EAS 502: Final Report

Sayem Khan

Wednesday, December 11, 2019

Problem 01

Zero finding and root finding methods:

(a) Summarize the methods and the criteria for them to work. List each one's advantages and disadvantages.

(b) (Coding) Use the bisection method, fixed point iteration, and Newton's method to find the zero of the function:

$$f(x) = x^3 - x^2 + 1$$

on the interval $x \in [-1, 1]$. For each method, how many iterations will you need to get the answer to within 10^{-6} ?

(c) Using the problems above, discuss the idea of the order of a method. Define it and explain what it means and how it relates to the problems above.

Solution 1(a)

Zero finding and root finding methods:

i **Bisection Method:** The first technique, based on the Intermediate Value Theorem, is called the Bisection, or Binary-search, method.

Suppose f is a continuous function defined on the interval $[a, b]$, with $f(a)$ and $f(b)$ of opposite sign. The

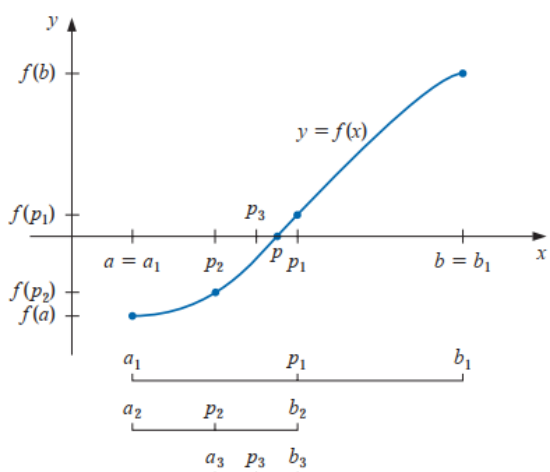


Figure 1: Bisection method: graphical representation.

Intermediate Value Theorem implies that a number p exists in (a, b) with $f(p) = 0$. Although the procedure will work when there is more than one root in the interval (a, b), we assume for simplicity that the root in this interval is unique. The method calls for a repeated halving (or bisecting) of sub-intervals of $[a, b]$ and, at each step, locating the half containing p.

To begin the, $a_1 = a$ and $b_1 = b$ and let p_1 be the midpoint $[a, b]$; that is,

$$p_1 = a_1 + \frac{b_1 - a_1}{2} = \frac{a_1 + b_1}{2}$$

- If $f(p_1) = 0$, then, $p = p_1$, and we are done.
- If $f(p_1) \neq 0$, then $f(p_1)$ has the same sign as either $f(a_1)$ or $f(b_1)$
  - If $f(p_1)$ and $f(a_1)$ have same sign, $p \in (P_1, b_1)$. Set $a_2 = p_1$ and $b_2 = b_1$
  - If $f(p_1)$ and $f(a_1)$ have opposite sign, $p \in (P_1, b_1)$. Set $a_2 = a_1$ and $b_2 = p_1$

  Then reapply the process to the interval $[a_2, b_2]$.

Stopping Criteria:

$$|p_n - p_{n-1}| < \varepsilon$$

$$\tfrac{p_n - p_{n-1}}{|p_n|} < \varepsilon, \qquad p_n \neq 0, \qquad \textit{or}$$

$$|f(p_n)| < \varepsilon$$

Error Bound:

$$|p_n - p| \leq \frac{b - a}{2^n}, \qquad for \qquad n \geq 1$$

Advantages:

(a) convergence is guaranteed to a root of continuous function in an interval.

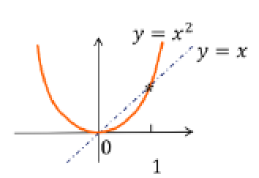
(b) Convergence is linear, absolute error is halved in every step.

Disadvantages:

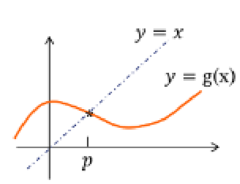
(a) It is not possible to find multiple root.

(b) Comparatively slow.

ii **Fixed-Point Iteration**
The number p is a fixed point for a given g if $g(p) = p$,

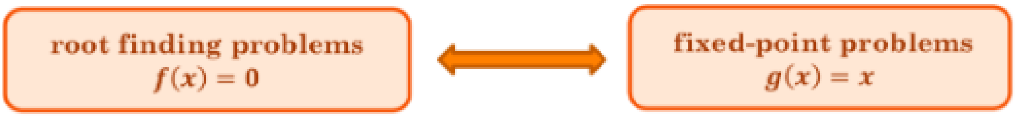


$f(x) = x^2$ has fixed points at $x = 0$ and $x = 1$.



$g(x)$ has a fixed point at p where p is the solution to $g(x) = x$.

The relation between root finding problems $f(x) = 0$ and fixed-point problems $g(x) = x$: The $f(x) = 0$ can be



converted into the form of $g(x) = x$ and using the recursive relation:

$$x_{i+1} = g(x_i), \qquad 0, 1, 2, \cdots$$

So, it can be said that, if $g(x)$ has a fixed point at p, then, $f(x) = x - g(x)$.

Criteria:

- **Existence and uniqueness of a fixed point**
  (a) If $g \in [a, b]$ and $g(x) \in [a, b]$ for all $x \in [a, b]$,then g has a fixed point in $[a, b]$
  (b) If, in addition, there exits a constant $0 < k < 1$ such that $|g'(x) \leq k|$ for all $x \in (a, b)$, then the fixed point in $[a, b]$ is unique.
- **Fixed point Theorem**
  Suppose that,
  (a) $g \in C[a, b]$ and $g(x) \in [a, b]$ for all $x \in [a, b]$
  (b) In addition, there exits a constant $0 < k < 1$ such that $|g'(x) \leq k|$ for all $x \in (a, b)$,

  Then for any initial approximation p_0 in the sequence defined by $p_n = g(p_{n-1})$ converges to the unique fixed point in $[a, b]$
- **Corollary (Error Estimate):**
  (a) $|p_n - p| \leq k^n \, max\, \{p_0 - a, b - p_0\}$
  (b) $|p_n - p| \leq \frac{k^n}{1-k}|P_1 - p_0|$, for all $n \geq 1$
- **Stopping Criteria**
  (a) A bound in absolute value of the error:

  $$|x_{n+1} - x_n| < \delta$$

  (b) A bound in the relative value of the error: $\frac{|x_{n+1} - x_n|}{|x_{n+1}|} < \delta$
  (c) A check whether $f(x_n) \sim 0$, i.e. ensure that

  $$|f(x_n)| < \delta$$

  (d) A limit on the number of steps in the iteration.

Advantages:

(a) Easy to implement.

Disadvantages:

(a) It is not possible to find multiple root.
(b) Comparatively slow.

iii **Newton's Method**

Solve nonlinear equations in form of $f(x) = 0$. Suppose that $f(x) = 0$ has a solution $p \in [a, b]$. Let $p_0 \in [a, b]$ be an approximation of p. We consider the first Taylor's polynomial for $f(x)$ expanded about p_0,

$$f(x) = f(p_0) + f'(p_0)(x - p_0) + \frac{f''(\xi(x))}{2}(x - p_0)^2$$

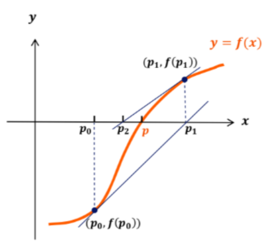
Evaluate at $x = p$,

$$f(p) = f(p_0) + f'(p_0)(p - p_0) + \frac{f''(\xi(p))}{2}(p - p_0)^2$$

If $|p - p_0|$ is "small",

$$0 \approx f(p_0) + f'(p_0)(p - p_0)$$

If $f'(p_0) \neq 0 \Rightarrow p \approx p_0 - \frac{f(p_0)}{f'(p_0)}$



The tangent line of $f(x)$ at $(p_0, f(p_0))$ is

$$y - f(p_0) = f'(p_0)(x - p_0).$$

It intersects the x-axis at $(p_1, 0)$. Then

$$0 - f(p_0) = f'(p_0)(p_1 - p_0),$$

$$p_1 = p_0 - \frac{f(p_0)}{f'(p_0)}.$$

Similarly,

$$p_2 = p_1 - \frac{f(p_1)}{f'(p_1)}$$

$$\vdots$$

Figure 2: Graphical representation of Newton's method.

- **Theorem** If
  (a) $f \in C^2[a, b]$
  (b) $f(x) = 0$ has a solution $p \in [a, b]$, and
  (c) $f'(p) \neq 0$

  Then there exists a $\delta > 0$ such that the sequence $\{p_n\}_{n=1}^{\infty}$ generated by Newton's method convergence to p for any $p_0 \in [p - \delta, p + \delta]$
- **Stopping Criteria**

$$\left|\frac{p_n - p_{n-1}}{p_n}\right| < \epsilon, \quad |p_n| \neq 0$$

Advantages:

(a) Easy to implement.

(b) When the method converges, it does so quadratically.

Disadvantages:

(a) It is not possible to find multiple root.

iv **Secant Method**

The secant method is modified Newton's method to avoid evaluating $f'(x)$ in each iteration.
From Newton's method,

$$p_n = P_{n-1} - \frac{f(p_{n-1})}{f'(p_{n-1})}, \qquad n \geq 1$$

$$f'(p_{n-1}) = \lim_{x \to p_{n-1}} \frac{f(x) - f(p_{n-1})}{x - p_{n-1}} \approx \frac{f(p_{n-2}) - f(p_{n-1})}{p_{n-2} - p_{n-1}}$$

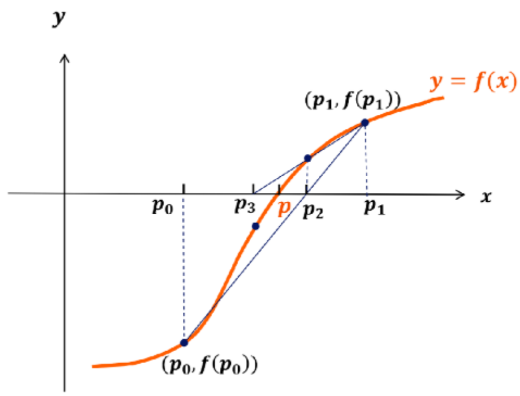


Figure 3: Graphical representation of Secant method.

Let,

$$p_n = p_{n-1} - \frac{f(p_{n-1})}{\frac{f(p_{n-1}) - f(p_{n-2})}{p_{n-1} - p_{n-2}}}$$

then, the Secant method,

$$p_n = p_{n-1} - \frac{f(p_{n-1})(p_{n-1} - p_{n-2})}{f(p_{n-1}) - f(p_{n-2})}, \quad n \geq 2$$

So, for the Secant method two approximations p_0 and p_1 is needed.

- **Stopping Criteria**

$$|\frac{p_n - p_{n-1}}{p_n}| < \epsilon, \quad |p_n| \neq 0$$

Advantages:

(a) It converges at faster than a linear rate, so that it is more rapidly convergent than the bisection method.

(b) It does not require use of the derivative of the function, something that is not available in a number of applications.

(c) It requires only one function evaluation per iteration, as compared with Newton's method which requires two.

Disadvantages:

(a) It may not converge.

(b) There is no guaranteed error bound for the computed iterates.

(c) It is likely to have difficulty if $f'(\alpha) = 0$. This means the x-axis is tangent to the graph of $y = f(x)$ at $x = \alpha$.

(d) Newton's method generalizes more easily to new methods for solving simultaneous systems of nonlinear equations.

v **Chord Method**

From Newton's method,

$$p_n = P_{n-1} - \frac{f(p_{n-1})}{f'(p_{n-1})}, \qquad n \geq 1$$

$$f'(p_{n-1}) = \lim_{x \to p_{n-1}} \frac{f(x) - f(p_{n-1})}{x - p_{n-1}} \approx \frac{f(b) - f(a)}{b - a}$$

Let,

$$p_n = p_{n-1} - \frac{f(p_{n-1})}{\frac{f(b)-f(a)}{b-a}}$$

Then, Chord Method:

$$p_n = p_{n-1} - \frac{f(p_{n-1}) - (b - a)}{f(b) - f(a)}, \quad n \geq 2$$

Note that, in Chord method, each iteration uses fixed slope.

- **Stopping Criteria**

$$|\frac{p_n - p_{n-1}}{p_n}| < \epsilon, \quad |p_n| \neq 0$$

Advantages:

(a) Easy to implement.
(b) No need to calculate derivative.

Disadvantages:

(a) It is not possible to find multiple root.
(b) Comparatively slow.

Solution 1(b)

Bisection Method:

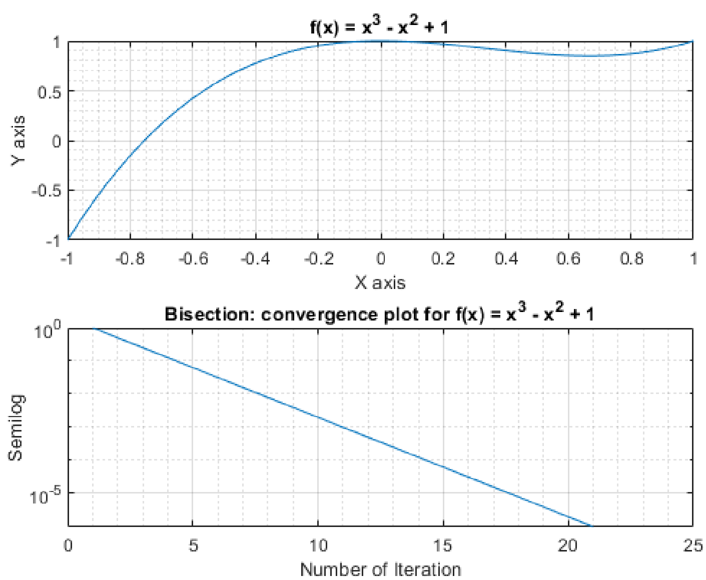


Figure 4: $f(x) = x^3 - x^2 + 1$ and Convergence curve for Bisection Method.

```
The k value = 2.567500--BIsection-- The value at 21 iteration is = -0.754878
```

```
function [p, err] = bisect(a, b, tol, Nmax, fqn)
    p = zeros(1, Nmax);
    err = zeros(1, Nmax);
    f_a = feval(fqn,a);
    if feval(fqn,a)* feval(fqn,b) > 0
        fprintf("This Equation has not any root in interval [%d, %d].", a, b);
        return;
    end
    if feval(fqn,a) == 0
        fprintf("The root of this eqn is %d", a);
        return;
    end
    if feval(fqn,b) == 0
        fprintf("The root of this eqn is %d", b);
        return;
    end

    i = 1;
    while i < Nmax
        p(i) = a + (b - a)/2;
        err(i) = abs((b-a)/2);
        f_p = feval(fqn, p(i));
        if f_p == 0 || err(i) < tol
            fprintf('--BIsection-- The value at %d iteration is = %f\n', i, p(i));
            break;
        end
        %i = i + 1;
        if f_a*f_p > 0
            a = p(i);
            f_a = f_p;
        else
            b = p(i);
        end
        i = i + 1;

        if i > Nmax
            fprintf('--BiSection-- Max iteration reached, could not solve.\n');
            break;
        end
    end
    end
```

Fixed Point Method:

$$f(x) = x^3 - x^2 + 1 = 0$$

we can write this function as:

$$x = \frac{(x-1)}{(x^2 - x + 1)} = g(x)$$

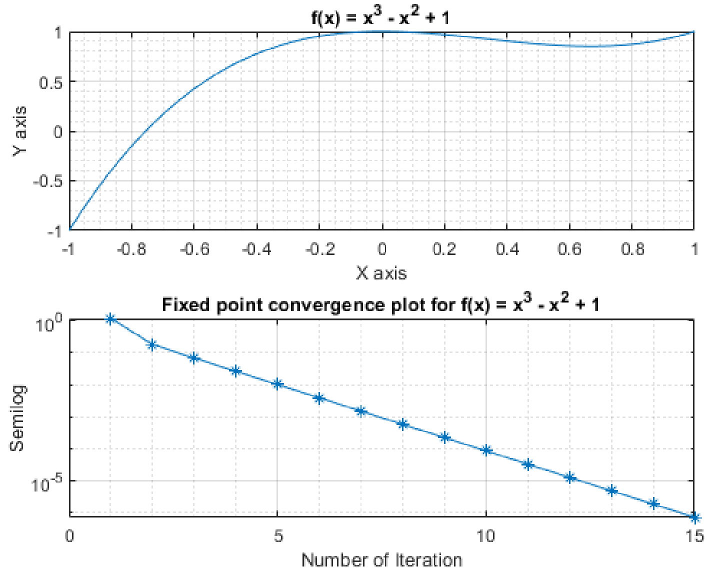


Figure 5: $f(x) = x^3 - x^2 + 1$ and Convergence curve for Bisection Method.

```
clear;
clc;
f = @(x) (x-1)./(x.^2 - x + 1);
f1 = @(x) x.^3 - x.^2 + 1;
x = linspace(-2,2, 1000);
y = feval(f1,x);
p0 = 0.3;
df = matlabFunction(diff(sym(f)));
dfeval = feval(df,p0);
k = abs(dfeval);
fprintf('The k value = %f', k);

tol = 1e-6;
Nmax = 1000;
[p, err] = fixpoint(f,p0,tol,Nmax);
subplot(2,1,1);
plot(x,y);
grid on;
grid minor;
title('f(x) = x^3 - x^2 + 1');
xlabel('X axis');
ylabel('Y axis');

%figure
subplot(2,1,2);
semilogy(err,'-*');
grid on;
grid minor;
title('Fixed point convergence plot for f(x) = x^3 - x^2 + 1')
xlabel ('Number of Iteration');
```

```matlab
ylabel('Semilog')
```

```
The k value = 0.817177--Fixed Point--Fixed Point Converged after 15 iteration and Root = -0.754878
```

Newton's Method:

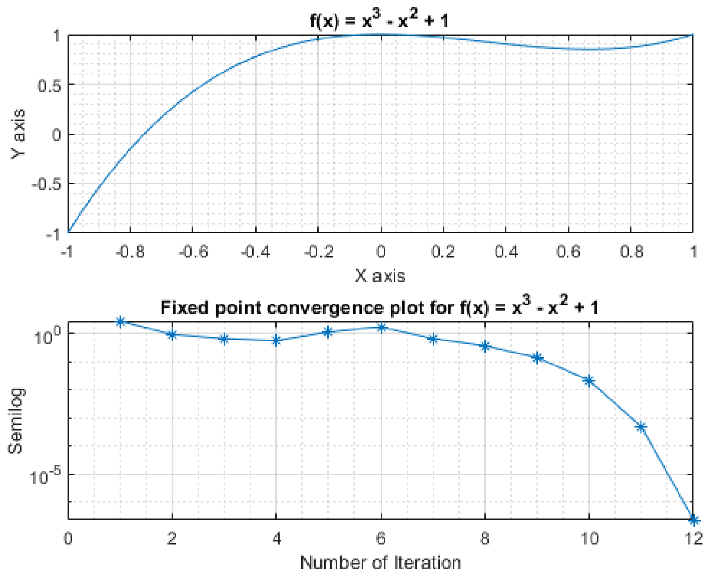


Figure 6: $f(x) = x^3 - x^2 + 1$ and Convergence curve for Newton's Method.

```
The k value = 0.330000--Newton Method -- The Value at 12 iteration is = -0.754878
```

```matlab
function [p, err] = newton(p0, tol, Nmax, fqn, dfqn)
i = 1;
p = zeros(1, Nmax);
err = zeros(1, Nmax);
while i <= Nmax
    m = dfqn(p0);
    if m ~= 0
        p(i) = p0 - fqn(p0)/m;
        err(i) = abs(p(i) - p0);
        if err(i) < tol
            fprintf('--Newton Method -- The Value at %d iteration is = %f \n',i,p(i));
            p = p(1:i);
            err = err(1:i);
            break;
        else
            p0 = p(i);
            i = i + 1;
        end
    else
```

```
            fprintf('Can not solve!! The derrivative is ZERO at %f point.',p0);
            break;
        end
        if i > Nmax
            fprintf('--Newtons Method-- Max iteration reached, could not solve.\n');
            break;
        end
    end
end
```

Solution 1(c)

• Order of Convergence:
Suppose $\{p_n\}_{n=0}^{\infty}$ is a sequence that convergences to p, with $p_n \neq p$ for all n. If positive constants λ and α exist with,

$$\lim_{n\to\infty} \frac{|p_{n+1} - p|}{|p_n - p|^{\alpha}} = \lambda$$

then $\{p_n\}_{n=0}^{\infty}$ converges to p of order α, with asymptotic error constant λ.
An iterative technique of the form $p_n = g(p_{n-1})$ is said to be of order α if the sequence $\{p_n\}_{n=0}^{\infty}$ converges to the solution $p = g(p)$ of order α. In general, a sequence with a higher order of convergence converges more rapidly that a sequence with a lower order. The asymptotic constant affects the speed of convergence but not to the extent of the order. Two cases of order are given special attention.

- If $\alpha = 1$ (and $\lambda < 1$), the sequence is linearly convergent.
- If $\alpha = 2$, the sequence is linearly convergent.

From the figure 4, 5, and (d), the order of Bisection method and Fixed point Method is one (1) and the order of Newton's Method is two (2). For more practical purpose, this is the formula for order of convergence:

$$\alpha = \frac{\frac{P_{n+1}-P}{P_n-P}}{\frac{P_n-P}{P_{n-1}-P}}$$

Problem 02

Polynomial Interpolation:

(a) Why does polynomial interpolation make sense?

(b) Explain the different methods to interpolate a function using a polynomial? What are the advantages and disadvantages of each? What circumstances make it appropriate to use each method?

(c) Explain cubic spline interpolation and derive the equations. (With explanations!)

Solution 2(a)

The polynomial interpolation is the interpolation of a given data set by the polynomial of lowest possible degree that passes through the points of the data-set.

- **Definition**
  Given a set of $n+1$ data points (x_i, y_i) where no two x_i are the same, one is looking for a polynomial p of degree at most n with the property,

$$p(x_i) = y_i, \qquad i = 0, 1, \cdots n$$

The unisolvence theorem states that such a polynomial p exists and is unique, and can be proved by the Vandermonde matrix, as described below.
The theorem states that for $n+1$ interpolation nodes (x_i), polynomial interpolation defines a linear bijection,

$$\mathbf{L}_n : \mathbb{K}^{n+1} \to \prod_n$$

where, where $\prod_n$ is the vector space of polynomials (defined on any interval containing the nodes) of degree at most n.

- **Constructing the interpolation polynomial**
  Suppose that the interpolation polynomial is in the form,

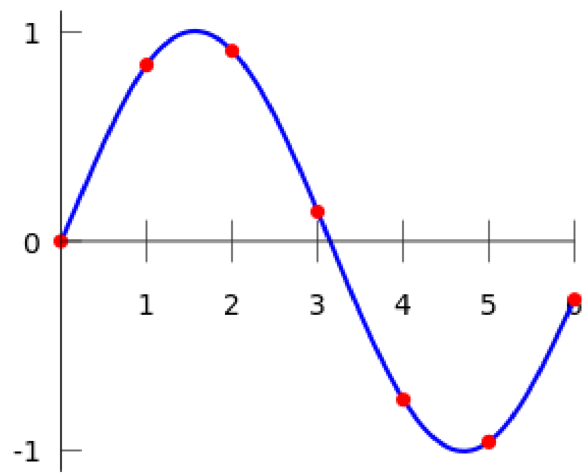


Figure 7: The red dots denote the data points (x_k, y_k), while the blue curve shows the interpolation polynomial.

$$p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_2 x + a_0 \tag{1}$$

The statement that p interpolates the data points means that,

$$p(x_i) = y_i, \qquad for \quad all \quad i \in \{0, 1, \cdots, n\}$$

If we substitute equation 1 in here, we get a system of linear equations in the coefficients a_k. The system in matrix-vector from reads the following multiplication:

$$\begin{bmatrix} x_0^n & x_0^{n-1} & x_0^{n-2} & \cdots & x_0 & 1 \\ x_1^n & x_1^{n-1} & x_2^{n-2} & \cdots & x_1 & 1 \\ \vdots & \vdots & \ddots & \vdots & & \\ x_n^n & x_n^{n-1} & x_n^{n-2} & \cdots & x_n & 1 \end{bmatrix} \begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_0 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix} \tag{2}$$

We have to solve this system for a_k to construct the interpolant $p(x)$. The matrix on the left is commonly referred to as a Vandermonde matrix.
The condition number of the Vandermonde matrix may be large , causing large errors when computing the coefficients a_i if the system of equation is solved using Gaussian elimination.
Several authors have therefore proposed algorithms which exploit the structure of the Vandermonde matrix to compute numerically stable solutions in $O(n^2)$ operations instead of the $O(n^3)$ required by Gaussian

elimination. These methods rely on constructing first a Newton interpolation of the polynomial and then converting it to the monomial form above.

Alternatively, we may write down the polynomial immediately in terms of Lagrange polynomials:

$$p(x) = \sum_{i=0}^{n} \left(\prod_{0 \le j \le n} \frac{x - x_j}{x_i - x_j} \right) y_i$$

For matrix arguments, this formula is called Sylvester's formula and the matrix-valued Lagrange polynomials are the Frobenius covariants.

- **Uniqueness of the interpolating polynomial**
  - **Proof 1**
    Suppose we interpolate through $n+1$ data points with an at-most n degree polynomial $p(x)$ (we need at least $n+1$ data points or else the polynomial cannot be fully solved for). Suppose also another polynomial exists also of degree at most n that also interpolates the $n+1$ points; call it $q(x)$.
    Consider $r(x) = p(x) - q(x)$. We know,
    1. $r(x)$ is a polynomial
    2. $r(x)$ has at most n, since $p(x)$ and $q(x)$ are no higher than this and we are just subtracting them.
    3. At the $n+1$ data points, $r(x_i) = p(x_i) - q(x_i) = y_i - y_i = 0$. Therefore, $r(x)$ has $n+1$ roots.

    But $r(x)$ is a polynomial of degree $\le n$. It has one root too many. Formally, if $r(x)$ is any non-zero polynomial, it must be written as $r(x) = A(x - x_0)(x - x_1) \cdots (x - x_n)$, for some constant A. By distributively, the $n+1$ x's multiply together to give leading term Ax^{n+1}, i.e. one degree higher than the maximum we set. So the only way $r(x)$ can exist is if $A = 0$, or equivalently, $r(x) = 0$.

    $$r(x) = 0 = q(x) - q(x) \Rightarrow p(x) = q(x)$$

    So, $q(x)$ (which could be any polynomial, so long as it interpolates the points) is identical with $p(x)$, and $q(x)$ is unique.
  - **Proof 2**
    Given the Vandermonde matrix used above to construct the interpolant, we can set up the system

    $$Va = y$$

    To prove that V is non-singular we use the Vandermonde determinant formula,

    $$det(V) = \prod_{i,j=0, i<j}^{n} (x_i - x_j)$$

    Since the $n+1$ points are distinct, the determinant can't be zero as $x_i - x_j$ is never zero, therefore V is non-singular and the system has unique solution.

Solution 2(b)

Different types of polynomial interpolation:

(a) **Mononial Interpolation**
Algebraic Polynomials,

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Where, n is a non-negative integer and $a_0 \cdots, a_n$ are real constants (or called coefficients). The above format is called Polynomial interpolation with Monomial Basis. They uniformly approximate continuous functions.

- **Advantage:**
  (a) It uniformly approximate continuous function.
- **Disadvantage:**
  (a) It is computationally expensive.
- **Applications:**
  (a) If the Vandermonde matrix is non singular, this interpolation method can be used.

(b) **Lagrange Interpolation**
Based on set $\{(x_i, y_i)\}_{i=0}^{n}$, find a polynomial $P_n x$ such that,

$$P_n(x_i) = y_i, \qquad for\ all\ i = 0, 1, \cdots, n$$

$\{x_i\}$ are called nodes and $P_n(x)$ is called Lagrange interpolation. Lagrange Basis $\{L_{n,k}(x)\}_{k=0}^{n}$:

$$L_{n,k}(x) = \frac{(x-x_0)(x-x_1)\cdots(x-x_{k-1})(x-x_{k+1})\cdots(x-x_n)}{(x_k-x_0)(x_k-x_1)\cdots(x_k-x_{k-1})\cdots(x_k-x_n)} = \prod_{i\neq j}\frac{(x-x_i)}{(x_k-x_i)}$$

Then the Lagrange Format is,

$$P_n(x) = \sum_{k=0}^{n} y_k L_{n,k}(x)$$

Property:

$$L_{n,k}(x_i) = \delta_{ij}.\begin{cases} 1 & if \quad k = i \\ 0 & if \quad k \neq i \end{cases}$$

- **Advantage:**
  (a) Because of identity matrix it is easy to solve.
- **Disadvantage:**
  (a) With huge data point, it is expensive to calculate with this method.
- **Applications:**
  (a) It can be applied to any data set.

(c) **Newton's Divided-Differences Form of the Interpolating Polynomial**
A Newton polynomial is an interpolation polynomial for a given set of data points. The Newton polynomial is sometimes called Newton's divided differences interpolation polynomial because the coefficients of the polynomial are calculated using Newton's divided differences method.

Given a set of $k+1$ data points,

$$(x_0, y_0), \cdots, (x_j, y_j), \cdots, (x_k, y_k)$$

where no two x_j are the same, the Newton interpolation polynomial is a linear combination of Newtons basis polynomials,

$$N(x) := \sum_{j=0}^{k} a_j n_j(x)$$

with the Newton basis polynomials defined as,

$$n_j(x) := \prod_{i=0}^{j-1}(x - x_i)$$

for $j > 0$ and $n_0(x) \equiv 1$
The coefficients are defined as,

$$a_j := [y_0, \cdots, y_j]$$

where,

$$[y_0, \cdots, y_j]$$

is the notation for divided differences.. Thus the Newton polynomial can be written as,

$$N(x) = [y_0] + [y_0, y_1](x - x_0) + \cdots + [y_0, \cdots, y_k](x - x_0)(x - x_0)(x - x_1) \cdots (x - x_{k-1})$$

The Newton polynomial can be expressed in a simplified form when,

$$x_0, x_1, \cdots, x_k$$

are arranged consecutively with equal spacing. Introducing the notation

$$h = x_{i+1} - x_i \qquad\qquad i = 0, 1, \cdots, (k-1)$$

and

$$x = x + sh$$

The difference $x - x_i$ can be written as $(s - i)h$. So, the Newton Polynomial becomes,

$$\begin{aligned} N(x) &= [y_0] + [y_0, y_1]sh + \cdots + [y_0, \cdots y_k]s(s-1)\cdots(s-k+1)h^k \\ &= \sum_{i=0}^{k} s(s-1)\cdots(s-i+1)h^i[y_0, \cdots, y_i] \\ &= \sum_{i=0}^{k} \binom{s}{i} i! h^i [y_0, \cdots, y_i] \end{aligned}$$

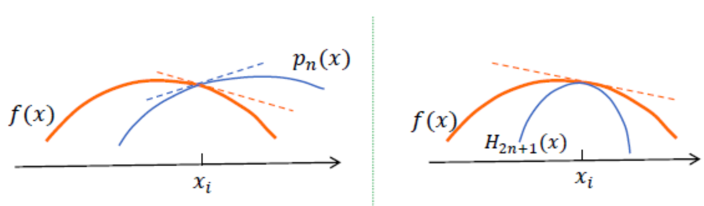
This is called the Newton forward divided difference formula.

- **Advantages**
  - In this method, new data can be added to the data set to create new interpolation polynomial without recalculating the old the old coefficients.
  - Furthermore, if the x_i are distributed equidistantly the calculation of the divided differences becomes significantly easier.
- **Application**
  - The divided-difference formulas are usually preferred over the Lagrange form for practical purposes

(d) **Hermite Interpolation**
Given $\{(x_i, f(x_i), f'(x_i))\}_{i=0}^{n}$ find a polynomials $H_{2n+1}(x)$ of the degree at most $2n + 1$, so that,

$$H_{2n+1}(x_i) = f(x_i) \qquad and \qquad H'_{2n+1}(x_i) = f'(x_i), \qquad for \qquad i = 0, 1, \cdots, n$$

Motivation for Hermite Interpolation is, at the nodes, the Hermite Interpolation $H_{2n_1}(x)$ provides accurate values for function $f(x_i)$ and its first derivative $f'(x_i)$, and hence same tangent lines at $(x_i, f(x_i))$.



• Hermite Interpolation using Lagrange Polynomials: Let, $f \in C^1[a,b]$ and $x_0, x_1, \cdots, x_n \in [a,b]$ are disticnt, the unique polynomial of least degree agreeing with f and f' at $x_0, x_1, \cdots, x_n$ is the Hermite polynomial of degree at most $2n+1$ given by,

$$H_{2n+1}(x) = \sum_{j=0}^{n} f(x_j)H_{n,j}(x) + \sum_{j=0}^{n} f'(x_j)\hat{H}_{n,j}(x)$$

where, $H_{n,j} = [1 - 2(x - x_j)L'_{n,j}(x_j)]L^2_{n,j}(x)$ and $\hat{H}_{n,j}(x) = (x - x_j)L^2_{n,j}(x)$ Moreover, if $f \in C^{2n+2}[a,b]$, then,

$$f(x) = H_{2n+1}(x) + \frac{f^{2n+2}(\xi(x))}{(2n+2)!}\omega^2_{n+1}(x)$$

for some $\xi(x)$ in the interval $[a,b]$.

Solution 2(c)

'Cubic Spline' — is a piece-wise cubic polynomial that is twice continuously differentiable. It is considerably 'stiffer' than a polynomial in the sense that it has less tendency to oscillate between data points. Let's imagine this as having

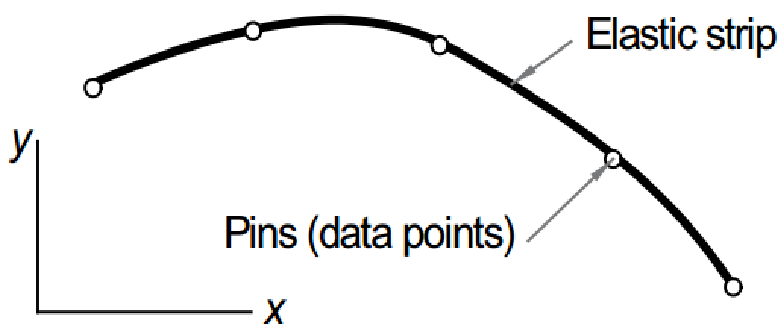


Figure 8: Cubic Spline

an elastic strip pinned to a cork board. Now presenting the main points:

- **Segment**: Each segment of the spline curve is a cubic polynomial.
- **At the pins:** the slope (first derivative) and the bending moment(second derivative) is continuous.
- **At the end points:** there are no bending moments. In mathematical language, this means that the second derivative of the spline at end points are zero. Since these end condition occur naturally in the beam model, the resulting curve is known as the natural cubic spline.
- **Pins:** represents data points or the term that is used in the formula later 'knots'
- **Formula for Cubic Spline**
  $C_i(x_i) = a_i x^3 + b_i x^2 + c_i x + d_i$ with parameters a_i, b_i, c_i and d_i can satisfy the following four equations required for $S(x)$ to be continuous and smooth $(k = 2)$:

  $$C_i(x_i) = y_i, \;\; C_i(x_{i-1} = y_{i-1}), \;\; C'_i(x_i) = C'_{i+1}(x_i), \;\; and$$

  To obtain the four parameters a_i, b_i, C_i and d_i in $C_i(x)$, we first consider

  $$C''_i(x) = (a_i x^3 + b_i x^2 + c_i x + d_i)'' = 6a_i x + 2b_i,$$

which, as a linear function, can be linearly fit by the two end points $f''(x_{i-1}) = M_{i-1}$ and $f''(x_i) = M_i$:

$$C_i''(x) = \frac{x_i - x}{h_i} M_i + \frac{x - x_{i-1}}{h_i} M_i$$

Integrating $C_i''(x)$ twice we get,

$$C_i(x) = \int \left(\int C_i''(x) dx \right) dx = \frac{x_i - x}{6h_i} M_{i-1} + \frac{x - (x_{i-1})^3}{6h_i} M_i + c_i x + d_i = y_i$$

As $C_i(x_{i-1}) = y_{i-1}$ and $C_i(x_i) = y_i$, we have:

$$C_i(x_{i-1}) = \frac{h_i^2}{6} M_{i-1} + c_i x_{i-1} + d_i = y_{i-1}$$

$$c_i(x_i) = \frac{h_i^2}{6} M_i + c_i x_i + d_i = y_i$$

Solving these two equations we get the two coefficients c_i and d_i:

$$c_i = \frac{y_i - y_{i-1}}{h_i} - \frac{h_i}{6}(M_i - M_{i-1})$$

$$d_i = \frac{x_i y_{i-1} - x_{i-1} y_i}{h_i} - \frac{h_i}{6}(x_i M_{i-1} - x_{i-1} M_i)$$

Substituting them back into $C_i(x)$ and rearranging the terms we get,

$$C_i(x) = \frac{(x_i - x)^3}{6h_i} M_{i-1} + \frac{(x - x_{i-1})^3}{6h_i} M_i + \left(\frac{y_i - y_{i-1}}{h_i} - \frac{h_i}{6} (M_i - M_{i-1}) \right) x + \frac{x_i y_{i-1} - x_{i-1} y_i}{h_i} - \frac{h_i}{6} (x_i M_{i-1} - x_{i-1} M_i)$$

$$\Rightarrow C_i(x) = \frac{(x_i - x)^3}{6h_i} M_{i-1} + \frac{(x - x_{i-1})^3}{6h_i} M_i + \left(\frac{y_{i-1}}{h_i} - \frac{M_{i-1} h_i}{6} \right) (x_i - x) + \left(\frac{y_i}{h_i} - \frac{M_i h_i}{6} \right) (x - x_{i-1})$$

To find M_i $(i = 1, \cdots, n-1)$, we take derivative of $C_i(x)$ and rearrange terms to get

$$C_i'(x) = -\frac{(x_i - x)}{2h_i} M_{i-1} + \frac{(x - x_{i-1})^2}{2h_i} M_i + \frac{y_i - y_{i-1}}{h_i} - \frac{h_i}{6}(M_i - M_{i-1})$$

which, when evaluated at $x = x_i$ and $x = x_{i-1}$, becomes:

$$C_i'(x_i) = \frac{h_i}{3} M_i + \frac{y_i - y_{i-1}}{h_i} + \frac{h_i}{6} M_{i-1} = \frac{h_i}{6} (2M_i + M_{i-1}) + f[x_{i-1}, x_i]$$

$$C_i'(x_{i-1}) = -\frac{h_i}{3} M_{i-1} + \frac{y_i - y_{i-1}}{h_i} - \frac{h_i}{6} M_i = -\frac{h_i}{6} (2M_{i-1} - M_i) + f[x_{i-1}, x_i]$$

Replacing i by $i+1$ in the second equation, we also get

$$C_{i+1}'(x_i) = -\frac{h_{i+1}}{3} M_i + \frac{y_{i+1} - y_i}{h_{i+1}} - \frac{h_{i+1}}{6} M_{i+1}$$

To satisfy $C_i'(x_i) = C_{i+1}'(x_i)$, we equate the above to the first equation to get:

$$\frac{h_i}{3} M_i + \frac{y_i - y_{i-1}}{h_i} + \frac{h_i}{6} M_{i-1} = -\frac{h_{i+1}}{3} M_i + \frac{y_{i+1} - y_i}{h_{i+1}} - \frac{h_{i+1}}{6} M_{i+1}$$

Multiplying both sides by $6/(h_{i+1} + h_i) = 6/(x_{i+1} - x_{i-1})$ and rearranging, we get:

$$\frac{h_i}{h_{i+1} + h_i} M_{i-1} + 2M_i + \frac{h_{i+1}}{h_{i+1} + h_i} M_{i+1} = \frac{6}{h_{i+1} + h_i} \left(\frac{y_{i+1} - y_i}{h_{i+1}} - \frac{y_i - y_{i-1}}{h_i} \right) = 6f[x_{i-1}, x_i, x_{i+1}]$$

We can rewrite the equation above as,

$$\mu_i M_{i-1} + 2M_i + \lambda_i M_{i+1} = 6f\left[x_{i-1}, x_i, x_{i+1}\right], \quad (i = 1, \cdots, n-1)$$

where,

$$\mu_i = \frac{h_i}{h_{i+1} + h_i}$$

$$\lambda_i = \frac{h_{i+1}}{h_{i+1} + h_i} = 1 - \mu_i$$

Here we have $n-1$ equations but $n+1$ unknowns $M_0, \cdots, M_n$. To obtain these unknowns, we need to get two additional equations based on certain assumed boundary conditions.

- Assume the first order derivatives at both ends $f'(x_0)$ and $f'(x_n)$ are known. Specially $f'(x_0) = f'(x_n) = 0$ is called clamped boundary condition. At the front end, we set:

  $$C_1'(x_0) = -\frac{h_1}{3}M_0 + \frac{y_1 - y_0}{h_1} - \frac{h_1}{6}M_1 = -\frac{h_1}{3}M_0 - \frac{h_1}{6}M_1 + f\left[x_0, x_1\right] = f'(x_0)$$

  Multiplying $-6/h_1 = -6/(x_1 - x_0)$ we get,

  $$\frac{6}{x_1 - x_0}\left[f\left[x_0, x_1\right] - f'(x_0)\right] = 6f\left[x_0, x_0, x_1\right]$$

  Similarly, at the back end, we also set,

  $$C_n'(x_n) = \frac{h_n}{3}M_n + \frac{y_n - y_{n-1}}{h_n} + \frac{h_n}{6}M_{n-1} = \frac{h_n}{3}M_n + \frac{h_n}{6}M_{n-1} + f\left[x_{n-1}, x_n\right] = f'(x_n)$$

  Multiplying $6/h_n = 6/(x_n - x_{n-1})$ we get,

  $$2M_n + M_{n-1} = \frac{6}{x_n - x_{n-1}}\left[f'(x_n) - f\left[x_{n-1,x_n}\right]\right] = 6f\left[x_{n-1}, x_n, x_n\right]$$

  So, We can write,

  $$\begin{bmatrix} 2 & 1 & & & \\ \mu_1 & 2 & \lambda_1 & & \\ & \ddots & \ddots & \ddots & \\ & & \mu_{n-1} & 2 & \lambda_{n-1} \\ & & & 1 & 2 \end{bmatrix} \begin{bmatrix} M_0 \\ M_1 \\ \vdots \\ M_{n-1} \\ M_n \end{bmatrix} = 6 \begin{bmatrix} f[x_0, x_0, x_1] \\ f[x_0, x_1, x_2] \\ \vdots \\ f[x_{n-2}, x_{n-1}, x_n] \\ f[x_{n-1}, x_n, x_n] \end{bmatrix}$$

- Alternatively, we can also assume $f''(x_0)$ and $f''(x_n)$ are known. Specially $f''(x_0) = f''(x_n) = 0$ is called natural boundary condition. Now we can simply get $M_0 = f''(x_0)$ and $M_n = f''(x_n)$ and solve the following system for the $n+1$ unknowns $M_0, \cdots, M_n$:

  $$\begin{bmatrix} 1 & 0 & & & \\ \mu_1 & 2 & \lambda_1 & & \\ & \ddots & \ddots & \ddots & \\ & & \mu_{n-1} & 2 & \lambda_{n-1} \\ & & & 0 & 1 \end{bmatrix} \begin{bmatrix} M_0 \\ M_1 \\ \vdots \\ M_{n-1} \\ M_n \end{bmatrix} = \begin{bmatrix} f''(x_0) \\ 6f[x_0, x_1, x_2] \\ \vdots \\ 6f[x_{n-2}, x_{n-1}, x_n] \\ f''(x_n) \end{bmatrix}$$

Problem 03

Numerical Differentiation and Integration:

(a) Using Taylor's expansions determine what does

$$\frac{f(x+\Delta x)-2f(x)+f(x-\Delta x)}{\Delta x^2}$$

approximate, and to what order?

(b) Using the approximation above (in number (a)) as a basis, use Richardson extrapolation to find a higher order method.

(c) Choosing the coefficients a,b,c and e wisely, of what order can we make the approximation:

$$f'(x) \approx af(x+2\Delta x)+bf(x+\Delta x)+cf(x)+df(x-\Delta x)+ef(x-2\Delta x)$$

(d) Using the approximation above (in number (c)) as a basis, use Richardson extrapolation to find higher order method.

(e) Write a little code to evaluate $\int_0^1 (x^2-3x+1)dx$ suing the trapezoid rule. Show the results using one interval, and the results using 5 and 10 sub-intervals. Use the errors in these results to explain the error formula and show the order.

(f) Write a little code to evaluate $\int_0^1 (x^2-3x+1)dx$ using higher order rules,Show the results using one interval, and the results using 5 and 10 sub-intervals. Use the errors in these results to explain the error formula and show the order.

(g) Write a little code to evaluate $\int_0^1 cos^2(x)$ using any rule, show the results using one intervals, and the results using 5 and 10 little intervals. Explain using the error formula.

Solution 3(a)

From Taylor's expansion,

$$f(x+\Delta x)=f(x)+f'(x)\Delta x+\frac{1}{2}f''(x)\Delta x^2+\frac{1}{6}f'''(x)\Delta x^3+\cdots \tag{3}$$

$$f(x-\Delta x)=f(x)-f'(x)\Delta x+\frac{1}{2}f''(x)\Delta x^2-\frac{1}{6}f'''(x)\Delta x^3+\cdots \tag{4}$$

(3) + (4),

$$\begin{aligned} f(x+\Delta x)+f(x-\Delta x) &= 2f(x)+f''(x)+\Delta x^2 \\ \Rightarrow f''(x)\Delta x^2 &= f(x+\Delta x)+f(x-\Delta x)-2f(x) \\ \Rightarrow f''(x) &= \frac{f(x+\Delta x)-2f(x)+f(x-\Delta x)}{\Delta x^2} \end{aligned}$$

This represent the central difference formula of order 2.

Solution 3(b)

From the Richardson Extrapolation, the first order $N_1(h)$ and second order $N_2(h)$. Suppose, $N_1(h)$ is the 2^{nd} order formula, where h is step size.
So, the unknown value,

$$M = N_1(h)+k_1h^2+k_2h^4+k_3h^6+\cdots \tag{5}$$

if $h = \frac{h}{2}$,

$$M = N_1(h/2) + K_1(h^2/4) + k_2(h^4/16) + k_3(h^6/64) + \cdots \quad (6)$$

$Eqn(6) \times 4 - Eqn(5)$,

$$M = \frac{1}{3}\left[4N_1(\frac{h}{2}) - N_1(h)\right] + \frac{k_2}{3}\left(\frac{h^4}{4} - h^4\right) + \frac{k_3}{3}\left(\frac{h^6}{16} - h^6\right) + \cdots$$

Define,

$$N_2(h) = \frac{1}{3}\left[4N_1(h/2) - N_1(h)\right] = N_1(\frac{h}{2}) + \frac{1}{3}\left[N_1(h/2) - N_1(h)\right] \quad (7)$$

Then Eqn 7 is an $O(h^4)$ approximation formula for M:

$$M = N_2(h) - k_2\frac{h^4}{4} - k_3\frac{5h^6}{16} + \cdots$$

Solution 3(c)

Given function,

$$f'(x) \approx af(x+\Delta x) + bf(x+\Delta x) + cf(x) + df(x-\Delta x) + ef(x-2\Delta x)$$

From, Taylor's Polynomials,

$$f(x+\Delta x) = f(x) + f'(x)\Delta x + \frac{1}{2}f''(x)\Delta x^2 + \frac{1}{6}f'''(x)\Delta x^3 + \cdots \quad (8)$$

$$f(x-\Delta x) = f(x) - f'(x)\Delta x + \frac{1}{2}f''(x)\Delta x^2 - \frac{1}{6}f'''(x)\Delta x^3 + \cdots \quad (9)$$

$$f(x+2\Delta x) = f(x) + 2f'(x)\Delta x + 2f''(x)\Delta x^2 + \frac{4}{3}f'''(x)\Delta x^3 + \cdots \quad (10)$$

$$f(x-2\Delta x) = f(x) - 2f'(x)\Delta x + 2f''(x)\Delta x^2 - \frac{4}{3}f'''(x)\Delta x^3 + \cdots \quad (11)$$

$Eqn(8) - Eqn(9)$,

$$f(x+\Delta x) - f(x-\Delta x) = 2f''(x)\Delta x + \frac{1}{3}f'''(x)\Delta x^3 + \cdots \quad (12)$$

$Eqn(10) - Eqn(11)$,

$$f(x+2\Delta x) - f(x+2\Delta x) = 4f'(x)\Delta x + \frac{8}{3}f'''(x)\Delta x^3 + \cdots \quad (13)$$

Now, $Eqn(12) \times 8 + Eqn(13)$,

$$8f(x+\Delta x) - 8f(x-\Delta x) - f(x+2\Delta x) + f(x-2\Delta x) = 12f'(x)$$

$$\Rightarrow f'(x) = \frac{-f(x+2\Delta x) + 8f(x+\Delta x) - 8f(x-\Delta x) + f(x-2\Delta x)}{12}$$

So, $a = -1, b = 8, c = 0, d = -8, e = 1$ with order 4.

Solution 3(d)

From Richardson extrapolation, First order $N_1(h)$, second order $N_2(h)$. In problem $3(c)$ the order is 4. So, the unknown value of M,

$$M = N_1(2h) + k_4(2h)^4 + k_5(2h)^5 + \cdots \quad (14)$$

Lets, assume the step size is $2h = h$, then, $Eqn(14)$ becomes,

$$M = N_1(h) + k_4(h)^4 + k_5(h)^5 + \cdots \quad (15)$$

$Eqn(15) \times 16 - Eqn(14)$,

$$16M - M = 16N_1(h) - N_1(2h) + 16k_5(h)^5 - k_5h^5 + \cdots$$
$$M = \frac{1}{15}\left[16N_1(h) - N_1(2h)\right] + \frac{1}{15}\left[16k_5h^5 - k_5h^5\right] + \cdots$$

So,

$$N_2(h) = \frac{1}{15}\left[16N_1(h) - N_1(2h)\right]$$

is 5^{th} order.

Solution 3(e)

```
clc;
clear;
f = @(x) x.^2. -3.*x + 1;
a = 0;
b = 1;

fint_1_trap = vpa(integral(f,a,b),3)
trap = vpa(cotes(f,a,b,2,2),3);
error_trap = vpa(abs(fint_1_trap -trap),3)

for i = 1:5
    m1 = a + (i-1)/5;
    m2 = m1 + 1/5;
    trap_5(i) = cotes(f,m1,m2,2,2);
end

trap_5 = vpa(sum(trap_5),3)
error_trap_5 = vpa(abs(fint_1_trap-trap_5),3)

for i=1:10
    l1 = a + (i - 1)/10;
    l2 = l1 + 1/10;
    trap_10(i) = cotes(f,l1,l2,2,2);
end

trap_10 = vpa(sum(trap_10),3)
error_trap_10 = vpa(abs(fint_1_trap - trap_10),3)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% fint_1_trap =
%
% -0.167
%
%
% error_trap =
%
% 0.0417
%
%
% trap_5 =
```

```
%
% -0.165
%
%
% error_trap_5 =
%
% 0.00167
%
%
% trap_10 =
%
% -0.166
%
%
% error_trap_10 =
%
% 4.17e-4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

For Trapizoidal rule, error formula is

$$\frac{(b-a)|f''(x)|}{12n^2}$$

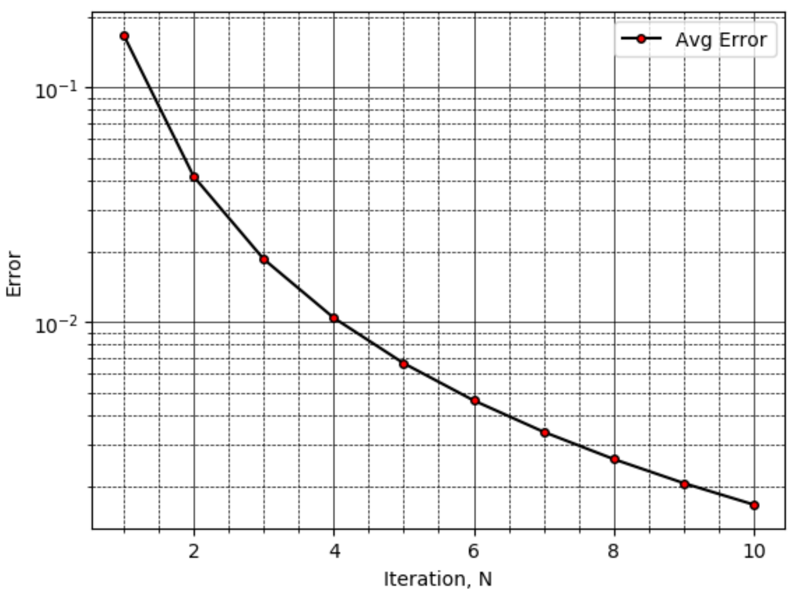


Figure 9: Error vs Iteration: Trapezoidal Method: $f(x) = x^2 - 3x + 1$

Solution 3(f)

```
clc;
clear;
f = @(x) x.^2. -3.*x + 1;
a = 0;
```

```
b = 1;

fint_1_trap = vpa(integral(f,a,b),3)
s = vpa(cotes(f,a,b,2,2),3);
error_simpson = vpa(abs(fint_1_trap - s),3)

for i = 1:5
    m1 = a + (i-1)/5;
    m2 = m1 + 1/5;
    s_5(i) = cotes(f,m1,m2,3,3);
end

s_5 = vpa(sum(s_5),3)
error_s_5 = vpa(abs(fint_1_trap-s_5),3)

for i=1:10
    l1 = a + (i - 1)/10;
    l2 = l1 + 1/10;
    s_10(i) = cotes(f,l1,l2,3,3);
end

s_10 = vpa(sum(s_10),3)
error_s_10 = vpa(abs(fint_1_trap - s_10),3)

%%%%%%%%%%%%%%%%%%%%%%%%%
% fint_1_trap =
%
% -0.167
%
%
% error_simpson =
%
% 0.0417
%
%
% s_5 =
%
% -0.167
%
%
% error_s_5 =
%
% 0.0
%
%
% s_10 =
%
% -0.167
%
%
% error_s_10 =
%
```

```
% 0.0
%%%%%%%%%%%%%%%%%%%%%%%%
```

Solution 3(g)

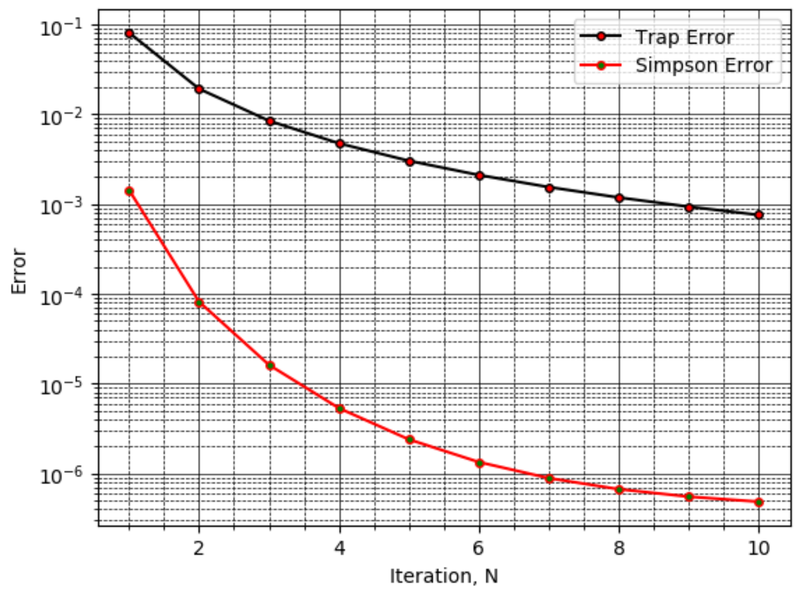


Figure 10: Error vs Iteration: $f(x) = cos^2(x)$

```
clc;
clear;
f = @(x) cos(x).^2;
a = 0;
b = 1;

fint_1_trap = vpa(integral(f,a,b),3)
s = vpa(cotes(f,a,b,2,2),3);
error_simpson = vpa(abs(fint_1_trap - s),3)

for i = 1:5
    m1 = a + (i-1)/5;
    m2 = m1 + 1/5;
    s_5(i) = cotes(f,m1,m2,3,3);
end

s_5 = vpa(sum(s_5),3)
error_s_5 = vpa(abs(fint_1_trap-s_5),3)

for i=1:10
    l1 = a + (i - 1)/10;
    l2 = l1 + 1/10;
    s_10(i) = cotes(f,l1,l2,3,3);
end

```

```
s_10 = vpa(sum(s_10),3)
error_s_10 = vpa(abs(fint_1_trap - s_10),3)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%5

% fint_1_trap =
%
% 0.727
%
%
% error_simpson =
%
% 0.0193
%
%
% s_5 =
%
% 0.727
%
%
% error_s_5 =
%
% 1.26e-7
%
%
% s_10 =
%
% 0.727
%
%
% error_s_10 =
%
% 7.9e-9
```

Problem 04

Fourier series expansions: Using the function,

$$f(x) = \begin{cases} 1+x, & -1 \le x \le 0 \\ x, & 0 \le x \le 1 \end{cases}$$

(a) Plot the magnitude of the Fourier coefficients and show how they decay.

(b) Show the performance of the truncated Fourier series in approximating this function, using an increasing number of terms, Comment on the effect of adding terms. Print out graphs and comment.

(c) Using different filters, show how these results change.

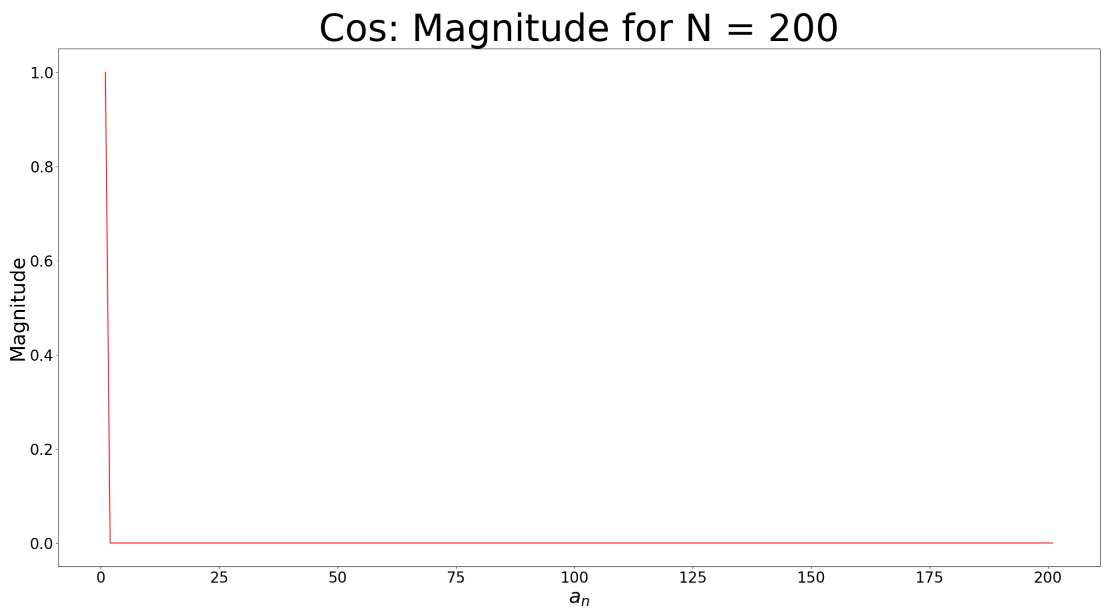
Solution 4(a)

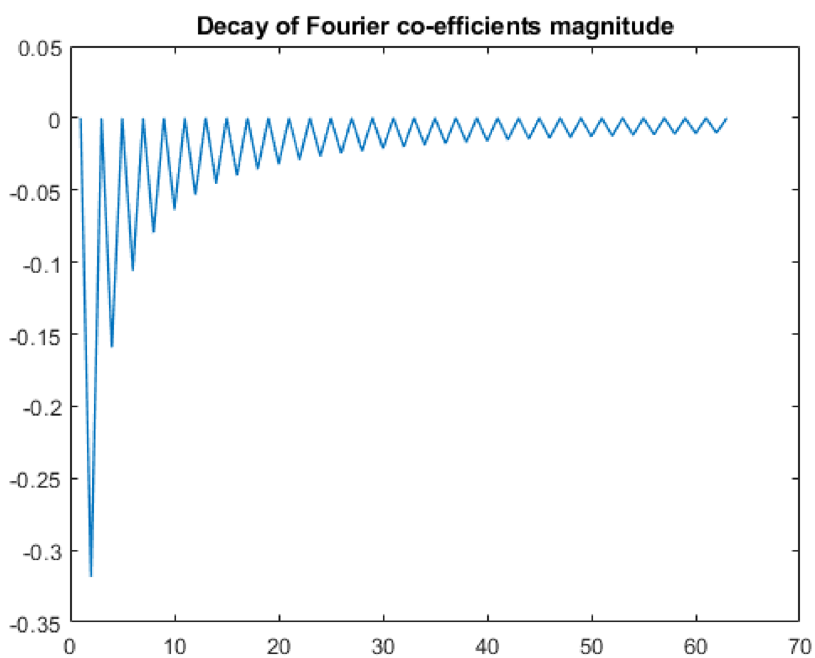
We know,

$$S_n(x) = \frac{a_0}{2} + \sum_{n=1}^{N} \left[a_n cos\left(\frac{2\pi nx}{P}\right) + b_n sin\left(\frac{2\pi nx}{P}\right) \right]$$

$$a_n = \frac{2}{P}\int_P f(x) \cdot cos(2\pi x\frac{n}{P})dx$$

$$b_n = \frac{2}{P}\int_P f(x) \cdot sin(2\pi x\frac{n}{P})dx$$





Solution 4(b) & 4(c)

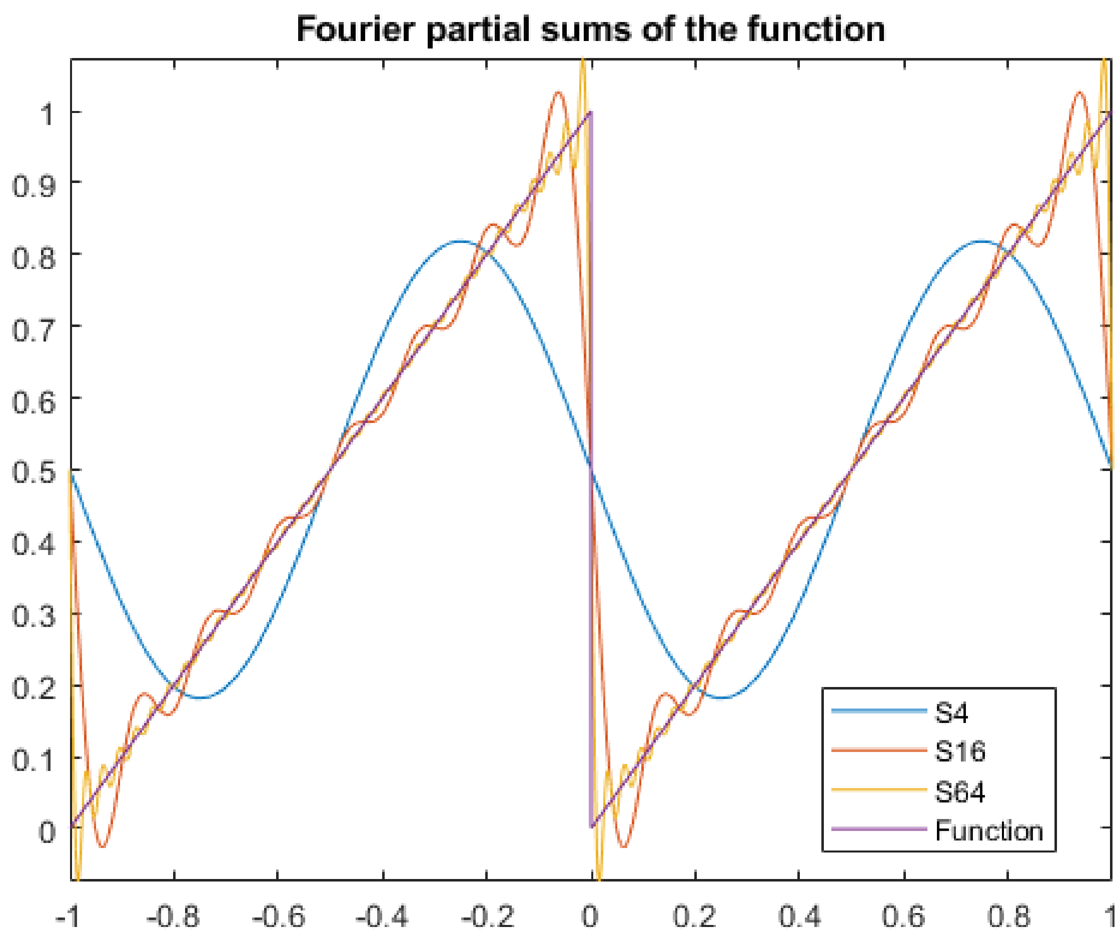


Figure 11: Fourier approximation for the function.

• Observation: With increasing number of term, in the jump there is increasing number of jumps.

Solution 4(c)

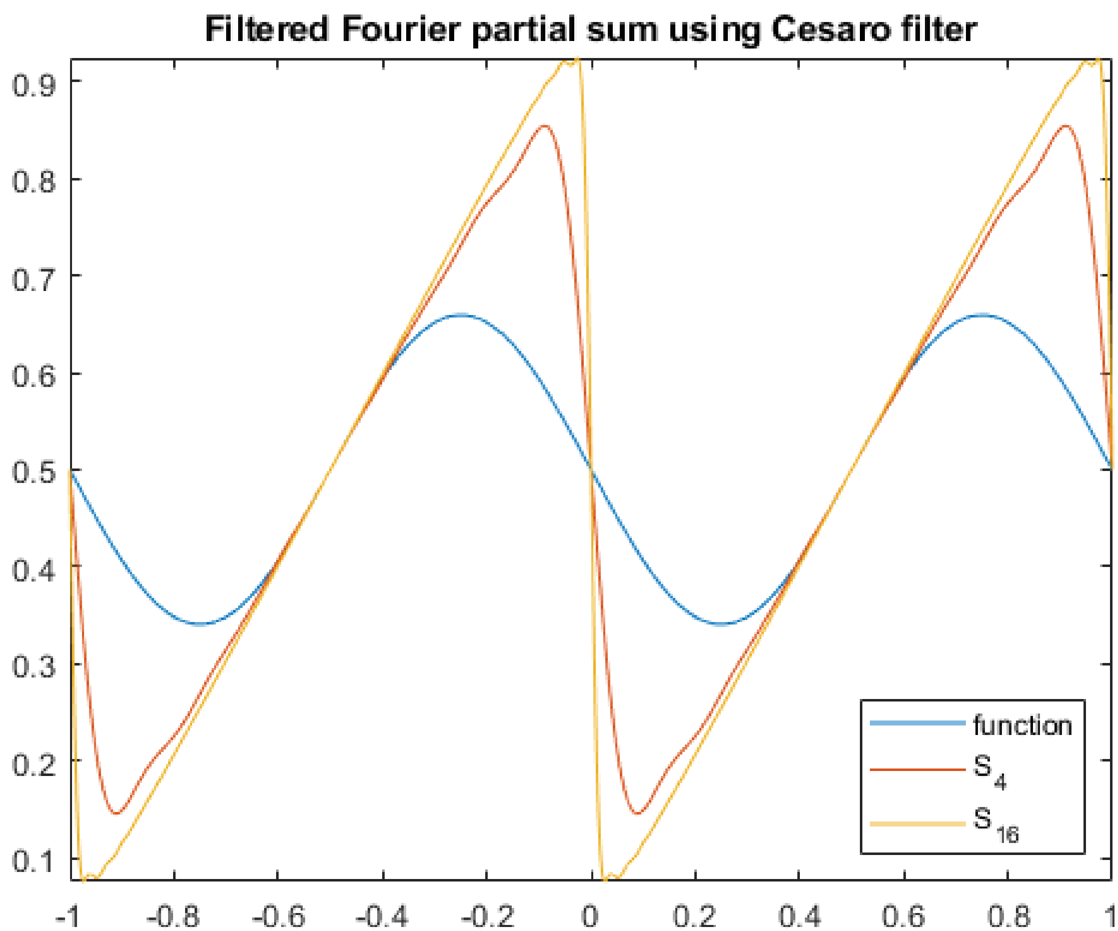


Figure 12: Cesaro Filter

- Since this is a first order filter, the accuracy away from the jump is not very good.
- This filter eliminates the oscillations and produces heavily smeared approximation of the original function

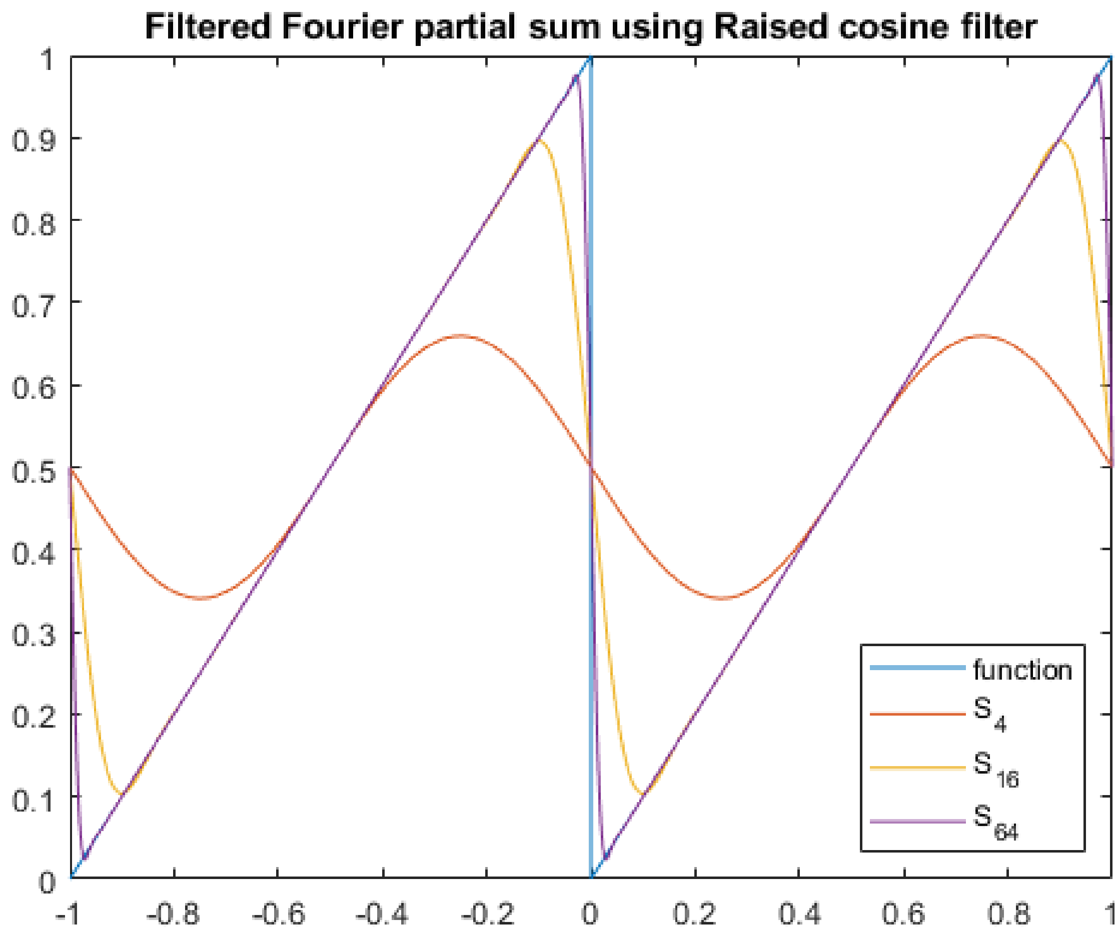


Figure 13: Raised Cone Filter

- This filter reduces the oscillations, but there is some the overshoot near the jump.
- The accuracy is increased away from the jump, since this is second order filter original function.

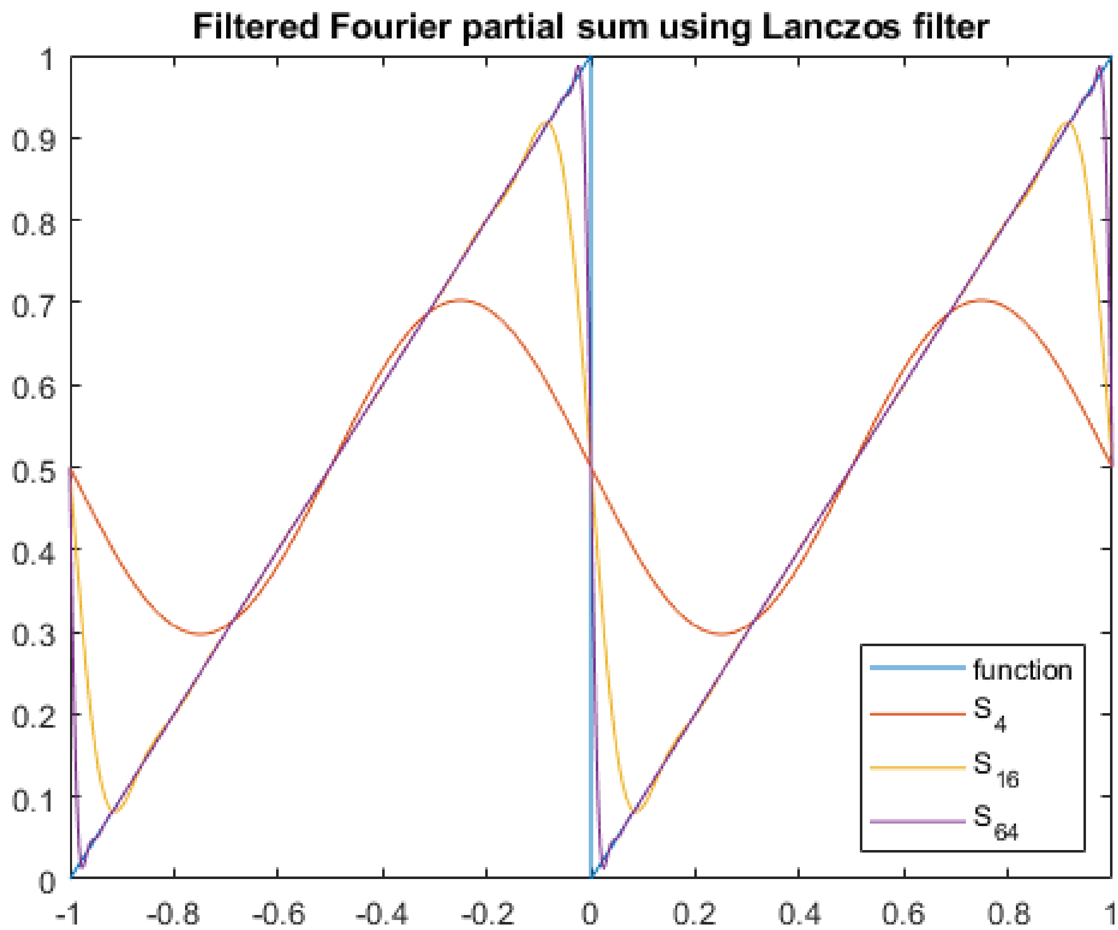


Figure 14: Lanczos Filter

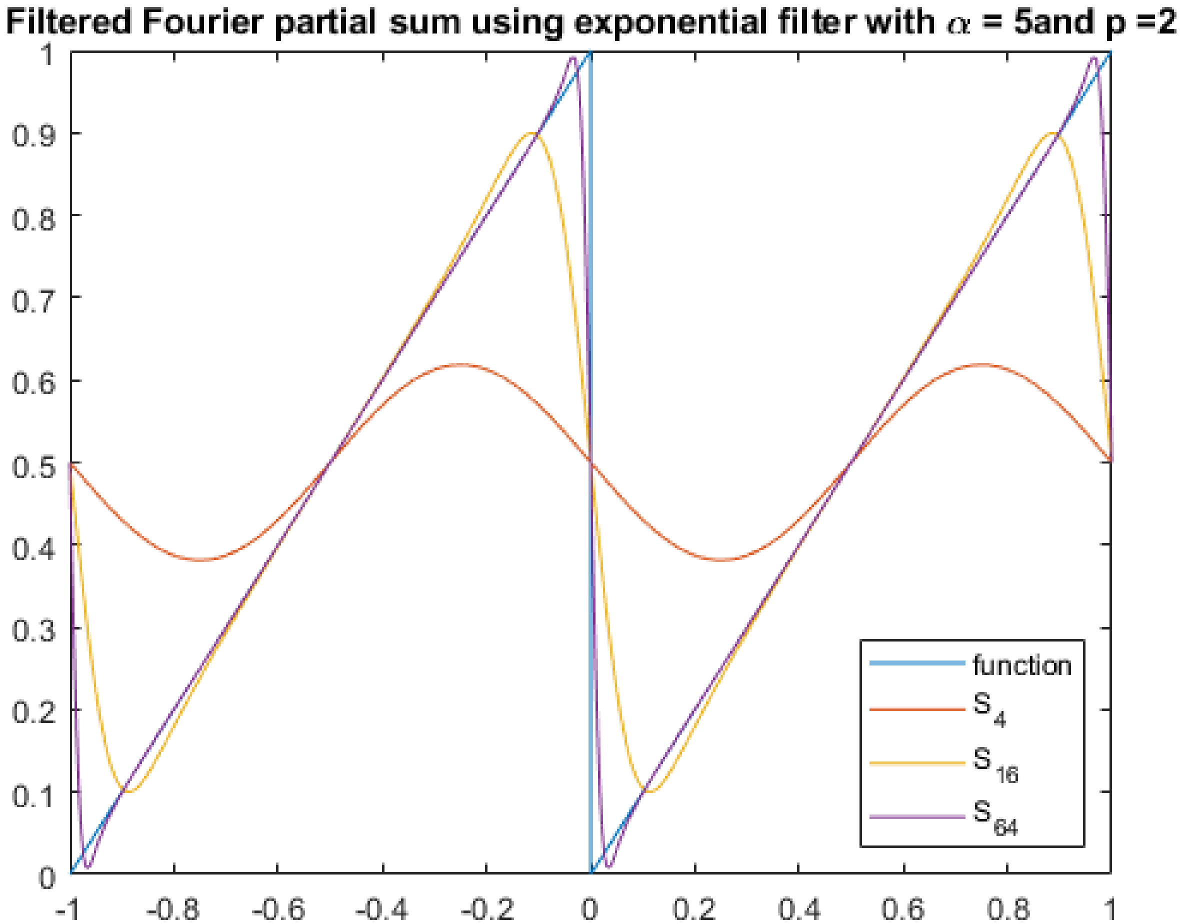


Figure 15: Exponential Filter P=2,$\alpha = 5$

1. Observation
   (a) This filter recover p-order convergence in N away from the jump
   (b) Exponential accuracy can be achieved away from the jump by choosing p as a linear function of N.
2. Comparison of results from all filters:
   (a) Second order filters shows good accuracy away from the jump.
   (b) No filters here can resolve the overshoot issue near the jump.
   (c) Exponential filter is better among all these filter since this filter can recover the convergence of any desired order.

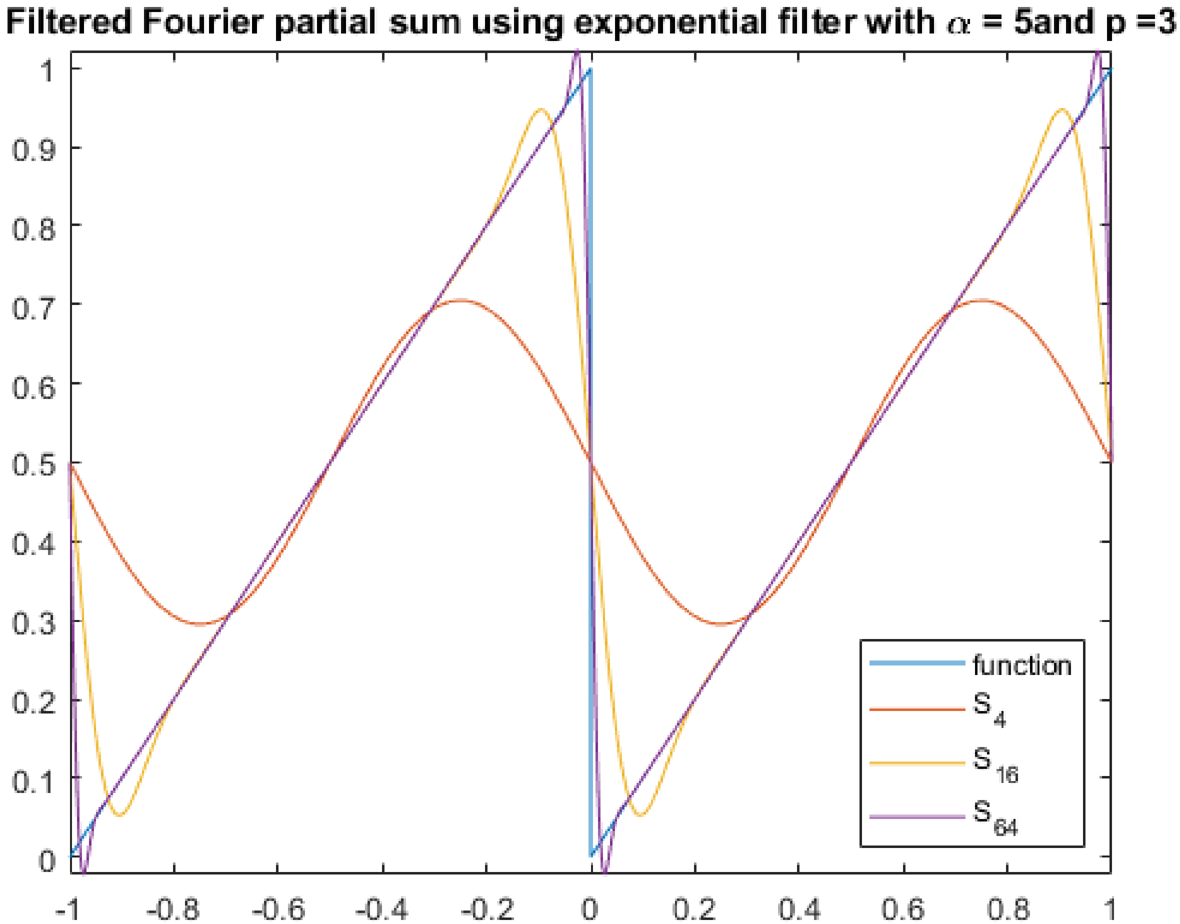


Figure 16: Exponential Filter P=3,$\alpha = 5$

```
clear all,clc
% Problem 4a;
fun1 = @(x) 1+x;
fun2 = @(x) x;

xa1 = -1;
xb1 = 0;
xa2 = 0;
xb2 = 1;
n = 64;
a = zeros(1,n+1);
b = zeros(1,n-1);
a(1) = integral(fun1,xa1,xb1)+integral(fun2,xa2,xb2);


for k=1:n-1
funck1 = @(x) fun1(x).*cos(2*pi.*k/2.*x) ;
```

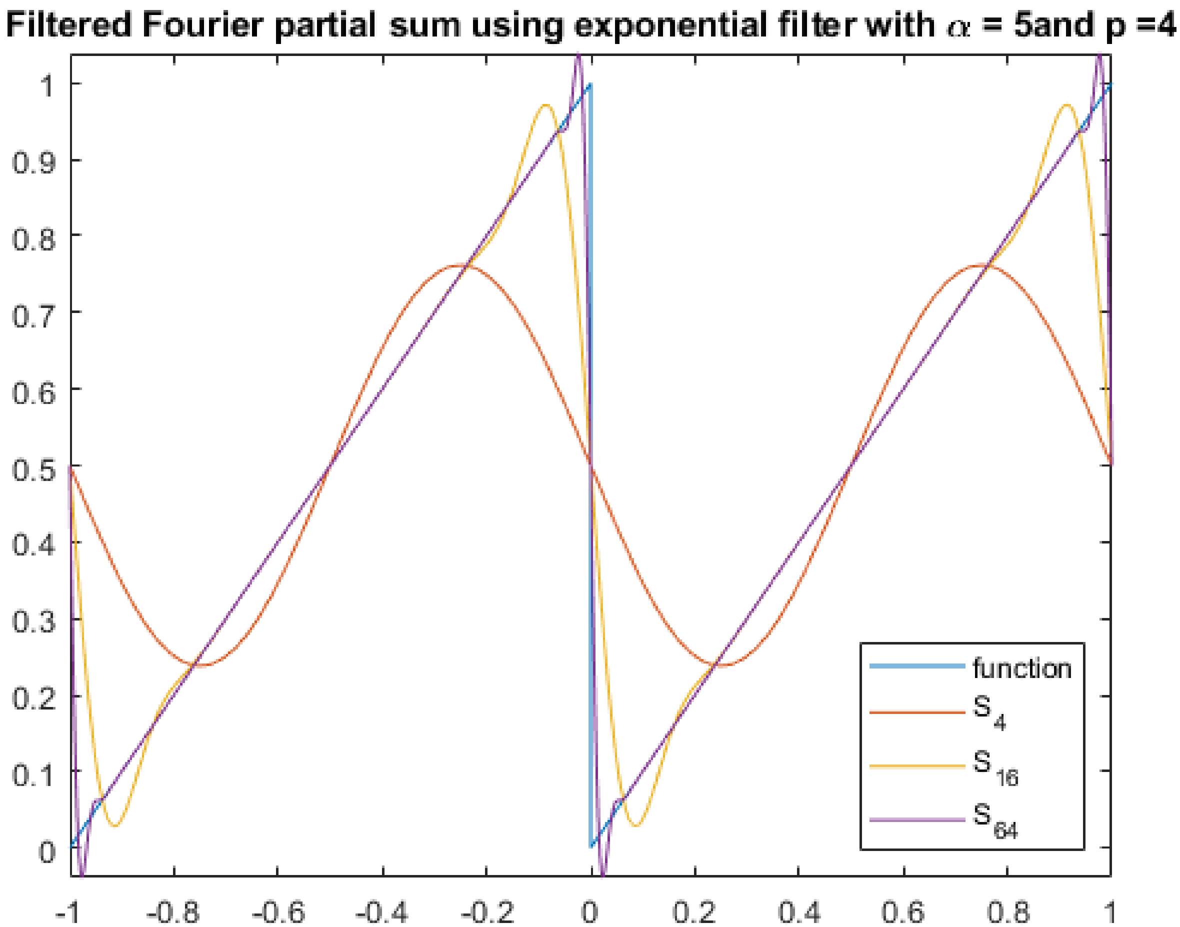


Figure 17: Exponential Filter P=4,$\alpha = 5$

```
funck2=@(x) fun2(x).*cos(2*pi.*k/2.*x)
a(k+1) = integral(funck1,xa1,xb1)+integral(funck2,xa2,xb2);
funsk1 = @(x) fun1(x).*sin(2*pi.*k/2.*x);
funsk2 = @(x) fun2(x).*sin(2*pi.*k/2.*x);
b(k) = integral(funsk1,xa1,xb1)+integral(funsk2,xa2,xb2);
end
funck1 = @(x) fun1(x).*cos(2*pi.*k/2.*x) ;
funck2=@(x) fun2(x).*cos(2*pi.*k/2.*x)
a(n+1) = integral(funck1,xa1,xb1)+integral(funck2,xa2,xb2);

max(abs(a));
figure (1)
plot(b,'linewidth',1)
title('Decay of Fourier co-efficients magnitude');

%Part b
%Show the performance of the truncated Fourier series in approximating
```

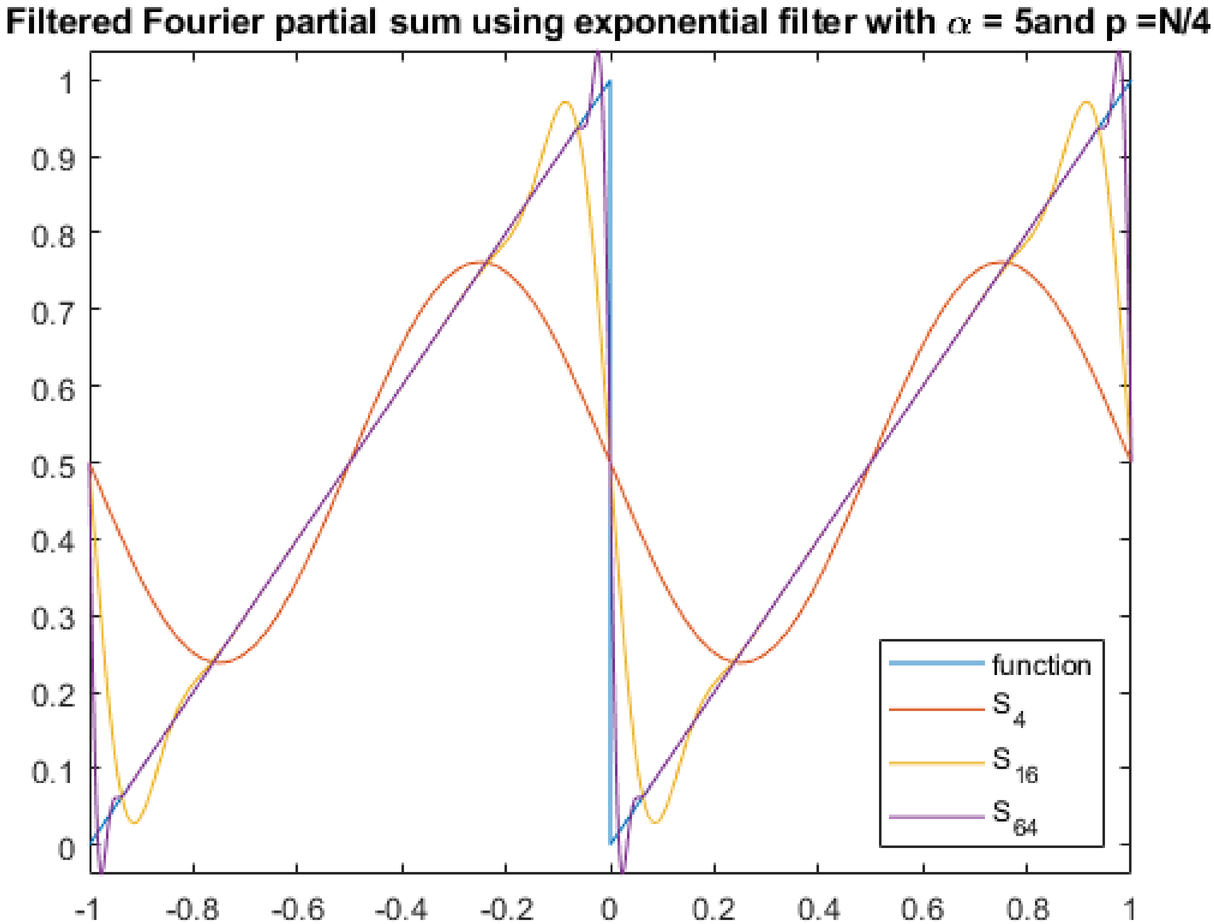


Figure 18: Exponential Filter P=2,$\alpha = N/4$

```
%this function, using an increasing number of terms
k=1;
funsk = @(x) a(1)/2.*cos(0.*x)+ a(k+1)*cos(2*pi.*k/2.*x);
funt = funsk;
kp=1;

for k=2:n
funsk = @(x) funt(x) + a(k+1)*cos(2*pi.*k/2.*x) + b(k-1)*sin(2*pi.*(k-1)/2.*x);
if pow2(2*floor(log(k)/log(4))) == k
figure(2)
fplot(funsk,[xa1,xb2]);hold on
%
kp = kp+1;
legendInfo{kp+1} = ['S_{',num2str(k),'}'];
end
funt = funsk;
end
```

```
f = @(x) [(1+x).*(-1<=x & x<=0) + (x).*(0<x & x<=1)];
fplot(f,[xa1,xb2]);hold off

legend('S4','S16','S64','Function')
title('Fourier partial sums of the step function');

% plot of filtered Fourier partial sums
% part-a: Cesaro filter
sigmaf = @(x) 1.-x;
figure(10)
f = @(x) [(1+x).*(-1<=x & x<=0) + (x).*(0<x & x<=1)];
fplot(f,[xa1,xb2]);hold off

legendInfo{1} = ['function'];

for kn = 1:3
NN = 4^kn;
k=1;
funsk = @(x) a(1)*sigmaf(0)./2.*cos(0.*x)+ a(k+1)*sigmaf(k/NN)*cos(2*pi.*k/2.*x);
funt = funsk;
for k=2:NN
funsk = @(x) funt(x) + a(k+1)*sigmaf(k/NN)*cos(2*pi.*k/2.*x)+ b(k-1)*sigmaf((k-1)/NN
    )*sin(2*pi.*(k-1)/2.*x);
funt = funsk;
end

figure(10)
fplot(funsk,[xa1,xb2]);hold on
legendInfo{kn+1} = ['S_{',num2str(k),'}'];
end

figure(10)
hold off
legend(legendInfo,'Location','southeast');
title('Filtered Fourier partial sum using Cesaro filter')% Cesaro filter

sigmaf = @(x) 0.5*(1.+cos(pi*x));
figure(21)

f = @(y) [(1+y).*(-1<=y & y<=0) + (y).*(0<y & y<=1)];
fplot(f,[xa1,xb2]);hold on
legendInfo{1} = ['function'];

for kn = 1:3
NN = 4^kn;
k=1;
funsk = @(x) a(1)*sigmaf(0)./2.*cos(0.*x)+ a(k+1)*sigmaf(k/NN)*cos(2*pi.*k/2.*x);
funt = funsk;
for k=2:NN
funsk = @(x) funt(x) + a(k+1)*sigmaf(k/NN)*cos(2*pi.*k/2.*x)+ b(k-1)*sigmaf((k-1)/NN
    )*sin(2*pi.*(k-1)/2.*x);
```

```
funt = funsk;
end

figure(21)
fplot(funsk,[xa1,xb2]);hold on
legendInfo{kn+1} = ['S_{',num2str(k),'}'];

end
figure(21)
hold off
legend(legendInfo,'Location','southeast');
title('Filtered Fourier partial sum using Raised cosine filter');

% Lanczos filter
sigmaf = @(x) (x==0) + sin(pi*x)./pi./ [Inf(x==0), x(x~=0)];
figure(31)
f = @(y) [(1+y).*(-1<=y & y<=0) + (y).*(0<y & y<=1)];
fplot(f,[xa1,xb2]);hold on

legendInfo{1} = ['function'];
hold on;
for kn = 1:3
NN = 4^kn;
k=1;
funsk = @(x) a(1)*sigmaf(0)./2.*cos(0.*x)+ a(k+1)*sigmaf(k/NN)*cos(2*pi.*k/2.*x);
funt = funsk;
for k=2:NN
funsk = @(x) funt(x) + a(k+1)*sigmaf(k/NN)*cos(2*pi.*k/2.*x)+ b(k-1)*sigmaf((k-1)/NN
    )*sin(2*pi.*(k-1)/2.*x);
funt = funsk;
end

figure(31)
fplot(funsk,[xa1,xb2]);hold on
legendInfo{kn+1} = ['S_{',num2str(k),'}'];

end
figure(31)
hold off
legend(legendInfo,'Location','southeast');
title('Filtered Fourier partial sum using Lanczos filter')


% exponential filter
x0 = 0.1; % threshold of the filter
p = 2; % order of the filter
alpha = 5; % strongness of the filter
sigmaf = @(x) (x<=x0) + (x>x0)*exp(-alpha.*((x-x0)/(1-x0))^p);
figure(41)
f = @(y) [(1+y).*(-1<=y & y<=0) + (y).*(0<y & y<=1)];
fplot(f,[xa1,xb2]);hold on

```

```
legendInfo{1} = ['function'];

for kn = 1:3
NN = 4^kn;
k=1;
funsk = @(x) a(1)*sigmaf(0)./2.*cos(0.*x)+ a(k+1)*sigmaf(k/NN)*cos(2*pi.*k/2.*x);
funt = funsk;
for k=2:NN
funsk = @(x) funt(x) + a(k+1)*sigmaf(k/NN)*cos(2*pi.*k/2.*x)+ b(k-1)*sigmaf((k-1)/NN
    )*sin(2*pi.*(k-1)/2.*x);
funt = funsk;
end

figure(41)
fplot(funsk,[xa1,xb2]);hold on
legendInfo{kn+1} = ['S_{',num2str(k),'}'];

end

figure(41)
hold off
legend(legendInfo,'Location','southeast');
title(['Filtered Fourier partial sum using exponential filter with \alpha = ',
    num2str(alpha),'and p =2'])

% exponential filter
x0 = 0.1; % threshold of the filter
p = 3; % order of the filter
alpha = 5; % strongness of the filter
sigmaf = @(x) (x<=x0) + (x>x0)*exp(-alpha.*((x-x0)/(1-x0))^p);
figure(51)
f = @(y) [(1+y).*(-1<=y & y<=0) + (y).*(0<y & y<=1)];
fplot(f,[xa1,xb2]);hold on

legendInfo{1} = ['function'];

for kn = 1:3
NN = 4^kn;
k=1;
funsk = @(x) a(1)*sigmaf(0)./2.*cos(0.*x)+ a(k+1)*sigmaf(k/NN)*cos(2*pi.*k/2.*x);
funt = funsk;
for k=2:NN
funsk = @(x) funt(x) + a(k+1)*sigmaf(k/NN)*cos(2*pi.*k/2.*x)+ b(k-1)*sigmaf((k-1)/NN
    )*sin(2*pi.*(k-1)/2.*x);
funt = funsk;
end

figure(51)
fplot(funsk,[xa1,xb2]);hold on
legendInfo{kn+1} = ['S_{',num2str(k),'}'];

end
```

```
figure(51)
hold off
legend(legendInfo,'Location','southeast');
title(['Filtered Fourier partial sum using exponential filter with \alpha = ',
    num2str(alpha),'and p =3'])


% exponential filter
x0 = 0.1; % threshold of the filter
p = 4; % order of the filter
alpha = 5; % strongness of the filter
sigmaf = @(x) (x<=x0) + (x>x0)*exp(-alpha.*((x-x0)/(1-x0))^p);
figure(61)
f = @(y) [(1+y).*(-1<=y & y<=0) + (y).*(0<y & y<=1)];
fplot(f,[xa1,xb2]);hold on

legendInfo{1} = ['function'];

for kn = 1:3
NN = 4^kn;
k=1;
funsk = @(x) a(1)*sigmaf(0)./2.*cos(0.*x)+ a(k+1)*sigmaf(k/NN)*cos(2*pi.*k/2.*x);
funt = funsk;
for k=2:NN
funsk = @(x) funt(x) + a(k+1)*sigmaf(k/NN)*cos(2*pi.*k/2.*x)+ b(k-1)*sigmaf((k-1)/NN
    )*sin(2*pi.*(k-1)/2.*x);
funt = funsk;
end

figure(61)
fplot(funsk,[xa1,xb2]);hold on
legendInfo{kn+1} = ['S_{',num2str(k),'}'];

end
figure(61)
hold off
legend(legendInfo,'Location','southeast');
title(['Filtered Fourier partial sum using exponential filter with \alpha = ',
    num2str(alpha),'and p =4'])


% exponential filter
x0 = 0.1; % threshold of the filter
%p = 4; % order of the filter
alpha = 5; % strongness of the filter
sigmaf = @(x) (x<=x0) + (x>x0)*exp(-alpha.*((x-x0)/(1-x0))^p);
figure(71)
f = @(y) [(1+y).*(-1<=y & y<=0) + (y).*(0<y & y<=1)];
fplot(f,[xa1,xb2]);hold on

legendInfo{1} = ['function'];
```

```
for kn = 1:3
NN = 4^kn;
p = NN/4; % p is a linear function of NN
k=1;
funsk = @(x) a(1)*sigmaf(0)./2.*cos(0.*x)+ a(k+1)*sigmaf(k/NN)*cos(2*pi.*k/2.*x);
funt = funsk;
for k=2:NN
funsk = @(x) funt(x) + a(k+1)*sigmaf(k/NN)*cos(2*pi.*k/2.*x)+ b(k-1)*sigmaf((k-1)/NN
    )*sin(2*pi.*(k-1)/2.*x);
funt = funsk;
end

figure(71)
fplot(funsk,[xa1,xb2]);hold on
legendInfo{kn+1} = ['S_{',num2str(k),'}'];

end
figure(71)
hold off
legend(legendInfo,'Location','southeast');
title(['Filtered Fourier partial sum using exponential filter with \alpha = ',
    num2str(alpha),'and p =N/4'])
```

References

[1] Class Note: EAS 502 by Dr. Zheng Leslie Chen, Dept. of Mathematics, UMassD

[2] Numerical Analysis, 9th Edition, Richard L. Burden, J. Douglas Faires

[3] wolframalpha.com

[4] Numerical Methods using MATLAB, Abhishek K Gupta, Apress (2014)

[5] A Graduate Introduction to Numerical Methods, Robert M. Corless, Nicolas Fillion,Verlag New York (2013)