SOLVING TOEPLITZ AND CIRCULANT SYSTEMS

Jordan Leiker | GTid: 903453031
Georgia Institute of Technology | MATH 6644

ABSTRACT

Toeplitz matrices are a common matrix structure that appear in many applications. Due to the high amount of structure in a Toeplitz matrix, i.e. constant in all diagonals, it is expected that an efficient solver for Toeplitz systems be possible.

This paper will focus on exploring methods to efficiently solve symmetric, positive definite Toeplitz systems. Specifically, this paper will compare several types of symmetric positive definite Toeplitz matrices and how they can be solved in O(Nlog(N)) operations using Conjugate Gradient and Pre-conditioned Conjugate Gradient with circulant matrices.

Index Terms**—** Iterative Methods, Toepltiz, Circulant, Preconditioned, Conjugate, Gradient

1. INTRODUCTION

The focus of this paper is to compare computational efficiency across several different Toeplitz matrices, pre-conditioners, iterative methods, and size of matrix. The two types of matrices, Toeplitz-A and Toeplitz-B, will be described in this section; followed by the two types of preconditioners, M-Strang and M-Chan; and finally a brief comment on the methods used, Conjugate Gradient (CG) and Pre-Conditioned Conjugate Gradient (PCG). Each combination of these matrices and methods will be evaluated across N = 50, 100, 200, 400, 800, 1600.

All computations for each combination of matrices and solvers will be terminated when:

$$\frac{\|\vec{r}_k\|}{\|\vec{r}_0\|} \leq 10^{-6} \qquad \text{or} \qquad \text{10,000 iterations reached}$$

1.1. Toeplitz Matrices

1.1.1. Toeplitz-A

The Toeplitz-A (TA) matrix is defined as:

$$a_k = |k+1|^{-p} \text{ for } p = 2, 1, 1/10, 1/100$$

Here, a_k describes the values of a column of the matrix TA, and each value in that column is constant down the diagonal of the matrix and then reflected across the diagonal to create a symmetric matrix. For example, the 4x4 matrix TA is created as (for p = 2):

$$a_k = [1 \;\; 0.25 \;\; 0.1111 \; 0.0625]^T$$

$$TA_{4x4} = \begin{bmatrix} 1 & 0.25 & 0.1111 & 0.0625 \\ 0.25 & 1 & 0.25 & 0.1111 \\ 0.1111 & 0.25 & 1 & 0.125 \\ 0.0625 & 0.1111 & 0.25 & 1 \end{bmatrix}$$

Note that this matrix, regardless of the value of p, is always symmetric (by definition) and positive definite. This matrix always has positive non-zero eigenvalues (which is the indicator of positive definiteness).

Proof of positive definiteness (for n = 2):

$$TA_{2x2} = \begin{bmatrix} |1|^{-p} & |2|^{-p} \\ |2|^{-p} & |1|^{-p} \end{bmatrix} \tag{1}$$

Characteristic Polynomial:

$$\lambda^2 - 2\lambda + \left(1 - \frac{1}{2^{2p}}\right) = 0 \tag{2}$$

Plugged into Quadratic Equation:

$$\frac{2 \pm \sqrt{2^2 - 4(1 - \frac{1}{2^{2p}})}}{2} = 0 \tag{3}$$

$$1 \pm \frac{1}{2^p} = 0 \tag{4}$$

Equation (4) results in always positive Characteristic Equation Roots (i.e. Eigenvalues) because $2^p > 1$ for all $p > 0$. Thus, TA is always positive definite (Note: that this proof can be extended to all n > 2).

1.1.2. Toeplitz-B

The Toeplitz-B (TB) matrix is defined as:

$a_k = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(\theta)e^{-ik\theta}d\theta$

where $f(\theta) = \theta^4 + 1$ for $-\pi \leq \theta \leq \pi$

Note that the matrix coefficients $a_k$are just the Fast Fourier Transform (FFT) of $f(\theta)$.

1.2. Preconditioners

1.2.1. M-Strang

The preconditioner M-Strang is defined as:

$[C_n]_{k,l} = c_{k-l}$ for $0 \leq k, l < n$

$$c_j = \begin{cases} a_j & 0 \leq j \leq [n/2] \\ a_{j-n} & [n/2] < j < n \\ c_{n+j} & 0 < -j < n \end{cases}$$

Where $[x]$ operator returns the closest integer (smaller) than x.

1.2.2. M-Chan

The preconditioner M-Chan is defined as:

$$c_j = \begin{cases} \dfrac{(n-j)a_j + ja_{j-n}}{n} & 0 \leq j < n \\ c_{n+j} & 0 \leq -j < n \end{cases}$$

1.3 Iterative Methods

Because the Toeplitz matrices are symmetric positive definite, the CG and PCG methods are used. To leverage the structure of the Toeplitz matrices and Circulant preconditioners, the CG and PCG algorithms will be slightly modified to use more efficient methods for matrix multiplication. These multiplication methods and modifications to the CG and PG routines are covered in the following section where the design and implementation decisions are described in detail.

2. APPROACH

This section will cover the design decisions at a theory level and at the implementation level. All code is written in Python 3 and leverage the NumPy and SciPy libraries heavily.

2.1. Circulant Matrix Multiply

2.1.1. Circulant Matrix Multiplication Theory

Circulant matrices can be diagonalized with the Fourier matrix, F_n. That is, for an $n\ x\ n$ circulant matrix C_n:

$C_n = F_n^*\Lambda_n F_n$

where $[F_n]_{j,k} = \frac{1}{\sqrt{n}}e^{2\pi ijk/n}$ for $0 \leq j, k \leq n-1$

and $\Lambda_n$is a diagonal matrix with eigenvalues of C_n

This means that any matrix-vector multiply with a circulant matrix can be computed as:

$$b = C_n x \tag{5}$$
$$b = (F_n^*\Lambda_n F_n)x \tag{6}$$
$$b = (F_n^*(\Lambda_n(F_n x))) \tag{7}$$
$$b = IFFT(\Lambda_n(FFT(x))) \tag{8}$$
$$b = IFFT(diag(\Lambda_n)(FFT(x))) \tag{9}$$

Note that the matrix multiplication, $\Lambda_n FFT(x)$, in (8) can be simplified to the vector multiplication in (9) because Λ_n is diagonal, i.e. this is an element-wise multiply of the eigenvalues of C_n and $FFT(x)$.

The diagonal of $\Lambda_n$can be quickly computed by taking the FFT of the first column of C_n. See proof below:

Note: The k^{th} eigenvector of C is

$$x^{(k)} = \frac{1}{\sqrt{n}}\left[e^{\frac{2\pi k}{n}i} \quad e^{\frac{2\pi 2k}{n}i} \quad \cdots \quad e^{\frac{2\pi(n-1)k}{n}i}\right]^T$$

Now consider the matrix-eigenvector multiplication, $y = Cx^{(k)}$, where the l-th component of y is:

$$y_l = \sum_{j=0}^{n-1} c_{j-l}e^{\frac{2\pi jk}{n}i} = e^{\frac{2\pi lk}{n}i}\sum_{j=0}^{n-1} c_{j-l}e^{\frac{2\pi(j-l)k}{n}i} \tag{10}$$

Evaluating this for any l we end up with the same sum, but re-arranged (circular shift of circulant matrices). Thus the sum is independent of l and there is no difference for $j \rightarrow j-l$, i.e.:

$$x^{(k)} = e^{\frac{2\pi lk}{n}i} \tag{11}$$

Knowing this we can now consider the k^{th} eigenvector/eigenvalue:

$$Cx^{(k)} = \lambda_k x^{(k)} \tag{12}$$

We can now derive the k^{th} eigenvalue, λ_k, by combining equations (10), (11), and (12):

$$\lambda_k = \sum_{j=0}^{n-1} c_j e^{\frac{2\pi jk}{n} i} \tag{13}$$

Now, defining a vector of eigenvalues, $\hat{c} = (\lambda_0, \lambda_1, \cdots, \lambda_{n-1})$, and using (13) we write:

$$\hat{c}_k = \sum_{j=0}^{n-1} c_{j0} e^{\frac{2\pi jk}{n} i} \tag{14}$$

$$\hat{c} = F c_0 \tag{15}$$

Note here that c_0 is the 0th column of C. This is telling us that the 0th column of C multiplied by the Fourier matrix gives us the eigenvalues. This multiplication by the Fourier matrix is equivalent to computing an FFT of the 0th column of C.

2.1.2. Circulant Matrix Multiplication Function

The implementation of the circulant multiply in Python is a minimal approach of the circulant multiply described in Section 2.1.1. Because the circulant multiply will be used inside the CG and PCG loops, removing unnecessary computations was the goal.

This "minimal" circulant multiply accepts just a vector containing the circulant matrix eigenvalues, L, and the vector, to be multiplied. By accepting the eigenvalues instead of the circulant matrix itself an entire FFT calculation was removed from the circulant multiply function.

The function code is shown below. This code exactly replicates (9). The `np.ravel` and `np.atleast_2d` function calls are used to handle Python array formatting.

```
def circMatMulMin(L,x):
  return np.atleast_2d(
         np.fft.ifft(
         L*np.fft.fft(np.ravel(x)))).T
```

2.2. Toeplitz Matrices

Toeplitz matrices are closely related to circulant matrices. In fact, a Toeplitz matrix can be converted to a circulant matrix of twice the size. By converting a Toeplitz matrix to circulant, a matrix multiply can be computed in $Nlog(N)$ computations using the circulant multiply methods described in Section 2.1.1.

2.2.1. Toeplitz to Circulant Matrices

Recall the definition of a Toeplitz matrix:

$$A_n = \begin{bmatrix} a_0 & a_{-1} & \cdots & a_{2-n} & a_{1-n} \\ a_1 & a_0 & a_{-1} & & a_{2-n} \\ \vdots & a_1 & a_0 & \ddots & \vdots \\ a_{n-2} & & \ddots & \ddots & a_{-1} \\ a_{n-1} & a_{n-2} & \cdots & a_1 & a_0 \end{bmatrix}$$

That is, a Toeplitz matrix is defined entirely by its first column and first row because it is diagonally constant. To convert an $n \times n$ Toeplitz matrix into a circulant matrix, we double the size to $2n \times 2n$ and define the first column as:

$$A_{2n,col0} = [a_0\ a_1 \cdots a_{n-2}\ a_{n-1}\ 0\ a_{1-n}\ a_{2-n} \cdots a_{-1}]^T$$

That is, the first column of the new circulant matrix is the first column of the Toeplitz matrix with a zero appended and then the first row (minus the first element) in reverse order appended after the zero.

A circulant matrix is completely defined by it's first column, so this is all we need to generate the new circulant matrix from a Toeplitz matrix.

Note that using this algorithm to create a circulant matrix from a Toeplitz matrix, the original Toeplitz matrix is embedded in the top left $n \times n$ block of the new circulant matrix.

The code implementing the Toeplitz to circulant matrix is:

```
def toep2Circ(T):
    # Create a col
    a = T[:,0]
    b = np.zeros(1)
    c = np.flip(T[0,1:T.shape[1]])
    col = np.concatenate((a,b,c))
    # Create a row
    a = T[0,:]
    b = np.zeros(1)
    c = np.flip(T[1:T.shape[0],0])
    row = np.concatenate((a,b,c))
    return sp.toeplitz(col,row)
```

2.2.2. Toeplitz Matrix Multiplies

A Toeplitz matrix multiply, $b_n = A_n x_n$, can be computed efficiently by using the Toeplitz-to-Circulant algorithm detailed in Section 2.2.1, padding the vector x with n zeros, and then use the circulant matrix multiply method described in Section 2.1.1. Finally the answer can be extracted from the result by taking just the first n values of the product vector.

An example of this Toeplitz multiply method implemented in Python (calculating the initial residual in the CG method), and using the `circMatMulMin()` function described in Section 2.1.2 is shown:

```
# Calculate Initial Residual / Direction
x_k = np.zeros((n, 1))
x_k2 = np.concatenate((np.ravel(x_k),
       np.zeros(x_k.shape[0])))
te = np.ravel(util.circMatMulMin(L2, x_k2))
r_k = b - np.atleast_2d(te[0:x_k.shape[0]]).T
```

This code (1) concatenates the zero padding on to x_k (2) calls the “minimal” circulant multiply routine, and (3) extracts just the first n values.

2.3. Iterative Methods

The only changes to the CG and PCG algorithms from their standard implementations are:

1. The $n \times n$ Toeplitz matrix is immediately converted to $2n \times 2n$ circulant.
2. The circulant matrices, i.e. the converted Toeplitz and circulant preconditioner in the case of the PCG algorithm, are immediately transformed via FFT to get the eigenvalues.
3. All matrix-vector multiplies are done via FFT using the circulant multiply method (and in the case of the converted Toeplitz matrix, the vector is also zero-padded to size $2n$).

These “efficient” CG and PCG algorithms use only circulant multiplies and as a result should scale at $Nlog(N)$ computations.

2.3.1. Conjugate Gradient Code

```
def cg_Toep_FFTmin(A,b,tol,maxIters):
  (m, n) = np.shape(A)

  # Concert nxn Toeplitz to a 2nx2n Circlant matrix
    (so we can use FFTs instead of matmuls)
  A2 = util.toep2Circ(A)

  # Diagonalize (via FFT because it is circulant)
    the main Matrix, A2
  L2 = np.fft.fft(A2[:, 0])

  # Calculate Initial Residual / Direction
  x_k = np.zeros((n, 1))
  x_k2 = np.concatenate((np.ravel(x_k),
         np.zeros(x_k.shape[0])))
  te = np.ravel(util.circMatMulMin(L2, x_k2))
  r_k = b - np.atleast_2d(te[0:x_k.shape[0]]).T
  t_km2 = 0

  # Initialize Loop exit conditions
  err = np.linalg.norm(r_k,2)/np.linalg.norm(b,2)
  numIters = 0

  while (err > tol) and (numIters < maxIters):
      t_km1 = r_k.T @ r_k
      if numIters == 0:
          p_k = r_k
      else:
          B_k = (t_km1)/(t_km2)
          p_k = r_k + B_k*p_k
      p_k2 = np.concatenate((np.ravel(p_k),
             np.zeros(p_k.shape[0])))
      te = np.ravel(util.circMatMulMin(L2, p_k2))
      w_k = np.atleast_2d(te[0:x_k.shape[0]]).T
      a_k = t_km1/(p_k.T @ w_k)
      x_k = x_k + a_k * p_k
      r_k = r_k - a_k * w_k
      t_km2 = t_km1

      # Stopping Condition Checks
      # Note: r_0=b, because r_0=b-Ax_0 where x_0=0
      err=np.linalg.norm(r_k,2)/np.linalg.norm(b,2)
      numIters += 1

  return x_k, numIters
```

2.3.2. Preconditioned Conjugate Gradient Code

```
def preCG_ToepCirc_FFTmin(A,M,b,tol,maxIters):
  (m, n) = np.shape(A)

  # Convert nxn Toeplitz to a 2nx2n Circlant
    matrix (so we can use FFTs instead of
    matmuls)
  A2 = util.toep2Circ(A)

  # Diagonalize (via FFT because they are
    circulant) the main Matrix, A2, and the
    preconditioner, M
  L2 = np.fft.fft(A2[:, 0])
  LM = np.fft.fft(M[:, 0])

  # Calculate Initial Residual / Direction
  x_k = np.zeros((n, 1))
  x_k2 = np.concatenate((np.ravel(x_k),
         np.zeros(x_k.shape[0])))
  te = np.ravel(util.circMatMulMin(L2, x_k2))
  r_k = b - np.atleast_2d(te[0:x_k.shape[0]]).T
  t_km2 = 0

  # Initialize Loop exit conditions
  err = np.linalg.norm(r_k,2)/np.linalg.norm(b, 2)
  numIters = 0

  while (err > tol) and (numIters < maxIters):
      z_k = util.circMatMulMin(LM,r_k)
      t_km1 = z_k.T @ r_k
      if numIters == 0:
          p_k = z_k
      else:
        B_k = (t_km1)/(t_km2)
        p_k = z_k + B_k*p_k
    p_k2 = np.concatenate((np.ravel(p_k),
           np.zeros(p_k.shape[0])))
    te = np.ravel(util.circMatMulMin(L2, p_k2))
    w_k = np.atleast_2d(te[0:x_k.shape[0]]).T
    a_k = t_km1/(p_k.T @ w_k)
    x_k = x_k + a_k * p_k
    r_k = r_k - a_k * w_k
    t_km2 = t_km1

    # Stopping Condition Checks
    # Note: r_0=b, because r_0=b-Ax_0 where x_0=0
    err=np.linalg.norm(r_k,2)/np.linalg.norm(b,2)
    numIters += 1

return x_k, numIters
```

2.3. General Functions

This section details the general-purpose functions that are used to set up the project environment based on the TA, TB, M-Strang, and M-Chan definitions from Sections 1.1 and 1.2.

2.3.1. Toeplitz-A Generation

```
def genToeplitzA(n, p):
    col = np.zeros(n)
    for k in range(n):
        col[k] = (np.abs(k+1))**(np.float(-p))
    return sp.toeplitz(col,col[0:])
```

2.3.2. Toeplitz-B Generation

```
def genToeplitzB(n):
    theta = np.linspace(-np.pi,np.pi,1000)
    f = (theta**4 + 1)
    col = np.fft.fft(f,n)
    return sp.toeplitz(col,col[0:])
```

2.3.3. Strang Preconditioner Generation

```
def genStrangPreconditioner(A):
    M = np.zeros(A.shape)
    M = M.astype(np.complex)
    n = M.shape[0]
    n2 = np.round(n/2)
    for l in range(n):
      for k in range(n):
        j = (k-l)
        if (j >= 0) and (j <= n2):
            M[k,l] = A[j,0]
        elif (j > n2) and (j < n):
            M[k,l] = A[0,n-j]
        else:
            M[k,l] = M[n+j,0]
    return M
```

2.3.4. Chan Preconditioner Generation

```
def genChanPreconditioner(A):
    M = np.zeros(A.shape)
    M = M.astype(np.complex)
    n = M.shape[0]
    for l in range(n):
      for k in range(n):
        j = (k - l)
        if (j == 0):
           M[k,l] = ((n-j)*A[j, 0])/n
        elif (j > 0) and (j < n):
           M[k,l] = ((n-j)*A[j,0] + j*A[0,n-j])/n
        else:
           M[k,l] = M[n+j,0]
    return M
```

3. RESULTS

This section compares the results across all CG/PCG runs using each of the Toeplitz matrices, preconditioners, and matrix sizes.

For all runs, the CG and PCG methods are solving $Ax = b$ where $b = [1\ 1\ \cdots 1]^T$. For each matrix (selected by the user via one variable), the range of matrix sizes, N, are iterated across. For each N the efficient CG, PCG with Strang, and PCG with Chan are run as well as standard CG and PCG (where “standard” CG/PCG implies matrix-vector multiply based routines). Each time a CG or PCG is run the number of iterations and computation time are logged.

When all computations are done, the efficient CG/PCG computation time vs. matrix size N is plotted on top of plot lines for $Nlog(N)$ and N^2. Additionally, the “standard” CG/PCG computation time vs. matrix size is plotted for comparison. Note that in all plots:

- Green: Efficient CG/PCG (FFT multiplies)
- Blue: Standard CG/PG (full matrix multiplies)
- Red: $Nlog(N)$
- Purple: N^2

All plots are CG, PCG-Strang, PCG-Chan from left to right.

3.1. Toeplitz-A (p=2)

Table 1. Toeplitz A (p=2) Number of Iterations per Method

N	**CG: Number of Iterations**	**PCG w/ Strang: Number of Iterations**	**PCG w/ Chan: Number of Iterations:**
50	9	13	13
100	10	16	16
200	10	17	17
400	10	18	18
800	10	17	17
1600	9	17	17

Table 2. Toeplitz A (p=2) Computation Time per Method

N	CG: Run-Time	PCG w/ Strang: Run-Time	PCG w/ Chan: Run-Time
50	0.001997	0.001995	0.001995
100	0.002979	0.003989	0.002991
200	0.002979	0.004949	0.005021
400	0.004987	0.009014	0.008960
800	0.017952	0.020944	0.020944
1600	0.063818	0.068818	0.069814

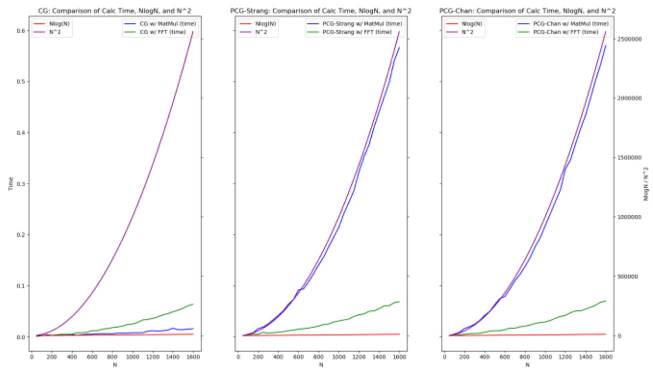

Figure 1. Toeplitz A (p=2): Compute time vs. N (efficient CG, PCG w/ Strang, PCG w/ Chen) compared to N^2, NlogN, and compute time vs N. (standard CG, PCG w/ Strang, PCG w/ Chen)

3.2. Toeplitz-A (p=1)

Table 3. Toeplitz A (p=1) Number of Iterations per Method

N	CG: Number of Iterations	PCG w/ Strang: Number of Iterations	PCG w/ Chan: Number of Iterations:
50	12	20	19
100	14	29	29
200	17	41	42
400	19	54	55
800	22	68	67
1600	23	81	82

Table 4. Toeplitz A (p=1) Computation Time per Method

N	CG: Run-Time	PCG w/ Strang: Run-Time	PCG w/ Chan: Run-Time
50	0.001996	0.004984	0.003989
100	0.002992	0.005984	0.006013
200	0.006981	0.012966	0.010003
400	0.007977	0.017951	0.019945
800	0.021939	0.042885	0.041888
1600	0.067820	0.109707	0.109706

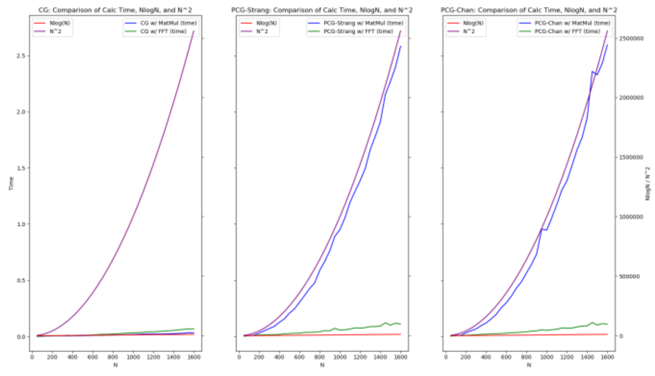

Figure 2. Toeplitz A (p=1): Compute time vs. N (efficient CG, PCG w/ Strang, PCG w/ Chen) compared to N^2, NlogN, and compute time vs N. (standard CG, PCG w/ Strang, PCG w/ Chen)

3.3. Toeplitz-A (p=1/10)

Table 5. Toeplitz A (p=1/10) Number of Iterations per Method

N	CG: Number of Iterations	PCG w/ Strang: Number of Iterations	PCG w/ Chan: Number of Iterations:
50	15	30	29
100	19	55	54
200	25	100	104
400	34	193	186
800	43	356	361
1600	58	624	619

Table 6. Toeplitz A (p=1/10) Computation Time per Method

N	CG: Run-Time	PCG w/ Strang: Run-Time	PCG w/ Chan: Run-Time
50	0.002962	0.005984	0.004985
100	0.003016	0.010970	0.010971
200	0.006008	0.022939	0.023972
400	0.012934	0.059840	0.052895
800	0.028922	0.162568	0.162566
1600	0.091755	0.479716	0.468746

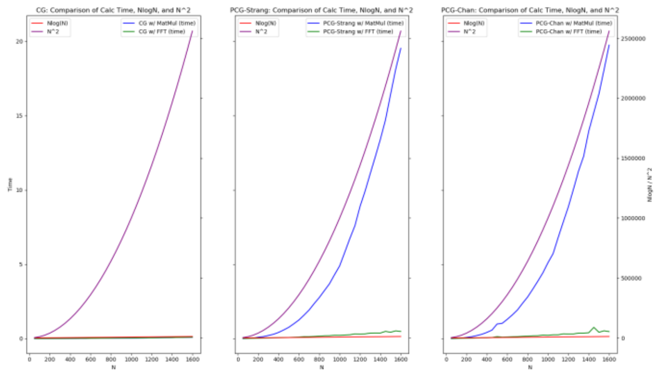

Figure 3. Toeplitz A (p=1/10): Compute time vs. N (efficient CG, PCG w/ Strang, PCG w/ Chen) compared to N^2, NlogN, and compute time vs N. (standard CG, PCG w/ Strang, PCG w/ Chen)

3.4. Toeplitz-A (p=1/100)

Table 7. Toeplitz A (p=1/100) Number of Iterations per Method

N	CG: Number of Iterations	PCG w/ Strang: Number of Iterations	PCG w/ Chan: Number of Iterations:
50	12	29	26
100	17	45	45
200	21	84	86
400	26	140	142
800	35	243	248
1600	41	405	420

Table 8. Toeplitz A (p=1/100) Computation Time per Method

N	CG: Run-Time	PCG w/ Strang: Run-Time	PCG w/ Chan: Run-Time
50	0.001992	0.004991	0.005008
100	0.002994	0.012968	0.009974
200	0.005983	0.019946	0.019947
400	0.009973	0.042884	0.043882
800	0.027923	0.111701	0.114695
1600	0.077791	0.327128	0.340117

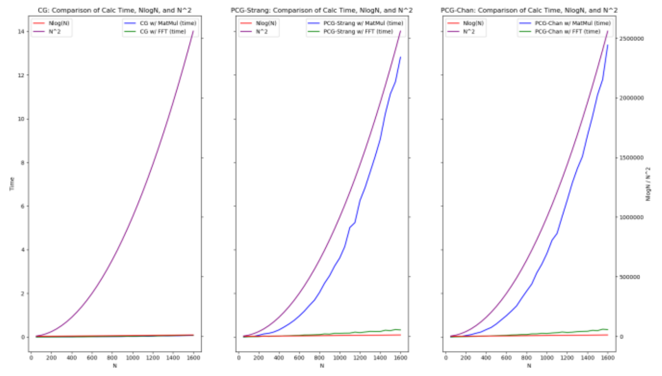

Figure 4. Toeplitz A (p=1/100): Compute time vs. N (efficient CG, PCG w/ Strang, PCG w/ Chen) compared to N^2, NlogN, and compute time vs N. (standard CG, PCG w/ Strang, PCG w/ Chen)

3.5. Toeplitz-B

Table 9. Toeplitz B Number of Iterations per Method

N	CG: Number of Iterations	PCG w/ Strang: Number of Iterations	PCG w/ Chan: Number of Iterations:
50	6	8	7
100	9	13	13
200	14	39	40
400	34	10000	10000
800	10000	10000	10000

Table 10. Toeplitz B Computation Time per Method

N	CG: Run-Time	PCG w/ Strang: Run-Time	PCG w/ Chan: Run-Time
50	0.001000	0.000993	0.001995
100	0.002025	0.002991	0.002992
200	0.005017	0.010994	0.010950
400	0.013962	2.840446	2.815979
800	3.491882	4.166481	4.177339

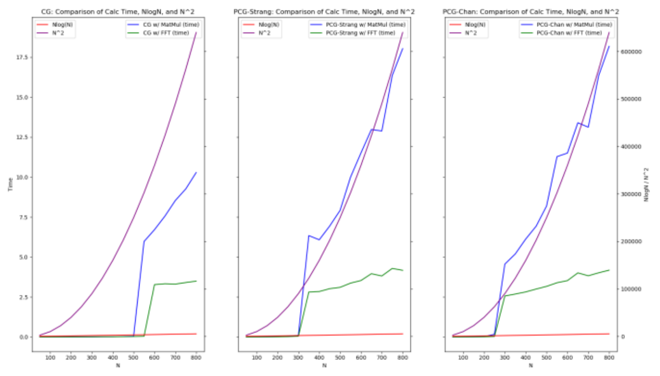

Figure 5. Toeplitz B: Compute time vs. N (efficient CG, PCG w/ Strang, PCG w/ Chen) compared to N^2, NlogN, and compute time vs N. (standard CG, PCG w/ Strang, PCG w/ Chen)

4. ANALYSIS

4.1. Computational Efficiency

It is immediately clear from Figures 1 through 4 that the "efficient" CG and PCG algorithms (green line) theorized and implemented in this paper are performing at or close to $Nlog(N)$ (red line).

The "worst" performance (i.e. largest deviation from $Nlog(N)$) was for Toeplitz-A (p=2), but even in this case the computation time was a fraction of N^2 computation time. Toeplitz-A seemed to converge closer to $Nlog(N)$ as the parameter p was decreased. As for Toeplitz-B, the performance was incredibly close to $Nlog(N)$ until the solvers reached a matrix size that did not converge.

It is worth noting that even when random b vectors were chosen (from $b = Ax$) instead of the ones-vector these same performance trends were observed; thus, these performance characteristics are not just because I happened to pick a convenient b.

4.2. Pre-conditioner Effect

The pre-conditioners did not seem to have any appreciable effect on reducing the number of iterations or computation time. In fact, in all cases using the pre-conditioners increased both number of iterations and computation time.

At this time it is unclear whether the Strang and Chan pre-conditioners were calculated incorrectly, or whether there is an error in the PCG method. The likelihood of an error in the PCG method is slim, because using the identity matrix as a preconditioner returns identical results as the CG method; thus indicating it is likely correct.

It is also likely that these matrices are not particularly receptive to using pre-conditioners. This papers goal was to prove that it is possible to use PCG with $Nlog(N)$ computations; it was not particularly concerned with whether the Strang and Chan circulant pre-conditioners were necessarily the best for the Toeplitz-A and Toeplitz-B matrices.

4.3 Unknowns

The unknown encountered (minus the unsatisfying results of the preconditioners with PCG) was that using CG with full matrix-multiplications seemed to result in $Nlog(N)$ computations instead of N^2. PCG did not see this affect and did indeed follow N^2.

It is unclear as to whether there is an optimization going on within the NumPy library that could utilize matrix structure to quickly compute matrix multiplies. At this time no answer was found within the NumPy documentation that seemed to indicate anything other than standard matrix multiplies occur.

Given that this was out of scope of the project, no major investigation into this was undertaken.

5. CONCLUSION

This paper demonstrates that it is possible to perform both CG and PCG in $Nlog(N)$ computations when using Toeplitz matrices and, in the case of PCG, circulant pre-conditioners. These efficient versions of the CG and PG algorithms leverage the unique attributes of the circulant matrix by computing matrix-multiplies using FFT's and the structure of Toepltiz matrices by realizing that a Toeplitz matrix can be embedded within a circulant matrix to further leverage efficient FFT-based matrix multiplies.