A Modular Algorithm for Computing the Characteristic Polynomial of an Integer Matrix in Maple.

Simon Lo and Michael Monagan*
Department of Mathematics, Simon Fraser University,
Burnaby, B.C., V5A 1S6, Canada.

Introduction

Let A be an $n \times n$ matrix of integers. In this paper we present details of our Maple implementation of a modular method for computing $c(x)$, the characteristic polynomial of A. Our implementation considers several different representations for the primes, including the use of double precision floats. The algorithm presently implemented in Maple releases 7–9 is the Berkowitz algorithm [2, 1]. We present some timings comparing the two methods on a 364×364 matrix arising from an application in combinatorics from Quaintance [6].

One way to compute the characteristic polynomial of A is to evaluate the characteristic matrix at n points, compute n determinants of integer matrices, then interpolate to obtain the characteristic polynomial. The determinants of the integer matrices can be computed using a fraction-free Gaussian elimination algorithm (see Chapter 9 of Geddes et. al [4]) in $\mathrm{O}(n^3)$ integer multiplications and divisions. This approach will lead to an algorithm that requires $\mathrm{O}(n^4)$ integer operations.

Another algorithm is the "Berkowitz" algorithm [2]. It is a division free algorithm and thus can be used to compute the characteristic polynomial of a matrix over any commutative ring R. It does $\mathrm{O}(n^4)$ multiplications in R. In [1], Abdeljaoued described a Maple implementation of a sequential version of the Berkowitz algorithm and compared it with the interpolation method and other methods. His implementation improves with sparsity to $\mathrm{O}(n^3)$ multiplications when the matrix has $\mathrm{O}(n)$ non-zero entries.

A Modular Algorithm

The modular algorithm we have implemented computes the characteristic polynomial of A modulo a sequence of machine primes $p_1, p_2, ...$ and applies the Chinese remainder theorem to reconstruct the coefficients of the characteristic polynomial. For each prime p it computes the characteristic polynomial modulo p via the Hessenberg matrix in $\mathrm{O}(n^3)$ arithmetic operations in $\mathbb{Z}_p$. The algorithm is described in Chapter 2 of Cohen's book. See [3].

Consider a sequence of k machine primes $p_1, p_2, \ldots, p_k$, where $2 < p_i < B$. B should be chosen small enough such that B^2 fits into a single register, so arithmetic operations in $\mathbb{Z}_p$ can be done directly by the hardware, but also large enough so that we don't require too many primes. Therefore, we let $B = \lfloor\sqrt{2^{31}-1}\rfloor$ for 32-bit integers, $B = \lfloor\sqrt{2^{63}-1}\rfloor$ for 64-bit integers, and $B = \lfloor\sqrt{2^{52}-1}\rfloor$ for 64-bit floating point representations.

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

Algorithm:

1. Compute a bound M for the size of coefficients of $c(x)$.
2. Choose k machine primes $p_1, p_2, \ldots, p_k < B$ such that $\prod_{i=1}^{k} p_i > 2M$.
3. **for** $i = 1$ to k **do**
   (a) $A_i \leftarrow A \bmod p_i$.
   (b) Compute $c_i(x)$ — the characteristic polynomial of A_i over $\mathbb{Z}_{p_i}$.
4. Apply the Chinese remainder theorem to reconstruct $c(x)$ from the $c_i(x)$'s.

Implementation Details and Timings

In order to improve the running time of our algorithm, we've implemented the Hessenberg algorithm over $\mathbb{Z}_{p_i}$ in the C programming language and the rest of the algorithm in Maple. We used the Maple foreign function interface to call the C code. See [5]. We've implemented both the 32-bit integer version and 64-bit integer versions, and also several versions using 64-bit double precision floating point values for comparison.

The following table consists of some timings of our modular Hessenberg algorithm for a sparse 364×364 input matrix arising from an application in combinatorics. See [6]:

Representation of values	time (*secs*)[1]	time (*secs*)[2]	time (*secs*)[3]
1. 64-bit integer	132.4	100.7	109.7
2. 32-bit integer	48.2	68.4	45.7
3. 64-bit float using *fmod()*	58.7	22.3	140.9
4. 64-bit float using *floor()* with fix	46.2	42.2	49.8
5. 64-bit float using *floor()* without fix	38.8	36.3	42.0
6. Berkowitz algorithm	2470.2	2053.6	1886.2

Explanations of the different representations of values:

1. The 64-bit integer version is implemented using the *long long int* datatype in C, or equivalently the *integer[8]* datatype in Maple. All modular arithmetic is first being done by executing the corresponding operation, then taking the result mod p because we work in $\mathbb{Z}_p$. In order to compute the inverses mod p, we have implemented the half extended Euclidean Algorithm in C.

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

3. The 64-bit float using *fmod()* version is implemented using the *double* datatype in C, or equivalently the *float[8]* datatype in Maple. This works because floating point numbers are stored as mantissa and exponent, thus any integer a with $a^2 \leq B^2$ can be represented exactly as a 64-bit floating point number. Operations such as additions, subtractions, multiplications are followed by a call to *fmod()* to reduce the results mod p, since we are working in $\mathbb{Z}_p$.

[1] Intel Xeon 2.0 GHz 32-bit processor
[2] Operon 246 2.0 GHz 64-bit processor
[3] Optipex Pentium IV 3 GHz 32-bit processor

4. The 64-bit float using *floor()* with fix version is similar to above, but uses *floor()* instead of *fmod()*. To compute $b \leftarrow a \bmod p$, we first compute $c \leftarrow a - p \times \lfloor a/p \rfloor$, then $b \leftarrow c$ if $c \neq p$, $b \leftarrow 0$ otherwise.

5. The 64-bit float using *floor()* without fix vision is similar to above, but does not do the extra check for equality to p at the end. So to compute $b \leftarrow a \bmod p$, we actually compute $b \leftarrow a - p \times \lfloor a/p \rfloor$, which results in $0 \leq b \leq p$.

Asymptotic Comparison of the Methods

Let A be an $n \times n$ matrix of integers. To compare the running times of the two algorithms, we suppose that the entries of A are bounded by B^m in magnitude, that is, they are m base B digits in length. For both algorithms, we need a bound on the size of the coefficients of the characteristic polynomial $c(x)$. A generic bound on the size of the determinant of A is sufficient since this is the largest coefficient of $c(x)$. The magnitude of the determinant of A is bounded by $M = n!B^{mn}$ and its length is bounded by $n \log_B n + mn$ base B digits. If $B > 2^{15}$ then we may assume $\log_B n < 2$ in practice and hence the length of the determinant is $O(mn)$ base B digits.

In Berkowitz's algorithm, the $O(n^4)$ integer multiplications are on integers of average size $O(mn)$ digits in length, hence the complexity (assuming classical integer arithmetic is used) is $O(n^4(mn)^2)$. Since Maple 9 uses the FFT for integer multiplication and division, the complexity is reduced to $\tilde{O}(n^5m)$.

In the modular algorithm, we will need $O(mn)$ machine primes. The cost of reducing the n^2 integers in 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 (which is what Maple uses) is $O(n(mn)^2)$. Thus the total complexity is $O(mnmn^2 + mnn^3 + n(mn)^2) = O(m^2n^3 + mn^4)$.

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. Porcessing 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] K. O. Geddes, S. R. Czapor, and G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publ., Boston, Massachusetts, USA, 1992.

[5] M. Monagan, K. Geddes, K. Heal, G. Labahn, S. Vorkoetter, J. McCarron, P. deMarco. *Maple 9 Advanced Programming Guide*, Ch. 6, Maplesoft, 2003.

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