Gauss

Extended Gauss functionality for GAP

2024.11-01

3 December 2024

Simon Görtzen

Simon Görtzen
Email: simon.goertzen@rwth-aachen.de
Homepage: https://www.linkedin.com/in/simongoertzen/
Address: Simon Görtzen
Lehrstuhl B fuer Mathematik, RWTH Aachen
Templergraben 64
52062 Aachen
Germany

Abstract

This document explains the primary uses of the Gauss package. Included is a documented list of the most important methods and functions needed to work with sparse matrices and the algorithms provided by the Gauss package.

Copyright

Acknowledgements

The Gauss package would not have been possible without the helpful contributions by

- Max Neunhöffer, University of St Andrews, and
- Mohamed Barakat, Lehrstuhl B für Mathematik, RWTH Aachen.

Many thanks to these two and the Lehrstuhl B für Mathematik in general. It should be noted that the GAP algorithms for `SemiEchelonForm` and other methods formed an important and informative basis for the development of the extended Gaussian algorithms. This manual was created with the help of the GAPDoc package by F. Lübeck and M. Neunhöffer [LN08].

Contents

Chapter 1

Introduction

1.1 Overview over this manual

Chapter 1 is concerned with the technical details of installing and running this package. Chapter 2 answers the question why and how the GAP functionality concerning a sparse matrix type and gaussian algorithms was extended. The following chapters are concerned with the workings of the sparse matrix type (3) and sparse Gaussian algorithms (4). Included is a documented list of the most important methods and functions needed to work with sparse matrices and the algorithms provided by the Gauss package. Anyone interested in source code should just check out the files in the `gap/pkg/Gauss/gap/` folder (→ Appendix A).

1.2 Installation of the Gauss Package

To install this package just extract the package's archive file to the GAP `pkg/` directory. The Gauss package utilizes some C-code by Max Neunhoeffer that has to be compiled before you can load Gauss. To compile the code, first run `./configure`. If the package is not installed in the `pkg/` subdirectory of GAP's root directory you will need to provide the correct path to the latter. This will create a makefile. Complete the installation of the package by running `make`. Recompiling the documentation is possible by the command `make doc` in the Gauss directory, but this should not be necessary.

By default the Gauss package is not automatically loaded by GAP when it is installed. You must load the package with `LoadPackage("Gauss");` before its functions become available. Please, send me an e-mail if you have any questions, remarks, suggestions, etc. concerning Gauss. Also, I would like to hear about applications of this package.
Simon Goertzen

Chapter 2

Extending Gauss Functionality

2.1 The need for extended functionality

GAP has a lot of functionality for row echelon forms of matrices. These can be called by `SemiEchelonForm` and similar commands. All of these work for the GAP matrix type over fields. However, these algorithms are not capable of computing a reduced row echelon form (RREF) of a matrix, there is no way to "Gauss upwards". While this is not neccessary for things like Rank or Kernel computations, this was one in a number of missing features important for the development of the GAP package homalg by M. Barakat [Bar20].

Parallel to this development I worked on SCO [Gör08b], a package for creating simplicial sets and computing the cohomology of orbifolds, based on the paper "Simplicial Cohomology of Orbifolds" by I. Moerdijk and D. A. Pronk [MP99]. Very early on it became clear that the cohomology matrices (with entries in $\mathbb{Z}$ or finite quotients of $\mathbb{Z}$) would grow exponentially in size with the cohomology degree. At one point in time, for example, a 50651 x 1133693 matrix had to be handled.

It should be quite clear that there was a need for a sparse matrix data type and corresponding Gaussian algorithms. After an unfruitful search for a computer algebra system capable of this task, the Gauss package was born - to provide not only the missing RREF algorithms, but also support a new data type, enabling GAP to handle sparse matrices of almost arbitrary size.

I am proud to tell you that, thanks to optimizing the algorithms for matrices over GF(2), it was possible to compute the GF(2)-Rank of the matrix mentioned above in less than 20 minutes with a memory usage of about 3 GB.

2.2 The applications of the Gauss package algorithms

Please refer to [ht22] to find out more about the homalg project and its related packages. Most of the motivation for the algorithms in the Gauss package can be found there. If you are interested in this project, you might also want to check out my GaussForHomalg [Gör08a] package, which, just as RingsForHomalg [BGKL08] does for external Rings, serves as the connection between homalg and Gauss. By allowing homalg to delegate computational tasks to Gauss this small package extends homalg's capabilities to dense and sparse matrices over fields and rings of the form $\mathbb{Z}/\langle p^n \rangle$.

For those unfamiliar with the homalg project let me explain a couple of points. As outlined in [BR08] by D. Robertz and M. Barakat homological computations can be reduced to three basic tasks:

- Computing a row basis of a module (`BasisOfRowModule`).

- Reducing a module with a basis (`DecideZeroRows`).
- Compute the relations between module elements (`SyzygiesGeneratorsOfRows`).

In addition to these tasks only relatively easy tools for matrix manipulation are needed, ranging from addition and multiplication to finding the zero rows in a matrix. However, to reduce the need for communication it might be helpful to supply homalg with some more advanced procedures.

While the above tasks can be quite difficult when, for example, working in noncommutative polynomial rings, in the Gauss case they can all be done as long as you can compute a Reduced Row Echelon Form. This is clear for `BasisOfRowModule`, as the rows of the RREF of the matrix are already a basis of the module. `EchelonMat` (4.2.1) is used to compute RREFs, based on the GAP internal method `SemiEchelonMat` for Row Echelon Forms.

Lets look at the second point, the basic function `DecideZeroRows`: When you face the task of reducing a module A with a given basis B, you can compute the RREF of the following block matrix:

$$\begin{array}{|ccc|c|} \hline 1 & & & \\ & \ddots & & A \\ & & 1 & \\ \hline & 0 & & B \\ \hline \end{array}$$

By computing the RREF (notice how important "Gaussing upwards" is here) A is reduced with B. However, the left side of the matrix just serves the single purpose of tricking the Gaussian algorithms into doing what we want. Therefore, it was a logical step to implement `ReduceMat` (4.2.3), which does the same thing but without needing unneccessary columns.

Note: When, much later, it became clear that it was important to compute the transformation matrices of the reduction, `ReduceMatTransformation` (4.2.4) was born, similar to `EchelonMatTransformation` (4.2.2). This corresponds to the homalg procedure `DecideZeroRowsEffectively`.

The third procedure, `SygygiesGeneratorsOfRows`, is concerned with the relations between rows of a matrix, each row representing a module element. Over a field these relations are exactly the kernel of the matrix. One can easily see that this can be achieved by taking a matrix

$$\begin{array}{|c|ccc|} \hline & 1 & & \\ A & & \ddots & \\ & & & 1 \\ \hline \end{array}$$

and computing its Row Echelon Form. Then the row relations are generated by the rows to the right of the zero rows of the REF. There are two problems with this approach: The computation diagonalizes the kernel, which might not be wanted, and, much worse, it does not work at all for rings with zero divisors. For example, the 1×1 matrix $[2+8\mathbb{Z}]$ has a row relation $[4+8\mathbb{Z}]$ which would not have been found by this method.

Approaching this problem led to the method `EchelonMatTransformation` (4.2.2), which additionally computes the transformation matrix T, such that RREF $= T \cdot M$. Similar to `SemiEchelonMatTransformation`, T is split up into the rows needed to create the basis vectors of the RREF, and the relations that led to zero rows. Focussing on the computations over fields, it was an easy step to write `KernelMat` (4.2.5), which terminates after the REF and returns the kernel generators.

The syzygy computation over $\mathbb{Z}/\langle p^n\rangle$ was solved by carefully keeping track of basis vectors with a zero-divising head. If, for $v = (0,\ldots,0,h,*,\ldots,*), h \neq 0$, there exists $g \neq 0$ such that $g \cdot h = 0$, the vector $g \cdot v$ is regarded as an additional row vector which has to be reduced and can be reduced with. After some more work this allowed for the implementation of `KernelMat` (4.2.5) for matrices over $\mathbb{Z}/\langle p^n\rangle$.

This concludes the explanation of the so-called basic tasks Gauss has to handle when called by homalg to do matrix calculations. Here is a tabular overview of the current capabilities of Gauss (p is a prime, $n \in \mathbb{N}$):

Matrix Type:	Dense	Dense	Sparse	Sparse	Sparse
Base Ring:	Field	$\mathbb{Z}/\langle p^n\rangle$	Field	GF(2)	$\mathbb{Z}/\langle p^n\rangle$
RankMat	GAP	n.a.	+	++	n.a.
EchelonMat	+	-	+	++	+
EchelonMatTransf.	+	-	+	++	+
ReduceMat	+	-	+	++	+
ReduceMatTransf.	+	-	+	++	+
KernelMat	+	-	+	++	+

As you can see, the development of hermite algorithms was not continued for dense matrices. There are two reasons for that: GAP already has very good algorithms for $\mathbb{Z}$, and for small matrices the disadvantage of computing over $\mathbb{Z}$, potentially leading to coefficient explosion, is marginal.

Chapter 3

The Sparse Matrix Data Type

3.1 The inner workings of Gauss sparse matrices

When doing any kind of computation there is a constant conflict between memory load and speed. On the one hand, memory usage is bounded by the total available memory, on the other hand, computation time should also not exceed certain proportions. Memory usage and CPU time are generally inversely proportional, because the computer needs more time to perform operations on a compactified data structure. The idea of sparse matrices mirrors exactly the need for less memory load, therefore it is natural that sparse algorithms take more time than dense ones. However, if the matrix is sufficiently large and sparse at the same time, sparse algorithms can easily be faster than dense ones while maintaining minimal memory load.

It should be noted that, although matrices that appear naturally in homological algebra are almost always sparse, they do not have to stay sparse under (R)REF algorithms, especially when the computation is concerned with transformation matrices. Therefore, in a perfect world there should be ways implemented to not only find out which data structure to use, but also at what point to convert from one to the other. This was, however, not the aim of the Gauss package and is just one of many points in which this package could be optimized or extended. Take a look at this matrix M:

0	0	2	9	0
0	5	0	0	0
0	0	0	1	0

The matrix M carries the same information as the following table, if and only if you know how many rows and columns the matrix has. There is also the matter of the base ring, but this is not important for now:

(i,j)	Entry
(1,3)	2
(1,4)	9
(2,2)	5
(3,4)	1

This table relates each index tuple to its nonzero entry, all other matrix entries are defined to be zero. This only works for known dimensions of the matrix, otherwise trailing zero rows and columns could get lost (notice how the table gives no hint about the existence of a 5th column). To convert the above table into a sparse data structure, one could list the table entries in this way:

$$[[1,3,2],[1,4,9],[2,2,5],[3,4,1]]$$

However, this data structure would not be very efficient. Whenever you are interested in a row i of M (this happens all the time when performing Gaussian elimination) the whole list would have to be searched for 3-tuples of the form $[i,*,*]$. This is why I tried to manage the row index by putting the tuples into the corresponding list entry:

$$\begin{array}{l}[[3,2],[4,9]],\\ [[2,5]],\\ [[4,1]]]\end{array}$$

As you can see, this looks fairly complicated. However, the same information can be stored in this form, which would become the final data structure for Gauss sparse matrices:

$$\begin{array}{lll}\text{indices} := & [\,[\,3,\,4\,], & \text{entries}:= \quad [\,[\,2,\,9\,],\\ & [\,2\,], & \qquad\qquad\;\; [\,5\,],\\ & [\,4\,]\,] & \qquad\qquad\;\; [\,1\,]\,]\end{array}$$

Although now the number of rows is equal to the Length of both 'indices' and 'entries', it is still stored in the sparse matrix. Here is the full data structure (→ `SparseMatrix` (3.2.1)):

from SparseMatrix.gi

```
DeclareRepresentation( "IsSparseMatrixRep",
     IsSparseMatrix, [ "nrows", "ncols", "indices", "entries", "ring" ] );
```

As you can see, the matrix stores its ring to be on the safe side. This is especially important for zero matrices, as there is no way to determine the base ring from the sparse matrix structure. For further information on sparse matrix construction and converting, refer to `SparseMatrix` (3.2.1).

3.1.1 A special case: GF(2)

from SparseMatrix.gi

```
DeclareRepresentation( "IsSparseMatrixGF2Rep",
     IsSparseMatrix, [ "nrows", "ncols", "indices", "ring" ] );
```

Because the nonzero entries of a matrix over GF(2) are all "1", the entries of M are not stored at all. It is of course crucial that all operations and algorithms make 100% sure that all appearing zero entries are deleted from the 'indices' as well as the 'entries' list as they arise.

3.2 Methods and functions for sparse matrices

3.2.1 SparseMatrix (constructor using gap matrices)

▷ `SparseMatrix(`*`mat[, R]`*`)` (function)
Returns: a sparse matrix over the ring *R*

▷ `SparseMatrix(`*`nrows, ncols, indices`*`)` (function)
Returns: a sparse matrix over GF(2)

▷ `SparseMatrix(`*`nrows, ncols, indices, entries[, R]`*`)` (function)
Returns: a sparse matrix over the ring *R*

The sparse matrix constructor. In the one-argument form the SparseMatrix constructor converts a GAP matrix to a sparse matrix. If not provided the base ring *R* is found automatically. For the multi-argument form *nrows* and *ncols* are the dimensions of the matrix. *indices* must be a list of length *nrows* containing lists of the column indices of the matrix in ascending order.

Example

```
gap> M := [ [ 0 , 1 ], [ 3, 0 ] ] * One( GF(2) );
[ [ 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2) ] ]
gap> SM := SparseMatrix( M );
<a 2 x 2 sparse matrix over GF(2)>
gap> IsSparseMatrix( SM );
true
gap> Display( SM );
 . 1
 1 .
gap> SN := SparseMatrix( 2, 2, [ [ 2 ], [ 1 ] ] );
<a 2 x 2 sparse matrix over GF(2)>
gap> SN = SM;
true
gap> SN := SparseMatrix( 2, 3,
>                   [ [ 2 ], [ 1, 3 ] ],
>                   [ [ 1 ], [ 3, 2 ] ] * One( GF(5) ) );
<a 2 x 3 sparse matrix over GF(5)>
gap> Display( SN );
 . 1 .
 3 . 2
```

3.2.2 ConvertSparseMatrixToMatrix

▷ `ConvertSparseMatrixToMatrix(`*sm*`)` (method)

Returns: a GAP matrix, [], or a list of empty lists

This function converts the sparse matrix *sm* into a GAP matrix. In case of `nrows(sm)=0` or `ncols(sm)=0` the return value is the empty list or a list of empty lists, respectively.

Example

```
gap> M := [ [ 0 , 1 ], [ 3, 0 ] ] * One( GF(3) );
[ [ 0*Z(3), Z(3)^0 ], [ 0*Z(3), 0*Z(3) ] ]
gap> SM := SparseMatrix( M );
<a 2 x 2 sparse matrix over GF(3)>
gap> N := ConvertSparseMatrixToMatrix( SM );
[ [ 0*Z(3), Z(3)^0 ], [ 0*Z(3), 0*Z(3) ] ]
gap> M = N;
true
```

3.2.3 CopyMat

▷ `CopyMat(`*sm*`)` (method)

Returns: a shallow copy of the sparse matrix *sm*

3.2.4 GetEntry

▷ `GetEntry(`*sm*`, `*i*`, `*j*`)` (method)

Returns: a ring element.

This returns the entry sm[i,j] of the sparse matrix *sm*

3.2.5 SetEntry

▷ SetEntry(*sm, i, j, elm*) (method)

Returns: nothing.

This sets the entry sm[i,j] of the sparse matrix *sm* to *elm*.

3.2.6 AddToEntry

▷ AddToEntry(*sm, i, j, elm*) (method)

Returns: true or a ring element

AddToEntry adds the element *elm* to the sparse matrix *sm* at the *(i,j)*-th position. This is a Method with a side effect which returns true if you tried to add zero or the sum of sm[i,j] and *elm* otherwise. Please use this method whenever possible.

3.2.7 SparseZeroMatrix (constructor using number of rows)

▷ SparseZeroMatrix(*nrows[, ring]*) (function)

Returns: a sparse <*nrows* x *nrows*> zero matrix over the ring *ring*

▷ SparseZeroMatrix(*nrows, ncols[, ring]*) (function)

Returns: a sparse <*nrows* x *ncols*> zero matrix over the ring *ring*

3.2.8 SparseIdentityMatrix

▷ SparseIdentityMatrix(*dim[, ring]*) (function)

Returns: a sparse <*dim* x *dim*> identity matrix over the ring *ring*. If no ring is specified (one should try to avoid this if possible) the Rationals are the default ring.

3.2.9 TransposedSparseMat

▷ TransposedSparseMat(*sm*) (method)

Returns: the transposed matrix of the sparse matrix *sm*

3.2.10 CertainRows

▷ CertainRows(*sm, list*) (method)

Returns: the submatrix sm{[list]} as a sparse matrix

3.2.11 CertainColumns

▷ CertainColumns(*sm, list*) (method)

Returns: the submatrix sm{[1..nrows(sm)]}{[list]} as a sparse matrix

3.2.12 SparseUnionOfRows (for a list of sparse matrices)

▷ SparseUnionOfRows(*L*) (function)

Returns: a sparse matrix

Stack the sparse matrices in the non-empty list *L*.

3.2.13 SparseUnionOfColumns (for a list of sparse matrices)

▷ `SparseUnionOfColumns(L)` (function)

Returns: a sparse matrix

Augment the sparse matrices in the non-empty list `L`.

3.2.14 SparseDiagMat

▷ `SparseDiagMat(list)` (function)

Returns: the block diagonal matrix composed of the sparse matrices in `list`

3.2.15 Nrows

▷ `Nrows(sm)` (method)

Returns: the number of rows of the sparse matrix `sm`. This should be preferred to the equivalent `sm!.nrows`.

3.2.16 Ncols

▷ `Ncols(sm)` (method)

Returns: the number of columns of the sparse matrix `sm`. This should be preferred to the equivalent `sm!.ncols`.

3.2.17 IndicesOfSparseMatrix

▷ `IndicesOfSparseMatrix(sm)` (method)

Returns: the indices of the sparse matrix `sm` as a ListList. This should be preferred to the equivalent `sm!.indices`.

3.2.18 EntriesOfSparseMatrix

▷ `EntriesOfSparseMatrix(sm)` (method)

Returns: the entries of the sparse matrix `sm` as a ListList of ring elements. This should be preferred to the equivalent `sm!.entries` and has the additional advantage of working for sparse matrices over GF(2) which do not have any entries.

3.2.19 RingOfDefinition

▷ `RingOfDefinition(sm)` (method)

Returns: the base ring of the sparse matrix `sm`. This should be preferred to the equivalent `sm!.ring`.

Chapter 4

Gaussian Algorithms

4.1 A list of the available algorithms

As decribed earlier, the main functions of Gauss are `EchelonMat` (4.2.1) and `EchelonMatTransformation` (4.2.2), `ReduceMat` (4.2.3) and `ReduceMatTransformation` (4.2.4), `KernelMat` (4.2.5) and, additionally `Rank` (4.2.6). These are all documented in the next section, but of course rely on specific algorithms depending on the base ring of the matrix. These are not fully documented but it should be very easy to find out how they work based on the documentation of the main functions.

EchelonMat		
	Field:	`EchelonMatDestructive`
	Ring:	`HermiteMatDestructive`
EchelonMatTransformation		
	Field:	`EchelonMatTransformationDestructive`
	Ring:	`HermiteMatTransformationDestructive`
ReduceMat		
	Field:	`ReduceMatWithEchelonMat`
	Ring:	`ReduceMatWithHermiteMat`
ReduceMatTransformation		
	Field:	`ReduceMatWithEchelonMatTransformation`
	Ring:	`ReduceMatWithHermiteMatTransformation`
KernelMat		
	Field:	`KernelEchelonMatDestructive`
	Ring:	`KernelHermiteMatDestructive`
Rank		
	Field (dense):	`Rank` (GAP method)
	Field (sparse):	`RankDestructive`
	GF(2) (sparse):	`RankOfIndicesListList`
	Ring:	n.a.

4.2 Methods and Functions for Gaussian algorithms

4.2.1 EchelonMat

▷ `EchelonMat(mat)` (method)

Returns: a record that contains information about an echelonized form of the matrix *mat*.

The components of this record are

'vectors'

the reduced row echelon / hermite form of the matrix *mat* without zero rows.

'heads'

list that contains at position <i>, if nonzero, the number of the row for that the pivot element is in column <i>.

computes the reduced row echelon form RREF of a dense or sparse matrix *mat* over a field, or the hermite form of a sparse matrix *mat* over $\mathbb{Z}/ < p^n >$.

Example

```
gap> M := [[0,0,0,1,0],[0,1,1,1,1],[1,1,1,1,0]] * One( GF(2) );;
gap> Display(M);
 . . . 1 .
 . 1 1 1 1
 1 1 1 1 .
gap> EchelonMat(M);
rec( heads := [ 1, 2, 0, 3, 0 ],
  vectors := [ <an immutable GF2 vector of length 5>,
      <an immutable GF2 vector of length 5>,
      <an immutable GF2 vector of length 5> ] )
gap> Display( last.vectors );
 1 . . . 1
 . 1 1 . 1
 . . . 1 .
gap> SM := SparseMatrix( M );
<a 3 x 5 sparse matrix over GF(2)>
gap> EchelonMat( SM );
rec( heads := [ 1, 2, 0, 3, 0 ], vectors := <a 3 x 5 sparse matrix over GF(
    2)> )
gap> Display(last.vectors);
 1 . . . 1
 . 1 1 . 1
 . . . 1 .
gap> SM := SparseMatrix( [[7,4,5],[0,0,6],[0,4,4]] * One( Integers mod 8 ) );
<a 3 x 3 sparse matrix over (Integers mod 8)>
gap> Display( SM );
 7 4 5
 . . 6
 . 4 4
gap> EchelonMat( SM );
rec( heads := [ 1, 2, 3 ],
  vectors := <a 3 x 3 sparse matrix over (Integers mod 8)> )
gap> Display( last.vectors );
 1 . 1
 . 4 .
 . . 2
```

4.2.2 EchelonMatTransformation

▷ `EchelonMatTransformation(mat)` (method)

Returns: a record that contains information about an echelonized form of the matrix `mat`.

The components of this record are

'vectors'

the reduced row echelon / hermite form of the matrix `mat` without zero rows.

'heads'

list that contains at position <i>, if nonzero, the number of the row for that the pivot element is in column <i>.

'coeffs'

the transformation matrix needed to obtain the RREF from `mat`.

'relations'

the kernel of the matrix `mat` if RingOfDefinition(`mat`) is a field. Otherwise these are only the obvious row relations of `mat`, there might be more kernel vectors - → `KernelMat` (4.2.5).

computes the reduced row echelon form RREF of a dense or sparse matrix `mat` over a field, or the hermite form of a sparse matrix `mat` over $\mathbb{Z}/ < p^n >$. In either case, the transformation matrix T is calculated as the row union of 'coeffs' and 'relations'.

Example

```
gap> M := [[1,0,1],[1,1,0],[1,0,1],[1,1,0],[1,1,1]] * One( GF(2) );;
gap> EchelonMatTransformation( M );
rec(
  coeffs := [ <an immutable GF2 vector of length 5>,
      <an immutable GF2 vector of length 5>,
      <an immutable GF2 vector of length 5> ], heads := [ 1, 2, 3 ],
  relations :=
    [ <an immutable GF2 vector of length 5>,
      <an immutable GF2 vector of length 5> ],
  vectors := [ <an immutable GF2 vector of length 3>,
      <an immutable GF2 vector of length 3>,
      <an immutable GF2 vector of length 3> ] )
gap> Display(last.vectors);
 1 . .
 . 1 .
 . . 1
gap> Display(last.coeffs);
 1 1 . . 1
 1 . . . 1
 . 1 . . 1
gap> Display(last.relations);
 1 . 1 . .
 . 1 . 1 .
gap> Display( Concatenation( last.coeffs, last.relations ) * M );
 1 . .
 . 1 .
 . . 1
 . . .
 . . .
gap> SM := SparseMatrix( M );
<a 5 x 3 sparse matrix over GF(2)>
gap> EchelonMatTransformation( SM );
```

```
rec( coeffs := <a 3 x 5 sparse matrix over GF(2)>,
  heads := [ 1, 2, 3 ],
  relations := <a 2 x 5 sparse matrix over GF(2)>,
  vectors := <a 3 x 3 sparse matrix over GF(2)> )
gap> Display(last.vectors);
 1 . .
 . 1 .
 . . 1
gap> Display(last.coeffs);
 1 1 . . 1
 1 . . . 1
 . 1 . . 1
gap> Display(last.relations);
 1 . 1 . .
 . 1 . 1 .
gap> Display( SparseUnionOfRows( [ last.coeffs, last.relations ] ) * SM );
 1 . .
 . 1 .
 . . 1
 . . .
 . . .
```

4.2.3 ReduceMat

▷ `ReduceMat(A, B)` (method)

Returns: a record with a single component 'reduced_matrix' := M. M is created by reducing A with B, where B must be in Echelon/Hermite form. M will have the same dimensions as A.

Example

```
gap> M := [[0,0,0,1,0],[0,1,1,1,1],[1,1,1,1,0]] * One( GF(2) );;
gap> Display(M);
 . . . 1 .
 . 1 1 1 1
 1 1 1 1 .
gap> N := [[1,1,0,0,0],[0,0,1,0,1]] * One( GF(2) );;
gap> Display(N);
 1 1 . . .
 . . 1 . 1
gap> ReduceMat(M,N);
rec(
  reduced_matrix := [ <a GF2 vector of length 5>, <a GF2 vector of length 5>,
      <a GF2 vector of length 5> ] )
gap> Display(last.reduced_matrix);
 . . . 1 .
 . 1 . 1 .
 . . . 1 1
gap> SM := SparseMatrix(M); SN := SparseMatrix(N);
<a 3 x 5 sparse matrix over GF(2)>
<a 2 x 5 sparse matrix over GF(2)>
gap> ReduceMat(SM,SN);
rec( reduced_matrix := <a 3 x 5 sparse matrix over GF(2)> )
gap> Display(last.reduced_matrix);
 . . . 1 .
```

```
. 1 . 1 .
. . . 1 1
```

4.2.4 ReduceMatTransformation

▷ `ReduceMatTransformation(A, B)` (method)

Returns: a record with a component 'reduced_matrix' := M. M is created by reducing *A* with *B*, where *B* must be in Echelon/Hermite form. M will have the same dimensions as *A*. In addition to the (identical) output as ReduceMat this record also includes the component 'transformation', which stores the row operations that were needed to reduce *A* with *B*. This differs from "normal" transformation matrices because only rows of *B* had to be moved. Therefore, the transformation matrix solves M = A + T * B.

Example

```
gap> M := [[0,0,0,1,0],[0,1,1,1,1],[1,1,1,1,0]] * One( GF(2) );;
gap> Display(M);
 . . . 1 .
 . 1 1 1 1
 1 1 1 1 .
gap> N := [[1,1,0,0,0],[0,0,1,0,1]] * One( GF(2) );;
gap> Display(N);
 1 1 . . .
 . . 1 . 1
gap> ReduceMatTransformation(M,N);
rec(
  reduced_matrix :=
    [ <a GF2 vector of length 5>, <a GF2 vector of length 5>,
      <a GF2 vector of length 5> ],
  transformation := [ <a GF2 vector of length 2>,
      <a GF2 vector of length 2>, <a GF2 vector of length 2> ] )
gap> Display(last.reduced_matrix);
 . . . 1 .
 . 1 . 1 .
 . . . 1 1
gap> Display(last.transformation);
 . .
 . 1
 1 1
gap> Display( M + last.transformation * N );
 . . . 1 .
 . 1 . 1 .
 . . . 1 1
gap> SM := SparseMatrix(M); SN := SparseMatrix(N);
<a 3 x 5 sparse matrix over GF(2)>
<a 2 x 5 sparse matrix over GF(2)>
gap> ReduceMatTransformation(SM,SN);
rec( reduced_matrix := <a 3 x 5 sparse matrix over GF(2)>,
  transformation := <a 3 x 2 sparse matrix over GF(2)> )
gap> Display(last.reduced_matrix);
 . . . 1 .
 . 1 . 1 .
 . . . 1 1
```

```
gap> Display(last.transformation);
 . .
 . 1
 1 1
gap> Display( SM + last.transformation * SN );
 . . . 1 .
 . 1 . 1 .
 . . . 1 1
```

4.2.5 KernelMat

▷ KernelMat(*M*) (function)

Returns: a record with a single component 'relations'.

If *M* is a matrix over a field this is the same output as EchelonMatTransformation (4.2.2) provides in the 'relations' component, but with less memory and CPU usage. If the base ring of *M* is a non-field, the Kernel might have additional generators, which are added to the output.

Example

```
gap> M := [[2,1],[0,2]];
[ [ 2, 1 ], [ 0, 2 ] ]
gap> SM := SparseMatrix( M * One( GF(3) ) );
<a 2 x 2 sparse matrix over GF(3)>
gap> KernelMat(SM);
rec( relations := <a 0 x 2 sparse matrix over GF(3)> )
gap> SN := SparseMatrix( M * One( Integers mod 4 ) );
<a 2 x 2 sparse matrix over (Integers mod 4)>
gap> KernelMat(SN);
rec( relations := <a 1 x 2 sparse matrix over (Integers mod 4)> )
gap> Display(last.relations);
 2 1
```

4.2.6 Rank

▷ Rank(*sm*[, *boundary*]) (method)

Returns: the rank of the sparse matrix *sm*. Only works for fields.

Computes the rank of a sparse matrix. If the optional argument *boundary* is provided, some algorithms take into account the fact that Rank(*sm*) <= *boundary*, thus possibly terminating earlier.

Example

```
gap> M := SparseDiagMat( ListWithIdenticalEntries( 10,
>         SparseMatrix( [[1,1],[1,1]] * One( GF(5) ) ) ) );
<a 20 x 20 sparse matrix over GF(5)>
gap> Display(M);
 1 1 . . . . . . . . . . . . . . . . . .
 1 1 . . . . . . . . . . . . . . . . . .
 . . 1 1 . . . . . . . . . . . . . . . .
 . . 1 1 . . . . . . . . . . . . . . . .
 . . . . 1 1 . . . . . . . . . . . . . .
 . . . . 1 1 . . . . . . . . . . . . . .
 . . . . . . 1 1 . . . . . . . . . . . .
 . . . . . . 1 1 . . . . . . . . . . . .
 . . . . . . . . 1 1 . . . . . . . . . .
```

```
  . . . . . . . . 1 1 . . . . . . . . . .
  . . . . . . . . . . 1 1 . . . . . . . .
  . . . . . . . . . . 1 1 . . . . . . . .
  . . . . . . . . . . . . 1 1 . . . . . .
  . . . . . . . . . . . . 1 1 . . . . . .
  . . . . . . . . . . . . . . 1 1 . . . .
  . . . . . . . . . . . . . . 1 1 . . . .
  . . . . . . . . . . . . . . . . 1 1 . .
  . . . . . . . . . . . . . . . . 1 1 . .
  . . . . . . . . . . . . . . . . . . 1 1
  . . . . . . . . . . . . . . . . . . 1 1
gap> Rank(M);
10
```

Appendix A

An Overview of the Gauss package source code

Filename	Content
SparseMatrix.gi	Definitions and methods for the sparse matrix type
SparseMatrixGF2.gi	Special case GF(2): no matrix entries needed
GaussDense.gi	Gaussian elmination for GAP matrices over fields
Sparse.gi	Documentation and forking depending on the base ring
GaussSparse.gi	Gaussian elimination for sparse matrices over fields
HermiteSparse.gi	Hermite elimination for sparse matrices over $\mathbb{Z}/\langle p^n \rangle$

Table: *The Gauss package files.*

References

[Bar20] M. Barakat. *homalg -- A homological algebra meta-package for computable Abelian categories*, 2007-2020. https://homalg-project.github.io/pkg/homalg. 5

[BGKL08] M. Barakat, S. Görtzen, M. Kirschmer, and M. Lange-Hegermann. *RingsForHomalg -- Dictionaries of external rings*, 2007-2008. https://homalg-project.github.io/pkg/RingsForHomalg/. 5

[BR08] M. Barakat and D. Robertz. homalg -- A Meta-Package for Homological Algebra. *J. Algebra Appl.*, 7(3):299--317, 2008. https://arxiv.org/abs/math/0701146. 5

[Gör08a] S. Görtzen. *GaussForHomalg -- Gauss functionality for the homalg project*, 2007-2008. https://homalg-project.github.io/pkg/GaussForHomalg/. 5

[Gör08b] S. Görtzen. *SCO -- Simplicial Cohomology of Orbifolds*, 2007-2008. https://homalg-project.github.io/pkg/SCO/. 5

[ht22] The homalg team. *The homalg project*, 2003-2022. https://homalg-project.github.io/prj/homalg_project/. 5

[LN08] F. Lübeck and M. Neunhöffer. *GAP Package GAPDoc*, 2007-2008. http://www.math.rwth-aachen.de/~Frank.Luebeck/GAPDoc. 2

[MP99] I. Moerdijk and D. A. Pronk. Simplicial cohomology of orbifolds. *Indag. Math. (N.S.)*, 10(2):269--293, 1999. 5

Index