Computing Characteristic Polynomials over $\mathbb{Z}$

Simon Lo   Michael Monagan   Allan Wittkopf

Introduction

We present a modular algorithm for computing the characteristic polynomial of an integer matrix. The computation modulo each prime is done using the Hessenberg algorithm. It is implemented in C and the rest of the algorithm is implemented in Maple. We compare three implementations for arithmetic over $\mathbb{Z}_p$: 32-bit integers, 64-bit integers, and also double precision floats. The best results use floats!

Modular Algorithm

Input: Matrix $\mathbf{A} \in \mathbb{Z}^{n\times n}$
Output: Characteristic polynomial $c(x) = \det(x\mathbf{I} - \mathbf{A}) \in \mathbb{Z}[x]$

1. Compute a bound S larger than the largest coefficient of $c(x)$.
2. Choose t machine primes $p_1, p_2, \ldots, p_t$ such that $\prod_{i=1}^{t} p_i > 2S$.
3. **for** $i = 1$ **to** t **do**
   (a) $\mathbf{A}_i \leftarrow \mathbf{A} \bmod p_i$.
   (b) Compute $c_i(x)$ — the characteristic polynomial of $\mathbf{A}_i$ over $\mathbb{Z}_{p_i}$ via the Hessenberg algorithm.
4. Apply the Chinese remainder theorem:
   Solve $c(x) \equiv c_i(x) \pmod{p_i}$ for $c(x)$.

Hessenberg Algorithm

Recall that a square matrix $\mathbf{M} = (m_{i,j})$ is in upper Hessenberg form if $m_{i,j} = 0$ for all $i \ge j+2$, in other words, the entries below the first subdiagonal are zero.

$$\begin{pmatrix} m_{1,1} & m_{1,2} & m_{1,3} & \cdots & m_{1,n-2} & m_{1,n-1} & m_{1,n} \\ m_{2,1} & m_{2,2} & m_{2,3} & \cdots & m_{2,n-2} & m_{2,n-1} & m_{2,n} \\ 0 & m_{3,2} & m_{3,3} & \cdots & m_{3,n-2} & m_{3,n-1} & m_{3,n} \\ 0 & 0 & m_{4,3} & \cdots & m_{4,n-2} & m_{4,n-1} & m_{4,n} \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ddots & m_{n-1,n-2} & m_{n-1,n-1} & m_{n-1,n} \\ 0 & 0 & 0 & \cdots & 0 & m_{n,n-1} & m_{n,n} \end{pmatrix}$$

The Hessenberg algorithm consists of the following two parts:

1. Reduce the matrix $\mathbf{M} \in \mathbb{Z}_p^{n\times n}$ into the upper Hessenberg form using a series of row and column operations in $\mathbb{Z}_p$, while preserving the characteristic polynomial (known as similarity transformations.) Below, $\mathbf{R}_i$ denotes the i'th row of $\mathbf{M}$ and $\mathbf{C}_j$ the j'th column of $\mathbf{M}$.

   for $j = 1$ **to** $n-2$ **do**
   search for a nonzero entry $m_{i,j}$ where $j+2 \le i \le n$
   if found **then**
   do $\mathbf{R}_i \leftrightarrow \mathbf{R}_{j+1}$ and $\mathbf{C}_i \leftrightarrow \mathbf{C}_{j+1}$ **if** $m_{j+1,j} = 0$
   for $k = j+2$ **to** n **do**
   (reduce using $m_{j+1,j}$ as pivot)
   $u \leftarrow m_{k,j}\, m_{j+1,j}{}^{-1}$
   $\mathbf{R}_k \leftarrow \mathbf{R}_k - u\mathbf{R}_{j+1}$
   $\mathbf{C}_{j+1} \leftarrow \mathbf{C}_{j+1} + u\mathbf{C}_k$
   else
   first j columns of $\mathbf{M}$ is already in upper Hessenberg form

2. The characteristic polynomial $c(x) = p_{n+1}(x) \in \mathbb{Z}_p[x]$ of the upper Hessenberg form can be efficiently computed from the following recurrence for $p_k(x)$ using computations in $\mathbb{Z}_p[x]$:

$$p_k(x) = \begin{cases} 1 & k = 1 \\ (x - m_{k,k})p_k(x) - \sum_{i=1}^{k-1}\left(\prod_{j=1}^{i} m_{j+1,j}\right) m_{i,k}\, p_i(x) & 1 < k \le n+1 \end{cases}$$

Complexity

Suppose that $\mathbf{A} = (a_{i,j})$ is a $n \times n$ integer matrix and $|a_{i,j}| < B^m$.
A bound for S is $n!B^{mn}$, therefore, under reasonable assumptions, length of the determinant of $\mathbf{A}$ is $O(mn)$ base B digits, so we'll need $O(mn)$ machine primes.
We have:

- Cost of reducing the n^2 entries in $\mathbf{A}$ modulo one prime is $O(mn^2)$.
- Cost of computing the characteristic polynomial modulo each prime p via the Hessenberg method is $O(n^3)$.
- Cost of a classical method for the Chinese remainder algorithm is $O(n(mn)^2)$.

Total complexity: $O(mnmn^2 + mnn^3 + n(mn)^2) = O(m^2n^3 + mn^4)$.
In contrast, the Berkowitz algorithm, the algorithm that Maple uses, has complexity $O(n^4(mn)^2)$, which reduces to $\tilde{O}(n^5 m)$ if the FFT is used.

Timings

The following is set of timings (in seconds) for a 364 by 364 sparse matrix arising from a combinatorial construction. Rows 1-8 below are for the modular algorithm using different implementations of arithmetic for $\mathbb{Z}_p$. The accelerated floating point versions using 25-bit primes generally give the best times.

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
fLA	17.6	19.9	21.9	26.2	27.3
Berkowitz	2053.6	2262.6	—.-	—.-	—.-

Explanations of the different versions:

64int 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 first executes the corresponding 64-bit integer machine instruction, then reduces the result mod p because we work in $\mathbb{Z}_p$. We allow both positive and negative integers of magnitude less than 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.

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 *fmod()* to reduce the results mod p, since we are working in $\mathbb{Z}_p$. 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 *trunc()* instead of *fmod()*. To compute $b \leftarrow a \bmod p$, we first compute $c \leftarrow a - p \times$ *trunc*(a/p), then $b \leftarrow c$ if $c \ne \pm p$, $b \leftarrow 0$ 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 $b \leftarrow a - p \times$ *trunc*(a/p), which results in $-p \le b \le p$.

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

fLA An improved **trunc** version using operation count acceleration. It is the default used in Maple's *LA:Modular* routines.

MITACS, NSERC
Computational Algebra Group
Centre for Experimental and Constructive Mathematics
Department of Mathematics
Simon Fraser University