Indian Institute of Technology Kanpur

MTH 696A Project Report

Guided by- Prof. Pravir Dutt

Chebyshev Differentiation using Fast Fourier Transform

Submitted by:

Arjun Shukla	171031
Mayank Singour	14817378
Shivansh Rai	14658

April 28, 2019

Contents

Abstract

In this paper, we shall use Chebyshev points to construct Chebyshev differentiation matrices and apply these matrices to differentiate a few functions. And, we will see how Chebyshev spectral differentiation methods can be implemented by the Fast Fourier Transform(FFT), which provides a crucial speedup for some calculations. We will then compare the results obtained from these two methods.

1 Introduction to the Problem

Given a set of grid points $\{x_j\}$ and corresponding function values $\{u(x_j)\}$, we want to approximate the derivative of u using this data. The most basic and direct method is some kind of finite difference formula. To formulate the problem consider a uniform grid $\{x_1, \dots x_N\}$, with $x_{j+1} - x_j$ = h for each j, and a set of corresponding data values $\{u_1, \dots u_N\}$. Let w_j denote the approximation to $u'(x_j)$, the derivative of u at x_j . The standard second-order finite difference approximation is :

$$w_j = \frac{u_{j+1} - u_{j-1}}{2h} \tag{1}$$

For simplicity assume the periodicity of function for now, just to represent the differentiation matrix. We can now represent the discrete differentiation as a matrix vector multiplication :

$$w = h^{-1} D_N u \tag{2}$$

where, w is the vector of derivative, u is the vector of function values and D_N is the following matrix:

$$\begin{bmatrix} 0 & 1/2 & 0 & \dots & 0 & -1/2 \\ -1/2 & 0 & 1/2 & \dots & 0 & 0 \\ 0 & -1/2 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & 1/2 \\ 1/2 & 0 & 0 & \dots & -1/2 & 0 \end{bmatrix}$$

All the other entries in coefficient matrix are zero. This method has order of accuracy as two. We can create differentiation matrix of order 4 using taylor series expansion.

Chebyshev Points :

Now, we consider taking chebyshev grid instead of uniform grid. With equally spaced nodes, Runge's phenomenon comes into account. Thus for continuous functions it may happen that, on using equally spaced nodes, as N (the number of nodes) increases the

interpolation suffers from wild oscillation near the endpoints. With Chebyshev nodes, this doesn't happen, in fact if our function f(x) is absolutely continuous the interpolations converge uniformly to f(x) as N$\longrightarrow\infty$. So, for an interpolation of derivative using a large N, Chebyshev is a better choice.

Fourier Transform:
Major work has been done on Fourier Transforms in the past. We can do fourier transform on chebyshev points and then use differentiation matrix to speed up the computation. Since we have a sequence of input $\{x_1, \ldots x_N\}$ as discrete values, we use Discrete Fourier Transform(DFT). Evaluating Discrete Fourier Transform directly requires $\mathcal{O}(n^2)$ operations: we will discuss about time complexity of calculating fourier transforms in section 4 in detail. Fast Fourier Transform (**FFT**) is any method to compute the same results in $\mathcal{O}(n \log n)$ operations. All known FFT algorithms require $\mathcal{O}(n \log n)$ operations, although there is no known proof that a lower complexity is impossible.

Now, in the next three sections we will study each of the method for finding derivative of a function in detail. Simulations and observations for each method is included their respective section.

2 Finite Difference Method

The simplest method is to use finite difference approximations. A simple two-point estimation is to compute the slope of a nearby secant line through the points (x, f(x)) and (x + h, f(x + h)) Choosing a small number h, h represents a small change in x, and it can be either positive or negative. The slope of this line is :

$$\frac{f(x+h) - f(x)}{h} \tag{3}$$

This expression is also known as a first-order divided difference. As h approaches zero, the slope of the secant line approaches the slope of the tangent line. Therefore,

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{4}$$

Another two-point formula is to compute the slope of a nearby secant line through the points (x - h, f(x − h)) and (x + h, f(x + h)). The slope of this line is

$$\frac{f(x+h) - f(x-h)}{2h} \tag{5}$$

This formula is known as the symmetric difference quotient. In this case the first-order errors cancel, so the slope of these secant lines differ from the slope of the tangent line by

an amount that is approximately proportional to h^2. Hence for small values of h this is a more accurate approximation to the tangent line than the one-sided estimation.
The estimation error is given by

$$R = \frac{-f^{(3)}(c)}{6}h^2 \tag{6}$$

where c is some point between $x - h$ and $x + h$.
An important consideration in practice when the function is calculated using floating-point arithmetic is how small a value of h to choose. If chosen too small, the subtraction will yield a large *rounding error*. In fact, all the finite-difference formulae are ill-conditioned and due to cancellation will produce a value of zero if h is small enough. If too large, the calculation of the slope of the secant line will be more accurately calculated, but the estimate of the slope of the tangent by using the secant could be worse.
A formula for h that balances the rounding error against the secant error for optimum accuracy is

$$h = 2\sqrt{\varepsilon \left| \frac{f(x)}{f''(x)} \right|} \tag{7}$$

Higher-order methods for approximating the derivative exist. Following is the five-point method for the first derivative:

$$f'(x) = \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h} + \frac{h^4}{30}f^{(5)}(c) \tag{8}$$

where $c \in [x - 2h, x + 2h]$.

3 Chebyshev Differentiation Matrices

We discussed in the first section why grid points must cluster at boundaries. Chebyshev points are :

$$x_j = \cos(j\pi/N), \quad j = 0, 1, \ldots, N \tag{9}$$

We now use these points to construct Chebyshev differentiation matrices and apply these matrices to differentiate a few functions. To construct the Chebyshev Differentiation matrix, first we use the Lagrangian interpolation of function using the given set of function values on set of input points $\{x_N\}$. Let 'p' be polynomial of degree $\leq$ N with $p(x_j) = v_j$, $0 \leq$ j $\leq$ N. Hence, $w_j = p'(x_j)$. This operation is linear in $\mathbf{v}$, so it can be represented by multiplication by an $(N + 1)$ x $(N + 1)$ matrix, which we denote by D_N:

$$w = D_N v \tag{10}$$

For each N ≥ 1, let the rows and columns of the $(N + 1\text{x}(N + 1)$ Chebyshev spectral differentiation matrix D_N be indexed from 0 to N. The entries of this matrix are:

$$
\begin{aligned}
(D_N)_{00} &= \tfrac{2N^2+1}{6}, \quad (D_N)_{NN} = -\tfrac{2N^2+1}{6} \\
(D_N)_{jj} &= \tfrac{-x_j}{2\left(1-x_j^2\right)}, \quad j = 1, \ldots, N-1
\end{aligned}
\tag{11}
$$

$$
(D_N)_{ij} = \tfrac{c_i}{c_j}\tfrac{(-1)^{i+j}}{(x_i - x_j)}, \quad i \neq j, \quad i, j = 1, \ldots, N-1
$$

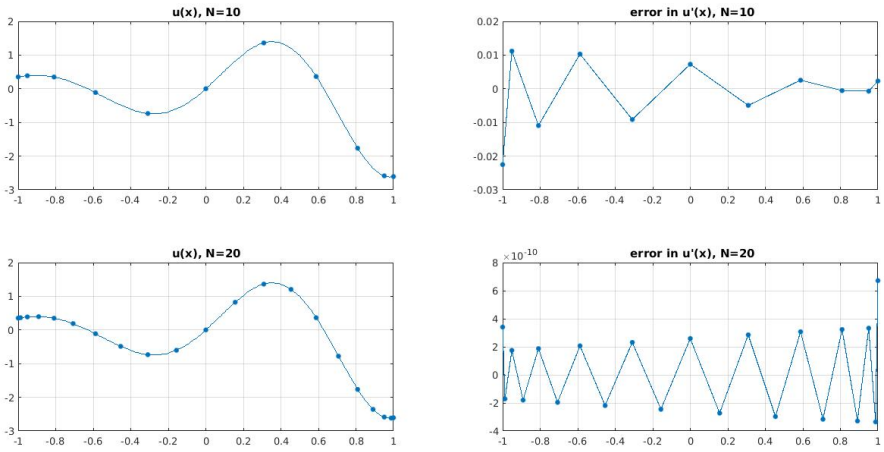
$$
\text{where } c_i = \begin{cases} 2 & i = 0 \text{ or } N \\ 1 & \text{otherwise} \end{cases}
\tag{12}
$$

3.1 Implementation

```
% p11.m - Chebyshev differentation of a smooth function

xx = -1:.01:1;
uu = exp(xx).*sin(5*xx); clf

for N = [10 20]
    [D,x] = cheb(N);
    u = exp(x).*sin(5*x);
    subplot('position',[.15 .66-.4*(N==20) .31 .28])
    plot(x,u,'.','markersize',14), grid on
    line(xx,uu,'linewidth',.8)
    title(['u(x), N=' int2str(N)])
    error = D*u - exp(x).*(sin(5*x)+5*cos(5*x));
    subplot('position',[.55 .66-.4*(N==20) .31 .28])
    plot(x,error,'.','markersize',14), grid on
    line(x,error,'linewidth',.8)
    title([' error in u''(x), N=' int2str(N)])
end
```

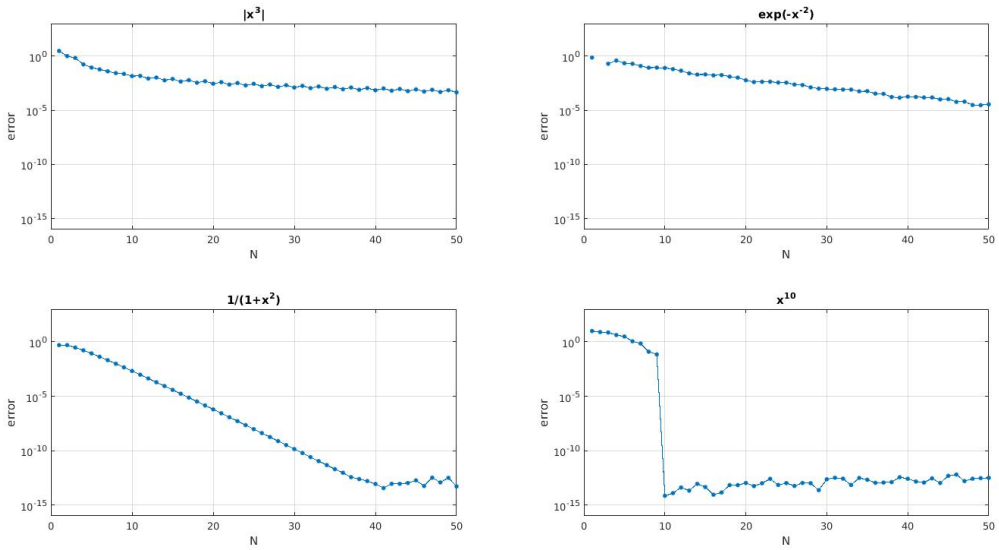


```
% p12.m - accuracy of Chebyshev spectral differentiation
%    (compare p7.m)
% Compute derivatives for various values of N:

tic;
Nmax = 500; E = zeros(3,Nmax);
for N = 1:Nmax;
    [D,x] = cheb(N);
    v = abs(x).^3; vprime = 3*x.*abs(x);
    % 3rd deriv in BV
    E(1,N) = norm(D*v-vprime,inf);
    v = exp(-x.^(-2)); vprime = 2.*v./x.^3; % C-infinity
    E(2,N) = norm(D*v-vprime,inf);
    v = 1./(1+x.^2); vprime = -2*x.*v.^2;
    % analytic in [-1,1]
    E(3,N) = norm(D*v-vprime,inf);
    v = x.^10; vprime = 10*x.^9;
    % polynomial
    E(4,N) = norm(D*v-vprime,inf);
end
toc;

% Plot results:
titles = {'|x^3|','exp(-x^{-2})','1/(1+x^2)','x^{10}'}; clf
for iplot = 1:4
```

```
        subplot(2,2,iplot)
        semilogy(1:Nmax,E(iplot,:),'.','markersize',12)
        line(1:Nmax,E(iplot,:),'linewidth',.8)
        axis([0 Nmax 1e-16 1e3]), grid on
        set(gca,'xtick',0:10:Nmax,'ytick',(10).^(-15:5:0))
        xlabel N, ylabel error, title(titles(iplot))
    end
```



4 Fast Fourier Transform (FFT)

A Fast Fourier Transform (FFT) is an algorithm that computes the Discrete Fourier transform (DFT) of a sequence. FFT provides a crucial speed up for some calculations. Let $x_0, \ldots, x_{N-1}$ be complex numbers. The Discrete Fourier Transform (DFT) is defined by the formula :

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N} \quad k = 0, \ldots, N-1 \tag{13}$$

where $e^{i2\pi/N}$ is a primitive N-th root of 1.

It can be clearly seen, an N-point sequence yields an N-point transform. We can express X_k as dot product of two vectors :

$$X_k = \begin{bmatrix} 1 & e^{-i\frac{2\pi k}{N}} & e^{-i\frac{2\pi k}{N}2} & \ldots & e^{-i\frac{2\pi k}{N}(N-1)} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{bmatrix} \tag{14}$$

Now, take $W_N = e^{-j\frac{2\pi}{N}}$. X_k becomes :

$$X_k = \begin{bmatrix} 1 & W_N^k & W_N^{2k} & \dots & W_N^{(N-1)k} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{bmatrix} \tag{15}$$

When we write this equation for k= 0 to N-1, we get the following:

$$\mathbf{X}=\mathbf{W}\mathbf{x} \tag{16}$$

Here, **W** is NxN matrix, known as **'DFT matrix'**. Each inner product requires N complex multiplications and there are N inner products. Hence, trivial computational complexity of DFT is $\mathcal{O}(N^2)$. So, even for moderately large number of N, the straightforward method is impractical. Now we look at a specific class of algorithms which can reduce the computation time. This class of algorithm uses Divide and Conquer approach.

Divide and Conquer :

In this approach, we re-express the discrete Fourier transform (DFT) of an arbitrary composite size N (such that N= N_1xN_2), in terms of N_1 smaller DFTs of sizes N_2, and then combine the individual DFT results to get the originally required DFT. We do this recursively and so the computation time reduces to $\mathcal{O}(N \log N)$. Now there are various modifications done to this basic approach in order to increase efficiency. The Cooley-Tukey algorithm is the most common FFT algorithm.

4.1 Cooley–Tukey FFT algorithm

Cooley–Tukey algorithm breaks the Discrete Fourier Transform (DFT) into smaller DFTs, it can be combined arbitrarily with any other algorithm for the DFT. We know that discrete Fourier transform (DFT) is defined by the formula:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk} \tag{17}$$

where **k** is an integer ranging from **0** to **N-1**. We form two sequences from $\{x_n\}$ as :

$$\{g_n\} = \{x_{2n}\}, \quad \{h_n\} = \{x_{2n+1}\} \tag{18}$$

where $\{g_n\}$ contains the even-indexed elements of sequence, and $\{h_n\}$ contains the odd-indexed elements of the sequence. Assuming that one of the factor of 'N' is 2, we get the

DFT of $\{x_n\}$ as:

$$\begin{aligned} X_k &= \sum_{n=0}^{N-1} x_n W_N^{nk} \\ &= \sum_{r=0}^{\frac{N}{2}-1} x_{2r} W_N^{(2r)k} + \sum_{r=0}^{\frac{N}{2}-1} x_{2r+1} W_N^{(2r+1)k} \\ &= \sum_{r=0}^{\frac{N}{2}-1} g_r W_N^{(2r)k} + W_N^k \sum_{r=0}^{\frac{N}{2}-1} h_r W_N^{(2r)k} \end{aligned} \tag{19}$$

We can write,

$$W_N^{2rk} = e^{-i\frac{2\pi}{N}(2rk)} = e^{-i\frac{2\pi}{N/2}(rk)} = W_{N/2}^{rk} \tag{20}$$

Hence,

$$\begin{aligned} X_k &= \sum_{r=0}^{\frac{N}{2}-1} g_r W_{N/2}^{rk} + W_N^k \sum_{r=0}^{\frac{N}{2}-1} h_r W_{N/2}^{rk} \\ &= G_k + W_N^k H_k \quad k = 0, 1, \ldots, N-1 \end{aligned} \tag{21}$$

where $\{G_k\}$ and $\{H_k\}$ are $\frac{N}{2}$ point DFTs.
The extra computation required for combining the two $\frac{N}{2}$ point DFTs is the multiplicative factor W_N^k for k= 0, 1, . . . , N-1 and a single addition. This W_N^k is called ***twiddle factor***. Now, using periodiciy of complex exponential function :

$$e^{i(\theta+2k\pi)} = e^{i\theta} \tag{22}$$

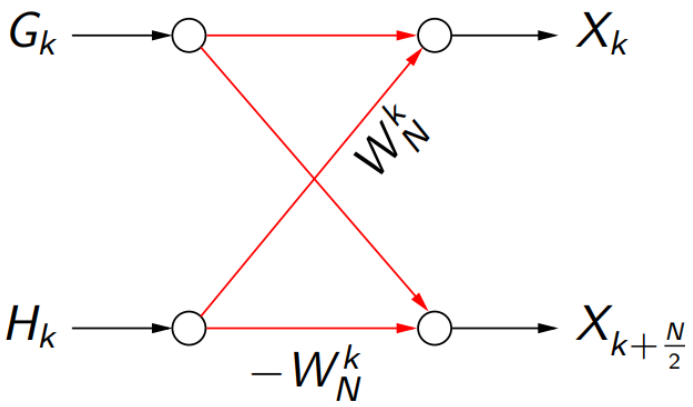
we can obtain $X_{k+\frac{N}{2}}$ as:

$$\begin{aligned} X_{k+\frac{N}{2}} &= \sum_{r=0}^{N/2-1} x_{2r} e^{-\frac{2\pi i}{N/2} r\left(k+\frac{N}{2}\right)} + e^{-\frac{2\pi i}{N}\left(k+\frac{N}{2}\right)} \sum_{r=0}^{N/2-1} x_{2r+1} e^{-\frac{2\pi i}{N/2} r\left(k+\frac{N}{2}\right)} \\ &= \sum_{r=0}^{N/2-1} g_r e^{-\frac{2\pi i}{N/2} rk} e^{-2\pi ri} + e^{-\frac{2\pi i}{N} k} e^{-\pi i} \sum_{r=0}^{N/2-1} h_r e^{-\frac{2\pi i}{N/2} rk} e^{-2\pi ri} \\ &= \sum_{r=0}^{N/2-1} g_r e^{-\frac{2\pi i}{N/2} rk} - e^{-\frac{2\pi i}{N} k} \sum_{r=0}^{N/2-1} h_r e^{-\frac{2\pi i}{N/2} rk} \\ &= G_k - W_N^k H_k \qquad\qquad \text{for k=0, 1,. . . , N-1} \end{aligned} \tag{23}$$

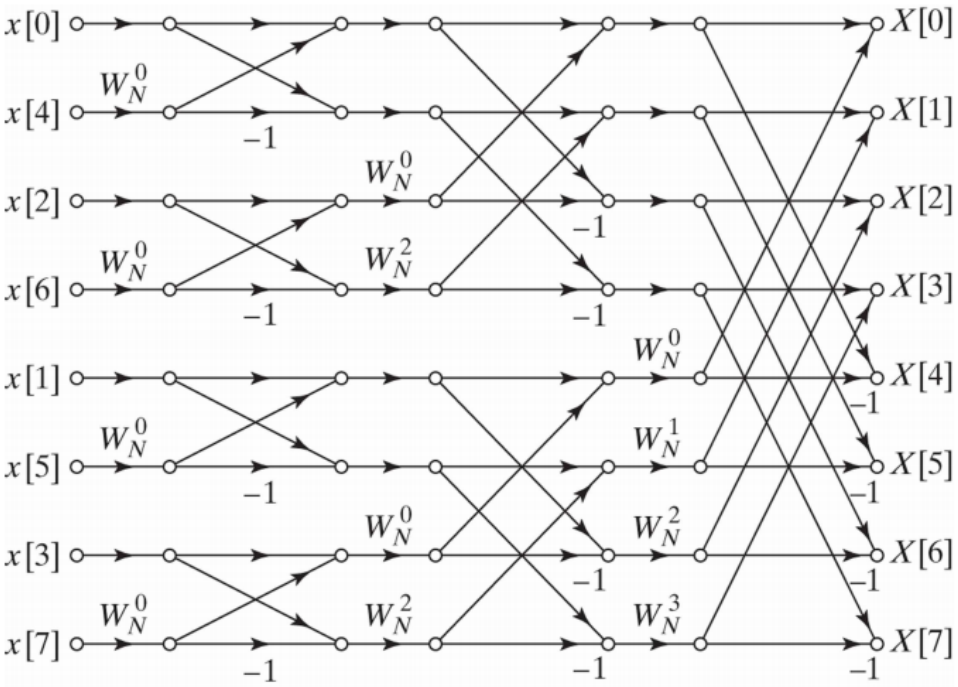
We can also observe from the above calculation that:

$$\begin{aligned} W_N^{k+\frac{N}{2}} &= -W_N^k \\ G_{k+\frac{N}{2}} &= G_k \\ H_{k+\frac{N}{2}} &= H_k \end{aligned} \tag{24}$$

Hence, if $X_k = G_k + W_N^k H_k$, then $X_{k+\frac{N}{2}} = G_k - W_N^k H_k$. We can take advantage of periodicity of complex exponential by computing $W_N^k H_k$ only for k= 0 to $\frac{N}{2}$-1 . Thus, the extra computations needed to combine the two $\frac{N}{2}$ point DFT is only $\frac{N}{2}$. We can picture these FFT operation by following diagram:



We apply this Divide and Conquer approach for further $\frac{N}{2}$ point DFT of the sequences $\{g_r\}$ and $\{h_r\}$. This idea can be applied recursively $\log_2$N times if N is a power of 2. Such algorithms are called radix 2 decimation-in-time (**DIT**) algorithms. The algorithm gains its speed by re-using the results of intermediate computations to compute multiple DFT outputs. Note that final outputs are obtained by a $+/-$ combination of G_k and $H_k e^{\frac{-2\pi i}{N}k}$, which is simply a size-2 DFT.



The above figure demonstrates the DIT flowgraph for N=8 (which is a power of 2).

Overall Computation time:
We know that direct method requires N^2 multiplications. For radix 2 DIT algorithm we can write the following recurrence relation for computational complexity:

$$\mathrm{T}(N) = 2\mathrm{T}(\frac{N}{2}) + \frac{N}{2} \tag{25}$$

Here, the extra $\frac{N}{2}$ is due to the *twiddle factors*. On solving this recurrence relation, we get $\mathrm{T}(N)=\mathcal{O}(N \log_2 N)$. For N=8, direct method would require 64 multiplications, while radix 2 DIT algorithm requires at most 24 computations, which is very efficient than the naive solution.
Now we discuss FFT algorithms for the case when N is not a power of two or more general case when N is a product of two co-prime number.

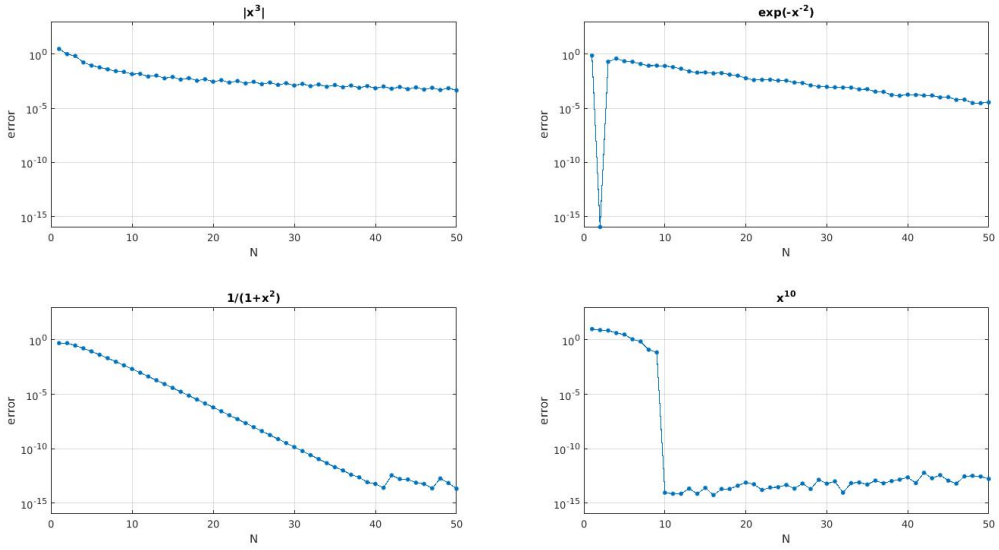
4.1.1 Implementation

We use a discrete Fourier transform algorithm for Chebyshev spectral differentiation. The key point is that the polynomial interpolant 'q' of a function 'f' can be differentiated by finding a trigonometric polynomial interpolant 'Q' of 'F' , differentiating in Fourier space, and transforming back to the x variable. Once we are working on a periodic equispaced grid, we can take advantage of the discrete Fourier transform.

```
% p18.m - Chebyshev differentiation via FFT (compare p11.m)

xx = -1:.01:1;
ff = exp(xx).*sin(5*xx); clf

for N = [10 20]
    x = cos(pi*(0:N)'/N); f = exp(x).*sin(5*x);
    subplot('position',[.15 .66-.4*(N==20) .31 .28])
    plot(x,f,'.','markersize',14), grid on
    line(xx,ff,'linewidth',.8)
    title(['f(x), N=' int2str(N)])
    error = chebfft(f, x) - exp(x).*(sin(5*x)+5*cos(5*x));
    subplot('position',[.55 .66-.4*(N==20) .31 .28])
    plot(x,error,'.','markersize',14), grid on
    line(x,error,'linewidth',.8)
    title(['error in f''(x), N=' int2str(N)])
end
```

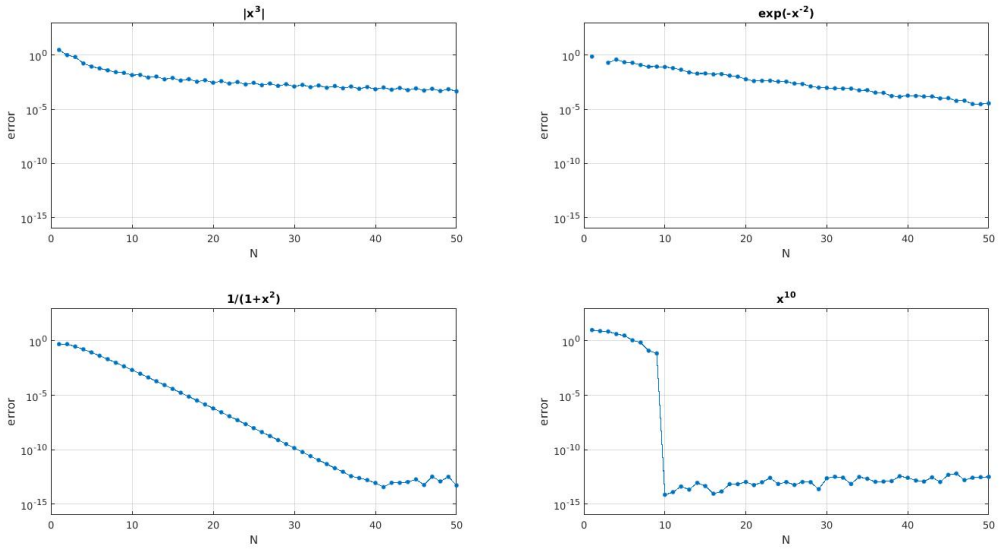



```
% p18_2.m - accuracy of FFT spectral differentiation (compare p12.m)
% The example functions in this program are the same as those in p12.m

tic;
Nmax = 500; E = zeros(3,Nmax);
for N = 1:Nmax;
    x = cos(pi*(0:N)'/N);
    v = abs(x).^3; vprime = 3*x.*abs(x);
    % 3rd deriv in BV
    E(1,N) = norm(chebfft(v, x)-vprime,inf);
    v = exp(-x.^(-2)); vprime = 2.*v./x.^3; % C-infinity
    E(2,N) = norm(chebfft(v, x)-vprime,inf);
    v = 1./(1+x.^2); vprime = -2*x.*v.^2;
    % analytic in [-1,1]
    E(3,N) = norm(chebfft(v, x)-vprime,inf);
    v = x.^10; vprime = 10*x.^9;
    % polynomial
    E(4,N) = norm(chebfft(v, x)-vprime,inf);
end
toc;

% Plot results:
titles = {'|x^3|','exp(-x^{-2})','1/(1+x^2)','x^{10}'}; clf
```

```
for iplot = 1:4
    subplot(2,2,iplot)
    semilogy(1:Nmax,E(iplot,:),'.','markersize',12)
    line(1:Nmax,E(iplot,:),'linewidth',.8)
    axis([0 Nmax 1e-16 1e3]), grid on
    set(gca,'xtick',0:10:Nmax,'ytick',(10).^(-15:5:0))
    xlabel N, ylabel error, title(titles(iplot))
end
```



4.2 Prime-factor FFT algorithm

The prime-factor algorithm **(PFA)**, also called the **Good–Thomas algorithm**, is a fast Fourier transform (FFT) algorithm that re-expresses the discrete Fourier transform (DFT) of a size N = N_1xN_2 as a two-dimensional N_1xN_2 DFT, but only for the case where N_1 and N_2 are relatively prime. These smaller transforms of size N_1 and N_2 can then be evaluated by applying PFA recursively or by using some other FFT algorithm. It uses a more complicated re-indexing of the data based on the Chinese remainder theorem (CRT). Recall DFT of a sequence $\{x_0, \ldots, x_{N-1}\}$ is defined by the formula :

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk} \quad k = 0, \ldots, N-1 \tag{26}$$

- **Re-indexing :** The first step involved in PFA is re-indexing of input and output sequence. Suppose $N = N_1$xN_2, where N_1 and N_2 are co-prime. We can define a bijective re-indexing of the input n and output k by:

$$\begin{aligned} n &= n_1 N_2 + n_2 N_1 \mod N \\ k &= k_1 N_2^{-1} N_2 + k_2 N_1^{-1} N_1 \mod N \end{aligned} \tag{27}$$

where N_1^{-1} denotes the modular multiplicative inverse of N_1 modulo N_2, and similarly, N_2^{-1} denotes the modular multiplicative inverse of N_2 modulo N_1. Also, k_m and n_m takes values from 0,..., $N_m - 1$, for m=1,2 . Here, re-indexing of k is called the CRT mapping. It refers to the fact that k is the solution to the Chinese remainder problem k = k_1 mod N_1 and k = k_2 mod N_2 .

- **Substitution:** The above re-indexing is then substituted into the formula for the DFT. Since, $e^{2\pi i} = 1$ and this exponent is evaluated modulo N, we can set any $N = N_1 \text{x} N_2$ term in the nk product to zero. Also, X_k and x_n are periodic in N, so their subscripts are evaluated modulo N. We also use the fact that $N_1^{-1} N_1$ is unity when evaluated modulo N_2 and $N_2^{-1} N_2$ is unity when evaluated modulo N_1 owing to the definition of N_1^{-1} and N_2^{-1}. After these simplifications, we get :

$$X_{k_1 N_2^{-1} N_2 + k_2 N_1^{-1} N_1} = \sum_{n_1=0}^{N_1-1} \left(\sum_{n_2=0}^{N_2-1} x_{n_1 N_2 + n_2 N_1} e^{-\frac{2\pi i}{N_2} n_2 k_2} \right) e^{-\frac{2\pi i}{N_1} n_1 k_1} \tag{28}$$

The inner and outer sums are simply DFTs of size N_2 and N_1, respectively.

- We can again recursively split these DFTs of sizes N_2 and N_1 after factorizing N_2 and N_1 into product of two co-prime numbers. Any composite number can be written as product of two co-prime numbers or as a power of any prime number 'p' . We have discussed PFA for the first case, now we will discuss one last algorithm for the case when N is a power of any prime number 'p' .

4.3 Rader's FFT algorithm

Rader's algorithm is a FFT algorithm that computes the discrete Fourier transform (DFT) of prime sizes by re-expressing the DFT as a cyclic convolution. Winograd extended Rader's algorithm to include prime-power DFT sizes p^m and today Rader's algorithm is sometimes described as a special case of Winograd's FFT algorithm. The Winograd FFT algorithm tends to reduce the number of multiplications at the price of increased additions. But, for composite sizes such as prime powers, the Cooley–Tukey FFT algorithm is much simpler and more practical to implement, so Rader's algorithm is typically only used for large-prime base cases of Cooley–Tukey's recursive decomposition of the DFT. Again consider the formula of DFT of a sequence $\{x_0, \ldots, x_{N-1}\}$:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk} \quad k = 0, \ldots, N-1 \tag{29}$$

- **Re-indexing :** If N is a prime number, then the set of non-zero indices n = 1,... N–1 forms a group under multiplication modulo N. Moreover, there exists a generator of the group, an integer g such that $n = g^q (mod\ N)$ for any non-zero index n and for a unique q in 0,... N–2, forming a bijection from q to non-zero n. Similarly $k = g^p (mod\ N)$ for any non-zero index k and for a unique p in 0,...,N–2, where the negative exponent denotes the multiplicative inverse of g^p modulo N. Using these information we can write DFT with new indices p and q as :

$$\begin{aligned} X_0 &= \textstyle\sum_{n=0}^{N-1} x_n \qquad k = 0, \\ X_{g^p} &= x_0 + \textstyle\sum_{q=0}^{N-2} x_{g^q} e^{-\frac{2\pi i}{N} g^{-(p-q)}} \quad p = 0, \ldots, N-2 \end{aligned} \tag{30}$$

  The final summation above, is a cyclic convolution of the two sequences a_q and b_q of length N–1 (q = 0,...,N–2) defined by:

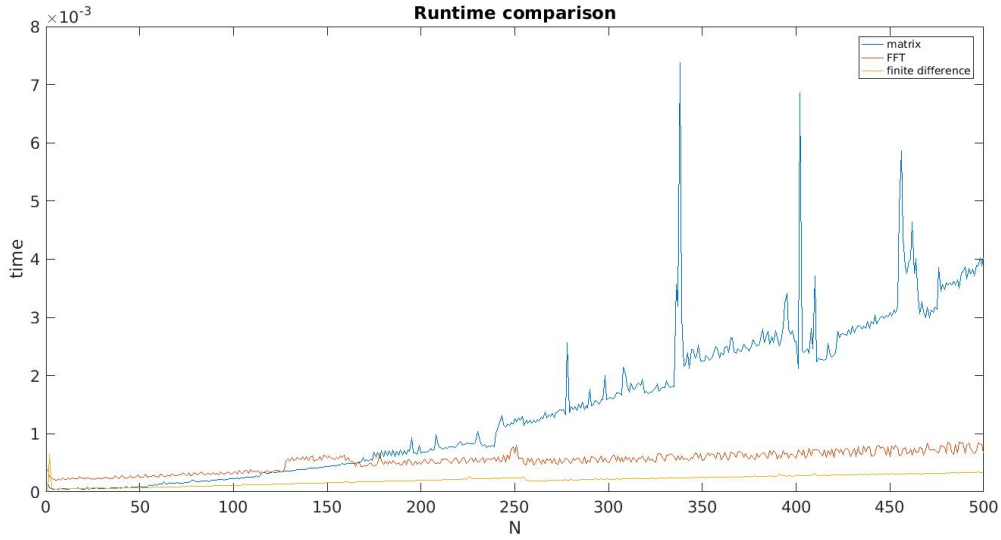
$$\begin{aligned} a_q &= x_{g^q} \\ b_q &= e^{-\frac{2\pi i}{N} g^{-q}} \end{aligned} \tag{31}$$

- **Evaluating the Convolution :** The convolution theorem states that under suitable conditions the Fourier transform of a convolution of two sequence is the pointwise product of their Fourier transforms. Since, N-1 is composite, this convolution can be performed directly. So, after multiplying DFT of a_q pointwise by the DFT of b_q, we calculate the inverse DFT of this inner product.

- The convolution derived above is of length N-1, which is composite. We can thus calculate the length N-1 discrete Fourier transforms required by the above approach using our Cooley-Tukey divide and conquer algorithm. But, this may not be efficient always (say if N-1 itself has large prime factors,which will require recursive use of Rader's algorithm). Hence we add zero padding (increasing the length of sequence by adding zeros) to make the length equal to power of 2. This can then be evaluated in O(N log N) time without the recursive application of Rader's algorithm.

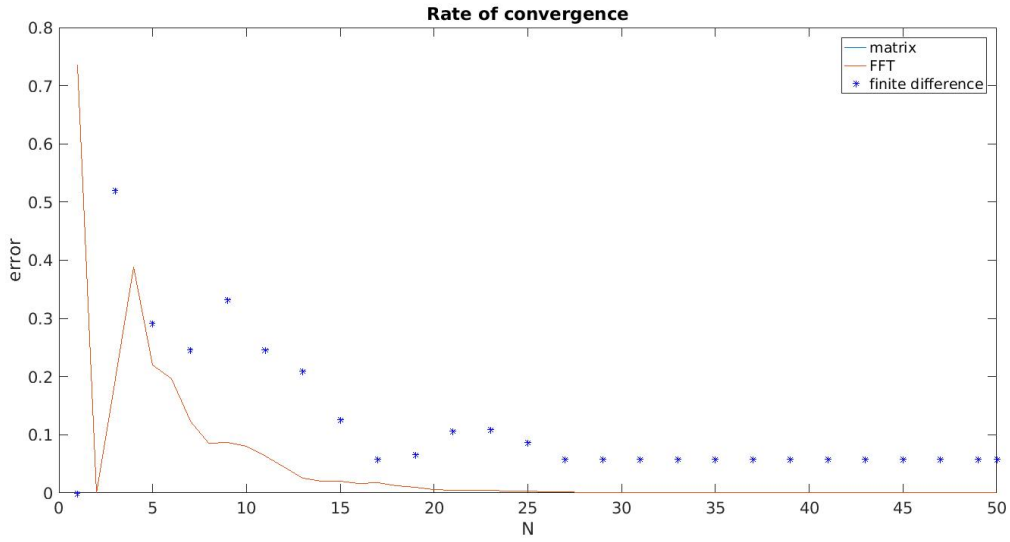
5 Conclusion

5.1 Runtime comparisons



- The running time of FFT algorithm is fairly stable with respect to changes in the number of Chebychev nodes.
- Finite difference method has the fastest running time owing to the larger overall error in the resulting approximation.

5.2 Rate of convergence



- The rate of convergence of Chebychev differentiation and FFT algorithm is exactly the same. The error stabilizes at 0 after N=20.

- Finite difference method has the slowest rate of convergence and the error stabilizes at 0.08 after N=25.

References

[1] Spectral methods in Matlab. Lloyd N. Trefethen

[2] Spectral Methods - Fundamentals in Single Domains. C. Canuto, M. Y. Hussaini, A. Quarteroni, T. A. Zang

[3] Cooley–Tukey FFT algorithm: en.wikipedia.org/wiki/Cooley-Tukey-FFT-algorithm

[4] Winograd Fourier transform algorithm. Encyclopedia of Mathematics. URL: http://www.encyclopediaofmath.org/index.php?title=Winograd-Fourier-transform-algorithmoldid=25476