Modular Techniques for Linear Algebra

AM 3038 / AM 3084 / AM 3036

Lasal Sandeepa Hettiarachchi
Department of Mathematics
University of Colombo
2019s17363@stu.cmb.ac.lk

Udani Anupama
Department of Mathematics
University of Colombo
2019s17616@stu.cmb.ac.lk

Sahashra Dinasiri
Department of Mathematics
University of Colombo
2019s17334@stu.cmb.ac.lk

ABSTRACT

In linear algebra, modular techniques are often used to solve problems involving large matrices. These techniques involve breaking down a matrix into smaller, more manageable submatrices, which can then be manipulated using various mathematical operations. One such technique is the use of "big prime" and "small primes" to compute the determinant of a matrix. This method involves dividing the matrix into submatrices and then multiplying the determinants of these submatrices together, using modular arithmetic to ensure that the result remains manageable. By using these modular techniques, complex problems in linear algebra can be solved more efficiently and with greater accuracy. In this report, the OneBigPrime method and SmallPrimes method will be discussed to solve for the determinant of a matrix. Also, the computational complexity of the approaches will be analyzed.

KEYWORDS

Matrix, Determinant, Linear Algebra, Modular Techniques, Chinese Remainder Theorem, Small Prime modular computation method

1 Introduction

Modular techniques in linear algebra involve the use of modular arithmetic to solve problems involving matrices and other mathematical objects. One common application of modular techniques in linear algebra is the calculation of determinants. A determinant is a value associated with a square matrix that encodes certain properties of the linear transformation represented by the matrix.

One approach to calculating determinants is to use the "big prime" method. This involves working modulo a large prime number, which allows us to reduce the calculations to a finite field and apply various algebraic manipulations to simplify the problem. For example, we can use row and column operations to put the matrix into reduced row echelon form, from which the determinant can be easily calculated.

In the following subsections, the theory behind the methods will be accessed.

1.1 Small Primes Method

For a given matrix(A), the Small Prime method can be used to find the determinant. First, the Hadamard's bound will be calculated. Then consecutive small primes (p_i) from 2 up to p_r, where the multiplication of them is greater than 2*(Hadamard's bound) +1 will be chosen. Next, the matrix A is converted into a modulo p_i matrix(A_{p_i}) for all i=1,..,r. Then find the determinants of all A_{p_i} s . From it, a system of equations will be generated. That system can be solved using Chinese Remainder Theorem. Then it will give a unique congruence class as the solution. The determinant of A will be in that congruence class. The element with minimum absolute value of the mentioned congruence class is the determinant of A.

1.2 Chinese remainder theorem

The Chinese Remainder Theorem is a mathematical theorem that gives the conditions necessary for multiple equations to have a simultaneous integer solution. If we know the remainders of the division of an integer by several integers, then we can determine uniquely the remainder of the division of this integer by the product of these integers under the condition that the devisors are pairwise coprime. In other words, we can explain it as following. Theorem says that, if we have a system of congruences (equations involving the modulo operation), then under certain conditions, we can find a solution to the system that is unique modulo the product of the moduli involved.

$x \equiv b_1 (\text{mod } p_1)$
$x \equiv b_2 (\text{mod } p_2)$
$x \equiv b_3 (\text{mod } p_3)$
$x \equiv b_4 (\text{mod } p_4)$
.
.
.
$x \equiv b_k (\text{mod } p_k)$

Let pi where I = {1, 2, 3, ..., k} $\in \mathbb{Z}$ be pairwise coprime [ie: gcd p_i, p_{i+n} = 1]
$B_i \in \mathbb{Z}$
Then the system has solutions as a unique congruence class.

$\{y \in \mathbb{Z} : y \equiv x \pmod{p_1 p_2 \ldots p_k}\}$
Given: $x \equiv a_i$ (mod mi) for i = 1,2, 3, …, r
(mi are pairwise relatively prime)
The solution set of congruences

$$a_1 b_1 \frac{M}{m_i} + \cdots \ldots + a_r b_r \frac{M}{m_r} (mod\ M)$$

1.3 One Big Prime method

For a given matrix(A), the One Big Prime method can be used to find the determinant. First, the Hadamard's bound will be calculated. Then a big prime number(P) which is greater than 2*(Hadamard's bound)+1 will be chosen. Next, the matrix A is converted into a modulo P matrix(A_P). Then find the determinant of A_P. Then the determinant of A will be in y congruence class where $\{y \in \mathbb{Z}: \det(A_P)\ (modP)\}$. The element with minimum absolute value of the mentioned congruence class is the determinant of A.

2 Methodology

In this section, the methodology pertaining to calculating the determinant of matrices using One Big prime method and small primes method will be discussed.

2.1 Algorithm to calculate the determinant using small primes method for an nxn matrix

```
INPUTS : Shape of the matrix , respective elements of the matrix
STEP 1 : Find the maximum absolute value of all entries.
        $B \leftarrow max|A_{ij}|\ for\ all\ i,j\ in\ shape\ of\ A$
STEP 2 : Calculate the Hadamard's bound (H).
        $H \leftarrow n^{n/2}B^n$
STEP 3 : Choose P such that,
        $P > 2H + 1$
STEP 4: Find small prime numbers $p_i$ where $p_1 * p_2 * p_3 \ldots * p_k \geq P$
STEP 5 : Convert all $a_{ij}$'s of the Matrix A into modulo matrices of the primes $p_m$ for all m = 1,2,..,k
        $A_{p_i} = (a_{ij}(mod\ p_i))_{nxn}$
STEP 6 : Find the determinants of all $A_{p_i}$'s.
(Make a system such that $\det(A) = \det(A_{p_i})\ mod\ p_i$
STEP 7 : Use Chinese Remainder Theorem to solve the equations of step  STEP 6.
STEP 8 : Find D.
        (Here D is the congruence class of $p_1 p_2 \ldots p_r$)
STEP 9 : Find the minimum absolute value ($|d_r|$) in D.
STEP 10 : the determinant of A is d such that
        d = $d_r$
OUTPUT : d
```

2.2 Chinese remainder theorem

```
INPUTS : $x \equiv a_i(mod m_i),\ for\ i = 1,2,..,r$
        ($m_i$' s are pairwise coprime.)
STEP 1 : Calculate M
        $M = m_1 m_2 \ldots m_r$
STEP 2 : Calculate $\frac{M}{m_i}$ for $i = 1,2,..,r$
STEP 3 : Determine $b_i$ where $b_i \frac{M}{m_i} \equiv 1(mod m_i)\ for\ all\ i = 1,\ldots,r$
STEP 4: Find the solution of set of congruence by,
        $x \equiv \sum_{i=1}^{r} \left(a_i b_i \frac{M}{m_i}\right) (mod\ M)$
OUTPUT : solution of set of congruence(x)
```

The small prime method code was implemented based on the above algorithms for solving equations.

```
import numpy as np
#method to take a matrix as user input
def store_matrix():
  #ask the user for the size of the matrix
  n = int(input('Enter the size of the matrix: '))
  #create an empty matrix
  A = np.zeros((n, n))
  #ask the user for the elements of the matrix
  for i in range(0, n):
    for j in range(0, n):
      A[i][j] = int(input('Enter the element at position ' + str(i) + ',' + str(j) + ': '))
  return A

# calculate the hadamard's upper bound for a determinant of a matrix
def hadamard_bound(A):
  #find the absolute value of the heighest value in the matrix
  max = np.amax(np.absolute(A))
  #find the degree of the matrix
  n = np.size(A, 0)
  H = n ** (n/2) * max ** n
  return int(H)

#create a function to check if a number is prime
def is_prime(p):
  for i in range(2, p):
    if p % i == 0:
      return False
  return True

#calculate all the prime numbers where the product of the primes is less than 2*H +
1
```

```
def find_primes(H):
    p = 2*H + 1
    primes = []
    for i in range(2, p):
        if is_prime(i):
            primes.append(i)
        if np.prod(primes) > p:
            return primes

#find the bezout coefficients
def bezout(a, b):
    if b == 0:
        return 1, 0
    else:
        x, y = bezout(b, a % b)
        return y, x - y * (a // b)

################################################################
A = store_matrix()
h = hadamard_bound(A)
print('hadamard_bound',h)
p = find_primes(h)
print('primes',p)

#for all elements in the array p, create matrices that are the same size as A and mod them with the prime number and assign it to an array
mod_matrices = []
for i in range(0, len(p)):
    mod_matrices.append(np.mod(A, p[i]))
print('The modulous matrices are:', mod_matrices)

#find the determinents of all the mod matrices
det_mod_matrices = []
for i in range(0, len(mod_matrices)):
    det_mod_matrices.append(round(np.linalg.det(mod_matrices[i])))
print('The determinents of modulous matrices are:', det_mod_matrices)


#implement the chinese remainder theorem
#find the product of all the primes
M = np.prod(p)
#find the product of all the primes divided by each prime
M_mi = []
for i in range(0, len(p)):
    M_mi.append(M / p[i])
print('M is:', M)
print('M_mi is:', M_mi)

#find the bezout coefficients of the product of all the primes divided by each prime and each prime
bezout_coeff = []
for i in range(0, len(p)):
    bezout_coeff.append(bezout(M_mi[i], p[i]))
print('bezout_coeff is:', bezout_coeff)

#find the sum of the determinent of the mod matrix times the bezout coefficient times the product of all the primes divided by each prime
sum = 0
for i in range(0, len(p)):
    sum += det_mod_matrices[i] * bezout_coeff[i][0] * M_mi[i]
print('sum is:', sum)

var = sum % M
#X = var (Mod M)
#find the array of X that satisfies x = var (Mod M)
# 210 x + 152

#write a switch case statement to find the smallest of 3 numbers


if abs(M * 1 + var) < abs(M * 0 + var) and abs(M * 1 + var) < abs(M * -1 + var):
    cal_det_A = M * 1 + var
elif abs(M * 0 + var) < abs(M * 1 + var) and abs(M * 0 + var) < abs(M * -1 + var):
    cal_det_A = M * 0 + var
else:
    cal_det_A = M * -1 + var



print('cal_det_A is:', cal_det_A)
```

Also the one big prime algorithm was implemented

```
import numpy as np

#method to take a matrix as user input
def store_matrix():
    #ask the user for the size of the matrix
    n = int(input('Enter the size of the matrix: '))
    #create an empty matrix
    A = np.zeros((n, n))
    #ask the user for the elements of the matrix
    for i in range(0, n):
        for j in range(0, n):
            A[i][j] = int(input('Enter the element at position ' + str(i) + ',' + str(j) + ': '))
    return A

# calculate the hadamard's upper bound for a determinant of a matrix
def hadamard_bound(A):
    #find the absolute value of the heighest value in the matrix
    max = np.amax(np.absolute(A))
    #find the degree of the matrix
    n = np.size(A, 0)
    H = n ** (n/2) * max ** n
    return int(H)

#create a function to check if a number is prime
def is_prime(p):
    for i in range(2, p):
        if p % i == 0:
            return False
    return True


#find a prime number that is greater than the two time hadamard's upper bound +1
def find_prime(H):
    #find the next prime number that is greater than 2*H
    p = 2*H + 1
    while True:
        if is_prime(p):
            return p
        p += 1

#take the element wise modulous of a matrix with a prime number
def mod_matrix(A, p):
    return np.mod(A, p)

#find the determinent of a matrix
def det(A):
    return round(np.linalg.det(A))
#find the bezout coefficients
def bezout(a, b):
    if b == 0:
        return 1, 0
    else:
        x, y = bezout(b, a % b)
        return y, x - y * (a // b)
```

```
# define a 2x2 matrix using np
A = store_matrix()
print('A =', A)
h = hadamard_bound(A)
print('hadamard_bound',h)
p = find_prime(h)
print('prime',p)
det_A_mod = det(mod_matrix(A,p))
print('det(A) mod p =', det_A_mod)
var = det_A_mod % p
print('var =', var)



if abs(p * 1 + var) < abs(p * 0 + var) and abs(p * 1 + var) < abs(p * -1 + var):
    cal_det_A = p * 1 + var
elif abs(p * 0 + var) < abs(p * 1 + var) and abs(p * 0 + var) < abs(p * -1 + var):
    cal_det_A = p * 0 + var
else:
    cal_det_A = p * -1 + var


print('cal_det_A is:', cal_det_A)
```

3 Analysis of the implementation

Compared to the BigPrime method, the computational complexity of the small prime method is low. For an nxn matrix,

Complexity of the OneBigPrime method: $O \sim (n^5)$
Complexity of the SmallPrimes method: $O \sim (n^4 + n^3)$

Therefore small prime method is much more efficient than onebig prime. But both methods are better than the traditional method. The following graph depicts how the computational time grows with respect to the number of elements in both methods.

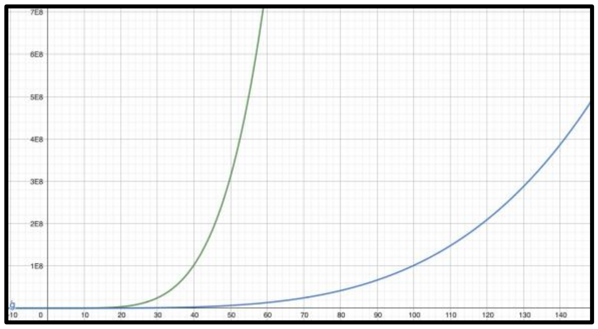
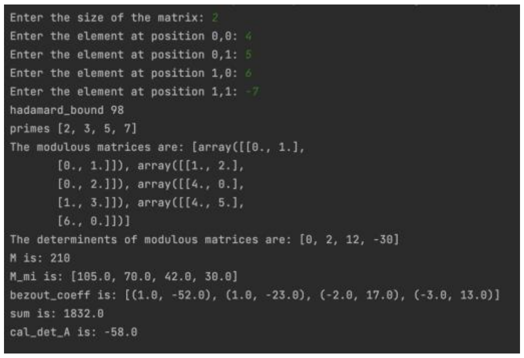


Figure 1: Figure of the growth in time complexities with n. The green graph is the growth of big prime and the blue graph is the growth of small prime method.

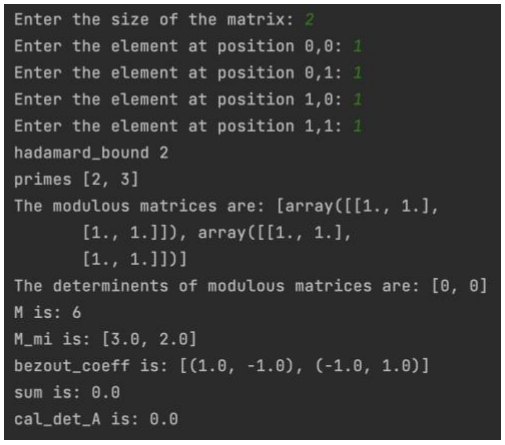
4 Results

The following are some results obtained using the code that was implemented in the methodology section.



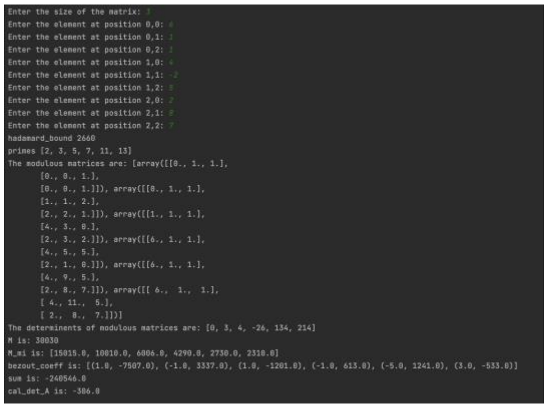
```
Enter the size of the matrix: 2
Enter the element at position 0,0: 4
Enter the element at position 0,1: 5
Enter the element at position 1,0: 6
Enter the element at position 1,1: -7
hadamard_bound 98
primes [2, 3, 5, 7]
The modulous matrices are: [array([[0., 1.],
       [0., 1.]]), array([[1., 2.],
       [0., 2.]]), array([[4., 0.],
       [1., 3.]]), array([[4., 5.],
       [6., 0.]])]
The determinents of modulous matrices are: [0, 2, 12, -30]
M is: 210
M_mi is: [105.0, 70.0, 42.0, 30.0]
bezout_coeff is: [(1.0, -52.0), (1.0, -23.0), (-2.0, 17.0), (-3.0, 13.0)]
sum is: 1832.0
cal_det_A is: -58.0
```

Figure 2: Calculation of the det value for 2x2 matrix using small primes method



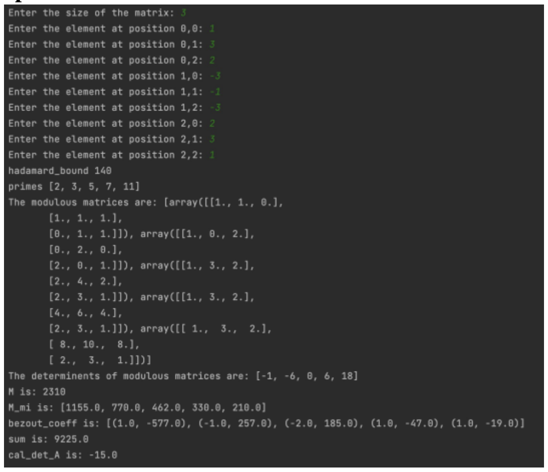
```
Enter the size of the matrix: 2
Enter the element at position 0,0: 1
Enter the element at position 0,1: 1
Enter the element at position 1,0: 1
Enter the element at position 1,1: 1
hadamard_bound 2
primes [2, 3]
The modulous matrices are: [array([[1., 1.],
       [1., 1.]]), array([[1., 1.],
       [1., 1.]])]
The determinents of modulous matrices are: [0, 0]
M is: 6
M_mi is: [3.0, 2.0]
bezout_coeff is: [(1.0, -1.0), (-1.0, 1.0)]
sum is: 0.0
cal_det_A is: 0.0
```

Figure 3: Calculation of the det value for 2x2 matrix using small primes method



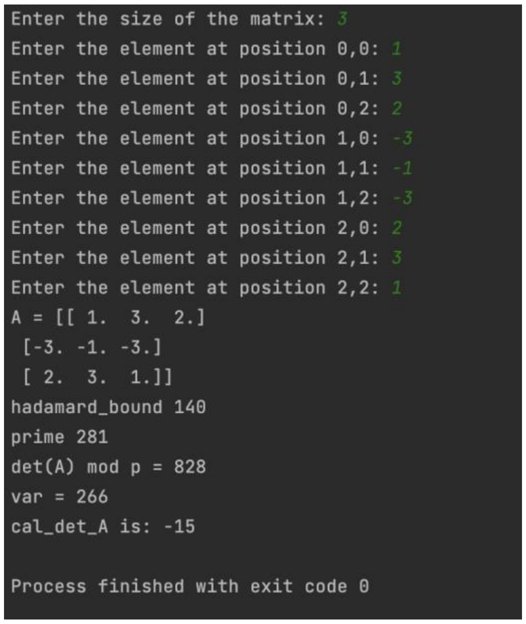
```
Enter the size of the matrix: 3
Enter the element at position 0,0: 6
Enter the element at position 0,1: 1
Enter the element at position 0,2: 1
Enter the element at position 1,0: 4
Enter the element at position 1,1: -2
Enter the element at position 1,2: 5
Enter the element at position 2,0: 2
Enter the element at position 2,1: 8
Enter the element at position 2,2: 7
hadamard_bound 2660
primes [2, 3, 5, 7, 11, 13]
The modulous matrices are: [array([[0., 1., 1.],
       [0., 0., 1.],
       [0., 0., 1.]]), array([[0., 1., 1.],
       [1., 1., 2.],
       [2., 2., 1.]]), array([[1., 1., 1.],
       [4., 3., 0.],
       [2., 3., 2.]]), array([[6., 1., 1.],
       [4., 5., 5.],
       [2., 1., 0.]]), array([[6., 1., 1.],
       [4., 9., 5.],
       [2., 8., 7.]]), array([[ 6.,  1.,  1.],
       [ 4., 11.,  5.],
       [ 2.,  8.,  7.]])]
The determinents of modulous matrices are: [0, 3, 4, -26, 134, 214]
M is: 30030
M_mi is: [15015.0, 10010.0, 6006.0, 4290.0, 2730.0, 2310.0]
bezout_coeff is: [(1.0, -7507.0), (-1.0, 3337.0), (1.0, -1201.0), (-1.0, 613.0), (-5.0, 1241.0), (3.0, -533.0)]
sum is: -240546.0
cal_det_A is: -306.0
```

Figure 4:Calculating the determinant of a 3x3 matrix using small primes method



```
Enter the size of the matrix: 3
Enter the element at position 0,0: 1
Enter the element at position 0,1: 3
Enter the element at position 0,2: 2
Enter the element at position 1,0: -3
Enter the element at position 1,1: -1
Enter the element at position 1,2: -3
Enter the element at position 2,0: 2
Enter the element at position 2,1: 3
Enter the element at position 2,2: 1
hadamard_bound 140
primes [2, 3, 5, 7, 11]
The modulous matrices are: [array([[1., 1., 0.],
       [1., 1., 1.],
       [0., 1., 1.]]), array([[1., 0., 2.],
       [0., 2., 0.],
       [2., 0., 1.]]), array([[1., 3., 2.],
       [2., 4., 2.],
       [2., 3., 1.]]), array([[1., 3., 2.],
       [4., 6., 4.],
       [2., 3., 1.]]), array([[ 1.,  3.,  2.],
       [ 8., 10.,  8.],
       [ 2.,  3.,  1.]])]
The determinents of modulous matrices are: [-1, -6, 0, 6, 18]
M is: 2310
M_mi is: [1155.0, 770.0, 462.0, 330.0, 210.0]
bezout_coeff is: [(1.0, -577.0), (-1.0, 257.0), (-2.0, 185.0), (1.0, -47.0), (1.0, -19.0)]
sum is: 9225.0
cal_det_A is: -15.0
```

Figure 5:Calculating the complexity of a 3x3 matrix using small primes method



```
Enter the size of the matrix: 3
Enter the element at position 0,0: 1
Enter the element at position 0,1: 3
Enter the element at position 0,2: 2
Enter the element at position 1,0: -3
Enter the element at position 1,1: -1
Enter the element at position 1,2: -3
Enter the element at position 2,0: 2
Enter the element at position 2,1: 3
Enter the element at position 2,2: 1
A = [[ 1.  3.  2.]
 [-3. -1. -3.]
 [ 2.  3.  1.]]
hadamard_bound 140
prime 281
det(A) mod p = 828
var = 266
cal_det_A is: -15

Process finished with exit code 0
```

Figure 6:Calculating the det of a 3x3 matrix using one big prime method

REFERENCES

[1] Eric Eric Weisstein. Chinese Remainder Theorem -- from Wolfram MathWorld. Retrieved December 13, 2022 from https://mathworld.wolfram.com/ChineseRemainderTheorem.html

[2] Ben Lynn. Retrieved from https://crypto.stanford.edu/pbc/notes/numbertheory/crt.html

[3] John D Dixon. “Exact solution of linear equations using P-adicexpansions”. In:Numerische Mathematik40.1 (1982), pp. 137–141

[4] Wissam Raji. Retrieved from https://math.libretexts.org/Bookshelves/Combinatorics_and_Discrete_Mathematics/Elementary_Number_Theory_(Raji)/03%3A_Congruences/3.04%3A_The_Chinese_Remainder_Theorem