The Berkowitz Algorithm, Maple and Computing the Characteristic Polynomial in an Arbitrary Commutative Ring

*Jounaidi Abdeljaoued**

Abstract: An improved sequential version of the Berkowitz algorithm is given that computes the coefficients of the characteristic polynomial of an $n \times n$ matrix with entries from an arbitrary commutative ring. Some other methods for solving the same problem are discussed and implemented in the Maple system. Several tests show that our algorithm runs faster in the most general case.

Keywords: matrix computations, determinant, characteristic polynomial, Berkowitz' algorithm, Gaussian elimination, computational complexity, parallel computation.

Introduction

The characteristic polynomial plays an important role in Linear Algebra and has many applications in various areas: the solution of numerous problems in Mathematics ([16], [5], [9]), Numerical Analysis ([12], [18]), Computer Algebra ([17], [7]), Computer Science ([3], [20]), Engineering and, Physics ([19]). Such problems often require the computation of the characteristic polynomial.

The determinant of a matrix is nothing but the constant term in its characteristic polynomial. Besides the determinant of the matrix, the knowledge of all the coefficients of the latter gives us the adjoint of the matrix and its inverse if the determinant is invertible in the coefficient ring.

Furthermore, the coefficients of the characteristic polynomial of a square matrix yield precise information about the mathematical objects represented by this matrix. They might be finite families of vectors, linear maps of finite dimensional vector spaces, as well as quadratic forms or module structures [9].

For instance, suppose you want to discuss the rank of a matrix whose coefficients are functions of a parameter with unspecified values, or to determine the signature of any quadratic form on the polynomial ring $\mathbf{Z}[\lambda]$ according to the values of λ, or to compute the sum of the principal minors of a given dimension in a square matrix. All these problems are naturally solved by computing the coefficients of the characteristic polynomial.

The next section contains a brief discussion of some classical methods for computing characteristic polynomials. Then we roughly indicate how Maple functions proceed and when they fail to solve the problem. Our main purpose, though, is to present the Berkowitz Method we have implemented in the Maple System and to illustrate it at the end of this paper by giving some experimental tests on various examples which show the good practical behaviour of this algorithm for univariate or multivariate matrices, but also for integer matrices of quite large size. As far as we know, it is the first time this algorithm had been implemented and tested on a sequential machine.

Moreover, we will try to make our exposition as self-contained as possible by giving some relevant preliminary results from Linear Algebra. From now on we make the assumption that Δ is a commutative ring with identity.

*Assistant at the "Ecole Supérieure des Sciences et Techniques of Tunis" (Tunisia), Equipe de Mathématiques de Besançon, Université de Franche–Comté, 25030 Besançon Cedex. E.Mail: Sonia.Zghal@cck.rnrt.tn

About some methods for computing the characteristic polynomial

Given an $n \times n$ matrix M with entries in Δ, the characteristic polynomial of M is the determinant of its characteristic matrix $M - XI_n \in (\Delta[X])^{n \times n}$ where I_n is the n-square identity matrix and X the monic monomial of degree 1 on Δ. Therefore, the computation of the characteristic polynomial of a matrix with coefficients in Δ reduces to the computation of a determinant with entries in $\Delta[X]$. So, we are interested in computing the polynomial $P(X) = \det(M - XI_n) = \sum_{i=0}^{n} p_i X^{n-i}$. Note that $p_0 = (-1)^n$, $p_n = \det(M)$ and by the well known Cayley-Hamilton theorem, that $\sum_{i=0}^{n-1} p_i M^{n-i} = -\det(M) I_n$.

Besides the conventional minor expansion method which is very efficient for sparse matrices[1], the most widely used method for computing the determinant is the Gaussian elimination (GE for short).

To compute the determinant of M we perform GE over the ring $\mathcal{R} = \Delta$ with the input matrix $A = M$. If we want to compute the characteristic polynomial of M, we work over the ring $\mathcal{R} = \Delta[X]$ with the input matrix $A = M - XI_n$. Since GE performs divisions in the fraction field of $\mathcal{R}$, we require, in general, that $\mathcal{R}$ be an integral domain.

[1] matrices with a relatively small number of non zero entries

Ignoring details of pivoting, we can express the main algorithm of the Gaussian elimination by the following Maple procedure where A is a given n-square matrix:

Procedure GE (Gaussian Elimination).

```
for k to n-1 do
    for i from k+1 to n do
     for j from k+1 to n do
      A[i,j] := A[i,j] - A[i,k]/A[k,k]*
                A[k,j]
     od;
     A[i,k] := 0
    od;
od:
```

The procedure, GE, transforms the input matrix A to upper triangular form using only elementary row operations. Upon termination, the determinant of A can be recovered as the product of the diagonal entries in the triangular output matrix.

The elimination method works well for univariate or numerical coefficients, but it is too costly when dealing with multivariate entries. The numerators and the denominators of the intermediate expressions quickly become too long and the fractions which arise in the calculations become harder and harder to simplify. To overcome this major disadvantage of the Gaussian elimination, an algorithm due to Jordan-Bareiss [4] uses exact divisions in the ground ring during the elimination process. The Jordan-Bareiss method, also called the fraction-free Gaussian elimination (FFGE for short), comes from a Sylvester's relation between the minors of the matrix considered. This was apparently already known to Jordan [12].

Let $A^{(k)}$ be the matrix obtained at the k-th step of the Gaussian elimination ($A^{(0)} = A$) and $a_{ij}^{(k)}$ its coefficients. Then the following relation is one of the Sylvester identities and it holds for $1 \leq k \leq n-1$ and $1 \leq i, j \leq n$:

$$a_{ij}^{(k)}\, a_{k-1,k-1}^{(k-2)} = \begin{vmatrix} a_{kk}^{(k-1)} & a_{kj}^{(k-1)} \\ a_{ik}^{(k-1)} & a_{ij}^{(k-1)} \end{vmatrix}$$

with the convention that $a_{ij}^{(-1)} = 1$ and $a_{ij}^{(0)} = a_{ij}$.

Therefore, the divisions are exact in the following Maple procedure using the Jordan-Bareiss Algorithm where details of pivoting are ignored:

Procedure FFGE (Fraction Free Gaussian Elimination).

```
dnm:=1;
for k to n-1 do
  for i from k+1 to n do
   for j from k+1 to n do
    A[i,j]:=(A[k,k]*A[i,j]-A[i,k]*
                A[k,j])/dnm
   od;
   A[i,k]:= 0
  od;
  dnm := A[k,k]
od:
```

A fundamental property of the FFGE algorithm is that after each iteration of the outer loop, the value `dnm:=A[k,k]` will be set equal to the determinant of the k-th principal submatrix of the input matrix. Upon termination, the value of `dnm` will be equal to the determinant of the input matrix.

Although ordinary Gaussian elimination requires fewer arithmetic operations than the Jordan-Bareiss algorithm, the latter usually runs faster, particularly for multivariate matrices. The Jordan-Bareiss algorithm considerably limits the growth of the intermediate expressions during the successive steps of computation.

Both GE and FFGE applied to a matrix over $\mathcal{R} = \Delta$ or $\mathcal{R} = \Delta[X]$ require, in the general case, that Δ be an integral domain. However, in the special case where $\mathcal{R} = \Delta[X]$, we compute the determinant of $M - XI_n$ where M is over Δ. The algorithm FFGE works even when Δ is not an integral domain. In particular, the divisor `dnm := A[k,k]` at each stage of the algorithm FFGE will be the characteristic polynomial of the k-th principal submatrix of M. A polynomial with a leading coefficient equal to ± 1, and division in the ring $\Delta[X]$ by a monic polynomial requires only basic ring operations from Δ.

This leads to the so-called Jordan-Bareiss modified method for computing the determinant of a matrix M over an arbitrary commutative ring via the computation of its characteristic polynomial.

While GE and FFGE require $O(n^3)$ arithmetic operations (with divisions) in the fraction field of Δ, the Jordan-Bareiss modified algorithm computes the determinant in $O(n^5)$ basic ring operations from Δ.

A fourth well known method for computing the characteristic polynomial is based on the Lagrange interpolation formula,

$$P(X) = \sum_{k=0}^{n} \left[P(x_k) \prod_{\substack{0 \leq i \leq n \\ i \neq k}} \frac{(X - x_i)}{(x_k - x_i)} \right] \quad \text{where}$$

$x_0, x_1, \ldots, x_n$ are $n+1$ distinct elements of Δ.

For instance one may choose $x_k = k$ for $0 \leq k \leq n$ when Δ is of characteristic 0 or greater than n. In this case, performing exact divisions by $2, \ldots, n$ is required in Δ

because

$$\det(A - XI_n) = \sum_{k=0}^{n} \left[\frac{\det(M - kI_n)}{k!\,(n-k)!} \prod_{\substack{0 \le i \le n \\ i \ne k}} (X - x_i) \right].$$

Thus, we have to compute $n+1$ determinants with coefficients in Δ and divide them by integers not greater than n. This requires $O(n^4)$ arithmetic operations in the basic ring Δ but uses a relatively small amount of memory location.

The interpolation method can be expressed in the form of the following Maple procedure:

```
interpoly:=proc(M:matrix,X:name)
  local n,Id,i,j,N,d,L,Kar:
  n:=linalg[coldim](M):
  Id:=array(identity, 1..n, 1..n):
  for i to n+1 do
    d[i]:=linalg[det](M-evalm((i-1)*Id))
  od;
  L:=[seq(d[j], j = 1..n+1)]:
  Kar:=interp([`$`(0..n)], L, X):
  Kar
end:
```

Another method due to Faddeev [15] and J. M. Souriau [24] uses the fact based on a Newton formula and discovered by the French astronomer Le Verrier [21], that the coefficients p_k of the characteristic polynomial $P(X) = \sum_{i=0}^{n} p_i X^{n-i}$ of an n-square matrix M are related to the traces $\mathrm{Tr}(M^k)$ of the matrix powers of M, for k ranging from 1 to n, by the recursive formula:

$$-kp_k = \sum_{i=0}^{k-1} p_i \mathrm{Tr}(M^{k-i}).$$

This method requires division by $n!$ for computing $P(X)$ (so Δ must be of characteristic 0 or greater than n).

We have implemented the Hessenberg Method, among others, for computing the characteristic polynomial. It uses elementary transformations that are simultaneously executed on rows and columns of the given matrix M. This yields an almost upper triangular matrix H which is similar to M. The matrix $H = (h_{ij})$, also called an upper Hessenberg form of M, is such that $h_{ij} = 0$ for $i - j \ge 2$ and $\det(M - XI_n) = \det(H - XI_n)$. The characteristic polynomial of M is then given by a recursive formula which relates the characteristic polynomial of H to that of a smaller Hessenberg matrix (see [14] and [9] for details).

Although theoretically efficient (it is $O(n^3)$), this method is very heavy in practice because of the fast growth of the size coefficients. In addition, it requires working in a field.

Hence, if Δ is not a field, these methods are impracticable and we must call for a method which avoids divisions in the computation. Of course, the minor expansion method applied to the characteristic matrix $M - XI_n \in (\Delta[X])^{n\times n}$ of M is valid for an arbitrary commutative ring, but it is adapted to sparse matrices and is too costly for dense matrices of large size. On the other hand, as already seen, the Jordan-Bareiss modified method obtained by applying the FFGE algorithm to the characteristic matrix is less costly and is valid in the general case.

Another method, we will not develop here, is due to Chistov (see [8], [20] and [25] for the complete theory and detailed proofs). It is based on the observation that $P(X) = \det(M - XI_n)$ is the reverse of the polynomial $Q(X) = \det(XM - I_n)$ modulo X^{n+1} (indeed, $P(X) = X^n Q(\frac{1}{X})$). One computes $Q(X) = ((Q(X))^{-1})^{-1}$ modulo X^{n+1} in the formal power series ring $\Delta[[X]]$, using the power series algebraic identities:

$$(Q(X))^{-1} = \prod_{r=1}^{n} ((XM_r - I_r)^{-1})_{rr}$$

$$\text{and} \quad (I_r - XM_r)^{-1} = \sum_{k=0}^{\infty} X^k M_r^k$$

where M_r and I_r denote the principal leading submatrices of M and I_n respectively, and $((XM_r - I_r)^{-1})_{rr}$ the r-th diagonal element of the inverse matrix of $(XM_r - I_r)$.

The Chistov method is valid in the general case and notably more efficient than the methods cited above (namely Gauss, Jordan-Bareiss and minor expansion) for computing the characteristic polynomial of a dense and full rank matrix. It is one of the algorithms we have implemented in Maple for testing and comparison[2].

What does Maple do ?

Maple essentially uses Gaussian elimination or minor expansion or the combination of the two for computing determinants and characteristic polynomials. These two methods have to deal with univariate rational fractions and large coefficients beside the hard problem of polynomial simplification. The Maple system switches to the Jordan-Bareiss fraction-free Gaussian elimination when the entries in the considered matrices are integers or multivariate polynomials on the rationals.

Consider the case of an integer input matrix. Maple's **linalg[charpoly]** function – which uses a homomorphic imaging scheme to control the size of intermediate coefficients – still requires a factor of at least $O(n)$ more space and time

[2] with a decisive simplification suggested by Arne Storjohann, one of the referees

(bit operations) compared to the improved Berkowitz algorithm we present in the next section, which performs all operations directly over the integers.

In many other cases, the appropriate Maple function from the "linalg package" fails to solve the problem of the characteristic polynomial. We now give an example to show this.

When you have to compute the characteristic polynomial of a matrix with entries in the polynomial ring **Z** [x] , even for relatively small dimensions, you can see that the Maple function **linalg[charpoly]** fails. For example, let M = Mathard (n, x, y) be a matrix given by the following procedure:

```
Mathard:=proc(n,x,y)
    local K,i,j:
    K:=array(1..n,1..n):
     for i to n do
      for j to n do
           K[i,j]:=randpoly([x,y],
           coeffs = rand(-5..5)):
      od:
     od:
    evalm(K):
end:
```

Take n:=15 and y:=1 which yields a matrix whose coefficients are random univariate polynomials of **Z** [x] and try to compute its characteristic polynomial by means of the **linalg[charpoly]** function. An error message occurs after a long time of calculation (approximately 2 hours) saying: "`System error, ran out of memory`" and the memory allocated reaches 120 Mbytes.

At the same time, the improved Berkowitz algorithm we are going to present now gives the result in less than one minute CPU and the memory doesn't go beyond 4.1 Mbytes (see Appendix II for other examples and tests).

The Berkowitz method

This method uses only basic operations (addition, substraction and multiplication) in the ground field (here an arbitrary commutative ring Δ with a multiplicative unit) to compute the characteristic polynomial of a given matrix $A = (a_{ij}) \in \Delta^{n\times n}$.

In fact, Berkowitz [6] provided a uniform family of arithmetic parallel circuits with size $O(n^{\alpha+1+\epsilon})$ and depth $O(\log^2 n)$, where ϵ is an arbitrary positive real number and α the exponent[3] for $n \times n$ fast parallel matrix multiplication with an $O(\log n)$-depth circuit.

In this way he gave a decisive improvement to the asymptotic complexity for the parallel computation of determinants, characteristic polynomials and adjoints of matrices with entries in an arbitrary commutative ring with arbitrary characteristic.

[3]The current estimation of α is due to Winograd & Coppersmith, 1987, with $\alpha < 2.376$.

A slight modification of the Berkowitz Algorithm allowed us to improve the complexity result by producing a family of arithmetic parallel circuits with size $O(n^{\alpha+1} \log n)$ and a precise evaluation of the overhead hidden constant in the "big O", as well as a simplified version of this algorithm.

It is not our purpose to develop the parallel version of the Berkowitz Algorithm here. For a detailed exposition, we refer the reader to [6], [13], [1], and [2]. On the other hand, we will look more closely at the sequential version of the improved Berkowitz Algorithm based on the "Samuelson Formula".

The Maple code of the main algorithm we have called "berkosam" is given in Appendix I and presented in the section "The main algorithm and its complexity" with a derived modular version called "berkomod" and another version called "berksparse" which is very efficient and particularly well adapted to the case of sparse matrices. Finally, some experimental results are presented in the section "Experimental results " and recorded on the comparison tables of Appendix II. For the convenience of the reader, some relevant facts from Linear Algebra are given in the following preliminaries.

PRELIMINARIES

Let $A_n = A$. For all integers r $(1 \leq r \leq n-1)$ we denote by:

- I_r the identity $r \times r$ matrix ;
- A_r the $r \times r$ submatrix of A built on the first r rows and the first r columns of A that is, $A_r = (a_{ij})$ with $1 \leq i \leq r$ and $1 \leq j \leq r$;
- $P_r(X) = \det(A_r - XI_r) = \sum_{i=0}^{r} p_{r-i}X^i$ the characteristic polynomial of A_r ;
- R_r the $1 \times r$ matrix of elements $a_{r+1,j}$ such that $1 \leq j \leq r$;
- S_r the $r \times 1$ matrix of elements $a_{i,r+1}$ such that $1 \leq i \leq r$.

For any n-square matrix $M = (m_{ij}) \in \Delta^{n\times n}$ we denote by M_{ij} the cofactor of m_{ij} in the matrix M, i.e. $M_{ij} = (-1)^{i+j} \det(M[i\downarrow j])$ where $M[i\downarrow j]$ is the submatrix $(n-1) \times (n-1)$ of M obtained by deleting the i-th row and the j-th column of M. Let $\mathrm{Adj}(M)$ be the adjoint of M, that is, the matrix transposed of the matrix (M_{ij}).

It is well known that $\det(M)$ can be obtained by means of a row (or a column) expansion, for instance $\det(M) = \sum_{i=1}^{n} m_{ij}M_{ij}$ (or $\det(M) = \sum_{j=1}^{n} m_{ij}M_{ij}$) and $\mathrm{Adj}(M)$ $M = M\mathrm{Adj}(M) = \det(M)\, I_n$.

In particular, one has $(XI_r - A_r)\ \mathrm{Adj}(A_r - XI_r) = -P_r(X)I_r$. Viewed as a polynomial equality whose coefficients are matrices in $\Delta^{r\times r}$, this identity shows that the polynomial $(XI_r - A_r)$ divides the polynomial $-P_r(X)I_r$ and that the quotient is $\mathrm{Adj}(A_r - XI_r)$, a polynomial of degree $r-1$ in X. The latter is obtained by performing the analog of an euclidean division in the polynomial ring $\Delta^{r\times r}[X]$ by the monic polynomial $(XI_r - A_r)$ with a re-

mainder $P_r(A_r) = 0$. This procedure is due to the famous algebraic identity known as the Cayley-Hamilton theorem. It is easy to check that:

Lemma 0.1

$$\mathrm{Adj}\,(A_r - XI_r) = -\sum_{k=2}^{r+1}(A_r^{k-2}p_0 + ... + I_r p_{k-2})X^{r+1-k}.$$

By expanding the determinant $\begin{vmatrix} A_r & S_r \\ R_r & a_{r+1,r+1} \end{vmatrix}$ of the $(r+1) \times (r+1)$ matrix A_{r+1} along its last row, then expanding the obtained cofactors along their last common column (which is nothing but S_r) we easily see that:

Lemma 0.2 *For all integers n $(n \geq 2)$ and r $(1 \leq r \leq n-1)$, given a matrix $A \in \Delta^{n\times n}$:*

$$\det(A_{r+1}) = a_{r+1,r+1}\ \det(A_r) - R_r\mathrm{Adj}\,(A_r)S_r.$$

Applying the two lemmas above to the characteristic matrix $A_{r+1} - XI_{r+1}$ and using the notations above, we obtain:

Proposition 0.1 : Samuelson's Formula

$$P_{r+1}(X) = (a_{r+1,r+1} - X)P_r(X) + \sum_{k=1}^{r}[(R_r A_r^{k-1}S_r)p_0 + ... + (R_r S_r)p_{k-2}]X^{r-k}.$$

ANOTHER FORM OF THE SAMUELSON FORMULA

We shall give another form of the Samuelson Formula derived by Berkowitz [6]. This yields a very practical algorithm for computing (without division) the characteristic polynomial followed by the determinant, the adjoint, and the inverse of a matrix whose coefficients belong to an arbitrary commutative ring.

We use special $(n+1) \times n$ matrices which are Toeplitz subdiagonal, namely matrices (a_{ij}) where for $1 \leq i \leq n+1$ and $1 \leq j \leq n$, $a_{ij} = a_{(i-j)}$ for some sequence $(a_k)_{0\leq k\leq n}$ of elements in Δ with the convention that $a_i = 0$ if $i < 0$.

For each polynomial $P(X) = \sum_{k=0}^{d} a_k\, X^{n-k}$ with coefficients in Δ, we consider:

- the vector $\overrightarrow{P} = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_d \end{pmatrix}$ of its coefficients ;
- the special $(d+1) \times d$ lower triangular Toeplitz matrix, as defined above, associated to the sequence $(a_k)_{0\leq k\leq d}$ of coefficients of P and denoted by $\mathrm{Toep}\,(P)$.

Thus $\mathrm{Toep}\,(P) = \begin{pmatrix} a_0 & 0 & 0 & \cdots & 0 \\ a_1 & a_0 & 0 & \cdots & 0 \\ a_2 & a_1 & a_0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ . & . & . & \cdots & a_0 \\ a_d & a_{d-1} & a_{d-2} & \cdots & a_1 \end{pmatrix}$

and the Samuelson Formula can be written in the form of an $(r+2)$-vector equality $\overrightarrow{P_{r+1}} = \mathrm{Toep}\,(Q_{r+1}) \times \overrightarrow{P_r}$ where Q_{r+1} is the polynomial in X: $Q_{r+1} = -X^{r+1} + a_{r+1,r+1}X^r + (R_r S_r)X^{r-1} + ... + (R_r A_r^i S_r)X^{r-1-i} + ... + (R_r A_r^{r-1}S_r)$. Application of the relation above to the vectors $\overrightarrow{P_r}$, ... , $\overrightarrow{P_2}$ for $r \geq 2$ and the fact that $\overrightarrow{P_1} = \begin{pmatrix} -1 \\ a_{11} \end{pmatrix} = \mathrm{Toep}\,(Q_1)$ show that:

Proposition 0.2 *The coefficients of the characteristic polynomial $P_r(X)$ of any principal submatrix A_r $(1 \leq r \leq n)$ of A are given by:*

$$\overrightarrow{P_r} = \mathrm{Toep}\,(Q_r) \times \mathrm{Toep}\,(Q_{r-1}) \times \cdots \times \mathrm{Toep}\,(Q_1).$$

In particular, the characteristic polynomial of the given matrix $A \in \Delta^{n\times n}$ is given by:

$$\overrightarrow{P_n} = \mathrm{Toep}\,(Q_n) \times \mathrm{Toep}\,(Q_{n-1}) \times \cdots \times \mathrm{Toep}\,(Q_1)$$

which leads to the Berkowitz Algorithm. Below, we will only deal with the improved sequential version of this algorithm.

The main algorithm and its complexity

Input: An n-square matrix $A \in \Delta^{n\times n}$.
Output: The characteristic polynomial P_n of A.

(1) **Initialize** the vector Vect to Vect := $\begin{pmatrix} -1 \\ a_{11} \end{pmatrix}$

(2) **for** r **from** 1 **to** $n-1$

- **Compute** the entries $\{R_r A_r^{k-1} S_r\}_{k=1}^{r}$ of the Toeplitz matrix $\mathrm{Toep}\,(Q_{r+1})$;
- **Update** Vect into Vect := $\mathrm{Toep}\,(Q_{r+1}) \times$ Vect ;

(3) **Return** P_n the unique polynomial satisfying $\overrightarrow{P_n}$ = Vect.

THE COMPLEXITY

It is important to emphasize that the computations of step (2) can be performed much more efficiently when executed serially by using only dotproducts and matrix-vector multiplications instead of matrix-matrix multiplications.

More precisely, instead of computing the matrix powers A_r^{k-1} for $1 \leq k \leq r$ we first compute $R_r S_r$ then for k ranging from 2 to r the matrix-vector product $A_r^{k-1}S_r$ and the dotproduct $R_r A_r^{k-1}S_r$. In this case, the algorithm is easily

shown to involve less than $\frac{1}{2}n^4 - n^3 + \frac{5}{2}n^2$ arithmetic operations (additions/substractions and multiplications) in the ground ring Δ.

On the other hand, the parallel version of the algorithm ([6], [1], [2], [13]) computes in parallel the $n-1$ Toeplitz matrices of step (2) and requires $O(\log n)$ n-square matrix squarings in each parallel step. Roughly speaking, this is why the parallel circuit size supports the asymptotic bound $O(n^{\alpha+1}\log n)$ when using the fast matrix multiplication ($\alpha < 3$) and leads to sequential time $O(n^4 \log n)$ instead of the $O(n^4)$ above, when using the standard matrix multiplication ($\alpha = 3$).

It is worthwhile to notice that the complexity bound is reduced to $O(n^3)$ basic operations for the improved Berkowitz algorithm, in the case of $n \times n$ sparse matrices with $O(n)$ non zero entries.

THE MAPLE CODES

In Appendix I, we give the Maple code of the improved Berkowitz algorithm (J. Abdeljaoued, 1994) with the calling sequence "berkosam (A, X)" for computing the characteristic polynomial (in X) of the given n-square matrix A. This code has received great benefit from the suggestions of the referees who pointed out that the explicit construction of the Toeplitz matrices is not required and that a slight modification provides a very efficient algorithm for sparse matrices.

We call it the simple code in opposition to the "modular" code **berkomod** which performs computations modulo a given ideal in a polynomial ring. In fact, in order to obtain the modular code, we have just adapted the simple code in a way which enables us to work in a quotient ring like $\mathbf{Z}_p[x,y]/(P(x),Q(x,y))$ where x, y are two independant variables; $P \in \Delta[x]$ and $Q \in \Delta[x,y]$ are respectively monic in x and y.

So, the procedure **berkomod** with the calling sequence "berkomod $(A, X, var, Ideal, p)$" computes by Berkowitz' method the characteristic polynomial (in X) of A, a given square matrix with coefficients in $\mathbf{Z}_p[var]/(Ideal)$, where p is a positive integer, var a list of indeterminates and $Ideal$ a list of polynomials in var. In our examples, we take the same number of polynomials and indeterminates and require that $Ideal[i]$ be monic in $var[i]$. For this, auxiliary procedures are used to construct such examples and perform the modular computations. The procedure **Matmod** creates random matrices with polynomial entries in the quotient ring $\mathbf{Z}_p[var]/(Ideal)$:

```
Matmod:= proc(n:posint,var:list,
Ideal:list,p:posint)
local K,i,j:
K:=array(1..n,1..n):
for i to n do
   for j to n do
     K[i,j]:=randpoly(var,
       coeffs=rand(-5..5)):
       K[i,j]:=simpomod(",var,Ideal,p)
   od:
od:
evalm(K):
end:
```

The procedure **simpomod** below takes a list of variables var, a polynomial P from $\mathbf{Z}[var]$, a list of polynomials $Ideal$ in $\mathbf{Z}[var]$, a positive integer p and outputs a representative of P in the ring $\mathbf{Z}_p[var]$ modulo $Ideal$:

```
simpomod:= proc(P,var,Ideal,p)
local locP,loci:
if not type(P,rational) then
  locP:=P:
  for loci to nops(var) do
    locP:=rem(locP,Ideal[loci],var[loci]):
    locP:=locP mod p:
  od:
  sort(locP) else P mod p
fi:
end:
```

The **simpomod** procedure enables us to obtain the modular **berkomod** from the simple procedure **berkosam** (see Appendix I).

Experimental results

Both simple, modular, and sparse versions of the improved algorithm have been experimented with in Maple V (Release 3) on DEC Alpha-600, Sun Sparc-10, IBM RS/6000 and Mac-Intosh platforms: the results were really remarkable when compared with the **linalg[charpoly]** function and other algorithms we have implemented in Maple, namely Chistov[4], Faddeev, Jordan-Bareiss modified, Hessenberg and the Interpolation Method ("interpoly").

The Berkowitz Algorithm referred to as **berkosam**, **berkomod** or **berksparse** according to the simple, modular or sparse version, runs faster and, in comparison to others as shown in our tables, does not blow up for large values of n with random integer entries or for small values of n and random polynomial entries with two variables.

However, in some particular cases, there is no significant difference between Berkowitz and some other algorithms namely, Faddeev and Jordan-Bareiss modified when dealing with small rank matrices or Interpolation Method with small size matrices.

On the other hand, even for simple integral domains, the Maple function **linalg[charpoly]** fails for relatively small $n \times n$ matrices with entries in $\mathbf{Z}[x,y]$.

For example, to compute the characteristic polynomial of the 10×10 matrix obtained by using the procedure Mathard

[4] with its two versions, for dense and sparse matrices

$(10, x, y)$ (see Section "What does Maple do?"), the running time for **berkosam** is less than 5' CPU while it is 48' 35" for **charpoly**. The latter is **"Out of Memory"** for $n := 12$ after 2H 30' of computation while **berkosam** takes 15' 30" CPU time and 27. 782 Mb of memory allocation for the same matrix.

In fact, even the Maple **linalg[det]** function fails in the computation of the sole determinant of such a matrix for $n := 15$ while **berkosam** terminates and gives the coefficients of the characteristic polynomial in less than 1H 24' and 61 Mbytes of memory allocation.

In Appendix II, at the end of this paper, we give some comparison tables which yield an interesting review of the capabilities (in terms of running time and memory allocation) of the seven algorithms written in the Maple programming language and executed on various examples. Note that for the fourth group of examples, the rank of the matrices $\mathrm{Jou}(n, x)$ for any positive integer n does not exceed 3. This is why the Faddeev and the Jordan-Bareiss modified methods run faster in this case. Notice that improved Berkowitz and Chistov algorithms works very well for sparse matrices with integer entries.

Conclusion

Some theoretical results about solving triangular systems of algebraic equations and the related algebraic complexity (see H. Lombardi's thesis [22]) are strongly relevant to algorithms absolutely cleared of divisions. The same remark holds for the quantifier elimination theory which calls upon such algorithms, or for Dynamic Evaluation [11] when a discussion concerning multivariate polynomials on a field $\mathcal{K}$ occurs and a question like: " is $P(x_1, x_2, \ldots, x_n) = 0 \mod \mathcal{I}_n$? " where $\mathcal{I}_n$ is the ideal generated by
$(P_1(x_1), P_2(x_1, x_2), \ldots, P_n(x_1, x_2, \ldots, x_n))$,
has to be answered before going on. One method is to compute the subresultant coefficients of the pair of polynomials (P, P_n) according to the variable x_n in the polynomial ring $K[x_1, x_2, \ldots, x_n]$ and to reduce the results obtained modulo the ideal $\mathcal{I}_{n-1}$ generated by
$(P_1(x_1), P_2(x_1, x_2), \ldots, P_{n-1}(x_1, x_2, \ldots, x_{n-1}))$.
For $n \geq 3$, this leads to very heavy calculations and a better suited solution is to carry out the computations in the quotient-ring $\mathcal{K}[x_1, x_2, \ldots, x_n] / \mathcal{I}_{n-1}$ which may contain zero divisors.

Now, the advantage of using Berkowitz' method lies in the very nice property that no division is needed and therefore, it is possible to work directly in such a ring with zero divisors.

The great disadvantage of Gaussian elimination as well as the fraction-free Gaussian elimination is that they require the possibility to perform divisions in the coefficient ring of the matrix. And when it happens that this division is not possible in such a ring, the linear algebra problems must often be treated by a suitable algorithm which does not require any division at all.

Compared with the Chistov method which has the same parallel and sequential theoretical complexity as the improved Berkowitz algorithm[5], the latter has the major advantage of running faster in practice beside the fact that instead of computing the characteristic polynomial of a single matrix A, it actually computes all those of principal leading submatrices of A. This is why, in the most general case, the improved Berkowitz method seems to be the best algorithm available today in practice for computing the characteristic polynomial over an arbitrary commutative ring.

Acknowledgments

The author is grateful to Henri Lombardi who made this work possible, Michael B. Monagan for his advice and his active interest in the publication of this paper, Erich Kaltofen for suggesting the comparison with the Chistov method, Joachim Von Zur Gathen for having pointed out the importance of the empirical experimentation results, Bruno Salvy and the referees of the Maple Technical Newsletter for their contribution in optimizing and alleviating the Maple codes of the algorithms and Joël Marchand for his precious help in testing these algorithms on the GDR Medicis machines at the Ecole Polytechnique.

The author is also greatly indebted to Arne Storjohann for the thorough job he has done in checking this paper and for giving improved codes for Berkowitz and Chistov algorithms beside pointing out the good behaviour of our algorithm in the case of sparse matrices.

References

[1] J. Abdeljaoued: *Sur l'algorithme de Berkowitz pour le calcul du déterminant dans un anneau commutatif arbitraire*, Prépublication de l'Equipe Mathématique de Besançon # 95/21, Université de Franche-Comté, (Décembre 1994).

[2] J. Abdeljaoued: *Algorithmes rapides pour le calcul du polynôme caractéristique*, Thèse de l'Université de Franche-Comté, (Mars 1997).

[3] A. V. Aho, J. E. Hopcroft and J. D. Ulmann: *The design and analysis of computer algorithms*, Addison Wesley, Reading, MA, (1974).

[4] E. H. Bareiss: Sylvester's identity and multistep integer-preserving Gaussian elimination, *Math. Comput.* **22**, pp. 565–578, (1968).

[5] They are $O(n^4)$. However, the Chistov sequential complexity algorithm is asymptotically $(2/3)\, n^4$ ([2]) while improved Berkowitz is $(1/2)\, n^4$.

[5] R. Bellman: *Introduction to matrix analysis*, McGraw-Hill, New York, Toronto, London, (1960).

[6] S. J. Berkowitz: On computing the determinant in small parallel time using a small number of processors, *Information Processing Letters*, **18**, pp. 147–150, (1984).

[7] D. Bini and V. Pan: *Polynomial and Matrix Computations*, **Vol. 1** Fundamental Algorithms, Birkhäuser, Boston Basel Berlin, (1994).

[8] A. L. Chistov: Fast parallel calculation of the rank of matrices over a field of arbitrary characteristic, Proc. FCT '85, Springer Lecture Notes in Computer Science **199**, pp. 147–150, (1985).

[9] H. Cohen: *A course in computational algebraic number theory*, Graduate Texts in Maths, **Vol. 138**, Springer Verlag, (1993).

[10] D. Coppersmith and S. Winograd: Matrix multiplication via arithmetic progressions, Proc. 19th Ann. ACM Symp. on Theory of Computing, pp. 1–6, (1987).

[11] Dora J. Della, C. Dicrescenzo and D. Duval: *About a new method for computing in algebraic number fields*, Eurocal '85, **Vol. 2**, Springer Lecture Notes in Computer Science **204**, pp. 289–290, (1985).

[12] E. Durand: *Solutions numériques des équations algébriques*, Tome II, Masson & Co Editeurs, Paris, (1961).

[13] W. Eberly: *Very fast parallel matrix and polynomial arithmetic.*, Technical Report # 178/85, PHD Thesis, University of Toronto CA, pp. 19–27, (1985).

[14] D. K. Faddeev and V. N. Faddeeva: *Computational methods of linear algebra*, W. H. Freeman & Co., San Francisco, (1963).

[15] D. K. Faddeev and I. S. Sominskii: *Collected Problems in Higher Algebra*, Problem No. 979, (1949).

[16] F. R. Gantmacher: *Thèorie des Matrices*, Tome 1, (Théorie Générale), Dunod, Paris, (1966).

[17] M. Giusti and J. Heintz: *La détermination des points isolés et de la dimension d'une variété algébrique peut se faire en temps polynomial*, to appear in Comptes Rendus of "Computational Algebraic Geometry & Commutative Algebra", Cortona, Italy, (1991).

[18] G. H. Golub and Ch. F. Van Loan: *Matrix Computations*, J. Hopkins Univ. Press, Baltimore and London, (1993).

[19] A. Jenning: *Matrix computation for Engineers and Scientists*, John Wiley & Sons, New York, (1977).

[20] R. M. Karp and V. Ramachandran: *Parallel Algorithms for Shared-Memory Machines*, in Handbook of Theoretical Computer Science, **Vol. A**, J. van Leeuwen Editor, Elsevier Sc. Publish., Amsterdam, pp. 869–941.

[21] U. J. J. Le Verrier: Sur les variations séculaires des éléments elliptiques des sept planètes principales: Mercure, Venus, La Terre, Mars, Jupiter, Saturne et Uranus, *J. Math. Pures Appli.*, **4**, pp. 220–254, (1840).

[22] H. Lombardi: *Sous-résultants, suite de Sturm, spécialisation*, in Thèse de l'Université de Nice, (1989).

[23] P. A. Samuelson: A method for determining explicitly the characteristic equation, *Ann. Math. Statist.*, **13**, pp. 424–429, (1942).

[24] J. M. Souriau: Une méthode pour la décomposition spectrale et l'inversion des matrices, *C. R. Acad. Sciences*, **227**, pp. 1010–1011, (1948).

[25] J. Von Zur Gathen: *Parallel Linear Algebra*, in Synthesis of parallel algorithms (J. H. Reif Editor), Morgan Kaufmann publishers, San Mateo, California, pp. 573–617, (1993).

Biography

Jounaidi Abdeljaoued, graduated in Mathematics from the "Ecole Normale Supérieure" of Tunis in 1966 and post-graduated from the "Faculté des Sciences" of Tunis (TUNISIA) in 1974 with a particular interest to Lie algebras and algebraic semi-groups. He is currently a Teaching Assistant of Mathematics at the "Ecole Supérieure des Sciences et Techniques" of Tunis where he is also concerned with the use of Computer algebra in Mathematical education. He has combined his job with preparing a Doctorat Thesis in Computer Algebra, Mathematics and Applications at the "Université de Franche-Comté" (Besançon, FRANCE). His doctoral thesis was completed in March 1997.

Appendix I

```
################################################

berkosam:= proc(A:matrix,X:name)
    local n,Vect,r,C,Ar,R,S,Q,i,j,t0,t1,wf,polsav;
    description `berkosam(A,X) computes the characteristic polynomial
    in X of the given matrix A by Berkowitz' method`:
    n:=linalg[coldim](A);
    if linalg[rowdim](A)<>n then ERROR(`not a square matrix`,A) fi;
    if n=1 then RETURN(A[1,1]-X) fi;
    if n=2 then RETURN(normal((A[1,1]-X)*(A[2,2]-X)-A[2,1]*A[1,2])) fi;
    t0:=time():
    Vect:=table([1=-1,2=A[1,1]]); C[1]:=-1;
    for r from 2 to n do
       for i to r-1 do S[i]:=A[i,r] od:
       C[2]:=A[r,r];
       for i from 1 to r-2 do
          C[i+2]:=normal(convert([seq(A[r,j]*S[j],j=1..r-1)],`+`)):
          for j to r-1 do
             Q[j]:=normal(convert([seq(A[j,k]*S[k],k=1..r-1)],`+`))
          od;
          for j to r-1 do S[j]:=Q[j] od;
       od;
       C[r+1]:=normal(convert([seq(A[r,j]*S[j],j=1..r-1)],`+`)):
       for i to r+1 do
          Q[i]:=normal(convert([seq(C[i+1-j]*Vect[j],j=1..min(r,i))],`+`))
       od;
       for i to r+1 do Vect[i]:=Q[i] od;
    od;
    polcar:=normal(convert([seq(Vect[i+1]*X^(n-i),i=0..n)],`+`)):
    t1:=time()-t0:wf:=(status[2]-w0)/125:wf:=trunc("):
    lprint(`The running time is :`,t1,`m.secs`):
    lprint(`The memory allocation is :`,wf,`Kbytes`):
    polcar:
end:

################################################
```

The procedure **berkosam** as well as the Chistov algorithm can be adapted to the particular case of sparse matrices. This is done for **berkosam** by a simple modification:
at each stage through the loop `for r from 2 to n` the sums are taken over integers where the matrix entries are non zero that is, the set $\{ j / 1 \leq j \leq r\text{-}1 \text{ and } A[r, j] \neq 0\}$.

The modular version **berkomod** is obtained from the simple procedure **berkosam** by calling the procedure **simpomod** (see section "The Maple codes") after each coefficient computation. For example, we replace the instruction
" `for i to r+1 do Vect[i]:=Q[i] od` " in **berkosam** by the following one:
" `for i to r+1 do Vect[i]:=simpomod(Q[i],var,Ideal,p) od` ".
Another example: just after the line
" `C[i+2]:=normal(convert([seq(A[r,j]*S[j],j=1..r-1)],`+`)):` "
we add the instruction " `C[i+2]:=simpomod(",var,Ideal,p):` " and so on.

We conclude the procedure **berkomod** with:
" `normal(convert([seq(Vect[i+1]*X^(n-i),i=0..n)],`+`)): simpomod(",var,Ideal,p):` ".

Appendix II : COMPARISON TABLES
(Memory allocation in Megabytes)

First Group : randmatrix(n, n)
(Dense integer matrices)

matrix		berkosam	linalpoly †	Faddeev	barmodif ‡	Chistov	Hessenberg
$n = 16$	CPU Time	1.82 "	5.75 "	12"	9.65 "	2.95 "	5"
	Mem. Alloc.	3. 014 Mb	4. 455 Mb	4. 752 Mb	4. 586 Mb	3. 538 Mb	5. 806 Mb
$n = 20$	CPU Time	4.43 "	17"	28"	24.15 "	6.92 "	20"
	Mem. Alloc.	3. 538 Mb	4. 717 Mb	6. 192 Mb	5. 111 Mb	3. 669 Mb	12. 500 Mb
$n = 25$	CPU Time	16.60 "	41.62 "	1' 07"	1' 05"	16.60 "	1' 28"
	Mem. Alloc.	3. 800 Mb	4. 848 Mb	6. 978 Mb	5. 635 Mb	1.900 Mb	28. 668 Mb
$n = 32$	CPU Time	29.75 "	2' 12"	3' 16"	3' 33"	43.80 "	$\approx$ 2H
	Mem. Alloc.	3.931 Mb	4. 980 Mb	8. 028 Mb	7. 732 Mb	3. 931 Mb	OUT*
$n = 40$	CPU Time	1' 13"	6' 21"	17' 58"	12' 06"	1' 48"	–
	Mem. Alloc.	8. 124 Mb	5. 242 Mb	12. 480 Mb	12. 581 Mb	4. 062 Mb	–
$n = 50$	CPU Time	3' 04 "	21' 05"	2H 38' 50"	46' 30"	4' 37"	–
	Mem. Alloc.	8. 386 Mb	5. 897 Mb	21. 000 Mb	18. 871 Mb	4. 193 Mb	–
$n = 64$	CPU Time	8' 47"	1H 16' 26"	13H 15' 45"	3H 56' 24"	13' 23"	–
	Mem. Alloc.	8. 648 Mb	7. 076 Mb	39. 600 Mb	31. 451 Mb	4. 324 Mb	–
$n = 128$	CPU Time	8H 17' 38"	71H 33' 31"	$\approx$ 170H	$\approx$ 7H	11H 22' 48"	–
	Mem. Alloc.	13. 104 Mb	$>$ 120 Mbytes	OUT*	OUT*	13. 629 Mb	–

† The Maple linalg[charpoly] function.

‡ Procedure corresponding to the Jordan-Bareiss modified method.

* OUT means that, after a more or less long time of calculation, an error message "Out of memory" is returned by the System and that the memory allocation had reached a threshold of about 350 Mbytes.

Second Group : Mathard(n, x, y)
(Matrices with polynomial entries)

matrix		berkosam	linalpoly	Faddeev	barmodif	Chistov
$n = 10$	CPU Time	4' 04"	48' 35"	53' 31"	$\approx$ 17'	13' 17"
	Mem. Alloc.	16. 119 Mb	90. 500 Mb	97. 000 Mb	OUT	20. 705 Mb
$n = 12$	CPU Time	15' 30"	$\approx$ 1H 25'	$\approx$ 1H 10'	–	54' 18"
	Mem. Alloc.	27. 782 Mb	OUT	OUT	–	27. 127 Mb
$n = 15$	CPU Time	1H 23' 24"	–	–	–	5H 28' 28"
	Mem. Alloc.	60. 806 Mb	–	–	–	53. 466 Mb
$n = 10$ & $y = 1$	CPU Time	4.53 "	47"	25.50 "	1' 12"	10.10 "
	Mem. Alloc.	4. 062 Mb	9. 924 Mb	7. 207 Mb	6. 290 Mb	4. 454 Mb
$n = 15$ & $y = 1$	CPU Time	42.33 "	$\approx$ 2H 10'	4' 02"	22' 06"	1' 33"
	Mem. Alloc.	6. 682 Mb	OUT	16. 381 Mb	9. 400 Mb	7. 207 Mb
$n = 20$ & $y = 1$	CPU Time	3' 26"	–	21' 57"	3H 24' 47"	7' 14"
	Mem. Alloc.	5. 110 Mb	–	38. 790 Mb	13. 600 Mb	9. 960 Mb
$n = 25$ & $y = 1$	CPU Time	11' 56"	–	$\approx$ 1H 10'	23H 54' 55"	24' 52"
	Mem. Alloc.	17. 166 Mb	–	OUT	20. 000 Mb	20. 050 Mb

Third Group : Matmod$(n, var, Ideal, p)$

(Matrices over polynomial quotient-rings)

1. For $var = [x]$, $Ideal = [x^3 - 1]$ *and* $p = 7$:

matrice		berkomod	linalpoly	Faddeev	barmodif	Chistov
$n = 10$	CPU Time	2.45 "	14"	*	43"	4.10 "
	Mem. Alloc.	3. 407 Mb	6. 280 Mb		5. 600 Mb	3. 538 Mb
$n = 12$	CPU Time	4.43 "	2' 03"	*	1' 52"	7.18 "
	Mem. Alloc.	3. 538 Mb	17. 900 Mb		5. 800 Mb	3. 538 Mb
$n = 16$	CPU Time	11.75 "	$\approx$ 1H 30'	*	8' 31"	18.68 "
	Mem. Alloc.	3. 668 Mb	OUT		6. 670 Mb	3. 800 Mb

2. For $var = [x, y]$; $Ideal = [H, L]^{\dagger}$; *and* $p = 11$:

matrix		berkomod	linalpoly	Faddeev	barmodif	Chistov
$n = 10$	CPU Time	26.62 "	34' 04"	1' 20"	1H 03' 09"	48.15 "
	Mem. Alloc.	4. 454 Mb	74. 492 Mb	6. 192 Mb	19. 860 Mb	6. 158 Mb
$n = 12$	CPU Time	48.18 "	$\approx$ 1H 50'	*	4H 56' 17"	1' 29"
	Mem. Alloc.	4. 454 Mb	OUT		26. 478 Mb	7. 600 Mb
$n = 16$	CPU Time	2' 04"	–	*	$\approx$ 58H	4'
	Mem. Alloc.	4. 848 Mb	–		OUT	13. 104 Mb

3. For $var = [x, y]$; $Ideal = [H, L]^{\dagger}$; *and* $p = 17$:

matrix		berkomod	linalpoly	Faddeev	barmodif	Chistov
$n = 10$	CPU Time	27.55 "	38' 13"	1' 28"	1H 10' 01"	50.27 "
	Mem. Alloc.	4. 324 Mb	76. 472 Mb	4. 586 Mb	21. 098 Mb	6. 028 Mb
$n = 12$	CPU Time	49.78 "	$\approx$ 2H	2' 43"	5H 27' 49"	1' 33"
	Mem. Alloc.	4. 586 Mb	OUT	4. 717 Mb	27. 520 Mb	7. 992 Mb
$n = 16$	CPU Time	2' 11"	–	7' 20"	$\approx$ 84H 40'	4' 25"
	Mem. Alloc.	4. 848 Mb	–	5. 372 Mb	OUT	14. 284 Mb

† We have taken $H = x^5 - 5xy + 1$ and $L = y^3 - 3y + 1$.

* Means that Faddeev is not applicable in this case (since $p < n$).

Fourth Group : Jou(n, x) †

(Matrices with very small rank)

matrix		berkosam	linalpoly	Faddeev	barmodif	Chistov
$n = 10$	CPU Time	6.27 "	9"	4.55 "	6"	14"
	Mem. Alloc.	4. 980 Mb	4. 600 Mb	4. 455 Mb	4. 586 Mb	5. 110 Mb
$n = 15$	CPU Time	1' 00"	5' 10"	16.47 "	26"	2' 08"
	Mem. Alloc.	7. 207 Mb	13. 200 Mb	4. 717 Mb	5. 504 Mb	7. 338 Mb
$n = 20$	CPU Time	5' 09"	$\approx$ 3H 30'	43"	1' 12"	10' 55"
	Mem. Alloc.	11. 400 Mb	OUT	5. 668 Mb	7. 077 Mb	10. 352 Mb
$n = 25$	CPU Time	19' 01"	$\approx$ 3H 30'	1' 39"	2' 37"	44' 21"
	Mem. Alloc.	17. 429 Mb	OUT	7. 240 Mb	7. 863 Mb	18. 476 Mb

† $\mathrm{Jou}(n, x)$ is an $n \times n$ matrix with entries $J_{ij}(x) \in \mathbf{Z}[x]$ given by the formula :
$J_{ij}(x) = x^2(x - ij)^2 + (x^2 + j)(x + i)^2 + x \quad \text{for } 1 \leq i, j \leq n.$
It is of rank ≤ 3 for all x and every positive integer n. Notice the striking superiority of the Faddeev and the Jordan-Bareiss modified methods in this exceptional case.

Fifth Group : randmatrix$(n, n, sparse)$

(Sparse integer matrices)

matrix		berksparse	chisparse ‡	linalpoly	barmodif	Faddeev
$n = 32$	CPU Time	5.13 "	35.72 "	59.23 "	15.73 "	2' 13"
	Mem. Alloc.	3. 668 Mb	4. 062 Mb	4. 848 Mb	4. 848 Mb	4. 586 Mb
$n = 50$	CPU Time	21"	56"	5' 32"	4' 32"	12' 29"
	Mem. Alloc.	3. 930 Mb	4. 586 Mb	5. 110 Mb	6. 945 Mb	5. 504 Mb
$n = 64$	CPU Time	42"	2' 00"	14' 46"	7' 10"	32' 35"
	Mem. Alloc.	4. 062 Mb	4. 848 Mb	5. 240 Mb	6. 814 Mb	6.028 Mb
$n = 128$	CPU Time	6' 27"	20' 38"	4H 29' 43"	8H 17' 55"	stopped
	Mem. Alloc.	4. 978 Mb	7. 076 Mb	18. 346 Mb	16. 774 Mb	$>$ 19 hours
$n = 200$	CPU Time	29' 20"	2H 00' 19"	stopped	stopped	–
	Mem. Alloc.	5. 110 Mb	21. 098 Mb	$>$ 16 hours	$>$ 16 hours	–

‡ The sparse version of Chistov's algorithm.