EDIM

Elementary Divisors and Integer Matrices

VERSION 1.3.2

June 2013

Frank Lübeck

Frank Lübeck Email: Frank.Luebeck@Math.RWTH-Aachen.De
Homepage: http://www.math.rwth-aachen.de/~Frank.Luebeck
Address: Lehrstuhl D für Mathematik RWTH Aachen Templergraben 64 52062 Aachen Germany

Copyright

Contents

Chapter 1

The EDIM-Package

(Elementary Divisors and Integer Matrices, by Frank Lübeck)

This chapter describes the functions defined in the GAP4 package EDIM. The main functions implement variants of an algorithm for computing for a given prime p the p-parts of the elementary divisors of an integer matrix. These algorithms use a p-adic method and are described by the author in [LÃ¼02] (see `ElementaryDivisorsPPartRk` (1.2.1)).

These functions were already applied to integer matrices of dimension greater than 11000 (which had many non-trivial elementary divisors which were products of small primes).

Furthermore there are functions for finding the biggest elementary divisor of an invertible integer matrix and the inverse of a rational invertible matrix (see `ExponentSquareIntMatFullRank` (1.3.3) and `InverseRatMat` (1.3.1)). These algorithms use p-adic approximations, explained in 1.7.

Finally we distribute implementations of some other algorithms for finding elementary divisors or normal forms of integer matrices: A p-modular algorithm by Havas and Sterling from [HS79] (see `ElementaryDivisorsPPartHavasSterling` (1.2.2)) and LLL-based algorithms for extended greatest common divisors of integers (see `GcdexIntLLL` (1.5.1)) and for Hermite normal forms of integer matrices with (very nice) transforming matrices (see `HermiteIntMatLLL` (1.5.2)).

By default the EDIM is automatically loaded by GAP when it is installed. If the automatic loading is disabled in your installation you must load the package with `RequirePackage("edim");` before its functions become available.

Please, send me an e-mail (`Frank.Luebeck@Math.RWTH-Aachen.De`) if you have any questions, remarks, suggestions, etc. concerning this mini-package. Also, I would like to hear about applications of this package.

Frank Lübeck

1.1 Installation of the EDIM package

To install this package first unpack it inside some GAP root directory into the subdirectory `pkg/edim` (see (**Reference: Installing a GAP Package**)). Then the EDIM package can already be loaded and used. But we strongly recommend to compile a kernel function as well during installation, otherwise the function `ElementaryDivisorsPPartRkExpSmall` (1.2.1) will not be available.

To install the kernel function go to the directory `pkg/edim` to which the package was extracted and call

```
/bin/sh ./configure [path] [CONFIGNAME=...]
```

where `path` is a path to the main GAP root directory (if not given, the default `../..` is assumed). If you have installed several GAP kernels you can compile the corresponding EDIM kernel function by specifying the `CONFIGNAME` that was used to configure that kernel.

Afterwards call `make` to compile a binary file.

If you have installed several GAP kernels repeat these two steps for each of them, using the various values of `CONFIGNAME`. You can run a test of the installation by typing `make test`.

1.1.1 InfoEDIM

▷ `InfoEDIM` (info class)

This is an `Info` class for the EDIM-package. By `SetInfoLevel(InfoEDIM, 1);` you can switch on the printing of some information during the computations of certain EDIM-functions.

1.2 *p*-Parts of Elementary Divisors

Here we explain the main functions of the package.

1.2.1 ElementaryDivisorsPPartRk

▷ `ElementaryDivisorsPPartRk(A, p[, rk])` (function)
▷ `ElementaryDivisorsPPartRkI(A, p, rk)` (function)
▷ `ElementaryDivisorsPPartRkII(A, p, rk)` (function)
▷ `ElementaryDivisorsPPartRkExp(A, p, rk, exp)` (function)
▷ `ElementaryDivisorsPPartRkExpSmall(A, p, rk, exp, il)` (function)

These functions return a list $[m_1, m_2, \ldots, m_r]$ where m_i is the number of nonzero elementary divisors of `A` divisible by p^i (see `ElementaryDivisorsMat` (**Reference: ElementaryDivisorsMat**) for a definition of the elementary divisors).

The algorithms for these functions are described in [LÃ¼02].

`A` must be a matrix with integer entries, `p` a prime, and `rk` the rank of `A` (as rational matrix). In the first version of the command `rk` is computed, if it is not given.

The first version of the command delegates its job to the fourth version by trying growing values for `exp`, see below.

The second and third versions implement the main algorithm described in [LÃ¼02] and a variation. Here `ElementaryDivisorsPPartRkII` has a bit more overhead, but can be advantageous because the intermediate entries during the computation can be much smaller.

In the fourth form `exp` must be an upper bound for the highest power of `p` appearing in an elementary divisor of `A`. This information allows reduction of matrix entries modulo p^{exp} during the computation.

If `exp` is too small or the given `rk` is not correct the function returns 'fail'.

As long as p^{exp} is smaller than 2^{28} and p^{exp+2} is smaller than 2^{31} we use internally a kernel function which can also be used directly in the fifth form of the command. There `il` can be 0 or 1 where in the second case some information is printed during the computation.

This last form of the function was already succesfully applied to dense matrices of rank up to 11000.

Note that you have to compile a file (see 1.1) while installing this package, if you want to have this kernel function available.

Example

```
gap> ReadPackage("edim",  "tst/mat");
Reading 242x242 integer matrix 'mat' with elementary divisors 'eldiv'.
true
gap> ElementaryDivisorsPPartRkI(mat, 2, 242); time; # mat has full rank
[ 94, 78, 69, 57, 23, 23, 9, 2, 2, 0 ]
490
gap> ElementaryDivisorsPPartRkExpSmall(mat, 2, 242, 10, 0); time;
[ 94, 78, 69, 57, 23, 23, 9, 2, 2, 0 ]
10
```

1.2.2 ElementaryDivisorsPPartHavasSterling

▷ `ElementaryDivisorsPPartHavasSterling(A, p, d)` (function)

For an integer matrix `A` and a prime p this function returns a list $[m_1, m_2, \ldots, m_r]$ where m_i is the number of nonzero elementary divisors of `A` divisible by p^i.

An upper bound `d` for the highest power of p appearing in an elementary divisor of `A` must be given. Smaller `d` improve the performance of the algorithm considerably.

This is an implementation of the modular algorithm described in [HS79].

We added a slight improvement: we divide the considered submatrices by the p-part of the greatest common divisor of all entries (and lower the `d` appropriately). This reduces the size of the entries and often shortens the pivot search.

Example

```
gap> ReadPackage("edim",  "tst/mat");
Reading 242x242 integer matrix 'mat' with elementary divisors 'eldiv'.
true
gap> ElementaryDivisorsPPartHavasSterling(mat, 2, 10); time;
[ 94, 78, 69, 57, 23, 23, 9, 2, 2 ]
1260
```

1.3 Inverse of Rational Matrices

1.3.1 InverseRatMat

▷ `InverseRatMat(A[, p])` (function)

This function returns the inverse of an invertible matrix over the rational numbers.

It first computes the inverse modulo some prime p, computes from this a p-adic approximation to the inverse and finally constructs the rational entries from their p-adic approximations. See section 1.7 for more details.

This seems to be better than GAP's standard Gauß algorithm (`A^-1`) already for small matrices. (Try, e.g., `RandomMat(20,20,[-10000..10000])` or `RandomMat(100,100)`.)

The optional argument p should be a prime such that `A` modulo p is invertible (default is $p = 251$). If `A` is not invertible modulo p then p is automatically replaced by the next prime.

1.3.2 RationalSolutionIntMat

▷ `RationalSolutionIntMat(A, v[, p[, invA]])` (function)

This function returns the solution x of the system of linear equations $xA = v$.

Here, `A` must be a matrix with integer entries which is invertible over the rationals and `v` must be a vector with integer entries of the appropriate length.

The optional arguments are a prime p such that `A` $\pmod p$ is invertible (if not given, $p = 251$ is assumed) and the inverse `invA` of `A` $\pmod p$.

The solution is computed via p-adic approximation as explained in 1.7.

1.3.3 ExponentSquareIntMatFullRank

▷ `ExponentSquareIntMatFullRank(A[, p[, nr]])` (function)

This function returns the biggest elementary divisor of a square integer matrix `A` of full rank.

For such a matrix `A` the least common multiple of the denominators of all entries of the inverse matrix A^{-1} is exactly the biggest elementary divisor of `A`.

This function is implemented by a slight modification of `InverseRatMat` (1.3.1). The third argument `nr` tells the function to return the least common multiple of the first `nr` rows of the rational inverse matrix only. Very often the function will already return the biggest elementary divisor with `nr` $= 2$ or 3 (and the command without this argument would spend most time in checking, that this is correct).

The optional argument `p` should be a prime such that `A` modulo `p` is invertible (default is $p = 251$). If `A` is not invertible modulo `p` then `p` is automatically replaced by the next prime.

Example

```
gap> ReadPackage("edim",  "tst/mat");
Reading 242x242 integer matrix 'mat' with elementary divisors 'eldiv'.
true
gap> inv := InverseRatMat(mat);; time;
840
gap> ExponentSquareIntMatFullRank(mat, 101, 3); # same as without the '3'
115200
```

1.4 All Elementary Divisors Using p-adic Method

In the following two functions we put things together. In particular we handle the prime parts of the elementary divisors efficiently for primes appearing with low powers in the highest elementary divisor respectively determinant divisor.

1.4.1 ElementaryDivisorsSquareIntMatFullRank

▷ `ElementaryDivisorsSquareIntMatFullRank(A)` (function)

This function returns a list of nonzero elementary divisors of an integer matrix `A`.

Here we start with computing the biggest elementary divisor via `ExponentSquareIntMatFullRank` (1.3.3). If it runs into a problem because `A` is singular

modulo a choosen prime (it starts by default with 251) then the prime is automatically replaced by the next one.

The rest is done using `ElementaryDivisorsPPartRkExp` (1.2.1) and `RankMod` (1.6.5).

The function fails if the biggest elementary divisor cannot be completely factored and the non-factored part is not a divisor of the biggest elementary divisor only.

Note that this function may for many matrices not be the best choice for computing all elementary divisors. You may first try the standard GAP library routines for Smith normal form instead of this function. Nevertheless remember `ElementaryDivisorsSquareIntMatFullRank` for hard and big examples. It is particularly good when the largest elementary divisor is a very small factor of the determinant.

Example

```
gap> Collected(ElementaryDivisorsSquareIntMatFullRank(mat));
[ [ 1, 49 ], [ 3, 99 ], [ 6, 7 ], [ 30, 9 ], [ 60, 9 ], [ 120, 2 ],
  [ 360, 10 ], [ 720, 22 ], [ 3600, 12 ], [ 14400, 14 ],
  [ 28800, 7 ], [ 115200, 2 ] ]
gap> time;
860
gap> last2 = Collected(DiagonalOfMat(NormalFormIntMat(mat, 1).normal));
true
gap> time;
5170
```

1.4.2 ElementaryDivisorsIntMatDeterminant

▷ `ElementaryDivisorsIntMatDeterminant(`*A*`, `*det*`[, `*rk*`])` (function)

This function returns a list of nonzero elementary divisors of an integer matrix *A*.

Here *det* must be an integer which is a multiple of the biggest determinant divisor of *A*. If the matrix does not have full rank then its rank *rk* must be given, too.

The argument *det* can be given in the form of `Collected(FactorsInt(`*det*`))`.

This function handles prime divisors of *det* with multiplicity smaller than 4 specially, for the other prime divisors p it delegates to `ElementaryDivisorsPPartRkExp` (1.2.1) where the *exp* argument is the multiplicity of the p in *det*. (Note that this is not very good when p has actually a much smaller multiplicity in the largest elementary divisor.)

Example

```
gap> ReadPackage("edim",  "tst/mat");
Reading 242x242 integer matrix 'mat' with elementary divisors 'eldiv'.
true
gap> # not so good:
gap> ElementaryDivisorsIntMatDeterminant(mat,Product(eldiv)) =
> Concatenation([1..49]*0+1, eldiv); time;
true
5490
```

1.5 Gcd and Normal Forms Using LLL

The EDIM-mini package also contains implementations of an extended Gcd-algorithm for integers and a Hermite and Smith normal form algorithm for integer matrices using LLL-techiques. They are well described in the paper [HMM98] by Havas, Majewski and Matthews.

They are particularly useful if one wants to have the normal forms together with transforming matrices. These transforming matrices have spectacularly nice (i.e., "small") entries in cases of input matrices which are non-square or not of full rank (otherwise the transformation to the Hermite normal form is unique).

In detail:

1.5.1 GcdexIntLLL

▷ `GcdexIntLLL(n1, n2, ...)` (function)

This function returns for integers $n1, n2, \ldots$ a list $[g,[c_1,c_2,\ldots]]$, where $g = c_1 n1 + c_2 n2 + \ldots$ is the greatest common divisor of the ni. Here all the c_i are usually very small.

Example

```
gap> GcdexIntLLL( 517, 244, -304, -872, -286, 854, 866, 224, -765, -38);
[ 1, [ 0, 0, 0, 0, 1, 0, 1, 1, 1, 1 ] ]
```

1.5.2 HermiteIntMatLLL

▷ `HermiteIntMatLLL(A)` (function)

This returns the Hermite normal form of an integer matrix A and uses the LLL-algorithm to avoid entry explosion.

1.5.3 HermiteIntMatLLLTrans

▷ `HermiteIntMatLLLTrans(A)` (function)

This function returns a pair of matrices $[H,L]$ where $H = LA$ is the Hermite normal form of an integer matrix A. The transforming matrix L can have surprisingly small entries.

Example

```
gap> ReadPackage("edim",  "tst/mat2");
Reading 34x34 integer matrix 'mat2' with elementary divisors 'eldiv2'.
true
gap> tr := HermiteIntMatLLLTrans(mat2);; Maximum(List(Flat(tr[2]), AbsInt));
606
gap> tr[2]*mat2 = tr[1];
true
```

1.5.4 SmithIntMatLLL

▷ `SmithIntMatLLL(A)` (function)

This function returns the Smith normal form of an integer matrix A using the LLL-algorithm to avoid entry explosion.

1.5.5 SmithIntMatLLLTrans

▷ `SmithIntMatLLLTrans(A)` (function)

This function returns $[S,L,R]$ where $S = LAR$ is the Smith normal form of an integer matrix `A`.

We apply the algorithm for Hermite normal form several times to get the Smith normal form, that is not in the paper [HMM98]. The transforming matrices need not be as nice as for the Hermite normal form.

Example

```
gap> ReadPackage("edim",  "tst/mat2");
Reading 34x34 integer matrix 'mat2' with elementary divisors 'eldiv2'.
true
gap> tr := SmithIntMatLLLTrans(mat2);;
gap> tr[2] * mat2 * tr[3] = tr[1];
true
```

1.6 Utility Functions from the EDIM-package

1.6.1 RatNumberFromModular

▷ `RatNumberFromModular(n, k, l, x)` (function)

This function returns $r/s = x \pmod n$, if it exists. More precisely:

`n`, `k`, `l` must be positive integers with $2kl \leq n$ and `x` an integer with $-n/2 < x \leq n/2$. If it exists this function returns a rational number r/s with $0 < s < l$, $\gcd(s,n) = 1$, $-k < r < k$ and r/s congruent to $x \pmod n$ (i.e., $n \mid r - sx$). Such an r/s is unique. The function returns `fail` if such a number does not exist.

1.6.2 InverseIntMatMod

▷ `InverseIntMatMod(A, p)` (function)

This function returns an inverse matrix modulo a prime `p` or `fail`. More precisely:

`A` must be an integer matrix and `p` a prime such that `A` is invertible modulo `p`. This function returns an integer matrix `inv` with entries in the range $]-p/2 \ldots p/2]$ such that `invA` reduced modulo p is the identity matrix.

It returns `fail` if the inverse modulo `p` does not exist. This function is particularly fast for primes smaller 256.

1.6.3 HadamardBoundIntMat

▷ `HadamardBoundIntMat(A)` (function)

The Hadamard bound for a square integer matrix `A` is the product of Euclidean norms of the nonzero rows (or columns) of `A`. It is an upper bound for the absolute value of the determinant of `A`.

1.6.4 CheapFactorsInt

▷ `CheapFactorsInt(n[, nr])` (function)

This function returns a list of factors of an integer `n`, including "small" prime factors - here the optional argument `nr` is the number of iterations for 'FactorsRho' (default is 2000).

This is only a slight modification of the library function `FactorsInt` (**Reference: FactorsInt**) which avoids an error message when the number is not completely factored.

1.6.5 RankMod

▷ `RankMod(A, p)` (function)

This function returns the rank of an integer matrix `A` modulo `p`. Here `p` must not necessarily be a prime. If it is not and this function returns an integer, then this is the rank of `A` for all prime divisors of `p`.

If during the computation a factorisation of `p` is found (because some pivot entry has nontrivial greatest common divisor with `p`) then the function is recursively applied to the found factors `f_i` of `p`. The result is then given in the form `[[f_1, rk_1], [f_2, rk_2], ...]`.

The idea to make this function useful for non primes was to use it with large factors of the biggest elementary divisor of `A` whose prime factorization cannot be found easily.

Example

```
gap> ReadPackage("edim",  "tst/mat");
Reading 242x242 integer matrix 'mat' with elementary divisors 'eldiv'.
true
gap> RankMod(mat, 5);
155
gap> RankMod(mat, (2*79*4001));
[ [ 2, 148 ], [ 79, 242 ], [ 4001, 242 ] ]
```

1.7 InverseRatMat - the Algorithm

The idea is to recover a rational matrix from an l-adic approximation for some prime l. This description came out of discussions with Jürgen Müller. I thank John Cannon for pointing out that the basic idea already appeared in the paper [Dix82] of Dixon.

Let A be an invertible matrix over the rational numbers. By multiplying with a constant we may assume that its entries are in fact integers.

(1) We first describe how to find an l-adic approximation of A^{-1}. Find a prime l such that A is invertible modulo l and let B be the integer matrix with entries in the range $]-l/2,l/2]$ such that BA is congruent to the identity matrix modulo l. (This can be computed fast by usual Gauß elimination.)

Now let $v \in \mathbb{Z}^r$ be a row vector. Define two sequences v_i and x_i of row vectors in $\mathbb{Z}^r$ by: $x_0 := 0 \in \mathbb{Z}^r$, $v_0 := -v$ and for $i > 0$ set x_i to the vector congruent to $-v_{i-1}B$ modulo l having entries in the range $]-l/2,l/2]$. Then all entries of $x_iA + v_{i-1}$ are divisible by l and we set $v_i := (1/l) \cdot (x_iA + v_{i-1})$.

Induction shows that for $y_i := \sum_{k=1}^{i} l^{k-1}x_k$ we have $y_iA = v + l^iv_i$ for all $i \geq 0$. Hence the sequence y_i, $i \geq 0$, gives an l-adic approximation to the vector $y \in \mathbb{Q}^r$ with $yA = v$.

(2) The second point is to show how we can get the vector y from a sufficiently good approximation y_i. Note that the sequence of y_i becomes constant for $i \geq i_0$ if all entries of y are integers of absolute

value smaller than $l^{i_0}/2$ because of our choice of representatives of residue classes modulo l in the interval $]-l/2,l/2]$.

More generally consider $a/b \in \mathbb{Q}$ with $b>0$ and a,b coprime. Then there is for each $n \in \mathbb{N}$ which is coprime to b a unique $c \in \mathbb{Z}$ with $-n/2 < c \leq n/2$ and $a \equiv cb \pmod{n}$. This c can be computed via the extended Euclidean algorithm applied to b and n.

Now let $n,\alpha,\beta \in \mathbb{N}$ with $2\alpha\beta \leq n$. Then the map $\{a/b \in \mathbb{Q} \mid -\alpha \leq a \leq \alpha, 1 \leq b < \beta\} \to]-n/2,n/2]$, $a/b \mapsto c$ (defined as above) is injective (since for a/b, a'/b' in the above set we have $ab'-a'b \equiv 0 \pmod{n}$ if and only if $ab'-a'b=0$).

In practice we can use for any $c \in]-n/2,n/2]$ a certain extended Euclidean algorithm applied to n and c to decide if c is in the image of the above map and to find the corresponding a/b if it exists.

(3) To put things together we apply (2) to the entries of the vectors y_i constructed in (1), choosing $n=l^i$, $\alpha=\sqrt{n}/2$ and $\beta=\sqrt{n}$. If we have found this way a candidate for y we can easily check if it is correct by computing yA. If μ is the maximal absolute value of all numerators and denominators of the entries of y it is clear from (2) that we will find y from y_i if $l^i > 2\mu^2$.

(4) If we take as v in (1) to(3) all standard unit vectors we clearly get the rows of A^{-1}. But we can do it better. Namely we can take as v the standard unit vectors multiplied by the least common multiple ε of the denominators of the already computed entries of A^{-1}. In many examples this ε actually equals ε_r after the computation of the first or first few rows. Therefore we will often find quickly the next row of A^{-1} already in (1), because we find a $v_i=0$ such that the sequence of y_i becomes constant ($=y$).

1.7.1 Rank of Integer Matrix

The following strategy has shown to be useful in proving that some very big integer matrix is not invertible.

- Check the rank modulo some small primes, say with `RankMod` (1.6.5).
- If the rank seems less than the number of rows choose a prime p, a collection of lines which is linearly independent modulo p, and another line linearly dependend on these. Guess that this last line is also linearly dependend on the chosen collection over the rational numbers (maybe check modulo several small primes).
- Find columns of the collection of lines which give an invertible matrix modulo some prime.
- Then use `RationalSolutionIntMat` (1.3.2) with the invertible submatrix and corresponding entries of the linearly dependend row to prove this.

Guessing the rank of a matrix from the rank modulo several primes, chosing a maximal set of lines which are linearly independent modulo some primes, and using `RationalSolutionIntMat` (1.3.2) with the remaining lines, one may also find the exact rank of a huge integer matrix.

References

[Dix82] J. D. Dixon. Exact solution of linear equations using p-adic expansions. *Numer. Math.*, 40:137–141, 1982. 11

[HMM98] G. Havas, B. S. Majewski, and K. R. Matthews. Extended gcd and Hermite normal form algorithms via lattice basis reduction. *Experimental Mathematics*, 7:125–135, 1998. 8, 10

[HS79] G. Havas and L. S. Sterling. Integer matrices and abelian groups. In *Symbolic and algebraic computation*, volume 72 of *Lecture Notes in Computer Science*, pages 431–451. Springer-Verlag, Berlin, 1979. 4, 6

[LÃ¼02] F. LÃ¼beck. On the computation of elementary divisors of integer matrices. *Journal of Symbolic Computation*, 33:57–65, 2002. 4, 5

Index