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 \( A \in \mathbb{Z}^{m \times n} 

**Output:** Characteristic polynomial \( c(x) = \det(xI - 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) \( A_i \leftarrow A \mod p_i \).
    (b) Compute \( c(x) \leftarrow \) the characteristic polynomial of \( A_i \) over \( \mathbb{Z}_p \), via the Hessenberg algorithm.
4. Apply the Chinese remainder theorem:
   \[ \text{Solve } c(x) \equiv c(x) \mod p_i \text{ for } c(x). \]

### Hessenberg Algorithm

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

\[
\begin{pmatrix}
m_{11} & m_{12} & m_{13} & \cdots & m_{1,n-2} & m_{1,n-1} & m_{1,n} \\
m_{21} & m_{22} & m_{23} & \cdots & m_{2,n-2} & m_{2,n-1} & 0 \\
0 & m_{32} & m_{33} & \cdots & m_{3,n-2} & 0 & 0 \\
0 & 0 & m_{43} & \cdots & 0 & 0 & 0 \\
\vdots & \vdots & \ddots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & m_{n-1,n-2} & m_{n-1,n-1} & 0 \\
0 & 0 & 0 & \cdots & 0 & m_{n,n-1} & m_{n,n} \\
\end{pmatrix}
\]

The Hessenberg algorithm consists of the following two parts:

1. Reduce the matrix \( M \in \mathbb{Z}^{n \times n} \) 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, \( R_i \) denotes the \( i \)-th row of \( M \) and \( C_i \) the \( j \)-th column of \( M \).
   - for \( j = 1 \) to \( n - 2 \) do
     - search for a nonzero entry \( m_{kj} \) where \( j + 2 \leq i \leq n 
   - if found then
     - do \( R_i \leftarrow R_{j+1} \) and \( C_i \leftarrow C_{j+1} \) if \( m_{j+1,j} = 0 \)
     - for \( k = j + 2 \) to \( n \) do
       - (reduce using \( m_{j+1,j+1} \) as pivot)
       - \( u \leftarrow m_{j+1,j+1} \)
       - \( R_i \leftarrow R_{j+1} + uR_{j+2} \)
       - \( C_i \leftarrow C_{j+1} + uC_{j+2} \)
   - else
     - first \( j \) columns of \( M \) is already in upper Hessenberg form
   - 2. The characteristic polynomial \( c(x) = p_n(x) \in \mathbb{Z}_n[x] \) of the upper Hessenberg form can be efficiently computed from the following recurrence for \( p_k(x) \) using computations in \( \mathbb{Z}_n[x] \):
     \[
p_k(x) = \begin{cases} 
1 & k = 1 \\
x - m_{n,k}p_{k-1}(x) - \sum_{j=1}^{k-1} \prod_{i=1}^j m_{k-j+1,i}m_{j,k}p_{j-1}(x) & 1 < k \leq n + 1 
\end{cases}
\]

### Complexity

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

- Cost of reducing the \( n^2 \) entries in \( A \) modulo one prime is \( O(nm^2) \).
- 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 \( O(n(nm)^2) \).

In contrast, the Berkowitz algorithm, the algorithm that Maple uses, has complexity \( O(n^3(mn)^2) \), which reduces to \( O(n^3m^2) \) if the FFT is used.

### Timings

The following is a 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 \( Z \). The accelerated floating point versions using 25-bit primes generally give the best times.

<table>
<thead>
<tr>
<th>Versions</th>
<th>Xeon 2.8 GHz</th>
<th>Opteron 2.0 GHz</th>
<th>AXP280 2.08 GHz</th>
<th>Pentium M 2.00 GHz</th>
<th>Pentium 4 2.80 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>64int</td>
<td>100.7</td>
<td>107.4</td>
<td>~    ~</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>32int</td>
<td>66.3</td>
<td>73.0</td>
<td>76.8</td>
<td>35.6</td>
<td>57.4</td>
</tr>
<tr>
<td>new 32int</td>
<td>49.7</td>
<td>54.7</td>
<td>56.3</td>
<td>25.5</td>
<td>39.6</td>
</tr>
<tr>
<td>fmod</td>
<td>29.5</td>
<td>32.1</td>
<td>33.0</td>
<td>35.8</td>
<td>81.1</td>
</tr>
<tr>
<td>trunc</td>
<td>67.8</td>
<td>73.7</td>
<td>69.6</td>
<td>88.5</td>
<td>110.6</td>
</tr>
<tr>
<td>modtr</td>
<td>56.3</td>
<td>62.5</td>
<td>59.5</td>
<td>81.0</td>
<td>82.6</td>
</tr>
<tr>
<td>new fmod</td>
<td>11.0</td>
<td>11.6</td>
<td>14.5</td>
<td>15.2</td>
<td>28.8</td>
</tr>
<tr>
<td>ILA</td>
<td>17.6</td>
<td>19.9</td>
<td>21.9</td>
<td>26.2</td>
<td>27.3</td>
</tr>
<tr>
<td>Berkowitz</td>
<td>2035.6</td>
<td>2262.6</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
</tbody>
</table>

### 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 modulo \( p \) since 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 32bit version, 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 modulo \( p \), 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 \( b \rightarrow a/p \), we first compute \( c \rightarrow a - p \times \text{trunc}(a/p) \), then \( b \rightarrow c \) if \( c \neq \pm b \), \( b \rightarrow 0 \) 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 \( \pm b \) at the end. So to compute \( b \rightarrow a/p \), we actually compute \( c \rightarrow a - p \times \text{trunc}(a/p) \), which results in \( c \in [-p, p] \).
- **new fmod** An improved fmod version, where we have reduced the number of times fmod() is called. In other words, we reduce the result modulo \( p \) 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.
- **ILA** An improved trunc version using operation count acceleration. It is the default used in Maple’s LA:Modular routines.