Strongly Connected Graph Components and Computing Characteristic Polynomials of Integer Matrices in Maple

Simon Lo *, Michael Monagan *, Allan Wittkopf *
{sclo,mmonagan,wittkopf}@cecm.sfu.ca
Centre for Experimental and Constructive Mathematics,
Department of Mathematics, Simon Fraser University,
Burnaby, B.C., V5A 1S6, Canada.

Abstract

Let $\mathbf{A}$ be an $n \times n$ matrix of integers. We present details of our Maple implementation of a simple modular method for computing the characteristic polynomial of $\mathbf{A}$. We consider several different representations for the computation modulo primes, in particular, the use of double precision floats.

The algorithm used in Maple releases 7–10 is the Berkowitz algorithm. We present some timings comparing the two algorithms on a sequence of matrices arising from an application in combinatorics of Jocelyn Quaintance. These matrices have a hidden block structure. Once identified, we can further reduce the computing time dramatically.

This work has been incorporated into Maple 11's LinearAlgebra package.

1 Introduction

Let $\mathbf{A}$ be an $n \times n$ matrix of integers and let $c(x)$ be the characteristic polynomial of $\mathbf{A}$. The algorithm used in Maple releases 7–10 to compute $c(x)$ is the Berkowitz algorithm [2]. The Berkowitz algorithm is a division free algorithm and thus can be used to compute the characteristic polynomial of a square matrix over any commutative ring R. It does $O(n^4)$ multiplications in R. The Berkowitz algorithm is not the fastest algorithm for matrices of integers, nor is it asymptotically the fastest division free algorithm, but it is a good method for matrices of multivariate polynomials and general formulae.

In [1], Abdeljaoued compares a Maple implementation of the Berkowitz algorithm with several other "classical" methods for computing characteristic polynomials. The data obtained shows the Berkowitz algorithm to be best for for a variety of coefficient rings R.

In section 2 we present details of our Maple implementation of a modular method for computing $c(x)$, the characteristic polynomial of an integer matrix $\mathbf{A}$. The algorithm computes

*Supported by NSERC of Canada and the MITACS NCE of Canada.

the characteristic polynomial modulo a sequence of primes and applies the Chinese remainder theorem. For each prime we use the "Hessenberg algorithm" as described in Chapter 2 of Cohen's book [3]. This is an $O(n^3)$ algorithm.

In section 3 we consider several different representations for computation modulo primes, including the use of double precision floats and delayed floating point divisions. Some timings are presented comparing the different representations on different hardware for a 364×364 sparse matrix given to us by Jocelyn Quaintance [8]. This matrix, which motivated our work on computing characteristic polynomials of integer matrices, arises from a problem in combinatorics involving 3-dimensional lego blocks. It is one of sequence of moderately sparse integer matrices of dimension 72×72, 364×364 and 1916×1916. The matrices are available from

`http://www.cecm.sfu.ca/CAG/products2005.shtml`.

They all have hidden block structures, such that, once identified, can easily be exploited to reduce the time further. In section 4 we describe the command
`StronglyConnectedBlocks`, a new command in Maple 11's LinearAlgebra package, which identifies the block structure. We present some timings for this command applied to the matrices from Quaintance, as well as timings for computation of the characteristic polynomial, eigenvalues, and determinant for these matrices utilizing the block structure.

For the 1916 by 1916 matrix, if we compute $c(x)$ using our modular algorithm only, we get a speedup of a factor of ≈ 80 in comparison with Maple's Berkowitz algorithm. By identifying the blocks we get a further speed up of a factor of ≈ 600.

2 Characteristic Polynomial Algorithms

2.1 A dense deterministic modular algorithm.

Let $\mathbf{A} \in \mathbb{Z}^{n \times n}$ be the input matrix. To compute the characteristic polynomial $c(x) \in \mathbb{Z}[x]$ we compute $c(x)$ modulo a sequence of primes $p_1, p_2, p_3, ...$ and apply the Chinese remainder theorem to reconstruct the integer coefficients of $c(x)$. To compute $c(x)$ modulo a prime p we use the "Hessenberg matrix algorithm" (see Chapter 2 of Cohen [3] for a description). The cost is $O(n^3)$ arithmetic operations in $\mathbb{Z}_p$.

The Hessenberg algorithm proceeds in two steps. In the first step the matrix is moved into upper Hessenberg form (i.e. only the first subdiagonal can be nonzero in the lower triangular part of the matrix) with a sequence of elementary row and corresponding column operations that preserves $c(x)$. This requires $O(n^3)$ operations in $\mathbb{Z}_p$. In the second step, the characteristic polynomial $c(x) = p_{n+1}(x) \in \mathbb{Z}_p[x]$ of the upper Hessenberg form can be efficiently computed bottom up (using $O(n^3)$ arithmetic operations in $\mathbb{Z}_p$) from the following recurrence for $p_k(x)$:

$$p_{k+1}(x) = \begin{cases} 1 & k = 0 \\ (x - A_{k,k})\, p_k(x) - \sum_{i=1}^{k-1} (\prod_{j=i}^{k-1} A_{j+1,j})\, A_{i,k}\, p_i(x) & 1 \le k \le n+1 \end{cases}$$

2.2 Complexity and related work.

Let $\mathbf{A}$ be an $n \times n$ matrix of integers. To compare the running times of the Berkowitz algorithm and the modular algorithm, we suppose that the entries of $\mathbf{A}$ are bounded by B^m in magnitude where B is the base (usually 2^{31} or 2^{32} on a 32 bit machine) of the integer representation, that is, they are bounded by m base B digits in length. For both algorithms, we need a bound S on the size of the coefficients of the characteristic polynomial $c(x)$. A generic bound on the size of the determinant of $\mathbf{A}$ is sufficient since, in general, this is the largest coefficient of $c(x)$. The magnitude of the determinant of $\mathbf{A}$ is bounded by $S = n!B^{mn}$ and its length is bounded by $n \log_B n + mn$ base B digits. Tighter bounds are available but they will not affect the asymptotics. If $B = 2^{31}$ then we may assume $\log_B n < 1$ in practice, hence, the length of the determinant is bounded by $O(mn)$ base B digits.

The Berkowitz algorithm does $O(n^4)$ integer multiplications, additions and subtractions on integers of average size $O(mn)$ base B digits, hence, it's complexity is $O(n^4 M(mn))$ where $M(mn)$ is the cost of multiplying integers of length $O(mn)$ base B digits. Maple 9 and subsequent versions use the GMP integer arithmetic package, which uses the FFT for integer multiplication. Thus the complexity of Maple's Berkowitz implementation is $\tilde{O}(n^5 m)$.

In the modular algorithm, we will need $O(mn)$ machine primes. The cost of reducing the n^2 integers in $\mathbf{A}$ modulo one prime is $O(mn^2)$. The cost of computing the characteristic polynomial modulo each prime p is $O(n^3)$. The cost of the Chinese remaindering assuming a classical method for the Chinese remainder algorithm (because that is what Maple uses) is $O(n(mn)^2)$. Thus the total complexity is $O(mnmn^2 + mnn^3 + n(mn)^2) = O(m^2 n^3 + mn^4)$.

If we assume $m = O(n)$, that is, the size of the integers grow proportionately with the size of the matrix, the complexity of Maple's Berkowitz algorithm is $\tilde{O}(n^6)$ and that of the modular algorithm is $O(n^5)$.

Another special case of interest is $m = 1$, that is, $|A_{i,j}| < B$ for all n. In this case the complexity of the Berkowitz algorithm is $\tilde{O}(n^5)$ and that of the modular algorithm is $O(n^4)$.

Asymptotically faster algorithms for computing $c(x)$ are known. Within the computer algebra community, quite a bit of work, both theoretical and practical has been done on this problem. Here we mention the algorithm of Kehller-Gehrig [6] which computes the characteristic polynomial of a matrix A in $O(n^\omega \log n)$ field operations where n^ω is the cost of matrix matrix multiplication. Combining this algorithm with the Chinese remainder algorithm gives an $O(mn^{\omega+1} \log n)$ algorithm.

Another general approach is to look for a linear dependency amongst the Krylov iterates, Av, A^2v, A^3v, ... The paper [4] by Dumas, Pernet, and Wan compares several approaches for computing characteristic polynomials modulo p and shows that the authors Krylov based algorithm, which is $O(n^3)$, is very good on dense matrices. Their algorithm takes advantage of a BLAS implementation of matrix-matrix multiplication.

Also Kaltofen and Villard in [5] give two algorithms, one using classical matrix multiplication in $\tilde{O}(mn^{3.2})$ and one (sub-cubic in n) assuming fast matrix multiplication in $\tilde{O}(mn^{2.6})$. They also give two division free algorithms for an arbitrary commutative ring. Again, one assuming classical matrix multiplication in $\tilde{O}(n^{3.2})$ ring operations and one assuming fast matrix multiplication in $O(n^{2.6})$ ring operations.

We have not implemented any of the fast algorithms.

3 Timings and Implementations

For a machine prime p, in order to improve the running time of our implementation, we've implemented the Hessenberg algorithm over $\mathbb{Z}_p$ in the C programming language and the rest of the algorithm in Maple. We used the Maple external function interface to call the C code (see Ch. 6 of [7]). We've implemented a 32-bit integer version, a 64-bit integer version, and several versions using 64-bit double precision floating point values for comparison.

The following table consists of timings (in CPU seconds) of our C implementation of the Hessenberg algorithm for the 364×364 input matrix from Quaintance [8]. Rows 1-9 below are for the modular algorithm using different implementations of arithmetic for $\mathbb{Z}_p$. The accelerated floating point version **fprem** using 25-bit primes generally gives the best times. The performance of 64 bit integer arithmetic is disappointing.

Versions	Xeon 2.8 GHz	Opteron 2.0 GHz	AXP2800 2.08 GHz	Pentium M 2.00 GHz	Pentium 4 2.80 GHz
64int	100.7	107.4			
32int	66.3	73.0	76.8	35.6	57.4
new 32int	49.7	54.7	56.3	25.5	39.6
fmod	29.5	32.1	33.0	35.8	81.1
trunc	67.8	73.7	69.6	88.5	110.6
modtr	56.3	62.5	59.5	81.0	82.6
new fmod	11.0	11.6	14.5	15.2	28.8
fprem	10.4	10.9	13.7	13.9	26.8
fLA	17.6	19.9	21.9	26.2	27.3
Berkowitz	2053.6	2262.6			

Implementations

64int The 64-bit integer version is implemented using the *long long int* datatype in C, or equivalently the *integer[8]* datatype in Maple. We use 32-bit primes. All modular arithmetic first executes the corresponding 64-bit integer machine instruction, then reduces the result mod p.

32int The 32-bit integer version is similar, but implemented using the *long int* datatype in C, or equivalently the *integer[4]* datatype in Maple. 16-bit primes are used here.

new 32int This is an improved **32int**, with various hand/compiler optimizations.

fmod This 64-bit float version is implemented using the *double* datatype in C, or equivalently the *float[8]* datatype in Maple. 64-bit float operations are used to simulate integer operations. Operations such as additions, subtractions, multiplications are followed by a call to the C library function *fmod()* to reduce the results mod p. We allow both positive and negative floating point representations of integers with magnitude less than p.

trunc This 64-bit float version is similar to above, but uses the C library function *trunc()* instead of *fmod()*. To compute $b \leftarrow a \bmod p$, we first compute $c \leftarrow a - p \times \mathit{trunc}(a/p)$, then $b \leftarrow c$ if $c \neq \pm p$, $b \leftarrow 0$ otherwise. The *trunc()* function rounds towards zero to the nearest integer.

modtr A modified **trunc** version where we do not do the extra check for equality to $\pm p$ at the end. So to compute $b \leftarrow a \bmod p$, we actually compute $b \leftarrow a - p \times \mathit{trunc}(a/p)$, which results in $-p \leq b \leq p$.

new fmod An improved **fmod** version, where we have reduced the number of times *fmod()* is called - we reduce the results mod p only when the number of accumulated arithmetic operations on an entry exceeds a certain threshold. To allow this we are restricted to use 25-bit primes. We call this delayed mod acceleration. See the next subsection.

fprem Equivalent to **new fmod** version, but via direct assembly programming using *fprem* instruction, removing the function call overhead and making some efficiency improvements.

fLA An improved **trunc** version using delayed mod acceleration. It is the default used in Maple's `LinearAlgebra:-Modular` routines in Maple 10 and earlier.

3.1 Efficiency considerations.

There are a few considerations for use of floating point for mod p computations. Keeping these in mind, one can implement faster code for the algorithms than is possible with the integer codes, and still have the advantage of using larger primes on 32-bit machines. These ideas are used by Maple's `LinearAlgebra:-Modular` package since Maple 8 for floating point modular operations.

1. Although floating point integer computations can represent 53-bit numbers accurately, restricting the modulus to $p < 2^{25}$ allows for more efficient mod operations, and multiple mod operations (up to 8 for maximal sized primes) to occur before having to reduce modulo p. We call this the delayed mod acceleration.

2. Leveraging the smaller primes allows up to 8 computations (using a maximal size prime) to occur before we must perform a mod. This can be efficiently utilized in row-based algorithms, as a counter associated with each row can count the number of operations performed, and the algorithms can be made to only perform the mod once the maximal number of computations is reached. Note, one can use level 1 BLAS here for single row/column operations but this did not improve the efficiency.

In our experiments we found the following: Use of the smaller primes, and delayed mod, mentioned in items 1 and 2 above increased performance by a factor of 2-3. With these modifications, use of floating point modular arithmetic demonstrated better performance than integer modular arithmetic.

The use of the C-library *fmod* function or direct assembly programming using the *fprem* instruction (essentially equivalent modulo function call overhead and some efficiency improvements made available for our specific use of *fmod*) showed better performance than the

other floating point schemes, except on the Pentium 4, on which it was approximately equal. Note also that on Pentium M the *fprem* performance was nearly a factor of 2 times better.

3.2 Timings for dense matrices.

The following table consists of some timings (in seconds) of our modular Hessenberg algorithm using float (**fprem**) and integer (**new 32int**) implementations on dense $n \times n$ matrices, with uniformly random integer entries between -999 and 999. We also compare with Maple's Berkowitz algorithm. The timings were made on an AMD Opteron 2.2 GHz processor. From the above data, we can see that the float version is always faster than the integer version (about 3 times faster).

n	float	integer	Berkowitz
50	<0.1	0.13	7.85
100	0.61	1.85	128.6
200	8.82	30.8	2248.1
300	45.4	153.2	
400	173.5	493.4	
600	1195.2	2973.1	
800	4968.8		

3.3 Making the algorithm output sensitive.

In [4], Dumas *et. al.* propose the use of early termination to recover the characteristic polynomial and they observe a 25% improvement on one set of data. Our implementation also uses early termination and the following example suggests that we should always do this because the improvement could be arbitrary. Consider the following matrix

$$\mathbf{A} = \begin{pmatrix} 1 & u & v & w \\ 0 & 2 & x & y \\ 0 & 0 & 3 & z \\ 0 & 0 & 0 & 4 \end{pmatrix}.$$

The characteristic polynomial of $\mathbf{A}$ is $c(x) = (x-1)(x-2)(x-3)(x-4) = x^4 - 10x^3 + 35x^2 - 50x + 24$. Notice that the largest coefficient of $c(x)$ does not depend on any of the entries u, v, w, x, y, z. So if u, v, w, x, y, z are large, a bound S for the largest coefficient of $c(x)$ could be arbitrarily far off and our modular algorithm would use many more primes than are necessary.[1]

This suggests that we use an output sensitive version of the algorithm and not use a bound at all. We incrementally apply the Chinese remainder theorem to reconstruct $c(x)$ and stop when the output of applying Chinese remaindering on K consecutive modular images "remains the same". On the sparse 364×364 example in the previous section, using $K = 4$, the timings improve by about 50%.

[1]Yes, we could improve the algorithm for computing the bound S to "notice" that this matrix is upper triangular. But we could fool such a bound algorithm by, for example, permuting the rows and columns of A

4 Strongly Connected Graph Components

Consider again the matrix

$$\mathbf{A} = \begin{pmatrix} 1 & u & v & w \\ 0 & 2 & x & y \\ 0 & 0 & 3 & z \\ 0 & 0 & 0 & 4 \end{pmatrix}.$$

The Hessenberg algorithm does not need to do any work since $\mathbf{A}$ is already in upper Hessenberg form. However, it would do work if the input were $\mathbf{A}^T$ even though the characteristic polynomial would be the same. Consider the block upper-triangular matrix

$$\mathbf{B} = \begin{pmatrix} a & b & w & x \\ c & d & y & z \\ 0 & 0 & e & f \\ 0 & 0 & g & h \end{pmatrix}.$$

Let $c_{\mathbf{B}}(x)$ be the characteristic polynomial of $\mathbf{B}$. Then $c_{\mathbf{B}}(x) = c_{\mathbf{B}_1}(x)c_{\mathbf{B}_2}(x)$ where

$$\mathbf{B}_1 = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \text{ and } \mathbf{B}_2 = \begin{pmatrix} e & f \\ g & h \end{pmatrix}.$$

If the input matrix is block upper (lower) triangular, then the computation of the characteristic polynomial reduces to the product of the characteristic polynomials of the diagonal blocks which are easier to compute. But what if the input matrix is a row and column permutation of a block upper (lower) triangular matrix? For example,

$$\mathbf{P} = \begin{pmatrix} h & 0 & g & 0 \\ z & d & y & c \\ f & 0 & e & 0 \\ x & b & w & a \end{pmatrix}$$

is the matrix $\mathbf{B}$ with rows 1,4 and columns 1,4 interchanged. Since simultaneous row and column interchanges do not change the characteristic polynomial, if we could efficiently determine the permutation that would make P block upper (lower) triangular, we could compute $c(x)$ faster. It turns out that we can compute this permutation in linear time by finding the *strongly connected components* of a directed graph.

Let $\mathbf{A}$ be a $n \times n$ matrix. We denote by $Graph(\mathbf{A})$ the weighted directed graph with n vertices such that $\mathbf{A}$ is the adjacency matrix of $Graph(\mathbf{A})$. Recall that a directed graph G is strongly connected if for each pair of vertices $u, v \in G, u \neq v$, there exists a path $u \rightsquigarrow v$. Also, every directed graph can be partitioned into maximal strongly connected components.

Denote by $\mathbf{A}_{(u_1,u_2,...,u_r),(v_1,v_2,...,v_s)}$ the $r \times s$ submatrix of $\mathbf{A}$ such that

$$(\mathbf{A}_{(u_1,u_2,...,u_r),(v_1,v_2,...,v_s)})_{i,j} = \mathbf{A}_{u_i,v_j}.$$

The method works as follows:

1. Compute (see below) the k strongly connected components of $Graph(\mathbf{A}) : V_1, V_2, ..., V_k$ where $V_i = \{v_{i1}, v_{i2}, ..., v_{in_i}\}$.

2. For $1 \leq i \leq k$, compute the characteristic polynomial of the matrix $\mathbf{A}_{V_i,V_i}$ which is the n_i by n_i submatrix $\mathbf{A}_{(v_{i1},v_{i2},\ldots,v_{in_i}),(v_{i1},v_{i2},\ldots,v_{in_i})}$.

3. Output (the characteristic polynomial $c(x)$ of $\mathbf{A}$) the product of the characteristic polynomials computed in step 2.

For the matrix P above, the strongly connected components of $Graph(\mathbf{P})$ are $V_1 = \{1, 3\}$ and $V_2 = \{2, 4\}$. To see this, it might help the reader to draw $Graph(\mathbf{P})$. Next we compute the product of the characteristic polynomials of

$$\begin{pmatrix} h & g \\ f & e \end{pmatrix} \text{ and } \begin{pmatrix} d & c \\ b & a \end{pmatrix}.$$

Clearly we would get the characteristic polynomial of $\mathbf{P}$, which is also the characteristic polynomial of $\mathbf{B}$. Observe that $V_2 \prec V_1$. Suppose we choose the ordering 2,4,1,3, then we obtain the matrix

$$\mathbf{P}' = \begin{pmatrix} d & c & z & y \\ b & a & x & w \\ 0 & 0 & h & g \\ 0 & 0 & f & e \end{pmatrix}.$$

It doesn't quite look like $\mathbf{B}$, but the characteristic polynomial of $\mathbf{P}'$ is the same as $\mathbf{B}$ above. The difference is that the vertices in V_1 and V_2 are permuted. If we choose the ordering 4,2,3,1, then we obtain $\mathbf{B}$.

4.1 LinearAlgebra[StronglyConnectedBlocks]

Computation of the strongly connected component blocks of a Matrix is implemented in the `LinearAlgebra` package in Maple 11 as the command `StronglyConnectedBlocks`. It computes the strongly connected components of $G = Graph(A)$ using the algorithm of Tarjan [9]. The running time is linear in the number of edges of G, that is, linear in the number of non-zero entries of A. The syntax for the Maple command is:

```
LinearAlgebra[StronglyConnectedBlocks](M, [returnsingular=truefalse])
```

where $\mathbf{M}$ is the input and the option **returnsingular** is provided to allow one to avoid reconstruction of the block matrices if the input matrix can be determined to be singular (useful for determinant computation).

The output of `StronglyConnectedBlocks` is a list of non-zero square matrices $\mathbf{A}_1, \mathbf{A}_2$, ..., $\mathbf{A}_r$. If the input $\mathbf{M}$ is an $n \times n$ matrix, and $m = \sum_{i=1}^{r} \dim(\mathbf{A}_i)$, the output satisfies $c_{\mathbf{M}}(x) = x^{n-m} \prod_{i=1}^{r} c_{\mathbf{A}_i}(x)$ where $c_{\mathbf{A}}(x)$ is the characteristic polynomial of $\mathbf{A}$.

We now provide data comparing the run-time and memory usage for the computation of the strongly connected blocks of the 72×72, 364×364, and 1916×1916 matrices from Quaintance [8]. Note that the 364×364 matrix decomposes into 12 blocks of sizes $5, 5$, 9, $10, 10, 10$, $22, 22$, 48, 54, 76, 93 while the 72×72 matrix decomposes into 6 blocks of sizes $4, 4$, 8, 13, 20, 23, and the 1916×1916 matrix decomposes into 31 blocks of sizes $6, 6, 6, 6, 6, 6$, $11, 11$, $12, 12, 12, 12, 12, 12, 12, 12, 12$, $13, 13$, $32, 32$, 59, $70, 70, 70$, $103, 103$, 241, 260, 306, 378.

Note that one bottleneck in the implementation of the algorithm is the extraction of the nonzero entries in the input matrix, so the timings table compares a Maple implementation v.s. a C implementation for both the SCC and extract steps.

The timings below are for an AMD Athlon(tm) 64 X2 4400+ (2.2GHz) with 2GB memory under 64-bit Linux.

Matrix	Maple SCC Maple Extract		Maple SCC C Extract		C SCC Maple Extract		C SCC C Extract	
72×72	0.01s	0.52M	0.02s	0.13M	0.01s	0.39M	.004s	<.001M
364×364	0.20s	5.24M	0.06s	2.75M	0.09s	5.11M	.004s	0.26M
1916×1916	5.20s	51.1M	1.06s	12.8M	3.57s	46.4M	0.11s	5.11M

From this we can see that the C implemented extraction process provides a significant reduction in the run time and memory usage of the implementation, and the combination of the two reduces the time and memory usage for the difficult example by factors of $\approx 50, 10$ respectively.

4.2 Final timings.

To complete the section on improvements we provide a timing comparison for computation of the characteristic polynomial, the eigenvalues and the determinant of the three matrices of the prior section (72×72, 364×364, and 1916×1916).

In the following table, the Maple 11 Berkowitz implementation is compared with the new Maple 11 Hessenberg implementation (with early termination), computing both with and without block decomposition. Timings were obtained on an AMD Athlon(tm) 64 X2 Dual Core 4400+ (2.2GHz) with 2GB memory under 64-bit Linux.

Matrix	Berkowitz No blocks	Berkowitz With blocks	Hessenberg No blocks	Hessenberg With blocks
72×72	2.18	0.04	0.16	0.02
364×364	966.43	9.59	28.45	0.16
1916×1916	2,349,828.7	16594.79	28468.55	46.87

It is clear from the timings that block decomposition is very effective.

The block decomposition algorithm has also been integrated into the *LinearAlgebra:-Determinant* code for the non-floating point cases. This integration took some care, as in cases where no decomposition is possible, the time for the check is simply added to the overall run-time. In the original integration, before converting the block algorithm to C code, most determinant tests in the Maplesoft test suite slowed down, and the overall hit was approximately 20% on average. After the conversion of the block algorithm to C code, most tests ran faster, and the average speedup was 4%, with the largest speed up at 88% (a test that ran in 10.7 sec. was reduced to 1.3 sec.), and the largest slowdown was 4.1% (a test that ran in 12.87 sec. increased to 13.39 sec.). The latter was simply a case where the Matrix was large and dense, and no block decomposition could be performed.

Note that the improvements to *LinearAlgebra:-CharacteristicPolynomial* also improve the timings for *LinearAlgebra:-Eigenvalues*, to the point where the majority of the time is spent

extracting the eigenvalues from the characteristic polynomial, i.e., factoring the $c(x)$ and expressing roots in terms of radicals where possible.

	Determinant		CharPoly		Eigenvalues	
Matrix	Old	New	Old	New	Old	New
72×72	0.03	0.02	2.18	0.02	2.34	0.14
364×364	0.60	0.12	966.43	0.16	964.91	1.57
1916×1916	24.97	2.37	27 days	46.87	N/A*	335.61

The old eigenvalue timing (N/A*) was deemed too expensive to bother measuring as it will be on the same order as the old characteristic polynomial timing (27 days).

We note that there is also a significant improvement in the determinant computation in Maple 11 that is independent of the block decomposition. Maple 11 is using a modular algorithm for determinant of integer matrices. For the 1916×1916 Matrix, Maple 10 required 4500 seconds and memory usage of 1.4 gigabytes, while in Maple 11 (prior to the block decomposition changes) the same computation only required 24.97 seconds with no excessive memory usage. The block decomposition has further improved this timing to 2.37 seconds.

References

[1] J. Abdeljaoued. The Berkowitz Algorithm, Maple and Computing the Characteristic Polynomial in an Arbitrary Commutative Ring. *MapleTech* **5**(1), pp. 21–32, Birkhauser, 1997.

[2] S. J. Berkowitz. On Computing the Determinant in Small Parallel time using a Small Number of Processors. *Inf. Processing Letters* **18**(3) pp. 147–150, 1984.

[3] H. Cohen. *A Course in Computational Algebraic Number Theory.* Graduate texts in mathematics, **138**, Springer-Verlag, 1995.

[4] J.-G. Dumas, C. Pernet, Z. Wan. Efficient Computation of the Characteristic Polynomial. *Proceedings of ISSAC 2005,* pp. 140–147, ACM Press, 2005

[5] E. Kaltofen, G. Villard. On the Complexity of Computing Determinants. *Journal of Computational Complexity* **13** pp. 91–130, 2004.

[6] W. Keller-Gehrig. Fast algorithms for the characteristic polynomial. *Theoretical Computer Science* **36** 309-317, Elsevier, 1985.

[7] M. Monagan, K. Geddes, K. Heal, G. Labahn, S. Vorkoetter, J. McCarron, P. deMarco. *Maple 9 Advanced Programming Guide*, Maplesoft, 2003.

[8] J. Quaintance. $m \times n$ Proper Arrays: Geometric Construction and the Associated Linear Cellular Automata. *Proceedings of the 2004 Maple Summer Workshop*, 2004.

[9] R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing* **1**(2) pp. 146–160, 1972.