The strategy employed by the functions `Order` and `FactoredOrder` may involve trying to obtain information on certain subgroups of  $G$ . Whether or not an attempt is made to construct a presentation for a subgroup arising in the course of the computation by means of Reidemeister-Schreier rewriting, is controlled by three parameters:

<code>UseRewrite</code>	BOOLELT	<i>Default</i> : <code>true</code>
<code>MinIndex</code>	RNGINTELT	<i>Default</i> : 10
<code>MaxIndex</code>	RNGINTELT	<i>Default</i> : 1000

If `UseRewrite` is set to `false`, attempts to construct presentations for subgroups are not made. Otherwise, `MinIndex` and `MaxIndex` specify for subgroups of which index range Reidemeister-Schreier rewriting is done.

The following strategy is used for trying to determine the order of  $G$ .

- (1) Check whether  $G$  is free. If so,  $G$  is either trivial or infinite.
- (2) Check whether the presentation for  $G$  is deficient (i.e. whether the number of relations is smaller than the number of generators). If it is,  $G$  is infinite.
- (3) Check the subgroups of  $G$  with known order. If such a subgroup is known to be infinite or if we can compute its index in  $G$ , we're done.
- (4) Try to compute the index of  $G$  in a supergroup of known order. (An infinite supergroup in which  $G$  has finite index proves  $G$  to be infinite.)
- (5) Try to enumerate the cosets of the trivial subgroup in  $G$ .
- (6) Check the subgroups of known or easily computable index in  $G$ . If we can compute the order of such a subgroup or prove that it is infinite, we're done.
- (7) Try to enumerate the cosets of some subgroups occurring "naturally" in the presentation of  $G$ .
- (8) Check the supergroups in which  $G$  has known or easily computable index. If we can compute the order of a supergroup or prove that it is infinite, we're done.
- (9) Try to rewrite  $G$  w.r.t. some supergroup and to enumerate the cosets of the trivial subgroup using the resulting presentation.

Steps requiring coset enumeration in  $G$  or a supergroup of  $G$  are skipped, if no relations are known for this group. Steps involving Reidemeister-Schreier rewriting may be skipped according to the values of the parameters mentioned above.

Experienced users can control the behaviour of coset enumerations which may be invoked by the functions `Order` and `FactoredOrder` with a wide range of parameters. Both functions – in addition to the parameters mentioned above – accept the same parameters as the function `CosetEnumerationProcess` described in Chapter 71.

**Example H70E45**

---

The Harada-Norton simple group has the presentation

$$\langle x, a, b, c, d, e, f, g \mid x^2, a^2, b^2, c^2, d^2, e^2, f^2, g^2, (x, a), (x, g), \\ (bc)^3, (bd)^2, (be)^2, (bf)^2, (bg)^2, (cd)^3, (ce)^2, (cf)^2, (cg)^2, \\ (de)^3, (df)^2, (dg)^2, (ef)^3, (eg)^2, (fg)^3, (b, xbx), (a, edcb), \\ (a, f)dc b d c d, (ag)^5, (cdef, xbx), (b, xcdefx), (cdef, xcdefx) \rangle$$

The subgroup generated by  $x, b, c, d, e, f, g$  has index 1,140,000. We use the parameter `CosetLimit` to request a sufficiently large coset table. For the enumeration we choose the predefined strategy `Hard` with the modification of a complete C-style lookahead (`Lookahead := 2`).

```
> HN<x, a, b, c, d, e, f, g> :=
>   Group< x, a, b, c, d, e, f, g |
>       x^2, a^2, b^2, c^2, d^2, e^2, f^2, g^2,
>       (x, a), (x, g),
>       (b*c)^3, (b*d)^2, (b*e)^2, (b*f)^2, (b*g)^2,
>       (c*d)^3, (c*e)^2, (c*f)^2, (c*g)^2,
>       (d*e)^3, (d*f)^2, (d*g)^2,
>       (e*f)^3, (e*g)^2,
>       (f*g)^3,
>       (b, x*b*x),
>       (a, e*d*c*b), (a, f)*d*c*b*d*c*d, (a*g)^5,
>       (c*d*e*f, x*b*x), (b, x*c*d*e*f*x),
>       (c*d*e*f, x*c*d*e*f*x)
>   >;
> H := sub<HN | x,b,c,d,e,f,g >;
> idx := Index(HN, H: Print := true, CosetLimit := 1200000,
>               Strategy := "Hard", Lookahead := 2);
INDEX = 1140000
(a=1140000 r=1471 h=1168483 n=1168483;
 l=2945 c=201.17;
 m=1142416 t=1470356)
> idx;
1140000
```

**Example H70E46**

---

We use a function representing a parametrised presentation to determine the order of a collection of groups obtained by systematically varying one relation. We select the predefined coset enumeration strategy `Easy` for the order computations.

```
> Grp := func< p, q, r, s |
>
>   Group<
>     x, y, z, h, k, a |
```

```

> x^2, y^2, z^2, (x,y), (y,z), (x,z), h^3, k^3, (h,k),
> (x,k), (y,k), (z,k), x^h*y, y^h*z, z^h*x, a^2, a*x*a*y,
> a*y*a*x, (a,z), (a,k), x^p*y^q*z^r*k^s*(a*h)^2 >
> >;
> [ < i,j,k,l >, Order(Grp(i,j,k,l) : Strategy := "Easy") >
> : i, j, k in [0..1], l in [0..2] ];
[ <<0, 0, 0, 0>, 144>, <<0, 0, 1, 0>, 18>, <<0, 1, 0, 0>, 72>,
  <<0, 1, 1, 0>, 36>, <<1, 0, 0, 0>, 18>, <<1, 0, 1, 0>, 144>,
  <<1, 1, 0, 0>, 36>, <<1, 1, 1, 0>, 72>, <<0, 0, 0, 1>, 144>,
  <<0, 0, 1, 1>, 18>, <<0, 1, 0, 1>, 72>, <<0, 1, 1, 1>, 36>,
  <<1, 0, 0, 1>, 18>, <<1, 0, 1, 1>, 144>, <<1, 1, 0, 1>, 36>,
  <<1, 1, 1, 1>, 72>, <<0, 0, 0, 2>, 144>, <<0, 0, 1, 2>, 18>,
  <<0, 1, 0, 2>, 72>, <<0, 1, 1, 2>, 36>, <<1, 0, 0, 2>, 18>,
  <<1, 0, 1, 2>, 144>, <<1,1, 0, 2>, 36>, <<1, 1, 1, 2>, 72> ]

```

---

### 70.6.3 Implicit Invocation of the Todd-Coxeter Algorithm

Several functions working with finitely presented groups at some point require a coset table of a subgroup and may invoke a coset enumeration indirectly, e.g. the function `meet` or the function `Normaliser`. The default behaviour for such implicitly called coset enumerations is the same as the one for coset enumerations invoked explicitly, e.g. using the function `ToddCoxeter`.

If such an implicitly called coset enumeration fails to produce a closed coset table, the calling function may terminate with a runtime error.

Experienced users can control the behaviour of indirectly invoked coset enumerations with a set of global parameters. These global parameters are valid for all implicitly called coset enumerations. For a detailed description of the available parameters and their meanings, we refer to Chapter 71. Note that coset enumerations which are *explicitly* invoked, e.g. by a call to the function `Index`, are not affected by this global set of parameters. Parameters for these functions have to be specified in the function call.

**SetGlobalTCParameters**(: *parameters*)

This function sets the parameter values used for indirect invocations of the Todd-Coxeter coset enumeration procedure. The parameters accepted and their default values are the same as for the function `CosetEnumerationProcess` described in Chapter 71.

**UnsetGlobalTCParameters**()

This function restores the default values for the parameters used for indirect invocations of the Todd-Coxeter coset enumeration procedure. For a description of the meanings of the parameters and their default values, see `CosetEnumerationProcess` in Chapter 71.

**Example H70E47**

We consider again the Harada-Norton simple group with the presentation

$$\begin{aligned} \langle x, a, b, c, d, e, f, g \mid & x^2, a^2, b^2, c^2, d^2, e^2, f^2, g^2, (x, a), (x, g), \\ & (bc)^3, (bd)^2, (be)^2, (bf)^2, (bg)^2, (cd)^3, (ce)^2, (cf)^2, (cg)^2, \\ & (de)^3, (df)^2, (dg)^2, (ef)^3, (eg)^2, (fg)^3, (b, xbx), (a, edcb), \\ & (a, f)dcbdcd, (ag)^5, (cdef, xbx), (b, xcdefx), (cdef, xcdefx) \rangle \end{aligned}$$

and the subgroup  $H$  generated by  $x, b, c, d, e, f, g$ .

```
> HN<x, a, b, c, d, e, f, g> :=
>   Group< x, a, b, c, d, e, f, g |
>     x^2, a^2, b^2, c^2, d^2, e^2, f^2, g^2,
>     (x, a), (x, g),
>     (b*c)^3, (b*d)^2, (b*e)^2, (b*f)^2, (b*g)^2,
>     (c*d)^3, (c*e)^2, (c*f)^2, (c*g)^2,
>     (d*e)^3, (d*f)^2, (d*g)^2,
>     (e*f)^3, (e*g)^2,
>     (f*g)^3,
>     (b, x*b*x),
>     (a, e*d*c*b), (a, f)*d*c*b*d*c*d, (a*g)^5,
>     (c*d*e*f, x*b*x), (b, x*c*d*e*f*x),
>     (c*d*e*f, x*c*d*e*f*x) >;
> H := sub<HN | x,b,c,d,e,f,g >;
```

$H$  has index 1,140,000 in  $HN$ . Using the default settings, the normaliser of  $H$  in  $HN$  cannot be computed.

```
> N := Normaliser(HN, H);
```

```
>> N := Normaliser(HN, H);
```

```
Runtime error in 'Normaliser': Coset table is not closed
```

We change the global parameters for implicitly called coset enumerations and try again.

```
> SetGlobalTCPParameters( : Strategy := "Hard");
> N := Normaliser(HN, H);
```

With these parameters, the computation works. We see that  $H$  is self-normalising in  $HN$ .

```
> Index(HN, N);
1140000
> IsSelfNormalising(HN, H);
true
```

---

## 70.6.4 Constructing a Presentation for a Subgroup

### 70.6.4.1 Introduction

Let  $H$  be a subgroup of finite index in the finitely presented group  $G$ . It frequently happens that it is desirable to construct a set of defining relations for  $H$  from those of  $G$ . Such a presentation can be obtained either on a set of Schreier generators for  $H$  or on the given generators of  $H$  using the Reidemeister-Schreier rewriting technique [MKS76], if necessary together with extended coset enumeration [AR84, HKRR84].

We emphasise that if the user wishes only to determine the structure of the maximal abelian quotient of  $H$ , then the function `AbelianQuotientInvariants` should be used. In this case there is no need to first construct a presentation for  $H$  using the `Rewrite` function described below, since `AbelianQuotientInvariants` employs a special form of the Reidemeister-Schreier rewriting process which abelianises each relator as soon as it is constructed. Thus, compared to the function `Rewrite`, the function `AbelianQuotientInvariants` can be applied to subgroups of much larger index.

### 70.6.4.2 Rewriting

<code>Rewrite(G, H : parameters)</code>
---

Given a finitely presented group  $G$  and a subgroup  $H$  having finite index in  $G$ , return a group  $R$  isomorphic to  $H$  with a presentation on (some of) the Schreier generators of  $H$  in  $G$ . The group  $R$  will be created as a subgroup of  $G$  and defining relations of  $R$  on its generators will be available. Note that the generators of  $R$  will, in general, not correspond to the generators of  $H$ . The isomorphism from  $H$  onto  $R$  is returned as second return value.

This function may require the computation of a coset table. Experienced users can control the behaviour of a possibly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function `SetGlobalTCPParameters`. For a detailed description of the available parameters and their meanings, we refer to Chapter 71.

<code>Simplify</code>	BOOLELT	<i>Default : true</i>
-----------------------	---------	-----------------------

If this Boolean-valued parameter is given the value `true`, then the resulting presentation for  $H$  will be simplified (default). The function `Rewrite` returns a finitely presented group that is isomorphic to  $H$ . If simplification is requested (by setting `Simplify := true`) then the simplification procedures are invoked (see next section). These procedures perform a sequence of Tietze transformations which typically result in a considerable simplification of the presentation produced by the rewriting process. Alternatively, the user can set `Simplify := false` and then perform the simplification directly if desired. (See next section). If simplification is not requested as part of `Rewrite`, a small amount of simplification is performed on the presentation before it is returned.

<code>EliminationLimit</code>	RNGINTELT	<i>Default : 100</i>
<code>ExpandLimit</code>	RNGINTELT	<i>Default : 150</i>
<code>GeneratorsLimit</code>	RNGINTELT	<i>Default : 0</i>

<b>LengthLimit</b>	RNGINTELT	<i>Default : <math>\infty</math></i>
<b>SaveLimit</b>	RNGINTELT	<i>Default : 10</i>
<b>SearchSimultaneous</b>	RNGINTELT	<i>Default : 20</i>
<b>Iterations</b>	RNGINTELT	<i>Default : 10000</i>
<b>Print</b>	RNGINTELT	<i>Default : 0</i>

These parameters control the simplification. See the description of **Simplify** for an explanation of these parameters.

**Rewrite**( $G, \sim H : parameters$ )

Given a finitely presented group  $G$  and a subgroup  $H$  having finite index in  $G$ , compute a defining set of relations for  $H$  on the existing generators, using extended coset enumeration and Reidemeister-Schreier rewriting, and change the presentation of  $H$  accordingly.

If the computation is successful, defining relations for  $H$  on its generators will be available at the end; any previously computed relations of  $H$  will be discarded. If the computation is unsuccessful,  $H$  is not changed. In any case, both the isomorphism type of  $H$  and its embedding into  $G$  as a subgroup is preserved by this function.

<b>Simplify</b>	BOOLELT	<i>Default : true</i>
-----------------	---------	-----------------------

If this parameter is given the value **true** (default), then an attempt will be made to simplify the constructed set of relations by substring searches, that is, Tietze transformations not changing any generators. The generating set of  $H$  is not modified by this process.

Moreover, the extended coset enumeration can be controlled by a wide range of parameters. The function **Rewrite** – in addition to the parameter **Simplify** – accepts the same parameters as the function **CosetEnumerationProcess** described in Chapter 71.

### Example H70E48

---

Starting with the group  $G$  defined by

$$\langle x, y \mid x^2, y^3, (xy)^{12}, (xy)^6(xy^{-1})^6 \rangle,$$

we construct a subgroup  $K$  of index 3 generated by the words  $x$ ,  $xyx^{-1}$  and  $yxxy^{-1}xy^{-1}xy$ . We present the subgroup  $K$ , compute its abelian quotient structure and then show that the class 30 2-quotient of  $K$  has order  $2^{62}$ .

```
> G<x, y> := Group< x, y | x^2, y^3, (x*y)^12, (x*y)^6*(x*y^-1)^6 >;
> G;
```

Finitely presented group G on 2 generators

Relations

```
x^2 = Id(G)
y^3 = Id(G)
(x * y)^12 = Id(G)
x * y * x * y * x * y * x * y * x * y * x * y * x * y * x * y^-1 * x *
```

```

      y^-1 * x * y^-1 * x * y^-1 * x * y^-1 * x * y^-1 = Id(G)
> K := sub< G | x, y*x*y^-1, y*x*y^-1*x*y^-1*x*y >;
> K;
Finitely presented group K on 3 generators
Generators as words in group G
      K.1 = x
      K.2 = y * x * y^-1
      K.3 = y * x * y^-1 * x * y^-1 * x * y
> Index(G, K);
3
> T := Rewrite(G, K);
> T;
Finitely presented group T on 3 generators
Generators as words in group G
      T.1 = x
      T.2 = y * x * y^-1
      T.3 = x^y
Relations
      T.1^2 = Id(T)
      T.2^2 = Id(T)
      T.3^2 = Id(T)
      (T.3 * T.2 * T.1 * T.3 * T.2)^2 = Id(T)
      (T.1 * T.3 * T.2 * T.1 * T.3)^2 = Id(T)
      (T.1 * T.2 * T.1 * T.3 * T.2)^2 = Id(T)
> AbelianQuotientInvariants(T);
[ 2, 2, 2 ]
> Q2 := pQuotient(T, 2, 30);
> FactoredOrder(Q2);
[ <2, 62> ]

```

### Example H70E49

---

In this example we illustrate how the function `Rewrite` can be used to obtain a presentation of a finitely presented group on a different set of generators.

We start with a presentation of  $L_2(7)$  on two generators  $x$  and  $y$ .

```

> F<x,y> := Group< x, y | x^3 = 1, y^3 = 1, (x*y)^4 = 1,
>          (y*y^x)^2 = y^x*y >;

```

The group is also generated by the elements  $a = (xy)^2$  and  $b = y$ .

```

> H<a,b> := sub<F | (x*y)^2, y >;
> Index(F,H);
1

```

At the moment, no defining relations of  $H \cong F$  on the generators  $a$  and  $b$  are known.

```

> H;
Finitely presented group H on 2 generators

```

```

Index in group F is 1
Generators as words in group F
  a = (x * y)^2
  b = y

```

We apply the function `Rewrite` to  $H$  as a subgroup of  $F$  in order to compute defining relations on the generators  $a$  and  $b$ .

```

> Rewrite(F, ~H);
> H;
Finitely presented group H on 2 generators
Index in group F is 1
Generators as words in group F
  a = (x * y)^2
  b = y
Relations
  a^2 = Id(H)
  b^3 = Id(H)
  (a * b)^7 = Id(H)
  (a * b^-1 * a * b)^4 = Id(H)
  (b * a * b^-1 * a * b * a)^4 = Id(H)

```

The last relation turns out to be redundant;  $a$  and  $b$  are standard generators for  $L_2(7)$ .

```

> Order(DeleteRelation(H,5)) eq Order(H);
true

```

## 70.7 Subgroups of Finite Index

The functions in this section are concerned with the construction of subgroups of finite index. We first describe a method for computing all subgroups whose index does not exceed some (modest) integer bound. The next family of functions are concerned with constructing new subgroups of finite index from one or more known ones.

### 70.7.1 Low Index Subgroups

`LowIndexSubgroups(G, R : parameters)`

Given a finitely presented group  $G$  (possibly the free group), and an expression  $R$  defining a positive integer range (see below), determine the conjugacy classes of subgroups of  $G$  whose indices lie in the range specified by  $R$ . The subgroups are generated by systematically building all coset tables consistent with the defining relations for  $G$  and which satisfy the range condition  $R$ . The argument  $R$  is one of the following:

- (a) An integer  $n$  representing the range  $[1, n]$ ;
- (b) A tuple  $\langle a, b \rangle$  representing the range  $[a, b]$ ;

The generation of subgroups can be controlled by a set of parameters described below. The returned sequence contains the subgroups found and is sorted in order of increasing index in  $G$ .

### Example H70E50

---

(Peter Lorimer) The two graphs known as Tutte's 8-cage and the Conder graph may be constructed as the Cayley graphs of two conjugacy classes of subgroups having index 10 in the finitely presented group

$$\langle q, r, s, h, a \mid h^3 = a^2 = p^2 = 1, p^h = p, p^a = q, q^h = r, r^a = s, \\ h^s = h^{-1}, r^h = pqr, sr = pqr s, pq = qp, pr = rp, ps = sp, qr = rq, qs = sq \rangle.$$

We use the low index function to construct these subgroups.

```
> G<p, q, r, s, h, a> := Group<p, q, r, s, h, a |
>                               h^3 = a^2 = p^2 = 1, p^h = p, p^a = q,
>                               q^h = r, r^a = s, h^s = h^-1, r^h = p * q * r,
>                               s * r = p * q * r * s, p * q = q * p,
>                               p * r = r * p, p * s = s * p, q * r = r * q,
>                               q * s = s * q>;
> LowIndexSubgroups(G, <10, 10>);
[
  Finitely presented group on 6 generators
  Index in group G is 10 = 2 * 5
  Generators as words in group G
    $.1 = p
    $.2 = s
    $.3 = h
    $.4 = q^-2
    $.5 = a * h^-1 * a * r^-1
    $.6 = a * h * a * h^-1 * a * q^-1,

  Finitely presented group on 6 generators
  Index in group G is 10 = 2 * 5
  Generators as words in group G
    $.1 = p
    $.2 = s
    $.3 = h
    $.4 = q^-2
    $.5 = a * h^-1 * a * r^-1
    $.6 = a * h * a * h^-1 * a
]
```

**Example H70E51**

---

A fairly surprising application for the low index subgroup algorithm is the enumeration of the conjugacy classes of a finite fp-group. In this example, we consider the group

$$G \simeq \text{PGL}_2(9) = \langle a, b \mid a^2, b^3, (ab)^8, [a, b]^5, [a, (ba)^3 b^{-1}]^2 \rangle.$$

```
> G<a,b> := Group< a,b | a^2, b^3, (a*b)^8, (a,b)^5,
>                (a, (b*a)^3*b^-1)^2 >;
> Order(G);
720
```

In an infinite fp-group, finding all classes of subgroups up to an index of 720 by applying the low index subgroup algorithm, would be extremely hard. In the case of the finite group  $G$ , however, we succeed.

```
> time sgG := LowIndexSubgroups(G, Order(G));
Time: 31.859
> #sgG;
26
```

We get a list of 26 representatives of the conjugacy classes of subgroups. For every representative, its index in  $G$  and a set of generating words are known. We just have a look at two of them.

```
> sgG[10];
Finitely presented group on 2 generators
Index in group G is 60 = 2^2 * 3 * 5
Generators as words in group G
$.1 = b
$.2 = a * b * a * b * a * b^-1 * a * b * a * b * a * b^-1 * a
      * b^-1 * a
> sgG[21];
Finitely presented group on 1 generator
Index in group G is 180 = 2^2 * 3^2 * 5
Generators as words in group G
$.1 = (b * a)^2
```

---

The function `LowIndexSubgroups` constructs all conjugacy classes of subgroups of  $G$  satisfying the following two conditions:

- (i) The index of each subgroup is in the range defined by  $R$ ;
- (ii) If the parameter `Subgroup` defines a subgroup  $H$ , then at least one subgroup in each conjugacy class contains the subgroup  $H$ .

The subgroups are returned as a sequence of subgroups  $G$ , unless otherwise specified by the parameter `GeneratingSets` (see below). The sequence is sorted by increasing index of the subgroups in  $G$ .

The subgroups are constructed using an algorithm due to Sims [Sim94, sect. 5.6]. This algorithm constructs the coset tables by using a backtrack algorithm. At a given position in the coset table, coset definitions are made systematically. Once a new definition has been made, the group relations are traced in an attempt to deduce further entries or to infer that this partial table will not extend to a table corresponding to a new class of subgroups. When either it cannot define a new entry, or when a complete table has been constructed, the algorithm backtracks to try the next possibility (this may introduce a new row, increasing the index). This algorithm may also be run as a process in such a way that the subgroups are returned one at a time, thereby allowing the user to analyze each subgroup as soon as it is found.

**ColumnMajor**                      **BOOLELT**                      *Default : false*

If **ColumnMajor** is set **false** (*default*), then the location for a new definition in the coset table is determined by searching the table in row major order for undefined entries. If **ColumnMajor** is set **true**, then the position for a new definition is determined by searching the table in column major order. If the presentation for  $G$  contains explicit relators expressing the fact that certain of the generators have large order, then the presentation should be organized so that these generators appear first and the column major order should be selected for new coset definition. This strategy often leads to greatly improved performance.

**GeneratingSets**                      **BOOLELT**                      *Default : false*

The conjugacy classes of subgroups are returned in the form of a sequence of sets of words, where the  $i$ -th set is a generating set for a representative subgroup from the  $i$ -th conjugacy class of subgroups satisfying the given conditions. This is a much more compact representation than returning the subgroups as a sequence of actual *subgroups* of  $G$  and should be used when a very large number of subgroups is expected, as there may be insufficient space to store each of them as a subgroup.

**Limit**                                      **RNGINTELT**                      *Default :  $\infty$*

Terminate after finding  $n$  conjugacy classes of subgroups satisfying the designated conditions.

**Long**                                      [ **RNGINTELT** ]                      *Default : []*

This option enables the user to designate certain of the defining relators for  $G$  as *long relators*. The relators of  $G$  are numbered from 1 to  $r$ , in the order they appear in the **quo-** or **Group-**constructors. The value  $L$  of **Long** is a subset of the integer set  $\{1, \dots, r\}$ . MAGMA interprets the relators whose numbers appear in  $L$  as long relators. A relator designated as long is not used during the construction of a coset table. Rather, it is applied once a complete table has been found. There is some evidence to suggest that better performance is achieved in those groups having one or more very long relators by deferring application of these relators until such time as a complete coset table has been obtained.

**Print**                                      **RNGINTELT**                      *Default : 0*

A description of each class of subgroups may be printed immediately after it is constructed. The value  $n$  assigned to the `Print` parameter specifies just what information is to be printed, according to the following rules:

$n = 0$  : No printing (*default*).

$n = 1$  : For each class, print a heading and a set of generators for the class representative.

$n = 2$  : The information printed for  $n = 1$ , together with the permutation representation of  $G$  on the right cosets of the class representative.

$n = 3$  : The information printed for  $n = 2$ , together with generators for the normalizer  $N$  of the class representative, and a system of right coset representatives for  $N$  in  $G$ .

<code>Subgroup</code>	<code>GRPFP</code>	<i>Default</i> : <code>sub&lt; G   &gt;</code>
-----------------------	--------------------	--

By specifying a value  $H$  for `Subgroup`, only subgroups containing  $H$  will be constructed.

<code>TimeLimit</code>	<code>RNGINTELT</code>	<i>Default</i> : 0 (no limit)
------------------------	------------------------	-------------------------------

A time limit in seconds. A value of 0 (default) means no limit.

<code>LowIndexProcess(G, R : parameters)</code>
---

<code>ColumnMajor</code>	<code>BOOLELT</code>	<i>Default</i> : <code>false</code>
<code>GeneratingSets</code>	<code>BOOLELT</code>	<i>Default</i> : <code>false</code>
<code>Long</code>	[ <code>RNGINTELT</code> ]	<i>Default</i> : []
<code>Print</code>	<code>RNGINTELT</code>	<i>Default</i> : 0
<code>Subgroup</code>	<code>GRPFP</code>	<i>Default</i> : <code>sub&lt; G   &gt;</code>
<code>TimeLimit</code>	<code>RNGINTELT</code>	<i>Default</i> : 0 (no limit)

Create a low index subgroups *process*. This process may be used to create the conjugacy classes of proper subgroups one at a time, with control being handed back to the MAGMA language processor each time a new class of subgroups is found. This function returns a process which is used by the function `NextSubgroup` to actually produce the subgroups.

The arguments and parameters have the same interpretation as for the function `LowIndexSubgroups`, except that `Limit` is not available (since the same effect can be achieved by limiting the number of calls to `NextSubgroup`).

Setting a time limit for a process  $P$  limits the total amount of time spent in calls to `NextSubgroup`. If the time limit is exceeded in a call to `NextSubgroup`, this call is aborted and  $P$  becomes invalid. Further attempts to access  $P$  will cause a runtime error. The function `IsValid` can be used to check whether a process is valid in order to avoid runtime errors in loops or user written functions.

`NextSubgroup( $\sim P$ )`

`NextSubgroup( $\sim P$ ,  $\sim G$ )`

Given a low index subgroups process  $P$ , construct the next conjugacy class of proper subgroups. The process  $P$  must have been previously created using the function `LowIndexProcess` and must be valid. Calling `NextSubgroup` for an empty process has no effect.

`ExtractGroup( $P$ )`

Extract a representative subgroup for the conjugacy class currently defined by the low index process  $P$ . The subgroup extracted will be the one found by the previous invocation of `NextSubgroup`, or the first subgroup if `NextSubgroup` has never been invoked on this process. Note that `ExtractGroup` will not search for a new subgroup. If  $P$  is empty or invalid, a runtime error will result.

`ExtractGenerators( $P$ )`

Extract a generating set for the representative subgroup of the conjugacy class currently defined by the low index process  $P$ . The subgroup extracted will be the one found by the previous invocation of `NextSubgroup`, or the first subgroup if `NextSubgroup` has never been invoked on this process. Note that `ExtractGenerators` will not search for a new subgroup. If  $P$  is empty or invalid, a runtime error will result.

`IsEmpty( $P$ )`

Return `true` if the low index process  $P$  has already found all conjugacy classes of subgroups. If `IsEmpty` is called immediately following the creation of the low index process, then it will return the value `false` if there are no subgroups satisfying the specified conditions or advance  $P$  to the first such subgroup otherwise.

`IsValid( $P$ )`

Return `true` if the low index process  $P$  is valid, that is, no limit has been exceeded. If `IsValid` is called immediately following the creation of the low index process, then it will return the value `false` if no subgroups satisfying the specified conditions can be found within the specified time or advance  $P$  to the first such subgroup otherwise.

### Example H70E52

---

We determine all conjugacy classes of subgroups having index at most 15 in the triangle group

$$\langle a, b \mid a^2, b^3, (ab)^7 \rangle .$$

```
> G<a, b> := Group< a, b | a^2, b^3, (a*b)^7 >;
> L := LowIndexSubgroups(G, 15: Print := 1);
Subgroup class 1      Index 7 Length 7      Subgroup generators :-
{ a, b * a * b^-1, b^-1 * a * b * a * b^-1 * a * b }
```

Subgroup class 2            Index 7 Length 7            Subgroup generators :-  
{ a,  $b^{-1} * a * b^{-1} * a * b * a * b$ ,  $b * a * b^{-1}$  }

Subgroup class 3            Index 15            Length 15            Subgroup generators :-  
{ a,  $b^{-1} * a * b * a * b^{-1} * a * b * a * b^{-1} * a * b$ ,  $b * a * b^{-1}$  }

Subgroup class 4            Index 15            Length 15            Subgroup generators :-  
{ a,  $b * a * b^{-1}$ ,  $b^{-1} * a * b^{-1} * a * b * a * b^{-1} * a * b * a * b$  }

Subgroup class 5            Index 14            Length 14            Subgroup generators :-  
{ a,  $b^{-1} * a * b * a * b^{-1} * a * b * a * b * a * b^{-1} * a * b$ ,  $b * a * b^{-1}$  }

Subgroup class 6            Index 14            Length 7            Subgroup generators :-  
{ a,  $b^{-1} * a * b * a * b^{-1} * a * b * a * b^{-1} * a * b * a * b^{-1} * a * b$ ,  $b * a * b * a * b^{-1}$  }

Subgroup class 7            Index 14            Length 14            Subgroup generators :-  
{ a,  $b^{-1} * a * b * a * b^{-1} * a * b^{-1} * a * b * a * b * a * b^{-1} * a * b$ ,  $b * a * b * a * b^{-1}$  }

Subgroup class 8            Index 14            Length 7            Subgroup generators :-  
{ a,  $b * a * b * a * b^{-1} * a * b^{-1} * a * b * a * b * a * b^{-1} * a * b^{-1}$ ,  $b^{-1} * a * b * a * b$  }

Subgroup class 9            Index 15            Length 15            Subgroup generators :-  
{ a,  $b * a * b * a * b^{-1} * a * b^{-1}$ ,  $b^{-1} * a * b^{-1} * a * b * a * b$  }

Subgroup class 10           Index 9 Length 9            Subgroup generators :-  
{ a,  $b * a * b * a * b * a * b^{-1}$  }

Subgroup class 11           Index 14            Length 7            Subgroup generators :-  
{ a,  $b * a * b * a * b * a * b^{-1} * a * b$ ,  $b * a * b^{-1} * a * b * a * b^{-1}$  }

Subgroup class 12           Index 14            Length 7            Subgroup generators :-  
{ a,  $b * a * b * a * b^{-1} * a * b * a * b$ ,  $b * a * b^{-1} * a * b * a * b^{-1}$  }

Subgroup class 13           Index 14            Length 7            Subgroup generators :-  
{ a,  $b^{-1} * a * b * a * b^{-1} * a * b$ ,  $b * a * b * a * b * a * b^{-1} * a * b^{-1}$  }

Subgroup class 14           Index 14            Length 7            Subgroup generators :-  
{ a,  $b * a * b * a * b^{-1} * a * b * a * b$ ,  $b^{-1} * a * b * a * b^{-1} * a * b$  }

Subgroup class 15           Index 14            Length 14            Subgroup generators :-  
{ a,  $b^{-1} * a * b * a * b * a * b^{-1} * a * b$ ,  $b * a * b * a * b * a * b^{-1} * a * b^{-1}$  }

Subgroup class 16           Index 8 Length 8            Subgroup generators :-  
{ b,  $a * b * a * b * a * b^{-1} * a$  }

**Example H70E53**

---

In this example we illustrate the use of the low index subgroup process by using it to determine whether the simple group  $\text{PSL}(2,8)$  is a homomorphic image of the triangle group

$$\langle x, y \mid x^2, y^3, (xy)^7 \rangle.$$

```
> F<x, y> := FreeGroup(2);
> G<x, y> := quo< F | x^2, y^3, (x*y)^7 >;
> LP := LowIndexProcess(G, 30);
> i := 0;
> while i le 100 and not IsEmpty(LP) do
>   H := ExtractGroup(LP);
>   NextSubgroup(~LP);
>   P := CosetImage(G, H);
>   if Order(P) eq 504 and IsSimple(P) then
>     Ψprint "The group G has L(2, 8) as a homomorphic image.";
>     print "It is afforded by the subgroup:-", H;
>     Ψbreak;
>   end if;
>   i += 1;
> end while;
```

The group G has L(2, 8) as a homomorphic image.

It is afforded by the subgroup:-

Finitely presented group H on 4 generators

Index in group G is 28 = 2<sup>2</sup> \* 7

Generators as words in group G

```
H.1 = x
H.2 = y * x * y^-1
H.3 = y^-1 * x * y * x * y^-1 * x * y * x * y^-1 * x * y * x * y^-1 *
x * y * x * y
H.4 = y^-1 * x * y * x * y^-1 * x * y^-1 * x * y * x * y^-1 * x * y *
x * y * x * y^-1 * x * y
```

**Example H70E54**

---

This example shows how the low index subgroup machinery may be used as part of a function trying to prove that a group is infinite:

```
> function MyIsInfinite(G)
>
> // ...
>
> // Low index subgroup approach: check whether an obviously
> //   infinite subgroup can be found in reasonable time.
> P := LowIndexProcess(G, 30 : TimeLimit := 5);
> while IsValid(P) and not IsEmpty(P) do
>   H := ExtractGroup(P);
```

```

> NextSubgroup(~P);
> if 0 in AbelianQuotientInvariants(H) then
>   print "The group G has subgroup:-", H;
>   print "whose abelian quotient is infinite";
>   print "Hence G is infinite.";
>   return true;
> end if;
> end while;
> print "Low index approach fails; trying other methods...";
>
> // ...
>
> end function;

```

We try the code fragment on the group

$$\langle x, z \mid z^3 x z^3 x^{-1}, z^5 x^2 z^2 x^2 \rangle .$$

```

> G<x, z> := Group<x, z | z^3*x*z^3*x^-1, z^5*x^2*z^2*x^2 >;
> MyIsInfinite(G);
The group G has subgroup:-
Finitely presented group H on 4 generators
Index in group G is 4 = 2^2
Generators as words in group G
  H.1 = x
  H.2 = z * x * z
  H.3 = z^3
  H.4 = z * x^-1 * z * x * z^-1
whose abelian quotient has structure [ 2, 6, 0 ]
Hence G is infinite.
true

```

---

**LowIndexNormalSubgroups(G, n: *parameters*)**

The normal subgroups of finitely presented group  $G$  up to index  $n$ ,  $n \leq 100\,000$ . The subgroups are returned as a sequence of records (ordered by subgroup index) where the  $i$ th record contains fields

**Group:** A presentation of the  $i$ th normal subgroup.

**Index:** The index of the  $i$ th normal subgroup in  $G$ .

**Supegroups:** The set of positions in the sequence of the groups which are supergroups of the  $i$ th group.

**PrintLevel**

RINGINTELT

*Default* : 0

This parameter may be set to 0, 1 or 2. At 0, the function prints no diagnostic output. At level 1, it outputs details of each normal subgroup being tested for further normal subgroups. Level 2 gives details of each test being performed on each normal subgroup.

**Simplify**

MONSTGELT

Default : "No"

The possible values are "No", "Yes" and "LengthLimit". This determines the parameter values passed to the `Rewrite(G,H)` function, when this function is used. The value "No" sets parameter `Simplify:=false`. The value "Yes" sets parameter `Simplify:=true`. The value "LengthLimit" sets parameter `LengthLimit:=Index(G,H)`.

## 70.7.2 Subgroup Constructions

Most operations described in this subsection require a closed coset table for at least one subgroup of an fp-group. If a closed coset table is needed and has not been computed, a coset enumeration will be invoked. If the coset enumeration does not produce a closed coset table, a runtime error is reported.

Experienced users can control the behaviour of such indirectly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function `SetGlobalTCPParameters`. For a detailed description of the available parameters and their meanings, we refer to Chapter 71.

`H ~ u``Conjugate(H, u)`

Given an fp-group  $H$  and a word  $u$  in an fp-group  $K$ , such that  $H$  and  $K$  are subgroups of some common fp-group  $G$  and words in terms of the generators of  $G$  are known for the generators of both  $H$  and  $K$ , construct the subgroup of  $G$  obtained by conjugating  $H$  by  $u$ .

`H meet K`

Given subgroups  $H$  and  $K$ , both of finite index in some fp-group  $G$ , return the subgroup which is the intersection of  $H$  and  $K$ .

This function requires closed coset tables for both,  $H$  and  $K$  in  $G$ .

`Core(G, H)`

Given a subgroup  $H$  of finite index in the fp-group  $G$ , construct the core of  $H$  in  $G$ .

This function requires a closed coset table for  $H$  in  $G$ .

`GeneratingWords(G, H)`

Given a subgroup  $H$  of the fp-group  $G$ , this function returns a set of words in the generators of  $G$ , generating  $H$  as a subgroup of  $G$  (assuming such words are known or can be constructed). Note that the returned generating set does not necessarily correspond to the internal generators of  $H$ . In particular, generating words obtained using the function `GeneratingWords` cannot be used to coerce elements from  $H$  to  $G$ .

`MaximalOvergroup(G, H)`

Given a subgroup  $H$  of finite index in the fp-group  $G$ , construct a maximal overgroup of  $H$  in  $G$ . A *maximal overgroup* of  $H$  is a maximal subgroup of  $G$  that contains  $H$ . If  $H$  is already maximal, the group  $G$  is returned.

This function requires a closed coset table for  $H$  in  $G$ .

`MinimalOvergroup(G, H)`

Given a subgroup  $H$  of finite index in the fp-group  $G$ , construct a minimal overgroup of  $H$  in  $G$ . A *minimal overgroup* of a subgroup  $H$  is a subgroup  $K$  of  $G$  such that  $K$  contains  $H$  as a maximal subgroup. If  $H$  is already maximal in  $G$ , the group  $G$  is returned.

This function requires a closed coset table for  $H$  in  $G$ .

`H ~ G`

`NormalClosure(G, H)`

Given a subgroup  $H$  of finite index in the fp-group  $G$ , construct the normal closure of  $H$  in  $G$ .

This function requires a closed coset table for  $H$  in  $G$ .

`Normaliser(G, H)`

`Normalizer(G, H)`

Given a subgroup  $H$  of finite index in the fp-group  $G$ , construct the normaliser of  $H$  in  $G$ . For a sample application of this function, see Example H70E47.

This function requires a closed coset table for  $H$  in  $G$ .

`SchreierGenerators(G, H : parameters)`

`Simplify`

BOOLELT

*Default : true*

Given a subgroup  $H$  of finite index in the fp-group  $G$ , return the Schreier generators for  $H$  as a set of words in  $G$ .

If the parameter `Simplify` is set to `true` (default), a heuristic method of eliminating redundant Schreier generators is applied. To switch this feature off, set `Simplify` to `false`.

This function requires a closed coset table for  $H$  in  $G$ .

`SchreierSystem(G, H)`

`Transversal(G, H)`

Given a subgroup  $H$  of finite index in the fp-group  $G$ , construct a (right) Schreier system of coset representatives for  $H$  in  $G$ . The function returns

- (a) the Schreier system as a set of words in  $G$ ;
- (b) the corresponding Schreier coset function.

This function requires a closed coset table for  $H$  in  $G$ .

Transversal( $G, H, K$ )
--------------------------

Given subgroups  $H$  and  $K$ , both of finite index in the fp-group  $G$ , return an indexed set of words which comprise a set of representatives for the double cosets  $HuK$  of  $H$  and  $K$  in  $G$ , as well as a map from  $G$  to the representatives. It should be noted that this function is evaluated by first constructing the right cosets of  $H$  in  $G$  and then computing the orbits of the cosets under the action of the generators of the subgroup  $K$ .

This function requires a closed coset table for  $H$  in  $G$ .

**Example H70E55**

We illustrate some of the subgroup constructions by using them to construct subgroups of small index in the two-dimensional space group  $p4g$  which has the presentation

$$\langle r, s \mid r^2, s^4, (r, s)^2 \rangle.$$

```
> p4g<r, s> := Group< r, s | r^2 = s^4 = (r*s^-1*r*s)^2 = 1 >;
> p4g;
```

Finitely presented group  $p4g$  on 2 generators

Relations

```
  r^2 = Id(p4g)
  s^4 = Id(p4g)
  (r * s^-1 * r * s)^2 = Id(p4g)
```

We define two subgroups of  $p4g$  and compute their indices in  $p4g$ .

```
> h := sub< p4g | (s^-1*r)^4, s*r >;
> k := sub< p4g | (s^-1*r)^2, (s*r)^2 >;
> Index(p4g, h);
8
> Index(p4g, k);
8
```

We construct the normal closure of  $h$  in  $p4g$ .

```
> n := NormalClosure(p4g, h);
> n;
Finitely presented group n on 6 generators
Index in group p4g is 2
Generators as words in group p4g
```

```
  n.1 = (s^-1 * r)^4
  n.2 = s * r
  n.3 = r * s
  n.4 = r^-1 * s * r^2
  n.5 = s^2 * r * s^-1
  n.6 = r * s
```

Next, we construct a subgroup of  $p4g$  containing  $h$  as maximal subgroup...

```
> m := MinimalOvergroup(p4g, h);
```

```

> m;
Finitely presented group m on 3 generators
Index in group p4g is 4 = 2^2
Generators as words in group p4g
  m.1 = (s^-1 * r)^4
  m.2 = s * r
  m.3 = (r * s)^2

```

... and a maximal subgroup of  $p4g$  containing  $k$ .

```

> n := MaximalOvergroup(p4g, k);
> n;
Finitely presented group n on 4 generators
Index in group p4g is 2
Generators as words in group p4g
  n.1 = (s^-1 * r)^2
  n.2 = (s * r)^2
  n.3 = r
  n.4 = s^2

```

Finally, we construct a transversal in  $p4g$  for the normaliser of  $h$  in  $p4g$ ...

```

> T := Transversal(p4g, Normaliser(p4g, h));
> T;
{@ Id(p4g), r, s^-1, r * s @}

```

... compute the intersection of  $h$  and the conjugate of  $h$  by  $r$ ...

```

> l := h meet h^r;
> l;
Finitely presented group l
Index in group p4g is 32 = 2^5
Subgroup of group p4g defined by coset table

```

... and construct the core of  $h$  in  $p4g$ .

```

> c := Core(p4g, h);
> c;
Finitely presented group c
Index in group p4g is 32 = 2^5
Subgroup of group p4g defined by coset table

```

Note, that the two subgroups  $l$  and  $c$  constructed last are defined as finite index subgroups of  $p4g$  by a coset table and that there are no generators known for them. Generators can be obtained e.g. by using the function `GeneratingWords`. We show this for the subgroup  $l$ .

```

> GeneratingWords(p4g, l);
{ (s * r)^4, (s^-1 * r)^4 }

```

Once computed, these generators are memorised. Compare the result of printing  $l$  to the output obtained above.

```

> l;

```

```

Finitely presented group 1 on 2 generators
Index in group p4g is 32 = 2^5
Generators as words in group p4g
  1.1 = (s * r)^4
  1.2 = (s^-1 * r)^4

```

### Example H70E56

---

Consider the group  $G$  given by the presentation  $\langle x, y \mid x^2, y^3, (xy)^7 \rangle$ .

```
> G<x,y> := Group< x,y | x^2, y^3, (x*y)^7 >;
```

We construct the subgroups of index less than or equal to 7 using the low index algorithm.

```

> L := LowIndexSubgroups(G, 7);
> L;
[
  Finitely presented group on 2 generators
  Index in group G is 1
  Generators as words in group G
    $.1 = x
    $.2 = y,
  Finitely presented group on 3 generators
  Index in group G is 7
  Generators as words in group G
    $.1 = x
    $.2 = y * x * y^-1
    $.3 = y^-1 * x * y^-1 * x * y * x * y,
  Finitely presented group on 3 generators
  Index in group G is 7
  Generators as words in group G
    $.1 = x
    $.2 = y * x * y^-1
    $.3 = y^-1 * x * y * x * y^-1 * x * y
]

```

We define a subgroup as the core of one of the subgroups of index 7. The function `Core` returns a subgroup of  $G$  defined by a coset table.

```

> H := Core(G, L[2]);
> H;
Finitely presented group H
Index in group G is 168 = 2^3 * 3 * 7
Subgroup of group G defined by coset table

```

A set of generators for  $H$  can be obtained e.g. with the function `SchreierGenerators`.

```

> sgH := SchreierGenerators(G, H);
> #sgH;

```

6

By default, `SchreierGenerators` returns a reduced generating set. The unreduced set of Schreier generators can be obtained by setting the value of the parameter `Simplify` to `false`.

```
> sgHu := SchreierGenerators(G, H : Simplify := false);
> #sgHu;
85
```

---

### 70.7.3 Properties of Subgroups

The operations described in this subsection all require a closed coset table for at least one subgroup of an fp-group. If a closed coset table is needed and has not been computed, a coset enumeration will be invoked. If the coset enumeration does not produce a closed coset table, a runtime error is reported.

Experienced users can control the behaviour of such indirectly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function `SetGlobalTCPParameters`. For a detailed description of the available parameters and their meanings, we refer to Chapter 71.

**u in H**

Given an fp-group  $H$  and a word  $u$  in an fp-group  $K$ , such that  $H$  and  $K$  are subgroups of some common fp-group  $G$ ,  $H$  is of finite index in  $G$ , and words for the generators of  $K$  in terms of the generators of  $G$  are known, return `true` if  $u$  is an element of  $H$  and `false` otherwise.

This function requires a closed coset table for  $H$  in  $G$ .

**u notin H**

Given an fp-group  $H$  and a word  $u$  in an fp-group  $K$ , such that  $H$  and  $K$  are subgroups of some common fp-group  $G$ ,  $H$  is of finite index in  $G$ , and words for the generators of  $K$  in terms of the generators of  $G$  are known, return `true` if  $u$  is not an element of  $H$  and `false` otherwise.

This function requires a closed coset table for  $H$  in  $G$ .

**H eq K**

Given subgroups  $H$  and  $K$ , both of finite index in the fp-group  $G$ , return `true` if  $H$  and  $K$  are equal and `false` otherwise.

This function may require closed coset tables for both,  $H$  and  $K$  in  $G$ .

**H ne K**

Given subgroups  $H$  and  $K$ , both of finite index in the fp-group  $G$ , return `true` if  $H$  and  $K$  are not equal and `false` otherwise.

This function may require closed coset tables for both,  $H$  and  $K$  in  $G$ .

**H subset K**

Given subgroups  $H$  and  $K$ , both of finite index in the fp-group  $G$ , return **true** if  $H$  is contained in  $K$  and **false** otherwise.

This function requires a closed coset table for  $K$  in  $G$ .

**H notsubset K**

Given subgroups  $H$  and  $K$ , both of finite index in the fp-group  $G$ , return **true** if  $H$  is not contained in  $K$  and **false** otherwise.

This function requires a closed coset table for  $K$  in  $G$ .

**IsConjugate(G, H, K)**

Given subgroups  $H$  and  $K$ , both of finite index in the fp-group  $G$ , return **true** if  $H$  and  $K$  are conjugate subgroups of  $G$  and **false** otherwise. If  $H$  and  $K$  are conjugate in  $G$ , a conjugating element is returned as second return value.

This function requires a closed coset table for both,  $H$  and  $K$  in  $G$ .

**IsNormal(G, H)**

Given a subgroup  $H$  of finite index in the fp-group  $G$ , return **true** if  $H$  is a normal subgroup of  $G$  and **false** otherwise.

This function requires a closed coset table for  $H$  in  $G$ .

**IsMaximal(G, H)**

Given a subgroup  $H$  of finite index in the fp-group  $G$ , return **true** if  $H$  is a maximal subgroup of  $G$  and **false** otherwise.

This function requires a closed coset table for  $H$  in  $G$ .

**IsSelfNormalizing(G, H)**

Given a subgroup  $H$  of finite index in the fp-group  $G$ , return **true** if  $H$  is a self-normalizing subgroup of  $G$  and **false** otherwise. For a sample application of this function, see Example H70E47.

This function requires a closed coset table for  $H$  in  $G$ .

### Example H70E57

---

We illustrate some of the subgroup predicates by applying them to some subgroups of the two-dimensional space group  $p4g = \langle r, s \mid r^2, s^4, (r, s)^2 \rangle$  from Example H70E55.

```
> p4g<r, s> := Group<r, s | r^2 = s^4 = (r*s^-1*r*s)^2 = 1 >;
> h := sub<p4g | (s^-1*r)^4, s*r >;
> k := sub<p4g | (s^-1*r)^2, (s*r)^2 >;
```

$h$  and  $k$  have the same index in  $p4g$ ...

```
> Index(p4g, h);
8
> Index(p4g, k);
```

8

... but they are not equal.

```
> h eq k;
false
```

We check for normality of  $h$  and  $k$  in  $p4g$ .

```
> IsNormal(p4g, h);
false
> IsNormal(p4g, k);
true
```

We construct the normal closure of  $h$  in  $p4g$ .

```
> n := NormalClosure(p4g, h);
```

We see that it is maximal...

```
> IsMaximal(p4g, n);
true
```

... and that it contains  $k$ .

```
> k subset n;
true
```

We define another subgroup of  $p4g$ .

```
> l := sub< p4g | (s*r)^4, s^-1*r >;
```

In fact, it is conjugate to  $h$ .

```
> IsConjugate(p4g, h, l);
true r^-1
```

I.e.  $l = h^{r^{-1}}$ . The intersection of  $h$  and  $h^{r^{-1}}$  already yields the core of  $h$  in  $p4g$ .

```
> h meet l eq Core(p4g, h);
true
```

### Example H70E58

---

The constructions of the previous section together with the Boolean function `IsMaximal` may be used to locate large maximal subgroups in a finite group. Consider the Hall-Janko group  $J_2$ , which may be defined by the presentation

$$\langle a, b, c \mid a^3, b^3, c^3, abab^{-1}a^{-1}b^{-1}, (ca)^5, (cb)^5, \\ (cb^{-1}cb)^2, a^{-1}baca^{-1}bac^{-1}a^{-1}b^{-1}ac^{-1}, \\ aba^{-1}caba^{-1}c^{-1}ab^{-1}a^{-1}c^{-1} \rangle.$$

We examine subgroups generated by pairs of randomly chosen short words. Whenever we obtain a proper subgroup, if it is not already maximal we replace it by a maximal subgroup that contains it.

```
> J2<a, b, c> := Group<a, b, c | a^3, b^3, c^3, a*b*a*b^-1*a^-1*b^-1, (c*a)^5,
>                                     (c*b)^5, (c*b^-1*c*b)^2,
>                                     a^-1*b*a*c*a^-1*b*a*c^-1*a^-1*b^-1*a*c^-1,
>                                     a*b*a^-1*c*a*b*a^-1*c^-1*a*b^-1*a^-1*c^-1>;
>
> Seen := { 0, 1};
> Found := { };
> Sgs := [ ];
> for i := 1 to 30 do
>   u := Random(J2, 1, 1);
>   v := Random(J2, 3, 5);
>   H := sub< J2 | u, v >;
>   Indx := Index(J2, H);
>   if Indx notin Seen then
>     Include(~Seen, Indx);
>     if not IsMaximal(J2, H) then
>       H := MaximalOvergroup(J2, H);
>     end if;
>     if Indx notin Found then
>       Include(~Sgs, H);
>       Include(~Seen, Indx);
>       Include(~Found, Indx);
>     end if;
>   end if;
> end for;
> Sgs;
[
```

Finitely presented group on 3 generators

Index in group J2 is 315 =  $3^2 * 5 * 7$

Generators as words in group J2

\$.1 =  $b^{-1}$

\$.2 =  $a^{-2} * c * a^{-1}$

\$.3 =  $c$ ,

Finitely presented group on 3 generators

Index in group J2 is 1008 =  $2^4 * 3^2 * 7$

Generators as words in group J2

\$.1 =  $b^{-1}$

\$.2 =  $c^{-1} * a^{-1} * c^{-1} * b$

\$.3 =  $a * c * b * c^{-1} * a^{-1} * c * b * c^{-1}$ ,

Finitely presented group on 3 generators

Index in group J2 is 100 =  $2^2 * 5^2$

Generators as words in group J2

```
$.1 = c
$.2 = (b * a^-1)^2
$.3 = a * b^-1
```

]

Thus after taking 30 2-generator random subgroups, we have obtained three maximal subgroups, including the two largest maximal subgroups.

---

## 70.8 Coset Spaces and Tables

Let  $G = \langle X | R \rangle$  be a finitely presented group. Suppose that  $H$  is a subgroup of  $G$  having finite index  $m$ . Let  $V = \{c_1 (= H), c_2, \dots, c_m\}$  denote the set of distinct right cosets of  $H$  in  $G$ . This set admits a natural  $G$ -action

$$f : V \times G \rightarrow V$$

where

$$f : \langle c_i, x \rangle = c_k \iff c_i * x = c_k,$$

for  $c_i \in V$  and  $x \in G$ . The set  $V$  together with this action is a  $G$ -set called a *right coset space* for  $H$  in  $G$ . The action may also be represented using a coset table  $T$ .

If certain of the products  $c_i * x$  are unknown, the corresponding images under  $f$  are undefined. In this case,  $T$  is not closed, and  $V$  is called an *incomplete coset space* for  $H$  in  $G$ .

### 70.8.1 Coset Tables

A coset table is represented in MAGMA as a mapping. Given a finitely-presented group  $G$  and a subgroup  $H$ , the corresponding (right) coset table is a mapping  $f : \{1, \dots, r\} \times G \rightarrow \{0, \dots, r\}$ , where  $r$  is the index of  $H$  in  $G$ .  $f(i, x)$  is the coset to which coset  $i$  is mapped under the action of  $x \in G$ . The value 0 is only included in the codomain if the coset table is not closed, and it denotes that the coset is not known.

<code>CosetTable(G, H: parameters)</code>
---

The (right) coset table for  $G$  over subgroup  $H$ , constructed by means of the Todd-Coxeter procedure. If the coset table does not close then the codomain will include the value 0.

Experienced users can control the Todd-Coxeter procedure invoked by this function with a wide range of parameters. This function accepts the same parameters as the function `CosetEnumerationProcess` described in Chapter 71.

<code>CosetTableToRepresentation(G, T)</code>
---

Given a coset table  $T$  for a subgroup  $H$  of  $G$ , construct the permutation representation of  $G$  given by its action on the cosets of  $H$ , using the columns of  $T$ . The function returns:

- (a) The homomorphism  $f : G \rightarrow P$ ;
- (b) The permutation group image  $P$ ;
- (c) The kernel  $K$  of the action (a subgroup of  $G$ ).

<code>CosetTableToPermutationGroup(G, T)</code>
---

Given a coset table  $T$  for a subgroup  $H$  of  $G$ , construct the permutation group image of  $G$  given by its action on the cosets of  $H$ , using the columns of  $T$ . This is the second return value of `CosetTableToRepresentation(G, T)`.

---

**Example H70E59**

Consider the infinite dihedral group.

```
> G<a,b> := DihedralGroup(GrpFP, 0);
> G;
Finitely presented group G on 2 generators
Relations
  b^2 = Id(G)
  (a * b)^2 = Id(G)
```

We define a subgroup  $S$  of  $G$  and compute the coset table map for  $S$  in  $G$ .

```
> S := sub<G|a*b, a^10>;
> ct := CosetTable(G, S);
> ct;
Mapping from: Cartesian Product<{ 1 .. 10 }, GrpFP: G>
to { 1 .. 10 }
  $1  $2 -$1
1.   2   2   3
2.   4   1   1
3.   1   4   5
4.   6   3   2
5.   3   6   7
6.   8   5   4
7.   5   8   9
8.  10   7   6
9.   7  10  10
10.  9   9   8
```

When printing the coset table, the action of the generators and of the non-trivial inverses of generators on the enumerated transversal is shown in table form.

Using the coset table, we now construct the permutation representation of  $G$  on the cosets of  $S$  in  $G$ . We assign the representation (a homomorphism), the image (a permutation group of degree  $[G : S] = 10$ ) and the kernel of the permutation representation (a subgroup of  $G$ ).

```
> fP, P, K := CosetTableToRepresentation(G, ct);
> fP;
Homomorphism of GrpFP: G into GrpPerm: P, Degree 10,
Order 2^2 * 5 induced by
  a |--> (1, 2, 4, 6, 8, 10, 9, 7, 5, 3)
  b |--> (1, 2)(3, 4)(5, 6)(7, 8)(9, 10)
```

Note that the images of  $a$  and  $b$  correspond to the first two columns of the printed coset table above.

```
> P;
Permutation group P acting on a set of cardinality 10
Order = 20 = 2^2 * 5
  (1, 2, 4, 6, 8, 10, 9, 7, 5, 3)
  (1, 2)(3, 4)(5, 6)(7, 8)(9, 10)
> K;
Finitely presented group K
Index in group G is 20 = 2^2 * 5
Subgroup of group G defined by coset table
```

Now, we define a subgroup of infinite index in  $G$  and compute a coset table for it.

```
> H := sub<G|b>;
> ct := CosetTable(G, H);
```

Of course, the coset table cannot be complete; note that 0 is in its codomain, indicating unknown images of cosets.

```
> ct;
Mapping from: Cartesian Product<{ 1 .. 1333331 }, GrpFP: G>
to { 0 .. 1333331 }
```

### 70.8.2 Coset Spaces: Construction

The indexed (right) coset space  $V$  of the subgroup  $H$  of the group  $G$  is a  $G$ -set consisting of the set of integers  $\{1, \dots, m\}$ , where  $i$  represents the right coset  $c_i$  of  $H$  in  $G$ . The action of  $G$  on this  $G$ -set is that induced by the natural  $G$ -action

$$f : V \times G \rightarrow V$$

where

$$f : \langle c_i, x \rangle = c_k \iff c_i * x = c_k,$$

for  $c_i \in V$  and  $x \in G$ . If certain of the products  $c_i * x$  are unknown, the corresponding images under  $f$  are undefined, and  $V$  is called an *incomplete coset space* for  $H$  in  $G$ .

`CosetSpace(G, H: parameters)`

This function attempts to construct a coset space for the subgroup  $H$  in the group  $G$  by means of the Todd-Coxeter procedure. If the enumeration fails to complete, the function returns an incomplete coset space. The coset space is returned as an indexed right coset space. For a sample application of this function see Example H70E61.

Experienced users can control the Todd-Coxeter procedure invoked by this function with a wide range of parameters. This function accepts the same parameters as the function `CosetEnumerationProcess` described in Chapter 71.

`RightCosetSpace(G, H: parameters)`

`LeftCosetSpace(G, H: parameters)`

The explicit right coset space of the subgroup  $H$  of the group  $G$  is a  $G$ -set containing the set of right cosets of  $H$  in  $G$ . The elements of this  $G$ -set are the pairs  $\langle H, x \rangle$ , where  $x$  runs through a transversal for  $H$  in  $G$ . Similarly, the explicit left coset space of  $H$  is a  $G$ -set containing the set of left cosets of  $H$  in  $G$ , represented as the pairs  $\langle x, H \rangle$ . These functions use the Todd-Coxeter procedure to construct the explicit right (left) coset space of the subgroup  $H$  of the group  $G$ . For a sample application see Example H70E61.

Experienced users can control the Todd-Coxeter procedure invoked by these functions with a wide range of parameters. Both functions accept the same parameters as the function `CosetEnumerationProcess` described in Chapter 71.

### 70.8.3 Coset Spaces: Elementary Operations

`H * g`

Right coset of the subgroup  $H$  of the group  $G$ , where  $g$  is an element of  $G$  (as an element of the right coset of  $H$ ).

`C * g`

Coset to which the right coset  $C$  of the group  $G$  is mapped by the (right) action of  $g$ , where  $g$  is an element of  $G$ .

`C * D`

and  $D$ .

Given two right cosets of the same normal subgroup  $H$  of the group  $G$ , return the right coset that is the product of  $C$  and  $D$ .

`g in C`

Return `true` if element  $g$  of group  $G$  lies in the coset  $C$ .

`g notin C`

Return `true` if element  $g$  of group  $G$  does not lie in the coset  $C$ .

`C1 eq C2`

Returns `true` if the coset  $C1$  is equal to the coset  $C2$ .

`C1 ne C2`

Returns `true` if the coset  $C1$  is not equal to the coset  $C2$ .

#### 70.8.4 Accessing Information

`#V`

The cardinality of the coset space  $V$ .

`Action(V)`

The mapping  $V \times G \rightarrow V$  giving the action of  $G$  on the coset space  $V$ . This mapping is a coset table.

`<i, w> @ T`

`T(i, w)`

The image of coset  $i$  as defined in the coset table  $T$ , under the action of word  $w$ .

`ExplicitCoset(V, i)`

The element of the explicit coset space that corresponds to indexed coset  $i$ .

`IndexedCoset(V, w)`

The element of the indexed coset space  $V$  to which the element  $w$  of  $G$  corresponds.

`IndexedCoset(V, C)`

The element of the indexed coset space  $V$  to which the explicit coset  $C$  of  $G$  corresponds.

`Group(V)`

The group  $G$  for which  $V$  is a coset space.

`Subgroup(V)`

The subgroup  $H$  of  $G$  such that  $V$  is a coset space for  $G$  over  $H$ .

`IsComplete(V)`

Returns `true` if the coset space  $V$  is complete.

ExcludedConjugates(V)
-----------------------

ExcludedConjugates(T)
-----------------------

Given a partial or complete coset space  $V$  for the group  $G$  over the subgroup  $H$ , or a coset table  $T$  corresponding to this coset space, this function returns the set of words  $E = \{g_i^{-1}h_jg_i \mid g_i \text{ a generator of } G, h_j \text{ a generator of } H, \text{ and, modulo } V, g_i^{-1}h_jg_i \text{ does not lie in } H\}$ . If  $E$  is empty, then  $H$  is a normal subgroup of  $G$ , while if  $E$  is non-empty, the addition of the elements of  $E$  to the generators of  $H$  will usually be a larger subgroup of the normal closure of  $H$ . This function may be used with incomplete coset spaces for  $G$  over  $H$ ; it may then happen that some of the elements of  $E$  actually lie in  $H$  but there is insufficient information for this to be detected. This function is normally used in conjunction with the Todd-Coxeter algorithm when seeking some subgroup having index sufficiently small so that the Todd-Coxeter procedure completes. The conjugates are returned as a set of words.

Transversal(G, H)
-------------------

RightTransversal(G, H)
------------------------

Given a finitely presented group  $G$  and a subgroup  $H$  of  $G$ , this function returns

- A set of elements  $T$  of  $G$  forming a right transversal for  $G$  over  $H$ ; and
- The corresponding transversal mapping  $\phi : G \rightarrow T$ . If  $T = [t_1, \dots, t_r]$  and  $g \in G$ ,  $\phi$  is defined by  $\phi(g) = t_i$ , where  $g \in H * t_i$ .

These functions may require the computation of a coset table. Experienced users can control the behaviour of a possibly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function `SetGlobalTCPParameters`. For a detailed description of the available parameters and their meanings, we refer to Chapter 71.

---

### Example H70E60

Consider the infinite dihedral group.

```
> G<a,b> := DihedralGroup(GrpFP, 0);
```

We define a subgroup  $H$  of index 10 in  $G$ ...

```
> H := sub< G | a*b, a^10 >;
```

```
> Index(G, H);
```

```
10
```

... and construct a right transversal for  $H$  in  $G$  and the associate transversal map.

```
> RT, transmap := Transversal(G, H);
```

```
> RT;
```

```
{@ Id(G), a, a^-1, a^2, a^-2, a^3, a^-3, a^4, a^-4, a^5 @}
> transmap;
```

```
Mapping from: GrpFP: G to SetIndx: RT
```

From this a left transversal is easily obtained:

```
> LT := {@ x^-1 : x in RT @};
```

```
> LT;
{@ Id(G), a^-1, a, a^-2, a^2, a^-3, a^3, a^-4, a^4, a^-5 @}

```

We construct the coset table and check whether the enumeration of the cosets is compatible to the enumeration of the right transversal  $RT$ .

```
> ct := CosetTable(G, H);
> forall(culprit){ i : i in [1..Index(G, H)]
>                 | ct(1, RT[i]) eq i};
true

```

It is. Thus, we can very easily define a function  $RT \times G \rightarrow RT$ , describing the action of  $G$  on the set of right cosets of  $H$  in  $G$ .

```
> action := func< r, g | RT[ct(Index(RT, r), g)] >;

```

Note that the definition of the function `action` relies on the fact that the computed right transversal and its enumeration match the ones internally used for the coset table `ct`.

```
> action(Id(G), b);
a

```

I.e.  $H * b = H * a$ .

```
> action(a^-4, a*b);
a^4

```

I.e.  $Ha^{-4} * (ab) = Ha^4$ .

### Example H70E61

---

Consider the group  $G = \langle a, b \mid a^8, b^7, (ab)^2, (a^{-1}b)^3 \rangle$  and the subgroup  $H$  of  $G$ , generated by  $a^2$  and  $a^{-1}b$ .

```
> F<x, y> := FreeGroup(2);
> G<a, b> := quo< F | x^8, y^7, (x*y)^2, (x^-1*y)^3 >;
> H := sub< G | a^2, a^-1*b >;

```

We construct an indexed right coset space  $V$  and an explicit right coset space  $Vr$  for  $H$  in  $G$ .

```
> V := CosetSpace(G, H);
> Vr := RightCosetSpace(G, H);

```

The coset  $H$  always has index 1.

```
> trivial_coset := ExplicitCoset(Vr, 1);
> trivial_coset;
<GrpFP: H, Id(G)>
> IndexedCoset(V, trivial_coset);
1

```

We now pick a coset...

```
> coset := ExplicitCoset(Vr, 42);
> coset;

```

```

<GrpFP: H, a^-1 * b^-1 * a^3 * b^-1>
... multiply from the right with b...
> coset * b;
<GrpFP: H, a^-1 * b^-1 * a^3>
... and check where this gets us in the indexed coset space V.
> IndexedCoset(V, coset * b);
23

```

### Example H70E62

---

We present a function which computes the derived subgroup  $G'$  for the finitely presented group  $G$ . It assumes that the Todd-Coxeter procedure can enumerate the coset space of  $G'$  in  $G$ .

```

> function DerSub(G)
>
> /* Initially define S to contain the commutator of each pair of distinct
> generators of G */
>
>   S := { (x,y) : x, y in Generators(G) | (x,y) ne Id(G) };
>
> /* successively extend S until it is closed under conjugation by the
> generators of G */
>
>   repeat
>     V := CosetSpace(G, sub<G | S>);
>     E := ExcludedConjugates(V);
>     S := S join E;
>   until # E eq 0;
>   return sub<G | S>;
> end function;

```

### Example H70E63

---

Given a subgroup  $H$  of the finitely presented group  $G$ , for which the Todd-Coxeter procedure does not complete, add excluded conjugates one at a time to the generators of  $G$  until a subgroup  $K$  is reached such that either  $K$  is normal in  $G$ , or  $K$  has sufficiently small index for the Todd-Coxeter method to complete. The set *hgens* contains a set of generating words for  $H$ .

```

> function NormClosure(G, hgens)
>   xgens := hgens;
>   kgens := hgens;
>   indx := 0;
>   while # xgens ne 0 do
>     Include(~kgens, Representative(xgens));
>     V := CosetSpace(G, sub<G | kgens>);
>     if IsComplete(V) then break; end if;

```

```

>      xgens := ExcludedConjugates(V);
> end while;
> if IsComplete(V) then
>     print "The subgroup generated by", kgens, "has index", #V;
>     return kgens;
> else
>     print "The construction was unsuccessful";
>     return {};
> end if;
> end function;

```

---

### 70.8.5 Double Coset Spaces: Construction

`DoubleCoset(G, H, g, K )`

The double coset  $H * g * K$  of the subgroups  $H$  and  $K$  of the group  $G$ , where  $g$  is an element of  $G$ .

`DoubleCosets(G, H, K)`

Set of double cosets  $H * g * K$  of the group  $G$ .

This function may require the computation of a coset table. Experienced users can control the behaviour of a possibly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function `SetGlobalTCPParameters`. For a detailed description of the available parameters and their meanings, we refer to Chapter 71.

#### Example H70E64

---

Consider again the infinite dihedral group  $G$ ...

```
> G<a,b> := DihedralGroup(GrpFP, 0);
```

... and the subgroup  $H$  of index 10 in  $G$ .

```
> H := sub< G | a*b, a^10 >;
```

The set of  $H$ - $H$  double cosets in  $G$  can be obtained with the statement

```

> DoubleCosets(G, H, H);
{ <GrpFP: H, Id(G), GrpFP: H>, <GrpFP: H, a^5, GrpFP: H>,
  <GrpFP: H, a^4, GrpFP: H>, <GrpFP: H, a^2, GrpFP: H>,
  <GrpFP: H, a, GrpFP: H>, <GrpFP: H, a^3, GrpFP: H> }

```

---

### 70.8.6 Coset Spaces: Selection of Cosets

<code>CosetsSatisfying(T, S: parameters)</code>
---

<code>CosetSatisfying(T, S: parameters)</code>
--

<code>CosetsSatisfying(V, S: parameters)</code>
---

<code>CosetSatisfying(V, S: parameters)</code>
--

Given a fp-group  $G$ , and a partial or complete coset space  $V$  or coset table  $T$  for  $G$  over the subgroup  $H$  generated by the set of words  $S$ , these functions return representatives for the cosets of  $V$  which satisfy the conditions defined in the parameters. In the description of the parameters below, the symbol  $l$  will denote a Boolean value, while the symbol  $n$  will denote a positive integer in the range  $[1, \#V]$ .

The functions are not identical. `CosetsSatisfying` returns a set of coset representatives for  $V$  as defined in the parameters. `CosetSatisfying` is the same as `CosetsSatisfying` with the `Limit` parameter equal to 1; thus it returns a set containing a single coset representative, or an empty set if no cosets satisfy the conditions.

<b>First</b>	RNGINTELT	<i>Default : 1</i>
Start looking for coset representatives satisfying the designated conditions beginning with coset $i$ of $V$ .		
<b>Last</b>	RNGINTELT	<i>Default : #V</i>
Stop looking for coset representatives after examining coset $j$ of $V$ .		
<b>Limit</b>	RNGINTELT	<i>Default : <math>\infty</math></i>
Terminate the search for coset representatives as soon as $l$ have been found which satisfy the designated conditions. This parameter is not available for <code>CosetSatisfying</code> , since <code>Limit</code> is set to 1 for this function.		
<b>Normalizing</b>	BOOLELT	<i>Default : false</i>
If <code>true</code> , select coset representatives $x$ such that, modulo $V$ , the word $x^{-1}h_1x, \dots, x_1h_sx$ is contained in $H$ .		
<b>Order</b>	RNGINTELT	<i>Default : 0</i>
Select coset representatives $x$ such that, modulo $V$ , the word $x^n$ is contained in $H$ .		
<b>Print</b>	RNGINTELT	<i>Default : 0</i>
If $t > 0$ , print the coset representatives found to satisfy the designated conditions.		

#### Example H70E65

---

Consider the braid group  $G$  on 4 strings with Artin generators  $a, b$  and  $c$  and the subgroup  $H$  of  $G$  generated by  $a$  and  $b$ .

```
> G<a,b,c> := BraidGroup(GrpFP, 4);
```

```
> H := sub< G | a,b >;
```

We construct an – obviously incomplete – explicit right coset space for  $H$  in  $G$ .

```
> V := RightCosetSpace(G, H);
```

Using the function `CosetSatisfying`, we compute an element of  $G$ , not contained in  $H$ , which normalises  $H$ .

```
> cs := CosetSatisfying(V, Generators(Subgroup(V))
>                               : Normalizing := true, First := 2);
> cs;
{
  <GrpFP: H, c * b * a^2 * b * c>
}
> rep := c * b * a^2 * b * c;
```

The conjugates of  $a$  and  $b$  by this element had better be in  $H$ ...

```
> rep^-1 * a * rep in H;
true
> rep^-1 * b * rep in H;
true
```

...OK.

### 70.8.7 Coset Spaces: Induced Homomorphism

`CosetAction(G, H)`

Given a subgroup  $H$  of the group  $G$ , this function constructs the permutation representation  $\phi$  of  $G$  given by the action of  $G$  on the cosets of  $H$ . It returns:

- (a) The homomorphism  $\phi$ ;
- (b) The image group  $\phi(G)$ .
- (c) (if possible) the kernel of  $\phi$ .

The permutation representation is obtained by using the Todd-Coxeter procedure to construct the coset table for  $H$  in  $G$ . Note that  $G$  may be an infinite group: it is only necessary that the index of  $H$  in  $G$  be finite.

`CosetAction(V)`

Construct the permutation representation  $L$  of  $G$  given by the action of  $G$  on the coset space  $V$ . It returns the permutation representation  $\phi$  (as a map) and its image.

`CosetImage(G, H)`

Construct the image of  $G$  given by its action on the (right) coset space of  $H$  in  $G$ .

**CosetImage(V)**

Construct the image of  $G$  as defined by its action on the coset space  $V$ .

**CosetKernel(G, H)**

The kernel of  $G$  in its action on the (right) coset space of  $H$  in  $G$ . (Only available when the index of  $H$  in  $G$  is very small).

This function may require the computation of a coset table. Experienced users can control the behaviour of a possibly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function **SetGlobalTCPParameters**. For a detailed description of the available parameters and their meanings, we refer to Chapter 71.

**CosetKernel(V)**

The kernel of  $G$  in its action on the (right) coset space  $V$ . (Only available when the index of the subgroup  $H$  of  $G$  defining the coset space is very small).

**Example H70E66**

The first Conway group has a representation as the image of the group

$$\begin{aligned}
 G = \langle a, b, c, d, e, f, g, h \mid & a^2, b^2, c^2, d^2, e^2, f^2, g^2, h^2, \\
 & (ab)^3, (ac)^2, (ad)^2, (ae)^4, (af)^2, (ag)^2, (ah)^3, \\
 & (bc)^5, (bd)^2, (be)^2, (bf)^2, (bg)^4, (bh)^4, \\
 & (cd)^3, (ce)^3, (cf)^4, (cg)^2, (ch)^2, \\
 & (de)^2, (df)^3, (dg)^2, (dh)^2, \\
 & (ef)^6, (eg)^2, (eh)^2, \\
 & (fg)^4, (fh)^6, \\
 & (gh)^2, \\
 & a(cf)^2 = (adfh)^3 = b(ef)^3 = (baefg)^3 = 1, \\
 & (cef)^7 = d(bh)^2 = d(aeh)^3 = e(bg)^2 = 1 \rangle
 \end{aligned}$$

under the homomorphism defined by the action of  $G$  on the cosets of the subgroup

$$H = \langle a, b, c, d, e, f, g, (adefcefg)^{39} \rangle.$$

The permutation group can be constructed as follows:

```

> F<s, t, u, v, w, x, y, z> := FreeGroup(8);
> G<a, b, c, d, e, f, g, h> := quo<F | s^2, t^2, u^2, v^2, w^2, x^2, y^2, z^2,
> (s*t)^3, (s*u)^2, (s*v)^2, (s*w)^4, (s*x)^2, (s*y)^2, (s*z)^3,
> (t*u)^5, (t*v)^2, (t*w)^2, (t*x)^2, (t*y)^4, (t*z)^4,
> (u*v)^3, (u*w)^3, (u*x)^4, (u*y)^2, (u*z)^2,
> (v*w)^2, (v*x)^3, (v*y)^2, (v*z)^2,

```

```

> (w*x)^6, (w*y)^2, (w*z)^2,
> (x*y)^4, (x*z)^6,
> (y*z)^2,
> s*(u*x)^2 = (s*v*x*z)^3 = t*(w*x)^3 = (t*s*w*x*y)^3 = 1,
> (u*w*x)^7 = v*(t*z)^2 = v*(s*w*z)^3 = w*(t*y)^2 = 1>;
> H := sub< G | a, b, c, d, e, f, g, (a*d*e*f*c*e*f*g*h)^39 >;
> V := CosetSpace(G, H: FillFactor := 100000);
> Co1 := CosetImage(V);
> Degree(Co1);
    98280

```

### Example H70E67

---

The group  $G_2(3)$  is a homomorph of the fp-group  $G$  defined below. We construct a permutation representation  $G1$  for  $G_2(3)$  on 351 points, and then compute the subgroup generated by the images of the first four generators of  $G$  in  $G1$ . (The functions applied to permutation groups are described in Chapter 58.)

```

> F<a,b,c,d,y,x,w> := FreeGroup(7);
> z := y*c*a^2*b;
> u := x*d;
> t := w*c*a*d*b^2*c;
> G<a,b,c,d,y,x,w>, g :=
>     quo< F | a^4, b^4, c^2, d^2, (a,b), (a*c)^2, (b*c)^2,
>             (c*d)^2, d*a*d*b^-1, y^3, (a^-1*b)^y*d*a^-1*b^-1,
>             (c*d*a^-1*b)^y*b^-1*a*d*c, a*d*y*d*a^-1*y, x^3,
>             a^x*b^-1, b^x*a*b, c^x*c, (x*d)^2, (u*z)^6, w^3,
>             (w,y), (a*b)^w*b^-1*a*d*c, (c*d*a^-1*b)^w*d*c*b^2,
>             (b^2*c*d)^w*a^-1*b^-1, (c*a^2*b*w)^2,
>             (a^-1*b)^w*d*a^-1*b^-1, (t*u)^3 >;
> z1 := g(z);
> u1 := g(u);
> t1 := g(t);
> H := sub< G | z1*a^2*b^2, u1*c, t1*a^2*b^2 >;
> f, G1, K := CosetAction(G, H);
> Degree(G1);
351
> print Order(G1), FactoredOrder(G1);
4245696 [ <2, 6>, <3, 6>, <7, 1>, <13, 1> ]
> CompositionFactors(G1);
    G
    | G(2, 3)
    1
> S := sub< G1 | f(a), f(b), f(c), f(d) >;
> BSGS(S);
> S;
Permutation group S of degree 351

```

Order = 64 =  $2^6$

Thus the images of a, b, c and d in  $G_1$  generate the Sylow 2-subgroup

---

## 70.9 Simplification

### 70.9.1 Reducing Generating Sets

Subgroups of finitely presented groups constructed in certain ways may be created with a generating set containing redundant generators. The most important case in which this situation may occur is a subgroup of the domain of a homomorphism  $f$  of groups, defined as the preimage under  $f$  of some given subgroup of the codomain of  $f$ . In this case, the generating set of the preimage will contain generators of the kernel of  $f$  and is likely to be not minimal.

Since reducing the generating set may be expensive and is not necessary in all situations, a reduction is not done automatically. Instead, MAGMA provides a function for reducing generating sets of finitely presented groups.

ReduceGenerators( $G$ )

Given a finitely presented group  $G$ , attempt to construct a presentation  $H$  on fewer generators.  $H$  is returned as a subgroup of  $G$  (which of course is equal to  $G$ ), so that element coerce is possible. The isomorphism from  $G$  to  $H$  is returned as second return value.

If a presentation for  $G$  is known, this function attempts to simplify this presentation (cf. section 70.9.2). Otherwise, it tries to rewrite  $G$  with respect to a suitable supergroup to obtain a presentation on fewer generators.

For a sample application of this function, see Example H70E73.

### 70.9.2 Tietze Transformations

Given a finitely presented group  $G$ , the user can attempt to simplify its presentation using Tietze transformations and substring searching. The choice of simplification strategy can either be left to MAGMA or selected by the user.

Simplify( $G$ : *parameters*)

Given a finitely presented group  $G$ , attempt to simplify the presentation of  $G$  by repeatedly eliminating generators and subsequently shortening relators by substitution of substrings that correspond to the left or right hand side of a relation. The order in which transformations are applied is determined by a set of heuristics. The transformation process terminates when no more eliminations of generators and no more length reducing substring replacements are possible.

A new group  $K$  isomorphic to  $G$  is returned which is (hopefully) defined by a simpler presentation than  $G$ .  $K$  is returned as a subgroup of  $G$ . The isomorphism  $f : G \rightarrow K$  is returned as second return value.

The simplification process can be controlled by a set of parameters described below.

### Example H70E68

---

Consider the Fibonacci group  $F(8)$ .

```
> F<x1, x2, x3, x4, x5, x6, x7, x8> := FreeGroup(8);
> F8<x1, x2, x3, x4, x5, x6, x7, x8> :=
>   quo< F | x1*x2=x3, x2*x3=x4, x3*x4=x5, x4*x5=x6,
>           x5*x6=x7, x6*x7=x8, x7*x8=x1, x8*x1=x2>;
```

We use the function `Simplify` in order to obtain a presentation of  $F(8)$  on two generators.

```
> H<[y]>, f := Simplify(F8);
> H;
Finitely presented group H on 2 generators
Generators as words in group F8
  y[1] = x3
  y[2] = x4
Relations
  y[2] * y[1]^-2 * y[2] * y[1]^-1 * y[2]^2 * y[1] * y[2]^2 *
  y[1]^-1 = Id(H)
  y[1] * y[2] * y[1] * y[2]^2 * y[1] * y[2] * y[1]^2 * y[2]^-1
  * y[1] = Id(H)
```

The isomorphism  $f$  can be used to express the old generators in terms of the new ones.

```
> f;
Mapping from: GrpFP: F8 to GrpFP: H
> f(x1);
y[1]^2 * y[2]^-1
```

---

The strategy employed by the function `Simplify` can be controlled using the following set of parameters.

**Preserve** [RNGINTELT] *Default* : []

This parameter can be used to indicate that certain generators of  $G$  should not be eliminated (*default*: no restrictions). **Preserve** is assigned a sequence of integers in the range  $[1, \dots, n]$ , where  $n$  is the number of generators of  $G$ , containing the numbers of those generators of  $G$  which should be preserved. See Example H70E70 for a sample application.

**Iterations** RNGINTELT *Default* : 10000

This parameter sets the maximal number of iterations of the main elimination / simplification cycle which will be performed.

**EliminationLimit** RNGINTELT *Default* : 100

This parameter sets the maximal number of generators which may be eliminated in each elimination phase.

**LengthLimit**                      RNGINTELT                      *Default* :  $\infty$

If **LengthLimit** is set to  $n$ , any eliminations which would make the total length of the relators grow beyond  $n$  will not be performed (*default*: no limit).

**ExpandLimit**                      RNGINTELT                      *Default* : 150

If **ExpandLimit** is set to  $n$ , the total length of the relators is not permitted to grow by more than a factor of  $n\%$  in any elimination phase (*default*: 150%). If this limit is reached, the elimination phase is aborted.

**GeneratorsLimit**                  RNGINTELT                      *Default* : 0

Any eliminations which would reduce the number of generators below the value of **GeneratorsLimit** will not be performed (*default*: 0).

**SaveLimit**                        RNGINTELT                      *Default* : 10

If **SaveLimit** is set to  $n$ , any simplification phase is repeated, if the reduction in the total length of the relators achieved during this phase exceeds  $n\%$  (*default*: 10%).

**SearchSimultaneous**              RNGINTELT                      *Default* : 20

This parameter sets the number of relators processed simultaneously in a simplification phase.

**Print**                                RNGINTELT                      *Default* : 0

This parameter controls the volume of printing. By default its value is that returned by `GetVerbose("Tietze")`, which is 0 unless it has been changed through use of `SetVerbose`.

#### `SimplifyLength(G: parameters)`

Given a finitely presented group  $G$ , attempt to eliminate generators and shorten relators by locating substrings that correspond to the left or right hand side of a relation. The order in which transformations are applied is determined by a set of heuristics. As opposed to the function `Simplify`, this function terminates the transformation process when the total length of the presentation starts to increase with the elimination of further generators.

A new group  $K$  isomorphic to  $G$  is returned which is (hopefully) defined by a simpler presentation than  $G$ .  $K$  is returned as a subgroup of  $G$ . The isomorphism  $f : G \rightarrow K$  is returned as second return value. This function accepts the same set of parameters as the function `Simplify`.

#### `TietzeProcess(G: parameters)`

Create a Tietze process that takes the presentation for the fp-group  $G$  as its starting point. This process may now be manipulated by various procedures that will be described below.

**Preserve**                            [RNGINTELT]                      *Default* : []

**Iterations**                        RNGINTELT                      *Default* : 10000

EliminationLimit	RNGINTELT	Default : 100
LengthLimit	RNGINTELT	Default : $\infty$
ExpandLimit	RNGINTELT	Default : 150
GeneratorsLimit	RNGINTELT	Default : 0
SaveLimit	RNGINTELT	Default : 10
SearchSimultaneous	RNGINTELT	Default : 20
Print	RNGINTELT	Default : 0

These parameters define the defaults used for the Tietze operations. Each of the various procedures described below allows some or all of these control parameters to be overridden.

For the meanings of the parameters, see the description under `Simplify` above.

`ShowOptions( $\sim P$  : parameters)`

Display the defaults associated with the Tietze process  $P$ . The current status of all the control parameters may be viewed by using this function.

`SetOptions( $\sim P$  : parameters)`

Change the defaults associated with the Tietze process  $P$ . All of the control parameters may be overridden permanently by using this function.

`Simplify( $\sim P$  : parameters)`

`SimplifyPresentation( $\sim P$  : parameters)`

Use the default strategy to simplify the presentation as much as possible. The transformation process is terminated when no more eliminations of generators and no more length reducing substring replacements are possible. All the control parameters may be overridden for this function.

`SimplifyLength( $\sim P$  : parameters)`

Use the default strategy to simplify the presentation as much as possible. The transformation process is terminated when the total length of the presentation starts to increase with the elimination of further generators. All the control parameters may be overridden for this function.

`Eliminate( $\sim P$ : parameters)`

`EliminateGenerators( $\sim P$ : parameters)`

Eliminate generators in the presentation defined by the Tietze process  $P$  under the control of the parameters. First any relators of length one are used to eliminate trivial generators. Then, if there are any non-involutory relators of length two, the generator with the higher number is eliminated. Of the control parameters, only `EliminationLimit`, `ExpandLimit`, `GeneratorsLimit` and `LengthLimit` may be overridden by this function.

**Relator**

RNGINTELT

*Default : 0*

If  $n > 0$ , try to eliminate a generator using the  $n$ -th relator. If no generator is specified by the parameter **Generator** below, then the generator which is eliminated will be the one occurring once in the relator that produced the smallest total relator length.

**Generator**

RNGINTELT

*Default : 0*

If  $n > 0$ , try to eliminate the  $n$ -th generator. If no relation is specified by the parameter **Relator** above, then the shortest relator in which the  $n$ -th generator occurs exactly once (if any) is used.

**Search( $\sim P$ : *parameters*)**

Simplifies the presentation by repeatedly searching for common substrings in pairs of relators where the length of the substring is greater than half the length of the shorter relator and making the corresponding transformations. Relators of length 1 or 2 are also used to generate simplifications. The control parameters **SaveLimit** and **SearchSimultaneous** can be overridden.

**SearchEqual( $\sim P$ : *parameters*)**

Modifies the presentation by repeatedly searching for common substrings in pairs of relators where the length of the substring is exactly half the length of the shorter relator and making the corresponding transformations. The control parameter **SearchSimultaneous** can be overridden.

**Group( $P$ )**

Extract the group  $G$  defined by the current presentation associated with the Tietze process  $P$ , together with the isomorphism between the original group and  $G$ .  $G$  is returned as a subgroup of the original group underlying  $P$ .

**NumberOfGenerators( $P$ )****Ngens( $P$ )**

The number of generators for the presentation currently stored by the Tietze process  $P$ .

**NumberOfRelations( $P$ )****Nrels( $P$ )**

The number of relations in the presentation currently stored by the Tietze process  $P$ .

**PresentationLength( $P$ )**

The sum of the lengths of the relators in the presentation currently stored by the Tietze process  $P$ .

**Example H70E69**

The Fibonacci group  $F(n)$  is generated by  $\{x_1, \dots, x_n\}$  with defining relations

$$x_i x_{i+1} = x_{i+2}, i \in \{1, \dots, n\},$$

where the subscripts are taken modulo  $n$ . Consider the Fibonacci group  $F(7)$ , which is defined in terms of the presentation

$$\langle x_1, x_2, x_3, x_4, x_5, x_6, x_7 \mid x_1 x_2 = x_3, x_2 x_3 = x_4, x_3 x_4 = x_5, \\ x_4 x_5 = x_6, x_5 x_6 = x_7, x_6 x_7 = x_1, x_7 x_1 = x_2 \rangle.$$

The following code will produce a 2-generator, 2-relator presentation for  $F(7)$ :

```
> F<x1, x2, x3, x4, x5, x6, x7> := FreeGroup(7);
> F7<x1, x2, x3, x4, x5, x6, x7> :=
>   quo< F | x1*x2=x3, x2*x3=x4, x3*x4=x5, x4*x5=x6,
>         x5*x6=x7, x6*x7=x1, x7*x1=x2 >;
> P := TietzeProcess(F7);
> for i := 7 to 3 by -1 do
>   Eliminate(~P: Generator := i);
> end for;
> Search(~P);
> H<x, y>, f := Group(P);
> H;
```

Finitely presented group H on 2 generators

Generators as words in group F7

```
x = x1
y = x2
```

Relations

```
x^-1 * y^-1 * x^-1 * y^-2 * x^-1 * y * x^-1 * y * x^-1 = Id(H)
x * y^3 * x^-1 * y * x^-1 * y^2 * x * y * x * y^-1 = Id(H)
```

The resulting presentation is

$$\langle a, b \mid a^{-1} b^{-1} a^{-1} b^{-2} a^{-1} b a^{-1} b a^{-1}, a b^3 a^{-1} b a^{-1} b^2 a b a b^{-1} \rangle.$$

We can use the isomorphism  $f$  returned by the function `Group` to express arbitrary words in the original generators of  $F(7)$  in terms of the new generators  $x$  and  $y$ :

```
> f;
Mapping from: GrpFP: F7 to GrpFP: H
> f(x7);
x * y^2 * x * y^2 * x * y * x * y^2 * x * y
```

Alternatively, a similar effect may be obtained using the `Simplify` function:

```
> F<x1, x2, x3, x4, x5, x6, x7> := FreeGroup(7);
> F7<x1, x2, x3, x4, x5, x6, x7> :=
>   quo< F | x1*x2=x3, x2*x3=x4, x3*x4=x5, x4*x5=x6,
```

```

>      x5*x6=x7, x6*x7=x1, x7*x1=x2>;
> H<x, y>, f := Simplify(F7: Iterations := 5);
> H;
Finitely presented group H on 2 generators
Generators as words in group F7
  x = x2
  y = x3
Relations
  x * y^-1 * x * y^2 * x * y * x^2 * y^-1 = Id(H)
  y * x * y^2 * x^-1 * y * x^-2 * y * x^-2 = Id(H)

```

Again, we can use the isomorphism  $f$  returned by the function `Simplify` to express arbitrary words in the original generators of  $F(7)$  in terms of the new generators  $x$  and  $y$ :

```

> f;
Mapping from: GrpFP: F7 to GrpFP: H
> f(x7);
y * x * y * x * y^2 * x * y

```

---

### Example H70E70

In a situation where some proper subset  $S$  of the original generating set of a finitely group  $G$  is sufficient to generate  $G$ , the function `Simplify` can also be used to rewrite words in the original generators in terms of the elements of  $S$ . Consider again one of the Fibonacci groups, say  $F(8)$ .

```

> F<x1, x2, x3, x4, x5, x6, x7, x8> := FreeGroup(8);
> F8<x1, x2, x3, x4, x5, x6, x7, x8> :=
>   quo< F | x1*x2=x3, x2*x3=x4, x3*x4=x5, x4*x5=x6,
>     x5*x6=x7, x6*x7=x8, x7*x8=x1, x8*x1=x2>;

```

Obviously,  $F(8)$  is generated by  $x_1$  and  $x_2$ . We utilise the function `Simplify` to obtain a presentation  $H$  of  $F(8)$  on  $x_1$  and  $x_2$ , using the parameter `Preserve` to indicate that  $x_1$  and  $x_2$  – i.e. the first and the second generator – are to be retained in the new presentation. We also compute the isomorphism  $f: F(8) \rightarrow H$ .

```

> H<x, y>, f := Simplify(F8: Preserve := [1,2]);

```

Mapping elements of  $F(8)$  to  $H$  using the map  $f$  basically means to rewrite these elements in terms of the generators  $x = x_1$  and  $y = x_2$ . Since  $H$  is returned as a subgroup of  $F(8)$ , the resulting words can be coerced from  $H$  back into  $F(8)$ , yielding words explicitly in  $x_1$  and  $x_2$ .

```

> F8 ! f(x5*x6);
x1 * x2^2 * x1 * x2 * x1^2 * x2^-1 * x1 * x2^-1

```

---

### Example H70E71

The finiteness of the last of the Fibonacci groups,  $F(9)$ , was settled in 1988 by M.F. Newman using the following result:

**Theorem.** Let  $G$  be a group with a finite presentation on  $b$  generators and  $r$  relations, and suppose  $p$  is an odd prime. Let  $d$  denote the rank of the elementary abelian group  $G_1 = [G, G]G^p$  and let  $e$  denote the rank of  $G_2 = [G_1, G]G^p$ . If

$$r - b < d^2/2 - d/2 - d - e$$

or

$$r - b \leq d^2/2 - d/2 - d - e + (e + d/2 - d^2/4)d/2,$$

then  $G$  has arbitrary large quotients of  $p$ -power order.

We present a proof that  $F(9)$  is infinite using this result.

```
> Left := func< b, r | r - b >;
> Right := func< d, e | d^2 div 2 - d div 2 - d - e +
>           (e + d div 2 - d^2 div 4)*(d div 2) >;
>
>
> F< x1,x2,x3,x4,x5,x6,x7,x8,x9 > :=
>   ΨGroup< x1, x2, x3, x4, x5, x6, x7, x8, x9 |
>           x1*x2=x3, x2*x3=x4, x3*x4=x5, x4*x5=x6, x5*x6=x7,
>           x6*x7=x8, x7*x8=x9, x8*x9=x1, x9*x1=x2 >;
>
> F;
Finitely presented group F on 9 generators
Relations
  x1 * x2 = x3
  x2 * x3 = x4
  x3 * x4 = x5
  x4 * x5 = x6
  x5 * x6 = x7
  x6 * x7 = x8
  x7 * x8 = x9
  x8 * x9 = x1
  x9 * x1 = x2
> AbelianQuotientInvariants(F);
[ 2, 38 ]
```

Thus the nilpotent quotient of  $F$  is divisible by 2 and 19. We examine the 2- and 19-quotients of  $F$ .

```
> Q1 := pQuotient(F, 2, 0: Print := 1);
Class limit set to 127.
Lower exponent-2 central series for F
Group: F to lower exponent-2 central class 1 has order 2^2
Group: F to lower exponent-2 central class 2 has order 2^3
Group completed. Lower exponent-2 central class = 2, Order = 2^3
> Q2 := pQuotient(F, 19, 0: Print := 1);
Class limit set to 127.
Lower exponent-19 central series for F
```

Group: F to lower exponent-19 central class 1 has order  $19^1$   
 Group completed. Lower exponent-19 central class = 1, Order =  $19^1$

Thus, the nilpotent residual of  $F$  has index 152. We try to locate this subgroup. We first take a 2-generator presentation for  $F$ .

```
> G := Simplify(F);
> G;
Finitely presented group G on 2 generators
Generators as words in group F
  G.1 = x4
  G.2 = x5
Relations
  G.2 * G.1 * G.2 * G.1 * G.2^2 * G.1 * G.2^2 * G.1^-1 * G.2 * G.1^-2 * G.2 *
    G.1^-2 = Id(G)
  G.1 * G.2^2 * G.1 * G.2 * G.1^2 * G.2^-1 * G.1^2 * G.2^-1 * G.1 * G.2^-1 *
    G.1^2 * G.2^-1 * G.1 * G.2^-1 = Id(G)
> H := ncl< G | (G.1, G.2) >;
> H;
Finitely presented group H
Index in group G is 76 =  $2^2 * 19$ 
Subgroup of group G defined by coset table
```

We haven't got the full nilpotent residual yet, so we try again.

```
> H := ncl< G | (G.1*G.1, G.2) >;
> H;
Finitely presented group H
Index in group G is 152 =  $2^3 * 19$ 
Subgroup of group G defined by coset table
```

Now, we have it.

```
> AbelianQuotientInvariants(H);
[ 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 ]
```

The nilpotent residual  $H$  has a 5-quotient. We construct a presentation for  $H$  and then calculate  $d$  and  $e$  for its 5-quotient.

```
> K := Rewrite(G, H: Simplify := false);
> KP := pQuotientProcess(K, 5, 1);
> d := FactoredOrder(ExtractGroup(KP))[1][2];
> NextClass(~KP);
> e := FactoredOrder(ExtractGroup(KP))[1][2] - d;
> "D = ", d, "e = ", e;
d = 18 e = 81
> "Right hand side = ", Right(d, e);
Right hand side = 135
> "Left hand side = ", Left(Ngens(K), #Relations(K));
```

Left hand side = 151

Since `Left` is greater than `Right`, this presentation for  $H$  doesn't work. Thus we start eliminating generators.

```
> K := Simplify(K: Iterations := 1);
> "Left hand side = ", Left(Ngens(K), #Relations(K));
Left hand side = 136
> K := Simplify(K: Iterations := 1);
> "Left hand side = ", Left(Ngens(K), #Relations(K));
Left hand side = 123
```

Got it! By Newman's theorem,  $H$  has an infinite 5-quotient and so  $F$  must be infinite.

---

### Example H70E72

In this example, we consider a – quite unpleasant – presentation for some group  $G$ . In fact, it is a presentation for the group  $\text{PSL}_3(7):2$ , but we assume that we do not know this and want to compute the order of the finitely presented group  $G$  using coset enumeration.

We note in passing that the strategy outlined in this example is, together with other approaches, applied by the function `Order`.

```
> F<a,b,c> := FreeGroup(3);
> rels := [ a^4, b^2, (a,b)^3, (a*b)^8, ((a*b)^2*(a^-1*b)^3)^2,
> c^2, (c*a*c*a^2)^2, (c*a)^3*(c*a^-1)^3,
> c*a*b*c*a^-1*b*a^-1*c*a*b*c*a^2*a*b*a^-1,
> c*a*b*c*b*a*c*a*c*a^-1*b*c*b*a^-1*c*a^-1,
> c*a*b*a^-1*c*a*b*a^-1*c*a*b*a^-1*c*a*b*a^-1,
> c*b*a^2*b*c*b*c*a^2*c*b*c*b*a^2*b,
> c*a^2*c*b*a*c*b*a*c*b*a*c*a^-1*c*a*b*a^-1,
> c*a^-1*b*a*c*a^-1*b*a*c*b*a*b*a^2*b*a^-1*b,
> c*a*b*a^-1*b*c*b*a^-1*b*c*a^-1*b*a*b*a*c*b*c*b,
> c*a*c*b*a*b*c*a*c*b*a*b*c*a*c*b*a*b,
> c*b*a^-1*b*c*a^-1*c*a^-1*b*a*b*c*b*c*a^2*b*a*b*a^-1,
> c*b*a^-1*b*a*b*c*b*a^-1*b*a*b*c*b*a^-1*b*a*b,
> c*a^2*b*a^-1*b*c*b*c*b*a^-1*b*a*c*b*a^2*b*a^-1*b
> ];
> G<a,b,c> := quo< F | rels >;
```

As it happens, trying to determine the order of  $G$  by enumerating the cosets of the trivial subgroup is quite hard. – Even the predefined enumeration strategy "Hard" (cf. `ToddCoxeter` and `CosetEnumerationProcess`) does not give a finite result.

```
> time ToddCoxeter(G, sub<G|> : Strategy := "Hard");
0
Time: 199.620
```

Of course we could try to increase the workspace for the coset enumeration, but we decide to be more clever. Trying random words in the generators of  $G$ , we easily find some subgroup  $S$  of  $G$  with pretty small index in  $G$ .

```
> S := sub< G | b, c*a*c*b*a*b >;
```

```
> time Index(G, S);
114
Time: 0.120
```

We now have to compute the order of  $S$ . In order to be able to do this using coset enumeration, we have to construct a presentation for  $S$  by rewriting  $S$  w.r.t.  $G$ .

```
> time R := Rewrite(G, S : Simplify := false);
Time: 0.030
```

However, the presentation obtained by Reidemeister-Schreier rewriting without any simplification is not suitable for coset enumeration: it contains too many generators and its total length is quite high.

```
> NumberOfGenerators(R);
133
> PresentationLength(R);
14384
```

An enumeration with the predefined enumeration strategy "Hard" does not produce a finite result. (Note that in consideration of the high total relator length, we select a coset table based enumeration style; cf. `CosetEnumerationProcess` in Chapter 71.)

```
> time ToddCoxeter(R, sub<R|> : Strategy := "Hard", Style := "C");
0
Time: 4.330
```

On the other hand, simplifying the presentation by reducing the number of generators as much as possible is not a good idea either, since the total relator length grows massively.

```
> time Rs := Simplify(R);
Time: 43.900
> NumberOfGenerators(Rs);
3
> PresentationLength(Rs);
797701
```

Again, an enumeration with the predefined enumeration strategy "Hard" does not produce a finite result.

```
> time ToddCoxeter(Rs, sub<Rs|> : Strategy := "Hard", Style := "C");
0
Time: 22015.849
```

The best strategy is, to simplify the presentation obtained from the Reidemeister-Schreier procedure by eliminating generators until the total length of the relators starts to grow.

```
> time Rsl := SimplifyLength(R);
Time: 0.330
> NumberOfGenerators(Rsl);
48
> PresentationLength(Rsl);
```

7152

A coset enumeration for this presentation produces a finite result in a reasonable amount of time.

```
> time ToddCoxeter(Rsl, sub<Rsl|> : Strategy := "Hard", Style := "C");
32928
Time: 289.410
```

This finally proves that  $G$  is finite and has order  $32928 * 114 = 3753792$ .

---

## 70.10 Representation Theory

This section describes some functions for creating  $R[G]$ -modules for a finitely presented group  $G$ , which are unique for this category or have special properties when called for fp-groups. For a complete description of the functions available for creating and working with  $R[G]$ -modules we refer to chapter 89.

Note that the function `GModuleAction` can be used to extract the matrix representation associated to an  $R[G]$ -module.

All operations described in this subsection may require a closed coset table for at least one subgroup of an fp-group. If a closed coset table is needed and has not been computed, a coset enumeration will be invoked. If the coset enumeration does not produce a closed coset table, a runtime error is reported.

Experienced users can control the behaviour of such indirectly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function `SetGlobalTCPParameters`. For a detailed description of the available parameters and their meanings, we refer to Chapter 71.

<code>GModulePrimes(G, A)</code>
----------------------------------

Let  $G$  be a finitely presented group and  $A$  a normal subgroup of  $G$  of finite index. Given any prime  $p$ , the maximal  $p$ -elementary abelian quotient of  $A$  can be viewed as a  $\mathbf{F}_p[G]$ -module  $M_p$ . This function determines all primes  $p$  such that  $M_p$  is not trivial (i.e. zero-dimensional) and the dimensions of the corresponding modules  $M_p$ . The return value is a multiset  $S$ . If  $0 \notin S$ , the maximal abelian quotient of  $A$  is finite and the multiplicity of  $p$  is the dimension of  $M_p$ . If  $S$  contains 0 with multiplicity  $m$ , the maximal abelian quotient of  $A$  contains  $m$  copies of  $\mathbf{Z}$ . In this case,  $M_p$  is non-trivial for every prime  $p$ . The rank of  $M_p$  in this case is the sum of  $m$  and the multiplicity of  $p$  in  $S$ .

**GModulePrimes(G, A, B)**

Let  $G$  be a finitely presented group,  $A$  a normal subgroup of finite index in  $G$  and  $B$  a normal subgroup of  $G$  contained in  $A$ . Given any prime  $p$ , the maximal  $p$ -elementary abelian quotient of  $A/B$  can be viewed as a  $\mathbf{F}_p[G]$ -module  $M_p$ . This function determines all primes  $p$  such that  $M_p$  is not trivial (i.e. zero-dimensional) and the dimensions of the corresponding modules  $M_p$ . The return value is a multiset  $S$ . If  $0 \notin S$ , the maximal abelian quotient of  $A/B$  is finite and the multiplicity of  $p$  is the dimension of  $M_p$ . If  $S$  contains 0 with multiplicity  $m$ , the maximal abelian quotient of  $A/B$  contains  $m$  copies of  $\mathbf{Z}$ . In this case,  $M_p$  is non-trivial for every prime  $p$ . The rank of  $M_p$  in this case is the sum of  $m$  and the multiplicity of  $p$  in  $S$ .

**GModule(G, A, p)**

Given a finitely presented group  $G$ , a normal subgroup  $A$  of finite index in  $G$  and a prime  $p$ , create the  $\mathbf{F}_p[G]$ -module  $M$  corresponding to the conjugation action of  $G$  on the maximal  $p$ -elementary abelian quotient of  $A$ . The function also returns the epimorphism  $\pi : A \rightarrow M$ .

Note that normality of  $A$  in  $G$  is not checked. The results for invalid input data are undefined.

**GModule(G, A, B, p)****GModule(G, A, B)**

Given a finitely presented group  $G$ , a normal subgroup  $A$  of  $G$  of finite index, a normal subgroup  $B$  of  $G$  contained in  $A$  and a prime  $p$ , create the  $\mathbf{F}_p[G]$ -module  $M$  corresponding to the conjugation action of  $G$  on the maximal  $p$ -elementary abelian quotient of  $A/B$ .

$p$  can be omitted, if the maximal elementary abelian quotient of  $A/B$  is a  $p$ -group for some prime  $p$ . Note, however, that the computation is much faster, if a prime is specified.

The function also returns the epimorphism  $\pi : A \rightarrow M$ .

Note that normality of  $A$  and  $B$  in  $G$  is not checked. The results for invalid input data are undefined.

**Pullback(f, N)**

Given a map  $f : A \rightarrow M$  from a normal subgroup  $A$  of an fp-group  $G$  onto a  $\mathbf{F}_p[G]$ -module  $M$  and a submodule  $N$  of  $M$ , try to compute the preimage of  $N$  under  $f$  using a fast pullback method. If successful, the preimage is returned as subgroup of  $A$ .

If the pullback works, it is in general faster than a direct computation of the preimage using the preimage operator and it produces a more concise generating set for the preimage; see the following example. In cases where the pullback fails, a runtime error is reported and a preimage construction should be used instead.

**Example H70E73**

---

Consider the group  $G$  defined by the presentation

$$\langle a, b, c, d, e \mid a^4, b^{42}, c^6, e^3, b^a = b^{-1}, [a, c], [a, d], [a, e], \\ c^b = ce, d^b = d^{-1}, e^b = e^2, d^c = de, e^c = e^2, [d, e] \rangle.$$

```
> F<a,b,c,d,e> := FreeGroup(5);
> G<a,b,c,d,e> := quo< F | a^4, b^42, c^6, e^3,
>                   b^a=b^-1, (a,c), (a,d), (a,e),
>                   c^b=c*e, d^b=d^-1, e^b=e^2,
>                   d^c=d*e, e^c=e^2,
>                   (d,e) >;
```

The finite index subgroup  $H$  of  $G$  generated by  $c, d, e$  is normal in  $G$ .

```
> H := sub< G | c,d,e >;
> Index(G, H);
168
> IsNormal(G, H);
true
```

We check, for which characteristics the action of  $G$  on  $H$  yields non-trivial modules.

```
> GModulePrimes(G, H);
{* 0, 2, 3 *}
```

We construct the  $\mathbf{F}_3[G]$ -module  $M$  given by the action of  $G$  on the maximal 3-elementary abelian quotient of  $H$  and the natural epimorphism  $\pi$  from  $H$  onto the additive group of  $M$ .

```
> M, pi := GModule(G, H, 3);
> M;
GModule M of dimension 2 over GF(3)
```

Using the function `Submodules`, we obtain the submodules of  $M$ . Their preimages under  $\pi$  are precisely the normal subgroups of  $G$  which are contained in  $H$  and contain  $\ker(\pi)$ .

```
> submod := Submodules(M);
> time nsgs := [ m @@ pi : m in submod ];
Time: 11.640
> [ Index(G, s) : s in nsgs ];
[ 1512, 504, 504, 168 ]
```

The generating sets for the normal subgroups obtained in this way, contain in general many redundant generators. (E.g. each will contain a generating set for  $\ker(\pi)$ .)

```
> [ NumberOfGenerators(s) : s in nsgs ];
[ 19, 20, 20, 21 ]
```

Optimised generating sets can be obtained using the function `ReduceGenerators`.

```
> nsgs_red := [ ReduceGenerators(s) : s in nsgs ];
> [ NumberOfGenerators(s) : s in nsgs_red ];
```

```
[ 2, 2, 3, 2 ]
```

Alternatively, and in fact this is the recommended way, we can use the function `Pullback` to compute the preimages of the submodules under  $\pi$ . Note that the generating sets for the preimages computed this way contain fewer redundant generators.

```
> time nsgs := [ Pullback(pi, m) : m in submod ];
Time: 8.560
> [ Index(G, s) : s in nsgs ];
[ 1512, 504, 504, 168 ]
> [ NumberOfGenerators(s) : s in nsgs ];
[ 4, 4, 3, 2 ]
```

### Example H70E74

---

Consider the group defined by the presentation

$$\langle a, b, c, d, e \mid a^5, b^5, c^6, d^5, e^3, b^a = bd, \\ (a, c), (a, d), (a, e), (b, c), (b, d), (b, e), (c, d), (c, e), (d, e) \rangle.$$

```
> G<a,b,c,d,e> := Group< a,b,c,d,e |
>                               a^5, b^5, c^6, d^5, e^3, b^a = b*d,
>                               (a,c), (a,d), (a,e), (b,c), (b,d), (b,e),
>                               (c,d), (c,e), (d,e) >;
```

Obviously the subgroup of  $G$  generated by  $b, c, d, e$  is normal in  $G$ .

```
> H := sub< G | b,c,d,e >;
> IsNormal(G, H);
true
```

We use the function `GModulePrimes` to determine the set of primes  $p$  for which the action of  $G$  on the maximal  $p$ -elementary abelian quotient of  $H$  induces a nontrivial  $\mathbf{F}_p[G]$ -module.

```
> P := GModulePrimes(G, H);
> 0 in P;
false
```

0 is not contained in  $P$ , i.e. the maximal free abelian quotient of  $H$  is trivial. Hence, there are only finitely many primes, satisfying the condition above.

We loop over the distinct elements of  $P$  and for each element  $p$  we construct the induced  $\mathbf{F}_p[G]$ -module, print its dimension and check whether it is decomposable. Note that the dimension of the module for  $p$  must be equal to the multiplicity of  $p$  in  $P$ .

```
> for p in MultisetToSet(P) do
>   M := GModule(G, H, p);
>   dim := Dimension(M);
>   decomp := IsDecomposable(M);
>
>   assert dim eq Multiplicity(P, p);
```

```

>
>   print "prime", p, ": module of dimension", dim;
>   if decomp then
>     print "  has a nontrivial decomposition";
>   else
>     print "  is indecomposable";
>   end if;
> end for;
prime 2 : module of dimension 1
  is indecomposable
prime 3 : module of dimension 2
  has a nontrivial decomposition
prime 5 : module of dimension 2
  is indecomposable

```

---

## 70.11 Small Group Identification

This section describes some special issues arising with the identification of a finitely presented group using the database of small groups described in Section 66.2.

### IdentifyGroup( $G$ )

Locate the pair of integers  $\langle o, n \rangle$  so that `SmallGroup(o, n)` is isomorphic to  $G$ . If the construction of a permutation representation for  $G$  fails or if there is no group isomorphic to  $G$  in the database, then an error will result.

When trying to look up a finitely presented group  $G$  in the database of small groups, MAGMA tries to construct a permutation representation of  $G$  by enumerating the cosets of the trivial subgroup in  $G$ . Assuming that a group isomorphic to  $G$  is contained in the database, the resulting coset table will be fairly small. Hence for performance reasons, a coset limit of  $100 \cdot o$  is imposed, where  $o$  is the maximal order of groups in the database, unless the order of  $G$  is known to be less or equal to  $o$ . If, on the other hand,  $|G| \leq o$  is known, the global set of parameters for implicitly invoked coset enumerations applies. This set of parameters can be changed using the function `SetGlobalTCPParameters`.

If the coset enumeration for  $G$  fails with the coset limit  $100 \cdot o$ , this can be seen as a reasonable indication that  $G$  is probably too large to be contained in the database of small groups.

To deal with cases where the coset enumeration fails although  $G$  is known or suspected to be small enough, it is recommended to attempt to compute the order of  $G$  using the function `Order` before the actual group identification. If  $G$  that way can be shown to be small enough to be contained in the database, the function `SetGlobalTCPParameters` can be used to control the behaviour of coset enumerations in a subsequent call to `IdentifyGroup`. The following example illustrates this.

**Example H70E75**

---

Consider the group defined by the presentation

$$\langle a, b \mid (ba^{-1})^3, (ba^{-1}b)^2, a^{12}b^2a^7b^2ab^2 \rangle.$$

```
> G := Group<a,b | (b*a^-1)^3, (b*a^-1*b)^2,
>           a^12*b^2*a^7*b^2*a*b^2 >;
```

We suspect (for some reason) that  $G$  is a small group and want to identify its isomorphism type in the database of small groups.

```
> IdentifyGroup(G);
IdentifyGroup(
  G: GrpFP: G
)
In file "/home/magma/package/Group/Grp/smallgps2.m", line 220,
column 25:
>>   res := IdentifyGroup(db, G);
```

```
Runtime error in 'IdentifyGroup': Coset enumeration failed; group
may be too large (see handbook entry for details)
```

The group couldn't be identified, because it was not possible to obtain a permutation representation with the default value of the coset limit.

Since we still think the group should be small, we try to prove this using the function `Order`.

```
> Order(G : Print := true);
INDEX = 6 (a=6 r=66709 h=999999 n=999999; l=1247 c=1.56; m=969169
t=999998)
6
```

We were correct; the group is in fact very small. However, we can see from the output of the `Order` command that this wasn't that easy to find out: almost one million cosets were used in the coset enumeration.

Since the order of  $G$  is now known to the system, a subsequent call to `IdentifyGroup` will now use the global parameters for implicitly invoked coset enumerations. To be on the safe side, we tell the system to be prepared to work a little harder using the function `SetGlobalTCPParameters` and try again to identify  $G$ .

```
> SetGlobalTCPParameters( : Strategy := "Hard");
>
> IdentifyGroup(G);
<6, 1>
```

Now  $G$  can be identified.

---

### 70.11.1 Concrete Representations of Small Groups

When an finitely-presented group is known to small, it may be useful to write it concretely as a permutation or PC-group. There are two utilities provided for this purpose.

`PermutationGroup(G)`

Construct a faithful permutation representation of  $G$ , and the isomorphism from  $G$  to the representation. The algorithm first computes the order of  $G$ , then takes the regular representation of  $G$ , and reduces the degree. Thus the command is restricted to small groups.

`PCGroup(G)`

Construct a faithful PC-group representation of  $G$ , and the isomorphism from  $G$  to the representation. The algorithm first computes the order of  $G$ , then computes the soluble quotient of  $G$  with the order found. Thus the command is restricted to small soluble groups.

## 70.12 Bibliography

- [AR84] D. G. Arrell and E. F. Robertson. A modified Todd-Coxeter algorithm. In *Computational group theory (Durham, 1982)*, pages 27–32. Academic Press, London, 1984.
- [CDHW73] John J. Cannon, Lucien A. Dimino, George Havas, and Jane M. Watson. Implementation and analysis of the Todd-Coxeter algorithm. *Math. Comp.*, 27:463–490, 1973.
- [CHN11] M. Conder, G. Havas, and M. Newman. On one-relator quotients of the modular group. In *Proc. Groups St Andrews 2009 in Bath*, number 387 in London Mathematical Society Lecture Note Series, pages 183–197. Cambridge University Press, 2011.
- [COS08] A. Cavicchioli, E. O’Brien, and F. Spaggiari. On some questions about a family of cyclically presented groups. *J. Algebra*, 320(11):4063–4072, 2008.
- [Fab09] Anna Fabianska. *Algorithmic analysis of presentations of groups and modules*. Dissertation, RWTH Aachen University, 2009.
- [Hav91] G. Havas. Coset enumeration strategies. In *ISSAC’91*, pages 191–199. ACM Press, 1991.
- [HH10] G. Havas and D.F. Holt. On Coxeter’s families of group presentations. *J. Algebra*, 324(5):1076–1082, 2010.
- [HKRR84] George Havas, P. E. Kenne, J. S. Richardson, and E. F. Robertson. A Tietze transformation program. In *Computational group theory (Durham, 1982)*, pages 69–73. Academic Press, London, 1984.
- [MKS76] Wilhelm Magnus, Abraham Karrass, and Donald Solitar. *Combinatorial group theory*. Dover Publications Inc., New York, revised edition, 1976. Presentations of groups in terms of generators and relations.

- [**Nic96**] Werner Nickel. Computing nilpotent quotients of finitely presented groups. In *Geometric and computational perspectives on infinite groups (Minneapolis, MN and New Brunswick, NJ, 1994)*, pages 175–191. Amer. Math. Soc., Providence, RI, 1996.
- [**NO96**] M. F. Newman and E. A. O’Brien. Application of computers to questions like those of Burnside. II. *Internat. J. Algebra Comput.*, 6(5):593–605, 1996.
- [**PF09**] W. Plesken and A. Fabianska. An L2-quotient algorithm for finitely presented groups. *J. Algebra*, 322(3):914–935, 2009.
- [**Ram**] Colin Ramsay. ACE. URL:<http://www.csee.uq.edu.au/~cram/>.
- [**Sim94**] Charles C. Sims. *Computation with finitely presented groups*. Cambridge University Press, Cambridge, 1994.



# 71 FINITELY PRESENTED GROUPS: ADVANCED

<p><b>71.1 Introduction . . . . . 2203</b></p> <p><b>71.2 Low Level Operations on Presentations and Words . . . . . 2203</b></p> <p><i>71.2.1 Modifying Presentations . . . . . 2204</i></p> <p>AddGenerator(G) . . . . . 2204</p> <p>AddGenerator(G, w) . . . . . 2204</p> <p>AddRelation(G, r) . . . . . 2204</p> <p>AddRelation(G, g) . . . . . 2204</p> <p>AddRelation(G, r, i) . . . . . 2204</p> <p>AddRelation(G, g, i) . . . . . 2204</p> <p>DeleteGenerator(G, x) . . . . . 2204</p> <p>DeleteRelation(G, r) . . . . . 2204</p> <p>DeleteRelation(G, g) . . . . . 2205</p> <p>DeleteRelation(G, i) . . . . . 2205</p> <p>ReplaceRelation(G, s, r) . . . . . 2205</p> <p>ReplaceRelation(G, h, r) . . . . . 2205</p> <p>ReplaceRelation(G, s, g) . . . . . 2205</p> <p>ReplaceRelation(G, h, g) . . . . . 2205</p> <p>ReplaceRelation(G, i, r) . . . . . 2205</p> <p>ReplaceRelation(G, i, g) . . . . . 2205</p> <p><i>71.2.2 Low Level Operations on Words . . . . . 2206</i></p> <p>Eliminate(u, x, v) . . . . . 2206</p> <p>Eliminate(U, x, v) . . . . . 2206</p> <p>Match(u, v, f) . . . . . 2207</p> <p>RotateWord(u, n) . . . . . 2207</p> <p>Substitute(u, f, n, v) . . . . . 2207</p> <p>Subword(u, f, n) . . . . . 2207</p> <p><b>71.3 Interactive Coset Enumeration 2208</b></p> <p><i>71.3.1 Introduction . . . . . 2208</i></p> <p><i>71.3.2 Constructing and Modifying a Coset Enumeration Process . . . . . 2209</i></p> <p>CosetEnumerationProcess(G, H: -) . . . . . 2209</p> <p>AddRelator(<math>\sim P</math>, w) . . . . . 2212</p> <p>AddSubgroupGenerator(<math>\sim P</math>, w) . . . . . 2213</p> <p>SetProcessParameters(<math>\sim P</math>: -) . . . . . 2214</p> <p><i>71.3.3 Starting and Restarting an Enumeration . . . . . 2214</i></p> <p>StartEnumeration(<math>\sim P</math>: -) . . . . . 2214</p> <p>RedoEnumeration(<math>\sim P</math>: -) . . . . . 2215</p> <p>CanRedoEnumeration(P) . . . . . 2215</p> <p>ContinueEnumeration(<math>\sim P</math>: -) . . . . . 2215</p> <p>CanContinueEnumeration(P) . . . . . 2215</p> <p>ResumeEnumeration(<math>\sim P</math>: -) . . . . . 2216</p> <p><i>71.3.4 Accessing Information . . . . . 2216</i></p> <p>CosetsSatisfying(P : -) . . . . . 2216</p> <p>CosetSatisfying(P : -) . . . . . 2216</p> <p>CosetTable(P) . . . . . 2217</p>	<p>IsValidCosetTable(P) . . . . . 2217</p> <p>HasClosedCosetTable(P) . . . . . 2217</p> <p>HasCompleteCosetTable(P) . . . . . 2217</p> <p>ExcludedConjugate(P) . . . . . 2218</p> <p>ExcludedConjugates(P) . . . . . 2218</p> <p>ExistsCosetSatisfying(P : -) . . . . . 2218</p> <p>ExistsExcludedConjugate(P) . . . . . 2218</p> <p>ExistsNormalisingCoset(P) . . . . . 2219</p> <p>ExistsNormalizingCoset(P) . . . . . 2219</p> <p>Group(P) . . . . . 2219</p> <p>Index(P) . . . . . 2219</p> <p>IsValidIndex(P) . . . . . 2219</p> <p>MaximalNumberOfCosets(P) . . . . . 2219</p> <p>Subgroup(P) . . . . . 2220</p> <p>TotalNumberOfCosets(P) . . . . . 2220</p> <p><i>71.3.5 Induced Permutation Representations . . . . . 2225</i></p> <p>CosetAction(P) . . . . . 2226</p> <p>CosetImage(P) . . . . . 2226</p> <p>CosetKernel(P) . . . . . 2226</p> <p><i>71.3.6 Coset Spaces and Transversals . . . . . 2226</i></p> <p>CosetSpace(P) . . . . . 2227</p> <p>RightCosetSpace(P) . . . . . 2227</p> <p>LeftCosetSpace(P) . . . . . 2227</p> <p>Transversal(P) . . . . . 2227</p> <p>RightTransversal(P) . . . . . 2227</p> <p><b>71.4 <math>p</math>-Quotients (Process Version) . 2229</b></p> <p><i>71.4.1 The <math>p</math>-Quotient Process . . . . . 2229</i></p> <p>pQuotientProcess(F, p, c: -) . . . . . 2229</p> <p>NextClass(<math>\sim P</math> : -) . . . . . 2230</p> <p>NextClass(<math>\sim P</math>, k : -) . . . . . 2230</p> <p><i>71.4.2 Using <math>p</math>-Quotient Interactively . . . . . 2230</i></p> <p>StartNewClass(<math>\sim P</math>: -) . . . . . 2230</p> <p>Tails(<math>\sim P</math>: -) . . . . . 2230</p> <p>Tails(<math>\sim P</math>, k: -) . . . . . 2230</p> <p>Consistency(<math>\sim P</math>: -) . . . . . 2231</p> <p>Consistency(<math>\sim P</math>, k: -) . . . . . 2231</p> <p>CollectRelations(<math>\sim P</math>) . . . . . 2231</p> <p>ExponentLaw(<math>\sim P</math> : -) . . . . . 2231</p> <p>ExponentLaw(<math>\sim P</math>, Start, Fin: -) . . . . . 2231</p> <p>EliminateRedundancy(<math>\sim P</math>) . . . . . 2232</p> <p>Display(P) . . . . . 2232</p> <p>Display(P, DisplayLevel) . . . . . 2232</p> <p>RevertClass(<math>\sim P</math>) . . . . . 2232</p> <p>pCoveringGroup(<math>\sim P</math>) . . . . . 2232</p> <p>pCoveringGroup(G) . . . . . 2232</p> <p>GeneratorStructure(P) . . . . . 2232</p> <p>GeneratorStructure(P, Start, Fin) . . . . . 2232</p>
---	---

Jacobi( $\sim P$ , c, b, a, $\sim r$ )	2233	71.5.1 Introduction . . . . .	2239
Jacobi( $\sim P$ , c, b, a)	2233	71.5.2 Construction . . . . .	2239
Collect(P, Q)	2233	71.5.3 Calculating the Relevant Primes. . . . .	2241
EcheloniseWord( $\sim P$ , $\sim r$ )	2233	71.5.4 The Functions . . . . .	2241
EcheloniseWord( $\sim P$ )	2233	SolubleQuotient(F, n : -)	2242
SetDisplayLevel( $\sim P$ , Level)	2233	SolvableQuotient(F, n : -)	2242
ExtractGroup(P)	2233	SolubleQuotient(F : -)	2242
Order(P)	2233	SolvableQuotient(F : -)	2242
FactoredOrder(P)	2233	SolubleQuotient(F, P : -)	2242
NumberOfPCGenerators(P)	2233	SolvableQuotient(F, P : -)	2242
pClass(P)	2234	<b>71.6 Bibliography . . . . .</b>	<b>2245</b>
NuclearRank(G)	2234		
NuclearRank(P)	2234		
pMultiplierRank(G)	2234		
pMultiplierRank(P)	2234		
<b>71.5 Soluble Quotients . . . . .</b>	<b>2239</b>		

# Chapter 71

## FINITELY PRESENTED GROUPS: ADVANCED

### 71.1 Introduction

This section presents some more advanced techniques available for computing with finitely-presented groups (fp-groups for short) within MAGMA. The features considered here are regarded as more advanced, either because they are technically or theoretically more complex and they are therefore expected to be used mainly by specialists, or because their efficient (and in a few cases even their merely correct) use requires some more detailed knowledge on the user's part.

Trying to summarise the expected main purpose of the functions described in this section, one could think of two main situations: On the one hand, user written functions, which may benefit from low-level tools for manipulating presentations or words, or which make use of interruptible process versions of some standard MAGMA functions for fp-groups. On the other hand, the solution of very hard problems, which require careful fine-tuning of the strategy employed or for which some iterative approach, using feedback of information obtained during the computation, is necessary.

The following topics are discussed in detail. First, some rather low-level operations on presentations and elements of fp-groups (words) are described. Then, the features for interactive coset enumeration in MAGMA are presented. This section also contains the complete description of all the parameters available for controlling the execution of the Todd-Coxeter procedure, which also applies to the appropriate standard functions documented in Chapter 70. After that, we describe the process version of the  $p$ -quotient algorithm. Note that some care has to be taken when interpreting results obtained with this interactive  $p$ -quotient computation; incorrect use of the existing functions may result in incomplete or wrong answers. The chapter ends with a treatise of the soluble quotient algorithm available in MAGMA. This final section contains a brief review of the theory underlying the soluble quotient algorithm, a description of the parameters available for functions computing soluble quotients, and the documentation of the interactive soluble quotient facilities.

### 71.2 Low Level Operations on Presentations and Words

In this section, we describe some rather low level operations on presentations and on elements of fp-groups. The main purpose of the functions described here, is to provide some efficient machinery for manipulating presentations and elements of fp-groups for user written functions.

### 71.2.1 Modifying Presentations

The functions described in this section construct a new fp-group from an existing one by adding or deleting a generator or by adding, deleting or changing a relation. The new group is created without any relationship to the existing group.

**AddGenerator( $G$ )**

Given an fp-group  $G$  with presentation  $\langle X \mid R \rangle$ , create a new fp-group with presentation  $\langle X \cup \{z\} \mid R \rangle$ , where  $z$  is a symbol not in  $X$ .

**AddGenerator( $G, w$ )**

Given an fp-group  $G$  with presentation  $\langle X \mid R \rangle$ , and given also a word  $w$  in the generators  $X$ , create a new fp-group having the presentation  $\langle X \cup \{z\} \mid R \cup \{z = w\} \rangle$ , where  $z$  is a symbol not in  $X$ .

**AddRelation( $G, r$ )**

Given an fp-group  $G$ , and a relation  $r$  on the generators of  $G$ , create a new fp-group whose presentation consists of the relations of  $G$  together with the relation  $r$ .

**AddRelation( $G, g$ )**

Given an fp-group  $G$ , and an element  $g$  of  $G$ , create a new fp-group whose presentation consists of the relations of  $G$  together with the relation  $g = \text{Id}(G)$ .

**AddRelation( $G, r, i$ )**

Given an fp-group  $G$ , and a relation  $r$  on the generators of  $G$ , create a new fp-group which has as its presentation the relations of  $G$  together with the relation  $r$  inserted after the  $i$ -th existing relation of  $G$ .

**AddRelation( $G, g, i$ )**

Given an fp-group  $G$ , and an element  $g$  of  $G$ , create a new fp-group which has as its presentation the relations of  $G$  together with the relation  $g = \text{Id}(G)$  inserted after the  $i$ -th existing relation of  $G$ .

**DeleteGenerator( $G, x$ )**

Given an fp-group  $G$  with presentation  $\langle X \mid R \rangle$ , and given also an element  $z$  in  $X$ , create a new fp-group with presentation  $\langle X \setminus \{z\} \mid R' \rangle$ , where the relations  $R'$  are obtained from  $R$  by deleting all relations containing an occurrence of  $z$ .

**DeleteRelation( $G, r$ )**

Given an fp-group  $G$ , which includes the relation  $r$  amongst its relations, create a new fp-group which has as its presentation the relations of  $G$  with relation  $r$  omitted.

DeleteRelation( $G, g$ )

Given an fp-group  $G$ , which includes the relation  $g = \text{Id}(G)$  amongst its relations, create a new fp-group which has as its presentation the relations of  $G$  with this relation omitted.

DeleteRelation( $G, i$ )

Given an fp-group  $G$ , create a new fp-group which has as its presentation the relations for  $G$  with the  $i$ -th relation deleted.

ReplaceRelation( $G, s, r$ )

ReplaceRelation( $G, h, r$ )

ReplaceRelation( $G, s, g$ )

ReplaceRelation( $G, h, g$ )

Given an fp-group  $G$ , which includes the relation  $s$  or  $h = \text{Id}(G)$  amongst its relations, create a new fp-group which has as its presentation the relations for  $G$  with the relation  $s$  replaced by the relation  $r$  or  $g = \text{Id}(G)$ .

ReplaceRelation( $G, i, r$ )

Given an fp-group  $G$  and a relation  $r$  in the generators of  $G$ , create a new fp-group which has as its presentation the relations for  $G$  with relation number  $i$  replaced by the relation  $r$ .

ReplaceRelation( $G, i, g$ )

Given an fp-group  $G$  and an element  $g$  of  $G$ , create a new fp-group which has as its presentation the relations for  $G$  with relation number  $i$  replaced by the relation  $g = \text{Id}(G)$ .

### Example H71E1

---

We use the function `ReplaceRelation` to vary a particular relation in a presentation. The order of the resulting group together with the index of a particular subgroup is determined.

```
> G<x,y,z,h,k,a> := Group< x, y, z, h, k, a |
>   x^2, y^2, z^2, (x,y), (y,z), (x,z), h^3, k^3, (h,k),
>   (x,k), (y,k), (z,k), x^h*y, y^h*z, z^h*x, a^2, a*x*a*y,
>   a*y*a*x, (a,z), (a,k), (a*h)^2 >;
> for i := 0 to 1 do
>   for j := 0 to 1 do
>     for k := 0 to 1 do
>       for l := 0 to 2 do
>         rel := G.1^i*G.2^j*G.3^k*G.5^l*(G.6*G.4)^2 = Id(G);
>         K := ReplaceRelation(G, 21, rel);
>         print Order(K), Index(K, sub< K | K.6, K.4>);
>       end for;
>     end for;
>   end for;
```

```

>     end for;
> end for;
<0, 0, 0, 0> 144 24
<0, 0, 0, 1> 144 8
<0, 0, 0, 2> 144 8
<0, 0, 1, 0> 18 3
<0, 0, 1, 1> 18 1
<0, 0, 1, 2> 18 1
<0, 1, 0, 0> 72 3
<0, 1, 0, 1> 72 1
<0, 1, 0, 2> 72 1
<0, 1, 1, 0> 36 6
<0, 1, 1, 1> 36 2
<0, 1, 1, 2> 36 2
<1, 0, 0, 0> 18 3
<1, 0, 0, 1> 18 1
<1, 0, 0, 2> 18 1
<1, 0, 1, 0> 144 6
<1, 0, 1, 1> 144 2
<1, 0, 1, 2> 144 2
<1, 1, 0, 0> 36 6
<1, 1, 0, 1> 36 2
<1, 1, 0, 2> 36 2
<1, 1, 1, 0> 72 12
<1, 1, 1, 1> 72 4
<1, 1, 1, 2> 72 4

```

---

### 71.2.2 Low Level Operations on Words

The functions described in this section perform low level string operations like substitution, elimination or substring matching on elements of fp-groups.

**Eliminate(u, x, v)**

Given words  $u$  and  $v$ , and a generator  $x$ , all belonging to a group  $G$ , return the word obtained from  $u$  by replacing each occurrence of  $x$  by  $v$  and each occurrence of  $x^{-1}$  by  $v^{-1}$ .

**Eliminate(U, x, v)**

Given a set of words  $U$ , a word  $v$ , and a generator  $x$ , all belonging to a group  $G$ , return the set of words obtained by taking each element  $u$  of  $U$  in turn, and replacing each occurrence of  $x$  in  $u$  by  $v$  and each occurrence of  $x^{-1}$  by  $v^{-1}$ .

**Match(u, v, f)**

Suppose  $u$  and  $v$  are words belonging to the same group  $G$ , and that  $f$  is an integer such that  $1 \leq f \leq \#u$ . The function seeks the least integer  $l$  such that:

- (a)  $l \geq f$ ; and
- (b)  $v$  appears as a subword of  $u$ , starting at the  $l$ -th letter of  $u$ .

If such an integer  $l$  is found **Match** returns the value true and  $l$ . If no such  $l$  is found, **Match** returns the value false.

**RotateWord(u, n)**

The word obtained by cyclically permuting the word  $u$  by  $n$  places. If  $n$  is positive, the rotation is from left to right, while if  $n$  is negative the rotation is from right to left. In the case where  $n$  is zero, the function returns  $u$ .

**Substitute(u, f, n, v)**

Given words  $u$  and  $v$  belonging to a group  $G$ , and non-negative integers  $f$  and  $n$ , this function replaces the substring of  $u$  of length  $n$ , starting at position  $f$ , by the word  $v$ . Thus, if  $u = x_{i_1}^{e_1} \cdots x_{i_f}^{e_f} \cdots x_{i_{f+n-1}}^{e_{f+n-1}} \cdots x_{i_m}^{e_m}$  then the substring  $x_{i_f}^{e_f} \cdots x_{i_{f+n-1}}^{e_{f+n-1}}$  is replaced by  $v$ . If the function is invoked with  $v = \text{Id}(G)$ , then the substring  $x_{i_f}^{e_f} \cdots x_{i_{f+n-1}}^{e_{f+n-1}}$  of  $u$  is deleted.

**Subword(u, f, n)**

The subword of the word  $u$  comprising the  $n$  consecutive letters commencing at the  $f$ -th letter of  $u$ .

**Example H71E2**

We demonstrate some of these operations in the context of the free group on generators  $x$ ,  $y$ , and  $z$ .

```
> F<x, y, z> := FreeGroup(3);
> u := (x, y*z);
> w := u^(x^2*y);
> #w;
12
> w;
y^-1 * x^-3 * z^-1 * y^-1 * x * y * z * x^2 * y
```

We replace each occurrence of the generator  $x$  in  $w$  by the word  $y * z^{-1}$ .

```
> Eliminate(w, x, y*z^-1);
y^-1 * z * y^-1 * z * y^-1 * z * y^-1 * z^-2 * y * z * y * z^-1 * y * z^-1 * y
```

We count the number of occurrences of each generator in  $w$ .

```
> [ ExponentSum(w, F.i) : i in [1..Ngens(F)] ];
[ 0, 0, 0 ]
> GeneratorNumber(w);
```

-2

We locate the start of the word  $u$  in the word  $w$ .

```
> b, p := Match(w, u, 1);
> b, p;
true 4
```

We now replace the subword  $u$  in  $w$  by the word  $y * x$ .

```
> t := Substitute(w, p, #u, y*x);
> t;
y^-1 * x^-2 * y * x^3 * y
```

We create the set of all distinct cyclic permutations of the word  $u$ .

```
> rots := { RotateWord(u, i) : i in [1 ..#u] };
> rots;
{ y^-1 * x * y * z * x^-1 * z^-1, x * y * z * x^-1 * z^-1 * y^-1,
x^-1 * z^-1 * y^-1 * x * y * z, z * x^-1 * z^-1 * y^-1 * x * y,
z^-1 * y^-1 * x * y * z * x^-1, y * z * x^-1 * z^-1 * y^-1 * x }
```

---

## 71.3 Interactive Coset Enumeration

### 71.3.1 Introduction

This section presents the interactive coset enumeration facility of MAGMA. This concept makes it possible to restart an enumeration after changing enumeration parameters or adding relators or subgroup generators, while making use of information obtained up to that point as much as possible. It is thus particularly suitable for very hard enumerations, requiring a careful and interactive choice of enumeration parameters or for series of similar enumerations.

The Todd-Coxeter implementation installed in MAGMA is based on the stand alone coset enumeration programme ACE3 developed by George Havas and Colin Ramsay at the University of Queensland. The reader should consult [CDHW73] and [Hav91] for an explanation of the terminology and a general description of the algorithm. A manual for ACE3 as well as the sources of ACE3 can be found online [Ram].

In MAGMA an interactive coset enumeration is realised as an object of the category `GrpFPCosetEnumProc` which can be created and modified, allows starting and restarting of coset enumerations and provides access to internal data like the coset table and various status information.





In R-style it is usual to scan each row of the table after its coset has been applied to all relators, and to make definitions to fill any holes encountered. Failure to do so can cause even simple enumerations to overflow. To switch row filling off, set the parameter `RowFilling` to false.

`PrefDefMode`                      `RNGINTELT`                      *Default : 3*

If the argument is 0, then Felsch style definitions are made using the next empty position in the coset table. Otherwise, gaps of length one found during relator scans are preferentially filled. If the argument is 1, they are filled immediately, and if it is 2, the consequent deduction is also made immediately. If the argument is 3, then the gaps are noted in the preferred definition queue and the next coset definition will be made to fill the oldest gap of length one.

`PrefDefSize`                      `RNGINTELT`                      *Default : 8*

This parameter controls the size of the preferred definition queue, which is implemented as a ring buffer, dropping earliest entries. Setting `PrefDefSize` to  $n$  allocates a buffer of size  $2^n$ .

`DeductionMode`                      `RNGINTELT`                      *Default : 4*

A completed table is only valid if every table entry has been tested in all essentially different relator positions. Untested deductions are stored on a stack. This parameter allows the user to specify how deductions should be handled. The possible actions are:

For `DeductionMode := 0` : discard deductions if there is no stack space left.

For `DeductionMode := 1` : as 0, but redundant cosets are purged off the top of the stack whenever a coincidence is found.

For `DeductionMode := 2` : as 0, but all redundant cosets are purged from the stack whenever a coincidence is found.

For `DeductionMode := 3` : discard the entire stack if it overflows.

For `DeductionMode := 4` : if the stack overflows, then double the stack size and purge all redundant cosets from the stack.

If deductions are discarded for any reason during an enumeration, then a final relator application pass will be done at the end of the enumeration automatically to check the result.

`DeductionSize`                      `RNGINTELT`                      *Default : 1000*

Sets the (initial) size of the deduction stack in words, with one deduction taking two words. A value of 0 selects the default size of 1000 words.

`PathCompression`                      `BOOLELT`                      *Default : false*

Switching this option on reduces the amount of data movement during coincidence processing at the expense of tracing and compressing coincidence paths, which involves many coset table accesses. The value of this parameter has no effect on the result but may influence the running time.

`TimeLimit`                      `RNGINTELT`                      *Default : -1*



	Default	Easy	Hard	Felsch	HLT
Compact	10	100	10	10	10
Workspace (in $10^6$ )	4	1	10	4	4
FillFactor	0	1	0	0	1
CTFactor	1000	0	1000	1000	0
RTFactor	$2000/l$	1000	1	0	1000
Style	R_CR	R	CR	C	R
Lookahead	0	0	0	0	1
Mendelsohn	false	false	false	false	false
RelationsInSubgroup	-1	0	-1	-1	0
RowFilling	true	true	true	false	true
PrefDefMode	3	0	3	3	0
DeductionMode	4	0	4	4	0

Table 1: Strategies

	CT	RT	Sims1	Sims3	Sims5
Compact	100	100	10	10	10
Workspace (in $10^6$ )	4	4	4	4	4
FillFactor	1	1	1	1	1
CTFactor	1000	0	0	0	0
RTFactor	0	1000	1000	1000	1000
Style	C	R	R	Rt	R
Lookahead	0	0	0	0	0
Mendelsohn	false	false	false	false	true
RelationsInSubgroup	0	0	0	0	0
RowFilling	false	false	true	true	true
PrefDefMode	0	0	0	0	0
DeductionMode	4	0	0	4	0

Table 2: Strategies (continued)

AddSubgroupGenerator( $\sim P$ ,  $w$ )

Add an element  $w$  of the group  $G$  underlying the coset enumeration process  $P$  to the generators of the subgroup. This means that a coset enumeration process  $P$  for the cosets of  $H$  in  $G$  is transformed into a coset enumeration process for the cosets of  $\langle H, w \rangle$  in  $G$ , where  $\langle H, w \rangle$  denotes the subgroup of  $G$  generated by  $H$  and  $w$ .

	Sims7	Sims9
Compact	10	10
Workspace (in $10^6$ )	4	4
FillFactor	1	1
CTFactor	0	1000
RTFactor	1000	0
Style	Rt	C
Lookahead	0	0
Mendelsohn	true	false
RelationsInSubgroup	0	0
RowFilling	true	false
PrefDefMode	0	0
DeductionMode	4	4

Table 3: Strategies (continued)

**SetProcessParameters**( $\sim P$ : parameters)

Change enumeration parameters of the coset enumeration process  $P$ . The set of parameters accepted by this function is the same as for the function `CosetEnumerationProcess`; see there for a description.

All parameters which are not explicitly changed or modified by selecting one of the predefined strategies retain their old values.

It should be noted that it is not possible to decrease the workspace allocated by a coset enumeration process, once an enumeration has been started. However the workspace can be extended without invalidating any information contained in the process.

### 71.3.3 Starting and Restarting an Enumeration

There are several ways of starting and restarting an enumeration for a coset enumeration process, which retain information from previous enumerations to a varying extent.

**StartEnumeration**( $\sim P$ : parameters)

Start a new enumeration for  $P$ . All information in  $P$  is discarded. This function can be called at any time for an existing coset enumeration process. The enumeration parameters for  $P$  can be modified by passing parameters to this function. (This is equivalent to calling the function `SetProcessParameters` before calling `StartEnumeration`.) The set of parameters accepted by this function is the same as for the function `CosetEnumerationProcess`; see there for a description.

**RedoEnumeration**( $\sim P$ : *parameters*)

Restart an enumeration for  $P$ . All information in  $P$  is retained and the enumeration is restarted at coset number 1. This function can be called for any coset enumeration process, which contains a valid coset table. (If  $P$  does not contain a valid coset table, a call to **RedoEnumeration** causes a runtime error. Use the function **CanRedoEnumeration** to check whether a call to **RedoEnumeration** is legal for a certain coset enumeration process.) Note that the coset table of  $P$  need not be complete to use this function.

This function is intended for the case where additional relators and/or subgroup generators have been introduced. The coset table contained in  $P$  is still valid. However, the additional data may allow the enumeration to compete, or cause a collapse to a smaller index.

The enumeration parameters for  $P$  can be modified by passing parameters to this function. (This is equivalent to calling the function **SetProcessParameters** before calling **RedoEnumeration**.) The set of parameters accepted by this function is the same as for the function **CosetEnumerationProcess**; see there for a description.

**CanRedoEnumeration**( $P$ )

Returns **true**, if a call to **RedoEnumeration** is legal for the coset enumeration process  $P$ .

**ContinueEnumeration**( $\sim P$ : *parameters*)

Continue an enumeration for  $P$ , which has been interrupted because some limit was exceeded. All information in  $P$  is retained and the enumeration is restarted at the coset where the previous enumeration was stopped. This function can only be called for a coset enumeration process, if the previous enumeration produced a valid coset table and the subgroup underlying  $P$  has not been changed since then. (Otherwise, a call to **ContinueEnumeration** causes a runtime error. Use the function **CanContinueEnumeration** to check whether a call to **ContinueEnumeration** is legal for a certain coset enumeration process.) Note that the coset table of  $P$  need not be complete to use this function.

This function is intended for the case where an enumeration stopped without producing a finite index. This function allows to continue the enumeration with modified enumeration parameters with the minimal possible overhead.

The enumeration parameters for  $P$  can be modified by passing parameters to this function. (This is equivalent to calling the function **SetProcessParameters** before calling **ContinueEnumeration**.) The set of parameters accepted by this function is the same as for the function **CosetEnumerationProcess**; see there for a description.

**CanContinueEnumeration**( $P$ )

Returns **true**, if a call to **ContinueEnumeration** is legal for the coset enumeration process  $P$ .



If this parameter is set to  $l > 0$ , the search for coset representatives is aborted as soon as  $l$  cosets satisfying the designated condition have been found. If it is set to 0 (default for `CosetsSatisfying`), no limit is in force.

This parameter is not available for the function `CosetSatisfying`.

**Normalizing**                      `BOOLELT`                      *Default : false*

If true, select coset representatives  $x$  such that  $x^{-1}h_i x$  is known to be contained in  $H$  from the information in the coset table of  $P$  for every generator  $h_i$  of  $H$ . (I.e.  $x$  is recognised as an element of the normaliser of  $H$  in  $G$ .)

**Order**                              `RNGINTELT`                      *Default : 0*

Select coset representatives  $x$  such that  $x^n$  is known to be contained in  $H$  from the information in the coset table of  $P$ .

**Print**                                `RNGINTELT`                      *Default : 0*

If the value of this parameter is positive, print the coset representatives found to satisfy the designated conditions.

These functions can be called for any coset enumeration process, which contains a valid coset table. If  $P$  does not contain a valid coset table, a call to any of these functions causes a runtime error. You can use the function `HasValidCosetTable` to check whether a call to these functions is legal for a certain coset enumeration process.

#### `CosetTable(P)`

The current coset table of  $P$  as a map  $f : \{1, \dots, r\} \times G \rightarrow \{0, \dots, r\}$ , where  $G$  and  $H$  are the finitely presented group and the subgroup underlying  $P$ , respectively, and  $r$  is number of active cosets.  $f(i, x)$  is the coset to which coset  $i$  is mapped under the action of  $x \in G$ . The value 0 is only included in the codomain if the coset table is not complete, and it denotes that the image of  $i$  under  $x$  is not known.

This function can be called for any coset enumeration process, which contains a valid coset table. If  $P$  does not contain a valid coset table, a call to `CosetTable` causes a runtime error. Use the function `HasValidCosetTable` to check whether a call to `CosetTable` is legal for a certain coset enumeration process. Note that the coset table of  $P$  need not be complete to use this function.

#### `HasValidCosetTable(P)`

Returns `true`, if  $P$  contains a valid (but not necessarily closed) coset table, i.e. if a call to `CosetTable` is legal for the coset enumeration process  $P$ .

#### `HasClosedCosetTable(P)`

#### `HasCompleteCosetTable(P)`

Returns `true`, if  $P$  contains a closed, valid coset table.

Note that if `HasClosedCosetTable` returns `true` for a coset enumeration process  $P$ , then in particular a call to `CosetTable` is legal for  $P$ .

<code>ExcludedConjugate(P)</code>
-----------------------------------

<code>ExcludedConjugates(P)</code>
------------------------------------

Given a coset enumeration process  $P$  with underlying group  $G$  and subgroup  $H$ , which contains a valid coset table, these functions return a set  $E$  containing either at most one word (`ExcludedConjugate`) or all words (`ExcludedConjugates`) of the form  $g_i^{-1}h_jg_i$ , where  $g_i$  is a generator of  $G$  and  $h_j$  is a generator of  $H$ , such that  $g_i^{-1}h_jg_i$  is not known to lie in  $H$  from the information contained in the coset table of  $P$ .

If  $E$  is empty, then  $H$  is a normal subgroup of  $G$ . Otherwise the addition of elements of  $E$  to the generators of  $H$  may yield a larger subgroup of the normal closure of  $H$  in  $G$ . In the case of a non-complete coset table it may happen, however, that excluded conjugates are found which actually lie in  $H$ .

These functions can be called for any coset enumeration process, which contains a valid coset table. If  $P$  does not contain a valid coset table, a call to `ExcludedConjugate` or `ExcludedConjugates` causes a runtime error. You can use the function `IsValidCosetTable` to check whether a call to these functions is legal for a certain coset enumeration process.

<code>ExistsCosetSatisfying(P : parameters)</code>
--

Given a coset enumeration process  $P$  with underlying group  $G$  and subgroup  $H$ , which contains a valid coset table, return whether or not there exists a coset other than  $H$ , which satisfies the conditions defined in the parameters. If such a coset exists, a representing word is returned as second return value. This function accepts the same set of parameters as the function `CosetSatisfying`; see there for a description.

This function can be called for any coset enumeration process, which contains a valid coset table. If  $P$  does not contain a valid coset table, a call to `ExistsCosetSatisfying` causes a runtime error. You can use the function `IsValidCosetTable` to check whether a call to `ExistsCosetSatisfying` is legal for a certain coset enumeration process.

<code>ExistsExcludedConjugate(P)</code>
---

Given a coset enumeration process  $P$  with underlying group  $G$  and subgroup  $H$ , which contains a valid coset table, return whether or not there exists a word of the form  $g_i^{-1}h_jg_i$ , where  $g_i$  is a generator of  $G$  and  $h_j$  is a generator of  $H$ , such that  $g_i^{-1}h_jg_i$  is not known to lie in  $H$  from the information contained in the coset table of  $P$ . If the answer is positive, such a word is returned as second return value.

This function can be called for any coset enumeration process, which contains a valid coset table. If  $P$  does not contain a valid coset table, a call to `ExistsExcludedConjugate` causes a runtime error. You can use the function `IsValidCosetTable` to check whether a call to `ExistsExcludedConjugate` is legal for a certain coset enumeration process.

Note that the coset table of  $P$  need not be complete to call this function. A negative result of `ExistsExcludedConjugate` always implies that  $H$  is a normal subgroup of  $G$ , even if the coset table of  $P$  is not complete. In the case of a non-complete coset table it may happen, however, that excluded conjugates are found which actually lie in  $H$ .

<code>ExistsNormalisingCoset(P)</code>
--

<code>ExistsNormalizingCoset(P)</code>
--

Returns `true`, if an element of  $G \setminus H$  which normalises  $H$  can be found from the coset table contained in  $P$ . (Here,  $G$  and  $H$  are the finitely presented group and the subgroup underlying  $P$ , respectively.) If the answer is positive, such an element is returned as second return value.

This function can be called for any coset enumeration process, which contains a valid coset table. If  $P$  does not contain a valid coset table, a call to `ExistsNormalisingCoset` causes a runtime error. You can use the function `IsValidCosetTable` to check whether a call to `ExistsNormalisingCoset` is legal for a certain coset enumeration process.

Note that the coset table of  $P$  need not be complete to call this function. However, no conclusion can be drawn from a negative result in the case of a non-complete coset table.

<code>Group(P)</code>
-----------------------

Returns the group underlying  $P$  as a finitely presented group.

<code>Index(P)</code>
-----------------------

Returns the index of  $H$  in  $G$ . (Here,  $G$  and  $H$  denote the finitely presented group and the subgroup underlying  $P$ , respectively.)

This function can only be called, if the last enumeration done for  $P$  has completed successfully with a finite index. Otherwise, a call to `Index` will cause a runtime error. Use the function `IsValidIndex` to check whether a call to `Index` is legal for a certain coset enumeration process.

<code>IsValidIndex(P)</code>
------------------------------

Returns `true`, if the last enumeration done for  $P$  has completed successfully with a finite index, i.e., if a call to `Index` is legal for the coset enumeration process  $P$ .

<code>MaximalNumberOfCosets(P)</code>
---------------------------------------

Returns the maximal number of cosets which were simultaneously active during the last enumeration done for  $P$ , or 1 if no enumeration has been done for  $P$ .

This function may be useful for assessing the performance of a certain set of enumeration parameters.

**Subgroup(P)**

Returns the subgroup  $H$  underlying the coset enumeration process  $P$ .  $H$  is returned as a subgroup of  $G$ , where  $G$  is the finitely presented group underlying  $P$ .

**TotalNumberOfCosets(P)**

Returns the total number of cosets defined during the last enumeration done for  $P$ , or 1 if no enumeration has been done for  $P$ .

This function may be useful for assessing the performance of a certain set of enumeration parameters.

**Example H71E3**

In the Harada-Norton sporadic simple group

$$\begin{aligned} < x, a, b, c, d, e, f, g \mid x^2, a^2, b^2, c^2, d^2, e^2, f^2, g^2, \\ & (x, a), (x, g), \\ & (bc)^3, (bd)^2, (be)^2, (bf)^2, (bg)^2, \\ & (cd)^3, (ce)^2, (cf)^2, (cg)^2, \\ & (de)^3, (df)^2, (dg)^2, \\ & (ef)^3, (eg)^2, \\ & (fg)^3, \\ & (b, xbx), \\ & (a, edcb), (a, f)dcbdcd, (ag)^5, \\ & (cdef, xbx), (b, xcdefx), (cdef, xcdefx) > \end{aligned}$$

we want to construct the coset table for the subgroup generated by  $x, b, c, d, e, f, g$  interactively. First, we create a coset enumeration process. Since the index of the chosen subgroup is 1 140 000, we request a coset limit of 1 200 000. Then we start the enumeration.

```
> HN<x, a, b, c, d, e, f, g> :=
>   Group< x, a, b, c, d, e, f, g |
>     x^2, a^2, b^2, c^2, d^2, e^2, f^2, g^2,
>     (x, a), (x, g),
>     (b*c)^3, (b*d)^2, (b*e)^2, (b*f)^2, (b*g)^2,
>     (c*d)^3, (c*e)^2, (c*f)^2, (c*g)^2,
>     (d*e)^3, (d*f)^2, (d*g)^2,
>     (e*f)^3, (e*g)^2,
>     (f*g)^3,
>     (b, x*b*x), (a, e*d*c*b), (a, f)*d*c*b*d*c*d,
>     (a*g)^5, (c*d*e*f, x*b*x), (b, x*c*d*e*f*x),
>     (c*d*e*f, x*c*d*e*f*x)
>   >;
> H := sub<HN | x,b,c,d,e,f,g >;
> P := CosetEnumerationProcess(HN, H : CosetLimit := 1200000, Print := true);
```

```
> StartEnumeration(~P);
Overflow
(a=1110331 r=58605 h=101193 n=1200001;
 l=5444 c=105.36;
 m=1110331 t=2001537)
```

The enumeration could not be completed successfully. Though  $P$  does contain a valid coset table, but this coset table fails to be closed.

```
> HasValidCosetTable(P);
true
> HasClosedCosetTable(P);
false
```

We extract the coset table map, check its domain and its codomain, and determine the position of the first “hole” in the coset table.

```
> ct := CosetTable(P);
> Domain(ct) : Minimal;
Cartesian Product<{ 1 .. 1110331 }, GrpFP: HN>
> Codomain(ct);
{ 0 .. 1110331 }
> row := 1;
> while forall(col){ gen : gen in {x, a, b, c, d, e, f, g}
> | ct(<row, gen>) ne 0 } do
>   row += 1;
> end while;
> row;
41881
> col;
x
```

We change the enumeration parameters for the process  $P$ , selecting the predefined strategy **Hard**. Since this predefined strategy sets a workspace of 10 000 000, we must expressly override this, if we want the old value to be retained. Because the workspace size never is decreased once an enumeration has been started, we can retain the old value simply by setting **Workspace** to 0; this overrides the setting done by selecting the predefined strategy, but actually doesn't change the process' workspace. We then continue the enumeration with the new set of parameters, after checking that this is legal.

```
> SetProcessParameters(~P : Strategy := "Hard",
> Workspace := 0);
> CanContinueEnumeration(P);
true
> ContinueEnumeration(~P);
INDEX = 1140000
(a=1140000 r=58512 h=1144534 n=1144534;
 l=70 c=6.50;
```

```
m=1140000 t=2035739)
```

We have obtained a closed coset table. Note that the codomain of the new coset table map does not contain 0.

```
> HasClosedCosetTable(P);
true
> ct := CosetTable(P);
> Domain(ct) : Minimal;
Cartesian Product<{ 1 .. 1140000 }, GrpFP: HN>
> Codomain(ct);
{ 1 .. 1140000 }
```

### Example H71E4

---

First, we create a coset enumeration process for the trivial subgroup of the finite group  $G := \langle a, b \mid a^8, b^7, (a * b)^2, (a^{-1} * b)^3 \rangle$  and start the enumeration.

```
> F<x, y> := FreeGroup(2);
> G<a, b> := quo<F | x^8, y^7, (x*y)^2, (x^-1*y)^3>;
> H := sub<G | >;
> P := CosetEnumerationProcess(G, H : Print := true);
> StartEnumeration(~P);
INDEX = 10752
(a=10752 r=57263 h=1 n=57263;
 l=292 c=0.11;
 m=47825t=57262)
```

We want to enumerate the cosets of the two non-trivial subgroups of  $G$  generated by  $a^{-1} * b$  and  $a^2$ , respectively. To do this, we create two copies of the coset enumeration process  $P$  and use the function `AddSubgroupGenerator` for each of them. Since the copies inherit all information contained in  $P$ , we then can call the function `RedoEnumeration` to enumerate the cosets of the two non-trivial subgroups, making use of the existing coset table.

```
> P1 := P;
> AddSubgroupGenerator(~P1, a^-1*b);
> Subgroup(P1);
Finitely presented group on 2 generators
Generators as words in group G
$.1 = Id(G)
$.2 = a^-1 * b
> CanRedoEnumeration(P1);
true
> RedoEnumeration(~P1);
INDEX = 3584
(a=3584 r=57263 h=1 n=57263;
 l=49 c=0.02;
 m=47825 t=57262)
>
> P2 := P;
```

```

> AddSubgroupGenerator(~P2, a^2);
> Subgroup(P2);
Finitely presented group on 2 generators
Generators as words in group G
  $.1 = Id(G)
  $.2 = a^2
> CanRedoEnumeration(P2);
true
> RedoEnumeration(~P2);
INDEX = 2688
(a=2688 r=57263 h=1 n=57263;
 l=37 c=0.02;
 m=47825 t=57262)

```

Finally, we are interested in the quotient of  $G$  by the normal closure of the subgroup generated by  $a^4$  and want to enumerate the cosets of the image of the subgroup generated by  $a^2$  in this quotient. Since this is the subgroup used in P2, we create a copy of P2 and add the relation  $a^4$ . Again, we are able to make use of information obtained earlier, by continuing the inherited enumeration.

```

> P3 := P2;
> AddRelator(~P3, a^4);
> CanContinueEnumeration(P3);
true
> ContinueEnumeration(~P3);
INDEX = 84
(a=84 r=57263 h=1 n=57263;
 l=2 c=0.00;
 m=47825 t=57262)

```

We extract the quotient and its subgroup from the process P3, using the appropriate access functions.

```

> G3<a3,b3> := Group(P3);
> G3;
Finitely presented group G3 on 2 generators
Relations
  a3^8 = Id(G3)
  b3^7 = Id(G3)
  (a3 * b3)^2 = Id(G3)
  (a3^-1 * b3)^3 = Id(G3)
  a3^4 = Id(G3)
> H3<u3, v3> := Subgroup(P3);
> H3;
Finitely presented group H3 on 2 generators
Generators as words in group G3
  u3 = Id(G3)
  v3 = a3^2

```

**Example H71E5**

---

Consider the subgroup  $H$  of the (infinite) group  $G := \langle a, b \mid b^7, (a * b)^2, (a^{-1} * b)^3 \rangle$  generated by  $a$ . We create a coset enumeration process and start an enumeration with the default parameters.

```
> F<x, y> := FreeGroup(2);
> G<a, b> := quo<F | y^7, (x*y)^2, (x^-1*y)^3>;
> H := sub<G | a>;
> P := CosetEnumerationProcess(G, H);
> StartEnumeration(~P : Print := true);
Overflow
(a=957026 r=415230 h=415230 n=999999;
 l=3553 c=2.38;
 m=960050 t=999998)
```

The enumeration produces a valid (albeit not complete) coset table.

```
> HasValidCosetTable(P);
true
> HasCompleteCosetTable(P);
false
```

Even the partial coset table is sufficient to find an element in the normaliser of  $H$  in  $G$ .

```
> found, elt := ExistsNormalisingCoset(P);
> found;
true
> elt;
b^-4 * a * b^-2
```

**Example H71E6**

---

Consider again the subgroup  $H$  of the (infinite) group  $G := \langle a, b \mid b^7, (a * b)^2, (a^{-1} * b)^3 \rangle$  generated by  $a$ . We create a coset enumeration process and start an enumeration with the default parameters.

```
> F<x, y> := FreeGroup(2);
> G<a, b> := quo<F | y^7, (x*y)^2, (x^-1*y)^3>;
> H := sub<G | a>;
> P := CosetEnumerationProcess(G, H);
> StartEnumeration(~P : Print := true);
Overflow
(a=957026 r=415230 h=415230 n=999999;
 l=3553 c=2.38;
 m=960050 t=999998)
```

We check, whether the coset table exhibits excluded conjugates.

```
> ExistsExcludedConjugate(P);
```

```
true a^b
```

It does. This means in particular, that  $H$  is not normal in  $G$ . We create a copy P1 of the coset enumeration process P, extend the subgroup of P1 by the excluded conjugates found in the previous step and restart the enumeration for P1.

```
> P1 := P;
> for c in ExcludedConjugates(P) do
>   AddSubgroupGenerator(~P1, c);
> end for;
> RedoEnumeration(~P1);
INDEX = 1 (a=1 r=2 h=2 n=2; l=2 c=0.94; m=960050 t=999998)
```

The new subgroup is equal to  $G$ . In particular, the normal closure of  $H$  in  $G$  is the whole of  $G$ . We return to the coset enumeration process P and check whether we can find a non-trivial element  $x \in \mathbf{N}_G(H)$  such that  $x^2 \in H$ .

```
> ExistsCosetSatisfying(P : Order := 2, Normalizing := true);
true b^-4 * a * b^-2
```

We can. In fact,  $b^{-4} \cdot a \cdot b^{-2}$  is the only non-trivial coset which is known to satisfy this condition...

```
> CosetsSatisfying(P : Order := 2, Normalizing := true);
{ Id(G), b^-4 * a * b^-2 }
```

... and we can't find in a similar way a non-trivial element  $x \in \mathbf{N}_G(H)$  such that  $x^3 \in H$ .

```
> ExistsCosetSatisfying(P : Order := 3, Normalizing := true);
false
```

Note the difference in the output of `CosetSatisfying` and `CosetsSatisfying`: The former takes into account only cosets other than  $H$ , whereas the latter (unless we set the parameter `First`) includes the coset  $H$ , which obviously satisfies the specified conditions.

```
> CosetSatisfying(P : Order := 3, Normalizing := true);
{}
> CosetsSatisfying(P : Order := 3, Normalizing := true);
{ Id(G) }
```

### 71.3.5 Induced Permutation Representations

Given a finite index subgroup  $H$  of a group  $G$ , the action of  $G$  on the set of right cosets of  $H$  in  $G$  by right multiplication defines a permutation representation  $\rho : G \rightarrow S$  of  $G$  onto a suitable subgroup  $S$  of the symmetric group on  $[G : H]$  letters. The kernel of  $\rho$  is the core of  $H$  in  $G$ , the maximal normal subgroup of  $G$  contained in  $H$ .

**CosetAction(P)**

Given a coset enumeration process  $P$  with underlying group  $G$  and subgroup  $H$  for which a valid finite index has been obtained, this function returns

- (a) The permutation representation  $\rho$  of  $G$ , induced by the action of  $G$  on the set of right cosets of  $H$  in  $G$ .
- (b) The image group  $\rho(G)$ .
- (c) (if possible) the kernel of  $\rho$ .

This function can only be called, if the last enumeration done for  $P$  has completed successfully with a finite index. Otherwise, a call to `CosetAction` will cause a runtime error. Use the function `IsValidIndex` to check whether a call to `CosetAction` is legal for a certain coset enumeration process.

**CosetImage(P)**

Given a coset enumeration process  $P$  with underlying group  $G$  and subgroup  $H$  for which a valid finite index has been obtained, this function returns the image of the permutation representation  $\rho$  of  $G$  induced by the action of  $G$  on the set of right cosets of  $H$  in  $G$  as a permutation group on  $[G : H]$  digits.

This function can only be called, if the last enumeration done for  $P$  has completed successfully with a finite index. Otherwise, a call to `CosetImage` will cause a runtime error. Use the function `IsValidIndex` to check whether a call to `CosetImage` is legal for a certain coset enumeration process.

**CosetKernel(P)**

Given a coset enumeration process  $P$  with underlying group  $G$  and subgroup  $H$  for which a valid finite index has been obtained, this function returns the kernel of the permutation representation  $\rho$  of  $G$  induced by the action of  $G$  on the set of right cosets of  $H$  in  $G$ . This function is only available if the index of  $H$  in  $G$  is sufficiently small.

This function can only be called, if the last enumeration done for  $P$  has completed successfully with a finite index. Otherwise, a call to `CosetKernel` will cause a runtime error. Use the function `IsValidIndex` to check whether a call to `CosetKernel` is legal for a certain coset enumeration process.

### 71.3.6 Coset Spaces and Transversals

The (right) indexed coset space  $V$  of the subgroup  $H$  of the group  $G$  is a  $G$ -set consisting of the set of integers  $\{1, \dots, m\}$ , where  $i$  represents some right coset  $c_i$  of  $H$  in  $G$ . The action of  $G$  on this  $G$ -set is that induced by the natural  $G$ -action

$$f : V \times G \rightarrow V$$

where

$$f : \langle c_i, x \rangle = c_k \iff c_i * x = c_k,$$

for  $c_i \in V$  and  $x \in G$ . If certain of the products  $c_i * x$  are unknown, the corresponding images under  $f$  are undefined, and  $V$  is called an *incomplete coset space* for  $H$  in  $G$ .

**CosetSpace(P)**

The coset space defined by the current state of the coset enumeration process  $P$ .

This function can be called for any coset enumeration process, which contains a valid coset table. (If  $P$  does not contain a valid coset table, a call to `CosetSpace` causes a runtime error. Use the function `IsValidCosetTable` to check whether a call to `CosetSpace` is legal for a certain coset enumeration process.) Note that the coset table of  $P$  need not be complete to use this function.

**RightCosetSpace(P)****LeftCosetSpace(P)**

The explicit right coset space of a subgroup  $H$  of some group  $G$  is a  $G$ -set containing the set of right cosets of  $H$  in  $G$ . The elements of this  $G$ -set are the pairs  $\langle H, x \rangle$ , where  $x$  runs through a transversal for  $H$  in  $G$ . Similarly, the explicit left coset space of  $H$  is a  $G$ -set containing the set of left cosets of  $H$  in  $G$ , represented as the pairs  $\langle x, H \rangle$ .

These functions return the explicit right (left) coset space defined by the current state of the coset enumeration process  $P$ .

This function can be called for any coset enumeration process, which contains a valid coset table. (If  $P$  does not contain a valid coset table, a call to any of these functions causes a runtime error. Use the function `IsValidCosetTable` to check whether a call to these functions is legal for a certain coset enumeration process.) Note that the coset table of  $P$  need not be complete to use these functions.

**Transversal(P)****RightTransversal(P)**

Given a coset enumeration process  $P$  with underlying group  $G$  and subgroup  $H$  for which a valid finite index has been obtained, these functions return

- (a) An indexed set  $T$  of elements of  $G$  forming a right transversal for  $H$  in  $G$ ; and
- (b) The corresponding transversal mapping  $\phi : G \rightarrow T$ . If  $T = \{ @ t_1, \dots, t_r @ \}$  and  $g \in G$ , then  $\phi$  is defined by  $\phi(g) = t_i$ , where  $g \in H * t_i$ .

These functions can only be called, if the last enumeration done for  $P$  has completed successfully with a finite index. Otherwise, a call to any of these functions will cause a runtime error. Use the function `IsValidIndex` to check whether a call to these functions is legal for a certain coset enumeration process.

**Example H71E7**

We construct a coset enumeration process for the subgroup  $H = \langle a^2, a^{-1}b \rangle$  in the group  $G = \langle a, b \mid a^8, b^7, (ab)^2, (a^{-1}b)^3 \rangle$  and start an enumeration.

```
> F<x, y> := FreeGroup(2);
> G<a, b> := quo<F | x^8, y^7, (x*y)^2, (x^-1*y)^3>;
> H := sub<G | a^2, a^-1*b>;
```

```
> P := CosetEnumerationProcess(G, H);
> StartEnumeration(~P);
```

After checking that a finite index has been obtained, we extract a transversal and the corresponding transversal map from P.

```
> HasValidIndex(P);
true
> T, f := Transversal(P);
> #T;
448
> f;
Mapping from: GrpFP: G to SetIndx: T
```

Finally, we construct the permutation representation of  $G$  on the cosets of  $H$  in  $G$ , its image and its kernel.

```
> r, S, K := CosetAction(P);
> r : Minimal;
Homomorphism of GrpFP: G into GrpPerm: S, Degree 448, Order 2^9 *
3 * 7
> S;
Permutation group S acting on a set of cardinality 448
Order = 10752 = 2^9 * 3 * 7
> K;
Finitely presented group K
Index in group G is 10752 = 2^9 * 3 * 7
Subgroup of group G defined by coset table
```

The kernel turns out to be trivial, i.e. the permutation representation is faithful.

```
> Order(K);
1
```

### Example H71E8

---

Consider the subgroup  $H$  of the (infinite) group  $G := \langle a, b \mid b^7, (a * b)^2, (a^{-1} * b)^3 \rangle$  generated by  $a$ . We create a coset enumeration process and start an enumeration with the default parameters.

```
> F<x, y> := FreeGroup(2);
> G<a, b> := quo<F | y^7, (x*y)^2, (x^-1*y)^3>;
> H := sub<G | a>;
> P := CosetEnumerationProcess(G, H);
> StartEnumeration(~P : Print := true);
Overflow
(a=957026 r=415230 h=415230 n=999999;
l=3553 c=2.38;
m=960050 t=999998)
```

The enumeration produces a valid (albeit not complete) coset table.

```
> HasValidCosetTable(P);
```

```

true
> HasCompleteCosetTable(P);
false

```

We extract the incomplete coset space from the process.

```

> V := CosetSpace(P);
> #V;
957026
> IsComplete(V);
false

```

## 71.4 $p$ -Quotients (Process Version)

Let  $F$  be a finitely presented group,  $p$  a prime and  $c$  a positive integer. A  $p$ -quotient algorithm constructs a consistent power-conjugate presentation for the largest  $p$ -quotient of  $F$  having lower exponent- $p$  class at most  $c$ . For details of this algorithm, see [NO96].

Assume that the  $p$ -quotient has order  $p^n$ , Frattini rank  $d$ , and that its generators are  $a_1, \dots, a_n$ . Then the power-conjugate presentation constructed has the following additional structure. The set  $\{a_1, \dots, a_d\}$  is a generating set for  $G$ . For each  $a_k$  in  $\{a_{d+1}, \dots, a_n\}$ , there is at least one relation whose right hand side is  $a_k$ . One of these relations is taken as the *definition* of  $a_k$ . (The list of definitions is also returned by `pQuotient`.) The power-conjugate generators also have a *weight* associated with them: a generator is assigned a weight corresponding to the stage at which it is added and this weight is extended to all normal words in a natural way.

The  $p$ -quotient process and its associated commands allows the user to construct a power-conjugate presentation (pcp) for a  $p$ -group.

### 71.4.1 The $p$ -Quotient Process

`pQuotientProcess(F, p, c: parameters)`

Given an fp-group  $F$ , a prime  $p$  and a positive integer  $c$ , create a  $p$ -quotient process for the group  $F$  with the indicated arguments. As part of the initialisation of the process, a pcp for the largest  $p$ -quotient of  $F$  having class at most  $c$  will be constructed. If  $c$  is given as 0, then the limit 127 is placed on the class. This function supports the same parameters as `pQuotient` and returns a process  $P$ .

NextClass( $\sim P$ : <i>parameters</i> )
---

NextClass( $\sim P$ , <i>k</i> : <i>parameters</i> )
--

Exponent	RNGINTELT	Default :
Metabelian	BOOLELT	Default :
Print	RNGINTELT	Default :
MaxOccurrence	[ RNGINTELT ]	Default : []

Assumes that a pcp has already been constructed for the class  $c$  quotient of  $F$ . It seeks to construct a pcp for the class  $c+1$   $p$ -quotient of  $F$ . If  $k$  is supplied, continue to construct until the pcp for the largest quotient of class  $k$  is constructed.

The parameters **Exponent**, **Print**, and **Metabelian** are used as before. If **MaxOccurrence** :=  $Q$ , then the sequence  $Q$  has length equal to the rank of the class 1 quotient of  $F$ ; its entries are integers which specify the maximum number of occurrences of the class 1 generators in the definitions of pcp generators of  $F$ . An entry of 0 for a particular generator indicates that no limit is placed on the number of occurrences of this generator.

Care should be exercised when supplying values for parameters. Once set, they retain their values until explicitly reassigned.

### 71.4.2 Using $p$ -Quotient Interactively

We assume that we have constructed a pcp for the largest class  $c$   $p$ -quotient of  $F$  and now seek to construct a pcp for the largest class  $c+1$   $p$ -quotient.

The following options allow the user to construct a pcp for the next class of the group interactively. The steps are laid out in one of a number of natural sequences in which they may be executed. Some of them may be interleaved; however, the user should pay particular attention to the assumptions mentioned below. The procedures that drive the process do not verify that the assumptions are satisfied.

StartNewClass( $\sim P$ : <i>parameters</i> )
---

If  $P$  is a process for a class  $c$   $p$ -quotient, commence construction of class  $c+1$ .

Tails( $\sim P$ : <i>parameters</i> )
---------------------------------------

Tails( $\sim P$ , <i>k</i> : <i>parameters</i> )
--

Metabelian	BOOLELT	Default : false
------------	---------	-----------------

Add tails to the current pcp; default is to add all tails for this class. If  $k$  is supplied, then tails for weight  $k$  only are added; in this case, it is assumed that the tails for each of weight  $c+1, c, \dots, k+1$  have already been added. The valid range of  $k$  is  $2, \dots, c+1$ . The one valid parameter is **Metabelian**; if **true**, then only the tails for the metabelian  $p$ -quotient are inserted.

Consistency( $\sim P$ : <i>parameters</i> )
---

Consistency( $\sim P$ , <i>k</i> : <i>parameters</i> )
--

**Metabelian**

BOOLELT

*Default : false*

Apply the consistency algorithm to the pcp to compute any redundancies among the tails already added. Default is to apply it to all tails; in this case, it is assumed that all tails have been added. If *k* is supplied, it is assumed that tails for weight *k* have been added; in this case, the tails added for weight *k* only are checked. The range of *k* is  $3, \dots, c+1$ . The one valid parameter is **Metabelian**; if **true**, we assume that the tails inserted were those for a metabelian *p*-quotient and hence invoke the (less expensive) metabelian consistency algorithm.

CollectRelations( $\sim P$ )
------------------------------

Collect the defining relations (if any) in the current pcp. If the tails operation is not complete, then the relations may be evaluated incorrectly.

ExponentLaw( $\sim P$ : <i>parameters</i> )
---

ExponentLaw( $\sim P$ , <b>Start</b> , <b>Fin</b> : <i>parameters</i> )
---

Enforce the supplied exponent law on the current pcp. If **Start** and **Fin** are supplied, then enforce the law for those weights between **Start** and **Fin**; otherwise, enforce the law for all weights. It is assumed that the tails operation is complete. If the display parameter **DisplayLevel** (which may be set using **SetDisplayLevel**) has value 2, those words whose powers are collected and give redundancies among the pcp generators are printed out. If **DisplayLevel** has value 3, all words whose powers are collected are printed out. The following additional parameters are available:

**Exponent**

RNGINTELT

*Default : 0*

If **Exponent** := *m*, enforce the exponent law,  $x^m = 1$ , on the group.

**Print**

RNGINTELT

*Default : 1*

As for **pQuotient**.

**Trial**

BOOLELT

*Default : false*

Generate the list of words used to enforce the exponent law and print out statistics but do not power words or echelonise the results.

**ShortList**

BOOLELT

*Default : false*

Generate the list of enforcement words whose entries have the form *w* or  $1 * w$  where *w* is an element of the Frattini subgroup of *F*.

**DisplayList**

BOOLELT

*Default : false*

Display the list of all enforcement words generated – not just those which are collected.

**IdentifyFilters**

BOOLELT

*Default : false*

Identify filters used to eliminate words from list.

**InitialSegment** [`<GRPFPELT, RNGINTELT>`] *Default* : []

If **InitialSegment** := *w*, generate only those enforcement words which have *w* as an initial segment, where *w* is supplied as a sequence of generator-exponent pairs.

**Report** `RNGINTELT` *Default* : 0

If **Report** := *n*, report after computing the powers of each collection of *n* enforcement words.

**EliminateRedundancy( $\sim P$ )**

Eliminate all redundant generators from the pcp defined by process *P*. This operation may be performed at any time.

We now list the remaining functions which can be applied to a *p*Quotient process.

**Display(*P*)**

**Display(*P*, *DisplayLevel*)**

Display the pcp for the *p*-quotient *G* of the fp-group *F*. The argument **DisplayLevel** may be 1, 2, or 3, and is used to control the amount of information given:

1 : Display order and class of *G*;

2 : Display non-trivial relations for *G*;

3 : Display the structure of pcp generators of *G*, non-trivial relations of *G*, and the map from the defining generators of *F* to the pcp generators of *G*.

The presentation displayed by this function is in power-commutator form. If **DisplayLevel** is not supplied, the information displayed is determined by its existing (or default) value.

**RevertClass( $\sim P$ )**

Given a pcp for the class *c* + 1 *p*-quotient of *F*, this procedure reverts to the pcp for the class *c* *p*-quotient of *F*. Note that this command can be applied only **once** during construction of a single class.

**pCoveringGroup( $\sim P$ )**

**pCoveringGroup(*G*)**

Given a process or a pcp for a *p*-group, this procedure computes a pcp for the *p*-covering group of this group. In the process case, it is equivalent to **Tails( $\sim P$ )**; **Consistency( $\sim P$ )**; **EliminateRedundancy( $\sim P$ )**.

**GeneratorStructure(*P*)**

**GeneratorStructure(*P*, *Start*, *Fin*)**

Display the structure of the generators in the pcp. If **Start** and **Fin** are given, then print out the structure of those pcp generators numbered from **Start** to **Fin**.

`Jacobi( $\sim P$ ,  $c$ ,  $b$ ,  $a$ ,  $\sim r$ )`

`Jacobi( $\sim P$ ,  $c$ ,  $b$ ,  $a$ )`

Calculate the Jacobi  $c, b, a$  and echelonise the resulting relation against the current pcp. If a redundant generator results from the echelonisation, the optional variable  $r$  is the number of that generator; otherwise  $r$  has value 0.

`Collect( $P$ ,  $Q$ )`

The sequence  $Q$ , consisting of generator-exponent pairs, defines a word  $w$  in the pcp generators of the group defined by the process  $P$ . Collect this word and return the resulting normal word as an exponent vector.

`EcheloniseWord( $\sim P$ ,  $\sim r$ )`

`EcheloniseWord( $\sim P$ )`

Echelonise the word most recently collected using `Collect` against the relations of the pcp. If a redundant generator results from the echelonisation, the optional variable  $r$  is the number of that generator; otherwise  $r$  has value 0. This function must be called immediately after `Collect`.

`SetDisplayLevel( $\sim P$ ,  $Level$ )`

This procedure alters the display level for the process to the supplied value,  $Level$ .

`ExtractGroup( $P$ )`

Extract the group  $G$  defined by the pcp associated with the process  $P$ , as a member of the category `GrpPC` of finite soluble groups. The function also returns the natural homomorphism  $\pi$  from the original group  $F$  to  $G$ , a sequence  $S$  describing the definitions of the pc-generators of  $G$  and a flag indicating whether  $G$  is the maximal  $p$ -quotient of  $F$ .

The  $k$ -th element of  $S$  is a sequence of two integers, describing the definition of the  $k$ -th pc-generator  $G.k$  of  $G$  as follows.

- If  $S[k] = [0, r]$ , then  $G.k$  is defined via the image of  $F.r$  under  $\pi$ .
- If  $S[k] = [r, 0]$ , then  $G.k$  is defined via the power relation for  $G.r$ .
- If  $S[k] = [r, s]$ , then  $G.k$  is defined via the conjugate relation involving  $G.r^{G.s}$ .

`Order( $P$ )`

The order of the group defined by the pcp associated with the process  $P$ .

`FactoredOrder( $P$ )`

The factored order of the group defined by the pcp associated with the process  $P$ .

`NumberOfPCGenerators( $P$ )`

The number of pc-generators of the group defined by the pcp associated with the process  $P$ .

pClass(P)
-----------

The lower exponent- $p$  class of the group defined by the pcp associated with the process  $P$ .

NuclearRank(G)
----------------

NuclearRank(P)
----------------

Return the rank of the  $p$ -multiplicator of the  $p$ -group  $G$ , where  $G$  may be supplied or defined by the process  $P$ .

pMultiplicatorRank(G)
-----------------------

pMultiplicatorRank(P)
-----------------------

Return the rank of the  $p$ -multiplicator of the  $p$ -group  $G$ , where  $G$  may be supplied or defined by the process  $P$ .

### Example H71E9

---

Starting with an exponent 9 group having two generators of order 3, we set up the largest class 4 quotient and then interactively compute two more classes.

```
> G<a, b> := Group<a, b | a^3, b^3>;
> q := pQuotientProcess(G, 3, 4: Exponent := 9, Print :=1);
```

Lower exponent-3 central series for G

Group: G to lower exponent-3 central class 1 has order 3<sup>2</sup>

Group: G to lower exponent-3 central class 2 has order 3<sup>3</sup>

Group: G to lower exponent-3 central class 3 has order 3<sup>5</sup>

Group: G to lower exponent-3 central class 4 has order 3<sup>7</sup>

```
> Display(q, 2);
```

Group: G to lower exponent-3 central class 4 has order 3<sup>7</sup>

Non-trivial powers:

.3<sup>3</sup> = .6<sup>2</sup>

Non-trivial commutators:

[ .2, .1 ] = .3

[ .3, .1 ] = .4

[ .3, .2 ] = .5

[ .4, .1 ] = .6

[ .4, .2 ] = .7

[ .5, .1 ] = .7

[ .5, .2 ] = .6

We construct the class 5 quotient using the function `NextClass`.

```
> NextClass(~q);
```

Group: G to lower exponent-3 central class 5 has order 3<sup>9</sup>

We now construct the class 6 quotient step by step. For this, we set the output level to 1.

```
> SetDisplayLevel(~q, 1);
```

Now we start the next class.

```
> StartNewClass(~q);
```

The first step is to add the tails.

```
> Tails(~q);
```

After that, we apply the consistency algorithm,...

```
> Consistency(~q);
```

... collect the defining relations,...

```
> CollectRelations(~q);
```

... and enforce the exponent law.

```
> ExponentLaw(~q);
```

Finally, we eliminate redundant generators.

```
> EliminateRedundancy(~q);
```

This results in the following presentation for class 6 quotient.

```
> Display(q, 2);
```

Group: G to lower exponent-3 central class 6 has order 3<sup>11</sup>

Non-trivial powers:

.3<sup>3</sup> = .6<sup>2</sup> .8<sup>2</sup> .10 .11

Non-trivial commutators:

[ .2, .1 ] = .3

[ .3, .1 ] = .4

[ .3, .2 ] = .5

[ .4, .1 ] = .6

[ .4, .2 ] = .7

[ .4, .3 ] = .8 .10<sup>2</sup> .11<sup>2</sup>

[ .5, .1 ] = .7 .8 .9<sup>2</sup> .10<sup>2</sup> .11<sup>2</sup>

[ .5, .2 ] = .6 .8 .9<sup>2</sup> .10<sup>2</sup>

[ .5, .3 ] = .9<sup>2</sup> .11

[ .5, .4 ] = .10 .11

```
[ .6, .2 ] = .10^2
[ .7, .1 ] = .8
[ .7, .2 ] = .9
[ .7, .3 ] = .10^2 .11
[ .8, .2 ] = .10
[ .9, .1 ] = .11
```

---

**Example H71E10**

Starting with the free product of two cyclic groups of order 5, we bound the number of occurrences of pcg generators of the class 1 quotient in definitions of new pcg generators.

We start with setting up the class 1 quotient of the group.

```
> G := Group<a, b | a^5, b^5>;
> q := pQuotientProcess(G, 5, 1);
> Display(q, 1);
Group: G to lower exponent-5 central class 1 has order 5^2
```

Now we start the next class, setting bounds on the number of occurrences of the pcg generators of the class 1 quotient in the definitions of new pcg generators.

```
> NextClass(~q, 6: MaxOccurrence := [3, 2]);
Group: G to lower exponent-5 central class 2 has order 5^3
Group: G to lower exponent-5 central class 3 has order 5^5
Group: G to lower exponent-5 central class 4 has order 5^7
Group: G to lower exponent-5 central class 5 has order 5^9
> Display(q, 2);
Group: G to lower exponent-5 central class 5 has order 5^9
Non-trivial powers:
Non-trivial commutators:
[ .2, .1 ] = .3
[ .3, .1 ] = .4
[ .3, .2 ] = .5
[ .4, .1 ] = .6
[ .4, .2 ] = .7
[ .4, .3 ] = .8^4 .9
[ .5, .1 ] = .7 .8^4 .9
[ .6, .2 ] = .8
[ .7, .1 ] = .9
```

---

**Example H71E11**

We construct the class 6 quotient  $q$  of  $R(2, 5)$  and then partially construct the class 7 quotient interactively to find out how many normal words having initial segment  $q.1^2$  need to be considered when imposing the exponent law.

```
> F := FreeGroup(2);
> q := pQuotientProcess(F, 5, 6: Exponent := 5);
Lower exponent-5 central series for F
```

```

Group: F to lower exponent-5 central class 1 has order 5^2
Group: F to lower exponent-5 central class 2 has order 5^3
Group: F to lower exponent-5 central class 3 has order 5^5
Group: F to lower exponent-5 central class 4 has order 5^8
Group: F to lower exponent-5 central class 5 has order 5^10
Group: F to lower exponent-5 central class 6 has order 5^14
> StartNewClass(~q);
> Tails(~q);
> Consistency(~q);
> SetDisplayLevel(~q, 3);
> ExponentLaw(~q, 1, 6: InitialSegment := [<1, 2>], Trial := true);
0 Relations of class 1 will be collected
0 Relations of class 2 will be collected
Will collect power 5 of the following word: 1^2 2^1
1 Relation of class 3 will be collected
Will collect power 5 of the following word: 1^2 2^2
1 Relation of class 4 will be collected
Will collect power 5 of the following word: 1^2 2^3
Will collect power 5 of the following word: 1^2 5^1
2 Relations of class 5 will be collected
Will collect power 5 of the following word: 1^2 2^4
Will collect power 5 of the following word: 1^2 2^1 4^1
2 Relations of class 6 will be collected

```

### Example H71E12

---

We demonstrate several of the remaining procedures in the course of interactively extending the class 6 quotient of the group

$$\langle a, b, c, d \mid (bc^{-1}d)^7, (cd^{-1})^7, (b, a) = c^{-1}, (c, a) = 1, (c, b) = d^{-1} \rangle$$

to class 7.

```

> G := Group<a, b, c, d | (b * c^-1 * d)^7, (c * d^-1)^7, (b,a) = c^-1,
>
(c,a) = 1, (c,b) = d^-1>;
> q := pQuotientProcess(G, 7, 6);
Lower exponent-7 central series for G
Group: G to lower exponent-7 central class 1 has order 7^2
Group: G to lower exponent-7 central class 2 has order 7^4
Group: G to lower exponent-7 central class 3 has order 7^6
Group: G to lower exponent-7 central class 4 has order 7^8
Group: G to lower exponent-7 central class 5 has order 7^11
Group: G to lower exponent-7 central class 6 has order 7^14
> StartNewClass(~q);
> Tails(~q);
> GeneratorStructure(q, 15, 34);
Class 7
15 is defined on [12, 1] = 2 1 2 2 1 2 1

```



## 71.5 Soluble Quotients

### 71.5.1 Introduction

This section presents a short overview towards the theory of the soluble quotient algorithm, the functions designed for computing soluble quotients and the functions designed for dealing with soluble quotient processes.

### 71.5.2 Construction

For a finite group  $G$ , the property of being soluble means that the derived series  $G = G^{(0)} > G^{(1)} > \dots > G^{(n)}$  terminates with  $G^{(n)} = \langle 1 \rangle$ . Each section  $G^{(i)}/G^{(i+1)}$  is a finite abelian group, hence it can be identified with a  $\mathbf{Z}G/G^{(i)}$ -module  $M$ , where the action of  $G/G^{(i)}$  on  $M$  is given by the conjugation action on  $G^{(i)}/G^{(i+1)}$ .

From module theory we see that there exists a series  $M = M^{(0)} > M^{(1)} > \dots > M^{(r_i)}$ , where each section is an irreducible  $GF(p)$   $G/G^{(i)}$ -module for some prime  $p$ . Using these series we obtain a refinement  $G = G_{(0)} > G_{(1)} > \dots > G_{(n)} = \langle 1 \rangle$  of the commutator series with the properties:

- The series is normal,
- Each section  $G_{(i)}/G_{(i+1)}$  is elementary abelian of prime power order, and is irreducible as a  $G/G_{(i)}$ -module.
- If  $G = H_{(0)} > H_{(1)} > \dots > H_{(t)} = \langle 1 \rangle$  is another series with these properties, then  $n = t$  and there exists a permutation  $\pi \in S_n$  such that  $G_{(i)}/G_{(i+1)}$  is isomorphic to  $H_{(i\pi)}/H_{(i\pi+1)}$  (as  $GF(p)$  modules).

**Note:** A PC-presentation defined by a further refinement of this series leads to a “conditioned” presentation. The converse is not always true, because the irreducibility of the sections is not required for a conditioned presentation.

The soluble quotient algorithm uses these series to construct soluble groups. Starting with the trivial group  $G/G_{(0)}$ , it successively chooses irreducible  $G/G_{(i)}$  modules  $M_i$  and extensions

$$\zeta_i \in H^2(G/G_{(i)}, M_i)$$

which give rise to exact sequences

$$1 \rightarrow M_i \rightarrow G/G_{(i+1)} = G/G_{(i)}.M_i \rightarrow G/G_{(i)} \rightarrow 1.$$

To describe the algorithmic approach, we consider the following situation. Let  $G$  be a finite soluble group,  $M$  a normal elementary abelian subgroup of  $G$  such that  $M$  is a  $H = G/M$  irreducible module. (The action of  $H$  on  $M$  is identified with the conjugation action of  $G/M$  on the subgroup  $M$ .) Then the relations of the group  $G$  have the shape

$$g_i^{p_i} = w_i m_i \text{ resp. } g_j^{q_j} = w_{ij} m_{ij}, m_i, m_{ij} \in M$$

where  $w_i, w_{ij}$  are the canonical representatives of  $H$  in  $G$ . Then  $\bar{g}_i^{p_i} = \bar{w}_i$  resp.  $\bar{g}_j^{q_j} = \bar{w}_{ij}$  is a PC-presentation of  $H$  and the set of images  $\left( (g_i^{p_i-1}, g_i) \rightarrow m_i, (g_j, g_i) \mapsto m_{ij} \right)$  determines a unique element of the cocycle space  $C^2(H, M)$ .

According to the  $p$ -group situation, we call the system  $t = (m_i, m_{ij})$ , the tail defining  $G = H.M_t$  as an extension of  $M$  by  $H$ .

Not every choice of a system  $t = (m_i, m_{ij})$  defines an extension. To make sure that  $t$  corresponds to an element  $s$  of  $C^2(G, M)$  it must satisfy a certain equation system, the so-called consistency equations (see Vaughan-Lee). These are linear homogeneous equations in  $M$ , so the solution space can be determined.

For the construction of soluble quotients we also have to find the epimorphism  $\varepsilon : F \rightarrow G$ . The existence of the epimorphism is obviously a restriction of the possible images  $G$ , and it can be checked simultaneously with constructing  $G$ . Let  $G$  be an extension  $G = H.M$ , where  $M$  is a  $H$ -module and  $H$  is a known soluble quotient  $\delta : F \rightarrow H$ . Then  $\delta$  is uniquely determined by the images  $\delta(f_i) = h_i, 1 \leq i \leq r$ , where  $\{f_1, \dots, f_r\}$  is a generating set of  $F$ . We want to find a lift  $\varepsilon$  of  $\delta$ , i.e.  $\varepsilon(f) = \delta(f) \bmod M$  for all  $f \in F$ . This means that  $\varepsilon(f_i) = h_i x_i$  for all  $i \leq r$ , where  $x_i \in M$  are to be determined. Since  $\varepsilon$  will be a homomorphism, we require  $\varepsilon(r_j(f_1, \dots, f_r)) = 1_G$  for the defining relations  $r_j$  of  $F$ . This leads to a linear equation system for the variables  $(x_1, \dots, x_r) \in M^r$ . We solve this equation system together with the consistency equations, hence we find a subspace  $S$  of  $M^r \times H^2(H, M)$  of those extensions for which the epimorphism  $\delta$  has a lift. Let us call an element of  $S$  an extended tail.

Let  $K$  be the minimal splitting field of  $M$ , i.e. the character field of the unique character of  $H$  which corresponds to  $M$ . Then  $M$  is obviously a  $KH$ -module and  $S$  is also a  $K$ -space. The space  $S$  has a  $K$ -subspace  $S_S$  of split extensions, i.e. the projection of  $S_S$  into  $H^2(H, M)$  is only the trivial element. For any element  $t \in S/S_S$  the corresponding map  $\varepsilon_t : F \rightarrow H.M_t$  is necessarily surjective, hence defines a soluble quotient. Let  $s_1, s_2$  be two elements of  $S/S_S$  and let  $\varepsilon_i : F \rightarrow H.M_{s_i} =: G_i$  denote the corresponding soluble quotients. If  $s_1$  and  $s_2$  are  $K$ -linear dependent, the groups  $G_1$  and  $G_2$  will be isomorphic. Unfortunately, the converse is not true, even  $K$ -linear independent elements may lead to isomorphic groups. Nevertheless, if  $s_1$  and  $s_2$  are independent, the kernels of the epimorphisms will be different, hence one can iterate the lifting to

$$\varepsilon_{1,2} : F \rightarrow (G_1).M_{s_2} = (H.M_{s_1}).M_{s_2} = H.(M \oplus M)_{s_1 \cdot s_2}.$$

(The last equality can be read as a definition of  $s_1 \cdot s_2$  in the righthand side term.)

The dimension  $a = \dim_K(S/S_S)$  is therefore characterised as the maximal multiplicity

$$a = \max \{z \in \mathbf{Z}^+ \mid \exists \varepsilon : F \rightarrow H, M^z\}.$$

$S_S$  itself also has a  $K$ -subspace  $S_C$ , for which the map

$$\varepsilon_s : F \rightarrow H.M_s, s \in S_C$$

is not surjective. The  $K$ -dimension  $b = \dim_K(S_S/S_C)$  is again characterised as the maximal multiplicity

$$b = \max \{z \in \mathbf{Z}^+ \mid \exists \varepsilon : F \rightarrow H, M^z\}.$$

Moreover, after taking a maximal extension  $\varepsilon : F \rightarrow H.M^b$ ,  $M$  never has to be considered for split extensions again.

### 71.5.3 Calculating the Relevant Primes

A crucial step in finding finite soluble quotients  $\varepsilon : F \twoheadrightarrow G$  is the calculation of the relevant primes, i.e. the prime divisors of  $|G|$ . This is the most time consuming part of the algorithm, so it is crucial to apply it as efficiently as possible, and any other information about possible prime divisors is very helpful. We do not explain this calculation in detail. However, to use the functions and options provided by MAGMA in a correct manner, we outline the idea of the calculation.

Let  $\varepsilon : F \twoheadrightarrow G$  be a finite soluble quotient and let  $N$  denote the kernel of  $\varepsilon$ . If a module  $M$  allows an extension  $\bar{\varepsilon} : F \twoheadrightarrow G.M$ , then  $M$  must be a constituent of  $N/N'$ , and the relevant primes are the prime divisors of  $N/N'$ . Again  $N/N'$  can be viewed as an  $F/N$ -module, hence an  $H$ -module. Therefore it is a direct sum  $N/N' \cong M_{p_1} \oplus M_{p_2} \oplus \dots \oplus \mathbf{Z}^d$ , where the  $M_{p_i}$  are finite modules of order a power of the prime  $p_i$ . Let  $p$  not divide  $|H|$ . Then by Zassenhaus the extension  $H.M_p$  must split. We consider an irreducible module  $M$  in the head of  $M_p$ , i.e. there is an  $H$ -epimorphism  $M_p \twoheadrightarrow M$ . Then there is a valid soluble quotient  $\varepsilon : F \twoheadrightarrow H.M$  and the extension  $H.M$  splits.

Now there exists an irreducible  $\mathbf{Z}H$ -module  $L$  such that  $M$  is a  $GF(p)$ -modular constituent of  $L/pL$ . Moreover,  $\Delta : H \rightarrow GL(L)$  is the corresponding representation of  $H$  and any  $\Delta$ -module will have this property.

Now using the theory of space groups, one can construct homomorphisms  $\varepsilon_1 : F \twoheadrightarrow H.L$  and  $\varepsilon_2 : H.L \twoheadrightarrow H.M$  such that the composition is surjective. Hence  $p$  can be detected in  $\Delta$ . Let  $P_\Delta$  denote the set of primes obtained from  $\Delta$ . To find these primes, we have to know a set  $D$  of representatives of the irreducible rational representations of  $F/N$ , hence of  $H$ . The set of prime divisors of  $K/K'$  is a subset of

$$P = \bigcup_{\Delta \in D} P_\Delta \cup \{p \mid p \text{ prime, } p \mid |G|\}.$$

We want to point out the following:

- Usually the set of primes dividing  $|K/K'|$  is a proper subset of  $P$ , because the primes dividing  $|G|$  may or may not divide  $|K/K'|$ .

Conversely, if  $p$  does not divide  $|G|$ , then there certainly exists a module  $M$  in characteristic  $p$  such that there is a soluble quotient  $F \twoheadrightarrow G.M$ , and the extension splits.

- The algorithm can also recognise infinite abelian sections. This means that already  $\varepsilon_1 : F \twoheadrightarrow H.L$  is surjective. All primes are relevant and no maximal finite soluble quotient exists.
- Let  $\phi : F \twoheadrightarrow H$  be a lift of  $\varepsilon : F \twoheadrightarrow G$ , i.e.  $G$  is a quotient of  $H$ :  $\phi : H \twoheadrightarrow G$ . If the relevant primes of  $G$  are known, these are also relevant primes of  $H$ , and only those representations  $\Delta$  of  $H$  must be considered for which  $\ker \phi \not\subseteq \ker \Delta$  is valid.

### 71.5.4 The Functions

MAGMA provides two different ways to calculate finite soluble quotients: a main function for the whole calculation, and a process which gives control over each individual step.

Let  $F$  be a finitely presented group.

SolubleQuotient(F, n : parameters)
------------------------------------

SolvableQuotient(F, n : parameters)
-------------------------------------

SolubleQuotient(F : parameters)
---------------------------------

SolvableQuotient(F : parameters)
----------------------------------

SolubleQuotient(F, P : parameters)
------------------------------------

SolvableQuotient(F, P : parameters)
-------------------------------------

Find a soluble quotient  $\varepsilon : F \twoheadrightarrow G$  with a specified order.  $n$  must be a nonnegative integer.  $P$  must be a set of primes.

The three forms reflect possible information about the order of an expected soluble quotient. In the first form the order of  $G$  is given by  $n$ , if  $n$  is greater than zero. If  $n$  equals zero, nothing about the order is known and the relevant primes will be calculated completely.

The second form, with no  $n$  argument, is equivalent to the first with  $n = 0$ . This is a standard argument, and usually it is the most efficient way to calculate soluble quotients.

Note that, if  $n > 0$  is not the order of the maximal finite soluble quotient, it may happen that no group of order  $n$  can be found, since an epimorphic image of size  $n$  may not be exhibited by the chosen series.

In the third form a set  $P$  of relevant primes is given. The algorithm calculates the biggest quotient such that the order has prime divisors only in  $P$ .  $P$  may have a zero as element, this is just for consistency reasons. It is equivalent to the first form with  $n$  equal zero.

The returned values are the group  $G$  and the epimorphism  $\varepsilon : F \twoheadrightarrow G$ . The third returned value is a sequence describing the series and modules by which  $G$  has been constructed.

The fourth value is a string which explain the reason for termination. The following list gives the termination conditions. The algorithm terminates normally if:

1. A quotient of the given order has been constructed.
2. A maximal quotient (with respect to the conditions given on the order) has been constructed.

The algorithm will be aborted and returns a warning if:

3. A bound on the length of a series or subseries has been hit.
4. A limit on the size of the quotient or a section has been hit.
5. The algorithm detects a free abelian section.

With the following options one can define abort conditions corresponding to the third and fourth item. The idea of all these conditions is to control the occurrence of infinite soluble quotients.

**SeriesLength**

RNGINTELT

*Default : 0*

Limits the length of the chief series to  $r$ . For sag-series it is the nilpotent length of the series, for derived series it is the derived length. The default value of zero means no limit.

If the algorithm hits the limit, it gives a warning message and returns the last soluble quotient.

**SubseriesLength**                      RNGINTELT                      *Default : 0*

Limits the length of a series in a section. If the sag-series is used, it is the length of the lower descending series. For the derived series, the limit is applied to the exponent of an element of a section with maximal prime power order. For example, if the limit is set to 3, the limit would be hit by a section of type  $C_8$ , but not by  $C_2^3$  and not even by  $C_4^2$ .

**#QuotientSize**                      RNGINTELT                      *Default : 0*

If the value  $n$  is bigger than zero, the algorithm returns if a quotient of order bigger or equal to  $n$  has been found. A warning message will be printed.

**#SectionSize**                      RNGINTELT                      *Default : 0*

If the value  $s$  is bigger than zero, the algorithm returns if the order of a section is bigger than or equal to  $s$ . A warning message will be printed.

**Note:** Since an additive bound on a group order is not as meaningful as a multiplicative bound, the latter options are only useful as break conditions when the quotient gets too big for further calculations. The return quotient which hits such a bound is somewhat randomly chosen, since only a change in the order of checking modules may lead to other quotients.

With the following options the strategy of the algorithm and some subalgorithms can be chosen.

**#MSQ\_Series**                      MONSTGELT                      *Default : "sag"*

Determines the series which is used for the construction of soluble groups.

The default value is "sag", since it is usually the most efficient choice. Of course, there is a value "derived", exhibiting the derived series.

Another choice is "lowercentral", choosing the lower central series. This restricts the algorithm to finite nilpotent quotients. The choice "pcentral" only exhibits  $p$ -groups as quotients.

The nilpotent resp.  $p$ -quotient algorithms are usually more efficient, so these options may only be useful to obtain additional information needed for a SQ-process.

**MSQ\_PrimeSearchModus**                      RNGINTELT                      *Default : 3*

Defines at what status of the algorithm the relevant prime search is called.

The possible choices reflect the different intentions of constructing a soluble quotient; for the general situation, (i.e. finding a finite soluble quotient without any information about relevant primes and check its maximality) this option makes only little difference in runtime behaviour.

- 0: No calculation of relevant primes. This is the default value, if the second argument does not request a prime calculation, e.g. if the order of the quotient or its relevant primes are known.
- 1: The relevant primes will be calculated after the soluble quotient algorithm (with the given input) terminates normally. Possibly new relevant primes are returned in a message. If, for example, the second argument is a set  $S$  (so no limit on the exponent) and no new relevant primes have been found, the maximality of the soluble quotient is proved.
- 2: As 1, but continues the algorithm when finding new relevant primes.
- 3: Perform the relevant prime calculation after a “main” step in the series, i.e. after completing a nilpotent section in a sag-series resp. a commutator section for the derived series. This is the default value, if prime calculation is required by the second argument. It is a good choice if one wants to construct large finite quotients quickly.

**Note:** This option can cause problems in case of a sag-series when infinite soluble quotients exist. For finite quotients, it seems to be the best choice.

- 4: Perform the relevant prime calculation after calculating an elementary abelian layer. This option is preferable, if the sag-series is used and an infinite soluble quotient is possible.
- 5: Perform the relevant prime calculation after each successful lift of a quotient. This option is preferable when infinite sections shall be detected as soon as possible (with respect to the chosen series).

`MSQ_ModulCalcModus`      `RNGINTELT`      *Default : 0*

In the construction of soluble quotients using a sag-series one can restrict the number of modules by using tensor products and skew symmetric products. This can improve the performance in the case of big soluble quotients, for small quotients the overhead may invalidate the improvement. For other series this option has no meaning. The possible values are:

- 0: Do not apply this technique (default).
- 1: Fast version, just apply those parts which can be calculated quickly.
- 2: Full version, this is only recommended for “big” soluble quotients, i.e. quotients with long descending series in nilpotent sections.

`MSQ_CollectorModus`      `RNGINTELT`      *Default : 2*

Defines the setup modus for the symbolic collector, i.e. the ratio of precalculation to dynamic setup:

- 0: Full precalculation, preferable for small soluble groups.
- 1: Partial precalculation (test version).
- 2: Dynamic setup (default).

The function also provides a general print option determining the amount of timings status information during the function call. Additionally there are some

verbose flags which determine the amount of information given about various sub-algorithms. If both a general print value and a verbose flag are given, the verbose flag has higher preference.

<b>Print</b>	RNGINTELT	<i>Default : 0</i>
Determines what timing information and status messages are given during the calculation (0 = no printing, 5 = maximal information).		
Verbose	MSQ_Messages	<i>Maximum : 2</i>
If set to 1, the sizes of new soluble quotients are printed.		
Verbose	MSQ_PrimeSearch	<i>Maximum : 15</i>
Bitflag for print levels during the calculation of relevant primes:		
1: Timings and statistics about the calculation of rational representations.		
2: Timings for transforming rational into integral representations.		
4: Timing for finding the relevant primes.		
8: Printing of new relevant primes.		
Verbose	MSQ_RepsCheck	<i>Maximum : 3</i>
1: Timing for checking extensions of modules.		
2: Statistics about the modules to be checked.		
Verbose	MSQ_RepsCalc	<i>Maximum : 3</i>
1: Timing information about the module calculation.		
2: Statistics about the module calculation.		
Verbose	MSQ_Collector	<i>Maximum : 1</i>
If set to 1, the timing for the setup of the symbolic collector is printed.		
Verbose	MSQ_TraceFunc	<i>Maximum : 2</i>
Give messages about the main function calls (1) resp. most function calls (2) in the MAGMA language during the algorithm.		

## 71.6 Bibliography

- [**CDHW73**] John J. Cannon, Lucien A. Dimino, George Havas, and Jane M. Watson. Implementation and analysis of the Todd-Coxeter algorithm. *Math. Comp.*, 27:463–490, 1973.
- [**Hav91**] G. Havas. Coset enumeration strategies. In *ISSAC'91*, pages 191–199. ACM Press, 1991.
- [**NO96**] M. F. Newman and E. A. O'Brien. Application of computers to questions like those of Burnside. II. *Internat. J. Algebra Comput.*, 6(5):593–605, 1996.
- [**Ram**] Colin Ramsay. ACE. URL:<http://www.csee.uq.edu.au/~cram/>.



# 72 POLYCYCLIC GROUPS

<p><b>72.1 Introduction</b> . . . . . <b>2249</b></p> <p><b>72.2 Polycyclic Groups and Polycyclic Presentations</b> . . . . . <b>2249</b></p> <p>72.2.1 Introduction . . . . . 2249</p> <p>72.2.2 Specification of Elements . . . . . 2250</p> <p>!</p> <p>Identity(G) 2250</p> <p>Id(G) 2250</p> <p>!</p> <p>72.2.3 Access Functions for Elements . . . 2250</p> <p>ElementToSequence(x) 2250</p> <p>Eltseq(x) 2250</p> <p>LeadingTerm(x) 2251</p> <p>LeadingGenerator(x) 2251</p> <p>LeadingExponent(x) 2251</p> <p>Depth(x) 2251</p> <p>72.2.4 Arithmetic Operations on Elements 2251</p> <p>*</p> <p>*:= 2251</p> <p>^</p> <p>^:= 2251</p> <p>/</p> <p>/:= 2252</p> <p>^</p> <p>^:= 2252</p> <p>(g<sub>1</sub>, . . . , g<sub>n</sub>) 2252</p> <p>72.2.5 Operators for Elements . . . . . 2252</p> <p>Order(x) 2252</p> <p>Parent(x) 2252</p> <p>72.2.6 Comparison Operators for Elements 2252</p> <p>eq 2252</p> <p>ne 2252</p> <p>IsIdentity(g) 2253</p> <p>IsId(g) 2253</p> <p>72.2.7 Specification of a Polycyclic Presentation . . . . . 2253</p> <p>quo&lt; &gt; 2253</p> <p>PolycyclicGroup&lt; &gt; 2254</p> <p>72.2.8 Properties of a Polycyclic Presentation . . . . . 2257</p> <p>IsConsistent(G) 2257</p> <p>IsIdenticalPresentation(G, H) 2257</p> <p>PresentationIsSmall(G) 2257</p> <p><b>72.3 Subgroups, Quotient Groups, Homomorphisms and Extensions</b> <b>2257</b></p> <p>72.3.1 Construction of Subgroups . . . . . 2257</p> <p>sub&lt; &gt; 2257</p> <p>ncl&lt; &gt; 2257</p>	<p>72.3.2 Coercions Between Groups and Subgroups . . . . . 2258</p> <p>!</p> <p>!</p> <p>!</p> <p>InclusionMap(G, H) 2258</p> <p>72.3.3 Construction of Quotient Groups . 2259</p> <p>quo&lt; &gt; 2259</p> <p>/ 2259</p> <p>72.3.4 Homomorphisms . . . . . 2259</p> <p>hom&lt; &gt; 2260</p> <p>72.3.5 Construction of Extensions . . . . 2260</p> <p>DirectProduct(G, H) 2260</p> <p>72.3.6 Construction of Standard Groups . 2260</p> <p>AbelianGroup(GrpGPC, Q) 2260</p> <p>CyclicGroup(GrpGPC, n) 2261</p> <p>DihedralGroup(GrpGPC, n) 2261</p> <p>ElementaryAbelianGroup(GrpGPC, p, n) 2261</p> <p>ExtraSpecialGroup(GrpGPC, p, n : -) 2261</p> <p>FreeAbelianGroup(GrpGPC, n) 2261</p> <p>FreeNilpotentGroup(r, e) 2261</p> <p><b>72.4 Conversion between Categories</b> <b>2263</b></p> <p>AbelianGroup(G) 2263</p> <p>FPGroup(G) 2263</p> <p>PCGroup(G) 2263</p> <p>GPCGroup(G) 2263</p> <p><b>72.5 Access Functions for Groups</b> . <b>2264</b></p> <p>.</p> <p>Generators(G) 2264</p> <p>PCGenerators(G) 2264</p> <p>Generators(H, G) 2264</p> <p>PCGenerators(H, G) 2264</p> <p>NumberOfGenerators(G) 2264</p> <p>Ngens(G) 2264</p> <p>NumberOfPCGenerators(G) 2264</p> <p>NPCgens(G) 2264</p> <p>PCExponents(G) 2265</p> <p>HirschNumber(G) 2265</p> <p><b>72.6 Set-Theoretic Operations in a Group</b> . . . . . <b>2265</b></p> <p>72.6.1 Functions Relating to Group Order 2265</p> <p>FactoredIndex(G, H) 2265</p> <p>FactoredOrder(G) 2265</p> <p>Index(G, H) 2265</p> <p>Order(G) 2265</p> <p># 2265</p> <p>72.6.2 Membership and Equality . . . . . 2265</p> <p>in 2265</p> <p>notin 2265</p>
--	--

subset	2265	72.9.2 Properties of Subgroups Requiring a Nilpotent Covering Group . . . . .	2272
notsubset	2266	IsConjugate(G, H, K)	2272
subset	2266	IsSelfNormalising(G, H)	2272
notsubset	2266	IsSelfNormalizing(G, H)	2272
eq	2266	<b>72.10 Normal Structure and Characteristic Subgroups . . . . .</b>	<b>2274</b>
ne	2266	72.10.1 Characteristic Subgroups and Subgroup Series . . . . .	2274
72.6.3 Set Operations . . . . .	2266	Centre(G)	2274
Representative(G)	2266	Center(G)	2274
Rep(G)	2266	DerivedLength(G)	2274
RandomProcess(G)	2266	DerivedSeries(G)	2275
Random(P)	2267	DerivedSubgroup(G)	2275
Random(G)	2267	DerivedGroup(G)	2275
Random(G, max)	2267	EFASeries(G)	2275
<b>72.7 Coset Spaces . . . . .</b>	<b>2267</b>	FittingLength(G)	2275
CosetTable(G, H)	2267	FittingSeries(G)	2275
Transversal(G, H)	2267	FittingSubgroup(G)	2275
RightTransversal(G, H)	2267	FittingGroup(G)	2275
CosetAction(G, H)	2268	HasComputableLCS(G)	2275
CosetImage(G, H)	2269	LowerCentralSeries(G)	2275
CosetKernel(G, H)	2269	NilpotencyClass(G)	2275
<b>72.8 The Subgroup Structure . . .</b>	<b>2270</b>	NilpotentPresentation(G)	2276
72.8.1 General Subgroup Constructions . . . . .	2270	SemisimpleEFASeries(G)	2276
~	2270	UpperCentralSeries(G)	2276
Conjugate(H, g)	2270	72.10.2 The Abelian Quotient Structure of a Group . . . . .	2278
~	2270	AbelianQuotient(G)	2278
ncl< >	2270	AbelianQuotientInvariants(G)	2278
NormalClosure(G, H)	2270	AQInvariants(G)	2278
CommutatorSubgroup(G, H, K)	2270	ElementaryAbelianQuotient(G, p)	2278
CommutatorSubgroup(H, K)	2270	FreeAbelianQuotient(G)	2278
72.8.2 Subgroup Constructions Requiring a Nilpotent Covering Group . . . . .	2270	<b>72.11 Conjugacy . . . . .</b>	<b>2278</b>
meet	2270	IsConjugate(G, g, h)	2278
meet:=	2270	IsConjugate(G, H, K)	2278
Centraliser(G, g)	2270	<b>72.12 Representation Theory . . .</b>	<b>2279</b>
Centralizer(G, g)	2270	EFAModuleMaps(G)	2279
Centraliser(G, H)	2271	EFAModules(G)	2280
Centralizer(G, H)	2271	GModule(G, A, p)	2280
Core(G, H)	2271	GModule(G, A)	2280
Normaliser(G, H)	2271	GModule(G, A, B, p)	2280
Normalizer(G, H)	2271	GModule(G, A, B)	2280
<b>72.9 General Group Properties . .</b>	<b>2271</b>	GModulePrimes(G, A)	2281
IsAbelian(G)	2271	GModulePrimes(G, A, B)	2281
IsCyclic(G)	2271	SemisimpleEFAModuleMaps(G)	2281
IsElementaryAbelian(G)	2271	SemisimpleEFAModules(G)	2281
IsFinite(G)	2271	<b>72.13 Power Groups . . . . .</b>	<b>2285</b>
IsNilpotent(G)	2271	Parent(G)	2285
IsPerfect(G)	2271	PowerGroup(G)	2285
IsSimple(G)	2271	<b>72.14 Bibliography . . . . .</b>	<b>2286</b>
IsSoluble(G)	2272		
IsSolvable(G)	2272		
72.9.1 General Properties of Subgroups . . . . .	2272		
IsCentral(G, H)	2272		
IsNormal(G, H)	2272		

# Chapter 72

## POLYCYCLIC GROUPS

### 72.1 Introduction

In this chapter, we consider a class of finitely presented groups for which the word problem is solvable, the category of – possibly infinite – polycyclic groups. The corresponding MAGMA category is called `GrpGPC`. To distinguish this class from finite solvable groups described by a power-conjugate presentation (MAGMA category `GrpPC`, cf. Chapter 63), we use the term *general polycyclic group*.

An introduction to the theory of polycyclic groups and a collection of some basic algorithms can be found in [Sim94, ch. 9]. Unless otherwise mentioned, implementations of MAGMA functions are mostly based on ideas described in this reference.

### 72.2 Polycyclic Groups and Polycyclic Presentations

#### 72.2.1 Introduction

A polycyclic group is a group  $G$  with a subnormal series  $G = G_1 \triangleright G_2 \triangleright \dots \triangleright G_{n+1} = 1$  in which each of the quotients  $G_i/G_{i+1}$  is cyclic. Every polycyclic group  $G$  has a presentation of the form

$$\begin{aligned} \langle a_1, \dots, a_n \mid & a_i^{m_i} = w_{i,i} \quad (i \in I), \\ & a_j^{a_i} = w_{i,j} \quad (1 \leq i < j \leq n), \\ & a_j^{a_i^{-1}} = w_{-i,j} \quad (1 \leq i < j \leq n, i \notin I) \rangle \end{aligned}$$

where

- (i)  $I \subseteq \{1, \dots, n\}$ ,
- (ii)  $m_i > 1$  for  $i \in I$ , and
- (iii) the words  $w_{i,j}$  are of the form  $w_{i,j} = a_{|i|+1}^{l(i,j,|i|+1)} \dots a_n^{l(i,j,n)}$ , with  $0 \leq l(i,j,k) < m_k$  if  $k \in I$ .

Such a presentation is called a *polycyclic presentation* for  $G$ . For  $1 \leq i \leq n$ , let  $G_i$  be the subgroup of  $G$  generated by  $a_i, \dots, a_n$  and define  $G_{n+1}$  to be the trivial group. The presentation is called *consistent*, if  $|G_i/G_{i+1}| = m_i$  whenever  $i \in I$  and  $G_i/G_{i+1}$  is infinite whenever  $i \notin I$ . The generators  $a_1, \dots, a_n$  are referred to as *polycyclic generators* (*pc-generators*) for  $G$  and the values  $m_i$  ( $i \in I$ ) are called the corresponding *pc-exponents*.

In MAGMA, the user can define a polycyclic group by providing a consistent polycyclic presentation or by using one of the existing category transfer functions.

Given a consistent polycyclic presentation for  $G$ , every element  $a$  of  $G$  can be written uniquely in the form  $a = a_1^{e_1} \dots a_n^{e_n}$ , where the  $e_i$  are integers satisfying  $0 \leq e_i < m_i$

if  $i \in I$ . This form is called *normal form*. There exists an algorithm (the *collection algorithm*), which given an arbitrary word in the generators  $a_1, \dots, a_n$ , will determine the corresponding normal word. MAGMA uses a collection algorithm written by Volker Gebhardt. The cost of collection for this algorithm grows logarithmically in a bound on the absolute values of the exponents  $e_i$  occurring during the collection [Geb02].

Infinite polycyclic groups are a comparatively new topic in computational group theory and the number of available algorithms is much smaller than in the case of finite polycyclic groups. For this reason, the data type GrpPC (cf. Chapter 63) should be preferred for finite polycyclic groups.

### 72.2.2 Specification of Elements

Elements of polycyclic groups are words. A *word* is defined inductively as follows:

- (i) A generator is a word;
- (ii) The expression  $(u)$  is a word, where  $u$  is a word;
- (iii) The product  $u * v$  of the words  $u$  and  $v$  is a word;
- (iv) The conjugate  $u^v$  of the word  $u$  by the word  $v$  is a word ( $u^v$  expands into the word  $v^{-1} * u * v$ );
- (v) The power of a word  $u^n$ , where  $u$  is a word and  $n$  is an integer, is a word;
- (vi) The commutator  $(u, v)$  of the words  $u$  and  $v$  is a word ( $(u, v)$  expands into the word  $u^{-1} * v^{-1} * u * v$ ).

<b>G ! Q</b>
--------------

Given the polycyclic group  $G$  and a sequence  $Q$  of length  $n$ , containing integers  $e_1 \dots e_n$ , where  $0 \leq e_i < m_i$  if  $i \in I$ , construct the element  $x$  of  $G$  given by

$$x = a_1^{e_1} \dots a_n^{e_n}.$$

<b>Identity(G)</b>
--------------------

<b>Id(G)</b>
--------------

<b>G ! 1</b>
--------------

Construct the identity element of the polycyclic group  $G$ .

### 72.2.3 Access Functions for Elements

Throughout this subsection,  $G$  will be a polycyclic group with pc-generators  $a_1, \dots, a_n$ .

<b>ElementToSequence(x)</b>
-----------------------------

<b>Eltseq(x)</b>
------------------

Given an element  $x$  belonging to the polycyclic group  $G$ , where  $x = a_1^{e_1} \dots a_n^{e_n}$  in normal form, return the sequence  $Q$  of  $n$  integers defined by  $Q[i] = e_i$ , for  $i = 1, \dots, n$ .

**LeadingTerm(x)**

Given an element  $x$  of a polycyclic group  $G$  with  $n$  pc-generators, where  $x$  is of the form  $a_1^{e_1} \dots a_n^{e_n}$ , return  $a_i^{e_i}$  for the smallest  $i$  such that  $e_i > 0$ . If  $x$  is the identity of  $G$ , then the identity is returned.

**LeadingGenerator(x)**

Given an element  $x$  of a polycyclic group  $G$  with  $n$  pc-generators, where  $x$  is of the form  $a_1^{e_1} \dots a_n^{e_n}$ , return  $a_i$  for the smallest  $i$  such that  $e_i > 0$ . If  $x$  is the identity of  $G$ , then the identity is returned.

**LeadingExponent(x)**

Given an element  $x$  of a polycyclic group  $G$  with  $n$  pc-generators, where  $x$  is of the form  $a_1^{e_1} \dots a_n^{e_n}$ , return  $e_i$  for the smallest  $i$  such that  $e_i > 0$ . If  $x$  is the identity of  $G$ , then 0 is returned.

**Depth(x)**

Given an element  $x$  of a polycyclic group  $G$  with  $n$  pc-generators, where  $x$  is of the form  $a_1^{e_1} \dots a_n^{e_n}$ , return the smallest  $i$  such that  $e_i > 0$ . If  $x$  is the identity of  $G$ , then  $n + 1$  is returned.

**Depth** returns the maximal value of  $i$ , such that  $x \in G_i$ .

**72.2.4 Arithmetic Operations on Elements**

Throughout this subsection,  $G$  will be a polycyclic group with pc-generators  $a_1, \dots, a_n$ .

**g \* h**

Product of the element  $g$  and the element  $h$ , where  $g$  and  $h$  belong to some common subgroup  $G$  of a polycyclic group  $U$ . If  $g$  and  $h$  are given as elements belonging to the same proper subgroup  $G$  of  $U$ , then the result will be returned as an element of  $G$ ; if  $g$  and  $h$  are given as elements belonging to distinct subgroups  $H$  and  $K$  of  $U$ , then the product is returned as an element of  $G$ , where  $G$  is the smallest subgroup of  $U$  known to contain both elements.

**g := h**

Replace  $g$  with the product of element  $g$  and element  $h$ .

**g ^ n**

The  $n$ -th power of the element  $g$ , where  $n$  is a positive or negative integer.

**g ^:= n**

Replace  $g$  with the  $n$ -th power of the element  $g$ .

$g / h$ 

Quotient of the element  $g$  by the element  $h$ , i.e. the element  $g * h^{-1}$ . Here  $g$  and  $h$  must belong to some common subgroup  $G$  of a polycyclic group  $U$ . The rules for determining the parent group of  $g/h$  are the same as for  $g * h$ .

 $g /:= h$ 

Replace  $g$  with  $g * h^{-1}$ .

 $g \hat{=} h$ 

Conjugate of the element  $g$  by the element  $h$ , i.e. the element  $h^{-1} * g * h$ . Here  $g$  and  $h$  must belong to some common subgroup  $G$  of a polycyclic group  $U$ . The rules for determining the parent group of  $g^h$  are the same as for  $g * h$ .

 $g \hat{:=} h$ 

Replace  $g$  with the conjugate of the element  $g$  by the element  $h$ .

 $(g_1, \dots, g_n)$ 

Given the  $n$  words  $g_1, \dots, g_n$  belonging to some common subgroup  $G$  of a polycyclic group  $U$ , compute the *left-normed* commutator of  $g_1, \dots, g_n$ . This is defined inductively as follows:  $(g_1, g_2) = g_1^{-1} * g_2^{-1} * g_1 * g_2$  and  $(g_1, \dots, g_n) = ((g_1, \dots, g_{n-1}), g_n)$ .

If  $g_1, \dots, g_n$  are given as elements belonging to the same proper subgroup  $G$  of  $U$ , then the result will be returned as an element of  $G$ ; if  $g_1, \dots, g_n$  are given as elements belonging to distinct subgroups of  $U$ , then the product is returned as an element of  $G$ , where  $G$  is the smallest subgroup of  $U$  known to contain all elements.

### 72.2.5 Operators for Elements

 $\text{Order}(x)$ 

The order of the element  $x$ .

 $\text{Parent}(x)$ 

The parent group  $G$  of the element  $x$ .

### 72.2.6 Comparison Operators for Elements

 $g \text{ eq } h$ 

Given elements  $g$  and  $h$  belonging to a common polycyclic group, return **true** if  $g$  and  $h$  are the same element, **false** otherwise.

 $g \text{ ne } h$ 

Given elements  $g$  and  $h$  belonging to a common polycyclic group, return **true** if  $g$  and  $h$  are distinct elements, **false** otherwise.



with disabled consistency check. It should be noted that the results of working with an inconsistent presentation are undefined, hence it is strongly advised to enable the consistency check in the constructor or to use the function `IsConsistent`.

The natural homomorphism from  $F \rightarrow G$  is returned as second return value.

<code>PolycyclicGroup&lt; x<sub>1</sub>, ..., x<sub>n</sub>   R : parameters &gt;</code>
--

<b>Check</b>	BOOLELT	<i>Default : true</i>
<b>Class</b>	MONSTGELT	<i>Default :</i>

Construct the polycyclic group  $G$  defined by the consistent polycyclic presentation  $\langle x_1, \dots, x_n | R \rangle$ .

The construct  $x_1, \dots, x_n$  defines names for the generators of  $G$  that are local to the constructor, i.e. they are used when writing down the relations to the right of the bar. However, no assignment of names to these generators is made. If the user wants to refer to the generators by these (or other) names, then the *generators assignment* construct must be used on the left hand side of an assignment statement.

The construct  $R$  denotes a list of polycyclic relations. The syntax and semantics for the relations clause is identical to that appearing in the `quo`-constructor above.

A map  $f$  from the free group on  $x_1, \dots, x_n$  to  $G$  is returned as second return value.

The parameter `Check` may be used as described for the `quo`-constructor.

If  $R$  is both, a valid power-conjugate presentation for a finite soluble group (cf. Chapter 63) and a consistent polycyclic presentation, this constructor by default returns a group in the category `GrpPC`. To force creation of a group in the category `GrpGPC`, the parameter `Class` must be set to `"GrpGPC"` in these situations.

### Example H72E1

---

(1) Consider the infinite polycyclic group defined by the presentation

$$\langle a, b, c \mid b^a = b * c, (a, c), (b, c) \rangle .$$

Starting from a free group and giving the relations in the form of a sequence, this presentation would be specified as follows:

```
> F<a,b,c> := FreeGroup(3);
> rels := [ b^a = b*c, b^(a^-1) = b*c^-1 ];
> G<a,b,c> := quo< GrpGPC : F | rels >;
> G;
GrpGPC : G of infinite order on 3 PC-generators
PC-Relations:
  b^a = b * c,
  b^(a^-1) = b * c^-1
```

Note, that the relation  $b^{a^{-1}} = b * c^{-1}$  has to be included, although it can be derived from the relations  $b^a = b * c$  and  $(a, b)$ .

(2) The infinite dihedral group is obtained as epimorphic image of the free group of rank two as follows:

```
> F<a,b> := FreeGroup(2);
> D<u,v>, pi := quo<GrpGPC: F | a^2, b^a = b^-1>;
> D;
GrpGPC : D of infinite order on 2 PC-generators
PC-Relations:
    u^2 = Id(D),
    v^u = v^-1
> pi;
Mapping from: GrpFP: F to GrpGPC: D
```

(3) We create an element  $e$  of the group  $D$  defined above from a sequence of coefficients and extract both its leading generator and its leading exponent.

```
> e := D ! [1,42];
> e;
u * v^42
> gen := LeadingGenerator(e);
> gen;
u
> Parent(gen);
GrpGPC : D of infinite order on 2 PC-generators
PC-Relations:
    u^2 = Id(D),
    v^u = v^-1
> exp := LeadingExponent(e);
> exp;
1
```

We obtain an element of depth 2 from  $e$ , by replacing  $e$  with its quotient by the appropriate power of the leading generator.

```
> e /= gen^exp;
> Depth(e);
2
> e;
v^-42
> ElementToSequence(e);
[ 0, -42 ]
```

### Example H72E2

---

Using the constructor `PolycyclicGroup` with different values of the parameter `Class`, we construct the dihedral group of order 10 first as a finite soluble group given by a power-conjugate presentation (`GrpPC`) and next as a general polycyclic group (`GrpGPC`). Note that the presentation  $\langle a, b \mid a^2, b^5, b^a = b^4 \rangle$  is both a valid power-conjugate presentation and a consistent polycyclic

presentation, so we have to set the parameter `Class` to `"GrpGPC"` if we want to construct a group in the category `GrpGPC`.

```
> G1<a,b> := PolycyclicGroup< a,b | a^2, b^5, b^a=b^4 >;
> G1;
GrpPC : G1 of order 10 = 2 * 5
PC-Relations:
  a^2 = Id(G1),
  b^5 = Id(G1),
  b^a = b^4
> G2<a,b> := PolycyclicGroup< a,b | a^2, b^5, b^a=b^4 : Class := "GrpGPC">;
> G2;
GrpGPC : G2 of order 10 = 2 * 5 on 2 PC-generators
PC-Relations:
  a^2 = Id(G2),
  b^5 = Id(G2),
  b^a = b^4
```

We construct the infinite dihedral group as a group in the category `GrpGPC` from a consistent polycyclic presentation. We do not have to use the parameter `Class` in this case.

```
> G3<a,b> := PolycyclicGroup< a,b | a^2, b^a=b^-1>;
> G3;
GrpGPC : G3 of infinite order on 2 PC-generators
PC-Relations:
  a^2 = Id(G3),
  b^a = b^-1
```

The presentation  $\langle a, b \mid a^2, b^4, b^a = b^3 \rangle$  is not a valid power-conjugate presentation for the dihedral group of order 8, since the exponent of  $b$  is not prime. However, it is a consistent polycyclic presentation. Consequently, the constructor `PolycyclicGroup` without specifying a value for the parameter `Class` returns a group in the category `GrpGPC`.

```
> G4<a,b> := PolycyclicGroup< a,b | a^2, b^4, b^a=b^3 >;
> G4;
GrpGPC : G4 of order 2^3 on 2 PC-generators
PC-Relations:
  a^2 = Id(G3),
  b^4 = Id(G3),
  b^a = b^3
```

---

## 72.2.8 Properties of a Polycyclic Presentation

`IsConsistent(G)`

Returns **true** if the stored presentation for  $G$  is consistent, **false** otherwise.

`IsIdenticalPresentation(G, H)`

Returns **true** if the polycyclic presentations for  $G$  and  $H$  are identical, **false** otherwise.

`PresentationIsSmall(G)`

Returns **true** if only small integers occur in the presentation of  $G$ , **false** otherwise. The category transfer functions `FPGroup` and `PCGroup` currently support only groups with a small presentation.

## 72.3 Subgroups, Quotient Groups, Homomorphisms and Extensions

### 72.3.1 Construction of Subgroups

`sub< G | L >`

Construct the subgroup  $H$  of the polycyclic group  $G$  generated by the elements specified by the terms of the *generator list*  $L$ .

A term  $L[i]$  of the generator list may consist of any of the following objects:

- (a) An element liftable to  $G$ ;
- (b) A sequence of integers representing an element of  $G$ ;
- (c) A subgroup of  $G$ ;
- (d) A set or sequence of (a), (b), or (c). The collection of words and groups specified by the list must all belong to the group  $G$  and  $H$  will be constructed as a subgroup of  $G$ .

The generators of  $H$  consist of the words specified directly by terms of  $L[i]$  together with the stored generating words for any groups specified by terms of  $L[i]$ . Repetitions of an element and occurrences of the identity element are removed.

The inclusion map from  $H$  to  $G$  is returned as a second value.

`ncl< G | L >`

Construct the subgroup  $N$  of the polycyclic group  $G$  as the normal closure of the subgroup generated by the elements specified by the terms of the *generator list*  $L$ .

The possible forms of a term  $L[i]$  of the generator list are the same as for the `sub`-constructor.

The inclusion map from  $N$  to  $G$  is returned as a second value.

### 72.3.2 Coercions Between Groups and Subgroups

**G ! g**

Given an element  $g$  belonging to the subgroup  $H$  of the group  $G$ , rewrite  $g$  as an element of  $G$ .

**H ! g**

Given an element  $g$  belonging to the group  $G$ , and given a subgroup  $H$  of  $G$  containing  $g$ , rewrite  $g$  as an element of  $H$ .

**K ! g**

Given an element  $g$  belonging to the group  $H$ , and a group  $K$ , such that  $H$  and  $K$  are subgroups of  $G$ , and both  $H$  and  $K$  contain  $g$ , rewrite  $g$  as an element of  $K$ .

**InclusionMap(G, H)**

The map from the subgroup  $H$  of  $G$  to  $G$ .

#### Example H72E3

---

Consider again the infinite polycyclic group  $G$  defined by the polycyclic presentation

$$\langle a, b, c \mid b^a = b * c, (a, c), (b, c) \rangle .$$

```
> F<a,b,c> := FreeGroup(3);
> rels := [ b^a = b*c, b^(a^-1) = b*c^-1 ];
> G<a,b,c> := quo< GrpGPC : F | rels >;
> G;
GrpGPC : G of infinite order on 3 PC-generators
PC-Relations:
  b^a = b * c,
  b^(a^-1) = b * c^-1
```

Using the function `PCGenerators` which is described later, the groups  $G_1, \dots, G_4$  and the corresponding inclusion maps can be defined as follows:

```
> G_ := []; incl_ := [ PowerStructure(Map) | ];
> for i := 1 to #PCGenerators(G)+1 do
>   G_[i], incl_[i] := sub< G | [ g : g in PCGenerators(G) |
>                               Depth(g) ge i ] >;
> end for;
> for i := 1 to #G_ do
>   printf "G_%o = <%o>", i, {@ G!x : x in
>                               PCGenerators(G_[i]) @}; print "";
> end for;
G_1 = <{@ a, b, c @}>
G_2 = <{@ b, c @}>
G_3 = <{@ c @}>
```

`G_4 = <{@ @}>`

Note that we must set the universe of the sequence `incl_` to `PowerStructure(Map)` manually, since we want to store maps which do not have a common domain. If we failed to do this, the universe would be chosen automatically to be the set of all maps from  $G_1$  to  $G$  when the first map is inserted into the sequence. Inserting the second map, which does not have the domain  $G_1$ , would then cause a runtime error.

---

### 72.3.3 Construction of Quotient Groups

`quo< G | L >`

Construct the quotient  $Q$  of the polycyclic group  $G$  by the normal subgroup  $N$ , where  $N$  is the smallest normal subgroup of  $G$  containing the elements specified by the terms of the *generator list*  $L$ .

The possible forms of a term  $L[i]$  of the generator list are the same as for the `sub`-constructor.

The quotient group  $Q$  and the corresponding natural homomorphism  $f : G \rightarrow Q$  are returned.

`G / N`

Given a normal subgroup  $N$  of the polycyclic group  $G$ , construct the quotient of  $G$  by  $N$ .

### 72.3.4 Homomorphisms

For a general description of homomorphisms, we refer to Chapter 16. This section describes some special aspects of homomorphisms the domain of which is a polycyclic group.

#### 72.3.4.1 General remarks

The kernel of a homomorphism with a domain of type `GrpGPC` can be computed using the function `Kernel`, if the codomain is of one of the types `GrpGPC`, `GrpPC` (cf. Chapter 63), `GrpAb` (cf. Chapter 69), `GrpPerm` (cf. Chapter 58), `ModAlg` or `ModGrp` (cf. Chapter 89) or if the codomain is of the type `GrpMat` (cf. Chapter 59) and the image is finite.

In particular, preimages of substructures can be computed in these situations. The kernel of a map will be computed automatically, if the preimage of a substructure of the codomain is to be computed.

The kernel (and hence preimages of substructures) may also be computable, if the codomain is of the type `GrpFP` (cf. Chapter 70) and the domain is nilpotent.

### 72.3.4.2 Construction of Homomorphisms

`hom< P -> G | S : parameters >`

Returns the homomorphism from the polycyclic group  $P$  to the group  $G$  defined by the assignment  $S$ .  $S$  can be the one of the following:

- (i) A list, sequence or indexed set containing the images of the  $n$  polycyclic generators  $P.1, \dots, P.n$  of  $P$ . Here, the  $i$ -th element of  $S$  is interpreted as the image of  $P.i$ , i.e. the order of the elements in  $S$  is important.
- (ii) A list, sequence, enumerated set or indexed set, containing  $r$  tuples  $\langle x_i, y_i \rangle$  or arrow pairs  $x_i \rightarrow y_i$ , where  $x_i \in P$ ,  $y_i \in G$  ( $i = 1, \dots, r$ ) and the set  $\{x_1, \dots, x_r\}$  is a generating set for  $P$ . In this case,  $y_i$  is assigned as the image of  $x_i$ , hence the order of the elements in  $S$  is not important. Note that the preimages  $x_i$  need not be the polycyclic generators of  $P$ .

If the data type of the codomain supports element arithmetic and element comparison, by default the constructed homomorphism is checked by verifying that the would-be images of the polycyclic generators satisfy the defining relations of  $P$  and that this assignment is consistent with the assignments made by the user. In this case, it is assured that the returned map is a well-defined homomorphism with the desired images. The most important situation in which it is not possible to perform checking is the case in which the domain is a finitely presented group (`FPGroup`; cf. Chapter 70) which is not free. Checking may be disabled using the parameter `Check`.

`Check`

`BOOLELT`

*Default : true*

If `Check` is set to `false`, checking of the homomorphism is disabled.

### 72.3.5 Construction of Extensions

`DirectProduct(G, H)`

The direct product  $K$  of the polycyclic groups  $G$  and  $H$ . The second value returned is a sequence containing the inclusion maps  $I_G : G \rightarrow K$  and  $I_H : H \rightarrow K$ . The third value returned is a sequence containing the projection maps  $P_G : K \rightarrow G$  and  $P_H : K \rightarrow H$ .

### 72.3.6 Construction of Standard Groups

A number of functions are provided which construct polycyclic presentations for various standard groups.

`AbelianGroup(GrpGPC, Q)`

Construct the abelian group defined by the sequence  $Q = [n_1, \dots, n_r]$  as a polycyclic group. The entries  $n_i$  may be either zero, indicating an infinite cyclic factor, or integers greater than 1. The function returns the polycyclic group which is the direct product of the cyclic groups  $Z_1 \times \dots \times Z_r$ , where  $Z_i$  is a cyclic group of infinite order if  $n_i = 0$ , or a cyclic group of order  $n_i$  if  $n_i > 1$ .



```
> f := hom< G->A | a->u, a*b->v >;
```

We compute the kernel  $K$  of  $f$  and express the generators of  $K$  as elements of  $G$ , using the function `PCGenerators` described later.

```
> K := Kernel(f);
> PCGenerators(K, G);
{@ b^2 @}
```

---

### Example H72E5

A polycyclic representation for the group  $D_3 \times D_\infty$  may be obtained as follows:

```
> G1<a,b> := DihedralGroup(GrpGPC, 3);
> G2<u,v> := DihedralGroup(GrpGPC, 0);
> D, incl, proj := DirectProduct(G1, G2);
> D;
GrpGPC : D of infinite order on 4 PC-generators
PC-Relations:
  D.1^2 = Id(D),
  D.2^3 = Id(D),
  D.3^2 = Id(D),
  D.2^D.1 = D.2^2,
  D.4^D.3 = D.-4
```

Using the inclusion maps returned by `DirectProduct`, we define a subgroup and compute the quotient by its normal closure:

```
> S := sub<D| incl[1](a)*incl[2](u), incl[1](b)*incl[2](v)>;
> S;
GrpGPC : S of infinite order on 3 PC-generators
PC-Relations:
  S.1^2 = Id(S),
  S.2^3 = S.3,
  S.2^S.1 = S.2^2 * S.-3,
  S.3^S.1 = S.-3
> Q, pi := quo<D|S>;
> Q;
GrpGPC : Q of order 2 on 1 PC-generators
PC-Relations:
  Q.1^2 = Id(Q)
> Q.1 @@ pi;
D.3
```

---

## 72.4 Conversion between Categories

This section describes category transfers from polycyclic groups to other MAGMA categories of groups. For category transfers to permutation groups (cf. Chapter 58) the functions `CosetAction` and `CosetImage` can be used. For category transfers to matrix groups (cf. Chapter 59) we refer to section 72.12.

### AbelianGroup( $G$ )

A `GrpAb` (cf. Chapter 69) representation  $A$  of the abelian polycyclic group  $G$  and the isomorphism from  $G$  to  $A$ .

### FPGroup( $G$ )

A `GrpFP` (cf. Chapter 70) representation  $F$  of  $G$  and the isomorphism from  $F$  to  $G$ . This category transfer is currently only possible provided that all exponents occurring in the presentation of  $G$  are small integers. This can be checked by means of the function `PresentationIsSmall`.

### PCGroup( $G$ )

A `GrpPC` (cf. Chapter 63) representation  $F$  of the finite polycyclic group  $G$  and the isomorphism from  $G$  to  $F$ .

This category transfer is currently only possible provided that all exponents occurring in the presentation of  $G$  are small integers. This can be checked by means of the function `PresentationIsSmall`.

### GPCGroup( $G$ )

A `GrpGPC` representation  $P$  of the solvable group  $G$  and the isomorphism from  $G$  to  $P$ . The group  $G$  can be of one of the following types: `GrpPerm` (cf. Chapter 58), `GrpMat` (cf. Chapter 59), `GrpAb` (cf. Chapter 69), `GrpPC` (cf. Chapter 63). Currently  $G$  must be finite, if it is of type `GrpMat`.

### Example H72E6

---

We define a finite, solvable matrix group and convert it to a (general) polycyclic group  $G$ .

```
> a := GL(2,3) ! [1,1,0,1];
> b := GL(2,3) ! [0,1,1,0];
> M := sub<Parent(a)|a,b>;
> IsSolvable(M);
true
> IsFinite(M);
true
> G, f := GPCGroup(M);
```

We now compute the direct product  $D$  of  $G$  and the infinite dihedral group  $H$ , define a subgroup  $S$  of  $D$ , its normal closure  $N$  and construct the quotient  $Q$  of  $D$  by  $N$ .

```
> H<u,v> := DihedralGroup(GrpGPC, 0);
> D, incl, proj := DirectProduct(G, H);
```

```

> S := sub<D | incl[1](f(a*b)), incl[2]((u,v)^2)>;
> N := ncl<D|S>;
> Q := D/N;
> Q;
GrpGPC : Q of order 2^3 on 2 PC-generators
PC-Relations:
  Q.1^2 = Id(Q),
  Q.2^4 = Id(Q),
  Q.2^Q.1 = Q.2^3

```

Since  $Q$  is finite, it can be transformed into a group of type `GrpPC` using the function `PCGroup`. This should (in non-trivial examples) be done, if further computations with it are intended.

```

> Q_ := PCGroup(Q);
> Q_;
GrpPC : Q_ of order 8 = 2^3
PC-Relations:
  Q_.2^2 = Q_.3,
  Q_.2^Q_.1 = Q_.2 * Q_.3

```

## 72.5 Access Functions for Groups

The functions described here provide access to basic information stored for a polycyclic group  $G$ .

`G . i`

The  $i$ -th polycyclic generator for  $G$  if  $i > 0$ , the inverse of the  $|i|$ -th polycyclic generator for  $G$  if  $i < 0$  and the identity element of  $G$  if  $i = 0$ .

`Generators(G)`

`PCGenerators(G)`

An indexed set containing the polycyclic generators of  $G$ .

`Generators(H, G)`

`PCGenerators(H, G)`

An indexed set containing the polycyclic generators of  $H$  as elements of  $G$ .

`NumberOfGenerators(G)`

`Ngens(G)`

`NumberOfPCGenerators(G)`

`NPCgens(G)`

The number of polycyclic generators for the polycyclic group  $G$ .

**PCExponents(G)**

The orders of the cyclic factors in the polycyclic series defined by the polycyclic presentation of  $G$ . The orders are returned in a sequence  $Q$ .  $|G_i/G_{i+1}| = m_i = Q[i]$  if  $Q[i] > 0$  and  $G_i/G_{i+1}$  is infinite (i.e.  $i \notin I$ ) if  $Q[i] = 0$ .

**HirschNumber(G)**

The Hirsch number of  $G$ , i.e. the number of infinite cyclic factors in the polycyclic series defined by the polycyclic presentation of  $G$ .

The Hirsch number of  $G$  is equal to  $n - |I|$ , i.e. to the number of polycyclic generators of  $G$  for which there is no power relation. A polycyclic group  $G$  is finite if and only if its Hirsch number is 0.

**72.6 Set-Theoretic Operations in a Group****72.6.1 Functions Relating to Group Order****FactoredIndex(G, H)**

Given a group  $G$  and a subgroup  $H$  of  $G$  of finite index, return the factored index of  $H$  in  $G$

**FactoredOrder(G)**

The factored order of the finite group  $G$ .

**Index(G, H)**

The index of the subgroup  $H$  in the group  $G$ , returned as an ordinary integer.

**Order(G)****#G**

The order of the group  $G$ , returned as an ordinary integer.

**72.6.2 Membership and Equality****g in G**

Given an element  $g$  and a group  $G$ , return **true** if  $g$  is an element of  $G$ , **false** otherwise.

**g notin G**

Given an element  $g$  and a group  $G$ , return **true** if  $g$  is not an element of  $G$ , **false** otherwise.

**S subset G**

Given an group  $G$  and a set  $S$  of elements belonging to a group  $H$ , where  $G$  and  $H$  have some covering group, return **true** if  $S$  is a subset of  $G$ , **false** otherwise.

**S notsubset G**

Given a group  $G$  and a set  $S$  of elements belonging to a group  $H$ , where  $G$  and  $H$  have some covering group, return **true** if  $S$  is not a subset of  $G$ , **false** otherwise.

**H subset G**

Given groups  $G$  and  $H$ , having some covering group, return **true** if  $H$  is a subgroup of  $G$ , **false** otherwise.

**H notsubset G**

Given groups  $G$  and  $H$ , having some covering group, return **true** if  $H$  is not a subgroup of  $G$ , **false** otherwise.

**G eq H**

Given groups  $G$  and  $H$ , having some covering group, return **true** if  $G$  and  $H$  are the same group, **false** otherwise.

**G ne H**

Given groups  $G$  and  $H$ , having some covering group, return **true** if  $G$  and  $H$  are distinct groups, **false** otherwise.

### 72.6.3 Set Operations

**Representative(G)**

**Rep(G)**

A representative element of  $G$ .

**RandomProcess(G)**

<b>Slots</b>	RNGINTELT	<i>Default : 10</i>
<b>Scramble</b>	RNGINTELT	<i>Default : 100</i>

Create a process to generate pseudo-randomly chosen elements from the group  $G$ . The process uses an ‘expansion’ procedure to construct a set of elements corresponding to fairly long words in the generators of  $G$  [CLGM<sup>+</sup>95]. At all times,  $N$  elements forming a generating set for  $G$  are stored. Here,  $N$  is the maximum of  $n + 1$  and the specified value for **Slots**, where  $n$  is the number of generators of  $G$ . Initially, these are just the generators of  $G$  and products of pairs of generators of  $G$ . Random elements are now produced by successive calls to **Random(P)**, where  $P$  is the process created by this function. Each such call chooses an element previously stored by the process as the new random element. The process then replaces this stored element with the product of this element and another one of the stored elements (on the left or the right). Setting **Scramble := m** causes  $m$  such operations to be performed before the process is returned.

Care should be taken when trying to apply this function to infinite polycyclic groups. Firstly, the computations may take a considerable amount of time and secondly, the quality of the pseudo-random element generator may be extremely poor, depending on the required properties of the sequence of pseudo-random elements.

**Random(P)**

Given a random element process  $P$  created by the function `RandomProcess(G)` for the group  $G$ , construct a pseudo-random element of  $G$  by forming a random product over the expanded generating set currently stored by the process. The remarks concerning random elements of infinite polycyclic groups given in the description of `RandomProcess` apply here.

**Random(G)****Random(G, max)**

An element, pseudo-randomly chosen, from the group  $G$ . An exponent vector in normal form is chosen at random. Exponents of polycyclic generators for which there is no power relation, are chosen to have absolute value less or equal to `max`. A default value for `max` is 10.

It should be kept in mind that the distribution of the elements returned by `Random` is uniform only in the case that  $G$  is finite.

## 72.7 Coset Spaces

**CosetTable(G, H)**

The (right) coset table for  $G$  over the subgroup  $H$  of finite index, relative to the polycyclic generators. This is defined to be a map

$$\{1, \dots, |G : H|\} \times G \rightarrow \{1, \dots, |G : H|\}$$

describing the action of  $G$  on the enumerated set of right cosets of  $H$  in  $G$  by right multiplication.

The underlying set of right coset representatives is identical to the right transversal returned by `Transversal` and `RightTransversal` and the same enumeration of the elements is used.

**Transversal(G, H)****RightTransversal(G, H)**

Given a group  $G$  and a subgroup  $H$  of  $G$ , this function returns:

- (a) An indexed set of elements  $T$  of  $G$  forming a right transversal for  $G$  over  $H$ . The right transversal and its enumeration are identical to those internally used by the function `CosetTable`.
- (b) The corresponding transversal mapping  $\phi : G \rightarrow T$ . If  $T = [t_1, \dots, t_r]$  and  $g$  in  $G$ ,  $\phi$  is defined by  $\phi(g) = t_i$ , where  $g \in H * t_i$ .

**Example H72E7**

---

We compute a right transversal of a subgroup  $H$  of the infinite dihedral group  $G$ .

```
> G<a,b> := DihedralGroup(GrpGPC, 0);
> H := sub<G|a*b, b^10>;
> Index(G, H);
10
> RT, transmap := Transversal(G, H);
> RT;
{@ Id(G), b^-1, b^-2, b^-3, b^-4, b^-5, b^-6, b^-7, b^-8, b^-9 @}
> transmap;
Mapping from: GrpGPC: G to SetIndx: RT
```

From this a left transversal is easily obtained:

```
> LT := {@ x^-1 : x in RT @};
> LT;
{@ Id(G), b, b^2, b^3, b^4, b^5, b^6, b^7, b^8, b^9 @}

```

We construct the coset table and define a function  $RT \times G \rightarrow RT$ , describing the action of  $G$  on the set of right cosets of  $H$  in  $G$ .

```
> ct := CosetTable(G, H);
> action := func< r, g | RT[ct(Index(RT, r), g)] >;
> action(Id(G), b);
b^-9
```

I.e.  $H * b = Hb^{-9}$ .

```
> action(b^-4, a*b);
b^-6
```

I.e.  $Hb^{-4} * (ab) = Hb^{-6}$ .

Note that the definition of the function `action` relies on the fact that the computed right transversal and its enumeration are identical to those internally used by the function `CosetTable`.

---

<b>CosetAction(G, H)</b>
--------------------------

Given a subgroup  $H$  of the group  $G$  of finite index, construct the permutation representation of  $G$ , induced by the action of  $G$  on the set of (right) cosets of  $H$  in  $G$ . The function returns:

- (a) The permutation representation  $f : G \rightarrow L \leq \text{Sym}(|G : H|)$ , induced by the action of  $G$  on the set of (right) cosets of  $H$  in  $G$ ;
- (b) The epimorphic image  $L$  of  $G$  under the representation  $f$ ;
- (c) The kernel  $K$  of the representation  $f$ .

**CosetImage(G, H)**

Given a subgroup  $H$  of the group  $G$  of finite index, construct the permutation group, induced by the action of  $G$  on the set of (right) cosets of  $H$  in  $G$ . The returned group is the epimorphic image  $L$  of  $G$  under the permutation representation  $f : G \rightarrow L \leq \text{Sym}(|G:H|)$ , induced by the action of  $G$  on the set of (right) cosets of  $H$  in  $G$ .

**CosetKernel(G, H)**

Given a subgroup  $H$  of the group  $G$  of finite index, construct the kernel of the permutation representation  $f : G \rightarrow L \leq \text{Sym}(|G:H|)$ , induced by the action of  $G$  on the set of (right) cosets of  $H$  in  $G$ .

**Example H72E8**

We use the function `CosetAction` to construct a (non-faithful) permutation representation of the group  $G$  defined by the polycyclic presentation

$$\langle a, b, c \mid b^a = b * c, (a, c), (b, c) \rangle .$$

```
> F<a,b,c> := FreeGroup(3);
> rels := [ b^a=b*c, b^(a^-1)=b*c^-1 ];
> G<a,b,c> := quo<GrpGPC: F | rels>;
>
> S := sub<G|(a*b)^3, c^7, b^21>;
> Index(G, S);
441
> pi, P, K := CosetAction(G, S);
> P;
Permutation group P acting on a set of cardinality 441
> K;
GrpGPC : K of infinite order on 3 PC-generators
PC-Relations:
  K.2^K.1 = K.2 * K.3^63,
  K.2^(K.-1) = K.2 * K.-3^63
> Index(G, K);
3087
```

We express the generators of the kernel  $K$  in terms of the generators of  $G$ :

```
> {@ G!x : x in PCGenerators(K) @}
{@ a^21, b^21, c^7 @}
```

$\pi(S)$  is a point stabiliser in the transitive permutation group  $P$  of degree 441 and hence should have index 441 in  $P$ :

```
> pi(S);
Permutation group acting on a set of cardinality 441
> Index(P, pi(S));
441
```

## 72.8 The Subgroup Structure

### 72.8.1 General Subgroup Constructions

The operators and functions which construct a subgroup of a polycyclic group always return the subgroup as a polycyclic group.

$H \sim g$
------------

Conjugate( $H, g$ )
---------------------

Construct the conjugate  $g^{-1} * H * g$  of the group  $H$  under the action of the element  $g$ . The group  $H$  and the element  $g$  must belong to a common group.

$H \sim G$
------------

ncl< $G \mid H$ >
-------------------

NormalClosure( $G, H$ )
-------------------------

Given a subgroup  $H$  of the group  $G$ , construct the normal closure of  $H$  in  $G$ .

CommutatorSubgroup( $G, H, K$ )
---------------------------------

CommutatorSubgroup( $H, K$ )
------------------------------

Construct the commutator subgroup of groups  $H$  and  $K$ , where  $H$  and  $K$  are subgroups of a common group  $G$ .

### 72.8.2 Subgroup Constructions Requiring a Nilpotent Covering Group

The operators and functions described in this section require the existence of a nilpotent covering group. They are based on algorithms published in [Lo98]. Again, the constructed subgroup is returned as a polycyclic group.

$H$ meet $K$
--------------

Given two groups  $H$  and  $K$ , contained in some common group  $G$  which is nilpotent, construct the intersection of  $H$  and  $K$ .

$H$ meet:= $K$
----------------

Given two groups  $H$  and  $K$ , contained in some common group  $G$  which is nilpotent, replace  $H$  with the intersection of  $H$  and  $K$ .

Centraliser( $G, g$ )
-----------------------

Centralizer( $G, g$ )
-----------------------

The subgroup of  $G$  centralising  $g$ . Both  $g$  and  $G$  must be contained in some common nilpotent group.

`Centraliser(G, H)`

`Centralizer(G, H)`

The subgroup of  $G$  centralising  $H$ . Both  $H$  and  $G$  must be subgroups of some common nilpotent group.

`Core(G, H)`

The maximal normal subgroup of the nilpotent group  $G$  that is contained in the subgroup  $H$  of  $G$ .

`Normaliser(G, H)`

`Normalizer(G, H)`

The subgroup of  $G$  normalising  $H$ . Both  $H$  and  $G$  must be subgroups of some common nilpotent group.

## 72.9 General Group Properties

`IsAbelian(G)`

Returns `true` if the group  $G$  is abelian, `false` otherwise.

`IsCyclic(G)`

Returns `true` if the group  $G$  is cyclic, `false` otherwise.

`IsElementaryAbelian(G)`

Returns `true` if the group  $G$  is elementary abelian, `false` otherwise. The following definition is used:

A group  $G$  is called elementary abelian if it is an abelian  $p$ -group of exponent  $p$  for some prime  $p$ .

`IsFinite(G)`

Returns `true` if the group  $G$  is finite, `false` otherwise.

`IsNilpotent(G)`

Returns `true` if the group  $G$  is nilpotent, `false` otherwise. This function uses an algorithm described in [Lo98].

`IsPerfect(G)`

Returns `true` if the group  $G$  is perfect, `false` otherwise. A polycyclic group  $G$  is perfect, if and only if it is trivial.

`IsSimple(G)`

Returns `true` if the group  $G$  is simple, `false` otherwise. A polycyclic group is simple, if and only if it is cyclic of prime order.

IsSoluble(G)
--------------

IsSolvable(G)
---------------

Returns **true** if the group  $G$  is solvable, **false** otherwise. Every polycyclic group is solvable.

### 72.9.1 General Properties of Subgroups

IsCentral(G, H)
-----------------

Returns **true** if the subgroup  $H$  of the group  $G$  lies in the centre of  $G$ , **false** otherwise.

IsNormal(G, H)
----------------

Returns **true** if the subgroup  $H$  of the group  $G$  is a normal subgroup of  $G$ , **false** otherwise.

### 72.9.2 Properties of Subgroups Requiring a Nilpotent Covering Group

The functions described in this section require the existence of a nilpotent covering group. They are based on algorithms published in [Lo98].

IsConjugate(G, H, K)
----------------------

Given groups  $G$ ,  $H$  and  $K$  with a nilpotent common covering group, return the value **true** if there exists  $c \in G$  such that  $H^c = K$ . If so, the function returns such a conjugating element as second value.

IsSelfNormalising(G, H)
-------------------------

IsSelfNormalizing(G, H)
-------------------------

Returns **true** if the subgroup  $H$  of the nilpotent group  $G$  is self-normalising in  $G$ , **false** otherwise.

#### Example H72E9

---

We define a group  $G$  on 5 generators  $a, \dots, e$  of infinite order by fixing the commutators of the generators:

$$(b, a) = e^2, \quad (d, c) = e^3$$

All other pairs of generators commute.

```
> F<a,b,c,d,e> := FreeGroup(5);
> rels := [ b^a = b*e^2, b^(a^-1) = b*e^-2, d^c = d*e^3,
>          d^(c^-1) = d*e^-3 ];
> G<a,b,c,d,e> := quo< GrpGPC: F | rels >;
> IsNilpotent(G);
true
```

Since  $G$  is nilpotent, we can compute intersections of subgroups of  $G$ .

We define the subgroups generated by  $a, \dots, e$  and their nontrivial commutator groups as subgroups of  $G$ .

```
> H1 := sub<G|a>;
> H2 := sub<G|b>;
> H3 := sub<G|c>;
> H4 := sub<G|d>;
> H5 := sub<G|e>;
>
> C12 := CommutatorSubgroup(H1, H2);
> {@ G!x : x in PCGenerators(C12) @}
{@ e^2 @}
> C12 subset H5;
true
>
> C34 := CommutatorSubgroup(H3, H4);
> {@ G!x : x in PCGenerators(C34) @}
{@ e^3 @}
> C34 subset H5;
true
```

Finally, we compute the intersection  $C$  of  $C12$  and  $C13$ .

```
> C := C12 meet C34;
> {@ G!x : x in PCGenerators(C) @}
{@ e^6 @}

```

This intersection  $C$  is cyclic and central in  $G$ .

```
> IsCyclic(C);
true
> IsCentral(G, C);
true
```

### Example H72E10

---

Consider the nilpotent group  $G := D_{16} \wr 2$  generated by the 5 generators  $a, b, c, d, t$  with the relations

$$\begin{aligned} a^2 = 1, \quad b^{16} = 1, \quad b^a = b^{15} \\ c^2 = 1, \quad d^{16} = 1, \quad d^c = d^{15} \\ t^2 = 1, \quad a^t = c, \quad b^t = d, \quad c^t = a, \quad d^t = b \end{aligned}$$

(All other pairs of generators commute.)

```
> F<t, a,b, c,d> := FreeGroup(5);
> G<t, a,b, c,d> := quo<GrpGPC: F | a^2, b^16, b^a=b^15,
>                                     c^2, d^16, d^c=d^15,
>                                     t^2, a^t=c, b^t=d, c^t=a, d^t=b>;
> IsNilpotent(G);
```

true

Since  $G$  is nilpotent, we can compute normalisers and centralisers in  $G$ .

We define the (dihedral) subgroup  $D3$  of  $G$  generated by  $ac$  and  $bd$  and compute its normaliser in  $G$  and its centraliser in the (dihedral) subgroup  $D2$  of  $G$  generated by  $c$  and  $d$ .

```
> D2 := sub<G|c,d>;
>
> D3<u,v> := sub<G|a*c, b*d>;
> D3;
GrpGPC : D3 of order 2^5 on 2 PC-generators
PC-Relations:
  u^2 = Id(D3),
  v^16 = Id(D3),
  v^u = v^15
>
> N3 := Normaliser(G, D3);
> PCGenerators(N3, G);
{@ t, a * c, b * d, d^8 @}
>
> C3 := Centraliser(D2, D3);
> PCGenerators(C3, G);
{@ d^8 @}
```

Finally we compute the centraliser of the element  $t$  in  $G$ .

```
> Ct := Centraliser(G, t);
> PCGenerators(Ct, G);
{@ t, a * c, b * d @}
```

## 72.10 Normal Structure and Characteristic Subgroups

### 72.10.1 Characteristic Subgroups and Subgroup Series

Centre( $G$ )
---------------

Center( $G$ )
---------------

The centre of the group  $G$ . For nilpotent groups the centre is computed using the centraliser algorithm [Lo98]. Otherwise, it is computed as the simultaneous fixed point space of the action of the generators of  $G$  on the centre of the Fitting subgroup of  $G$  [Eic01].

DerivedLength( $G$ )
----------------------

The derived length of the group  $G$ .

**DerivedSeries(G)**

The derived series of the group  $G$ . The series is returned as a sequence of subgroups.

**DerivedSubgroup(G)****DerivedGroup(G)**

The derived subgroup of the group  $G$ .

**EFASeries(G)**

Returns a normal series of  $G$ , the factors of which are either elementary abelian  $p$ -groups or free abelian groups.

**FittingLength(G)**

The Fitting length of the group  $G$ , i.e. the smallest integer  $k$  such that  $F_k = G$  where the groups  $F_i$  are defined recursively by  $F_0 := 1$  and  $F_i/F_{i-1} := \text{Fit}(G/F_{i-1})$  ( $i > 0$ ). Note that such a  $k$  exists for every polycyclic group  $G$ .

**FittingSeries(G)**

The Fitting series of the group  $G$ , where the groups  $F_i$  are defined recursively by  $F_0 := 1$  and  $F_i/F_{i-1} := \text{Fit}(G/F_{i-1})$  ( $i > 0$ ). The series is returned as the sequence  $[F_0, \dots, F_k]$  of subgroups of  $G$ . Note that every polycyclic group  $G$  has a finite Fitting series ending in  $G$ .

**FittingSubgroup(G)****FittingGroup(G)**

The Fitting subgroup of the group  $G$ , i.e. the maximal nilpotent normal subgroup of  $G$ . This function uses an algorithm described in [Eic01].

**HasComputableLCS(G)**

This function returns the value **true** if the lower central series of  $G$  is computable, otherwise it returns the value **false**. This is useful to avoid runtime errors, when **LowerCentralSeries** is called in user written loops or functions.

**LowerCentralSeries(G)**

The lower central series for the group  $G$ . The series is returned as a sequence of subgroups. Since infinite polycyclic groups need not satisfy the descending chain condition for subgroups, computation of the lower central series may fail. To determine if the series can be computed and thereby avoid runtime errors, the function **HasComputableLCS** may be used. This function uses an algorithm described in [Lo98].

**NilpotencyClass(G)**

The nilpotency class of the group  $G$ . If  $G$  is not nilpotent, then  $-1$  is returned.

**NilpotentPresentation(G)**

A polycyclic presentation is called *nilpotent*, if for all  $i = 1, \dots, n$ ,  $G_{i+1}$  is normal in  $G$  and  $G_i/G_{i+1}$  is central in  $G/G_{i+1}$ . Every nilpotent polycyclic group has a nilpotent polycyclic presentation. A suitable polycyclic series can be obtained by refining the lower central series.

The function `NilpotentPresentation` computes a group  $N$  isomorphic to  $G$ , given by a nilpotent polycyclic presentation and the isomorphism from  $G$  to  $N$ .

**SemisimpleEFASeries(G)**

Returns a normal series of  $G$ , the factors of which are either elementary abelian  $p$ -groups which are semisimple as  $\mathbf{F}_p[G]$ -modules or free abelian groups which are semisimple as  $\mathbf{Q}[G]$ -modules.

The normal series returned by the function `SemisimpleEFASeries` is a refinement of the normal series returned by the function `EFASeries`.

**UpperCentralSeries(G)**

The upper central series  $[Z_0, \dots, Z_k]$  of the group  $G$ , where the groups  $Z_k$  are defined recursively by  $Z_0 := 1$  and  $Z_i/Z_{i-1} := Z(G/Z_{i-1})$  ( $i > 0$ ). The series is returned as a sequence of subgroups of  $G$ . Note that since polycyclic groups satisfy the ascending chain condition for subgroups, every polycyclic group  $G$  has a finite upper central series.

**Example H72E11**

---

The dihedral group of order 32 is nilpotent; we compute its lower central series.

```
> D16<a,b> := DihedralGroup(GrpGPC, 16);
> IsNilpotent(D16);
true
> NilpotencyClass(D16);
4
> L := LowerCentralSeries(D16);
```

The generators of the subgroups in the lower central series expressed as elements of  $D16$  are:

```
> for i := 1 to 1+NilpotencyClass(D16) do
>   print i, ":", {@ D16!x : x in PCGenerators(L[i]) @};
> end for;
1 : {@ a, b @}
2 : {@ b^2 @}
3 : {@ b^4 @}
4 : {@ b^8 @}
5 : {@ @}
```

We compute a nilpotent presentation and express the new generators in terms of  $a$  and  $b$ :

```
> N<p,q,r,s,t>, f := NilpotentPresentation(D16);
> N;
```

GrpGPC : N of order  $2^5$  on 5 PC-generators

PC-Relations:

```

p^2 = Id(N),
q^2 = r,
r^2 = s,
s^2 = t,
t^2 = Id(N),
q^p = q * r * s * t,
r^p = r * s * t,
s^p = s * t

```

```
> {@ x@@f : x in PCGenerators(N) @};
```

```
{@ a, b, b^2, b^4, b^8 @}
```

The infinite dihedral group has an infinite, strictly descending, lower central series which cannot be computed:

```
> D := DihedralGroup(GrpGPC, 0);
```

```
> HasComputableLCS(D);
```

```
false
```

It is easy to see, that  $b^{2^{i-1}}$  would be a generator of  $L_i$ .

The symmetric group on 3 letters is not nilpotent, but has a lower central series which becomes stationary and which can be computed:

```
> F2<a,b> := FreeGroup(2);
```

```
> rels := [ a^2 = F2!1, b^3 = F2!1, b^a = b^2 ];
```

```
> G<a,b> := quo<GrpGPC : F2 | rels>;
```

```
> G;
```

GrpGPC : G of order  $6 = 2 * 3$  on 2 PC-generators

PC-Relations:

```

a^2 = Id(G),
b^3 = Id(G),
b^a = b^2

```

```
> IsNilpotent(G);
```

```
false
```

```
> HasComputableLCS(G);
```

```
true
```

```
> L := LowerCentralSeries(G);
```

```
> for i := 1 to #L do
```

```
>   print i, ":", {@ G!x : x in PCGenerators(L[i]) @};
```

```
> end for;
```

```
1 : {@ a, b @}
```

```
2 : {@ b @}
```

### 72.10.2 The Abelian Quotient Structure of a Group

`AbelianQuotient(G)`

The maximal abelian quotient  $G/G'$  of the group  $G$  as `GrpAb` (cf. Chapter 69). The natural epimorphism is returned as second return value.

`AbelianQuotientInvariants(G)`

`AQInvariants(G)`

Returns a sequence containing the invariants of the maximal abelian quotient  $G/G'$  of the group  $G$ . Each infinite cyclic factor of  $G/G'$  is represented by zero.

`ElementaryAbelianQuotient(G, p)`

The maximal  $p$ -elementary abelian quotient of the group  $G$  as `GrpAb` (cf. Chapter 69). The natural epimorphism is returned as second return value.

`FreeAbelianQuotient(G)`

The maximal free abelian quotient of the group  $G$  as `GrpAb` (cf. Chapter 69). The natural epimorphism is returned as second return value.

### 72.11 Conjugacy

The functions described in this section require the existence of a nilpotent covering group. They are based on algorithms published in [Lo98].

`IsConjugate(G, g, h)`

Given elements  $g$  and  $h$  and a group  $G$ , which are contained in some nilpotent common group, return the value `true` if there exists  $c \in G$  such that  $g^c = h$ . If so, the function returns such a conjugating element as second value.

`IsConjugate(G, H, K)`

Given groups  $G$ ,  $H$  and  $K$  with a nilpotent common covering group, return the value `true` if there exists  $c \in G$  such that  $H^c = K$ . If so, the function returns such a conjugating element as second value.

#### Example H72E12

---

We again consider the nilpotent group  $G := D_{16} \wr 2$ .

```
> F<t, a,b, c,d> := FreeGroup(5);
> G<t, a,b, c,d> := quo<GrpGPC: F | a^2, b^16, b^a=b^15,
>                               c^2, d^16, d^c=d^15,
>                               t^2, a^t=c, b^t=d, c^t=a, d^t=b>;
> IsNilpotent(G);
true
```

Since  $G$  is nilpotent, a test for conjugacy in  $G$  is available.

We define the following subgroups of  $G$ :  $D1$  generated by  $a$  and  $b$ ,  $D2$  generated by  $c$  and  $d$  and  $D3$  generated by  $ac$  and  $bd$ .

```
> D1 := sub<G|a,b>;
> D2 := sub<G|c,d>;
> D3<u,v> := sub<G|a*c, b*d>;
>
```

$D1$  and  $D2$  are, of course, conjugate in  $G$ ;  $t$  is a conjugating element.

```
> IsConjugate(G, D1, D2);
true t
```

The elements  $b$  and  $d^{-1}$  are conjugate in  $G$ ; we compute a conjugating element.

```
> IsConjugate(G, b, d^-1);
true t * a * c
```

However, neither the subgroups  $D1$  and  $D2$  nor the elements  $b$  and  $d^{-1}$ , are conjugate in the subgroup  $D3$ .

```
> IsConjugate(D3, D1, D2);
false
> IsConjugate(D3, b, d^-1);
false
```

## 72.12 Representation Theory

This section describes some functions for creating  $R[G]$ -modules for a polycyclic group  $G$ , which are unique for this category or have special properties when called for polycyclic groups. For a complete description of the functions available for creating and working with  $R[G]$ -modules we refer to Chapter 89.

Note that the function `GModuleAction` can be used to extract the matrix representation associated to an  $R[G]$ -module.

### EFAModuleMaps( $G$ )

Every polycyclic group  $G$  has a normal series  $G = N_1 \triangleright N_2 \triangleright \dots \triangleright N_{r+1} = 1$ , such that every quotient  $M_i := N_i/N_{i+1}$  is either free abelian or  $p$ -elementary abelian for some prime  $p$ . The action of  $G$  by conjugation induces a  $\mathbf{Z}[G]$ -module structure on  $M_i$  if  $M_i$  is free abelian and an  $\mathbf{F}_p[G]$ -module structure if  $M_i$  is  $p$ -elementary abelian.

This function returns a sequence  $[f_1, \dots, f_r]$ , where  $f_i : N_i \rightarrow M_i$  is the natural epimorphism onto the additive group of an  $R_i[G]$ -module  $M_i$  ( $R_i \in \{\mathbf{F}_p, \mathbf{Z}\}$ ), constructed as above.

The functions `EFAModuleMaps` and `EFAModules` use the normal series returned by the function `EFASeries`.

Note that the kernels of the epimorphisms  $f_i$  can be computed and hence it is possible to form preimages of submodules of  $M_i$ , which are normal subgroups of  $G$  contained in  $N_i$  and containing  $N_{i+1}$ .

**EFAModules(G)**

Every polycyclic group  $G$  has a normal series  $G = N_1 \triangleright N_2 \triangleright \dots \triangleright N_{r+1} = 1$ , such that every quotient  $M_i := N_i/N_{i+1}$  is either free abelian or  $p$ -elementary abelian for some prime  $p$ . The action of  $G$  by conjugation induces a  $\mathbf{Z}[G]$ -module structure on  $M_i$  if  $M_i$  is free abelian and an  $\mathbf{F}_p[G]$ -module structure if  $M_i$  is  $p$ -elementary abelian.

This function returns a sequence  $[M_1, \dots, M_r]$  of  $R_i[G]$ -modules (where  $R_i \in \{\mathbf{F}_p, \mathbf{Z}\}$ ), constructed as above.

The functions **EFAModuleMaps** and **EFAModules** use the normal series returned by the function **EFASeries**.

**GModule(G, A, p)****GModule(G, A)**

Let  $A$  be a normal subgroup of the polycyclic group  $G$ . If  $p = 0$ , the function returns the  $\mathbf{Z}[G]$ -module corresponding to the conjugation action of  $G$  on the maximal free abelian quotient of  $A$ . If  $p$  is a prime, it returns the  $\mathbf{F}_p[G]$ -module corresponding to the conjugation action of  $G$  on the maximal  $p$ -elementary abelian quotient of  $A$ . The epimorphism  $\pi : A \rightarrow M$  onto the additive group of the constructed module  $M$  is returned as second return value. Note that the kernel of  $\pi$  can be computed and hence it is possible to form preimages of submodules of  $M$ , which are normal subgroups of  $G$  contained in  $A$ .

If the maximal abelian quotient  $A/A'$  of  $A$  is either free abelian or  $p$ -elementary abelian for some prime  $p$ ,  $p$  can be omitted in the function call.

Note that it is the user's responsibility to ensure that  $A$  is in fact normal in  $G$ .

**GModule(G, A, B, p)****GModule(G, A, B)**

Let  $A$  and  $B < A$  be normal subgroups of the polycyclic group  $G$ . If  $p = 0$ , the function returns the  $\mathbf{Z}[G]$ -module corresponding to the conjugation action of  $G$  on the maximal free abelian quotient of  $A/B$ . If  $p$  is a prime, it returns the  $\mathbf{F}_p[G]$ -module corresponding to the conjugation action of  $G$  on the maximal  $p$ -elementary abelian quotient of  $A/B$ . The epimorphism  $\pi : A \rightarrow M$  onto the additive group of the constructed module  $M$  is returned as second return value. Note that the kernel of  $\pi$  can be computed and hence it is possible to form preimages of submodules of  $M$ , which are normal subgroups of  $G$  contained in  $A$  and containing  $B$ .

If the maximal abelian quotient of  $A/B$  is either free abelian or  $p$ -elementary abelian for some prime  $p$ ,  $p$  can be omitted in the function call.

Note that it is the user's responsibility to ensure that  $A$  and  $B$  are in fact normal in  $G$ .

**GModulePrimes(G, A)**

Let  $G$  be a polycyclic group and  $A$  a normal subgroup of  $G$ . Given any prime  $p$ , the maximal  $p$ -elementary abelian quotient of  $A$  can be viewed as an  $\mathbf{F}_p[G]$ -module  $M_p$ . The maximal free abelian quotient of  $A$  can be viewed as a  $\mathbf{Z}[G]$ -module  $M_0$ . This function determines those primes  $p$  for which the module  $M_p$  is non-trivial (i.e. not zero-dimensional) and the dimensions of the corresponding modules  $M_p$ . The return value is a multiset  $S$ . If  $0 \notin S$ , the maximal abelian quotient of  $A$  is finite and the multiplicity of  $p$  is the dimension of  $M_p$ . If  $S$  contains 0 with multiplicity  $m$ , the maximal abelian quotient of  $A$  contains  $m$  copies of  $\mathbf{Z}$ . In this case, the rank of  $M_0$  is  $m$  and  $M_p$  is non-trivial for every prime  $p$  and its rank is the sum of  $m$  and the multiplicity of  $p$  in  $S$ .

**GModulePrimes(G, A, B)**

Let  $G$  be a polycyclic group,  $A$  a normal subgroup of  $G$  and  $B$  a normal subgroup of  $G$  contained in  $A$ . Given any prime  $p$ , the maximal  $p$ -elementary abelian quotient of  $A/B$  can be viewed as an  $\mathbf{F}_p[G]$ -module  $M_p$ . The maximal free abelian quotient of  $A/B$  can be viewed as a  $\mathbf{Z}[G]$ -module  $M_0$ . This function determines those primes  $p$  for which the module  $M_p$  is non-trivial (i.e. not zero-dimensional) and the dimensions of the corresponding modules  $M_p$ . The return value is a multiset  $S$ . If  $0 \notin S$ , the maximal abelian quotient of  $A/B$  is finite and the multiplicity of  $p$  is the dimension of  $M_p$ . If  $S$  contains 0 with multiplicity  $m$ , the maximal abelian quotient of  $A/B$  contains  $m$  copies of  $\mathbf{Z}$ . In this case, the rank of  $M_0$  is  $m$  and  $M_p$  is non-trivial for every prime  $p$  and its rank is the sum of  $m$  and the multiplicity of  $p$  in  $S$ .

**SemisimpleEFAModuleMaps(G)**

Every polycyclic group  $G$  has a normal series  $G = N_1 \triangleright N_2 \triangleright \dots \triangleright N_{r+1} = 1$ , such that every quotient  $M_i := N_i/N_{i+1}$  is either free abelian and semisimple as a  $\mathbf{Q}[G]$ -module or  $p$ -elementary abelian and semisimple as an  $\mathbf{F}_p[G]$ -module for some prime  $p$ .

This function returns a sequence  $[f_1, \dots, f_r]$ , where  $f_i : N_i \rightarrow M_i$  is the natural epimorphism onto the additive group of an  $R_i[G]$ -module  $M_i$  ( $R_i \in \{\mathbf{F}_p, \mathbf{Z}\}$ ), constructed as above.

The functions **SemisimpleEFAModules** and **SemisimpleEFAModuleMaps** use the normal series returned by the function **SemisimpleEFASeries**. Moreover, this normal series is a refinement of the normal series returned by the function **EFASeries**.

Note that the kernels of the epimorphisms  $f_i$  can be computed and hence it is possible to form preimages of submodules of  $M_i$ , which are normal subgroups of  $G$  contained in  $N_i$  and containing  $N_{i+1}$ .

**SemisimpleEFAModules(G)**

Every polycyclic group  $G$  has a normal series  $G = N_1 \triangleright N_2 \triangleright \dots \triangleright N_{r+1} = 1$ , such that every quotient  $M_i := N_i/N_{i+1}$  is either free abelian and semisimple as a  $\mathbf{Q}[G]$ -module or  $p$ -elementary abelian and semisimple as an  $\mathbf{F}_p[G]$ -module for some prime  $p$ .

This function returns a sequence  $[M_1, \dots, M_r]$  of  $R_i[G]$ -modules (where  $R_i \in \{\mathbf{F}_p, \mathbf{Z}\}$ ), constructed as above.

The functions `SemisimpleEFAModules` and `SemisimpleEFAModuleMaps` use the normal series returned by the function `SemisimpleEFASeries`. Moreover, this normal series is a refinement of the normal series returned by the function `EFASeries`.

### Example H72E13

---

Consider the group  $G$  defined by the polycyclic presentation

$$\langle a, b, c, d, e \mid c^6, e^3, d^c = de, e^c = e^2, b^a = b^{-1}, b^{a^{-1}} = b^{-1}, \\ c^b = ce, c^{b^{-1}} = ce, d^b = d^{-1}, d^{b^{-1}} = d^{-1}, e^b = e^2, e^{b^{-1}} = e^2 \rangle.$$

(Trivial commutator relations have been omitted.)

```
> F<a,b,c,d,e> := FreeGroup(5);
> G<a,b,c,d,e> := quo< GrpGPC : F | c^6 = F!1, e^3 = F!1,
>                               d^c=d*e, e^c=e^2,
>                               b^a=b^-1,
>                               b^(a^-1)=b^-1,
>                               c^b=c*e,
>                               c^(b^-1)=c*e,
>                               d^b=d^-1,
>                               d^(b^-1)=d^-1,
>                               e^b=e^2,
>                               e^(b^-1)=e^2 >;
```

The subgroup  $H$  of  $G$  generated by  $c, d, e$  is normal in  $G$ .

```
> H := sub< G | c,d,e >;
> IsNormal(G, H);
true
```

We determine the primes such that the action of  $G$  on  $H$  yields non-trivial modules.

```
> GModulePrimes(G, H);
{* 0, 2, 3 *}
```

We construct the  $\mathbf{F}_3[G]$ -module  $M$  given by the action of  $G$  on the maximal 3-elementary abelian quotient of  $H$  and the natural epimorphism  $\pi$  from  $H$  onto the additive group of  $M$ .

```
> M, pi := GModule(G, H, 3);
> M;
GModule M of dimension 2 over GF(3)
```

Using the function `Submodules`, we obtain the submodules of  $M$ . Their preimages under  $\pi$  are precisely the normal subgroups of  $G$  which are contained in  $H$  and contain  $\ker(\pi)$ .

```
> submod := Submodules(M);
> nsgs := [ m @@ pi : m in submod ];
> [ PCGenerators(s, G) : s in nsgs ];
```

```
[
  {0 c^3, d^3, e 0},
  {0 c, d^3, e 0},
  {0 c^3, d, e 0},
  {0 c, d, e 0}
]
```

---

**Example H72E14**

Consider the group defined by the polycyclic presentation

$$\langle a, b, c, d, e \mid a^5, b^5, c^6, d^5, e^3, b^a = bd \rangle.$$

```
> F<a,b,c,d,e> := FreeGroup(5);
> G<a,b,c,d,e> := quo< GrpGPC : F |
>           a^5, b^5, c^6, d^5, e^3, b^a = b*d >;
```

Obviously the subgroup of  $G$  generated by  $b, c, d, e$  is normal in  $G$ .

```
> H := sub< G | b,c,d,e >;
> IsNormal(G, H);
true
```

We use the function `GModulePrimes` to determine the set of primes  $p$  for which the action of  $G$  on the maximal  $p$ -elementary abelian quotient of  $H$  induces a nontrivial  $\mathbf{F}_p[G]$ -module.

```
> P := GModulePrimes(G, H);
> 0 in P;
false
```

0 is not contained in  $P$ , i.e. the maximal free abelian quotient of  $H$  is trivial. Hence, there are only finitely many primes, satisfying the condition above.

We loop over the distinct elements of  $P$  and for each element  $p$  we construct the induced  $\mathbf{F}_p[G]$ -module, print its dimension and check whether it is decomposable. Note that the dimension of the module for  $p$  must be equal to the multiplicity of  $p$  in  $P$ .

```
> for p in MultisetToSet(P) do
>   M := GModule(G, H, p);
>   dim := Dimension(M);
>   decomp := IsDecomposable(M);
>
>   assert dim eq Multiplicity(P, p);
>
>   print "prime", p, ": module of dimension", dim;
>   if decomp then
>     print " has a nontrivial decomposition";
>   else
>     print " is indecomposable";
>   end if;
```

```

> end for;
prime 2 : module of dimension 1
  is indecomposable
prime 3 : module of dimension 2
  has a nontrivial decomposition
prime 5 : module of dimension 2
  is indecomposable

```

### Example H72E15

---

The Fitting subgroup of a polycyclic group  $G$  can be characterised as the intersection of the centralisers in  $G$  of the semisimple  $G$ -modules defined by the action of  $G$  on the factors of a semisimple EFA-series of  $G$ . The centraliser in  $G$  of a  $G$ -module is just the kernel of the action map.

We illustrate this with the group  $G$  defined in the example above.

```

> F<a,b,c,d,e> := FreeGroup(5);
> G<a,b,c,d,e> := quo< GrpGPC : F | c^6 = F!1, e^3 = F!1,
>                               b^a = b * d,
>                               b^(a^-1) = b * d^-1 >;

```

We first construct the  $G$ -modules defined by the action of  $G$  on the factors of a semisimple EFA-series of  $G$ .

```

> modules := SemisimpleEFAModules(G);
> modules;
[
  GModule of dimension 2 over Integer Ring,
  GModule of dimension 1 over Integer Ring,
  GModule of dimension 1 over GF(2),
  GModule of dimension 2 over GF(3)
]

```

Now, we compute the intersection of the kernels of the module action maps, which can be obtained using the function `GModuleAction`.

```

> S := G;
> for m in modules do
>   S meet:= Kernel(GModuleAction(m));
> end for;

```

Finally, we compare the result with the Fitting subgroup of  $G$ , returned by the MAGMA function `FittingSubgroup`.

```

> S eq FittingSubgroup(G);
true

```

**Example H72E16**

---

The functions `EFAModuleMaps` and `SemisimpleEFAModuleMaps` are useful whenever it is desired to refine an EFA-series or a semisimple EFA-series by computing the subgroups corresponding to submodules of the  $G$ -modules given by the factors of the series. Consider again the group defined above.

```
> F<a,b,c,d,e> := FreeGroup(5);
> G<a,b,c,d,e> := quo< GrpGPC : F | c^6 = F!1, e^3 = F!1,
>                               b^a = b * d,
>                               b^(a^-1) = b * d^-1 >;
```

We extract the map  $f$  from  $G$  (the first group in any EFA-series of  $G$ ) onto the module given by the first factor of an EFA-series of  $G$ .

```
> f := EFAModuleMaps(G)[1];
> f;
Mapping from: GrpGPC: G to GModule of dimension 2 over Integer
Ring
```

The module itself can be accessed as the codomain of  $f$ .

```
> M := Codomain(f);
> M;
GModule M of dimension 2 over Integer Ring
```

Spinning up random elements, we try to construct a submodule  $S$  of  $M$ .

```
> repeat
>   S := sub<M|[Random(-1, 1): i in [1 .. Dimension(M)]]>;
>   until Dimension(S) gt 0 and S ne M;
> S;
GModule S of dimension 1 over Integer Ring
```

The preimage  $N$  of  $S$  under  $f$  is a normal subgroup of  $G$ , which lies between the first and the second subgroup of the original EFA-series for  $G$ .

```
> N := S @@ f;
> PCGenerators(N, G);
{@ a^2 * b^4, c, d, e @}
```

---

**72.13 Power Groups**

Parent( $G$ )

The `PowerStructure` of category `GrpGPC`.

PowerGroup( $G$ )

The set of all subgroups of  $G$ . This is very useful when constructing sets of polycyclic groups. If the user will be building a set of subgroups of a polycyclic group  $G$ , then it is best to specify the set's universe to be `PowerGroup( $G$ )`. If the set's universe is not specified it will be the parent structure of  $G$  as returned by `Parent( $G$ )`.

## 72.14 Bibliography

- [**CLGM<sup>+</sup>95**] Frank Celler, Charles R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and E. A. O'Brien. Generating random elements of a finite group. *Comm. Algebra*, 23(13):4931–4948, 1995.
- [**Eic01**] Bettina Eick. On the Fitting subgroup of a polycyclic-by-finite group and its applications. *J. Algebra*, 242(1):176–187, 2001.
- [**Geb02**] Volker Gebhardt. Efficient collection in infinite polycyclic groups. *J. Symbolic Comput.*, 34(3):213–228, 2002.
- [**Lo98**] Eddie H. Lo. Finding intersections and normalizers in finitely generated nilpotent groups. *J. Symbolic Comput.*, 25(1):45–59, 1998.
- [**Sim94**] Charles C. Sims. *Computation with finitely presented groups*. Cambridge University Press, Cambridge, 1994.

# 73 BRAID GROUPS

<b>73.1 Introduction . . . . .</b>	<b>2289</b>	<i>73.4.1 Accessing Information . . . . .</i>	<i>2303</i>
73.1.1 Lattice Structure and Simple Elements . . . . .	2290	Parent(u)	2303
73.1.2 Representing Elements of a Braid Group . . . . .	2291	#	2303
73.1.3 Normal Form for Elements of a Braid Group . . . . .	2292	CanonicalFactorRepresentation(u: -)	2303
73.1.4 Mixed Canonical Form and Lattice Operations . . . . .	2293	CFP(u: -)	2303
73.1.5 Conjugacy Testing and Conjugacy Search . . . . .	2294	WordToSequence(u: -)	2303
<b>73.2 Constructing and Accessing Braid Groups . . . . .</b>	<b>2296</b>	ElementToSequence(u: -)	2303
BraidGroup(n: -)	2296	Eltseq(u: -)	2303
BraidGroup(GrpBrd, n: -)	2296	InducedPermutation(u)	2304
GetPresentation(B)	2296	CanonicalLength(u: -)	2304
SetPresentation(~B, s)	2296	Infimum(u: -)	2304
GetForceCFP(B)	2296	Supremum(u: -)	2304
SetForceCFP(~B, b)	2296	SuperSummitCanonicalLength(u: -)	2305
GetElementPrintFormat(B)	2296	SuperSummitInfimum(u: -)	2305
SetElementPrintFormat(~B, s)	2296	SuperSummitSupremum(u: -)	2305
NumberOfStrings(B)	2297	<i>73.4.2 Computing Normal Forms of Elements . . . . .</i>	<i>2306</i>
NumberOfGenerators(B)	2297	LeftNormalForm(u: -)	2307
Ngens(B)	2297	NormalForm(u: -)	2307
<b>73.3 Creating Elements of a Braid Group . . . . .</b>	<b>2297</b>	LeftNormalForm(~u: -)	2307
Representative(B)	2297	NormalForm(~u: -)	2307
Rep(B)	2297	RightNormalForm(u: -)	2307
Identity(B)	2297	RightNormalForm(~u: -)	2307
Id(B)	2297	LeftMixedCanonicalForm(u: -)	2307
!	2297	MixedCanonicalForm(u: -)	2307
FundamentalElement(B: -)	2297	RightMixedCanonicalForm(u: -)	2308
Generators(B: -)	2297	<i>73.4.3 Arithmetic Operators and Functions for Elements . . . . .</i>	<i>2309</i>
.	2297	*	2310
.	2298	*:=	2310
!	2298	/	2310
!	2298	/:=	2310
!	2298	^	2310
!	2299	^:=	2310
!	2299	^	2310
!	2299	^:=	2310
IsProductOfParallelDescending Cycles(p)	2299	Inverse(u)	2310
Random(B, r, s, m, n: -)	2299	Inverse(~u)	2310
RandomCFP(B, r, s, m, n: -)	2299	LeftConjugate(u, v)	2311
Random(B: -)	2299	LeftConjugate(~u, v)	2311
RandomCFP(B: -)	2299	LeftDiv(u, v)	2311
Random(B, m, n: -)	2299	LeftDiv(u, ~v)	2311
RandomWord(B, m, n: -)	2299	Cycle(u: -)	2311
RandomWord(B: -)	2299	Cycle(~u: -)	2311
<b>73.4 Working with Elements of a Braid Group . . . . .</b>	<b>2303</b>	Decycle(u: -)	2312
		Decycle(~u: -)	2312
		<i>73.4.4 Boolean Predicates for Elements . . . . .</i>	<i>2313</i>
		in	2313
		notin	2314
		IsEmptyWord(u: -)	2314
		AreIdentical(u, v: -)	2314
		IsSimple(u: -)	2314
		IsSuperSummitRepresentative(u: -)	2314

IsUltraSummitRepresentative(u: -)	2314	73.4.6 Invariants of Conjugacy Classes . .	2321
IsIdentity(u: -)	2314	PositiveConjugates(u: -)	2322
IsId(u: -)	2314	SuperSummitRepresentative(u: -)	2322
eq	2314	SuperSummitSet(u: -)	2322
ne	2315	UltraSummitRepresentative(u: -)	2322
le	2315	UltraSummitSet(u: -)	2322
IsLE(u, v: -)	2315	PositiveConjugatesProcess(u: -)	2325
IsLe(u, v: -)	2315	SuperSummitProcess(u: -)	2325
ge	2315	UltraSummitProcess(u: -)	2325
IsGE(u, v: -)	2315	BaseElement(P)	2325
IsGe(u, v: -)	2315	#	2325
IsConjugate(u, v: -)	2315	Representative(P)	2325
73.4.5 Lattice Operations . . . . .	2317	Rep(P)	2325
LeftGCD(u, v: -)	2318	IsEmpty(P)	2325
LeftGcd(u, v: -)	2318	Elements(P)	2325
LeftGreatestCommonDivisor(u, v: -)	2318	in	2326
GCD(u, v: -)	2318	notin	2326
Gcd(u, v: -)	2318	NextElement(~P)	2326
GreatestCommonDivisor(u, v: -)	2318	Complete(~P)	2326
RightGCD(u, v: -)	2318	MinimalElementConjugatingTo	
RightGcd(u, v: -)	2318	Positive(x, s: -)	2328
RightGreatestCommonDivisor(u, v: -)	2318	MinimalElementConjugatingTo	
LeftGCD(S: -)	2318	SuperSummit(x, s: -)	2328
LeftGcd(S: -)	2318	MinimalElementConjugatingTo	
LeftGreatestCommonDivisor(S: -)	2318	UltraSummit(x, s: -)	2328
GCD(S: -)	2318	Transport(x, s: -)	2329
Gcd(S: -)	2318	Pullback(x, s: -)	2329
GreatestCommonDivisor(S: -)	2318	<b>73.5 Homomorphisms . . . . .</b>	<b>2330</b>
RightGCD(S: -)	2318	73.5.1 General Remarks . . . . .	2330
RightGcd(S: -)	2318	73.5.2 Constructing Homomorphisms . .	2330
RightGreatestCommonDivisor(S: -)	2318	hom< >	2330
LeftLCM(u, v: -)	2319	73.5.3 Accessing Homomorphisms . . . .	2331
LeftLcm(u, v: -)	2319	e @ f	2331
LeftLeastCommonMultiple(u, v: -)	2319	f(e)	2331
LCM(u, v: -)	2319	B @ f	2331
Lcm(u, v: -)	2319	f(B)	2331
LeastCommonMultiple(u, v: -)	2319	u @@ f	2331
RightLCM(u, v: -)	2319	Domain(f)	2331
RightLcm(u, v: -)	2319	Codomain(f)	2331
RightLeastCommonMultiple(u, v: -)	2319	Image(f)	2331
LeftLCM(S: -)	2319	73.5.4 Representations of Braid Groups .	2334
LeftLcm(S: -)	2319	SymmetricRepresentation(B)	2334
LeftLeastCommonMultiple(S: -)	2319	BureauRepresentation(B)	2335
LCM(S: -)	2319	BureauRepresentation(B, p)	2335
Lcm(S: -)	2319	<b>73.6 Bibliography . . . . .</b>	<b>2336</b>
LeastCommonMultiple(S: -)	2319		
RightLCM(S: -)	2320		
RightLcm(S: -)	2320		
RightLeastCommonMultiple(S: -)	2320		

# Chapter 73

## BRAID GROUPS

### 73.1 Introduction

This chapter deals with another, quite specialised, class of finitely presented groups for which the word problem is solvable, the category of braid groups. The corresponding MAGMA category is called `GrpBrd`.

The notion of braid groups was introduced by Artin [Art47], who considered a sequence  $B_n$  ( $n = 1, 2, \dots$ ) of groups, where  $B_n$  is presented on  $n - 1$  generators  $\sigma_1, \dots, \sigma_{n-1}$  with the defining relations

$$\begin{aligned}\sigma_i \sigma_j &= \sigma_j \sigma_i & (1 \leq i < j < n, \quad j - i > 1), \\ \sigma_i \sigma_{i+1} \sigma_i &= \sigma_{i+1} \sigma_i \sigma_{i+1} & (1 \leq i < n - 1) \quad .\end{aligned}$$

$B_n$  is called the *braid group on  $n$  strings*. In the sequel, we refer to the above presentation as *Artin presentation* and to  $\sigma_1, \dots, \sigma_{n-1}$  as *Artin generators* of  $B_n$ .

Birman, Ko and Lee introduced an alternative way of presenting braid groups [BKL98]. Here,  $B_n$  is presented on  $n(n - 1)/2$  generators  $a_{r,t}$  ( $n \geq r > t \geq 1$ ) with the defining relations

$$\begin{aligned}a_{t,s} a_{r,q} &= a_{r,q} a_{t,s} & \text{for } n \geq t > s \geq 1, \quad n \geq r > q \geq 1, \quad (t - r)(t - q)(s - r)(s - q) > 0 \\ a_{t,s} a_{s,r} &= a_{t,r} a_{t,s} = a_{s,r} a_{t,r} & \text{for } n \geq t > s > r > 0 \quad .\end{aligned}$$

We refer to this presentation as *BKL presentation* and to  $a_{r,t}$  ( $n \geq r > t \geq 1$ ) as *BKL generators* of  $B_n$ .

A possible choice for the BKL generators in terms of Artin generators is  $a_{r,t} = (\sigma_{r-1} \cdots \sigma_{t+1}) \sigma_t (\sigma_{t+1}^{-1} \cdots \sigma_{r-1}^{-1})$ . This identification is used in MAGMA.

Recently, braid groups came under consideration as possible sources for public key cryptosystems [AAG99, KLC<sup>+</sup>00]. The features of braid groups which make them interesting for public key cryptography are the following.

- The basic group operations in braid groups can be implemented efficiently on a computer.
- The word problem in braid groups is solvable, that is, there is a normal form for elements of a braid group and elements can be compared. Moreover, there are algorithms which are able to compute the normal form of an element efficiently.
- There are several problems in braid groups which are believed to be mathematically hard and whose use for cryptographic purposes has been suggested. The most important examples are variations of the conjugacy problem.

However, both recent attacks on particular cryptosystems [GKT<sup>+</sup>02, HS03, Hug02, LL02, LP03] and advances in the analysis of the conjugacy problem [GM02, Geb03] in general shed some doubts on the security of braid group cryptosystems. At the time of this writing it is an open question whether braid group cryptosystems can be made secure by an appropriate choice of parameters and keys or whether they have to be considered as insecure. More research into these issues is necessary.

The MAGMA category `GrpBrd` was introduced mainly with these applications in mind. Focus was put on providing fast operations with elements and on giving the user as much control over the details of computations as possible.

### 73.1.1 Lattice Structure and Simple Elements

In this section we briefly recall the basic terminology used for describing elements of braid groups. More detailed descriptions can be found in [ECH<sup>+</sup>92] for the Artin presentation and in [BKL98] for the BKL presentation.

We remark that both Artin presentation and BKL presentation are special cases of so-called *Garside groups* [Deh02].

In the sequel, let  $M$  be either the Artin presentation or the BKL presentation of the braid group  $B$  on  $n$  strings, let  $X$  denote the generators of  $M$  and let  $R$  denote the relations of  $M$ . As the relations in  $R$  do not contain inverses of generators, we can interpret  $M$  as monoid presentation. We denote the finitely presented monoid defined by  $M$  by  $B^+$ . The natural homomorphism from  $B^+$  to  $B$  can be shown to be injective. We identify its image with  $B^+$  and call it the set of *positive* elements of  $B$ . Finally, we denote the identity of  $B$  by 1.

We can now define two partial orderings on  $B$ . For elements  $u, v \in B$  we say  $u \preceq v$ , if there exists a positive element  $a$  such that  $ua = v$ , and we say  $v \succeq u$ , if there exists a positive element  $a$  such that  $v = au$ . Note that these partial orderings are different;  $u \preceq v$  is not equivalent to  $v \succeq u$ .

$B$  can be shown to be a lattice with respect to both partial orderings, that is, for elements  $u, v \in B$  there are elements  $d_l, m_l, d_r, m_r \in B$  such that

$$\begin{aligned} d_l \preceq u, d_l \preceq v & \quad \text{and} \quad d \preceq u, d \preceq v \text{ implies } d \preceq d_l \text{ for all } d \in B \\ u \preceq m_l, v \preceq m_l & \quad \text{and} \quad u \preceq m, v \preceq m \text{ implies } m_l \preceq m \text{ for all } m \in B \\ u \succeq d_r, v \succeq d_r & \quad \text{and} \quad u \succeq d, v \succeq d \text{ implies } d_r \succeq d \text{ for all } d \in B \\ m_r \succeq u, m_r \succeq v & \quad \text{and} \quad m \succeq u, m \succeq v \text{ implies } m \succeq m_r \text{ for all } m \in B \quad . \end{aligned}$$

We call  $d_l, m_l, d_r$  and  $m_r$  the *left-gcd*, the *left-lcm*, the *right-gcd* and the *right-lcm*, respectively, of  $u$  and  $v$ .

It can be shown that the left-lcm of the elements of  $X$  and the right-lcm of the elements of  $X$  are equal; we call this element the *fundamental element* of the presentation  $M$  and denote it by  $D$ . The fundamental element is crucial for the study of braid groups. One of its most important properties is that a certain power  $D^N$  of  $D$  generates the centre of  $B$ . ( $N = 2$  for the Artin presentation and  $N = n$  for the BKL presentation.) Moreover,  $u \preceq D^k$  is equivalent to  $D^k \succeq u$  and  $D^k \preceq u$  is equivalent to  $u \succeq D^k$  for all  $k \in \mathbf{Z}, u \in B$ .

In MAGMA, the partial ordering  $\preceq$  is provided as operator `le` and the partial ordering  $\succeq$  is provided as operator `ge`; see Section 73.4.4. For a description of the functions computing lcm and gcd of elements, see Section 73.4.5.

The positive elements  $c$  of  $B$  satisfying  $c \preceq D$  are called *simple elements*; we denote the set of simple elements by  $C$ . Simple elements can be uniquely described by permutations on  $n$  points. In MAGMA, a simple element  $c$  inducing a permutation  $\pi$  on the strings on which  $B$  is defined, is represented by the permutation  $\pi^{-1}$ .

If  $M$  is the Artin presentation, every permutation on  $n$  points corresponds to a simple element, that is,  $|C| = n!$ .

If  $M$  is the BKL presentation,  $|C| = (2n)!/(n!(n+1)!)$  and only permutations on  $n$  points which are products of parallel, descending cycles correspond to simple elements. Here, a cycle  $(i_1, \dots, i_r)$  is called *descending* if  $i_1 > \dots > i_r$  and two descending cycles  $(i_1, \dots, i_r)$  and  $(j_1, \dots, j_s)$  are called *parallel* if  $(i_k - j_l)(i_k - j_{l'}) (i_{k'} - j_l)(i_{k'} - j_{l'}) > 0$  for all  $1 \leq k, k' \leq r$  and  $1 \leq l, l' \leq s$ . The descending cycle  $(i_1, \dots, i_r)$  corresponds to the element  $a_{i_1, i_2} a_{i_2, i_3} \cdots a_{i_{r-1}, i_r}$  of  $B$ . It is obvious from the defining relations that the simple elements defined by two parallel descending cycles commute.

Every element  $u$  of  $B$  can be written in the form  $u = D^l c_1 \cdots c_k$ , where  $l$  is a suitable integer and  $c_1, \dots, c_k$  are simple elements. We call representations of this form *simple element representations* or *canonical factor products* (CFP).

### 73.1.2 Representing Elements of a Braid Group

This section describes the ways in which elements of a braid group can be represented internally by MAGMA. From the user's point of view, this mainly affects input and printing of elements. This section is intended to be a concise overview; for a detailed description of functions and for examples we refer to Section 73.2, Section 73.3 and Section 73.4.1.

Since an element of a braid group  $B$  can be represented either as word in the generators or as product of simple elements (see Section 73.4.5) with respect to either the Artin presentation or the BKL presentation of  $B$ , there are four different ways of representing elements of  $B$ , which can be used for entering or printing elements and for computing with elements.

#### 73.1.2.1 Automatic Conversions

MAGMA can work with all the above representations and conversions are done automatically when necessary, for example, when multiplying an element defined as word in the Artin generators with an element given as product of simple elements for the BKL presentation. Hence, the user normally does not have to give too much thought about how elements are represented. It should be noted, however, that automatic conversions can affect performance and that in time critical situations, the best results in general are obtained if automatic conversions are avoided.

#### 73.1.2.2 Default Presentations

When creating a braid group  $B$  using the command `BraidGroup`, the user can specify whether the Artin presentation or the BKL presentation should be used as *default presentation* for  $B$ . Unless specified otherwise by the user, this presentation is used in all

subsequent operations with  $B$  or with elements of  $B$ . In particular, group operations and printing of elements are performed with respect to this presentation. It is possible to change the default presentation using the command `SetPresentation`. Certain commands accept a parameter `Presentation`, which can be used to perform that command with respect to a presentation other than the default presentation.

### 73.1.2.3 Representation Used for Group Operations

By default, group operations with elements of a braid group  $B$  are performed using representations of the elements as products of simple elements for the default presentation of  $B$ . Experienced users can change this behaviour using the command `SetForceCFP`. If this flag is set to `false`, arguments of a group operation are not automatically converted into CFP representation if both arguments are represented as words in the generators of the default presentation of  $B$ , but the operation is performed, if possible, using the word representations instead.

### 73.1.2.4 Printing of Elements

The default printing format for an element  $u$  of a braid group  $B$  is that both a representation of  $u$  as word in the generators of the default presentation of  $B$  and a representation of  $u$  as product of simple elements for the default presentation of  $B$  are printed.

Depending on the application, the user may wish to change the print format so that only one of the above representations of  $u$  is printed. This can be achieved using the command `SetElementPrintFormat`.

## 73.1.3 Normal Form for Elements of a Braid Group

This section briefly describes the normal form for elements of braid groups. For details we refer to [ECH<sup>+</sup>92] and [BKL98]. The MAGMA commands for computing normal forms are described in Section 73.4.2.

Let  $B$  be the braid group on  $n$  strings and fix a presentation  $M$  for  $B$ , either the Artin presentation or the BKL presentation. A product of simple elements  $D^l c_1 \cdots c_k$  is said to be in *left normal form* with respect to  $M$ , if  $c_1 \neq D$ ,  $c_k \neq 1$  and  $(c_i^{-1} D) \wedge_l c_{i+1} = 1$  for  $i = 1, \dots, k-1$ , where  $(c_i^{-1} D) \wedge_l c_{i+1}$  denotes the left-gcd of  $c_i^{-1} D$  and  $c_{i+1}$  with respect to the presentation  $M$ .

Similarly, we define  $c_1 \cdots c_k D^l$  to be in *right normal form* with respect to  $M$ , if  $c_k \neq D$ ,  $c_1 \neq 1$  and  $c_i \wedge_r (D c_{i+1}^{-1}) = 1$  for  $i = 1, \dots, k-1$ , where  $\wedge_r$  denotes right-gcd with respect to the presentation  $M$ .

It can be shown that the numbers of simple elements and the powers of  $D$  in the left and right normal forms of an element are equal, that is, if  $x \in B$  has left normal form  $D^l c_1 \cdots c_k$  and right normal form  $\bar{c}_1 \cdots \bar{c}_{k'} D^{l'}$  then  $k' = k$  and  $l' = l$ . In this situation we call  $l$  the *infimum* of  $x$ , denoted by  $\text{inf}(x)$ ,  $k$  the *canonical length* of  $x$ , denoted by  $\text{len}(x)$ , and  $l + k$  the *supremum* of  $x$ , denoted by  $\text{sup}(x)$ .  $l$  is the maximal integer  $d$  satisfying  $D^d \preceq x$  and  $l + k$  is the minimal integer  $d$  satisfying  $x \preceq D^d$ .

To bring a product  $D^l c_1 \cdots c_k$  of simple elements into left normal form, we proceed by induction, assuming that  $D^l c_1 \cdots c_{k-1}$  is in left normal form. For  $i = k-1, \dots, 1$  we now

compute  $d = (c_i^{-1}D) \wedge_l c_{i+1}$  and, if  $d \neq 1$ , replace  $c_i$  by  $c_i d$  and  $c_{i+1}$  by  $d^{-1}c_{i+1}$ . Finally, we delete trailing trivial simple elements and absorb simple elements equal to  $D$  into the leading power of  $D$ . The result can be shown to be in left normal form [ECH<sup>+</sup>92, BKL98].

Both the theoretical complexity of this algorithm and its performance in practice are determined by the gcd computations.

For the Artin presentation, the cost of computing the left-gcd of two simple elements is  $O(n \log n)$  [ECH<sup>+</sup>92], whence the complexity of bringing a product of simple elements as above into left normal form is  $O(k^2 n \log n)$ .

For the Artin presentation, the cost of computing the left-gcd of two simple elements is  $O(n)$  [BKL98], whence the complexity of bringing a product of simple elements as above into left normal form is  $O(k^2 n)$ .

Computing right normal forms is analogous.

### 73.1.4 Mixed Canonical Form and Lattice Operations

This section outlines the algorithms used for lattice operations in a braid group. Let  $u$  and  $v$  be elements of a braid group  $B$  and let  $M$  be either the Artin presentation or the BKL presentation of  $B$ . The MAGMA commands for computing mixed canonical forms are described in Section 73.4.2 and the commands providing lattice operations are described in Section 73.4.5.

Evaluating partial orderings for  $u$  and  $v$  with respect to  $M$  is straightforward.  $u \preceq v$  if and only if  $u^{-1}v$  is a positive element with respect to  $M$ . The latter can be decided by computing the left normal form  $D^l c_1 \cdots c_k$  of  $u^{-1}v$  with respect to  $M$ :  $u^{-1}v$  is positive if and only if  $l \geq 0$ . Evaluating the partial ordering  $\succeq$  is analogous.

We call the tuple  $\langle a, b \rangle$  the *left-mixed canonical form* of an element  $x \in B$ , if  $a = a_1 \cdots a_k$  and  $b = b_1 \cdots b_s$  are positive elements in left normal form ( $a_1 = D$ ,  $b_1 = D$  is permitted),  $x = a^{-1}b$  and the left-gcd of  $a_1$  and  $b_1$  is trivial.

Similarly, we call the tuple  $\langle a, b \rangle$  the *right-mixed canonical form* of  $x$ , if  $a = a_1 \cdots a_k$  and  $b = b_1 \cdots b_s$  are positive elements in right normal form ( $a_k = D$ ,  $b_s = D$  is permitted),  $x = ab^{-1}$  and the right-gcd of  $a_k$  and  $b_s$  is trivial.

It is not difficult to show that the left-gcd of  $u$  and  $v$  is given by  $ua^{-1}$ , where  $\langle a, b \rangle$  is the left-mixed canonical form of  $u^{-1}v$ , and that the right-gcd of  $u$  and  $v$  is given by  $a^{-1}u$ , where  $\langle a, b \rangle$  is the right-mixed canonical form of  $uv^{-1}$  [ECH<sup>+</sup>92].

Similarly, the left-lcm of  $u$  and  $v$  is given by  $ua$ , where  $\langle a, b \rangle$  is the right-mixed canonical form of  $u^{-1}v$  and the right-lcm of  $u$  and  $v$  is given by  $au$ , where  $\langle a, b \rangle$  is the left-mixed canonical form of  $uv^{-1}$ .

Computing the left-mixed canonical form of an element  $x$  can, after writing  $x = a^{-1}b$  with two positive elements  $a$  and  $b$ , easily be reduced to computing repeatedly the left-normal forms of  $a$  and  $b$  and cancelling the left-gcd of the leading simple elements. Computing the right-mixed canonical form is analogous.

### 73.1.5 Conjugacy Testing and Conjugacy Search

Conjugacy testing, that is, deciding whether two given braids are conjugate, and conjugacy search, that is, computing a conjugating element for a pair of conjugate braids, are of particular importance to public key cryptosystems based on braid groups. Known algorithms for both conjugacy testing and conjugacy search require the (at least partial) computation of an invariant of the conjugacy classes of the elements in question, either the *super summit set* [Gar69, ERM94] or the *ultra summit set* [Geb03].

This section recalls the definition of these invariants and sketches the algorithms used for computing them, for conjugacy testing and for conjugacy search. The relevant MAGMA commands are described in Section 73.4.6.

For this section let  $B$  be a braid group and let  $M$  be either the Artin presentation or the BKL presentation of  $B$ .

#### 73.1.5.1 Definition of the Class Invariants

We define two operations, the *cycling* operation  $\mathbf{c}$  and the *decycling* operation  $\mathbf{d}$ , each mapping an arbitrary element  $x \in B$  to a conjugate of  $x$  as follows. Let  $x \in B$  be a braid with left normal form  $x = D^l c_1 \cdots c_k$  as defined in Section 73.1.3. If  $k = 0$ , we define  $\mathbf{c}(x) = x$  and  $\mathbf{d}(x) = x$ . Otherwise, we define  $\mathbf{c}(x) = D^l c_2 \cdots c_k (c_1^{D^{-l}})$  and  $\mathbf{d}(x) = D^l (c_k^{D^l}) c_1 \cdots c_{k-1}$ .

We now fix an element  $x \in B$  and consider the set  $C_x$  of all conjugates of  $x$ . Proofs for the following facts can be found in [ECH<sup>+</sup>92] or [BKL98].

- The set  $\{\inf(y) : y \in C_x\}$  is bounded above; we denote its maximum by  $\text{ss-inf}(x)$ .
- The set  $\{\sup(y) : y \in C_x\}$  is bounded below; we denote its minimum by  $\text{ss-sup}(x)$ .
- The maximum of  $\inf$  on  $C_x$  and the minimum of  $\sup$  on  $C_x$  can be achieved simultaneously.

We define three sets of conjugates of  $x$  as follows.

- The set  $P_x = \{y \in C_x : y \in B^+\}$ , containing the positive conjugates of  $x$ .
- The set  $S_x = \{y \in C_x : \inf(y) = \text{ss-inf}(x), \sup(y) = \text{ss-sup}(x)\}$ , called the *super summit set* of  $x$ .
- The set  $U_x = \{y \in S_x : \exists i > 0 : \mathbf{c}^i(y) = y\}$ , called the *ultra summit set* of  $x$ .

Clearly, the sets  $P_x$ ,  $S_x$  and  $U_x$  only depend on the conjugacy class of  $x$ . Moreover, the set  $P_x$  is empty if  $\text{ss-inf}(x) < 0$  and it contains  $S_x$  if  $\text{ss-inf}(x) \geq 0$ .

Proofs of the following properties can be found in [ECH<sup>+</sup>92] and [BKL98] for the sets  $P_x$  and  $S_x$  and in [Geb03] for the set  $U_x$ .

- The sets  $P_x$ ,  $S_x$  and  $U_x$  are finite.
- The sets  $S_x$  and  $U_x$  are non-empty.
- Representatives of  $P_x$ ,  $S_x$  and  $U_x$ , respectively, can be obtained from  $x$  by a finite number of cycling and decycling operations.

### 73.1.5.2 Computing the Class Invariants

The main tools for computing the class invariants introduced in Section 73.1.5.1 are the following “convexity” results established in [ERM94] and [FGM03] for the sets  $P_x$  and  $S_x$  and in [Geb03] for the set  $U_x$ . Let  $I_x \in \{P_x, S_x, U_x\}$ .

- For  $y, z \in I_x$ , there exists a finite sequence  $y = y_0, \dots, y_r = z$  such that for  $i = 1, \dots, r$ ,  $y_i \in I_x$  and  $y_i = y_{i-1}^{c_i}$  for a simple element  $c_i$ .
- For  $y \in I_x$  and a simple element  $c$ , there exists a unique  $\preceq$ -minimal element  $\iota_y(c)$  such that  $c \preceq \iota_y(c)$  and  $y^{\iota_y(c)} \in I_x$ . Moreover,  $\iota_y(c)$  is simple.

By the above results, any non-empty subset  $I \subseteq I_x$  with the property that  $y^{\iota_y(s)} \in I$  for all  $y \in I$  and all generators  $s$  of the presentation  $M$  is equal to  $I_x$ . In particular,  $I_x$  can be computed, starting from a single representative, as closure under conjugation with minimal simple elements.

Algorithms for computing the minimal simple elements  $\iota_y(c)$  are given in [FGM03] for the case  $I_x \in \{P_x, S_x\}$  and in [Geb03] for the case  $I_x = U_x$ .

The MAGMA commands for computing the class invariants  $P_x$ ,  $S_x$  and  $U_x$  as well as corresponding minimal simple elements  $\iota_y(c)$  are described in Section 73.4.6.

### 73.1.5.3 Conjugacy Testing and Conjugacy Search

Testing conjugacy of two braids  $x, y \in B$  can be performed using either super summit sets or ultra summit sets. It is obvious from the results cited in Section 73.1.5.1 that the following are equivalent.

- $x$  and  $y$  are conjugate in  $B$ .
- $S_x = S_y$ .
- $U_x = U_y$ .
- $S_x \cap S_y \neq \emptyset$ .
- $U_x \cap U_y \neq \emptyset$ .

If  $x$  and  $y$  are conjugate, a conjugating element can be obtained by establishing an element  $z \in S_x \cap S_y$  or  $z \in U_x \cap U_y$  both as conjugate of  $x$  and of  $y$  and keeping track of the conjugating elements in each step.

The size of super summit sets grows rapidly with increasing values of braid index  $n$  and canonical length. In general, computing super summit sets is difficult or infeasible for braids on more than 5-10 strings, except for very short canonical lengths. Ultra summit sets, on the other hand tend to be much smaller and can frequently be computed for braids on up to 100 strings and canonical length up to 1000, provided sufficient memory is available [Geb03]. Conjugacy search may be successful even in situations where the entire class invariant is too large to be computed.

In MAGMA, conjugacy testing and conjugacy search based on ultra summit sets is provided by the function `IsConjugate`.

## 73.2 Constructing and Accessing Braid Groups

This section describes the facilities for creating a braid group and for accessing and changing its basic properties.

**BraidGroup**(*n*: *parameters*)

**BraidGroup**(*GrpBrd*, *n*: *parameters*)

Given a small integer  $n > 0$ , return the braid group on  $n$  strings, that is,  $n - 1$  Artin generators.

**Presentation**

MONSTGELT

*Default* : “Artin”

The presentation can be selected using the parameter **Presentation**. Possible values for this parameter are "Artin" (default) and "BKL".

**GetPresentation**(*B*)

Returns a string  $s$  indicating the presentation currently used for  $B$ .  $s$  is either "Artin" or "BKL".

**SetPresentation**( $\sim B$ , *s*)

Set the presentation used for  $B$  to the presentation indicated by  $s$ . Possible values for  $s$  are "Artin" and "BKL".

**GetForceCFP**(*B*)

Returns whether arithmetic operations with elements of  $B$  are always done using representations of the elements as products of simple elements.

**SetForceCFP**( $\sim B$ , *b*)

By default, arithmetic operations with elements of  $B$  are always done using representations of the elements as products of simple elements. If necessary, such representations are computed.

Experienced users can turn this feature off for a braid group  $B$  using the procedure **SetForceCFP** with  $b$  set to **false**.

**GetElementPrintFormat**(*B*)

Returns a string  $s$  indicating the format currently used for printing elements of  $B$ .  $s$  is one of "Word", "CFP" or "Both".

**SetElementPrintFormat**( $\sim B$ , *s*)

When printing an element of a braid group  $B$ , by default both the representation as word in the generators and the representation as product of simple elements in the presentation selected for  $B$  are printed.

Experienced users can use the procedure **SetElementPrintFormat** for changing the print format. Possible values for  $s$  are "Word", "CFP" and "Both".

`NumberOfStrings(B)`

Given a braid group  $B$ , return the number of strings on which  $B$  is defined.

`NumberOfGenerators(B)`

`Ngens(B)`

Given a braid group  $B$ , return the number of Artin generators of  $B$ . Note that the number of Artin generators is returned regardless of the presentation selected for  $B$ .

### 73.3 Creating Elements of a Braid Group

This section describes the facilities for creating elements of a braid group.

`Representative(B)`

`Rep(B)`

Given a braid group  $B$ , return a representative of  $B$ .

`Identity(B)`

`Id(B)`

`B ! 1`

Given a braid group  $B$ , return the identity element of  $B$ .

`FundamentalElement(B: parameters)`

**Presentation**

MONSTGELT

*Default :*

Return the fundamental element for the presentation of  $B$  indicated by the parameter **Presentation**. Possible values for this parameter are "Artin" and "BKL". If the parameter **Presentation** is not used, the fundamental element for the presentation currently selected for  $B$  is returned.

`Generators(B: parameters)`

**Presentation**

MONSTGELT

*Default :*

Return a sequence containing the generators for the presentation of  $B$  indicated by the parameter **Presentation**. Possible values for this parameter are "Artin" and "BKL". If the parameter **Presentation** is not used, a sequence containing the generators for the presentation currently selected for  $B$  is returned.

`B . i`

Given a braid group  $B$  on  $n$  strings and an integer  $i$ , where  $0 < |i| < n$ , return the  $|i|$ -th Artin generator  $\sigma_i$ , if  $i > 0$ , or its inverse  $\sigma_{|i|}^{-1}$ , if  $i < 0$ .

**B . T**

Given a braid group  $B$  on  $n$  strings and an tuple  $T = \langle r, t \rangle$ , where  $1 \leq |t| < |r| \leq n$ , return the BKL generator  $a_{|r|,|t|}$ , if  $r, t > 0$ , or its inverse  $a_{|r|,|t|}^{-1}$  otherwise.

**B ! [ i<sub>1</sub>, ..., i<sub>k</sub> ]**

Given a braid group  $B$  on  $n$  strings and a sequence  $[i_1, \dots, i_k]$  of integers satisfying  $0 < |i_j| < n$  ( $j = 1, \dots, k$ ), return the element of  $B$  given by the product

$$\sigma_{|i_1|}^{\text{sgn}(i_1)} \dots \sigma_{|i_k|}^{\text{sgn}(i_k)}.$$

**B ! [ T<sub>1</sub>, ..., T<sub>k</sub> ]**

Given a braid group  $B$  on  $n$  strings and a sequence  $[T_1, \dots, T_k]$  of tuples satisfying  $T_j = \langle r_j, t_j \rangle$ ,  $1 \leq |t_j| < |r_j| \leq n$  ( $j = 1, \dots, k$ ), return the element of  $B$  given by the product

$$a_{|r_1|,|t_1|}^{e_1} \dots a_{|r_k|,|t_k|}^{e_k}$$

where  $e_j = 1$  if  $r_j, t_j > 0$  and  $e_j = -1$  otherwise ( $j = 1, \dots, k$ ).

**B ! p**

Given a braid group  $B$  on  $n$  strings and a permutation  $p$  on  $n$  points, return the simple element defined by  $p$  in the presentation currently selected for  $B$  as new element of  $B$ .

Note that the result in general depends on the presentation selected for  $B$ . Note further that in the BKL presentation, only permutations which are products of parallel descending cycles correspond to simple elements; attempting to coerce an invalid permutation will result in a runtime error. The function `IsProductOfParallelDescendingCycles` can be used to test whether a given permutation corresponds to a BKL simple element.

**B ! [ p<sub>1</sub>, ..., p<sub>k</sub> ]**

Given a braid group  $B$  on  $n$  strings and a sequence  $[p_1, \dots, p_k]$  of permutations on  $n$  points, return the product  $c_1 \dots c_k$  as new element of  $B$ , where  $c_j$  is the simple element defined by  $p_j$  in the presentation currently selected for  $B$  ( $j = 1, \dots, k$ ).

Note that the result in general depends on the presentation selected for  $B$ . Note further that in the BKL presentation, only permutations which are products of parallel descending cycles correspond to simple elements; attempting to coerce a sequence containing an invalid permutation will result in a runtime error. The function `IsProductOfParallelDescendingCycles` can be used to test whether a given permutation corresponds to a BKL simple element.

**B ! T**

Given a braid group  $B$  on  $n$  strings and an tuple  $T = \langle s, l, S, r \rangle$ , where  $s$  is either the string "Artin" or the string "BKL",  $l$  and  $r$  are integers and  $S$  is a sequence  $[p_1, \dots, p_k]$  of permutations on  $n$  points, return the product  $D^l c_1 \cdots c_k D^r$  as new element of  $B$ , where  $D$  is the fundamental element and  $c_j$  is the simple element defined by  $p_j$  ( $j = 1, \dots, k$ ) in the presentation indicated by  $s$ .

Note that in the BKL presentation, only permutations which are products of parallel descending cycles correspond to simple elements; if  $S$  contains an invalid permutation, a runtime error will result. Whether the elements of a given sequence  $S$  correspond to BKL simple elements can be tested using the function `IsProductOfParallelDescendingCycles`.

**IsProductOfParallelDescendingCycles(p)**

Given a permutation  $p$  on  $n$  points, return whether  $p$  is a product of parallel descending cycles, that is, whether  $p$  defines a simple element in the BKL monoid on  $n$  strings.

**Random(B, r, s, m, n: parameters)****RandomCFP(B, r, s, m, n: parameters)****Random(B: parameters)****RandomCFP(B: parameters)****Presentation**

MONSTGELT

*Default :*

Given a braid group  $B$  and integers  $r, s, m, n$ , satisfying  $r \leq s$  and  $0 \leq m \leq n$ , a pseudo-random element of  $B$  is constructed as follows. Let  $D$  be the fundamental element and  $C$  the set of simple elements for the presentation indicated by the parameter **Presentation**. First, integers  $e \in [r, s]$  and  $l \in [m, n]$  are chosen using uniform distributions on these sets. Then, for  $i = 1, \dots, l$ ,  $c_i \in C$  is chosen using a uniform distribution on  $C$  and the element  $D^e c_1 \cdots c_l$  is returned.

If no value is given for the parameter **Presentation**, the presentation selected for  $B$  is used.

The versions with a single argument are short for `Random(B, 0, 0, 0, 42)`.

**Random(B, m, n: parameters)****RandomWord(B, m, n: parameters)****RandomWord(B: parameters)****Presentation**

MONSTGELT

*Default :*

Given a braid group  $B$  and two integers  $0 \leq m \leq n$ , `Random(B, m, n)` returns a pseudo-random element of  $B$  constructed as follows. First, a length  $l \in [m, n]$  is chosen using a uniform distribution. Then, for  $i = 1, \dots, l$ ,  $g_i \in X \cup X^{-1} \setminus \{g_{i-1}^{-1}\}$  is chosen using a uniform distribution on this set. Here,  $X$  is the set of generators of the presentation indicated by the parameter **Presentation** and  $X^{-1}$  is the set of generator inverses.

If no value is given for the parameter `Presentation`, the presentation selected for  $B$  is used.

The signature `RandomWord(B)` is short for `RandomWord(B, 0, 42)`.

### Example H73E1

---

We construct the braid group  $B$  on 6 strings and the symmetric group  $S$  on 6 points.

```
> S := Sym(6);
> B := BraidGroup(6);
> B;
GrpBrd : B on 6 strings
```

By default,  $B$  is created using the Artin presentation.

```
> GetPresentation(B);
Artin
```

We now define the fundamental element with respect to the BKL presentation of  $B$  and print this element with respect to the presentation currently used for  $B$ , that is, with respect to the Artin presentation. Note that both a word in the Artin generators and a representation of the element in terms of Artin simple elements are printed.

```
> D_BKL := FundamentalElement(B : Presentation := "BKL");
> D_BKL;
B.5 * B.4 * B.3 * B.2 * B.1
<Artin, 0, [
  (1, 6, 5, 4, 3, 2)
], 0>
> GetElementPrintFormat(B);
Both
```

We print the BKL generator  $a_{3,1}$ .

```
> B.<3,1>;
B.2 * B.1 * B.2^-1
<Artin, 0, [
  (1, 3, 2),
  (1, 6)(2, 5, 3, 4)
], -1>
```

Next we change the format for printing elements of  $B$  using the function `SetElementPrintFormat` so that only a representation in terms of simple elements is printed.

```
> SetElementPrintFormat(~B, "CFP");
```

We now define and print several elements of  $B$ , illustrating the use of some of the functions described in the previous section.

First we create a pseudo-random element of  $B$  as product of 3 random simple elements for the Artin presentation.

```
> u := Random(B, 0, 0, 3, 3);
> u;
```

```
<Artin, 0, [
  (1, 6)(3, 5, 4),
  (1, 3)(2, 6)(4, 5),
  (2, 3)
], 0>
```

Next we define an element of  $B$  by a product of simple elements for the BKL presentation using the coercion operator '!'. Note that printing of this element is still done with respect to the Artin presentation.

```
> v := B ! <"BKL", 0, [ S | (1,6)(3,5,4), (1,3)(4,5)], 0>;
> v;
<Artin, 0, [
  (1, 6, 5, 4, 3, 2),
  (1, 2, 6)(3, 5),
  (2, 4, 5, 6),
  (1, 6)(2, 4, 3, 5)
], -2>
```

Finally, we look at the simple elements defined by the permutations  $p = (1, 3)(4, 2)$  and  $q = (1, 4, 3)$  on 6 points.

```
> p := S ! (1,3)(4,2);
> q := S ! (1,4,3);
```

Creating the simple elements for the Artin presentation defined by  $p$  and  $q$  is straightforward using the coercion operator '!'.

```
> p_Artin := B!p;
> p_Artin;
<Artin, 0, [
  (1, 3)(2, 4)
], 0>

> q_Artin := B!q;
> q_Artin;
<Artin, 0, [
  (1, 4, 3)
], 0>
```

We now change the presentation used for  $B$  to the BKL presentation. Note that this also changes the presentation with respect to which elements are printed.

```
> SetPresentation(~B, "BKL");
> GetPresentation(B);
BKL
```

The attempt to define a simple element for the BKL presentation using the permutation  $p$  fails.

```
> p_BKL := B!p;

>> p_BKL := B!p;
~
```

```
Runtime error in '!': Illegal coercion
LHS: GrpBrd
RHS: GrpPermElt
```

We should have been more careful: using the function `IsProductOfParallelDescendingCycles` we see that  $p$  is not a product of parallel descending cycles and hence does not define a simple element for the BKL presentation.

```
> IsProductOfParallelDescendingCycles(p);
false
```

$q$ , on the other hand, does define a simple element for the BKL presentation and we can coerce  $q$  to an element of  $B$  using the operator '!'.

```
> IsProductOfParallelDescendingCycles(q);
true
> q_BKL := B!q;
> q_BKL;
<BKL, 0, [
  (1, 4, 3)
], 0>
```

Note however, that the simple element for the BKL presentation defined by  $q$  and the simple element for the Artin presentation defined by  $q$  are different elements of  $B$ ! (The comparison operator `eq` is described in Section 73.4.4.)

```
> q_BKL eq q_Artin;
false
```

The representations of the Artin simple elements defined by  $p$  and  $q$  in terms of BKL simple elements have no obvious connection to  $p$  and  $q$ , respectively.

```
> p_Artin;
<BKL, 0, [
  (1, 3, 2),
  (2, 4, 3)
], 0>

> q_Artin;
<BKL, 0, [
  (1, 4, 3, 2),
  (2, 3)
], 0>
```

---

## 73.4 Working with Elements of a Braid Group

### 73.4.1 Accessing Information

This sections describes how the internal representations of an element and a number of basic invariants can be accessed.

`Parent(u)`

Given an element  $u$  of a braid group  $B$ , return the parent group of  $u$ , that is  $B$ .

`#u`

Given an element  $u$  of a braid group  $B$ , return the length of the representing word in the generators corresponding to the presentation selected for  $B$ . Note that this is not an invariant of  $u$ .

`CanonicalFactorRepresentation(u: parameters)`

`CFP(u: parameters)`

**Presentation**

MONSTGELT

*Default :*

Given an element  $u$  of a braid group  $B$ , return a tuple  $T = \langle s, l, S, r \rangle$  describing the representation in terms of simple elements for the presentation indicated by the value of the parameter **Presentation**. If no value for **Presentation** is given, the presentation selected for  $B$  is used.

The interpretation of the components of  $T$  is as follows:  $s$  is a string, either equal to "Artin" or equal to "BKL" indicating the presentation,  $l$  and  $r$  are integers and  $S$  is a sequence  $[p_1, \dots, p_k]$  of permutations on  $n$  points, such that

$$D^l c_1 \cdots c_k D^r$$

is the representation of  $u$  in terms of simple elements, where  $D$  is the fundamental element and  $c_j$  is the simple element defined by  $p_j$  ( $j = 1, \dots, k$ ) in the presentation indicated by  $s$ .

`WordToSequence(u: parameters)`

`ElementToSequence(u: parameters)`

`Eltseq(u: parameters)`

**Presentation**

MONSTGELT

*Default :*

Given an element  $u$  of a braid group  $B$ , return a sequence describing the representing word in the generators corresponding to the presentation indicated by the value of the parameter **Presentation**. If no value for **Presentation** is given, the presentation selected for  $B$  is used.

For a representing word

$$\sigma_{i_1}^{e_1} \cdots \sigma_{i_k}^{e_k}$$

in the Artin generators with  $0 < i_j < n$  and  $e_j \in \{-1, 1\}$  for  $j = 1, \dots, k$ , the sequence of integers

$$[e_1 i_1, \dots, e_k i_k]$$

is returned.

For a representing word

$$a_{r_1, t_1}^{e_1} \dots a_{r_k, t_k}^{e_k}$$

in the BKL generators with  $1 \leq t_j < r_j \leq n$  and  $e_j \in \{-1, 1\}$  for  $j = 1, \dots, k$ , the sequence of tuples

$$[\langle e_1 r_1, e_1 t_1 \rangle, \dots, \langle e_k r_k, e_k t_k \rangle]$$

is returned.

**InducedPermutation(u)**

Given an element  $u$  of a braid group  $B$  on  $n$  strings, return the permutation on  $n$  points induced by  $u$  acting on the strings on which  $B$  is defined.

For the following description of the functions **CanonicalLength**, **Infimum** and **Supremum**, let  $D$  be the fundamental element and let  $D^l c_1 \dots c_k$  be the left normal form of the element  $u$  of the braid group  $B$  in the presentation indicated by the value of the parameter **Presentation**. If no value for **Presentation** is given, the presentation selected for  $B$  is used.

**CanonicalLength(u: parameters)**

**Presentation** MONSTGELT *Default :*

Given an element  $u$  of a braid group  $B$ , return the canonical length  $k$  of  $u$  for the appropriate presentation of  $B$ . The argument is converted into left normal form.

**Infimum(u: parameters)**

**Presentation** MONSTGELT *Default :*

Given an element  $u$  of a braid group  $B$ , return the infimum  $l$  of  $u$  for the appropriate presentation of  $B$ . The argument is converted into left normal form.

**Supremum(u: parameters)**

**Presentation** MONSTGELT *Default :*

Given an element  $u$  of a braid group  $B$ , return the supremum  $l + k$  of  $u$  for the appropriate presentation of  $B$ . The argument is converted into left normal form.

For the following description of the functions **SuperSummitCanonicalLength**, **SuperSummitInfimum** and **SuperSummitSupremum**, let  $D$  be the fundamental element and let  $D^l c_1 \dots c_k$  be the normal form of a representative of the super summit set the element  $u$  of the braid group  $B$  with respect to the presentation indicated by the value of the parameter **Presentation**. If no value for **Presentation** is given, the presentation selected for  $B$  is used.

SuperSummitCanonicalLength(u: <i>parameters</i> )
---

Presentation

MONSTGELT

Default :

Given an element  $u$  of a braid group  $B$ , return the canonical length  $k$  of a representative of the super summit set of  $u$  with respect to the appropriate presentation of  $B$ , that is, the minimal canonical length among all conjugates of  $u$ . The argument is converted into left normal form.

SuperSummitInfimum(u: <i>parameters</i> )
---

Presentation

MONSTGELT

Default :

Given an element  $u$  of a braid group  $B$ , return the infimum  $l$  of a representative of the super summit set of  $u$  with respect to the appropriate presentation of  $B$ , that is, the maximal infimum among all conjugates of  $u$ . The argument is converted into left normal form.

SuperSummitSupremum(u: <i>parameters</i> )
--

Presentation

MONSTGELT

Default :

Given an element  $u$  of a braid group  $B$ , return the supremum  $l+k$  of a representative of the super summit set of  $u$  with respect to the appropriate presentation of  $B$ , that is, the minimal supremum among all conjugates of  $u$ . The argument is converted into left normal form.

### Example H73E2

---

We define the braid group  $B$  on 6 strings using the BKL presentation and create an element  $u$  as a random word of length between 5 and 10 in the BKL generators.

```
> B := BraidGroup(6 : Presentation := "BKL");
> u := RandomWord(B, 5, 10);
```

The parent group of  $u$  can be accessed using the function `Parent`.

```
> Parent(u);
GrpBrd : B on 6 strings
```

As word in the BKL generators,  $u$  has length 5. We define a sequence describing the representation of  $u$  as word in the BKL generators.

```
> #u;
5
> seq_BKL := WordToSequence(u);
> seq_BKL;
[ <5, 1>, <5, 2>, <5, 4>, <4, 1>, <2, 1> ]
```

When we ask for a representation as word in the Artin generators, such a representation is created automatically.

```
> seq_Artin := WordToSequence(u : Presentation := "Artin");
> seq_Artin;
```

```
[ 4, 3, 2, 1, 2, 1, -2, -3, 1 ]
```

We now define the permutation  $p$  induced by  $u$  on the strings on which  $B$  is defined.

```
> p := InducedPermutation(u);
> p;
(4, 5)
```

The representation of  $u$  in terms of simple elements for the Artin presentation can be obtained using the function `CanonicalFactorRepresentation`.

```
> CanonicalFactorRepresentation(u : Presentation := "Artin");
<Artin, 0, [
  (1, 5, 4, 3),
  (1, 2),
  (1, 6)(2, 5, 3),
  (5, 6)
], -1>
```

We now compute the canonical lengths of  $u$  with respect to the Artin presentation and with respect to the BKL presentation. Note that these lengths are different.

```
> CanonicalLength(u : Presentation := "Artin");
4
> CanonicalLength(u : Presentation := "BKL");
3
```

Finally, we compute for both presentations the canonical lengths of a super summit representative of  $u$ .

```
> SuperSummitCanonicalLength(u : Presentation := "Artin");
2
> SuperSummitCanonicalLength(u : Presentation := "BKL");
3
```

Obviously,  $u$  does not belong to its super summit set with respect to the Artin presentation. (We cannot tell for the BKL presentation from the information we have computed.)

### 73.4.2 Computing Normal Forms of Elements

This section describes functions and procedures for computing various normal forms of elements of a braid group  $B$ . All normal forms are defined in terms of representations of elements as products of simple elements and depend on the presentation of  $B$  which is used. The functions documented in this section all accept a parameter `Presentation` which can be used to specify the presentation of  $B$  with respect to which the computation should be performed. Possible values for this parameter are the strings "Artin" and "BKL". If no value is given for `Presentation`, the presentation selected for  $B$  is used.

LeftNormalForm(u: parameters)
-------------------------------

NormalForm(u: parameters)
---------------------------

Presentation                      MONSTGELT                      *Default :*

Given an element  $u$  of a braid group  $B$ , return a new element of  $B$  defined by the left normal form of  $u$  with respect to the indicated presentation.

LeftNormalForm(~u: parameters)
--------------------------------

NormalForm(~u: parameters)
----------------------------

Presentation                      MONSTGELT                      *Default :*

Given an element  $u$  of a braid group  $B$ , bring  $u$  into left normal form with respect to the indicated presentation.

RightNormalForm(u: parameters)
--------------------------------

Presentation                      MONSTGELT                      *Default :*

Given an element  $u$  of a braid group  $B$ , return a new element of  $B$  defined by the right normal form of  $u$  with respect to the indicated presentation.

RightNormalForm(~u: parameters)
---------------------------------

Presentation                      MONSTGELT                      *Default :*

Given an element  $u$  of a braid group  $B$ , bring  $u$  into right normal form with respect to the indicated presentation.

LeftMixedCanonicalForm(u: parameters)
---------------------------------------

MixedCanonicalForm(u: parameters)
-----------------------------------

Presentation                      MONSTGELT                      *Default :*

Given an element  $u$  of a braid group  $B$ , return two tuples  $T_1$  and  $T_2$  defining products  $v_1 \cdots v_k$  and  $w_1 \cdots w_l$ , respectively, of simple elements for the indicated presentation, such that  $v_1 \cdots v_k$  and  $w_1 \cdots w_l$  are in left normal form, the left-gcd of  $v_1$  and  $w_1$  is trivial and

$$u = (v_1 \cdots v_k)^{-1}(w_1 \cdots w_l).$$

See the entry for `CanonicalFactorRepresentation` for a description of the tuple format. Note that the tuples can be coerced into elements of  $B$  using the coercion operator '!'.

RightMixedCanonicalForm(u: parameters)
--

Presentation

MONSTGELT

Default :

Given an element  $u$  of a braid group  $B$ , return two tuples  $T_1$  and  $T_2$  defining products  $v_1 \cdots v_k$  and  $w_1 \cdots w_l$ , respectively, of simple elements for the indicated presentation, such that  $v_1 \cdots v_k$  and  $w_1 \cdots w_l$  are in right normal form, the right-gcd of  $v_1$  and  $w_1$  is trivial and

$$u = (v_1 \cdots v_k)(w_1 \cdots w_l)^{-1}.$$

See the entry for `CanonicalFactorRepresentation` for a description of the tuple format. Note that the tuples can be coerced into elements of  $B$  using the coercion operator '!'.  


---

**Example H73E3**

We define the braid group  $B$  on 6 strings using the Artin presentation, set the print format for elements to "CFP" and define an element  $u$  of  $B$ .

```
> B := BraidGroup(6);
> SetElementPrintFormat(~B, "CFP");
>
> u := B ! <"Artin",
>      0,
>      [ Sym(6) | (1,6)(3,5,4), (1,3)(2,6)(4,5), (2,3)],
>      0>;
> u;
<Artin, 0, [
  (1, 6)(3, 5, 4),
  (1, 3)(2, 6)(4, 5),
  (2, 3)
], 0>
```

We compute and print the left normal form of  $u$  with respect to the Artin presentation of  $B$ .

```
> u_Artin := LeftNormalForm(u);
> u_Artin;
<Artin, 1, [
  (1, 2, 6, 5, 4, 3),
  (2, 3)
], 0>
```

We now compute the left normal form of  $u$  with respect to the BKL presentation of  $B$ . Since elements are printed with respect to the presentation selected for the parent group, that is, in the Artin presentation in our example, we use the function `CanonicalFactorRepresentation` to print the representation in terms of simple elements for the BKL presentation.

```
> u_BKL := LeftNormalForm(u : Presentation := "BKL");
> CFP(u_BKL : Presentation := "BKL");
<BKL, 2, [
  (2, 6, 5, 4, 3),
```

```

      (2, 6, 5, 4),
      (2, 6, 5),
      (2, 6),
      (2, 3)
], 0>

```

We define another element  $v$  of  $B$  in left normal form.

```

> v := LeftNormalForm(B.5*B.2^-2*B.4*B.3^-1*B.5^-1*B.3^-1*B.5);
> v;
<Artin, -3, [
  (1, 6)(2, 4, 3, 5),
  (1, 2, 6)(3, 5),
  (1, 6, 2, 5)(3, 4),
  (3, 5)(4, 6)
], 0>

```

As can easily be read off the representation of  $v$  in terms of simple elements which is in left normal form,  $v$  has infimum -3, canonical length 4 and supremum 1 with respect to the Artin presentation.

```

> Infimum(v);
-3
> CanonicalLength(v);
4
> Supremum(v);
1

```

Note that infimum, canonical length and supremum of an element can also be obtained from its right normal form.

```

> RightNormalForm(v);
<Artin, 0, [
  (4, 6, 5),
  (1, 6)(2, 5, 3, 4),
  (1, 6)(2, 4, 5),
  (1, 6)(2, 5)
], -3>

```

### 73.4.3 Arithmetic Operators and Functions for Elements

This section describes the basic arithmetic operations for elements of a braid group. Strictly speaking, all functions should be considered as functions on *representatives* of elements, that is, words in the generators or products of simple elements.

Unless stated otherwise, arithmetic operations with elements of a braid group  $B$  are performed using representations with respect to the presentation selected for  $B$ . This presentation can be changed using the function `SetPresentation`.

By default, arithmetic operations with elements of  $B$  are performed using representations in terms of simple elements; such representations are created if necessary. Experienced users can change this behaviour, if desired, using the function `SetForceCFP`.

The complexity of all basic arithmetic operations is linear in the length of the representations of the input elements. No normalisations are performed automatically, as doing so would restrict the user's control of operations with elements. It is, however, recommended to use the function `NormalForm` or its procedural version in time critical situations to limit the length of representations of elements; see Example H73E4.

`u * v`

Given elements  $u$  and  $v$  belonging to the same braid group  $B$ , return the product  $uv$  as a new element of  $B$ .

`u *:= v`

Given elements  $u$  and  $v$  belonging to the same braid group  $B$ , replace  $u$  with the product  $uv$ .

`u / v`

Given elements  $u$  and  $v$  belonging to the same braid group  $B$ , return the product  $uv^{-1}$  as a new element of  $B$ .

`u /:= v`

Given elements  $u$  and  $v$  belonging to the same braid group  $B$ , replace  $u$  with the product  $uv^{-1}$ .

`u ^ n`

Given an element  $u$  of a braid group  $B$  and an integer  $n$ , return the power  $u^n$  as a new element of  $B$ .

`u ^:= n`

Given an element  $u$  of a braid group  $B$  and an integer  $n$ , replace  $u$  with the power  $u^n$ .

`u ^ v`

Given elements  $u$  and  $v$  belonging to the same braid group  $B$ , return the conjugate  $u^v = v^{-1}uv$  as a new element of  $B$ .

`u ^:= v`

Given elements  $u$  and  $v$  belonging to the same braid group  $B$ , replace  $u$  with the conjugate  $u^v = v^{-1}uv$ .

`Inverse(u)`

Given an element  $u$  of a braid group  $B$ , return its inverse  $u^{-1}$  as a new element of  $B$ .

`Inverse(~u)`

Given an element  $u$  of a braid group  $B$ , replace  $u$  with its inverse  $u^{-1}$ .

<code>LeftConjugate(u, v)</code>
----------------------------------

Given elements  $u$  and  $v$  belonging to the same braid group  $B$ , return the “left conjugate”  $vuv^{-1}$  as a new element of  $B$ .

<code>LeftConjugate(~u, v)</code>
-----------------------------------

Given elements  $u$  and  $v$  belonging to the same braid group  $B$ , replace  $u$  with the “left conjugate”  $vuv^{-1}$ .

<code>LeftDiv(u, v)</code>
----------------------------

Given elements  $u$  and  $v$  belonging to the same braid group  $B$ , return the product  $u^{-1}v$  as a new element of  $B$ .

<code>LeftDiv(u, ~v)</code>
-----------------------------

Given elements  $u$  and  $v$  belonging to the same braid group  $B$ , replace  $v$  with the product  $u^{-1}v$ .

The following functions `Cycle` and `Decycle` accept a parameter `Presentation` which can be set either to "Artin" or to "BKL". The results of the cycling and decycling operations are defined in terms of the left normal form  $D^l c_1 \cdots c_k$  of the argument  $u \in B$  in terms of simple elements for a presentation of  $B$  and the results in general depend on the presentation used. The results of these functions are returned in left normal form.

If no value for the parameter `Presentation` is given, the presentation selected for the parent group of the argument will be used.

<code>Cycle(u: parameters)</code>
-----------------------------------

`Presentation`

MONSTGELT

*Default :*

Given an element  $u \in B$  with left normal form  $D^l c_1 \cdots c_k$ , return the result of a cycling operation on  $u$ , that is,

$$u^{(c_1^{D^{-l}})}$$

as new element of  $B$  in left normal form.

<code>Cycle(~u: parameters)</code>
------------------------------------

`Presentation`

MONSTGELT

*Default :*

Given an element  $u \in B$  with left normal form  $D^l c_1 \cdots c_k$ , replace  $u$  by the result of a cycling operation on  $u$ , that is, by

$$u^{(c_1^{D^{-l}})}$$

in left normal form.

<code>Decycle(u: parameters)</code>
-------------------------------------

**Presentation**

MONSTGELT

*Default :*

Given an element  $u \in B$  with left normal form  $D^l c_1 \cdots c_k$ , return the result of a decycling operation on  $u$ , that is,

$$u^{(c_k^{-1})}$$

as new element of  $B$  in left normal form.

<code>Decycle(~u: parameters)</code>
--------------------------------------

**Presentation**

MONSTGELT

*Default :*

Given an element  $u \in B$  with left normal form  $D^l c_1 \cdots c_k$ , replace  $u$  by the result of a decycling operation on  $u$ , that is, by

$$u^{(c_k^{-1})}$$

in left normal form.

---

**Example H73E4**

We illustrate the importance of limiting the length of representations of elements using the function `NormalForm` when performing a sequence of arithmetic operations on an element.

Consider the following computation in the braid group  $B$  on 6 strings. Starting with an element  $w$ , we repeatedly replace  $w$  by the product  $ww^{\sigma_1}$  where  $\sigma_1$  is the first Artin generator of  $B$ .

A naive way of implementing this computation would be as follows.

```
> B := BraidGroup(6);
> u := B.5*B.2^-2*B.4*B.3^-1;
> v := B.1;
> N := 14;
>
> T := Cputime();
> w := u;
> for i := 1 to N do
>   w := w * w^v;
> end for;
```

This, however, yields an extremely complicated representation for the result; the representation in terms of simple elements has the length 114686.

```
> #CFP(w) [3];
114686
```

Performing subsequent computations with the result, for example computing its normal form, is very expensive.

```
> NormalForm(~w);
> print "total time used: ", Cputime()-T;
```

```
total time used: 149.229
```

One might be tempted to solve this problem by working only with elements in normal form, that is, by bringing every result of an arithmetic operation into normal form after computing it. Using this approach, computing the result of the above iteration is indeed much faster.

```
> T := Cputime();
> w := u;
> for i := 1 to N do
>   t := w^v;
>   NormalForm(~t);
>   w := w * t;
>   NormalForm(~w);
> end for;
> print "total time used: ", Cputime()-T;
total time used: 0.53
```

However, this strategy is not optimal either. For the above example, the optimal performance is obtained if the result is normalised every third pass through the iteration.

```
> T := Cputime();
> w := u;
> for i := 1 to N do
>   w := w * w^v;
>   if i mod 3 eq 0 then
>     NormalForm(~w);
>   end if;
> end for;
> NormalForm(~w);
> print "total time used: ", Cputime()-T;
total time used: 0.171
```

Unfortunately, the frequency of normalisation giving best results depends heavily on the situation, that is, both on the arithmetic operations and on the characteristics of the arguments.

As a rule of thumb, the effects of normalising results too frequently are less of a problem than normalising results not often enough or not at all.

#### 73.4.4 Boolean Predicates for Elements

This section describes the tests for membership, equality and partial orderings which are available for elements of a braid group  $B$ .

Unless stated otherwise, all computations are performed in the presentation selected for  $B$  or in the presentation specified by the value of the parameter `Presentation`, either "Artin" or "BKL" if a value for this parameter is given.

<code>u in B</code>
---------------------

Given an element  $u$  of a braid group and a braid group  $B$ , return `true` if  $u \in B$  and `false` otherwise.

**u notin B**

Given an element  $u$  of a braid group and a braid group  $B$ , return **false** if  $u \in B$  and **true** otherwise.

**IsEmptyWord(u: parameters)**

**Presentation** MONSTGELT *Default :*

Given an element  $u$  of a braid group, return **true** if  $u$  is the represented by the empty word in the specified presentation and **false** otherwise.

**AreIdentical(u, v: parameters)**

**Presentation** MONSTGELT *Default :*

Given elements  $u$  and  $v$  belonging to the same braid group  $B$ , return **true** if  $u$  and  $v$  are represented by identical words in the specified presentation and **false** otherwise.

**IsSimple(u: parameters)**

**Presentation** MONSTGELT *Default :*

Given an element  $u$  of a braid group, return **true** if  $u$  is a simple element with respect to the specified presentation and **false** otherwise. The argument is converted into normal form.

**IsSuperSummitRepresentative(u: parameters)**

**Presentation** MONSTGELT *Default :*

Given an element  $u$  of a braid group, return **true** if  $u$  is an element of its super summit set with respect to the specified presentation and **false** otherwise. The argument is converted into normal form.

**IsUltraSummitRepresentative(u: parameters)**

**Presentation** MONSTGELT *Default :*

Given an element  $u$  of a braid group, return **true** if  $u$  is an element of its ultra summit set with respect to the specified presentation and **false** otherwise. The argument is converted into normal form.

**IsIdentity(u: parameters)**

**IsId(u: parameters)**

**Presentation** MONSTGELT *Default :*

Given an element  $u$  of a braid group  $B$ , return **true** if  $u$  is the identity element of  $B$  and **false** otherwise. The argument is converted into normal form.

**u eq v**

Given elements  $u$  and  $v$  belonging to the same braid group  $B$ , return **true** if  $u = v$  and **false** otherwise. Both arguments are converted into normal form.

<code>u ne v</code>
---------------------

Given elements  $u$  and  $v$  belonging to the same braid group  $B$ , return **false** if  $u = v$  and **true** otherwise. Both arguments are converted into normal form.

<code>u le v</code>
---------------------

<code>IsLE(u, v: parameters)</code>
-------------------------------------

<code>IsLe(u, v: parameters)</code>
-------------------------------------

**Presentation**

MONSTGELT

*Default :*

Given elements  $u$  and  $v$  belonging to the same braid group  $B$ , return **true** if  $u \preceq v$ , that is, if  $u^{-1}v$  is a positive element, with respect to the specified presentation and **false** otherwise.

Note that the parameter **Presentation** is not available for the operator version of this predicate.

<code>u ge v</code>
---------------------

<code>IsGE(u, v: parameters)</code>
-------------------------------------

<code>IsGe(u, v: parameters)</code>
-------------------------------------

**Presentation**

MONSTGELT

*Default :*

Given elements  $u$  and  $v$  belonging to the same braid group  $B$ , return **true** if  $u \succeq v$ , that is, if  $uv^{-1}$  is a positive element, with respect to the specified presentation and **false** otherwise.

Note that the parameter **Presentation** is not available for the operator version of this predicate.

<code>IsConjugate(u, v: parameters)</code>
--

**Presentation**

MONSTGELT

*Default :*

Given elements  $u$  and  $v$  belonging to the same braid group  $B$ , return **true** and an element  $c \in B$  satisfying  $u^c = v$  if  $u$  and  $v$  are conjugate and return **false** otherwise.

The function first computes representatives  $u_s$  and  $v_s$  of the ultra summit sets of  $u$  and  $v$ , respectively, with respect to the specified presentation. If this does not prove that the elements are not conjugate, the function tries to compute elements of the ultra summit set of  $u$  until either the element  $v_s$  is found, proving that  $u$  and  $v$  are conjugate, or the ultra summit set of  $u$  is seen not to contain  $v_s$ , proving that  $u$  and  $v$  are not conjugate. See Example H73E8 for a more detailed description.

Note that testing elements for conjugacy is a hard problem and may require significant amounts of memory and CPU time.

**Example H73E5**

---

We define the braid group  $B$  on 6 strings using the Artin presentation and set the print format for elements to "CFP".

```
> B:= BraidGroup(6);
> SetElementPrintFormat(~B, "CFP");
```

(1) We create pseudo-random elements of  $B$  until we find an element  $u$  which is contained in its super summit set with respect to the Artin presentation of  $B$ .

```
> repeat
>   u := Random(B, 5, 10);
> until IsSuperSummitRepresentative(u);
> NormalForm(u);
<Artin, -2, [
  (1, 5)(2, 3, 6),
  (1, 5, 6, 3, 2, 4),
  (2, 6)(3, 4, 5)
], 0>
```

$u$  is not contained in its super summit set with respect to the BKL presentation of  $B$ , showing that the super summit set of an element in general depends on the presentation with respect to which it is defined.

```
> IsSuperSummitRepresentative(u : Presentation := "Artin");
true
> IsSuperSummitRepresentative(u : Presentation := "BKL");
false
```

(2) This example shows that the Artin presentation and the BKL presentation give rise to distinct partial orderings on  $B$ .

$\sigma_1^{-1}\sigma_2\sigma_1$  has negative infimum with respect to the Artin presentation and hence is not a positive element with respect to this presentation.

```
> Infimum(B.1^-1*B.2*B.1 : Presentation := "Artin");
-1
```

Consequently,  $\sigma_1 \not\leq \sigma_2\sigma_1$  in the partial ordering defined with respect to the Artin presentation.

```
> B.1 le B.2*B.1;
false
```

We can also use the function version to check this.

```
> IsLE(B.1, B.2*B.1 : Presentation := "Artin");
false
```

However,  $\sigma_1^{-1}\sigma_2\sigma_1$  is equal to the BKL generator  $a_{3,1}$  and hence is, in particular, a positive element with respect to the BKL presentation. in the BKL generators.

```
> B.1^-1*B.2*B.1 eq B.<3,1>;
```

```
true
```

Hence,  $\sigma_1 \preceq \sigma_2\sigma_1$  in the partial ordering defined with respect to the BKL presentation.

```
> IsLE(B.1, B.2*B.1 : Presentation := "BKL");
true
```

(3) We change the print format for elements of  $B$  so that only words in the Artin generators are printed.

```
> SetElementPrintFormat(~B, "Word");
```

Inducing permutations with different cycle structure,  $\sigma_1$  and  $\sigma_1\sigma_2$  cannot be conjugate in  $B$ .

```
> InducedPermutation(B.1);
(1, 2)
> InducedPermutation(B.2*B.1);
(1, 2, 3)
> IsConjugate(B.1, B.2*B.1);
false
```

$\sigma_1$  and  $\sigma_2$ , however, are conjugate in  $B$ . We compute a conjugating element  $c$ .

```
> res, c := IsConjugate(B.1, B.2);
> res;
true
> NormalForm(c);
B.2 * B.1
```

$c$ , as desired, conjugates  $\sigma_1: s_1:$  to  $\sigma_2: s_2:$ .

```
> B.1^c eq B.2;
true
```

### 73.4.5 Lattice Operations

This section describes the functions available for computing lattice operations, least common multiple and greatest common divisor, for elements of a braid group  $B$ . The results of all lattice operations depend on the presentation used for  $B$  and on the partial ordering considered.

The functions documented in this section all accept a parameter **Presentation** which can be used to specify the presentation of  $B$  with respect to which the computation should be performed. Possible values for this parameter are the strings "Artin" and "BKL". If no value is given for **Presentation**, the presentation selected for  $B$  is used.

LeftGCD( $u, v$ : parameters)
-------------------------------

LeftGcd( $u, v$ : parameters)
-------------------------------

LeftGreatestCommonDivisor( $u, v$ : parameters)
---

GCD( $u, v$ : parameters)
---------------------------

Gcd( $u, v$ : parameters)
---------------------------

GreatestCommonDivisor( $u, v$ : parameters)
---

Presentation

MONSTGELT

Default :

Given elements  $u$  and  $v$  belonging to the same braid group  $B$ , return the left-gcd of  $u$  and  $v$ , that is, the with respect to  $\preceq$  maximal element  $d$  of  $B$  satisfying  $d \preceq u$  and  $d \preceq v$ . Here,  $\preceq$  is the partial ordering on  $B$  defined as follows:  $a \preceq b$  iff  $a^{-1}b$  is representable as a positive word in the specified presentation of  $B$ .

RightGCD( $u, v$ : parameters)
--------------------------------

RightGcd( $u, v$ : parameters)
--------------------------------

RightGreatestCommonDivisor( $u, v$ : parameters)
--

Presentation

MONSTGELT

Default :

Given elements  $u$  and  $v$  belonging to the same braid group  $B$ , return the right-gcd of  $u$  and  $v$ , that is, the with respect to  $\succeq$  maximal element  $d$  of  $B$  satisfying  $u \succeq d$  and  $v \succeq d$ . Here,  $\succeq$  is the partial ordering on  $B$  defined as follows:  $a \succeq b$  iff  $ab^{-1}$  is representable as a positive word in the specified presentation of  $B$ .

LeftGCD( $S$ : parameters)
----------------------------

LeftGcd( $S$ : parameters)
----------------------------

LeftGreatestCommonDivisor( $S$ : parameters)
--

GCD( $S$ : parameters)
------------------------

Gcd( $S$ : parameters)
------------------------

GreatestCommonDivisor( $S$ : parameters)
--

Presentation

MONSTGELT

Default :

Given a set or a sequence  $S$  containing elements of a braid group  $B$ , return the left-gcd of the elements of  $S$ , that is, the with respect to  $\preceq$  maximal element  $d$  of  $B$  satisfying  $d \preceq s$  for all  $s \in S$ , where  $\preceq$  is defined as above.

RightGCD( $S$ : parameters)
-----------------------------

RightGcd( $S$ : parameters)
-----------------------------

RightGreatestCommonDivisor( $S$ : parameters)
---

Presentation

MONSTGELT

Default :

Given a set or a sequence  $S$  containing elements of a braid group  $B$ , return the right-gcd of the elements of  $S$ , that is, the with respect to  $\succeq$  maximal element  $d$  of  $B$  satisfying  $s \succeq d$  for all  $s \in S$ , where  $\succeq$  is defined as above.

LeftLCM( $u, v$ : parameters)
-------------------------------

LeftLcm( $u, v$ : parameters)
-------------------------------

LeftLeastCommonMultiple( $u, v$ : parameters)
---

LCM( $u, v$ : parameters)
---------------------------

Lcm( $u, v$ : parameters)
---------------------------

LeastCommonMultiple( $u, v$ : parameters)
---

Presentation

MONSTGELT

Default :

Given elements  $u$  and  $v$  belonging to the same braid group  $B$ , return the left-lcm of  $u$  and  $v$ , that is, the with respect to  $\preceq$  minimal element  $d$  of  $B$  satisfying  $u \preceq d$  and  $v \preceq d$ , where  $\preceq$  is defined as above.

RightLCM( $u, v$ : parameters)
--------------------------------

RightLcm( $u, v$ : parameters)
--------------------------------

RightLeastCommonMultiple( $u, v$ : parameters)
--

Presentation

MONSTGELT

Default :

Given elements  $u$  and  $v$  belonging to the same braid group  $B$ , return the right-lcm of  $u$  and  $v$ , that is, the with respect to  $\succeq$  minimal element  $d$  of  $B$  satisfying  $d \succeq u$  and  $d \succeq v$ , where  $\succeq$  is defined as above.

LeftLCM( $S$ : parameters)
----------------------------

LeftLcm( $S$ : parameters)
----------------------------

LeftLeastCommonMultiple( $S$ : parameters)
--

LCM( $S$ : parameters)
------------------------

Lcm( $S$ : parameters)
------------------------

LeastCommonMultiple( $S$ : parameters)
--

Presentation

MONSTGELT

Default :

Given a set or a sequence  $S$  containing elements of a braid group  $B$ , return the left-gcd of the elements of  $S$ , that is, the with respect to  $\preceq$  minimal element  $d$  of  $B$  satisfying  $s \preceq d$  for all  $s \in S$ , where  $\preceq$  is defined as above.

RightLCM(S: parameters)
-------------------------

RightLcm(S: parameters)
-------------------------

RightLeastCommonMultiple(S: parameters)
---

Presentation

MONSTGELT

Default :

Given a set or a sequence  $S$  containing elements of a braid group  $B$ , return the right-lcm of the elements of  $S$ , that is, the with respect to  $\succeq$  minimal element  $d$  of  $B$  satisfying  $d \succeq s$  for all  $s \in S$ , where  $\succeq$  is defined as above.

### Example H73E6

---

We define the braid group  $B$  on 6 strings.

```
> B := BraidGroup(6);
> SetElementPrintFormat(~B, "CFP");
```

(1) For both Artin and BKL presentation, the fundamental element is the (left or right) least common multiple of the generators. We check this for the Artin presentation...

```
> D_Artin := LeftLCM({B.i : i in [1..NumberOfGenerators(B)]});
> D_Artin eq FundamentalElement(B);
true
> D_Artin eq RightLCM({B.i : i in [1..NumberOfGenerators(B)]});
true
```

... and for the BKL presentation.

```
> idx := { <r,t> : r,t in {1..NumberOfStrings(B)} | r gt t };
> D_BKL := LeftLCM({B.T : T in idx} : Presentation := "BKL");
> D_BKL eq FundamentalElement(B : Presentation := "BKL");
true
> D_BKL eq RightLCM({B.T : T in idx} : Presentation := "BKL");
true
```

In general, left and right least common multiple of elements are different.

```
> LeftLCM(B.1,B.1*B.2) eq RightLCM(B.1,B.1*B.2);
false
```

(2) For both Artin and BKL presentation, the following hold. Let  $D$  denote the fundamental element.

- The simple elements are those positive elements  $s$  satisfying  $s \preceq D$  (or  $D \succeq s$ ).
- A product  $u_1 \cdots u_r$  of simple elements is in left normal form, if and only if the left greatest common divisor of  $u_i^{-1}D$  and  $u_{i+1}$  is trivial for all  $i = 1, \dots, r-1$ .

We illustrate this for the Artin presentation.

```
> D := FundamentalElement(B);
> forall{ s : s in Sym(6) | B!1 le B!s and B!s le D };
true
> forall{ s : s in Sym(6) | D ge B!s and B!s ge B!1 };
```

true

We create an element  $u$  as product of random simple elements.

```
> u := Random(B, 0, 0, 3, 5);
> u;
<Artin, 0, [
  (1, 5, 2)(3, 6),
  (1, 6, 5, 3),
  (1, 6, 5, 3, 2)
], 0>
```

We define a sequence of elements of  $B$ , containing the simple elements of the above representation of  $u$  using the function `CanonicalFactorRepresentation` and the coercion operator `!`.

```
> cfu := [ B!x : x in CFP(u)[3] ];
```

This representation is not in left normal form, as the above condition is violated for  $i = 1$ .

```
> IsId(LeftGCD(cfu[1]^-1*D, cfu[2]));
false
```

We now bring  $u$  into left normal form and extract again the sequence of simple elements.

```
> n := NormalForm(u);
> n;
<Artin, 0, [
  (1, 5, 3, 6, 2),
  (1, 6, 3, 2, 5)
], 0>
> cfn := [ B!x : x in CFP(n)[3] ];
```

This time, the above condition is satisfied.

```
> IsId(LeftGCD(cfn[1]^-1*D, cfn[2]));
true
```

### 73.4.6 Invariants of Conjugacy Classes

This section describes the functions for computing the set of positive conjugates, the super summit set and the ultra summit set for an element of a braid group  $B$  as defined in Section 73.1.5, as well as related MAGMA functions.

All the class invariants in general depend on the presentation of  $B$  used for their definition. Many functions documented in this section accept a parameter `Presentation` which can be used to specify the presentation of  $B$  with respect to which the computation should be performed. Possible values for this parameter are the strings "Artin" and "BKL". If no value is given for `Presentation`, the presentation selected for  $B$  is used.

For any given element  $u \in B$ , all the invariants defined in Section 73.1.5 are finite and can be computed in principle. In practice, however, computations may fail because the sets can get very large with increasing canonical length of  $u$  or with increasing braid index of  $B$ . This is in particular the case for sets of positive conjugates and for super summit sets.

**PositiveConjugates(u: parameters)**

**Presentation**

MONSTGELT

*Default :*

Given an element  $u$  of a braid group  $B$ , return an indexed set containing the conjugates of  $u$  which can be represented as positive words in the specified presentation of  $B$ .

**SuperSummitRepresentative(u: parameters)**

**Presentation**

MONSTGELT

*Default :*

Given an element  $u$  of a braid group  $B$ , return an element  $u_s$  of the super summit set of  $u$  with respect to the specified presentation of  $B$  and an element  $c$  of  $B$  satisfying  $u^c = u_s$ .

Note that  $u_s$  is a positive conjugate of  $u$ , if  $u_s$  has non-negative infimum and that  $u$  does not have any positive conjugates if the infimum of  $u_s$  is negative.

**SuperSummitSet(u: parameters)**

**Presentation**

MONSTGELT

*Default :*

Given an element  $u$  of a braid group  $B$ , return the super summit set of  $u$  with respect to the specified presentation as indexed set of elements of  $B$ .

**UltraSummitRepresentative(u: parameters)**

**Presentation**

MONSTGELT

*Default :*

Given an element  $u$  of a braid group  $B$ , return an element  $u_s$  of the ultra summit set of  $u$  with respect to the specified presentation of  $B$  and an element  $c$  of  $B$  satisfying  $u^c = u_s$ .

Note that  $u_s$  is an element of the super summit set of  $u$ , that  $u_s$  is a positive conjugate of  $u$ , if  $u_s$  has non-negative infimum and that  $u$  does not have any positive conjugates if the infimum of  $u_s$  is negative.

**UltraSummitSet(u: parameters)**

**Presentation**

MONSTGELT

*Default :*

Given an element  $u$  of a braid group  $B$ , return the ultra summit set of  $u$  with respect to the specified presentation as indexed set of elements of  $B$ .

### Example H73E7

---

(1) In the braid group  $B$  on 4 strings we compute the sets of positive conjugates and the super summit sets of  $u = \sigma_1\sigma_2\sigma_1$  with respect to both Artin presentation and BKL presentation.

```
> B := BraidGroup(4);
> u := B.1*B.2*B.1;
> p_Artin := PositiveConjugates(u : Presentation := "Artin");
> p_BKL := PositiveConjugates(u : Presentation := "BKL");
> s_Artin := SuperSummitSet(u : Presentation := "Artin");
```

```
> s_BKL := SuperSummitSet(u : Presentation := "BKL");
```

Since the Artin generators form a subset of the BKL generators, every element which is positive with respect to the Artin presentation is also positive with respect to the BKL presentation. In particular, `p_Artin` is a subset of `p_BKL`.

```
> p_Artin subset p_BKL;
true
```

The converse inclusion does not hold.

```
> #p_Artin;
10
> #p_BKL;
36
```

For both presentations the super summit set is a subset of the set of positive conjugates, as  $u$  is positive. The converse inclusions do not hold.

```
> s_Artin subset p_Artin;
true
> s_BKL subset p_BKL;
true
> #s_Artin;
2
> #s_BKL;
12
```

(2) As we have seen in Section 73.1.5, we can decide whether two braids are conjugate by checking whether their super summit sets are equal.

We illustrate this approach with two elements of  $B$ , using the Artin presentation of  $B$ .

```
> u := B.2 * B.1 * B.2^2 * B.1 * B.2;
> v := B.2^2 * B.1 * B.3 * B.1 * B.3;
```

Suppose we want to prove that  $u$  and  $v$  are not conjugate in  $B$ . We could start by checking the cycle structure of the induced permutations on the strings on which  $B$  acts.

```
> CycleStructure(InducedPermutation(u));
[ <1, 4> ]
> CycleStructure(InducedPermutation(v));
[ <1, 4> ]
```

This does not help. Next we can check the infima and the suprema of super summit representatives.

```
> SuperSummitInfimum(u) eq SuperSummitInfimum(v);
true
> SuperSummitSupremum(u) eq SuperSummitSupremum(v);
true
```

Again, we cannot conclude anything. We decide to compare the super summit sets of  $u$  and  $v$ .

```
> SuperSummitSet(u) eq SuperSummitSet(v);
```

false

Success! The super summit sets of  $u$  and  $v$  are different, proving that  $u$  and  $v$  are not conjugate. For a more efficient version of conjugacy testing see Example H73E8.

(3) Finally, we illustrate the significant difference in the sizes of super summit sets and ultra summit sets for slightly larger values of braid index and canonical length.

```
> B := BraidGroup(8);
```

We create a pseudo-random element of  $B$  as product of 5 simple elements independently chosen at random.

```
> x := B.4 * B.3 * B.2 * B.1 * B.5 * B.4 * B.5 *
> B.6 * B.7 * B.6 * B.5;
> x := x^2;
> Sx := SuperSummitSet(x);
> #Sx;
10972
> Ux := UltraSummitSet(x);
> #Ux;
36
```

The ultra summit set is much smaller than the super summit set. We try again.

```
> x := B.4 * B.3 * B.2 * B.1 * B.5 * B.4 * B.5;
> x := x^3;
> Sx := SuperSummitSet(x);
> #Sx;
882
> Ux := UltraSummitSet(x);
> #Ux;
18
```

The difference in sizes is still large. The behaviour exhibited by these examples is quite typical. In particular, the sizes of super summit sets for braids on a given number of strings and with a given canonical length show much larger fluctuations than the sizes of ultra summit sets. For a more detailed analysis we refer to [Geb03].

---

### 73.4.6.1 Computing Class Invariants Interactively

This section describes the functions relevant for interactive computation of the set of positive conjugates, the super summit set and the ultra summit set for a given element of a braid group  $B$  as defined in Section 73.1.5.

Process versions of the algorithms used by the functions `PositiveConjugates`, `SuperSummitSet` and `UltraSummitSet` are available for computing these invariants one element at a time.

**PositiveConjugatesProcess**(*u*: *parameters*)

**Presentation**

MONSTGELT

*Default* :

Given an element  $u$  of a braid group  $B$ , return a process for constructing the conjugates of  $u$  which can be represented as positive words in the specified presentation of  $B$ .

The returned process contains the first positive conjugate of  $u$  if positive conjugates exist and is *empty* otherwise.

**SuperSummitProcess**(*u*: *parameters*)

**Presentation**

MONSTGELT

*Default* :

Given an element  $u$  of a braid group  $B$ , return a process for constructing the super summit elements of  $u$  with respect to the specified presentation of  $B$ .

The returned process contains the first super summit element of  $u$ .

**UltraSummitProcess**(*u*: *parameters*)

**Presentation**

MONSTGELT

*Default* :

Given an element  $u$  of a braid group  $B$ , return a process for constructing the ultra summit elements of  $u$  with respect to the specified presentation of  $B$ .

The returned process contains the first ultra summit element of  $u$ .

**BaseElement**( $P$ )

Return the element used for the construction of the process  $P$ .

**#P**

Return the number of elements that have been found by the process  $P$ .

**Representative**( $P$ )

**Rep**( $P$ )

Given a non-empty process  $P$ , return the element most recently found by  $P$ .

If  $P$  is empty, a runtime error will occur. The function `IsEmpty` can be used for checking whether a process is empty, in order to avoid runtime errors in loops and user written functions.

**IsEmpty**( $P$ )

Return `true` if  $P$  is empty and `false` otherwise.

This function can be used to check whether `Representative` can be called for a process  $P$ .

**Elements**( $P$ )

Return an indexed set containing the elements found so far by the process  $P$ .

**u in P**

Given an element  $u$  of a braid group and a process  $P$  for computing positive conjugates or super summit elements of the element  $b$ , return **true** and an element  $c$  satisfying  $b^c = u$  if  $u$  is one of the elements that have been constructed by  $P$  and **false** otherwise.

**u notin P**

Given an element  $u$  of a braid group and a process  $P$ , return **false** if  $u$  is one of the elements that have been constructed by  $P$  and **true** otherwise.

**NextElement( $\sim P$ )**

Given a process  $P$ , continue searching for elements until the next element is found or the search completes without finding a new element.

If a new element is found, it can subsequently be accessed using the function **Representative**. If the search completes without finding a new element,  $P$  is marked as *empty*. Calling **NextElement** on an empty process has no effect.

**Complete( $\sim P$ )**

Given a process  $P$ , complete the search for elements. After executing this procedure,  $P$  is *empty* and the set of all elements found by  $P$  can be accessed using the function **Elements**. Calling **Complete** on an empty process has no effect.

### Example H73E8

---

We sketch how the functions described in the preceding section could be used for testing whether two elements are conjugate and for computing a conjugating element if they are.

The approach outlined here is basically the algorithm used by the function **IsConjugate**.

```
> function MyIsConjugate(u, v)
>
> // check obvious invariants
> infu := SuperSummitInfimum(u);
> infv := SuperSummitInfimum(v);
> supu := SuperSummitSupremum(u);
> supv := SuperSummitSupremum(v);
> if infu ne infv or supu ne supv then
>   return false, _;
> end if;
>
> // compute an ultra summit element for v
> sv, cv := UltraSummitRepresentative(v);
>
> // set up a process for computing the ultra summit set of u
> P := UltraSummitProcess(u);
>
> // compute ultra summit elements of u until sv is found
> //   or sv is seen not to be in the ultra summit set of u
```

```

> while sv notin P and not IsEmpty(P) do
>   NextElement(~P);
> end while;
>
> print #P, "elements computed";
> isconj, c := sv in P;
> if isconj then
>   // return true and an element conjugating u to v
>   return true, c*cv^-1;
> else
>   return false, _;
> end if;
>
> end function;

```

We test our function using two pairs of elements of the braid group  $B$  on 4 strings.

```
> B := BraidGroup(4);
```

As we have seen in Example H73E7, the following elements  $u$  and  $v$  are not conjugate.

```

> u := B.2 * B.1 * B.2^2 * B.1 * B.2;
> v := B.2^2 * B.1 * B.3 * B.1 * B.3;

```

To prove this, our function has to compute the whole ultra summit set of  $u$ .

```

> MyIsConjugate(u,v);
2 elements computed
false
> #UltraSummitSet(u);
2

```

We try our function on another pair of elements.

```

> r := B.3*B.2*B.3*B.2^2*B.1*B.3*B.1*B.2;
> s := B.3^-1*B.2^-1*B.3*B.2*B.3*B.2^2*B.1*B.3*B.1*B.2^2*B.3;
> isconj, c := MyIsConjugate(r,s);
3 elements computed
> isconj;
true
> r^c eq s;
true

```

The ultra summit representative of  $s$  was the 3rd ultra summit element of  $r$  found. Note that the function did not have to compute the whole ultra summit set of  $r$  to find the answer.

```

> #UltraSummitSet(r);
6

```

In this small example, we could also have used super summit sets for conjugacy testing, as the super summit set of  $r$  is not much larger than its ultra summit set.

```
> #SuperSummitSet(r);
```

22

A more challenging application of the function `MyIsConjugate` from above will be presented in Example H73E10.

### 73.4.6.2 Computing Minimal Simple Elements

This section describes the functions for computing minimal simple elements as introduced in Section 73.1.5.2 and functions for computing the *transport* and the *pullback* as defined in [Geb03].

All functions documented in this section accept two parameters, `Presentation` and `CheckArguments`. The parameter `Presentation` can be used to specify the presentation of a braid group  $B$  with respect to which the computation should be performed. Possible values for this parameter are the strings "Artin" and "BKL". If no value is given for `Presentation`, the presentation selected for  $B$  is used. The parameter `CheckArguments` can be used to turn off argument checking for performance reasons. It should be noted that the results are undefined if functions are called with invalid arguments and argument checking is disabled.

<code>MinimalElementConjugatingToPositive(x, s: parameters)</code>		
--	--	--

<code>Presentation</code>	MONSTGELT	<i>Default :</i>
<code>CheckArguments</code>	BOOLELT	<i>Default : true</i>

Given a positive element  $x$  of a braid group  $B$  and a simple element  $s$ , return the minimal simple element  $r_x(s)$  satisfying  $s \preceq r_x(s)$  and  $x^{r_x(s)} \in B^+$ .

<code>MinimalElementConjugatingToSuperSummit(x, s: parameters)</code>		
---	--	--

<code>Presentation</code>	MONSTGELT	<i>Default :</i>
<code>CheckArguments</code>	BOOLELT	<i>Default : true</i>

Given an element  $x$  of a braid group  $B$  which is contained in its super summit set  $S_x$  and a simple element  $s$ , return the minimal simple element  $\rho_x(s)$  satisfying  $s \preceq \rho_x(s)$  and  $x^{\rho_x(s)} \in S_x$ .

<code>MinimalElementConjugatingToUltraSummit(x, s: parameters)</code>		
---	--	--

<code>Presentation</code>	MONSTGELT	<i>Default :</i>
<code>CheckArguments</code>	BOOLELT	<i>Default : true</i>

Given an element  $x$  of a braid group  $B$  which is contained in its ultra summit set  $U_x$  and a simple element  $s$ , return the minimal simple element  $c_x(s)$  satisfying  $s \preceq c_x(s)$  and  $x^{c_x(s)} \in U_x$ .

**Transport(x, s: parameters)**

<b>Presentation</b>	MONSTGELT	<i>Default :</i>
<b>CheckArguments</b>	BOOLELT	<i>Default : true</i>

Given an element  $x$  of a braid group  $B$  and a simple element  $s$  such that both  $x$  and  $x^s$  are super summit elements, return the *transport* of  $s$  along  $x \rightarrow \mathbf{c}(x)$ , that is, the element  $\phi_x(s) = (D \wedge_l x D^{-\text{inf}(x)})^{-1} \cdot s \cdot (D \wedge_l x^s D^{-\text{inf}(x)})$ , where  $D$  is the fundamental element of  $B$ . The transport is a simple element satisfying  $\mathbf{c}(x^s) = \mathbf{c}(x)^{\phi_x(s)}$  [Geb03].

**Pullback(x, s: parameters)**

<b>Presentation</b>	MONSTGELT	<i>Default :</i>
<b>CheckArguments</b>	BOOLELT	<i>Default : true</i>

Given an element  $x$  of a braid group  $B$  which is contained in its super summit set  $S_x$  and a simple element  $s$ , return the *pullback* of  $s$  along  $x \rightarrow \mathbf{c}(x)$ , that is, the unique  $\preceq$ -minimal element  $\pi_x(s)$  satisfying  $x^{\pi_x(s)} \in S_x$  and  $s \preceq \phi_x(\pi_x(s))$  [Geb03].

**Example H73E9**

The following function uses the technique sketched in Section 73.1.5 for computing the ultra summit set of a given braid. This is basically the algorithm used by the MAGMA function `UltraSummitSet`.

```
> function MyUltraSummitSet(x)
>
> // create a subset of the ultra summit set of x
> U := {@ UltraSummitRepresentative(x) @};
> gens := Generators(Parent(x));
> pos := 1;
>
> // close U under conjugation with minimal simple elements
> while pos le #U do
>   y := U[pos];
>   // add missing conjugates of y
>   for z in { y^MinimalElementConjugatingToUltraSummit(y, s)
>             : s in gens } do
>     if z notin U then
>       Include(~U, z);
>     end if;
>   end for;
>   pos += 1;
> end while;
>
> return U;
>
> end function;
```

## 73.5 Homomorphisms

For a general description of homomorphisms, we refer to Chapter 16. This section describes some special aspects of homomorphisms whose domain is in the category `GrpBrd`.

### 73.5.1 General Remarks

An important special case of homomorphisms with domain in the category `GrpBrd` is the following. A homomorphism  $f : B \rightarrow G$ , where  $B$  and  $B'$  are braid groups on  $n$  and  $m$  strings, respectively, and  $f$  is an embedding of  $B$  in  $G$  induced by

$$\sigma_i \rightarrow \sigma'_{k+\epsilon i} \quad (i = 1, \dots, n-1)$$

where the  $\sigma_i$  are the Artin generators of  $B$ , the  $\sigma'_j$  are the Artin generators of  $B'$ ,  $|\epsilon| = 1$  and  $k$  is a suitable constant.

The MAGMA implementation uses special optimisation techniques, if a homomorphism with domain in the category `GrpBrd` has the additional properties listed above. Compared to the general case, this results in faster evaluation of such homomorphisms, in particular in the case  $\epsilon = 1$ .

Computing preimages under homomorphisms with domain in the category `GrpBrd` currently is only supported for the special case described above. Moreover, computing the preimage of an element  $u$  under a map  $f$  may fail, even if  $u$  is contained in the image of  $f$ .

### 73.5.2 Constructing Homomorphisms

```
hom< B -> G | S : parameters >
```

Check

BOOLELT

Default : true

Returns the homomorphism from the braid group  $B$  to the group  $G$  defined by the assignment  $S$ .  $S$  can be the one of the following:

- (i) A list, sequence or indexed set containing the images of all  $k$  Artin generators  $B.1, \dots, B.k$  of  $B$ . Here, the  $i$ -th element of  $S$  is interpreted as the image of  $B.i$ , that is, the order of the elements in  $S$  is important.
- (ii) A list, sequence, enumerated set or indexed set, containing  $k$  tuples  $\langle x_i, y_i \rangle$  or arrow pairs  $x_i \rightarrow y_i$ , where  $x_i$  is a generator of  $B$  and  $y_i \in G$  ( $i = 1, \dots, k$ ) and the set  $\{x_1, \dots, x_k\}$  is the full set of Artin generators of  $B$ . In this case,  $y_i$  is assigned as the image of  $x_i$ , hence the order of the elements in  $S$  is not important.

Note, that it is currently not possible to define a homomorphism by assigning images to the elements of an arbitrary generating set of  $B$ .

If the category of the codomain supports element arithmetic and element comparison, by default the constructed homomorphism is checked by verifying that the would-be images of the Artin generators satisfy the braid relations of  $B$ . In this case, it is assured that the returned map is a well-defined homomorphism. The most important situation in which it is not possible to perform checking is the case in which the domain is a finitely presented group (`FPGroup`; cf. Chapter 70) which is not free. Checking may be disabled by setting the parameter `Check` to `false`.

### 73.5.3 Accessing Homomorphisms

`e @ f`

`f(e)`

Given a homomorphism whose domain is a braid group  $B$  and an element  $e$  of  $B$ , return the image of  $e$  under  $f$  as element of the codomain of  $f$ .

`B @ f`

`f(B)`

Given a homomorphism whose domain is a braid group  $B$ , return the image of  $B$  under  $f$  as a subgroup of the codomain of  $f$ .

This function is not supported for all codomain categories.

`u @@ f`

Given a homomorphism whose domain is a braid group  $B$  and an element  $u$  of the image of  $f$ , return the preimage of  $u$  under  $f$  as an element of  $B$ .

This function currently is only supported if  $f$  is an embedding of one braid group into another as described in Section 73.5.1. Note, moreover, that computing the preimage of  $u$  may fail, even if  $u$  is contained in the image of  $f$ .

`Domain(f)`

The domain of the homomorphism  $f$ .

`Codomain(f)`

The codomain of the homomorphism  $f$ .

`Image(f)`

The image or range of the homomorphism  $f$  as a subgroup of the codomain of  $f$ .

This function is not supported for all codomain categories.

#### Example H73E10

---

(1) The symmetric group on  $n$  letters is an epimorphic image of the braid group on  $n$  strings, where for  $0 < i < n$  the image of the Artin generator  $\sigma_i$  is given by the transposition  $(i, i + 1)$ . We construct this homomorphism for the case  $n = 10$ .

```
> Bn := BraidGroup(10);
> Sn := Sym(10);
> f := hom< Bn->Sn | [ Sn!(i,i+1) : i in [1..Ngens(Bn)] ] >;
```

Of course, the image of `f` is the full symmetric group.

```
> Image(f) eq Sn;
true
```

Now we compute the image of a pseudo-random element of `Bn` under `f`.

```
> f(Random(Bn));
```

(1, 5, 8)(2, 4, 9, 7, 6, 3)

(2) (Key exchange as proposed in [KLC<sup>+</sup>00])

Consider a collection of  $l+r$  strings  $t_1, \dots, t_{l+r}$  and the braid group  $B$  acting on  $t_1, \dots, t_{l+r}$  with Artin generators  $\sigma_1, \dots, \sigma_{l+r-1}$ . The subgroups of  $B$  fixing the strings  $t_{l+1}, \dots, t_{l+r}$  and  $t_1, \dots, t_l$  may be identified with braid groups  $L$  on  $l$  strings and  $R$  on  $r$  strings, respectively, with the Artin generators of  $L$  and  $R$  corresponding to  $\sigma_1, \dots, \sigma_{l-1}$  and  $\sigma_{l+1}, \dots, \sigma_{l+r-1}$ , respectively.

We set up these groups for  $l = 6$  and  $r = 7$  using the BKL presentations.

```
> l := 6;
> r := 7;
> B := BraidGroup(l+r : Presentation := "BKL");
> L := BraidGroup(l : Presentation := "BKL");
> R := BraidGroup(r : Presentation := "BKL");
```

We now construct the embeddings  $f : L \rightarrow B$  and  $g : R \rightarrow B$ .

```
> f := hom< L-> B | [ L.i -> B.i : i in [1..Ngens(L)] ] >;
> g := hom< R-> B | [ R.i -> B.(l+i) : i in [1..Ngens(R)] ] >;
```

To complete the preparatory steps, we choose a random element of  $B$  which is not too short.

```
> x := Random(B, 15, 25);
```

The data constructed so far is assumed to be publicly available. Each time two users A and B require a shared key, the following steps are performed.

- (a) A chooses a random secret element  $a \in L$  and sends the normal form of  $y_1 := x^a$  to B.
- (b) B chooses a random secret element  $b \in R$  and sends the normal form of  $y_2 := x^b$  to A.
- (c) A receives  $y_2$  and computes the normal form of  $y_2^a$ .
- (d) B receives  $y_1$  and computes the normal form of  $y_1^b$ .

Note the following.

- Transmitting  $y_1$  and  $y_2$  in normal form disguises their structure as products  $a^{-1}xa$  and  $b^{-1}xb$ , provided the words used are long enough and prevents simply reading off the conjugating elements  $a$  and  $b$ .
- Since the subgroups  $L$  and  $R$  of  $B$  commute, we have  $ab = ba$ , which implies  $y_2^a = x^{ba} = x^{ab} = y_1^b$ . Thus, the normal forms computed by A and B in steps (c) and (d), respectively, must be identical and can be used to extract a secret shared key.

We illustrate this, using the groups set up above. For step (a):

```
> a := Random(L, 15, 25);
> y1 := NormalForm(x^f(a));
```

Now for step (b):

```
> b := Random(R, 15, 25);
> y2 := NormalForm(x^g(b));
```

We now verify that A and B arrive at the same information in steps (c) and (d).

```
> K_A := NormalForm(y2^f(a));
```

```
> K_B := NormalForm(y1^g(b));
> AreIdentical(K_A, K_B);
true
```

We see that the information computed by A and B in steps (c) and (d) is indeed identical and hence can be used (in suitable form) as a common secret. Note, however, that the number of strings and the lengths of the elements used in the example above are much smaller than the values suggested for real cryptographic purposes.

### (3) (Attack on key exchange)

We now show an attack on the key exchange outlined above using conjugacy search based on ultra summit sets.

An eavesdropper can try to compute an element  $c$  conjugating  $x$  to  $y_1$ . While this is not guaranteed to reproduce the braid  $a$ , the chances for a successful key recovery are quite good.

We decide to change to the Artin presentation of  $B$  and use the function `MyIsConjugate` from Example H73E8 for computing a conjugating element  $c$  as above.

```
> SetPresentation(~B, "Artin");
> time _, c := MyIsConjugate(x, y1);
42 elements computed
Time: 0.020
```

Finding a conjugating element is no problem at all. Using the conjugating element, we can try to recover the shared secret. In this example we are lucky.

```
> NormalForm(y2^c) eq K_A;
true
```

In the conjugacy search above, a conjugating element was found after computing 42 ultra summit elements. The ultra summit set itself is larger, but can still be computed very easily.

```
> time Ux := UltraSummitSet(x);
Time: 3.150
> #Ux;
1584
```

The super summit set is, even in this small example, too large to be computed; conjugacy search based on super summit sets would quite likely fail.

```
> time Sx := SuperSummitSet(x);
Current total memory usage: 4055.1MB
System error: Out of memory.
```

Finally, we show that the attack using conjugacy search based on ultra summit sets is also applicable to larger examples. We try to recover a key, which is generated using elements with canonical lengths between 500 and 1000 in a braid group on 100 strings.

```
> l := 50;
> r := 50;
> B := BraidGroup(l+r);
> L := BraidGroup(l);
```

```

> R := BraidGroup(r);
>
> f := hom< L-> B | [ L.i -> B.i : i in [1..Ngens(L)] ] >;
> g := hom< R-> B | [ R.i -> B.(l+i) : i in [1..Ngens(R)] ] >;
>
> x := Random(B, 0, 1, 500, 1000);
>
> a := Random(L, 0, 1, 500, 1000);
> y1 := NormalForm(x^f(a));
>
> b := Random(R, 0, 1, 500, 1000);
> y2 := NormalForm(x^g(b));
>
> K_A := NormalForm(y2^f(a));
> K_B := NormalForm(y1^g(b));
> AreIdentical(K_A, K_B);
true

```

We again try to recover the key by computing an element conjugating  $x$  to  $y_1$ . This time, we use the built-in MAGMA function for efficiency reasons.

```

> time _, c := IsConjugate(x, y1);
Time: 18.350
> K_A eq NormalForm(y2^c);
false

```

Bad luck. – We managed to compute a conjugating element, but this failed to recover the key. We try with an element conjugating  $x$  to  $y_2$ .

```

> time _, c := IsConjugate(x, y2);
Time: 3.800
> K_B eq NormalForm(y1^c);
true

```

Success! – Good that we didn't use this key to encrypt our credit card number!

---

### 73.5.4 Representations of Braid Groups

This section describes the functions available for creating a number of well known representations of braid groups.

<b>SymmetricRepresentation(B)</b>
-----------------------------------

Given a braid group  $B$  on  $n$  strings, return the natural epimorphism from  $B$  onto the symmetric group on  $n$  points, induced by the action of  $B$  on the strings by which  $B$  is defined.

**BureauRepresentation(B)**

Given a braid group  $B$  on  $n$  strings, return the Bureau representation of  $B$  as homomorphism from  $B$  to the matrix algebra of degree  $n$  over the rational function field over the integers.

**BureauRepresentation(B, p)**

Given a braid group  $B$  on  $n$  strings and a prime  $p$ , return the  $p$ -modular Bureau representation of  $B$  as homomorphism from  $B$  to the matrix algebra of degree  $n$  over the rational function field over the field with  $p$  elements.

**Example H73E11**

---

We construct the Bureau representation of the braid group on 4 strings.

```
> B := BraidGroup(4);
> f := BureauRepresentation(B);
```

Its codomain is a matrix algebra of degree 4 over the rational function field over the integers.

```
> A := Codomain(f);
> A;
GL(4, FunctionField(IntegerRing()))
> F := BaseRing(A);
> F;
Univariate rational function field over Integer Ring
Variables: $.1
```

To obtain nicer printing, we assign the name  $t$  to the generator of the function field  $F$ .

```
> AssignNames(~F, ["t"]);
```

Now we can check easily whether we remembered the definition of the Bureau representation correctly.

```
> f(B.1);
[-t + 1    t    0    0]
[    1    0    0    0]
[    0    0    1    0]
[    0    0    0    1]

> f(B.2);
[    1    0    0    0]
[    0 -t + 1    t    0]
[    0    1    0    0]
[    0    0    0    1]

> f(B.3);
[    1    0    0    0]
[    0    1    0    0]
[    0    0 -t + 1    t]
[    0    0    1    0]
```

---

### 73.6 Bibliography

- [**AAG99**] Iris Anshel, Michael Anshel, and Dorian Goldfeld. An algebraic method for public-key cryptography. *Math. Res. Lett.*, 6(3-4):287–291, 1999.
- [**Art47**] E. Artin. Theory of braids. *Ann. of Math. (2)*, 48:101–126, 1947.
- [**BKL98**] Joan Birman, Ki Hyoung Ko, and Sang Jin Lee. A new approach to the word and conjugacy problems in the braid groups. *Adv. Math.*, 139(2):322–353, 1998.
- [**Deh02**] Patrick Dehornoy. Groupes de Garside. *Ann. Sci. École Norm. Sup. (4)*, 35(2):267–306, 2002.
- [**ECH<sup>+</sup>92**] David B. A. Epstein, James W. Cannon, Derek F. Holt, Silvio V. F. Levy, Michael S. Paterson, and William P. Thurston. *Word processing in groups*. Jones and Bartlett Publishers, Boston, MA, 1992.
- [**ERM94**] Elsayed A. El-Rifai and H. R. Morton. Algorithms for positive braids. *Quart. J. Math. Oxford Ser. (2)*, 45(180):479–497, 1994.
- [**FGM03**] Nuno Franco and Juan González-Meneses. Conjugacy problem for braid groups and Garside groups. *J. Algebra*, 266(1):112–132, 2003.
- [**Gar69**] F. A. Garside. The braid group and other groups. *Quart. J. Math. Oxford Ser. (2)*, 20:235–254, 1969.
- [**Geb03**] Volker Gebhardt. A new approach to the conjugacy problem in Garside groups. Preprint; available at URL:<http://www.arxiv.org/math.GT/0306199>, 2003.
- [**GKT<sup>+</sup>02**] D. Garber, S. Kaplan, M. Teicher, B. Tsaban, and U. Vishne. Length-based conjugacy search in the braid group. Preprint; available at URL:<http://www.arxiv.org/math.GR/0209267>, 2002.
- [**GM02**] Juan González-Meneses. Improving an algorithm to solve Multiple Simultaneous Conjugacy Problems in braid groups. Preprint; available at URL:<http://www.arxiv.org/math.GT/0212150>, 2002.
- [**HS03**] D. Hofheinz and R. Steinwandt. A practical attack on some braid group based cryptographic primitives. In *Public Key Cryptography, 6th International Workshop on Practice and Theory in Public Key Cryptography, PKC 2003*, volume 2567 of *LNCS*, pages 187–198. Springer, 2003.
- [**Hug02**] J. Hughes. A linear algebraic attack on the AAFG1 braid group cryptosystem. In *The 7th Australasian Conference on Information Security and Privacy, ACISP 2002*, volume 2384 of *LNCS*, pages 176–189. Springer, 2002.
- [**KLC<sup>+</sup>00**] Ki Hyoung Ko, Sang Jin Lee, Jung Hee Cheon, Jae Woo Han, Ju-sung Kang, and Choonsik Park. New public-key cryptosystem using braid groups. In *Advances in cryptology—CRYPTO 2000*, pages 166–183. Springer, Berlin, 2000.
- [**LL02**] Sang Jin Lee and Eonkyung Lee. Potential Weakness of the Commutator Key Agreement Protocol Based on Braid Groups. In *Advances in Cryptology—EuroCrypt 2002*, volume 2332 of *LNCS*, pages 14–28. Springer, 2002.
- [**LP03**] E. Lee and J. H. Park. Cryptanalysis of the public key encryption based on braid groups. In *Advances in Cryptology—EuroCrypt 2003*, volume 2656 of *LNCS*, pages 477–490. Springer, 2003.

# 74 GROUPS DEFINED BY REWRITE SYSTEMS

<b>74.1 Introduction . . . . .</b>	<b>2339</b>
74.1.1 Terminology . . . . .	2339
74.1.2 The Category of Rewrite Groups . . . . .	2339
74.1.3 The Construction of a Rewrite Group	2339
<b>74.2 Constructing Confluent Presentations . . . . .</b>	<b>2340</b>
74.2.1 The Knuth-Bendix Procedure . . . . .	2340
RWSGroup(F: -)	2340
74.2.2 Defining Orderings . . . . .	2341
RWSGroup(F: -)	2341
74.2.3 Setting Limits . . . . .	2343
RWSMonoid(F: -)	2344
SetVerbose("KBMAG", v)	2345
74.2.4 Accessing Group Information . . . . .	2345
.	2346
Generators(G)	2346
NumberOfGenerators(G)	2346
Ngens(G)	2346
Relations(G)	2346
NumberOfRelations(G)	2346
Nrels(G)	2346
Ordering(G)	2346
<b>74.3 Properties of a Rewrite Group</b>	<b>2347</b>
IsConfluent(G)	2347
IsFinite(G)	2347
Order(G)	2347
#	2347
<b>74.4 Arithmetic with Words . . . . .</b>	<b>2348</b>
74.4.1 Construction of a Word . . . . .	2348
Identity(G)	2348
Id(G)	2348
!	2348
!	2348
Parent(w)	2349
74.4.2 Element Operations . . . . .	2349
*	2349
/	2349
^	2349
~	2350
Inverse(w)	2350
(u, v)	2350
(u <sub>1</sub> , . . . , u <sub>r</sub> )	2350
eq	2350
ne	2350
IsId(w)	2350
IsIdentity(w)	2350
#	2350
ElementToSequence(u)	2350
Eltseq(u)	2350
<b>74.5 Operations on the Set of Group Elements . . . . .</b>	<b>2351</b>
Random(G, n)	2351
Random(G)	2351
Representative(G)	2351
Rep(G)	2351
Set(G, a, b)	2352
Set(G)	2352
Seq(G, a, b)	2352
Seq(G)	2352
<b>74.6 Homomorphisms . . . . .</b>	<b>2353</b>
74.6.1 General Remarks . . . . .	2353
74.6.2 Construction of Homomorphisms . . . . .	2353
hom< >	2353
<b>74.7 Conversion to a Finitely Presented Group . . . . .</b>	<b>2354</b>
<b>74.8 Bibliography . . . . .</b>	<b>2354</b>



# Chapter 74

## GROUPS DEFINED BY REWRITE SYSTEMS

### 74.1 Introduction

The class of finitely presented groups defined by finite rewrite systems provide a Magma level interface to Derek Holt's KBMAG programs, and specifically to the Knuth–Bendix completion procedure for groups defined by a finite (monoid) presentation. Much of the material in this chapter is taken from the KBMAG documentation [Hol97]. Familiarity with the Knuth–Bendix completion procedure is assumed. Some familiarity with KBMAG would be beneficial.

#### 74.1.1 Terminology

A rewrite group  $G$  is a finitely presented group in which equality between elements of  $G$ , called *words* or *strings*, may be decidable via a sequence of rewriting equations, called *reduction relations*, *rules*, or *equations*. In the interests of efficiency the reduction rules are codified into a finite state automaton called a *reduction machine*. The words in a rewrite group  $G$  are ordered, as are the reduction relations of  $G$ . Several possible orderings of words are supported, namely short-lex, recursive, weighted short-lex and wreath-product orderings. A rewrite group can be *confluent* or *non-confluent*. If a rewrite group  $G$  is confluent its reduction relations, or more specifically its reduction machine, can be used to reduce words in  $G$  to their irreducible normal forms under the given ordering, and so the word problem for  $G$  can be efficiently solved.

#### 74.1.2 The Category of Rewrite Groups

The family of all rewrite groups forms a category. The objects are the rewrite groups and the morphisms are group homomorphisms. The MAGMA designation for this category of groups is GrpRWS. Elements of a rewrite group are designated as GrpRWSElt.

#### 74.1.3 The Construction of a Rewrite Group

A rewrite group  $G$  is constructed in a three-step process:

- (i) We construct a free group  $FG$ .
- (ii) We construct a quotient  $F$  of  $FG$ .
- (iii) We create a monoid presentation of  $F$  and then run a Knuth–Bendix completion procedure on this presentation to create a rewrite group  $G$ .

The Knuth–Bendix procedure may or may not succeed. If it fails the user may need to perform the above steps several times, manually adjusting the parameters to the Knuth–Bendix procedure. If it succeeds then the rewrite system constructed will be confluent.

## 74.2 Constructing Confluent Presentations

### 74.2.1 The Knuth-Bendix Procedure

RWSGroup(F: *parameters*)

Suppose  $F$  is a finitely presented group. Internally, the first step is to construct a presentation for a monoid  $M$ . By default, the generators of  $M$  are taken to be  $g_1, g_1^{-1}, \dots, g_n, g_n^{-1}$ , where  $g_1, \dots, g_n$  are the generators of  $F$ . The relations for  $M$  are taken to be the relations of  $F$  together with the trivial relations  $g_1 g_1^{-1} = g_1^{-1} g_1 = 1$ . The Knuth–Bendix completion procedure for monoids is now applied to  $M$ . Regardless of whether or not the completion procedure succeeds, the result will be a rewrite monoid,  $M$ , containing a reduction machine and a sequence of reduction relations. If the procedure succeeds  $M$  will be marked as confluent, and the word problem for  $M$  is therefore decidable. If, as is very likely, the procedure fails then  $M$  will be marked as non-confluent. In this case  $M$  will contain both the reduction relations and the reduction machine computed up to the point of failure. Reductions made using these relations will be correct in  $F$ , but words that are equal in  $F$  are not guaranteed to reduce to the same word.

The Knuth–Bendix procedure requires ordering to be defined on both the generators and the words. The default generator ordering is that induced by the ordering of the generators of  $F$  while the default ordering on strings is the `ShortLex` order. We give a simple example and then discuss the parameters that allow the user to specify these two orderings.

As the Knuth–Bendix procedure will more often than not run forever, some conditions must be specified under which it will stop. These take the form of limits that are placed on certain variables, such as the number of reduction relations. If any of these limits are exceeded during a run of the completion procedure it will fail, returning a non-confluent rewrite monoid. The optimal values for these limits vary from example to example. The various parameters that allow the user to specify the limits for these variables will be described in a subsequent section.

---

#### Example H74E1

We construct the Von Dyck  $(2, 3, 5)$  group. Since a string ordering is not specified the default `ShortLex` ordering is used. Similarly, since a generator ordering is not specified, the default generator ordering, in this case  $[a, a^{-1}, b, b^{-1}]$ , is used.

```
> FG<a,b> := FreeGroup(2);
> F := quo< FG | a*a=1, b*b=b^-1, a*b^-1*a*b^-1*a=b*a*b*a*b>;
> G := RWSGroup(F);
> G;
A confluent rewrite group.
Generator Ordering = [ a, a^-1, b, b^-1 ]
Ordering = ShortLex.
The reduction machine has 39 states.
The rewrite relations are:
```

```

a^2 = Id(F)
b * b^-1 = Id(F)
b^-1 * b = Id(F)
b^2 = b^-1
b * a * b * a * b = a * b^-1 * a * b^-1 * a
b^-2 = b
b^-1 * a * b^-1 * a * b^-1 = a * b * a * b * a
a^-1 = a
a * b^-1 * a * b^-1 * a * b = b * a * b * a * b^-1
b * a * b^-1 * a * b^-1 * a = b^-1 * a * b * a * b
a * b * a * b * a * b^-1 = b^-1 * a * b^-1 * a * b
b^-1 * a * b * a * b * a = b * a * b^-1 * a * b^-1
b * a * b * a * b^-1 * a * b * a = a * b^-1 * a * b * a * b^-1 * a * b^-1
b^-1 * a * b^-1 * a * b * a * b^-1 * a = a * b * a * b^-1 * a * b * a * b
b^-1 * a * b * a * b^-1 * a * b * a * b^-1 = b * a * b^-1 * a * b * a * b^-1
    * a * b
b * a * b^-1 * a * b * a * b^-1 * a * b^-1 = (b^-1 * a * b * a)^2
b^-1 * a * b * a * b^-1 * a * b * a * b = (b * a * b^-1 * a)^2
b * a * b^-1 * a * b * a * b^-1 * a * b * a = a * b * a * b^-1 * a * b * a *
    b^-1 * a * b

```

## 74.2.2 Defining Orderings

RWSGroup(F: *parameters*)

Attempt to construct a confluent presentation for the finitely presented group  $F$  using the Knuth-Bendix completion algorithm. In this section we describe how the user can specify the generator order and the ordering on strings.

**GeneratorOrder**                      SEQENUM                      *Default* :

Give an ordering for the generators. This ordering affects the ordering of words in the alphabet. If not specified the ordering defaults to the order induced by  $F$ 's generators, that is  $[g_1, \dots, g_n]$  where  $g_1, \dots, g_n$  are the generators of  $F$ .

**Ordering**                                      MONSTGELT                      *Default* : "ShortLex"

**Levels**    SEQENUM                      *Default* :

**Weights**    SEQENUM                      *Default* :

**Ordering** := "ShortLex": Use the short-lex ordering on strings. Shorter words come before longer, and for words of equal length lexicographical ordering is used, using the given ordering of the generators.

**Ordering** := "Recursive" | "RtRecursive": Use a recursive ordering on strings. There are various ways to define this. Perhaps the quickest is as follows. Let  $u$  and  $v$  be strings in the generators. If one of  $u$  and  $v$ , say  $v$ , is empty, then  $u \geq v$ . Otherwise, let  $u = u'a$  and  $v = v'b$ , where  $a$  and  $b$  are generators. Then  $u > v$  if and only if one of the following holds:

- (i)  $a = b$  and  $u' > v'$ ;
- (ii)  $a > b$  and  $u > v'$ ;
- (iii)  $b > a$  and  $u' > v$ .

The `RtRecursive` ordering is similar to the `Recursive` ordering, but with  $u = au'$  and  $v = bv'$ . Occasionally one or the other runs significantly quicker, but usually they perform similarly.

`Ordering := "WtLex"`: Use a weighted-lex ordering. `Weights` should be a sequence of non-negative integers, with the  $i$ -th element of `Weights` giving the weight of the  $i$ -th generator. The length of `Weights` must equal the number of generators. The length of words in the generators is then computed by adding up the weights of the generators in the words. Otherwise, ordering is as for short-lex.

`Ordering := "Wreath"`: Use a wreath-product ordering. `Levels` should be a sequence of non-negative integers, with the  $i$ -th element of `Levels` giving the level of the  $i$ -th generator. The length of `Levels` must equal the number of generators. In this ordering, two strings involving generators of the same level are ordered using short-lex, but all strings in generators of a higher level are larger than those involving generators of a lower level. That is not a complete definition; one can be found in [Sim94, pp. 46–50]. Note that the recursive ordering is the special case in which the level of generator number  $i$  is  $i$ .

### Example H74E2

---

A confluent presentation is constructed for an infinite non-hopfian group using the `Recursive` ordering.

```
> F<a, b> := Group< a, b | b^-1*a^2*b=a^3>;
> G := RWSGroup(F : Ordering :="Recursive");
> G;
```

A confluent rewrite group.

Generator Ordering = [ a, a^-1, b, b^-1 ]

Ordering = Recursive.

The reduction machine has 7 states.

The rewrite relations are:

```
a * a^-1 = Id(FG)
a^-1 * a = Id(FG)
b * b^-1 = Id(FG)
b^-1 * b = Id(FG)
a^3 * b^-1 = b^-1 * a^2
a^2 * b = b * a^3
a^-1 * b^-1 = a^2 * b^-1 * a^-2
a^-1 * b = a * b * a^-3
```

**Example H74E3**

---

A confluent presentation of a free nilpotent group of rank 2 and class 2 is constructed by the following code. Note that the lower weight generators (in the sense of nilpotency class) need to come first in the ordering of generators.

```
> FG<a,b,c> := FreeGroup(3);
> F := quo< FG | b*a=a*b*c, c*a=a*c, c*b=b*c>;
> G := RWSGroup(F : Ordering := "Recursive",
>               GeneratorOrder := [c,c^-1,b,b^-1,a,a^-1]);
> G;
```

A confluent rewrite group.

Generator Ordering = [ c, c<sup>-1</sup>, b, b<sup>-1</sup>, a, a<sup>-1</sup> ]

Ordering = Recursive.

The reduction machine has 7 states.

The rewrite relations are:

```
c * c^-1 = Id(FG)
c^-1 * c = Id(FG)
b * b^-1 = Id(FG)
b^-1 * b = Id(FG)
a * a^-1 = Id(FG)
a^-1 * a = Id(FG)
b * a = a * b * c
c * a = a * c
c * b = b * c
c^-1 * a = a * c^-1
c * a^-1 = a^-1 * c
c^-1 * b = b * c^-1
c * b^-1 = b^-1 * c
b^-1 * a = a * b^-1 * c^-1
b^-1 * a^-1 = a^-1 * b^-1 * c
c^-1 * a^-1 = a^-1 * c^-1
c^-1 * b^-1 = b^-1 * c^-1
b * a^-1 = a^-1 * b * c^-1
```

---

**74.2.3 Setting Limits**

In this section we introduce the various parameters used to control the execution of the Knuth-Bendix procedure.

RWSMonoid(F: <i>parameters</i> )
----------------------------------

Attempt to construct a confluent presentation for the finitely presented group  $F$  using the Knuth-Bendix completion algorithm. We present details of the various parameters used to control the execution of the Knuth-Bendix.

<b>MaxRelations</b>	RNGINTELT	<i>Default : 32767</i>
---------------------	-----------	------------------------

Limit the maximum number of reduction equations to **MaxRelations**.

<b>TidyInt</b>	RNGINTELT	<i>Default : 100</i>
----------------	-----------	----------------------

After finding **TidyInt** new reduction equations, the completion procedure interrupts the main process of looking for overlaps, to tidy up the existing set of equations. This will eliminate any redundant equations performing some reductions on their left and right hand sides to make the set as compact as possible. (The point is that equations discovered later often make older equations redundant or too long.)

<b>RabinKarp</b>	TUP	<i>Default :</i>
------------------	-----	------------------

Use the Rabin-Karp algorithm for word-reduction on words having length at least  $l$ , provided that there are at least  $n$  equations, where **RabinKarp** :=  $\langle l, n \rangle$ . This uses less space than the default reduction automaton, but it is distinctly slower, so it should only be used when seriously short of memory. Indeed this option is only really useful for examples in which collapse occurs - i.e. at some intermediate stage of the calculation there is a very large set of equations, which later reduces to a much smaller confluent set. Collapse is not uncommon when analysing pathological presentations of finite groups, and this is one situation where the performance of the Knuth-Bendix algorithm can be superior to that of Todd-Coxeter coset enumeration. The best setting for **RabinKarp** varies from example to example - generally speaking, the smaller  $l$  is, the slower things will be, so set it as high as possible subject to not running out of memory. The number of equations  $n$  should be set to a value greater than the expected final number of equations.

<b>MaxStates</b>	RNGINTELT	<i>Default :</i>
------------------	-----------	------------------

Limit the maximum number of states of the finite state automaton used for word reduction to **MaxStates**. By default there is no limit, and the space allocated is increased dynamically as required. The space needed for the reduction automaton can also be restricted by using the **RabinKarp** parameter. This limit is not usually needed.

<b>MaxReduceLen</b>	RNGINTELT	<i>Default : 32767</i>
---------------------	-----------	------------------------

Limit the maximum allowed length that a word can reach during reduction to **MaxReduceLen**. It is only likely to be exceeded when using the recursive ordering on words. This limit is usually not needed.

<b>ConfNum</b>	RNGINTELT	<i>Default : 500</i>
----------------	-----------	----------------------

If **ConfNum** overlaps are processed and no new equations are discovered, then the overlap searching process is interrupted, and a fast check for confluence performed on the existing set of equations. Doing this too often wastes time, but doing it at the

right moment can also save a lot of time. If `ConfNum = 0`, then the fast confluence check is performed only when the search for overlaps is complete.

**Warning:** Changing the default setting on any of the following parameters may either cause the procedure to terminate without having found a confluent presentation or may change the underlying group.

`MaxStoredLen`                      `TUP`                      *Default :*

Only equations in which the left and right hand sides have lengths at most  $l$  and  $r$ , respectively, where `MaxStoredLen := <l, r>` are kept. Of course this may cause the overlap search to complete on a set of equations that is not confluent. In some examples, particularly those involving collapse (i.e. a large intermediate set of equations, which later simplifies to a small set), it can result in a confluent set being found much more quickly. It is most often useful when using a recursive ordering on words. Another danger with this option is that sometimes discarding equations can result in information being lost, and the monoid defined by the equations changes.

`MaxOverlapLen`                      `RNGINTELT`                      *Default :*

Only overlaps of total length at most `MaxOverlapLen` are processed. Of course this may cause the overlap search to complete on a set of equations that is not confluent.

`Sort`                                      `BOOLELT`                      *Default : false*

`MaxOpLen`                              `RNGINTELT`                      *Default : 0*

If `Sort` is set to `true` then the equations will be sorted in order of increasing length of their left hand sides, rather than the default, which is to leave them in the order in which they were found. `MaxOpLen` should be a non-negative integer. If `MaxOpLen` is positive, then only equations with left hand sides having length at most `MaxOpLen` are output. If `MaxOpLen` is zero, then all equations are sorted by length. Of course, if `MaxOpLen` is positive, there is a danger that the monoid defined by the output equations may be different from the original.

`SetVerbose("KB MAG", v)`

Set the verbose printing level for the Knuth-Bendix completion algorithm. Setting this level allows a user to control how much extra information on the progress of the algorithm is printed. Currently the legal values for  $v$  are 0 to 3 inclusive. Setting  $v$  to 0 corresponds to the ‘-silent’ option of KBMAG in which no extra output is printed. Setting  $v$  to 2 corresponds to the ‘-v’ (verbose) option of KBMAG in which a small amount of extra output is printed. Setting  $v$  to 3 corresponds to the ‘-vv’ (very verbose) option of KBMAG in which a huge amount of diagnostic information is printed.

#### 74.2.4 Accessing Group Information

The functions in this group provide access to basic information stored for a rewrite group  $G$ .

**G . i**

The  $i$ -th defining generator for  $G$ . The integer  $i$  must lie in the range  $[-r, r]$ , where  $r$  is the number of group  $G$ .

**Generators(G)**

A sequence containing the defining generators for  $G$ .

**NumberOfGenerators(G)****Ngens(G)**

The number of defining generators for  $G$ .

**Relations(G)**

A sequence containing the defining relations for  $G$ . The relations will be given between elements of the free group of which  $G$  is a quotient. In these relations the (image of the) left hand side (in  $G$ ) will always be greater than the (image of the) right hand side (in  $G$ ) in the ordering on words used to construct  $G$ .

**NumberOfRelations(G)****Nrels(G)**

The number of relations in  $G$ .

**Ordering(G)**

The ordering of  $G$ .

**Example H74E4**

---

We illustrate the access operations using the following presentation of  $\mathbf{Z} \wr C_2$ .

```
> FG<a,b,t> := FreeGroup(3);
> F := quo< FG | t^2=1, b*a=a*b, t*a*t=b>;
> G<x,y,z> := RWSGroup(F);
> G;
A confluent rewrite group.
Generator Ordering = [ a, a^-1, b, b^-1, t, t^-1 ]
Ordering = ShortLex.
The reduction machine has 6 states.
The rewrite relations are:
a * a^-1 = Id(F)
a^-1 * a = Id(F)
b * b^-1 = Id(F)
b^-1 * b = Id(F)
t^2 = Id(F)
b * a = a * b
t * a = b * t
b^-1 * a = a * b^-1
t * b = a * t
```

```

    b * a^-1 = a^-1 * b
    t * a^-1 = b^-1 * t
    t^-1 = t
    b^-1 * a^-1 = a^-1 * b^-1
    t * b^-1 = a^-1 * t
> G.1;
x
> G.1*G.2;
x * y
> Generators(G);
[ x, y, z ]
> Ngens(G);
3
> Relations(G);
[ a * a^-1 = Id(F), a^-1 * a = Id(F), b * b^-1 = Id(F), b^-1 * b = Id(F), t^2 =
Id(F), b * a = a * b, t * a = b * t, b^-1 * a = a * b^-1, t * b = a * t, b *
a^-1 = a^-1 * b, t * a^-1 = b^-1 * t, t^-1 = t, b^-1 * a^-1 = a^-1 * b^-1, t *
b^-1 = a^-1 * t ]
> Nrels(G);
14
> Ordering(G);
ShortLex

```

---

### 74.3 Properties of a Rewrite Group

**IsConfluent(G)**

Returns true if  $G$  is confluent, false otherwise.

**IsFinite(G)**

Given a confluent group  $G$  return true if  $G$  has finite order and false otherwise. If  $G$  does have finite order also return the order of  $G$ .

**Order(G)**

**#G**

The order of the group  $G$  as an integer. If the order of  $G$  is known to be infinite, the symbol  $\infty$  is returned.

**Example H74E5**

We construct the Weyl group  $E_8$ .

```
> FG<a,b,c,d,e,f,g,h> := FreeGroup(8);
> Q := quo< FG | a^2=1, b^2=1, c^2=1, d^2=1, e^2=1, f^2=1, g^2=1,
>   h^2=1, b*a*b=a*b*a, c*a=a*c, d*a=a*d, e*a=a*e, f*a=a*f,
>   g*a=a*g, h*a=a*h, c*b*c=b*c*b, d*b=b*d, e*b=b*e, f*b=b*f,
>   g*b=b*g, h*b=b*h, d*c*d=c*d*c, e*c=e*c, f*c=c*f,
>   g*c=c*g, h*c=c*h, e*d=d*e, f*d=d*f, g*d=d*g, h*d=d*h,
>   f*e*f=e*f*e, g*e=e*g, h*e=e*h, g*f*g=f*g*f, h*f=f*h,
>   h*g*h=g*h*g>;
> G := RWSGroup(Q);
> IsConfluent(G);
true
> IsFinite(G);
true 696729600
```

So the group is finite of order 696,729,600.

**Example H74E6**

We construct a 2-generator 2-relator group and use the order function to show that the group is infinite. The symbol `Infinity`, returned by `Order`, indicates that the group has infinite order.

```
> G := Group< x, y | x^2 = y^2, x*y*x = y*x*y >;
> R := RWSGroup(G);
> print Order(G);
Infinity
```

## 74.4 Arithmetic with Words

### 74.4.1 Construction of a Word

Identity(G)
-------------

Id(G)
-------

G ! 1
-------

Construct the identity word in  $G$ .

G ! [ $i_1, \dots, i_s$ ]
---------------------------

Given a rewrite group  $G$  defined on  $r$  generators and a sequence  $[i_1, \dots, i_s]$  of integers lying in the range  $[-r, r]$ , excluding 0, construct the word

$$G.|i_1|^{\epsilon_1} * G.|i_2|^{\epsilon_2} * \dots * G.|i_s|^{\epsilon_s}$$

where  $\epsilon_j$  is +1 if  $i_j$  is positive, and -1 if  $i_j$  is negative. The resulting word is reduced using the reduction machine associated with  $G$ .

Parent( $w$ )
---------------

The parent group  $G$  for the word  $w$ .

---

**Example H74E7**

We construct the Fibonacci group  $F(2, 7)$ , and its identity.

```
> FG<a,b,c,d,e,f,g> := FreeGroup(7);
> F := quo< FG | a*b=c, b*c=d, c*d=e, d*e=f, e*f=g, f*g=a, g*a=b>;
> G := RWSGroup(F : TidyInt := 1000);
> Id(G);
Id(G)
> G!1;
Id(G)
> G![1,2];
G.3
```

---

### 74.4.2 Element Operations

Having constructed a rewrite group  $G$  one can perform arithmetic with words in  $G$ . Assuming we have  $u, v \in G$  then the product  $u * v$  will be computed as follows:

- (i) the product  $w = u * v$  is formed as a product in the appropriate free group.
- (ii)  $w$  is reduced using the reduction machine associated with  $G$ .

If  $G$  is confluent, then  $w$  will be the unique minimal word that represents  $u * v$  under the ordering of  $G$ . If  $G$  is not confluent, then there are some pairs of words which are equal in  $G$ , but which reduce to distinct words, and hence  $w$  will not be a unique normal form. Note that:

- (i) reduction of  $w$  can cause an increase in the length of  $w$ . At present there is an internal limit on the length of a word – if this limit is exceeded during reduction an error will be raised. Hence any word operation involving reduction can fail.
- (ii) the implementation is designed more with speed of execution in mind than with minimizing space requirements; thus, the reduction machine is always used to carry out word reduction, which can be space-consuming, particularly when the number of generators is large.

$u * v$
---------

Given words  $w$  and  $v$  belonging to a common group, return their product.

$u / v$
---------

Given words  $w$  and  $v$  belonging to a common group, return the product of the word  $u$  by the inverse of the word  $v$ , i.e. the word  $u * v^{-1}$ .

$u \wedge n$
--------------

The  $n$ -th power of the word  $w$ .

`u ~ v`

Given words  $w$  and  $v$  belonging to a common group, return the conjugate of the word  $u$  by the word  $v$ , i.e. the word  $v^{-1} * u * v$ .

`Inverse(w)`

The inverse of the word  $w$ .

`(u, v)`

Given words  $w$  and  $v$  belonging to a common group, return the commutator of the words  $u$  and  $v$ , i.e., the word  $u^{-1}v^{-1}uv$ .

`(u1, ..., ur)`

Given  $r$  words  $u_1, \dots, u_r$  belonging to a common group, return their commutator. Commutators are *left-normed*, so they are evaluated from left to right.

`u eq v`

Given words  $w$  and  $v$  belonging to the same group, return **true** if  $w$  and  $v$  reduce to the same normal form, **false** otherwise. If  $G$  is confluent this tests for equality. If  $G$  is non-confluent then two words which are the same may not reduce to the same normal form.

`u ne v`

Given words  $w$  and  $v$  belonging to the same group, return **false** if  $w$  and  $v$  reduce to the same normal form, **true** otherwise. If  $G$  is confluent this tests for non-equality. If  $G$  is non-confluent then two words which are the same may reduce to different normal forms.

`IsId(w)``IsIdentity(w)`

Returns **true** if the word  $w$  is the identity word.

`#u`

The length of the word  $w$ .

`ElementToSequence(u)``Eltseq(u)`

The sequence  $Q$  obtained by decomposing the element  $u$  of a rewrite group into its constituent generators and generator inverses. Suppose  $u$  is a word in the rewrite group  $G$ . Then, if  $u = G.i_1^{e_1} \dots G.i_m^{e_m}$ , with each  $e_i = \pm 1$ , then  $Q[j] = i_j$  if  $e_j = +1$  and  $Q[j] = -i_j$  if  $e_j = -1$ , for  $j = 1, \dots, m$ .

**Example H74E8**

---

We illustrate the word operations by applying them to elements of the Fibonacci group  $F(2, 5)$ .

```

> FG<a,b,c,d,e> := FreeGroup(5);
> F := quo< FG | a*b=c, b*c=d, c*d=e, d*e=a, e*a=b>;
> G<a,b,c,d,e> := RWSGroup(F);
> a*b^-1;
e^-1
> a/b;
e^-1
> (c*d)^4;
a
> a^b, b^-1*a*b;
a a
> a^-2,
> Inverse(a)^2;
d d
> c^-1*d^-1*c*d eq (c,d);
true
> IsIdentity(a*b*c^-1);
true
> #(c*d);
1

```

---

## 74.5 Operations on the Set of Group Elements

`Random(G, n)`

A random word of length at most  $n$  in the generators of  $G$ .

`Random(G)`

A random word (of length at most the order of  $G$ ) in the generators of  $G$ .

`Representative(G)`

`Rep(G)`

An element chosen from  $G$ .

Set( $G, a, b$ )
------------------

Search

MONSTGELT

Default : "DFS"

Create the set of reduced words,  $w$ , in  $G$  with  $a \leq \text{length}(w) \leq b$ . If **Search** is set to "DFS" (depth-first search) then words are enumerated in lexicographical order. If **Search** is set to "BFS" (breadth-first-search) then words are enumerated in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker. Since the result is a set the words may not appear in the resultant set in the search order specified (although internally they will be enumerated in this order).

Set( $G$ )
------------

Search

MONSTGELT

Default : "DFS"

Create the set of reduced words that is the carrier set of  $G$ . If **Search** is set to "DFS" (depth-first search) then words are enumerated in lexicographical order. If **Search** is set to "BFS" (breadth-first-search) then words are enumerated in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker. Since the result is a set the words may not appear in the resultant set in the search order specified (although internally they will be enumerated in this order).

Seq( $G, a, b$ )
------------------

Search

MONSTGELT

Default : "DFS"

Create the sequence  $S$  of reduced words,  $w$ , in  $G$  with  $a \leq \text{length}(w) \leq b$ . If **Search** is set to "DFS" (depth-first search) then words will appear in  $S$  in lexicographical order. If **Search** is set to "BFS" (breadth-first-search) then words will appear in  $S$  in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker.

Seq( $G$ )
------------

Search

MONSTGELT

Default : "DFS"

Create a sequence  $S$  of reduced words in the carrier set of  $G$ . If **Search** is set to "DFS" (depth-first search) then words will appear in  $S$  in lexicographical order. If **Search** is set to "BFS" (breadth-first-search) then words will appear in  $S$  in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker.

---

### Example H74E9

We construct the group  $D_{22}$ , together with a representative word from the group, a random word and a random word of length at most 5 from the group, and the set of elements of the group.

```
> FG<a,b,c,d,e,f> := FreeGroup(6);
> Q := quo< FG | a*c^-1*a^-1*d=1, b*f*b^-1*e^-1=1, c*e*c^-1*d^-1=1,
>                d*f^-1*d^-1*a=1, e*b*e^-1*a^-1=1, f*c^-1*f^-1*b^-1=1>;
```

```

> G<a,b,c,d,e,f> := RWSGroup(Q);
> Representative(G);
Id(G)
> Random(G);
b
> Random(G, 5);
a * d * b
> Set(G);
{ a * d * b, a * b, a * b * e, a * c, a * d, d * b, b * e, a * b * a,
  a * b * d, b * a, a * c * e, Id(G), b * d, c * e, e, f, a, a * e, b,
  c, a * f, d }
> Seq(G : Search := "DFS");
[ Id(G), a, a * b, a * b * a, a * b * d, a * b * e, a * c, a * c * e,
  a * d, a * d * b, a * e, a * f, b, b * a, b * d, b * e, c, c * e, d,
  d * b, e, f ]

```

---

## 74.6 Homomorphisms

For a general description of homomorphisms, we refer to chapter 16. This section describes some special aspects of homomorphisms whose domain or codomain is a rewrite group.

### 74.6.1 General Remarks

Groups in the category `GrpRWS` currently are accepted as codomains only in some special situations. The most important cases in which a rewrite group can be used as a codomain are group homomorphisms whose domain is in one of the categories `GrpFP`, `GrpGPC`, `GrpRWS` or `GrpAtc`.

### 74.6.2 Construction of Homomorphisms

<code>hom&lt; R -&gt; G   S &gt;</code>
---

Returns the homomorphism from the rewrite group  $R$  to the group  $G$  defined by the expression  $S$  which can be the one of the following:

- (i) A list, sequence or indexed set containing the images of the  $n$  generators  $R.1, \dots, R.n$  of  $R$ . Here, the  $i$ -th element of  $S$  is interpreted as the image of  $R.i$ , i.e. the order of the elements in  $S$  is important.
- (ii) A list, sequence, enumerated set or indexed set, containing  $n$  tuples  $\langle x_i, y_i \rangle$  or arrow pairs  $x_i \rightarrow y_i$ , where  $x_i$  is a generator of  $R$  and  $y_i \in G$  ( $i = 1, \dots, n$ ) and the set  $\{x_1, \dots, x_n\}$  is the full set of generators of  $R$ . In this case,  $y_i$  is assigned as the image of  $x_i$ , hence the order of the elements in  $S$  is not important.

It is the user's responsibility to ensure that the provided generator images actually give rise to a well-defined homomorphism. No checking is performed by the constructor.

Note that it is currently not possible to define a homomorphism by assigning images to the elements of an arbitrary generating set of  $R$ .

## 74.7 Conversion to a Finitely Presented Group

There is a standard way to convert a rewrite group into a finitely presented group using the functions `Relations` and `Simplify`. This is shown in the following example.

### Example H74E10

---

We construct a two generator free abelian group as a rewrite group and then convert it into a finitely presented group.

```
> FG<a,b> := FreeGroup(2);
> F := quo< FG | b^-1*a*b=a >;
> G := RWSGroup(F);
> print G;
A confluent rewrite group.
Generator Ordering = [ a, a^-1, b, b^-1 ]
Ordering = ShortLex.
The reduction machine has 5 states.
The rewrite relations are:
  a * a^-1 = Id(FG)
  a^-1 * a = Id(FG)
  b * b^-1 = Id(FG)
  b^-1 * b = Id(FG)
  b^-1 * a = a * b^-1
  b * a = a * b
  b^-1 * a^-1 = a^-1 * b^-1
  b * a^-1 = a^-1 * b
> P<x,y> := Simplify(quo< FG | Relations(G)>);
> print P;
Finitely presented group P on 2 generators
Generators as words
  x = $.1
  y = $.2
Relations
  (y, x^-1) = Id(P)
```

---

## 74.8 Bibliography

- [Hol97] Derek Holt. *KB MAG – Knuth-Bendix in Monoids and Automatic Groups*. University of Warwick, 1997.
- [Sim94] Charles C. Sims. *Computation with finitely presented groups*. Cambridge University Press, Cambridge, 1994.

# 75 AUTOMATIC GROUPS

<b>75.1 Introduction . . . . . 2357</b>	
75.1.1 Terminology . . . . . 2357	
75.1.2 The Category of Automatic Groups 2357	
75.1.3 The Construction of an Automatic Group . . . . . 2357	
<b>75.2 Creation of Automatic Groups 2358</b>	
75.2.1 Construction of an Automatic Group 2358	
AutomaticGroup(F: -) 2358	
IsAutomaticGroup(F: -) 2358	
75.2.2 Modifying Limits . . . . . 2359	
AutomaticGroup(F: -) 2359	
IsAutomaticGroup(F: -) 2359	
SetVerbose("KBMAG", v) 2360	
75.2.3 Accessing Group Information . . . 2363	
. 2363	
Generators(G) 2363	
NumberOfGenerators(G) 2363	
Ngens(G) 2363	
FPGroup(G) 2364	
WordAcceptor(G) 2364	
WordAcceptorSize(G) 2364	
WordDifferenceAutomaton(G) 2364	
WordDifferenceSize(G) 2364	
WordDifferences(G) 2364	
GeneratorOrder(G) 2364	
<b>75.3 Properties of an Automatic Group . . . . . 2364</b>	
IsFinite(G) 2364	
Order(G) 2364	
# 2364	
<b>75.4 Arithmetic with Words . . . 2366</b>	
75.4.1 Construction of a Word . . . . . 2366	
! 2366	
Identity(G) 2366	
Id(G) 2366	
! 2366	
Parent(w) 2366	
75.4.2 Operations on Elements . . . . . 2367	
* 2367	
/ 2367	
^ 2368	
~ 2368	
Inverse(w) 2368	
(u, v) 2368	
(u <sub>1</sub> , ..., u <sub>r</sub> ) 2368	
eq 2368	
ne 2368	
IsId(w) 2368	
IsIdentity(w) 2368	
# 2368	
ElementToSequence(u) 2368	
Eltseq(u) 2368	
<b>75.5 Homomorphisms . . . . . 2369</b>	
75.5.1 General Remarks . . . . . 2369	
75.5.2 Construction of Homomorphisms . 2370	
hom< > 2370	
<b>75.6 Set Operations . . . . . 2370</b>	
Random(G, n) 2370	
Random(G) 2370	
Representative(G) 2370	
Rep(G) 2370	
Set(G, a, b) 2370	
Set(G) 2371	
Seq(G, a, b) 2371	
Seq(G) 2371	
<b>75.7 The Growth Function . . . . . 2372</b>	
GrowthFunction(G) 2372	
<b>75.8 Bibliography . . . . . 2373</b>	



# Chapter 75

## AUTOMATIC GROUPS

### 75.1 Introduction

Automatic groups provide a Magma level interface to Derek Holt's KBMAG programs, and specifically to KBMAG's automatic groups program *autgroup*. Much of the material in this chapter is based on the KBMAG documentation [Hol97]. Familiarity with the Knuth–Bendix completion procedure and the automata associated with a short-lex automatic group is assumed. Some familiarity with KBMAG would be beneficial.

#### 75.1.1 Terminology

An automatic group  $G$  is a finitely presented group in which various group operations, notably equality between words of  $G$  and word enumeration, are decidable through the use of various automata. The words in the automatic group  $G$  that can be computed in MAGMA are ordered using the short-lex ordering on words (shorter words come before longer, and for words of equal length lexicographical ordering is used, based on the given ordering of the generators).

#### 75.1.2 The Category of Automatic Groups

The family of all automatic groups forms a category. The objects are the automatic groups and the morphisms are group homomorphisms. The MAGMA designation for this category of groups is `GrpAtc`. Elements of a automatic group are designated as `GrpAtcElt`.

#### 75.1.3 The Construction of an Automatic Group

An automatic group  $G$  is constructed in a three-step process:

- (i) We construct a free group  $FG$ .
- (ii) We construct a quotient  $F$  of  $FG$ .
- (iii) We create a monoid presentation for  $F$  and then run procedures which attempt to construct the automata associated with  $G$  and to prove them correct.

These procedures may or may not succeed. Of course, if  $G$  is not an automatic group then they have no chance of succeeding.

## 75.2 Creation of Automatic Groups

### 75.2.1 Construction of an Automatic Group

<code>AutomaticGroup(F: parameters)</code>
--

<code>IsAutomaticGroup(F: parameters)</code>
--

Internally a monoid presentation  $P$  of the group  $F$  is constructed. By default the generators of  $P$  are taken to be  $g_1, g_1^{-1}, \dots, g_n, g_n^{-1}$  where  $g_1, \dots, g_n$  are the generators of  $F$ . The relations of  $P$  are taken to be the relations of  $F$ . The trivial relations between the generators and their inverses are also added. The word ordering is the short-lex ordering. The Knuth–Bendix completion procedure for monoids is now run on  $P$  to calculate the word difference automata corresponding to the generated equations, which are then used to calculate the finite state automata associated with a short-lex automatic group. In successful cases these automata are proved correct in the final step.

If the procedure succeeds the result will be an automatic group,  $G$ , containing four automata. These are the first and second word-difference machines, the word acceptor, and the word multiplier. The form `AutomaticGroup` returns an automatic group while the form `IsAutomaticGroup` returns the boolean value *true* and the automatic group. If the procedure fails, the first form does not return a value while the second returns the boolean value *false*.

For simple examples, the algorithms work quickly, and do not require much space. For more difficult examples, the algorithms are often capable of completing successfully, but they can sometimes be expensive in terms of time and space requirements. Another point to be borne in mind is that the algorithms sometimes produce temporary disk files which the user does not normally see (because they are automatically removed after use), but can occasionally be very large. These files are stored in the `/tmp` directory. If you interrupt a running automatic group calculation you must remove these temporary files yourself.

As the Knuth–Bendix procedure will more often than not run forever, some conditions must be specified under which it will stop. These take the form of limits that are placed on certain variables, such as the number of reduction relations. If any of these limits are exceeded during a run of the completion procedure it will fail, returning a non-confluent automatic group. The optimal values for these limits varies from example to example. Some of these limits may be specified by setting parameters (see the next section). In particular, if a first attempt to compute the automatic structure of a group fails, it should be run again with the parameter `Large` (or `Huge`) set to `true`.

---

#### Example H75E1

We construct the automatic structure for the fundamental group of the torus. Since a generator ordering is not specified, the default generator ordering,  $[a, a^{-1}, b, b^{-1}, c, c^{-1}, d, d^{-1}]$ , is used.

```
> FG<a,b,c,d> := FreeGroup(4);
```

```

> F := quo< FG | a^-1*b^-1*a*b=d^-1*c^-1*d*c>;
> f, G := IsAutomaticGroup(F);
Running Knuth-Bendix with the following parameter values
MaxRelations = 200
MaxStates = 0
TidyInt = 20
MaxWdiffs = 512
HaltingFactor = 100
MinTime = 5
#Halting with 118 equations.
#First word-difference machine with 33 states computed.
#Second word-difference machine with 33 states computed.
#System is confluent, or halting factor condition holds.
#Word-acceptor with 36 states computed.
#General multiplier with 104 states computed.
#Validity test on general multiplier succeeded.
#General length-2 multiplier with 220 states computed.
#Checking inverse and short relations.
#Checking relation: _8*_6*_7*_5 = _2*_4*_1*_3
#Axiom checking succeeded.
> G;
An automatic group.
Generator Ordering = [ a, a^-1, b, b^-1, c, c^-1, d, d^-1 ]
The second word difference machine has 33 states.
The word acceptor has 36 states.

```

## 75.2.2 Modifying Limits

In this section we describe the various parameters used to control the execution of the procedures employed to determine the automatic structure.

AutomaticGroup(F: <i>parameters</i> )
---------------------------------------

IsAutomaticGroup(F: <i>parameters</i> )
---

Attempt to construct an automatic structure for the finitely presented group  $F$  (see the main entry). We now present details of the various parameters used to control the execution of the procedures.

Large	BOOLELT	<i>Default : false</i>
-------	---------	------------------------

If **Large** is set to **true** large hash tables are used internally. Also the Knuth-Bendix algorithm is run with larger parameters, specifically **TidyInt** is set to 500, **MaxRelations** is set to 262144, **MaxStates** is set to unlimited, **HaltingFactor** is set to 100, **MinTime** is set to 20 and **ConfNum** is set to 0. It is advisable to use this option only after having first tried without it, since it will result in much longer execution times for easy examples.

Huge	BOOLELT	<i>Default : false</i>
------	---------	------------------------

Setting `Huge` to `true` doubles the size of the hash tables and `MaxRelations` over the `Large` parameter. As with the `Large` parameter, it is advisable to use this option only after having first tried without it.

`MaxRelations`                      `RNGINTELT`                      *Default* : 200

Limit the maximum number of reduction equations to `MaxRelations`.

`TidyInt`                              `RNGINTELT`                      *Default* : 20

After finding  $n$  new reduction equations, the completion procedure interrupts the main process of looking for overlaps, to tidy up the existing set of equations. This will eliminate any redundant equations performing some reductions on their left and right hand sides to make the set as compact as possible. (The point is that equations discovered later often make older equations redundant or too long.) The word-differences arising from the equations are calculated after each such tidying and the number reported if verbose printing is on. The best strategy in general is to try a small value of `TidyInt` first and, if that is not successful, try increasing it. Large values such as 1000 work best in really difficult examples.

`GeneratorOrder`                      `SEQENUM`                              *Default* :

Give an ordering for the generators of  $P$ . This ordering affects the ordering of words in the alphabet. If not specified, the ordering defaults to  $[g_1, g_1^{-1}, \dots, g_n, g_n^{-1}]$  where  $g_1, \dots, g_n$  are the generators of  $F$ .

`MaxWordDiffs`                      `RNGINTELT`                      *Default* :

Limit the maximum number of word differences to `MaxWordDiffs`. The default behaviour is to increase the number of allowed word differences dynamically as required, and so usually one does not need to set this option.

`HaltingFactor`                      `RNGINTELT`                      *Default* : 100

`MinTime`                              `RNGINTELT`                      *Default* : 5

These options are experimental halting options. `HaltingFactor` is a positive integer representing a percentage. After each tidying it is checked whether both the number of equations and the number of states have increased by more than `HaltingFactor` percent since the number of word-differences was last less than what it is now. If so the program halts. A sensible value seems to be 100, but occasionally a larger value is necessary. If the `MinTime` option is also set then halting only occurs if at least `MinTime` seconds of cpu-time have elapsed altogether. This is sometimes necessary to prevent very early premature halting. It is not very satisfactory, because of course the cpu-time depends heavily on the particular computer being used, but no reasonable alternative has been found yet.

<code>SetVerbose("KBMAG", v)</code>
-------------------------------------

Set the verbose printing level for the Knuth–Bendix completion algorithm. Setting this level allows a user to control how much extra information on the progress of the algorithm is printed. Currently the legal values for  $v$  are 0 to 3 inclusive. Setting  $v$  to 0 corresponds to the ‘-silent’ option of KBMAG in which no extra output is

printed. Setting  $v$  to 2 corresponds to the ‘-v’ (verbose) option of KBMAG in which a small amount of extra output is printed. Setting  $v$  to 3 corresponds to the ‘-vv’ (very verbose) option of KBMAG in which a huge amount of diagnostic information is printed.

### Example H75E2

---

We attempt to construct an automatic structure for one of Listing’s knot groups.

```
> F := Group< d, f | f*d*f^-1*d*f*d^-1*f^-1*d*f^-1*d^-1=1>;
> b, G := IsAutomaticGroup(F);
Running Knuth-Bendix with the following parameter values
MaxRelations = 200
MaxStates = 0
TidyInt = 20
MaxWdiffs = 512
HaltingFactor = 100
MinTime = 5
#Maximum number of equations exceeded.
#Halting with 195 equations.
#First word-difference machine with 45 states computed.
#Second word-difference machine with 53 states computed.
> b;
false;
```

So this attempt has failed. We run the `IsAutomaticGroup` function again setting `Large` to `true`. This time we succeed.

```
> f, G := IsAutomaticGroup(F : Large := true);
Running Knuth-Bendix with the following parameter values
MaxRelations = 262144
MaxStates = 0
TidyInt = 500
MaxWdiffs = 512
HaltingFactor = 100
MinTime = 5
#Halting with 3055 equations.
#First word-difference machine with 49 states computed.
#Second word-difference machine with 61 states computed.
#System is confluent, or halting factor condition holds.
#Word-acceptor with 101 states computed.
#General multiplier with 497 states computed.
#Multiplier incorrect with generator number 3.
#General multiplier with 509 states computed.
#Multiplier incorrect with generator number 3.
#General multiplier with 521 states computed.
#Multiplier incorrect with generator number 3.
#General multiplier with 525 states computed.
#Validity test on general multiplier succeeded.
```

```

#General length-2 multiplier with 835 states computed.
#Checking inverse and short relations.
#Checking relation:  _3*_1*_4*_1*_3 = _1*_3*_2*_3*_1
#Axiom checking succeeded.
> G;
An automatic group.
Generator Ordering = [ d, d^-1, f, f^-1 ]
The second word difference machine has 89 states.
The word acceptor has 101 states.

```

### Example H75E3

---

We construct the automatic group corresponding to the fundamental group of the trefoil knot. A generator order is specified.

```

> F<a, b> := Group< a, b | b*a^-1*b=a^-1*b*a^-1>;
> f, G := IsAutomaticGroup(F: GeneratorOrder := [a,a^-1, b, b^-1]);
Running Knuth-Bendix with the following parameter values
MaxRelations = 200
MaxStates = 0
TidyInt = 20
MaxWdiffs = 512
HaltingFactor = 100
MinTime = 5
#Halting with 83 equations.
#First word-difference machine with 15 states computed.
#Second word-difference machine with 17 states computed.
#System is confluent, or halting factor condition holds.
#Word-acceptor with 15 states computed.
#General multiplier with 67 states computed.
#Multiplier incorrect with generator number 4.
#General multiplier with 71 states computed.
#Validity test on general multiplier succeeded.
#General length-2 multiplier with 361 states computed.
#Checking inverse and short relations.
#Checking relation:  _3*_2*_3 = _2*_3*_2
#Axiom checking succeeded.
> G;
An automatic group.
Generator Ordering = [ a, a^-1, b, b^-1 ]
The second word difference machine has 21 states.
The word acceptor has 15 states.

```

---

### 75.2.3 Accessing Group Information

The functions in this group provide access to basic information stored for an automatic group  $G$ .

`G . i`

The  $i$ -th defining generator for  $G$ . The integer  $i$  must lie in the range  $[-r, r]$ , where  $r$  is the number of group  $G$ .

`Generators(G)`

A sequence containing the defining generators for  $G$ .

`NumberOfGenerators(G)`

`Ngens(G)`

The number of defining generators for  $G$ .

---

#### Example H75E4

We illustrate the access operations using the Von Dyck (2,3,5) group (isomorphic to  $A_5$ ).

```
> F<a,b> := FreeGroup(2);
> Q := quo<F | a*a=1, b*b=b^-1, a*b^-1*a*b^-1*a=b*a*b*a*b>;
> f, G<a,b> := IsAutomaticGroup(Q);
> G;
An automatic group.
Generator Ordering = [ a, a^-1, b, b^-1 ]
The second word difference machine has 33 states.
The word acceptor has 28 states.
> print G.1*G.2;
a * b
> print Generators(G);
[ a, b ]
> print Ngens(G);
2
> rels := Relations(G);
> print rels[1];
Q.2 * Q.2^-1 = Id(Q)
> print rels[2];
Q.2^-1 * Q.2 = Id(Q)
> print rels[3];
Q.1^2 = Id(Q)
> print rels[4];
Q.2^2 = Q.2^-1
> print Nrels(G);
18
> print Ordering(G);
ShortLex
```

---

**FPGroup(G)**

Returns the finitely presented group  $F$  used in the construction of  $G$ , and the isomorphism from  $F$  to  $G$ .

**WordAcceptor(G)**

A record describing the word acceptor automaton stored in  $G$ .

**WordAcceptorSize(G)**

The number of states of the word acceptor automaton stored in  $G$ , and the size of the alphabet of this automaton.

**WordDifferenceAutomaton(G)**

A record describing the word difference automaton stored in  $G$ .

**WordDifferenceSize(G)**

The number of states of the 2nd word difference automaton stored in  $G$ , and the size of the alphabet of this automaton.

**WordDifferences(G)**

The labels of the states of the word difference automaton stored in  $G$ . The result is a sequence of elements of the finitely presented group used in the construction of  $G$ .

**GeneratorOrder(G)**

The value of the `GeneratorOrder` parameter used in the construction of  $G$ . The result is a sequence of generators and their inverses from the finitely presented group used in the construction of  $G$ .

### 75.3 Properties of an Automatic Group

**IsFinite(G)**

Given an automatic group  $G$  return `true` if  $G$  has finite order and `false` otherwise. If  $G$  does have finite order also return the order of  $G$ .

**Order(G)**

**#G**

The order of the group  $G$  as an integer. If the order of  $G$  is known to be infinite, the symbol  $\infty$  is returned.

**Example H75E5**

---

We construct the group  $\mathbf{Z} \wr C_2$  and compute its order. The result of `Infinity` indicates that the group has infinite order.

```
> F<a,b,t> := FreeGroup(3);
> Q := quo< F | t^2=1, b*a=a*b, t*a*t=b>;
> f, G := IsAutomaticGroup(Q);
Running Knuth-Bendix with the following parameter values
MaxRelations = 200
MaxStates = 0
TidyInt = 20
MaxWdiffs = 512
HaltingFactor = 100
MinTime = 5
#System is confluent.
#Halting with 14 equations.
#First word-difference machine with 14 states computed.
#Second word-difference machine with 14 states computed.
#System is confluent, or halting factor condition holds.
#Word-acceptor with 6 states computed.
#General multiplier with 27 states computed.
#Validity test on general multiplier succeeded.
#Checking inverse and short relations.
#Axiom checking succeeded.
> Order(G);
Infinity
```

**Example H75E6**

---

We construct a three fold cover of  $A_6$  and check whether it has finite order.

```
> FG<a,b> := FreeGroup(2);
> F := quo< FG | a^3=1, b^3=1, (a*b)^4=1, (a*b^-1)^5=1>;
> f, G := IsAutomaticGroup(F : GeneratorOrder := [a,b,a^-1,b^-1]);
Running Knuth-Bendix with the following parameter values
MaxRelations = 200
MaxStates = 0
TidyInt = 20
MaxWdiffs = 512
HaltingFactor = 100
MinTime = 5
#System is confluent.
#Halting with 183 equations.
#First word-difference machine with 289 states computed.
#Second word-difference machine with 360 states computed.
#System is confluent, or halting factor condition holds.
#Word-acceptor with 314 states computed.
#General multiplier with 1638 states computed.
```

```

#Multiplier incorrect with generator number 4.
#General multiplier with 1958 states computed.
#Multiplier incorrect with generator number 2.
#General multiplier with 2020 states computed.
#Multiplier incorrect with generator number 1.
#General multiplier with 2038 states computed.
#Validity test on general multiplier succeeded.
#General length-2 multiplier with 4252 states computed.
#Checking inverse and short relations.
#Checking relation:  _1*_2*_1*_2 = _4*_3*_4*_3
#Checking relation:  _1*_4*_1*_4*_1 = _2*_3*_2*_3*_2
#Axiom checking succeeded.
> IsFinite(G);
true 1080
> isf, ord := IsFinite(G);
> isf, ord;
true 1080

```

---

## 75.4 Arithmetic with Words

### 75.4.1 Construction of a Word

$G ! [ i_1, \dots, i_s ]$
---------------------------

Given an automatic group  $G$  defined on  $r$  generators and a sequence  $[i_1, \dots, i_s]$  of integers lying in the range  $[-r, r]$ , excluding 0, construct the word

$$G \cdot |i_1|^{\epsilon_1} * G \cdot |i_2|^{\epsilon_2} * \dots * G \cdot |i_s|^{\epsilon_s}$$

where  $\epsilon_j$  is +1 if  $i_j$  is positive, and -1 if  $i_j$  is negative. The word will be returned in reduced form.

Identity( $G$ )
-----------------

Id( $G$ )
-----------

$G ! 1$
---------

Construct the identity word in the automatic group  $G$ .

Parent( $w$ )
---------------

The parent group  $G$  for the word  $w$ .

**Example H75E7**

---

We construct some words in a two-generator two-relator group.

```
> F<a, b> := Group< a, b | a^2 = b^2, a*b*a = b*a*b >;
> f, G<a, b> := IsAutomaticGroup(F);
> G;
```

An automatic group.

```
Generator Ordering = [ $.1, $.1^-1, $.2, $.2^-1 ]
```

The second word difference machine has 11 states.

The word acceptor has 8 states.

```
> Id(G);
Id(G)
> print G!1;
Id(G)
> a*b*a*b^3;
a^4 * b * a
> G![1,2,1,2,2,2];
a^4 * b * a
```

---

**75.4.2 Operations on Elements**

Having constructed an automatic group  $G$  one can perform arithmetic with words in  $G$ . Assuming we have  $u, v \in G$  then the product  $u * v$  will be computed as follows:

- (i) The product  $w = u * v$  is formed as a product in the appropriate free group.
- (ii)  $w$  is reduced using the second word difference machine associated with  $G$ .

Note that:

- (i) Reduction of  $w$  can cause an increase in the length of  $w$ . At present there is an internal limit on the length of a word – if this limit is exceeded during reduction an error will be raised. Hence any word operation involving reduction can fail.
- (ii) The implementation is designed more with speed of execution in mind than with minimizing space requirements; thus, the reduction machine is always used to carry out word reduction, which can be space-consuming, particularly when the number of generators is large.

$u * v$
---------

Given words  $w$  and  $v$  belonging to a common group, return their product.

$u / v$
---------

Given words  $w$  and  $v$  belonging to a common group, return the product of the word  $u$  by the inverse of the word  $v$ , i.e. the word  $u * v^{-1}$ .

`u ^ n`

The  $n$ -th power of the word  $w$ .

`u ^ v`

Given words  $w$  and  $v$  belonging to a common group, return the conjugate of the word  $u$  by the word  $v$ , i.e. the word  $v^{-1} * u * v$ .

`Inverse(w)`

The inverse of the word  $w$ .

`(u, v)`

Given words  $w$  and  $v$  belonging to a common group, return the commutator of the words  $u$  and  $v$ , i.e., the word  $u^{-1}v^{-1}uv$ .

`(u1, ..., ur)`

Given  $r$  words  $u_1, \dots, u_r$  belonging to a common group, return their commutator. Commutators are *left-normed*, so they are evaluated from left to right.

`u eq v`

Given words  $w$  and  $v$  belonging to the same group, return **true** if  $w$  and  $v$  reduce to the same normal form, **false** otherwise. If  $G$  is confluent this tests for equality. If  $G$  is non-confluent then two words which are the same may not reduce to the same normal form.

`u ne v`

Given words  $w$  and  $v$  belonging to the same group, return **false** if  $w$  and  $v$  reduce to the same normal form, **true** otherwise. If  $G$  is confluent this tests for non-equality. If  $G$  is non-confluent then two words which are the same may reduce to different normal forms.

`IsId(w)``IsIdentity(w)`

Returns **true** if the word  $w$  is the identity word.

`#u`

The length of the word  $w$ .

`ElementToSequence(u)``Eltseq(u)`

The sequence  $Q$  obtained by decomposing the element  $u$  of a rewrite group into its constituent generators and generator inverses. Suppose  $u$  is a word in the rewrite group  $G$ . Then, if  $u = G.i_1^{e_1} \dots G.i_m^{e_m}$ , with each  $e_i = \pm 1$ , then  $Q[j] = i_j$  if  $e_j = +1$  and  $Q[j] = -i_j$  if  $e_j = -1$ , for  $j = 1, \dots, m$ .

**Example H75E8**

---

We illustrate the word operations by applying them to elements of the fundamental group of a 3-manifold.

We illustrate the word operations by applying them to elements of a free group of rank two (with lots of redundant generators).

```
> FG<a,b,c,d,e> := FreeGroup(5);
> F := quo< FG | a*d*d=1, b*d*d*d=1, c*d*d*d*d*d*e*e*e=1>;
> f, G<a,b,c,d,e> := IsAutomaticGroup(F);
> G;
An automatic group.
Generator Ordering = [ a, a^-1, b, b^-1, c, c^-1, d, d^-1, e, e^-1 ]
The second word difference machine has 41 states.
The word acceptor has 42 states.
> print a*d;
d^-1
> print a/(d^-1);
d^-1
> print c*d^5*e^2;
e^-1
> print a^b, b^-1*a*b;
a a
> print (a*d)^-2, Inverse(a*d)^2;
a^-1 a^-1
> print c^-1*d^-1*c*d eq (c,d);
true
> print IsIdentity(b*d^3);
true
> print #(c*d*d*d*d*d*e*e);
1
```

---

## 75.5 Homomorphisms

For a general description of homomorphisms, we refer to chapter 16. This section describes some special aspects of homomorphisms whose domain or codomain is an automatic group.

### 75.5.1 General Remarks

Groups in the category `GrpAtc` currently are accepted as codomains only in some special situations. The most important cases in which an automatic group can be used as a codomain are group homomorphisms whose domain is in one of the categories `GrpFP`, `GrpGPC`, `GrpRWS` or `GrpAtc`.

### 75.5.2 Construction of Homomorphisms

`hom< A -> G | S >`

Returns the homomorphism from the automatic group  $A$  to the group  $G$  defined by the expression  $S$  which can be the one of the following:

- (i) A list, sequence or indexed set containing the images of the  $n$  generators  $A.1, \dots, A.n$  of  $A$ . Here, the  $i$ -th element of  $S$  is interpreted as the image of  $A.i$ , i.e. the order of the elements in  $S$  is important.
- (ii) A list, sequence, enumerated set or indexed set, containing  $n$  tuples  $\langle x_i, y_i \rangle$  or arrow pairs  $x_i \rightarrow y_i$ , where  $x_i$  is a generator of  $A$  and  $y_i \in G$  ( $i = 1, \dots, n$ ) and the set  $\{x_1, \dots, x_n\}$  is the full set of generators of  $A$ . In this case,  $y_i$  is assigned as the image of  $x_i$ , hence the order of the elements in  $S$  is not important.

It is the user's responsibility to ensure that the provided generator images actually give rise to a well-defined homomorphism. No checking is performed by the constructor.

Note that it is currently not possible to define a homomorphism by assigning images to the elements of an arbitrary generating set of  $A$ .

### 75.6 Set Operations

In this section we describe functions which allow the user to enumerate various sets of elements of an automatic group  $G$ .

`Random(G, n)`

A random word of length at most  $n$  in the generators of  $G$ .

`Random(G)`

A random word (of length at most the order of  $G$ ) in the generators of  $G$ .

`Representative(G)`

`Rep(G)`

An element chosen from  $G$ .

`Set(G, a, b)`

**Search**

MONSTGELT

*Default* : "DFS"

Create the set of words,  $w$ , in  $G$  with  $a \leq \text{length}(w) \leq b$ . If **Search** is set to "DFS" (depth-first search) then words are enumerated in lexicographical order. If **Search** is set to "BFS" (breadth-first-search) then words are enumerated in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker. Since the result is a set the words may not appear in the resultant set in the search order specified (although internally they will be enumerated in this order).

**Set(G)****Search**

MONSTGELT

*Default : "DFS"*

Create the set of words that is the carrier set of  $G$ . If **Search** is set to "DFS" (depth-first search) then words are enumerated in lexicographical order. If **Search** is set to "BFS" (breadth-first-search) then words are enumerated in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker. Since the result is a set the words may not appear in the resultant set in the search order specified (although internally they will be enumerated in this order).

**Seq(G, a, b)****Search**

MONSTGELT

*Default : "DFS"*

Create the sequence  $S$  of words,  $w$ , in  $G$  with  $a \leq \text{length}(w) \leq b$ . If **Search** is set to "DFS" (depth-first search) then words will appear in  $S$  in lexicographical order. If **Search** is set to "BFS" (breadth-first-search) then words will appear in  $S$  in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker.

**Seq(G)****Search**

MONSTGELT

*Default : "DFS"*

Create a sequence  $S$  of words from the carrier set of  $G$ . If **Search** is set to "DFS" (depth-first search) then words will appear in  $S$  in lexicographical order. If **Search** is set to "BFS" (breadth-first-search) then words will appear in  $S$  in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker.

**Example H75E9**

We construct the group  $D_{22}$ , together with a representative word from the group, a random word and a random word of length at most 5 from the group, and the set of elements of the group.

```
> FG<a,b,c,d,e,f> := FreeGroup(6);
> F := quo< FG | a*c^-1*a^-1*d=1, b*f*b^-1*e^-1=1,
>           c*e*c^-1*d^-1=1, d*f^-1*d^-1*a=1,
>           e*b*e^-1*a^-1=1, f*c^-1*f^-1*b^-1=1 >;
> f, G<a,b,c,d,e,f> := IsAutomaticGroup(F);
Running Knuth-Bendix with the following parameter values
MaxRelations = 200
MaxStates    = 0
TidyInt      = 20
MaxWdiffs   = 512
HaltingFactor = 100
MinTime      = 5
#System is confluent.
#Halting with 41 equations.
#First word-difference machine with 16 states computed.
```

```

#Second word-difference machine with 17 states computed.
#System is confluent, or halting factor condition holds.
#Word-acceptor with 6 states computed.
#General multiplier with 58 states computed.
#Validity test on general multiplier succeeded.
#Checking inverse and short relations.
#Axiom checking succeeded.
> Representative(G);
Id(G)
> Random(G);
a*c;
> Random(G, 5);
a * b
> Set(G);
{ a * d * b, a * b, a * b * e, a * c, a * d, d * b, b * e,
  a * b * a, a * b * d, b * a, a * c * e, Id(G), b * d, c * e,
  e, f, a, a * e, b, c, a * f, d }
> Seq(G : Search := "BFS");
[ Id(G), a, b, c, d, e, f, a * b, a * c, a * d, a * e, a * f,
  b * a, b * d, b * e, c * e, d * b, a * b * a, a * b * d,
  a * b * e, a * c * e, a * d * b ]

```

---

## 75.7 The Growth Function

GrowthFunction(G)
-------------------

Primes

SEQENUM

Default : []

Compute the growth function of the word acceptor automaton associated with  $G$ . The growth function of a DFA,  $A$ , is the quotient of two integral polynomials in a single variable  $x$ . The coefficient of  $x^n$  in the Taylor expansion (about 0) of this quotient is equal to the number of words of length  $n$  accepted by  $A$ . That is, the result is a closed form for this generating function.

The algorithm is by Derek Holt. The **Primes** parameter is no longer used, but is kept for backward compatibility. It may be removed in future releases.

### Example H75E10

---

We construct a dihedral group of order 10 and compute the growth function of its word acceptor. As the group is finite, the result will be a polynomial. Note here that the **R!f** is only necessary to get pretty printing, specifically to ensure that  $f$  is printed in the variable  $x$ .

```

> R<x> := RationalFunctionField(Integers());
> FG<a,b> := FreeGroup(2);
> Q := quo< FG | a^5, b^2, a^b = a^-1>;
> G := AutomaticGroup(Q);
> f := GrowthFunction(G);

```

```
> R!f;  
2*x^3 + 4*x^2 + 3*x + 1
```

Now we take as example an infinite dihedral group. The group is infinite, so the result cannot be polynomial. We then extract the coefficients of the growth function for word lengths 0 to 14.

```
> FG2<d,e> := FreeGroup(2);  
> Q2 := quo<FG2| e^2, d^e = d^-1>;  
> G2 := AutomaticGroup(Q2);  
> f2 := GrowthFunction(G2);  
> R!f2;  
(-x^2 - 2*x - 1)/(x - 1)  
> PSR := PowerSeriesRing(Integers():Precision := 15);  
> Coefficients(PSR!f2);  
[ 1, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
```

---

## 75.8 Bibliography

[Hol97] Derek Holt. *KBMAG – Knuth-Bendix in Monoids and Automatic Groups*. University of Warwick, 1997.



# 76 GROUPS OF STRAIGHT-LINE PROGRAMS

<p><b>76.1 Introduction . . . . . 2377</b></p> <p><b>76.2 Construction of an SLP-Group and its Elements . . . . . 2377</b></p> <p>76.2.1 <i>Structure Constructors</i> . . . . . 2377</p> <p>SLPGroup(n) . . . . . 2377</p> <p>76.2.2 <i>Construction of an Element</i> . . . . . 2378</p> <p>Identity(G) . . . . . 2378</p> <p>Id(G) . . . . . 2378</p> <p>! . . . . . 2378</p> <p><b>76.3 Arithmetic with Elements . . . 2378</b></p> <p>* . . . . . 2378</p> <p>^ . . . . . 2378</p> <p>~ . . . . . 2378</p> <p># . . . . . 2378</p> <p>76.3.1 <i>Accessing the Defining Generators and Relations</i> . . . . . 2378</p> <p>. . . . . 2378</p> <p>Generators(G) . . . . . 2378</p> <p>NumberOfGenerators(G) . . . . . 2378</p> <p>Ngens(G) . . . . . 2378</p> <p>Parent(u) . . . . . 2378</p> <p><b>76.4 Addition of Extra Generators . 2379</b></p> <p>AddRedundantGenerators(G, Q) . . . . . 2379</p>	<p><b>76.5 Creating Homomorphisms . . . 2379</b></p> <p>hom&lt; &gt; . . . . . 2379</p> <p>Evaluate(u, Q) . . . . . 2379</p> <p>Evaluate(u, G) . . . . . 2379</p> <p>Evaluate(v, Q) . . . . . 2379</p> <p>Evaluate(v, G) . . . . . 2379</p> <p><b>76.6 Operations on Elements . . . 2381</b></p> <p>76.6.1 <i>Equality and Comparison</i> . . . . . 2381</p> <p>eq . . . . . 2381</p> <p>ne . . . . . 2381</p> <p><b>76.7 Set-Theoretic Operations . . . 2381</b></p> <p>76.7.1 <i>Membership and Equality</i> . . . . . 2381</p> <p>in . . . . . 2381</p> <p>notin . . . . . 2381</p> <p>subset . . . . . 2381</p> <p>notsubset . . . . . 2381</p> <p>76.7.2 <i>Set Operations</i> . . . . . 2382</p> <p>RandomProcess(G) . . . . . 2382</p> <p>Random(P) . . . . . 2382</p> <p>Rep(G) . . . . . 2382</p> <p>76.7.3 <i>Coercions Between Related Groups</i> . . . . . 2383</p> <p>! . . . . . 2383</p> <p><b>76.8 Bibliography . . . . . 2383</b></p>
--	--



# Chapter 76

## GROUPS OF STRAIGHT-LINE PROGRAMS

### 76.1 Introduction

This Chapter describes the category of straight-line program groups (SLP-groups). A straight-line program is formally a sequence  $[s_1, s_2, \dots, s_n]$  such that each  $s_i$  is one of the following:

- (i) A generator of the SLP-group;
- (ii) A product  $s_j s_k$ ,  $j < i, k < i$ ;
- (iii) A power  $s_j^n$ ,  $j < i$ ;
- (iv) A conjugate  $s_j^{s_k}$ ,  $j < i, k < i$ .

Effectively, a straight-line program can be regarded as a word in the generators which is stored as an expression tree instead of a list of generator-exponent pairs.

The importance of such a category of groups is that storing a word as an expression tree allows much faster evaluation of homomorphisms given as the unique extension of a mapping of the generators into a group of any category, as common subexpressions may be computed once only, and powers or conjugates may be more efficiently computed in the target group than by a linear product of generators and their inverses.

The name in MAGMA for the category of SLP-groups is GrpSLP.

### 76.2 Construction of an SLP-Group and its Elements

#### 76.2.1 Structure Constructors

SLPGroup(n)
-------------

Construct the free group  $F$  of straight-line programs on  $n$  generators, where  $n$  is a non-negative integer. The  $i$ -th generator may be referenced by the expression  $F.i$ ,  $i = 1, \dots, n$ .

#### Example H76E1

---

The statement

```
> F := SLPGroup(2);
```

creates the free group on two generators. Here the generators may be referenced using the standard names,  $F.1$  and  $F.2$ . Group operations on the elements will be stored as part of the result.

---

### 76.2.2 Construction of an Element

`Identity(G)`

`Id(G)`

`G ! 1`

Construct the identity element (the straight-line program `[]` of length 0) for the SLP-group  $G$ .

### 76.3 Arithmetic with Elements

`u * v`

Given straight-line programs  $u = [u_1, \dots, u_m]$  and  $v = [v_1, \dots, v_n]$  belonging to the same SLP-group  $G$ , return a straight-line program corresponding to the product of  $u$  and  $v$ . It is clear that the straight-line program  $[u_1, \dots, u_m, v_1, \dots, v_n, u_m v_n]$  satisfies the formal definition. In practice, the  $u_i$  and  $v_i$  need not be distinct, so the resulting program may be shorter.

`u ^ m`

Given an integer  $m$  and a straight-line program  $u$ , return the straight-line program corresponding to the  $m$ -th power of  $u$ .

`u ^ v`

Given straight-line programs  $u$  and  $v$ , return the straight-line program corresponding to the conjugate of  $u$  by  $v$ .

`#u`

Given a straight-line program  $u$ , return the number of multiplication, power or conjugate operations required to evaluate a homomorphism on  $u$ .

#### 76.3.1 Accessing the Defining Generators and Relations

The functions described here provide access to basic information stored for an SLP-group  $G$ .

`G . i`

The  $i$ -th generator for  $G$ .

`Generators(G)`

A set containing the generators for  $G$ .

`NumberOfGenerators(G)`

`Ngens(G)`

The number of generators for  $B$ .

`Parent(u)`

The parent group  $G$  of the straight-line program  $u$ .

## 76.4 Addition of Extra Generators

It is often the case that a particular expression in the original generators is important in some homomorphic image of the group in the sense that it will very often form a common subexpression of subsequent expression. For this reason, MAGMA provides facilities to build related SLP-groups in which particular words acquire generator status themselves.

```
AddRedundantGenerators(G, Q)
```

An SLP-group  $H$  on  $n + q$  generators (where  $G$  has  $n$  generators and  $Q$  has  $q$  elements). Furthermore, the identification of  $G.i$  with  $H.i$  ( $i \leq n$ ) and of  $Q[i]$  with  $H.(n + i)$  is maintained, allowing coercion between  $G$  and  $H$  and also simple definition of homomorphisms.

## 76.5 Creating Homomorphisms

Because SLP-groups exist primarily to allow the user to write efficient code for evaluating words under a homomorphism, there are some extra features in the homomorphism constructor which rely on the user providing correct input.

When evaluating single words, it may not be desirable to explicitly construct the homomorphism. The `Evaluate` function uses the same evaluation mechanism as the homomorphisms and may be a useful alternative.

```
hom< G -> H | L: parameters >
```

`CheckCodomain`

BOOLELT

*Default : true*

Return the group homomorphism  $\phi : G \rightarrow H$  defined by the list  $L$ . The list may contain:

- (i) Elements of the codomain. This form can only be used when all the preceding entries have given the image of the corresponding generator of  $G$ ;
- (ii) Generator-image pairs of the form `G.i -> x` or `<G.i, x>`;
- (iii) A homomorphism  $\psi$  from an SLP-group  $B$  to  $H$  where  $G$  has been defined as a result of adding redundant generators to  $B$ . If this item appears, it must appear first. After the remaining generators have been processed, any images which are not yet assigned are computed from  $\psi$ . If the parameter `CheckCodomain` has the value `false`, then it is assumed that the generator images lie in the codomain.

```
Evaluate(u, Q)
```

```
Evaluate(u, G)
```

```
Evaluate(v, Q)
```

```
Evaluate(v, G)
```

Evaluate the word  $u$  using the elements of  $Q$  as images of the generators of the parent of  $u$ . The sequence  $Q$  must contain at least as many group elements as the parent of  $u$  has generators.

The second form evaluates all the words in  $v$  simultaneously, which is usually quicker than doing individual evaluations.

When the second argument is a group  $G$ ,  $Q$  is taken as the sequence of generators of  $G$ .

---

### Example H76E2

An illustration of the use of `AddRedundantGenerators` and the homomorphism constructing machinery.

```
> G := SLPGroup(2);
> M := GeneralLinearGroup(19, 7);
> P := RandomProcess(G);
> x := Random(P);
> #x;
74
```

We evaluate  $x$  in  $M$  using the `Evaluate` function.

```
> m := Evaluate(x, [M.1, M.2]);
> Order(m);
118392315154200
```

If we wish to evaluate several different words, we may be better off using a homomorphism.

```
> Q := [x^G.1, x^G.2, x^(G.1*G.2)];
> phi := hom<G -> M | M.1, M.2>;
> time R1 := phi(Q);
Time: 0.129
```

We note that  $x$  has become important since it is now a common sub-expression of several straight-line programs. We can build a homomorphism which will store the image of  $x$  by adding  $x$  as a redundant generator and defining the same homomorphism from the resulting group.

```
> H := AddRedundantGenerators(G, [x]);
> QQ := [H | x: x in Q];
```

We will define  $\psi$  as the unique map on  $H$  which matches  $\phi$ .

```
> psi := hom<H -> M | phi>;
> time R2 := psi(QQ);
Time: 0.000
> R1 eq R2;
true
```

In fact, if we had looked at the expression lengths of the straight-line programs involved, we would have found the following, which explains the significant speed up:

```
> [#x: x in Q];
[ 75, 75, 75 ]
> [#x: x in QQ];
[ 1, 1, 2 ]
```

---

## 76.6 Operations on Elements

### 76.6.1 Equality and Comparison

`u eq v`

Returns **true** if and only if the straight-line programs  $u$  and  $v$  are identical. Identical here means they are identical as expression trees, not that they will always evaluate to the same word in the generators.

`u ne v`

Returns **true** if and only if the straight-line programs  $u$  and  $v$  are not identical. Identical here means they are identical as expression trees, not that they will always evaluate to the same word in the generators.

## 76.7 Set-Theoretic Operations

### 76.7.1 Membership and Equality

`g in G`

Given a straight-line program  $g$  and an SLP-group  $G$ , return **true** if  $g$  is an element of  $G$ , **false** otherwise.

`g notin G`

Given an straight-line program  $g$  and an SLP-group  $G$ , return **true** if  $g$  is not an element of  $G$ , **false** otherwise.

`S subset G`

Given an group  $G$  and a set  $S$  of elements belonging to a group  $H$ , where  $G$  and  $H$  are related, return **true** if  $S$  is a subset of  $G$ , **false** otherwise.

`S notsubset G`

Given a group  $G$  and a set  $S$  of elements belonging to a group  $H$ , where  $G$  and  $H$  are related, return **true** if  $S$  is not a subset of  $G$ , **false** otherwise.

## 76.7.2 Set Operations

### RandomProcess(G)

<b>Slots</b>	RNGINTELT	<i>Default : 10</i>
<b>Scramble</b>	RNGINTELT	<i>Default : 100</i>

Create a process to generate randomly chosen elements from the group  $G$ . The process is based on the product-replacement algorithm of [CLGM<sup>+</sup>95], modified by the use of an accumulator. At all times,  $N$  elements are stored where  $N$  is the maximum of the specified value for **Slots** and  $\text{Ngens}(G) + 1$ . Initially, these are just the generators of  $G$ . As well, one extra group element is stored, the accumulator. Initially, this is the identity. Random elements are now produced by successive calls to **Random(P)**, where  $P$  is the process created by this function. Each such call chooses one of the elements in the slots and multiplies it into the accumulator. The element in that slot is replaced by the product of it and another randomly chosen slot. The random value returned is the new accumulator value. Setting **Scramble** :=  $m$  causes  $m$  such operations to be performed before the process is returned.

It should be noted that this process is not suitable for infinite groups, since all elements produced are products of the generators only and not their inverses. However, as long as the homomorphic image of  $G$  that is being worked with is finite, there is no problem.

### Random(P)

Given a random element process  $P$  created by the function **RandomProcess(G)** for the SLP-group  $G$ , construct a random element of  $G$  by forming a random product over the expanded generating set stored as part of the process. The expanded generating set stored with the process is modified by replacing an existing generator by the element returned.

### Rep(G)

A representative element of  $G$ .

### Example H76E3

---

As an illustration of the efficiency of computing homomorphisms from SLP-groups, we set up the same random expression as both a straight-line program and a linear word.

```
> G := SLPGroup(3);
> F := FreeGroup(3);
> M := GeneralOrthogonalGroup(7, 3);
> gf := hom<G -> F | F.1, F.2, F.3>;
> gm := hom<G -> M | M.1, M.2, M.3>;
> fm := hom<F -> M | M.1, M.2, M.3>;
> P := RandomProcess(G);
> x := Random(P);
> #x;
```

85

The evaluation of the straight-line program will take 85 operations in  $M$ .

```
> w := gf(x);  
> #w;  
52307
```

The evaluation of the word will take 52306 multiplications in  $M$ .

```
> time h1 := gm(x);  
Time: 0.020  
> time h2 := fm(w);  
Time: 1.640  
> h1 eq h2;  
true
```

---

### 76.7.3 Coercions Between Related Groups

$G \wr g$
-----------

Given an element  $g$  belonging to an SLP-group  $H$  related to the group  $G$ , rewrite  $g$  as an element of  $G$ .

### 76.8 Bibliography

[CLGM<sup>+</sup>95] Frank Celler, Charles R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and E. A. O'Brien. Generating random elements of a finite group. *Comm. Algebra*, 23(13):4931–4948, 1995.



# 77 FINITELY PRESENTED SEMIGROUPS

<b>77.1 Introduction . . . . .</b>	<b>2387</b>	Generators(S)	2391
<b>77.2 The Construction of Free Semi-</b>	<b>2387</b>	NumberOfGenerators(S)	2391
<b>groups and their Elements . .</b>	<b>2387</b>	Ngens(S)	2391
77.2.1 <i>Structure Constructors . . . . .</i>	2387	Parent(u)	2391
FreeSemigroup(n)	2387	Relations(S)	2392
FreeMonoid(n)	2387	<b>77.5 Subsemigroups, Ideals and Quo-</b>	
77.2.2 <i>Element Constructors . . . . .</i>	2388	<b>tients . . . . .</b>	<b>2392</b>
!	2388	77.5.1 <i>Subsemigroups and Ideals . . . . .</i>	2392
Id(M)	2388	sub< >	2392
!	2388	ideal< >	2392
<b>77.3 Elementary Operators for Words</b>	<b>2388</b>	lideal< >	2392
77.3.1 <i>Multiplication and Exponentiation .</i>	2388	rideal< >	2392
*	2388	77.5.2 <i>Quotients . . . . .</i>	2393
^	2388	quo< >	2393
!	2388	<b>77.6 Extensions . . . . .</b>	<b>2393</b>
77.3.2 <i>The Length of a Word . . . . .</i>	2388	DirectProduct(R, S)	2393
#	2388	FreeProduct(R, S)	2393
77.3.3 <i>Equality and Comparison . . . . .</i>	2389	<b>77.7 Elementary Tietze</b>	
eq	2389	<b>Transformations . . . . .</b>	<b>2393</b>
ne	2389	AddRelation(S, r)	2393
lt	2389	AddRelation(S, r, i)	2393
le	2389	DeleteRelation(S, r)	2394
ge	2389	DeleteRelation(S, i)	2394
gt	2389	ReplaceRelation(S, r <sub>1</sub> , r <sub>2</sub> )	2394
IsOne(u)	2389	ReplaceRelation(S, i, r)	2394
<b>77.4 Specification of a Presentation</b>	<b>2390</b>	AddGenerator(S)	2394
77.4.1 <i>Relations . . . . .</i>	2390	AddGenerator(S, w)	2394
=	2390	DeleteGenerator(S, y)	2394
LHS(r)	2390	<b>77.8 String Operations on Words .</b>	<b>2395</b>
RHS(r)	2390	Eliminate(u, x, v)	2395
77.4.2 <i>Presentations . . . . .</i>	2390	Match(u, v, f)	2395
Semigroup< >	2390	Random(S, m, n)	2395
Monoid< >	2391	RotateWord(u, n)	2395
77.4.3 <i>Accessing the Defining Generators</i>		Substitute(u, f, n, v)	2395
<i>and Relations . . . . .</i>	2391	Subword(u, f, n)	2395
.	2391	ElementToSequence(u)	2395
		Eltseq(u)	2395



# Chapter 77

## FINITELY PRESENTED SEMIGROUPS

### 77.1 Introduction

This Chapter presents the functions designed for computing with finitely-presented semigroups (fp-semigroups for short).

### 77.2 The Construction of Free Semigroups and their Elements

#### 77.2.1 Structure Constructors

`FreeSemigroup(n)`

Construct the free semigroup  $F$  on  $n$  generators, where  $n$  is a positive integer. The  $i$ -th generator may be referenced by the expression `F.i`,  $i = 1, \dots, n$ . Note that a special form of the assignment statement is provided which enables the user to assign names to the generators of  $F$ . In this form of assignment, the list of generator names is enclosed within angle brackets and appended to the variable name on the **left hand side** of the assignment statement.

`FreeMonoid(n)`

Construct the free monoid  $F$  on  $n$  generators, where  $n$  is a positive integer. The  $i$ -th generator may be referenced by the expression `F.i`,  $i = 1, \dots, n$ . Note that a special form of the assignment statement is provided which enables the user to assign names to the generators of  $F$ . In this form of assignment, the list of generator names is enclosed within angle brackets and appended to the variable name on the **left hand side** of the assignment statement.

#### Example H77E1

---

The statement

```
> F := FreeSemigroup(2);
```

creates the free semigroup on two generators. Here the generators may be referenced using the standard names,  $F.1$  and  $F.2$ .

The statement

```
> F<x, y> := FreeSemigroup(2);
```

defines  $F$  to be the free semigroup on two generators and assigns the names  $x$  and  $y$  to the generators.

---

### 77.2.2 Element Constructors

Suppose  $S$  is an fp-semigroup, not necessarily free, for which generators have already been defined. A *word* is defined inductively as follows:

- (i) A generator is a word;
- (ii) The product  $uv$  of the words  $u$  and  $v$  is a word;
- (iii) The power of a word,  $u^n$ , where  $u$  is a word and  $n$  is an integer, is a word.

An element (word) of  $S$  may be constructed as an expression in the generators as outlined below.

$S ! [i_1, \dots, i_s]$
-------------------------

Given a semigroup  $S$  defined on  $r$  generators and a sequence  $Q = [i_1, \dots, i_s]$  of integers lying in the range  $[1, r]$ , construct the word  $G.i_1G.i_2 \cdots G.i_s$ .

$\text{Id}(M)$
----------------

$M ! 1$
---------

Construct the identity element (empty word) for the fp-monoid  $M$ .

## 77.3 Elementary Operators for Words

### 77.3.1 Multiplication and Exponentiation

The word operations defined here may be applied either to the words of a free semigroup or the words of a semigroup with non-trivial relations.

$u * v$
---------

Given words  $u$  and  $v$  belonging to the same fp-semigroup  $S$ , return the product of  $u$  and  $v$ .

$u \hat{=} n$
---------------

The  $n$ -th power of the word  $u$ , where  $n$  is a positive integer.

$G ! Q$
---------

Given a sequence  $Q$  of words belonging to the fp-semigroup  $G$ , return the product  $Q[1]Q[2] \cdots Q[n]$  of the terms of  $Q$  as a word in  $G$ .

### 77.3.2 The Length of a Word

$\#u$
-------

The length of the word  $u$ .

### 77.3.3 Equality and Comparison

The words of an fp-semigroup  $S$  are ordered first by length and then lexicographically. The lexicographic ordering is determined by the following ordering on the generators:

$$S.1 < S.2 < S.3 < S.4 < \dots$$

Here,  $u$  and  $v$  are words belonging to some common fp-semigroup.

`u eq v`

Returns **true** if the words  $u$  and  $v$  are identical (as elements of the appropriate free semigroup), **false** otherwise.

`u ne v`

Returns **true** if the words  $u$  and  $v$  are not identical (as elements of the appropriate free semigroup), **false** otherwise.

`u lt v`

Returns **true** if the word  $u$  precedes the word  $v$ , with respect to the ordering defined above for elements of an fp-semigroup, **false** otherwise.

`u le v`

Returns **true** if the word  $u$  either precedes, or is equal to, the word  $v$ , with respect to the ordering defined above for elements of an fp-semigroup, **false** otherwise.

`u ge v`

Returns **true** if the word  $u$  either follows, or is equal to, the word  $v$ , with respect to the ordering defined above for elements of an fp-semigroup, **false** otherwise.

`u gt v`

Returns **true** if the word  $u$  follows the word  $v$ , with respect to the ordering defined above for elements of an fp-semigroup.

`IsOne(u)`

Returns **true** if the word  $u$ , belonging to the monoid  $M$ , is the identity word, **false** otherwise.

## 77.4 Specification of a Presentation

### 77.4.1 Relations

$w_1 = w_2$

Given words  $w_1$  and  $w_2$  over the generators of an fp-semigroup  $S$ , create the relation  $w_1 = w_2$ . Note that this relation is not automatically added to the existing set of defining relations  $R$  for  $S$ . It may be added to  $R$ , for example, through use of the `quo`-constructor (see below).

LHS( $r$ )

Given a relation  $r$  over the generators of  $S$ , return the left hand side of the relation  $r$ . The object returned is a word over the generators of  $S$ .

RHS( $r$ )

Given a relation  $r$  over the generators of  $S$ , return the right hand side of the relation  $r$ . The object returned is a word over the generators of  $S$ .

### 77.4.2 Presentations

A semigroup with non-trivial relations is constructed as a quotient of an existing semigroup, possibly a free semigroup.

`Semigroup< generators | relations >`

Given a *generators* clause consisting of a list of variables  $x_1, \dots, x_r$ , and a set of relations *relations* over these generators, first construct the free semigroup  $F$  on the generators  $x_1, \dots, x_r$  and then construct the quotient of  $F$  corresponding to the ideal of  $F$  defined by *relations*.

The syntax for the *relations* clause is the same as for the `quo`-constructor. The function returns:

- (a) The quotient semigroup  $S$ ;
- (b) The natural homomorphism  $\phi : F \rightarrow S$ .

Thus, the statement

`S< y1, ..., yr > := Semigroup< x1, ..., xr | w1, ..., ws >;`  
is an abbreviation for

`F< x1, ..., xr > := FreeSemigroup(r);`  
`S< y1, ..., yr > := quo< F | w1, ..., ws >;`

**Monoid** $\langle$  *generators* | *relations*  $\rangle$

Given a *generators* clause consisting of a list of variables  $x_1, \dots, x_r$ , and a set of relations *relations* over these generators, first construct the free monoid  $F$  on the generators  $x_1, \dots, x_r$  and then construct the quotient of  $F$  corresponding to the ideal of  $F$  defined by *relations*.

The syntax for the *relations* clause is the same as for the **quo**-constructor. The function returns:

- (a) The quotient monoid  $M$ ;
- (b) The natural homomorphism  $\phi : F \rightarrow M$ .

Thus, the statement

$M \langle y_1, \dots, y_r \rangle := \text{Monoid} \langle x_1, \dots, x_r \mid w_1, \dots, w_s \rangle$ ;  
is an abbreviation for

$F \langle x_1, \dots, x_r \rangle := \text{FreeMonoid}(r)$ ;  
 $M \langle y_1, \dots, y_r \rangle := \text{quo} \langle F \mid w_1, \dots, w_s \rangle$ ;

---

### Example H77E2

We create the monoid defined by the presentation  $\langle x, y \mid x^2, y^2, (xy)^2 \rangle$ .

```
> M<x,y> := Monoid< x, y | x^2, y^2, (x*y)^2 >;
> M;
```

Finitely presented monoid

Relations:

```
x^2 = Id(M)
y^2 = Id(M)
(x * y)^2 = Id(M)
```

---

### 77.4.3 Accessing the Defining Generators and Relations

The functions in this group provide access to basic information stored for a finitely-presented semigroup  $G$ .

**S . i**

The  $i$ -th defining generator for  $S$ .

**Generators(S)**

A set containing the generators for  $S$ .

**NumberOfGenerators(S)**

**Ngens(S)**

The number of generators for  $S$ .

**Parent(u)**

The parent semigroup  $S$  of the word  $u$ .

Relations(S)
--------------

A sequence containing the defining relations for  $S$ .

## 77.5 Subsemigroups, Ideals and Quotients

### 77.5.1 Subsemigroups and Ideals

sub< S   L <sub>1</sub> , . . . , L <sub>r</sub> >
--

Construct the subsemigroup  $R$  of the fp-semigroup  $S$  generated by the words specified by the terms of the *generator list*  $L_1, \dots, L_r$ .

A term  $L_i$  of the generator list may consist of any of the following objects:

- (a) A word;
- (b) A set or sequence of words;
- (c) A sequence of integers representing a word;
- (d) A set or sequence of sequences of integers representing words;
- (e) A subsemigroup of an fp-semigroup;
- (f) A set or sequence of subsemigroups.

The collection of words and semigroups specified by the list must all belong to the semigroup  $S$ , and  $R$  will be constructed as a subgroup of  $S$ .

The generators of  $R$  consist of the words specified directly by terms  $L_i$  together with the stored generating words for any semigroups specified by terms of  $L_i$ . Reiterations of an element and occurrences of the identity element are removed (unless  $R$  is trivial).

ideal< S   L <sub>1</sub> , . . . , L <sub>r</sub> >
--

Construct the two-sided ideal  $I$  of the fp-semigroup  $S$  generated by the words specified by the terms of the *generator list*  $L_1, \dots, L_r$ .

The possible forms of a term  $L_i$  of the generator list are the same as for the sub-constructor.

lideal< G   L <sub>1</sub> , . . . , L <sub>r</sub> >
---

Construct the left ideal  $I$  of the fp-semigroup  $S$  generated by the words specified by the terms of the *generator list*  $L_1, \dots, L_r$ .

The possible forms of a term  $L_i$  of the generator list are the same as for the sub-constructor.

rideal< G   L <sub>1</sub> , . . . , L <sub>r</sub> >
---

Construct the right ideal  $I$  of the fp-semigroup  $S$  generated by the words specified by the terms of the *generator list*  $L_1, \dots, L_r$ .

The possible forms of a term  $L_i$  of the generator list are the same as for the sub-constructor.

### 77.5.2 Quotients

`quo< F | relations >`

Given an fp-semigroup  $F$ , and a list of relations  $relations$  over the generators of  $F$ , construct the quotient of  $F$  by the ideal of  $F$  defined by  $relations$ .

The expression defining  $F$  may be either simply the name of a previously constructed semigroup, or an expression defining an fp-semigroup.

Each term of the list  $relations$  must be a *relation*, a *relation list* or, if  $S$  is a monoid, a *word*.

A *word* is interpreted as a relator if  $S$  is a monoid.

A *relation* consists of a pair of words, separated by '='. (See above).

A *relation list* consists of a list of words, where each pair of adjacent words is separated by '=':  $w_1 = w_2 = \dots = w_r$ . This is interpreted as the relations  $w_1 = w_r, \dots, w_{r-1} = w_r$ .

Note that the relation list construct is only meaningful in the context of the **fp semigroup**-constructor.

In the context of the **quo**-constructor, the identity element (empty word) of a monoid may be represented by the digit 1.

Note that this function returns:

- (a) The quotient semigroup  $S$ ;
- (b) The natural homomorphism  $\phi : F \rightarrow S$ .

### 77.6 Extensions

`DirectProduct(R, S)`

Given two fp-semigroups  $R$  and  $S$ , construct the direct product of  $R$  and  $S$ .

`FreeProduct(R, S)`

Given two fp-semigroups  $R$  and  $S$ , construct the free product of  $R$  and  $S$ .

### 77.7 Elementary Tietze Transformations

`AddRelation(S, r)`

`AddRelation(S, r, i)`

Given an fp-semigroup  $S$  and a relation  $r$  in the generators of  $S$ , create the quotient semigroup obtained by adding the relation  $r$  to the defining relations of  $S$ . If an integer  $i$  is specified as third argument, insert the new relation after the  $i$ -th relation of  $S$ . If the third argument is omitted,  $r$  is added to the end of the relations that are carried across from  $S$ .

**DeleteRelation(S, r)**

Given an fp-semigroup  $S$  and a relation  $r$  that occurs among the given defining relations for  $S$ , create the semigroup  $T$ , having the same generating set as  $S$  but with the relation  $r$  removed.

**DeleteRelation(S, i)**

Given an fp-semigroup  $S$  and an integer  $i$ ,  $1 \leq i \leq m$ , where  $m$  is the number of defining relations for  $S$ , create the semigroup  $T$  having the same generating set as  $S$  but with the  $i$ -th relation omitted.

**ReplaceRelation(S, r<sub>1</sub>, r<sub>2</sub>)**

Given an fp-semigroup  $S$  and relations  $r_1$  and  $r_2$  in the generators of  $S$ , where  $r_1$  is one of the given defining relations for  $S$ , create the semigroup  $T$  having the same generating set as  $S$  but with the relation  $r_1$  replaced by the relation  $r_2$ .

**ReplaceRelation(S, i, r)**

Given an fp-semigroup  $S$ , an integer  $i$ ,  $1 \leq i \leq m$ , where  $m$  is the number of defining relations for  $S$ , and a relation  $r$  in the generators of  $S$ , create the semigroup  $T$  having the same generating set as  $S$  but with the  $i$ -th relation of  $S$  replaced by the relation  $r$ .

**AddGenerator(S)**

Given an fp-semigroup  $S$  with presentation  $\langle X \mid R \rangle$ , create the semigroup  $T$  with presentation  $\langle X \cup \{y\} \mid R \rangle$ , where  $y$  denotes a new generator.

**AddGenerator(S, w)**

Given an fp-semigroup  $S$  with presentation  $\langle X \mid R \rangle$  and a word  $w$  in the generators of  $S$ , create the semigroup  $T$  with presentation  $\langle X \cup \{y\} \mid R \cup \{y = w\} \rangle$ , where  $y$  denotes a new generator.

**DeleteGenerator(S, y)**

Given an fp-semigroup  $S$  with presentation  $\langle X \mid R \rangle$  and a generator  $y$  of  $S$  such that either  $S$  has no relations involving  $y$ , or a single relation  $r$  containing a single occurrence of  $y$ , create the semigroup  $T$  with presentation  $\langle X - \{y\} \mid R - \{r\} \rangle$ .

## 77.8 String Operations on Words

**Eliminate(u, x, v)**

Given words  $u$  and  $v$ , and a generator  $x$ , belonging to a semigroup  $S$ , return the word obtained from  $u$  by replacing each occurrence of  $x$  by  $v$ .

**Match(u, v, f)**

Suppose  $u$  and  $v$  are words belonging to the same semigroup  $S$ , and that  $f$  is an integer such that  $1 \leq f \leq \#u$ . If  $v$  is a subword of  $u$ , the function returns true, as well as the least integer  $l$  such that:

- (a)  $l \geq f$ ; and,
- (b)  $v$  appears as a subword of  $u$ , starting at the  $l$ -th letter of  $u$ .

If no such  $l$  is found, **Match** returns only false.

**Random(S, m, n)**

A random word of length  $l$  in the generators of the semigroup  $S$ , where  $m \leq l \leq n$ .

**RotateWord(u, n)**

The word obtained by cyclically permuting the word  $u$  by  $n$  places. If  $n$  is positive, the rotation is from left to right, while if  $n$  is negative the rotation is from right to left. In the case where  $n$  is zero, the function returns  $u$ .

**Substitute(u, f, n, v)**

Given words  $u$  and  $v$  belonging to a semigroup  $S$ , and non-negative integers  $f$  and  $n$ , this function replaces the substring of  $u$  of length  $n$ , starting at position  $f$ , by the word  $v$ . Thus, if  $u = x_{i_1} \cdots x_{i_f} \cdots x_{i_{f+n-1}} \cdots x_{i_m}$  then the substring  $x_{i_f} \cdots x_{i_{f+n-1}}$  is replaced by  $v$ . If  $u$  and  $v$  belong to a monoid  $M$  and the function is invoked with  $v = \text{Id}(M)$ , then the substring  $x_{i_f} \cdots x_{i_{f+n-1}}$  of  $u$  is deleted.

**Subword(u, f, n)**

The subword of the word  $u$  comprising the  $n$  consecutive letters commencing at the  $f$ -th letter of  $u$ .

**ElementToSequence(u)**

**Eltseq(u)**

The sequence obtained by decomposing  $u$  into the indices of its constituent generators. Thus, if  $u = x_{i_1} \cdots x_{i_m}$ , then the sequence constructed by **ElementToSequence** is  $[i_1, i_2, \dots, i_m]$ .



# 78 MONOIDS GIVEN BY REWRITE SYSTEMS

<b>78.1 Introduction . . . . .</b>	<b>2399</b>		
78.1.1 Terminology . . . . .	2399	!	2408
78.1.2 The Category of Rewrite Monoids . . . . .	2399	!	2408
78.1.3 The Construction of a Rewrite Monoid . . . . .	2399	78.3.4 Arithmetic with Words . . . . .	2408
<b>78.2 Construction of a Rewrite Monoid . . . . .</b>	<b>2400</b>	*	2409
RWSMonoid(Q: -)	2400	^	2409
SetVerbose("KBMAG", v)	2402	eq	2409
<b>78.3 Basic Operations . . . . .</b>	<b>2405</b>	ne	2409
78.3.1 Accessing Monoid Information . . . . .	2405	IsId(w)	2409
.	2405	IsIdentity(w)	2409
Generators(M)	2405	#	2409
NumberOfGenerators(M)	2405	ElementToSequence(u)	2409
Ngens(M)	2405	Eltseq(u)	2409
Relations(M)	2405	<b>78.4 Homomorphisms . . . . .</b>	<b>2410</b>
NumberOfRelations(M)	2405	78.4.1 General Remarks . . . . .	2410
Nrels(M)	2405	78.4.2 Construction of Homomorphisms . . . . .	2410
Ordering(M)	2405	hom< >	2410
Parent(w)	2405	<b>78.5 Set Operations . . . . .</b>	<b>2410</b>
78.3.2 Properties of a Rewrite Monoid . . . . .	2406	Random(M, n)	2410
IsConfluent(M)	2406	Random(M)	2410
IsFinite(M)	2406	Representative(M)	2410
Order(M)	2407	Rep(M)	2410
#	2407	Set(M, a, b)	2411
78.3.3 Construction of a Word . . . . .	2408	Set(M)	2411
Identity(M)	2408	Seq(M, a, b)	2411
Id(M)	2408	Seq(M)	2411
		<b>78.6 Conversion to a Finitely Presented Monoid . . . . .</b>	<b>2412</b>
		<b>78.7 Bibliography . . . . .</b>	<b>2413</b>



## Chapter 78

# MONOIDS GIVEN BY REWRITE SYSTEMS

### 78.1 Introduction

The category of monoids defined by finite sets of rewrite rules provide a Magma level interface to Derek Holt's KBMAG programs, and specifically to KBMAG's Knuth–Bendix completion procedure on monoids defined by a finite presentation. As such much of the documentation in this chapter is taken from the KBMAG documentation [Hol97]. Familiarity with the Knuth–Bendix completion procedure is assumed. Some familiarity with KBMAG would be beneficial.

#### 78.1.1 Terminology

A rewrite monoid  $M$  is a finitely presented monoid in which equality between elements of  $M$ , called *words* or *strings*, is decidable via a sequence of rewriting equations, called *reduction relations*, *rules*, or *equations*. In the interests of efficiency the reduction rules are codified into a finite state automaton called a *reduction machine*. The words in a rewrite monoid  $M$  are ordered, as are the reduction relations of  $M$ . Several possible orderings of words are supported, namely short-lex, recursive, weighted short-lex and wreath-product orderings. A rewrite monoid can be *confluent* or *non-confluent*. If a rewrite monoid  $M$  is confluent its reduction relations, or more specifically its reduction machine, can be used to reduce words in  $M$  to their irreducible normal forms under the given ordering, and so the word problem for  $M$  can be efficiently solved.

#### 78.1.2 The Category of Rewrite Monoids

The family of all rewrite monoids forms a category. The objects are the rewrite monoids and the morphisms are monoid homomorphisms. The MAGMA designation for this category of monoids is `MonRWS`. Elements of a rewrite monoid are designated as `MonRWSElt`.

#### 78.1.3 The Construction of a Rewrite Monoid

A rewrite monoid  $M$  is constructed in a three-step process:

- (i) A free monoid  $F$  of the appropriate rank is defined.
- (ii) A quotient  $Q$  of  $F$  is created.
- (iii) The Knuth–Bendix completion procedure is applied to the monoid  $Q$  to produce a monoid  $M$  defined by a rewrite system.

The Knuth–Bendix procedure may or may not succeed. If it fails the user may need to perform the above steps several times, manually adjusting parameters that control the execution of the Knuth–Bendix procedure. If it succeeds then the rewrite systems constructed will be confluent.

## 78.2 Construction of a Rewrite Monoid

RWSMonoid(Q: *parameters*)

The Knuth–Bendix completion procedure for monoids is run, with the relations of  $Q$  taken as the initial reduction rules for the procedure. Regardless of whether or not the completion procedure succeeds, the result will be a rewrite monoid,  $M$ , containing a reduction machine and a sequence of reduction relations. If the procedure succeeds  $M$  will be marked as confluent, and the word problem for  $M$  is therefore decidable. If, as is very likely, the procedure fails then  $M$  will be marked as non-confluent. In this case  $M$  will contain both the reduction relations and the reduction machine computed up to the point of failure.

As the Knuth–Bendix procedure will more often than not run forever, some conditions must be specified under which it will stop. These take the form of limits that are placed on certain variables, such as the number of reduction relations. If any of these limits are exceeded during a run of the completion procedure it will fail, returning a non-confluent rewrite monoid. The optimal values for these limits varies from example to example.

<b>MaxRelations</b>	RNGINTELT	<i>Default : 32767</i>
---------------------	-----------	------------------------

Limit the maximum number of reduction equations to **MaxRelations**.

<b>GeneratorOrder</b>	SEQENUM	<i>Default :</i>
-----------------------	---------	------------------

Give an ordering for the generators. This ordering affects the ordering of words in the alphabet. If not specified the ordering defaults to the order induced by  $Q$ 's generators, that is  $[g_1, \dots, g_n]$  where  $g_1, \dots, g_n$  are the generators of  $Q$ .

<b>Ordering</b>	MONSTGELT	<i>Default : "ShortLex"</i>
-----------------	-----------	-----------------------------

<b>Levels</b>	SEQENUM	<i>Default :</i>
---------------	---------	------------------

<b>Weights</b>	SEQENUM	<i>Default :</i>
----------------	---------	------------------

**Ordering := "ShortLex"**: Use the short-lex ordering on strings. Shorter words come before longer, and for words of equal length lexicographical ordering is used, using the given ordering of the generators.

**Ordering := "Recursive" | "RTRecursive"**: Use a recursive ordering on strings. There are various ways to define this. Perhaps the quickest is as follows. Let  $u$  and  $v$  be strings in the generators. If one of  $u$  and  $v$ , say  $v$ , is empty, then  $u \geq v$ . Otherwise, let  $u = u'a$  and  $v = v'b$ , where  $a$  and  $b$  are generators. Then  $u > v$  if and only if one of the following holds:

- (i)  $a = b$  and  $u' > v'$ ;
- (ii)  $a > b$  and  $u > v'$ ;
- (iii)  $b > a$  and  $u' > v$ .

The **RTRecursive** ordering is similar to the **Recursive** ordering, but with  $u = au'$  and  $v = bv'$ . Occasionally one or the other runs significantly quicker, but usually they perform similarly.





**Example H78E1**

---

Starting with a monoid presentation for the alternating group  $A_4$ , we construct a rewrite system. Since we don't specify an ordering the default `ShortLex` ordering is used. Since we don't specify a generator ordering, the default generator ordering, in this case that from  $Q$ , is used.

```
> FM<g10,g20,g30> := FreeMonoid(3);
> Q := quo< FM | g10^2=1, g20*g30=1, g30*g20=1,
>       g20*g20=g30, g30*g10*g30=g10*g20*g10>;
> M := RWSMonoid(Q);
> print M;
```

A confluent rewrite monoid.

Generator Ordering = [ g10, g20, g30 ]

Ordering = ShortLex.

The reduction machine has 12 states.

The rewrite relations are:

```
g10^2 = Id(FM)
g20 * g30 = Id(FM)
g30 * g20 = Id(FM)
g20^2 = g30
g30 * g10 * g30 = g10 * g20 * g10
g30^2 = g20
g20 * g10 * g20 = g10 * g30 * g10
g30 * g10 * g20 * g10 = g20 * g10 * g30
g10 * g20 * g10 * g30 = g30 * g10 * g20
g20 * g10 * g30 * g10 = g30 * g10 * g20
g10 * g30 * g10 * g20 = g20 * g10 * g30
```

**Example H78E2**

---

We construct the second of Bernard Neumann's series of increasingly complicated presentations of the trivial monoid. The example runs best with a large value of `TidyInt`. Again the default `ShortLex` ordering is used.

```
> FM<x,X,y,Y,z,Z> := FreeMonoid(6);
> Q := quo< FM |
>       x*X=1, X*x=1, y*Y=1, Y*y=1, z*Z=1, Z*z=1,
>       y*y*X*Y*x*Y*z*y*Z*Z*X*y*x*Y*Y*z*z*Y*Z*y*z*z*Y*Z*y=1,
>       z*z*Y*Z*y*Z*x*z*X*X*Y*z*y*Z*Z*x*x*Z*X*z*x*x*Z*X*z=1,
>       x*x*Z*X*z*X*y*x*Y*Y*Z*x*z*X*X*y*y*X*Y*x*y*y*X*Y*x=1>;
> M := RWSMonoid(Q : TidyInt := 3000);
> print M;
```

A confluent rewrite monoid.

Generator Ordering = [ x, X, y, Y, z, Z ]

Ordering = ShortLex.

The reduction machine has 1 state.

The rewrite relations are:

```
Z = Id(FM)
Y = Id(FM)
```

```

z = Id(FM)
X = Id(FM)
y = Id(FM)
x = Id(FM)

```

### Example H78E3

---

We construct a confluent presentation of a submonoid of a nilpotent group.

```

> FM<a,b,c> := FreeMonoid(6);
> Q := quo< FM | b*a=a*b*c, c*a=a*c, c*b=b*c >;
> M := RWSMonoid(Q:Ordering:="Recursive", GeneratorOrder:=[c,b,a]);
> M;
A confluent rewrite monoid.
Generator Ordering = [ c, b, a ]
Ordering = Recursive.
The reduction machine has 3 states.
    b * a = a * b * c
    c * a = a * c
    c * b = b * c
> Order(M);
Infinity

```

### Example H78E4

---

We construct a monoid presentation corresponding to the Fibonacci group  $F(2, 7)$ . This is a very difficult calculation unless the parameters of `RWSMonoid` are selected carefully. The best approach is to run the Knuth-Bendix once using a `Recursive` ordering with a limit on the lengths of equations stored (`MaxStoredLen := <15,15>` works well). This will halt returning a non-confluent rewrite monoid  $M$ , and give a warning message that the Knuth-Bendix procedure only partly succeeded. The original equations should then be appended to the relations of  $M$ , and the Knuth-Bendix re-run with no limits on lengths. It will then quickly complete with a confluent set. This is typical of a number of difficult examples, where good results can be obtained by running more than once.

```

> FM<a,b,c,d,e,f,g> := FreeMonoid(7);
> I := [a*b=c, b*c=d, c*d=e, d*e=f, e*f=g, f*g=a, g*a=b];
> Q := quo<FM | I>;
> M := RWSMonoid(Q: Ordering := "Recursive", MaxStoredLen := <15,15>);
Warning: Knuth Bendix only partly succeeded
> Q := quo< FM | Relations(M) cat I>;
> M := RWSMonoid(Q: Ordering := "Recursive");
> print M;
A confluent rewrite monoid.
Generator Ordering = [ a, b, c, d, e, f, g ]
Ordering = Recursive.
The reduction machine has 30 states.
The rewrite relations are:

```

```

c = a^25
d = a^20
e = a^16
f = a^7
g = a^23
b = a^24
a^30 = a
> Order(M);
30

```

It turns out that the non-identity elements of this monoid form a submonoid isomorphic to the Fibonacci group  $F(2, 7)$  which is cyclic of order 29.

---

## 78.3 Basic Operations

### 78.3.1 Accessing Monoid Information

The functions in this section provide access to basic information stored for a rewrite monoid  $M$ .

`M . i`

The  $i$ -th defining generator for  $M$ .

`Generators(M)`

A sequence containing the defining generators for  $M$ .

`NumberOfGenerators(M)`

`Ngens(M)`

The number of defining generators for  $M$ .

`Relations(M)`

A sequence containing the defining relations for  $M$ . The relations will be given between elements of the free monoid of which  $M$  is a quotient. In these relations the (image of the) left hand side (in  $M$ ) will always be greater than the (image of the) right hand side (in  $M$ ) in the ordering on words used to construct  $M$ .

`NumberOfRelations(M)`

`Nrels(M)`

The number of relations in  $M$ .

`Ordering(M)`

The ordering of  $M$ .

`Parent(w)`

The parent monoid  $M$  for the word  $w$ .

**Example H78E5**

---

We illustrate the access operations using the following presentation of  $S_4$ .

```

> FM<a,b> := FreeMonoid(2);
> Q := quo< FM | a^2=1, b^3=1, (a*b)^4=1 >;
> M<x,y> := RWSMonoid(Q);
> print M;
A confluent rewrite monoid.
Generator Ordering = [ a, b ]
Ordering = ShortLex.
The reduction machine has 12 states.
  a^2 = Id(FM)
  b^3 = Id(FM)
  b * a * b * a * b = a * b^2 * a
  b^2 * a * b^2 = a * b * a * b * a
  b * a * b^2 * a * b * a = a * b * a * b^2 * a * b
> print Order(M);
24
> print M.1;
x
> print M.1*M.2;
x * y
> print Generators(M);
[ x, y ]
> print Ngens(M);
2
> print Relations(M);
[ a^2 = Id(FM), b^3 = Id(FM), b * a * b * a * b = a * b^2 * a, b^2 * a * b^2 = a
* b * a * b * a, b * a * b^2 * a * b * a = a * b * a * b^2 * a * b ]
> print Nrels(M);
5
> print Ordering(M);
ShortLex

```

---

**78.3.2 Properties of a Rewrite Monoid**

IsConfluent(M)
----------------

Returns true if  $M$  is confluent, false otherwise.

IsFinite(M)
-------------

Given a confluent monoid  $M$  return true if  $M$  has finite order and false otherwise. If  $M$  does have finite order also return the order of  $M$ .

Order(M)
----------

#M
----

Given a monoid  $M$  defined by a confluent presentation, this function returns the cardinality of  $M$ . If the order of  $M$  is known to be infinite  $\infty$  is returned.

---

**Example H78E6**

We construct a threefold cover of  $A_6$ .

```
> FM<a,b> := FreeMonoid(2);
> Q := quo< FM | a^3=1, b^3=1, (a*b)^4=1, (a*b^2)^5 = 1 >;
> M := RWSMonoid(Q);
> print Order(M);
1080
> IsConfluent(M);
true
```

---

**Example H78E7**

We construct the 2-generator free abelian group and compute its order. The result `Infinity` indicates that the group has infinite order.

```
> FM<a,A,b,B> := FreeMonoid(4);
> Q := quo< FM | a*A=1, A*a=1, b*B=1, B*b=1, B*a*b=a>;
> M := RWSMonoid(Q);
> Order(M);
Infinity
```

---

**Example H78E8**

We construct the Weyl group  $E_8$  and test whether or not it has finite order.

```
> FM<a,b,c,d,e,f,g,h> := FreeMonoid(8);
> Q := quo< FM | a^2=1, b^2=1, c^2=1, d^2=1, e^2=1, f^2=1, g^2=1,
>       h^2=1, b*a*b=a*b*a, c*a=a*c, d*a=a*d, e*a=a*e, f*a=a*f,
>       g*a=a*g, h*a=a*h, c*b*c=b*c*b, d*b=b*d, e*b=b*e, f*b=b*f,
>       g*b=b*g, h*b=b*h, d*c*d=c*d*c, e*c=e*c, f*c=c*f,
>       g*c=c*g, h*c=c*h, e*d=d*e, f*d=d*f, g*d=d*g, h*d=d*h,
>       f*e*f=e*f*e, g*e=e*g, h*e=e*h, g*f*g=f*g*f, h*f=f*h,
>       h*g*h=g*h*g>;
> M := RWSMonoid(Q);
> print IsFinite(M);
true
> isf, ord := IsFinite(M);
> print isf, ord;
true 696729600
```

---

### 78.3.3 Construction of a Word

Identity(M)
-------------

Id(M)
-------

M ! 1
-------

Construct the identity word in  $M$ .

M ! [ $i_1, \dots, i_s$ ]
---------------------------

Given a rewrite monoid  $M$  defined on  $r$  generators and a sequence  $[i_1, \dots, i_s]$  of integers lying in the range  $[1, r]$ , construct the word  $M.i_1 * M.i_2 * \dots * M.i_s$ .

#### Example H78E9

---

We construct the Fibonacci group  $F(2, 7)$ , and its identity.

```
> FM<a,A,b,B,c,C,d,D,e,E,f,F,g,G> := FreeMonoid(14);
> Q := quo< FM | a*A=1, A*a=1, b*B=1, B*b=1, c*C=1, C*c=1,
>          d*D=1, D*d=1, e*E=1, E*e=1, f*F=1, F*f=1, g*G=1, G*g=1,
>          a*b=c, b*c=d, c*d=e, d*e=f, e*f=g, f*g=a, g*a=b>;
> M := RWSMonoid(Q : TidyInt := 1000);
> print Id(M);
Id(M)
> print M!1;
Id(M)
> Order(M);
29
```

---

### 78.3.4 Arithmetic with Words

Having constructed a rewrite monoid  $M$  one can perform arithmetic with words in  $M$ . Assuming we have  $u, v \in M$  then the product  $u * v$  will be computed as follows:

- (i) The product  $w = u * v$  is formed as a product in the appropriate free monoid.
- (ii) The word  $w$  is reduced using the reduction machine associated with  $M$ .

If  $M$  is confluent, then  $w$  will be the unique minimal word that represents  $u * v$  under the ordering of  $M$ . If  $M$  is not confluent, then there are some pairs of words which are equal in  $M$ , but which reduce to distinct words, and hence  $w$  will not be a unique normal form. Note that:

- (i) Reduction of  $w$  can cause an increase in the length of  $w$ . At present there is an internal limit on the length of a word – if this limit is exceeded during reduction an error will be raised. Hence any word operation involving reduction can fail.
- (ii) The implementation is designed more with speed of execution in mind than with minimizing space requirements; thus, the reduction machine is always used to carry out word reduction, which can be space-consuming, particularly when the number of generators is large.

`u * v`

Product of the words  $w$  and  $v$ .

`u ^ n`

The  $n$ -th power of the word  $w$ , where  $n$  is a positive or zero integer.

`u eq v`

Given words  $w$  and  $v$  belonging to the same monoid, return true if  $w$  and  $v$  reduce to the same normal form, false otherwise. If  $M$  is confluent this tests for equality. If  $M$  is non-confluent then two words which are the same may not reduce to the same normal form.

`u ne v`

Given words  $w$  and  $v$  belonging to the same monoid, return false if  $w$  and  $v$  reduce to the same normal form, true otherwise. If  $M$  is confluent this tests for non-equality. If  $M$  is non-confluent then two words which are the same may reduce to different normal forms.

`IsId(w)``IsIdentity(w)`

Returns true if the word  $w$  is the identity word.

`#u`

The length of the word  $w$ .

`ElementToSequence(u)``Eltseq(u)`

The sequence  $Q$  obtained by decomposing the element  $u$  of a rewrite monoid into its constituent generators. Suppose  $u$  is a word in the rewrite monoid  $M$ . If  $u = M.i_1 \cdots M.i_m$ , then  $Q[j] = i_j$ , for  $j = 1, \dots, m$ .

---

### Example H78E10

We illustrate the word operations by applying them to elements of the Fibonacci monoid  $FM(2, 5)$ .

```
> FM<a,b,c,d,e> := FreeMonoid(5);
> Q:=quo< FM | a*b=c, b*c=d, c*d=e, d*e=a, e*a=b >;
> M<a,b,c,d,e> := RWSMonoid(Q);
> a*b*c*d;
b^2
> (c*d)^4 eq a;
true
> IsIdentity(a^0);
true
> IsIdentity(b^2*e);
false
```

---

## 78.4 Homomorphisms

For a general description of homomorphisms, we refer to Chapter 16. This section describes some special aspects of homomorphisms whose domain is a rewrite monoid.

### 78.4.1 General Remarks

Monoids in the category `MonRWS` currently are accepted as codomains only for monoid homomorphisms, whose codomain is a rewrite monoid as well.

### 78.4.2 Construction of Homomorphisms

`hom< M -> N | S >`

Returns the homomorphism from the rewrite group  $M$  to the monoid  $N$  defined by the expression  $S$  which must be the one of the following:

- (i) A list, sequence or indexed set containing the images of the  $n$  generators  $M.1, \dots, M.n$  of  $M$ . Here, the  $i$ -th element of  $S$  is interpreted as the image of  $M.i$ , i.e. the order of the elements in  $S$  is important.
- (ii) A list, sequence, enumerated set or indexed set, containing  $n$  tuples  $\langle x_i, y_i \rangle$  or arrow pairs  $x_i \rightarrow y_i$ , where  $x_i$  is a generator of  $M$  and  $y_i \in N$  ( $i = 1, \dots, n$ ) and the set  $\{x_1, \dots, x_n\}$  is the full set of generators of  $M$ . In this case,  $y_i$  is assigned as the image of  $x_i$ , hence the order of the elements in  $S$  is not important.

It is the user's responsibility to ensure that the provided generator images actually give rise to a well-defined homomorphism. No checking is performed by the constructor. Presently,  $N$  must be either a rewrite monoid or a group, and it is not possible to define a homomorphism by assigning images to the elements of an arbitrary generating set of  $M$ .

## 78.5 Set Operations

`Random(M, n)`

A random word of length at most  $n$  in the generators of  $M$ .

`Random(M)`

A random word (of length at most the order of  $M$ ) in the generators of  $M$ .

`Representative(M)`

`Rep(M)`

An element chosen from  $M$ .

Set( $M, a, b$ )
------------------

Search

MONSTGELT

Default : "DFS"

Create the set of words,  $w$ , in  $M$  with  $a \leq \text{length}(w) \leq b$ . If **Search** is set to "DFS" (depth-first search) then words are enumerated in lexicographical order. If **Search** is set to "BFS" (breadth-first-search) then words are enumerated in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker. Since the result is a set the words may not appear in the resultant set in the search order specified (although internally they will be enumerated in this order).

Set( $M$ )
------------

Search

MONSTGELT

Default : "DFS"

Create the set of words that is the carrier set of  $M$ . If **Search** is set to "DFS" (depth-first search) then words are enumerated in lexicographical order. If **Search** is set to "BFS" (breadth-first-search) then words are enumerated in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker. Since the result is a set the words may not appear in the resultant set in the search order specified (although internally they will be enumerated in this order).

Seq( $M, a, b$ )
------------------

Search

MONSTGELT

Default : "DFS"

Create the sequence  $S$  of words,  $w$ , in  $M$  with  $a \leq \text{length}(w) \leq b$ . If **Search** is set to "DFS" (depth-first search) then words will appear in  $S$  in lexicographical order. If **Search** is set to "BFS" (breadth-first-search) then words will appear in  $S$  in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker.

Seq( $M$ )
------------

Search

MONSTGELT

Default : "DFS"

Create a sequence  $S$  of words from the carrier set of  $M$ . If **Search** is set to "DFS" (depth-first search) then words will appear in  $S$  in lexicographical order. If **Search** is set to "BFS" (breadth-first-search) then words will appear in  $S$  in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker.

---

**Example H78E11**

We construct the group  $D_{22}$ , together with a representative word from the group, a random word and a random word of length at most 5 from the group, and the set of elements of the group.

```
> FM<a,b,c,d,e,f> := FreeMonoid(6);
> Q := quo< FM | a^2=1, f^2=1,
>           d*a=a*c, e*b=b*f, d*c=c*e, d*f=a*d, a*e=e*b, b*f*c=f >;
> M<a,b,c,d,e,f> := RWSMonoid(Q);
```

```

> print Order(M);
22
> print Representative(M);
Id(M)
> print Random(M);
c * e
> print Random(M, 5);
d
> Set(M);
{ a * c, e, a * d, f, a * e, a * f, Id(M), a * c * e, b * a,
  b * d, a * d * b, b * e, c * e, a * b * a, d * b, a * b * d,
  a, a * b * e, b, c, a * b, d }
> Seq(M : Search := "BFS");
[ Id(M), a, b, c, d, e, f, a * b, a * c, a * d, a * e, a * f,
  b * a, b * d, b * e, c * e, d * b, a * b * a, a * b * d,
  a * b * e, a * c * e, a * d * b ]

```

---

## 78.6 Conversion to a Finitely Presented Monoid

There is a standard way to convert a rewrite monoid into a finitely presented monoid using the function `Relations`. This is shown in the following example.

### Example H78E12

---

We construct the Fibonacci monoid  $FM(2,4)$  as a rewrite monoid, and then convert it into a finitely presented monoid.

```

> FM<a,b,c,d> := FreeMonoid(4);
> Q := quo< FM | a*b=c, b*c=d, c*d=a, d*a=b >;
> M := RWSMonoid(Q);
> Order(M);
11
> P<w,x,y,z> := quo < FM | Relations(M) >;
> P;

```

Finitely presented monoid

Relations

```

w * x = y
x * y = z
y * z = w
z * w = x
y^2 = w * z
z^2 = x * w
z * y = x^2
y * x = w^2
x * w * z = x^2
w^3 = w * z
x * w^2 = x * z

```

$$\begin{aligned}x^2 * z &= x \\w^2 * y &= w \\x^3 &= x * w \\x^2 * w &= z \\z * x &= x * z \\y * w &= w * y \\w^2 * z &= y \\x * w * y &= x\end{aligned}$$

---

## 78.7 Bibliography

- [Hol97] Derek Holt. *KB MAG – Knuth-Bendix in Monoids and Automatic Groups*. University of Warwick, 1997.
- [Sim94] Charles C. Sims. *Computation with finitely presented groups*. Cambridge University Press, Cambridge, 1994.