HW2

Lifan Yu lifany

September 2023

Question 1

1 - a

Program:

```
import numpy as np
import matplotlib.pyplot as plt

def divided_difference(x, f):
    # x are the known x's
    # f are the known y's
    n = len(f)
    A = np.zeros([n, n])
    # all A_0's are f
    A[:,0] = f
    for i in range(1,n):
        for k in range(n-i):
            A[k][i] = (A[k+1][i-1] - A[k][i-1])/(x[i+k]-x[k])
    return A

def interpolate(new_x, A):
    # new_x is the x at which to plot
    A = A[0]
    n = len(A)-1
    poly = A[n]
    for j in range(1, n+1):
      poly = poly*(new_x - x[n-j]) + A[n-j]
    return poly
```

Test code:

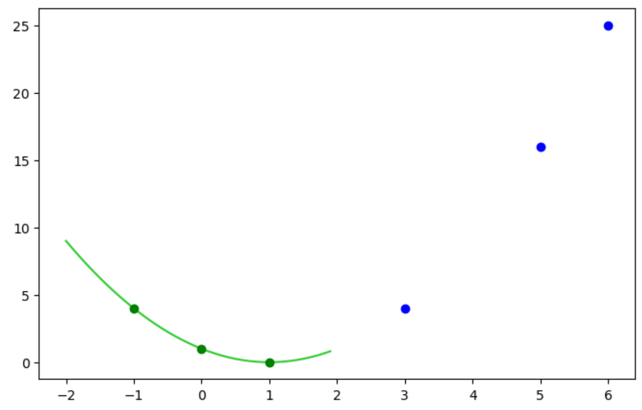
```

# Testing the code
x = np.array([0, 1, -1])
y = np.array([1, 0, 4])

to_plot = np.arange(np.min(x)-1, np.max(x)+1, .1)
A = divided_difference(x, y)
y_plot = [interpolate(item, A) for item in to_plot]

x1 = 3
x2 = 5
```

```
x3 = 6
y1 = interpolate(x1, A)
y2 = interpolate(x2, A)
y3 = interpolate(x3, A)

print("x1= "+  str(x1) + " y1= "+str(y1), "x2= "+str(x2)+ " y2= "+
                                    str(y2), "x3= " + str(x3) + " y3=
                                     "+ str(y3), sep = "\n")

plt.figure(figsize = (8, 5))

plt.plot(to_plot, y_plot, "limegreen")
plt.plot(x, y, 'go')
plt.plot([x1, x2, x3], [y1, y2, y3], 'bo')
```

Output:

```
x1= 3 y1= 4.0
x2= 5 y2= 16.0
x3= 6 y3= 25.0
```

Visualization of the interpolation, green points are the known points, green curve is the interpolated function curve, blue points are the predicted value for some arbitrary points



1 - b

Program:

```
x = np.array([-1/2, -1/4, 0, 1/2, 1/4])
y = np.cosh(x)
x_new = 1/5

A = divided_difference(x, y)
y_new = interpolate(x_new, A)
```

Output:

```
x [-0.5  -0.25  0.    0.5   0.25]
y [1.12762597 1.0314131  1.         1.12762597 1.0314131 ]
A [[ 1.12762597 -0.38485146  0.51839812 -0.01052568  0.04210273]
 [ 1.0314131  -0.1256524   0.50787244  0.02105137  0.        ]
 [ 1.          0.25525193  0.51839812  0.          0.        ]
 [ 1.12762597  0.38485146  0.          0.          0.        ]
 [ 1.0314131   0.          0.          0.          0.        ]]
Interpolate cosh(1/5):  1.0200664914617155
```

In the A matrix, the first row is $A_0, A_1, A_2, ..., A_n$

1 - c

The function below calculates the n known points for interpolation

```
# function to generate known points for Question 1 c
def q1c(n):
  x = np.array([i*(2/n)-1 for i in range(n+1)])
  y = 4/(4+121*(x**2))
  return x, y
```

Interpolation with n = 2

```
x, y = q1c(2)
A = divided_difference(x, y)
y_new = interpolate(0.05, A)
```

Output of interpolation at x = 0.05 (n=2) is **0.99758**

```
x [-1.  0.  1.]
y [0.032 1.    0.032]
A [[ 0.032  0.968 -0.968]
 [ 1.    -0.968  0.   ]
 [ 0.032  0.     0.   ]]
Estimation of f(x) at x=0.05, n=2:  0.99758
```

Interpolation with n = 4

```
x, y = q1c(4)
A = divided_difference(x, y)
y_new = interpolate(0.05, A)
```

Output of interpolation at x = 0.05 (n=4) is **0.989051884671533**

```
x [-1.  -0.5  0.   0.5  1. ]
y [0.032      0.11678832 1.         0.11678832 0.032     ]
A [[ 0.032       0.16957664  1.59684672 -3.41979562  3.41979562]
 [ 0.11678832  1.76642336 -3.53284672  3.41979562  0.        ]
 [ 1.          -1.76642336  1.59684672  0.          0.        ]
 [ 0.11678832 -0.16957664  0.          0.          0.        ]
 [ 0.032       0.          0.          0.          0.        ]]
Estimation of f(x) at x=0.05, n=4:  0.989051884671533
```

Interpolation with n = 40

```
x, y = q1c(40)
A = divided_difference(x, y)
y_new = interpolate(0.05, A)
```

Output of interpolation at x = 0.05 (n=40) is **0.9296920395119112**

```
x [-1.   -0.95 -0.9  -0.85 -0.8  -0.75 -0.7  -0.65 -0.6  -0.55 -0.5
                                    -0.45
 -0.4  -0.35 -0.3  -0.25 -0.2  -0.15 -0.1  -0.05  0.    0.05  0.1
                                     0.15
  0.2   0.25  0.3   0.35  0.4   0.45  0.5   0.55  0.6   0.65  0.7
                                      0.75
  0.8   0.85  0.9   0.95  1.  ]
y [0.032      0.03533491 0.03921184 0.04375291 0.04911591 0.
                                   05550737
 0.06320114 0.07256565 0.08410429 0.0985161  0.11678832 0.14033857
 0.17123288 0.21251162 0.26863667 0.34594595 0.45248869 0.59501673
 0.76775432 0.92969204 1.         0.92969204 0.76775432 0.59501673
 0.45248869 0.34594595 0.26863667 0.21251162 0.17123288 0.14033857
 0.11678832 0.0985161  0.08410429 0.07256565 0.06320114 0.05550737
 0.04911591 0.04375291 0.03921184 0.03533491 0.032     ]
A [[ 3.20000000e-02  6.66981736e-02  1.08404923e-01 ...  8.
                                   07978793e+11
  -8.81166598e+11  8.81166598e+11]
 [ 3.53349087e-02  7.75386659e-02  1.32826039e-01 ... -9.10296072e+
                                    11
   8.81166598e+11  0.00000000e+00]
 [ 3.92118420e-02  9.08212698e-02  1.64388920e-01 ...  8.07978793e+
                                    11
   0.00000000e+00  0.00000000e+00]
 ...
 [ 3.92118420e-02 -7.75386659e-02  1.08404923e-01 ...  0.00000000e+
                                    00
   0.00000000e+00  0.00000000e+00]
 [ 3.53349087e-02 -6.66981736e-02  0.00000000e+00 ...  0.00000000e+
                                    00
   0.00000000e+00  0.00000000e+00]
 [ 3.20000000e-02  0.00000000e+00  0.00000000e+00 ...  0.00000000e+
                                    00
   0.00000000e+00  0.00000000e+00]]
Estimation of f(x) at x=0.05, n=40:  0.9296920395119112
```

Actual value of f(0.05): **0.9296920395119116**

```
f_actual = 4/(4+121*(0.05**2))
0.9296920395119116
```

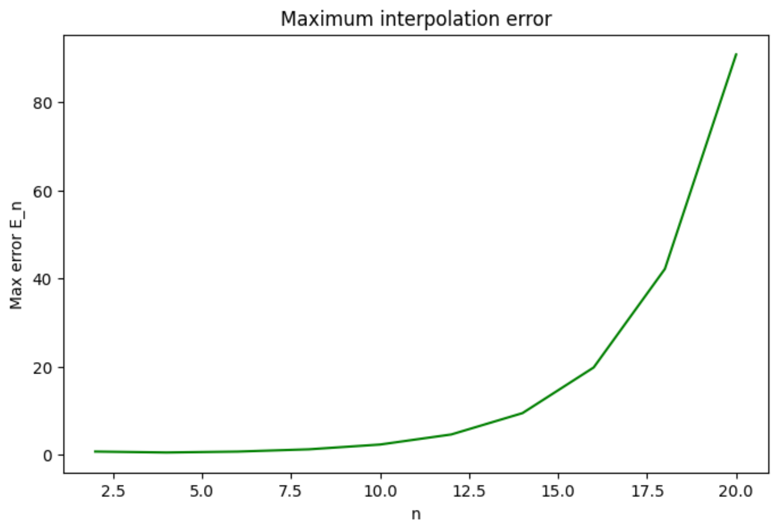
1 - d

```
# descretize the interval [-1, 1] much finer than n
# 1000 discrete points
x_test = np.array(np.arange(-1, 1.0001, 0.001))
# real f values at the points above
y_real = 4/(4+121*(x_test**2))
# the n values to test out
n_vals = np.arange(2, 21, 2).tolist()
n_vals.append(40)
# record the max errors
max_errs = []

# estimate the maximum interpolation error for
# n = 2, 4, ..., 20, and 40
for n in n_vals:
  # generate known points for n = 2, 4, 6, ..., 20
  x, y = q1c(n)
  A = divided_difference(x, y)
  # interpolate the discrete points
  y_test = np.array([interpolate(item, A) for item in x_test])
  errs = np.absolute(y_real - y_test)
  max_errs.append(max(errs))

# Plotting max error results
for i in range(len(n_vals)):
  print("n = ", n_vals[i], "E_n = ", max_errs[i])
plt.figure(figsize = (8, 5))
plt.xlabel("n")
plt.ylabel("Max error E_n")
plt.plot(np.arange(2, 21, 2), max_errs[:-1], "g")
plt.title("Maximum interpolation error")
```

Numerical output of the errors:

```
n =  2 E_n =  0.6742284503961329
n =  4 E_n =  0.4568519889200251
n =  6 E_n =  0.6661072447658077
n =  8 E_n =  1.1790313522966922
n =  10 E_n =  2.266342798431067
n =  12 E_n =  4.552373039031871
n =  14 E_n =  9.39797786204935
n =  16 E_n =  19.76669100592222
n =  18 E_n =  42.14165168940854
n =  20 E_n =  90.82676890215886
n =  40 E_n =  262578.231687619
```

Visualizations of the errors

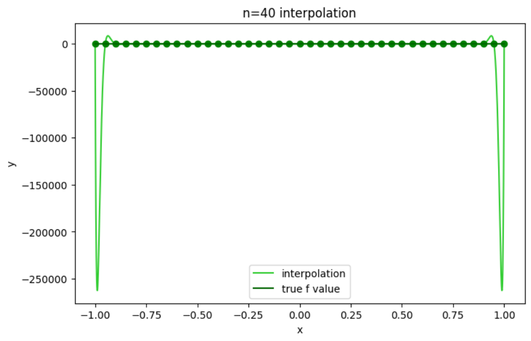
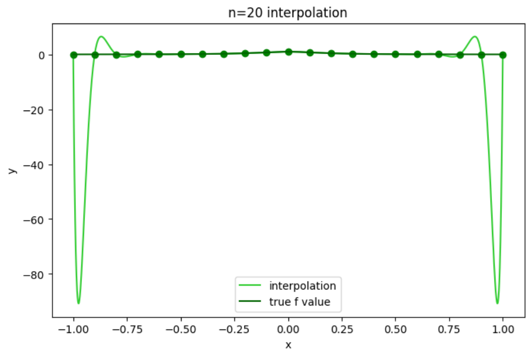
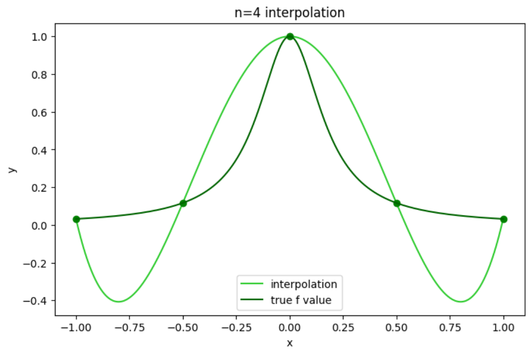
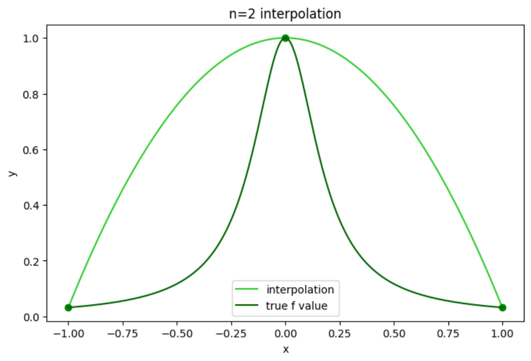


Why the error makes sense

The max error E_n for small n is relatively small, however, for **higher degree interpolating polynomials with big n's, the error can be significant**.

The Faber's Theorem tells us that for every interpolating array there exists a continuous function $g : [a.b] \to R$ and there exists a point x in (a, b) such that $P_n(g)(x)$ does not converge to g(x) as $n \to \infty$.

That is, for any preset interpolating scheme, there is some continuous function as which the scheme **fails to produce good convergence**. Often this failure of convergence is very apparent with **uniformly spaced interpolating points**. Therefore, we should be wary of higher order interpolating polynomials and we certainly should not put much faith in extrapolation.


n=40 interpolation
interpolation
true f value
x
y

n=20 interpolation
interpolation
true f value
x
y

n=4 interpolation
interpolation
true f value
x
y

n=2 interpolation
interpolation
true f value
x
y

Question2

$$f(x) = cos(2\pi x)$$

Linear interpolation

The table contains the values $f(x_i), i = 0, 1, 2, d..., n$ with $n = \frac{1}{h}$ where $x_i = -\frac{1}{4} + i * \frac{1}{h}$. If $\bar{x} \in [x_{i-1}, x_i]$ then we approximate $f(\bar{x})$ with $p_1(x)$ where $p_1(x)$ is the polynomial of degree 1 that interpolates f at x_{i-1}, x_i

The interpolation error is shown below, where ξ depends on $\bar{x}$

$$e_1(\bar{x}) = \frac{f^{''}(\xi)}{2!}(\bar{x} - x_{i-1})(\bar{x} - x_i)$$

(1) We first estimate $|f^{''}(\xi)|$

$$|f^{''}(\xi)| \leq max_{-\frac{1}{4} \leq x \leq \frac{3}{4}} |f^{''}(x)| \leq max_{-\frac{1}{4} \leq x \leq \frac{3}{4}} (-4\pi^2 cos(2\pi x)) = 4\pi^2$$

This maximum value is reached when $x = \frac{1}{2}$, meaning $cos(2\pi x) = -1$

(2) We then evaluate $|(\bar{x} - x_{i-1})(\bar{x} - x_i)|$

Let $y = \bar{x} - x_i$, substitute into the above expression:

$$|(\bar{x} - x_{i-1})(\bar{x} - x_i)| \leq max_{y \in [0,h]} |(y - h)y|$$

Let $g(y) = (y - h)y = y^2 - hy$, and we have $g(h) = g(0) = 0$. Therefore $g(y)$ achieves its maximum on $[0, h]$'s interior.

$$g'(y) = 2y - h$$

$g'(y) = 0$ is satisfied if and only if $y = \frac{h}{2}$

$$|g(\frac{h}{2})| = |(\frac{h^2}{4} - \frac{h^2}{2})| = \frac{h^2}{4}$$

$$|(\bar{x} - x_{i-1})(\bar{x} - x_i)| \leq \frac{h^2}{4}$$

(3) Substitute the above two results into $|e_2(\bar{x})|$:

$$|e_1(\bar{x})| \leq \frac{4\pi^2}{2!} \frac{h^2}{4} = \frac{\pi^2}{2} h^2$$

If we want to achieve 6 decimal digit accuracy, then we need to pick an h such that $2|e_1(\bar{x})| < 1 \times 10^{-6}$, then

$$\frac{\pi^2}{2} h^2 < 5 \times 10^{-7}$$

$$h \approx 0.000318309$$

Therefore, the minimum number of intervals $\frac{1}{h} \approx 3142$

The number of table entries is therefore **3143**

Quadratic interpolation

The table contains the values $f(x_i), i = 0, 1, 2, d..., n$ with $n = \frac{1}{h}$ where $x_i = -\frac{1}{4} + i * \frac{1}{h}$. If $\bar{x} \in [x_{i-1}, x_{i+1}]$ then we approximate $f(\bar{x})$ with $p_2(x)$ where $p_2(x)$ is the polynomial of degree 2 that interpolates f at x_{i-1}, x_i, x_{i+1}

The interpolation error is shown below, where ξ depends on $\bar{x}$

$$e_2(\bar{x}) = \frac{f^{'''}(\xi)}{3!}(\bar{x} - x_{i-1})(\bar{x} - x_i)(\bar{x} - x_{i+1})$$

(1) We first estimate $|f^{'''}(\xi)|$

$$|f^{'''}(\xi)| \leq max_{-\frac{1}{4} \leq x \leq \frac{3}{4}} |f^{'''}(x)| \leq max_{-\frac{1}{4} \leq x \leq \frac{3}{4}} 8\pi^3 sin(2\pi x) = 8\pi^3$$

This maximum value is reached when $x = \frac{1}{4}$, meaning $sin(2\pi x) = 1$

(2) We then evaluate $|(\bar{x} - x_{i-1})(\bar{x} - x_i)(\bar{x} - x_{i+1})|$

Let $y = \bar{x} - x_i$, substitute into the above expression:

$$|(\bar{x} - x_{i-1})(\bar{x} - x_i)(\bar{x} - x_{i+1})| \leq max_{y \in [-h,h]} |(y - h)y(y + h)|$$

Let $g(y) = (y - h)y(y + h) = y^3 - h^2 y$, and we have $g(h) = g(-h) = g(0) = 0$. Therefore $g(y)$ achieves its maximum on $[-h, h]$'s interior.

$$g'(y) = 3y^2 - h^2$$

$g'(y) = 0$ is satisfied if and only if $y = \pm \frac{h}{\sqrt{3}}$

$$|g(\pm \frac{h}{\sqrt{3}})| = | \pm (\frac{h^3}{3\sqrt{3}} - \frac{h^3}{\sqrt{3}})| = \frac{2h^3}{3\sqrt{3}}$$

$$|(\bar{x} - x_{i-1})(\bar{x} - x_i)(\bar{x} - x_{i+1})| \leq \frac{2h^3}{3\sqrt{3}}$$

(3) Substitute the above two results into $|e_2(\bar{x})|$:

$$|e_2(\bar{x})| \leq \frac{1}{3!} 8\pi^3 \frac{2h^3}{3\sqrt{3}} = \frac{8\pi^3}{9\sqrt{3}} h^3$$

If we want to achieve 6 decimal digit accuracy, then we need to pick an h such that $2|e_2(\bar{x})| < 1 \times 10^{-6}$, then

$$\frac{8\pi^3}{9\sqrt{3}} h^3 < 5 \times 10^{-7}$$

$$h \approx 0.0031556$$

Therefore, the minimum number of intervals $\frac{1}{h} \approx 317$

The number of table entries is therefore **318**

Question 3

We solve for $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$ until convergence or when we exceed maximum iterations. Program for Newton's method:

```
def Newtons_nethod(f, df, z, thresh, iter_max):
  # f: original function which we want to find root
  # df: derivative of f
  # z: initial guess
  # thresh: threshold for detecting convergence
  # iter_max: maximum iteration
  res = z
  iter = 0
  while iter < iter_max:
    f_res = f(res)
    df_res = df(res)
    # if initial guess is good enough, return
    if np.absolute(f_res) <= thresh:
      return float(res)
    elif df_res == 0:
      # zero division error from df == 0
      return False
    else:
      # iterative method
      res = res - (f_res/df_res)
    iter += 1
  return False
```

```
f_test = lambda x: mpmath.tan(x) - x
df_test = lambda x: (mpmath.sec(x))**2 - 1
thresh_test = 0.0001
iter_max_test = 1000
zs = np.arange(95, 105, 0.05)
results = []
# find root on either side of this number
num = 100
for z_test in zs:
  res_test = Newtons_nethod(f_test, df_test, z_test, thresh_test,
                                    iter_max_test)
  results.append(res_test)
results = [j for j in results if j != False]
```

I tested with initial guess values from 95 to 105, with an interval of 0.05 between each guess. The output of the program is as follows:

$$x_{low} = 98.95006282434697, x_{high} = 102.09196646490774$$

```
Roots found:  [95.80813878814617, 98.95006282434697, 102.
                                  09196646490774, 108.
                                  37571965204596, 133.
                                  51019785553368, 290.
                                  59387924241156]
solutions on either side of 100:
low root:  98.95006282434697
high root:  102.09196646490774
```

Question 4

(a)

When ξ is a root of order 2 of $f(x)$, $f(\xi) = 0, f'(\xi) = 0, f''(\xi) \neq 0$

Using Newton's method

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Subtracting ξ from both sides gives

$$x_{n+1} - \xi = x_n - \xi - \frac{f(x_n)}{f'(x_n)} \qquad (1)$$

Assuming $f(x)$ is at least 2 times continuously differentiable and f, f', f'' are continuous in ξ's neighborhood, for x, ξ R, and $c, c' \in$R in the interval between x, ξ, using Taylor's formula with remainder around ξ

$$f(x) = f(\xi)+f'(\xi)(x-\xi)+\frac{f''(\xi)}{2!}(x-\xi)^2+...+\frac{f^k(\xi)}{k!}(x-\xi)^k+\frac{f^{k+1}(c)}{(k+1)!}(x-\xi)^{k+1}$$

$$f'(x) = f'(\xi) + f''(\xi)(x-\xi) + ... + \frac{f^k(\xi)}{k!}(x-\xi)^k + \frac{f^{k+1}(c')}{(k+1)!}(x-\xi)^{k+1}$$

At convergence we ignore the higher order terms as they approach zero

$lim_{x\to\xi} \frac{f^k(\xi)}{k!}(x-\xi)^k \to 0$

$lim_{x\to\xi} \frac{f^{k+1}(c)}{(k+1)!}(x-\xi)^{k+1} \to 0$

$lim_{x\to\xi} \frac{f^{k+1}(c')}{(k+1)!}(x-\xi)^{k+1} \to 0$

When $x \to \xi$, expression (1) can then be written as

$$x_{n+1} - \xi = x_n - \xi - \frac{f(\xi) + f'(\xi)(x_n - \xi) + \frac{f''(\xi)}{2!}(x_n-\xi)^2}{f'(\xi) + f''(\xi)(x_n - \xi)}$$

Substituting $f(\xi) = 0, f'(\xi) = 0$

$$x_{n+1} - \xi = x_n - \xi - \frac{\frac{f''(\xi)}{2}(x_n-\xi)^2}{f''(\xi)(x_n-\xi)}$$

Divide both sides by $(x_n - \xi)$, and because $f''(\xi) \neq 0$, we have

$$lim_{x\to\xi}\frac{|x_{n+1}-\xi|}{|x_n-\xi|} = 1 - \frac{1}{2} = \frac{1}{2}$$

Therefore, Newton's method converges linearly for root of degree 2.

(b)

When we have

$$x_{n+1} = x_n - 2\frac{f(x_n)}{f'(x_n)}$$

We get the expression below by repeating steps in (a)

$$x_{n+1} - \xi = x_n - \xi - 2\frac{\frac{f''(\xi)}{2}(x_n - \xi)^2}{f''(\xi)(x_n - \xi)}$$

Divide both sides by $(x_n - \xi)$, and because $f''(\xi) \neq 0$, we have

$$lim_{x \to \xi}\frac{|x_{n+1} - \xi|}{|x_n - \xi|} = 1 - 2(\frac{1}{2}) = 0$$

Assuming $f(x)$ is at least 3 times continuously differentiable and f, f', f'', f''' are continuous in ξ's neighborhood. Using the Taylor Theorem with remainder, for x, ξ R, expanding Taylor's formula up to the third degree term

$$x_{n+1} - \xi = x_n - \xi - 2\frac{f(\xi) + f'(\xi)(x - \xi) + \frac{f''(\xi)}{2!}(x - \xi)^2 + \frac{f'''(\xi)}{3!}(x - \xi)^3}{f'(\xi) + f''(\xi)(x - \xi)}$$

Substituting $f(\xi) = 0, f'(\xi) = 0$

$$x_{n+1} - \xi = x_n - \xi - 2\frac{\frac{f''(\xi)}{2!}(x - \xi)^2 + \frac{f'''(\xi)}{3!}(x_n - \xi)^3}{f''(\xi)(x_n - \xi)}$$

$$x_{n+1} - \xi = x_n - \xi - (x_n - \xi) - \frac{\frac{1}{3}f'''(\xi)(x_n - \xi)^2}{f''(\xi)}$$

$$lim_{x_n \to \xi}\frac{|x_{n+1} - \xi|}{(x_n - \xi)^2} = | - \frac{1}{3}\frac{f'''(\xi)}{f''(\xi)}|$$

Because $f'''(\xi) \neq 0, f''(\xi) \neq 0$, the right side of the equation is a constant, thus Newton's method in this case converges quadratically

Question 5

(a)

Implementation of Muller's method.

```python
import math
import numpy as np
import copy

def muller_method(x0, x1, x2, iter_max, thresh):
  roots = []
  iter = 0
  while iter < iter_max:
    h0 = x1 - x0
    h1 = x2 - x1
    f_x0 = f(x0, roots_found)
    f_x1 = f(x1, roots_found)
    f_x2 = f(x2, roots_found)
    d0 = (f_x1 - f_x0) / h0
    d1 = (f_x2 - f_x1) / h1

    a = (d1 - d0) / (h1 + h0)
    b = a*h1 + d1
    c = f_x2

    val = b**2 - 4*a*c
    if val < 0:
      desc = np.sqrt(val + 0j)
    else:
      desc = np.sqrt(val)

    # sign agrees with that of bso we get largest denominator
    if b < 0:
      de = b - desc
    else:
      de = b + desc

    x3 = x2 + (-2*c) / de
    err = np.absolute((x3-x2)/x3)

    # if converges save this root
    if err < thresh:
      if np.iscomplex(x3):
        if np.abs(np.imag(x3)) < 0.0000001:
          x3 = np.real(x3)
        else:
          roots.append(np.conjugate(x3))
      roots.append(x3)
      return roots

    # otherwise keep finding
    x0 = copy.deepcopy(x1)
    x1 = copy.deepcopy(x2)
    x2 = copy.deepcopy(x3)
    iter += 1
  return False
```

(b)

The function below implements the given polynomial function. It takes a list of already found roots as parameter so that it can do deflation.

```
def f(x, lst):
  res = x**7 + x**6 + x**5 + x**4 + x**3 + x**2 + x + 1
  if len(lst) == 0:
    return res

  else:
    # do deflation when roots are found
    to_divide = 1
    for item in lst:
      to_divide *= (x - item)
    return res / to_divide
```

The following code does the root finding:

```
iter_max_test = 1000

# this is initial x1
initial_x = 0
shift = 1

thresh_test = 0.00001

degrees = 7
roots_found = []

while len(roots_found) < degrees:
  res = muller_method(initial_x - shift, initial_x, initial_x +
                                    shift, iter_max_test,
                                    thresh_test)
  if res != False:
    roots_found.extend(res)
    initial_x = initial_x + shift


print("roots found", roots_found)
```

Output is as follows, with 7 roots in total

```
roots found

[(-0.7071067811864944+0.7071067811854436j),

(-0.7071067811864944-0.7071067811854436j),

(0.7071067811864156+0.7071067811865268j),

(0.7071067811864156-0.7071067811865268j),

(-5.5554820990926645e-18-1j),

(-5.5554820990926645e-18+1j),

-1.0]
```

Question 6

(a)

$p(x), q(x)$ **share a common root** from the method of resultants implemented:

```
import numpy as np

# take the coefficients
def resultants_root(c1, c2, thresh):
  dim = len(c1) - 1 + len(c2) - 1
  Q = np.zeros((dim, dim))

  for i in range(len(c1)-1):
    Q[i, i:i+len(c1)] = c1
  for j in range(len(c2)-1):
    Q[len(c1) - 1 + j, j:j+len(c2)] = c2
  print("matrix Q")
  print(Q)

  det = np.linalg.det(Q)
  print("Q's determinant is ", det)

  # if common root exists, use the ration method
  if np.absolute(det) < thresh:
    print("We consider Q has a determinant of zero")
    Q = np.array(Q[:-1])
    q1 = Q[:, 1:]
    columns = np.array([i for i in range(Q.shape[1]) if i != 1],
                                   dtype=np.intp)
    q2 = Q[:, columns]
    print("q1", q1, "q2", q2, sep = "\n")

    r = -1 * ((np.linalg.det(q1)) / (np.linalg.det(q2)))
    print("Obtain the root by dividing det(q1) / det(q2) gives: ",
                                   r)
    return r
```

The main function that passes the coefficients of p and q

```
if __name__ == "__main__":
  c1 = [1, -2, 1, -2]
  c2 = [1, -1, -4, 4]
  thresh = 0.0001
  r = resultants_root(c1, c2, thresh)
  print("The common root is ", r)
```

Output for common root existence

```
matrix Q
[[ 1. -2.  1. -2.  0.  0.]
 [ 0.  1. -2.  1. -2.  0.]
 [ 0.  0.  1. -2.  1. -2.]
 [ 1. -1. -4.  4.  0.  0.]
 [ 0.  1. -1. -4.  4.  0.]
 [ 0.  0.  1. -1. -4.  4.]]
Q's determinant is  -2.2204460492503077e-14
We consider Q has a determinant of zero
```

(b)

The ratio method is implemented as part of the above function

```
# if common root exists, use the ration method
  if np.absolute(det) < thresh:
    print("We consider Q has a determinant of zero")
    Q = np.array(Q[:-1])
    q1 = Q[:, 1:]
    columns = np.array([i for i in range(Q.shape[1]) if i != 1],
                                   dtype=np.intp)
    q2 = Q[:, columns]
    print("q1", q1, "q2", q2, sep = "\n")

    r = -1 * ((np.linalg.det(q1)) / (np.linalg.det(q2)))
```

Output for finding the common root

```
q1
[[-2.  1. -2.  0.  0.]
 [ 1. -2.  1. -2.  0.]
 [ 0.  1. -2.  1. -2.]
 [-1. -4.  4.  0.  0.]
 [ 1. -1. -4.  4.  0.]]
q2
[[ 1.  1. -2.  0.  0.]
 [ 0. -2.  1. -2.  0.]
 [ 0.  1. -2.  1. -2.]
 [ 1. -4.  4.  0.  0.]
 [ 0. -1. -4.  4.  0.]]
Obtain the root by dividing det(q1) / det(q2):  2.000000000000002
The common root is  2.000000000000002
```

Question 7

$$p(x, y) = 4x^2 + 4y^2 - 1$$
$$q(x, y) = 2x^2 + y$$

Treating the polynomials p and q as functions of y gives

$$p(x, y) = 4y^2 + 0y + (4x^2 - 1)$$

$$q(x, y) = y + 2x^2$$

We construct Q as

$$Q = \begin{bmatrix} 4 & 0 & 4x^2 - 1 \\ 1 & 2x^2 & 0 \\ 0 & 1 & 2x^2 \end{bmatrix}$$

$$det(Q) = 16x^4 + 4x^2 - 1$$

We need $det(Q) = 0$

$$16x^4 + 4x^2 - 1 = 0$$

$$x^2 = \frac{-4 \pm \sqrt{4^2 - 4 * (16) * (-1)}}{2 * 16}$$

Take the positive roots gives $x^2 = \frac{\sqrt{5}}{8} - \frac{1}{8} \approx 0.15451$

$$p(x, y) = 4y^2 + 0y + (4 * 0.15451 - 1) = 4y^2 - 0.38196 = 0$$

$$q(x, y) = y + 2 * 0.15451 = y + 0.30902 = 0$$

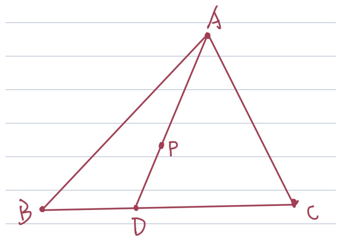
$$\mathbf{y = -0.30902}$$

$$\mathbf{x = \pm 0.393078}$$

With approximations

Question 8

(a)



The above images visualizes a 2D point P that falls inside or on the triangle formed by three known 2D points A, B, C. Let D be any point on the line segment BC, let P be any point on the line segment AD. Then P always falls inside the triangle. We solve for the coordinates of an arbitrary P. Denote $A(x^{(i)}, y^{(i)}), B(x^{(j)}, y^{(j)}), C(x^{(k)}, y^{(k)})$, and $P(x, y)$. An arbitrary point D $(x^{(D)}, y^{(D)})$ on the BC line can be represented as

$$(x^{(k)}, y^{(k)}) + \lambda[(x^{(j)}, y^{(j)}) - (x^{(k)}, y^{(k)})] \qquad \lambda \in [0,1]$$

$$\begin{bmatrix} x^{(D)} \\ y^{(D)} \end{bmatrix} = \begin{bmatrix} \lambda x^{(j)} + (1-\lambda)x^{(k)} \\ \lambda y^{(j)} + (1-\lambda)y^{(k)} \end{bmatrix} \qquad \lambda \in [0,1]$$

An arbitrary point P (x, y) on the AD line can be represented as

$$(x^{(i)}, y^{(i)}) + \xi[(x^{(D)}, y^{(D)}) - (x^{(i)}, y^{(i)})] \qquad \xi \in [0,1]$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \xi x^{(D)} + (1-\xi)x^{(i)} \\ \xi y^{(D)} + (1-\xi)y^{(i)} \end{bmatrix} = \begin{bmatrix} \xi\lambda x^{(j)} + \xi(1-\lambda)x^{(k)} + (1-\xi)x^{(i)} \\ \xi\lambda y^{(j)} + \xi(1-\lambda)y^{(k)} + (1-\xi)y^{(i)} \end{bmatrix} \qquad \lambda, \xi \in [0,1]$$

Therefore, $\xi\lambda, \xi(1-\lambda), (1-\xi) \in [0,1]$

Denote $v_2 = \xi\lambda, v_3 = \xi(1-\lambda), v_1 = (1-\xi)$, then their values satifsy

$$v_1, v_2, v_3 \in [0,1]$$

$$v_1 + v_2 + v_3 = \xi\lambda + \xi - \xi\lambda + 1 - \xi = 1$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} v_1 x^{(i)} + v_2 x^{(j)} + v_3 x^{(k)} \\ v_1 y^{(i)} + v_2 y^{(j)} + v_3 y^{(k)} \end{bmatrix} = \begin{bmatrix} x^{(i)} & x^{(j)} & x^{(k)} \\ y^{(i)} & y^{(j)} & y^{(k)} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

Ensuring that $v_1 + v_2 + v_3 = 1$, we rearrange the expression as

$$\begin{bmatrix} x^{(i)} & x^{(j)} & x^{(k)} \\ y^{(i)} & y^{(j)} & y^{(k)} \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

This equation can be expressed as $\mathbf{A}v = \mathbf{b}$, $\mathbf{A} = \begin{bmatrix} x^{(i)} & x^{(j)} & x^{(k)} \\ y^{(i)} & y^{(j)} & y^{(k)} \\ 1 & 1 & 1 \end{bmatrix}$ denotes the matrix formed by coordinates of the 3 known points, $v = [v_1, v_2, v_3]^T$, and $\mathbf{b} = [x, y, 1]^T$

The above expression is simply a matrix form of the **barycentric coordinates**

$$P(x, y) = v_1 A + v_2 B + v_3 C$$

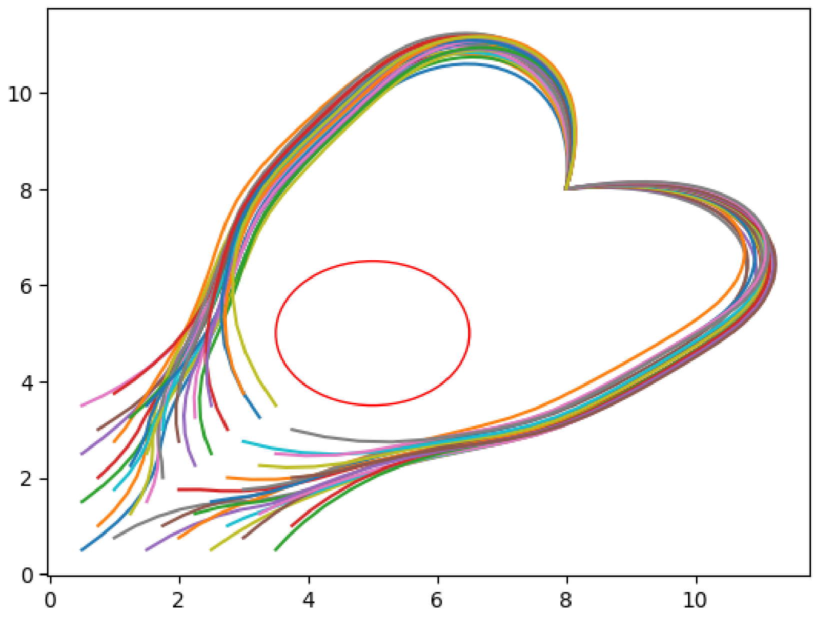
$$v_1, v_2, v_3 \in [0, 1], v_1 + v_2 + v_3 = 1$$

(b)

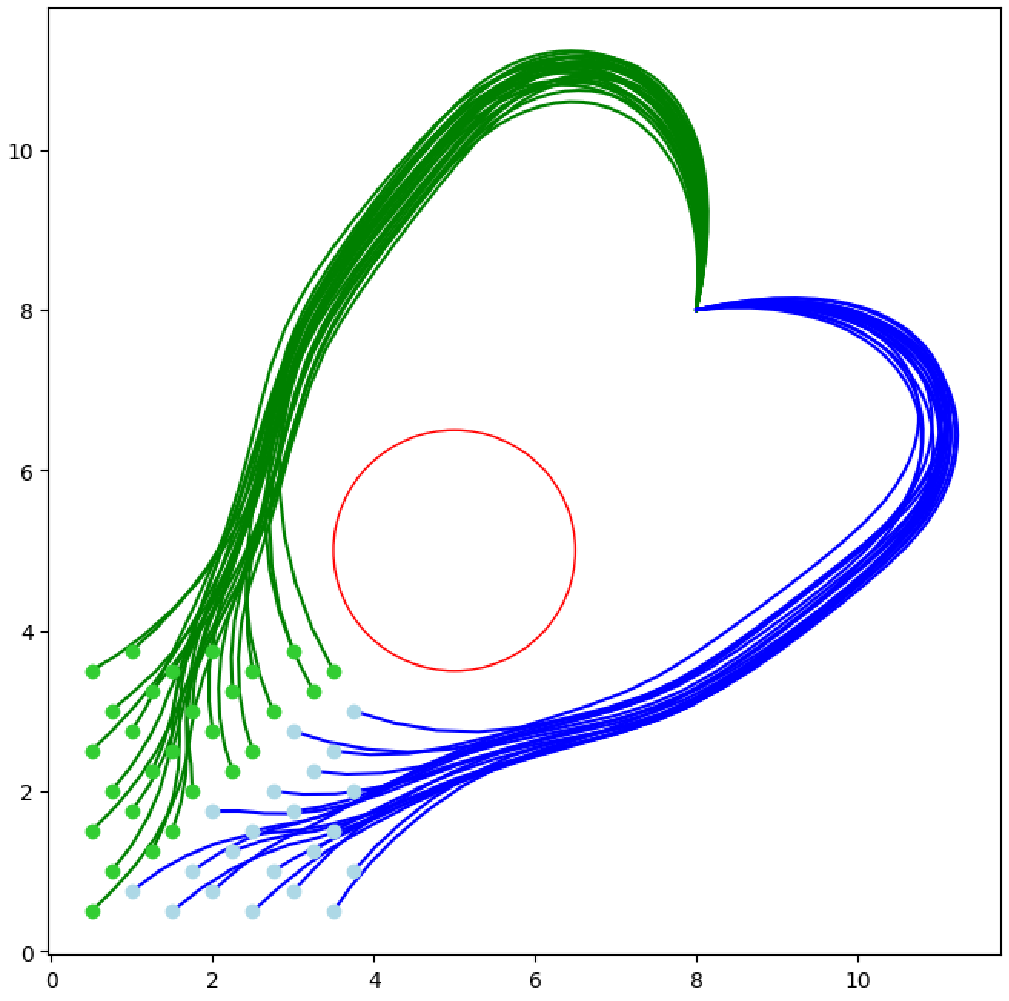
My code is as follows:

```python
import matplotlib.pyplot as plt
import numpy as np

# this block of code reads paths data
f = open("paths.txt", "r" )
path_arr = []
for line in f:
  line = line.rstrip('\n')
  row = line.split()
  row_num = [float(item) for item in row]
  path_arr.append(row_num)
f.close()
print(path_arr)

fig, ax = plt.subplots()
for i in range(0, len(path_arr), 2):
  plt.plot(np.array(path_arr[i]), np.array(path_arr[i+1]))
circle1 = plt.Circle( (5, 5 ),1.5 ,fill = False, color = "r" )
ax.add_patch(circle1)
plt.show()
```

Visualizing all the paths



Splitting the paths into upper and lower paths



A block of all functions I defined

```
##### This block defines all the functions

# split paths into upper and lower
def split_up_low(paths, c, d):
  paths = np.array(paths)
  # return: the indices of up and low paths
  u_idx = []
  l_idx = []
  # expression of the line that splits upper and lower y = k*x + b
  x1, x2, y1, y2 = c[0], d[0], c[1], d[1]
  k = (y1-y2)/(x1-x2)
  b = y1 - x1*k

  for i in range(0, len(paths), 2):
    path_xs = paths[i]
    path_ys = paths[i+1]
    line_ys = k*path_xs + b
    dists = path_ys - line_ys
    if np.average(dists) > 0:
      u_idx.append(i)
    else:
      l_idx.append(i)
  return u_idx, l_idx

# decide from p0 whether to take upper or lower paths
def decide_up_low(p0, up0s, lp0s):
  udist = np.average(np.linalg.norm(p0 - up0s, axis=1))
  ldist = np.average(np.linalg.norm(p0 - lp0s, axis=1))
  # 1 means the p0 belongs to upper paths
  if udist < ldist:
    return 1
  else:
    return 0

# choose initial three points
def choose_initial_points(p0, p0s):
  # compute the distances between p0 and other starting points
  ds = np.linalg.norm(p0 - p0s, axis=1)
  close_p0_idx = np.argsort(ds, axis=0)

  # first locate the 2 points closest to p0
  pi = p0s[close_p0_idx[0]]
  pj = p0s[close_p0_idx[1]]
  k = 2

  # look for a 3rd point that forms a triangle with the first 2
  # iterate until p0 falls in a triangle formed by 3 initial points
  while k < len(p0s):
    pk = p0s[close_p0_idx[k]]
    # the initial weights fpr Barymetric coordinates
    A = np.matrix([[pi[0], pj[0], pk[0]], [pi[1], pj[1], pk[1]], [1
                                   .0, 1.0, 1.0]])
    b = np.array([p0[0], p0[1], 1.0])
    weights = np.linalg.solve(A, b)

    if (np.all(weights> 0.0)) and np.absolute(np.sum(weights)-1) <=
                                          0.000001:
      return np.array([close_p0_idx[0], close_p0_idx[1],
                                           close_p0_idx[k]]), weights
    k += 1
  return False

# compute new path from w: initial weights and ps: 3 chosen paths
def compute_new_path(w, ps):
  #w = np.array(w)[0]
  new_path = np.zeros((2, ps.shape[1]))
  for i in range(len(w)):
    new_path[0] += w[i] * ps[i*2]
    new_path[1] += w[i] * ps[i*2+1]
  return new_path

#######  The complete process of creating a new path   #######
def create_new_path(p0, u_paths, l_paths, interpolation = False):
  l_p0s = l_paths[:, 0].reshape(int(len(l_paths)/2), 2)
  u_p0s = u_paths[:, 0].reshape(int(len(u_paths)/2), 2)
  path_choice = decide_up_low(p0, u_p0s, l_p0s)

  # choose upper or lower path for p0
  if path_choice == 1:
    my_paths = u_paths
  else:
    my_paths = l_paths
  my_p0s = my_paths[:, 0].reshape(int(len(my_paths)/2), 2)

  # choose inital 3 p0's to determine the 3 paths
  init_p0_idx, init_weights = choose_initial_points(p0, my_p0s)
  init_p0s = my_p0s[init_p0_idx]

  # the 3 corresponding chosen paths
  chosen_paths_idx = []
  for idx in init_p0_idx:
    chosen_paths_idx.append(2*idx)
    chosen_paths_idx.append(2*idx+1)
  chosen_paths = my_paths[np.array(chosen_paths_idx)]

  # compute the new path
  new_path = compute_new_path(init_weights, chosen_paths)

  # plot the results
  fig, ax = plt.subplots(figsize = (8, 8))
  plt.xlim(0, 12)
  plt.ylim(0, 12)
  ax = plt.gca()
  ax.set_aspect('equal', adjustable='box')
  plt.text(p0[0], p0[1], '({}, {})'.format(p0[0], p0[1]))
  plt.plot(init_p0s[:, 0], init_p0s[:, 1], 'o', color = 'orange')
  for i in range(0, len(chosen_paths), 2):
    plt.plot(chosen_paths[i], chosen_paths[i+1], color = 'orange',
                                    label = "chosen paths" if i =
                                    = 0 else None)
  plt.plot(new_path[0], new_path[1], 'o', color = 'lightgreen',
                                  label="new path")
  plt.plot(p0[0], p0[1], 'o', color = 'red', label = "p0")
  circle1 = plt.Circle( (5, 5 ),1.5 ,fill = False, color = "r" )
  ax.add_patch(circle1)

  # if we want to interpolate as well, with continuous t
  # approximated with finely chosen points
  if interpolation == True:
    t_arr = np.arange(0, new_path.shape[1], 0.01)
    xt_arr, pt_arr = interpolate(t_arr, new_path)
    plt.plot(xt_arr, pt_arr, color = "green", label = "
                                  interpolation")
    plt.legend( loc ="lower right")
    plt.show()
    return new_path, xt_arr, pt_arr

  plt.legend( loc ="lower right")
  plt.show()
  return new_path

# interpolation for one t value
def interpolate_step(t, path):
  # time scale for t is 1
  if t >= path.shape[1]-1:
    return path[0][-1], path[1][-1]
  elif t < 0:
    return path[0][0], path[1][0]
  else:
    # interpolate linearly between each points
    # floor and ceiling of t
    t_fl = int(t)
    t_cl = int(t) + 1
    x1, x2, y1, y2 = path[0][t_fl], path[0][t_cl], path[1][t_fl],
                                  path[1][t_cl]
    k = (y1-y2) / (x1-x2)
    b = y1 - x1*k
    # as (xt - x1)/(x2 - x1) = (t - t_fl)/(t_cl - t_fl) = (t - t_fl
                                  )/1, we have
    xt = x1 + (t - t_fl)*(x2 - x1)
    pt = k*xt + b
  return xt, pt

# interpolation for all t's
def interpolate(t_arr, path):
  pt_arr = []
  xt_arr = []
  for t in t_arr:
    res = interpolate_step(t, path)
    pt_arr.append(res[1])
    xt_arr.append(res[0])
  pt_arr = np.array(pt_arr)
  xt_arr = np.array(xt_arr)
  return xt_arr, pt_arr
```

The main function:

```
if __name__ == "__main__":
  ##### STEP 0: initialize

  circle_c = np.array([5, 5])
  dest = np.array([8, 8])
  uidx, lidx = split_up_low(path_arr, circle_c, dest)
  # upper and lower paths
  u_paths = []
  l_paths = []
  for i in uidx:
    u_paths.append(path_arr[i])
    u_paths.append(path_arr[i+1])
  for j in lidx:
    l_paths.append(path_arr[j])
    l_paths.append(path_arr[j+1])

  ##### STEP 1: split paths into upper and lower

  # arrays of the upper and lower paths
  u_paths, l_paths = np.array(u_paths), np.array(l_paths)
  # plot upper and lower paths
  fig, ax = plt.subplots(figsize=(8, 8))
  for i in range(0, len(u_paths), 2):
    plt.plot(u_paths[i], u_paths[i+1], color = "green")
  for i in range(0, len(l_paths), 2):
    plt.plot(l_paths[i], l_paths[i+1], color = "blue")
  circle1 = plt.Circle( (5, 5 ),1.5 ,fill = False, color = "r" )
  ax.add_patch(circle1)

  # starting points of the paths
  l_p0s = l_paths[:, 0].reshape(int(len(l_paths)/2), 2)
  u_p0s = u_paths[:, 0].reshape(int(len(u_paths)/2), 2)
  plt.plot(l_p0s[:, 0], l_p0s[:, 1], 'o', color="lightblue")
  plt.plot(u_p0s[:, 0], u_p0s[:, 1], 'o', color="limegreen")
  plt.show()

  ##### STEP 2: calculate weights for each step

  # the starting point
  p0_arr = [np.array([0.8, 1.8]), np.array([2.2, 1.0]), np.array([2
                                 .7, 1.4])]
  new_path1 = create_new_path(p0_arr[0], u_paths, l_paths)
  new_path2 = create_new_path(p0_arr[1], u_paths, l_paths)
  new_path3 = create_new_path(p0_arr[2], u_paths, l_paths)


  ##### STEP 3: interpolate for continuous t

  # here pt_arr's are the interpolated p(t) values
  new_path1, xt_arr1, pt_arr1 = create_new_path(p0_arr[0], u_paths,
                                  l_paths, interpolation = True)
  new_path2, xt_arr2, pt_arr2 = create_new_path(p0_arr[1], u_paths,
                                  l_paths, interpolation = True)
  new_path3, xt_arr3, pt_arr3 = create_new_path(p0_arr[2], u_paths,
                                  l_paths, interpolation = True)
```

(c)

(1) How I picked the triple of paths $p^{(1)}, p^{(j)}, p^{(k)}$:

I first identify the p_0 given, compute its mean distance with the p_0's in the upper and lower paths *(see (5) in this section) to decide which batch of paths I should pick from

From the chosen batch of paths (either the upper paths or the lower), I compute the distance between the path p_0's with my starting point p_0, sort them from closest to furthest, record their indices.

I keep the 2 points closest to p_0 fixed, and look for a third point that forms a triangle around p_0, from the third closest point all the way up to the furthest.

The criteria for succesfully finding such a third point is that the $Av = b$ or $Aa = [x, y]^T$ calculation gives positive a_i, a_j, a_k and $a_i, a_j, a_k = 0$

As the indices of path starting points are recorded, I pick the corresponding three paths to the three starting points I picked.

(2) How I choose the weights a_i, a_j, a_k

The weights are computed from the coordinates of three starting point chosen, and compute as in question 8.a, $Aa = [x, y]^T$ for the a_i, a_j, a_k satisfying the conditions required to let p fall inside the triangle formed by the three chosen points.

(3) How I decided on the time scale t = 1

As time can take arbitrary units, I take scale = 1 as on unit of time interval, which is the time interval between two consecutive points.

(4) How I decided on the interpolation method as piece-wise linear interpolation

It is the most computationally efficient strategy.

The y values between two path points are computed as

$$y = kx + b$$

$$k = \frac{y_i - y_{i-1}}{x_i - x_{i-1}} \qquad b = y_i - k$$

For a given time t (continuous, so might not be an integer as designed by the time scale = 1), we take floor(t) and ceiling(t) to compute its floor and ceiling integers, and use these integers as indices to refer to the floor(t)th and ceiling(t)th point on the path.

because

$$\frac{t - floor(t)}{ceiling(t) - floor(t)} = \frac{x_t - x_{i-1}}{x_i - x_{i-1}}$$

We use

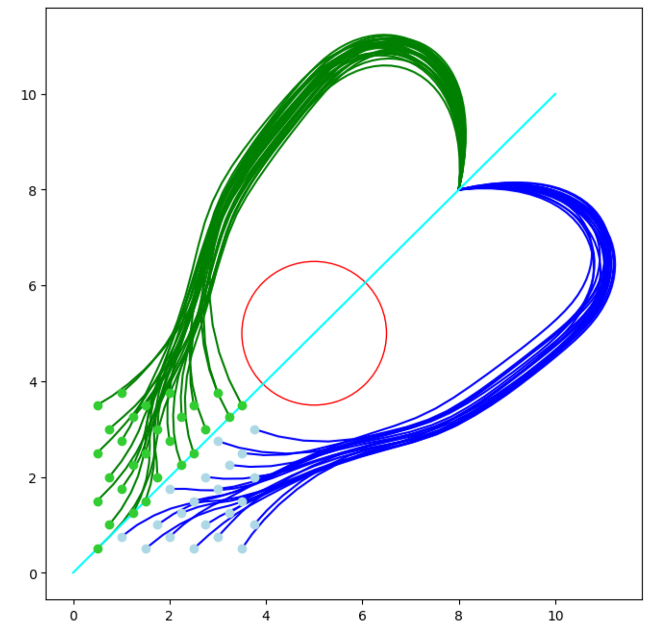
$$x_t = x_{i-1} + (t - floor(t))(x_i - x_{i-1})$$

to compute the y_t at the given timestep t.

(5) How I split the paths into upper and lower in the beginning

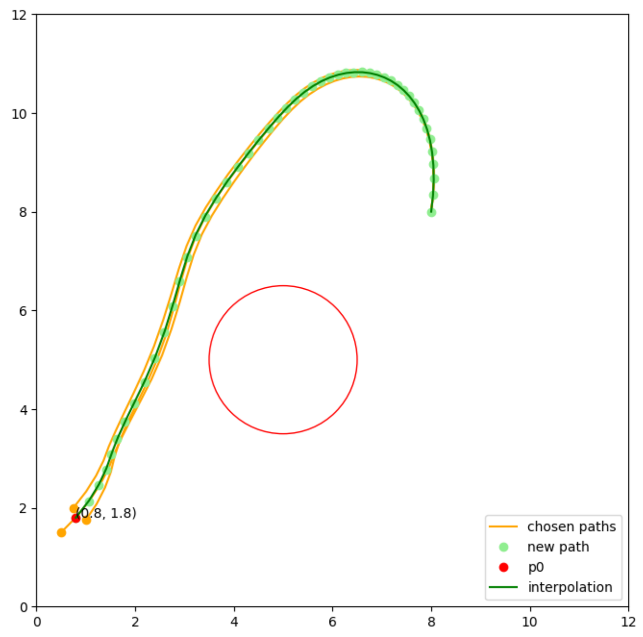
I first compute the separating line (in light blue color) that passes through the center of the circle of fire and the end destination.

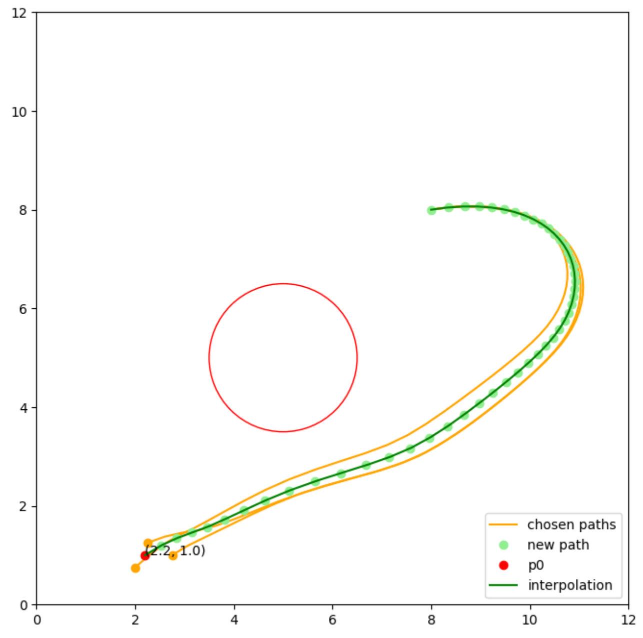
Then for each path, I take the difference between the points' y value and the y value on the separating line. if the average difference is greater than 0, then the path is an upper path (visualized in green color, otherwise it is a lower path (visualized in blue color)

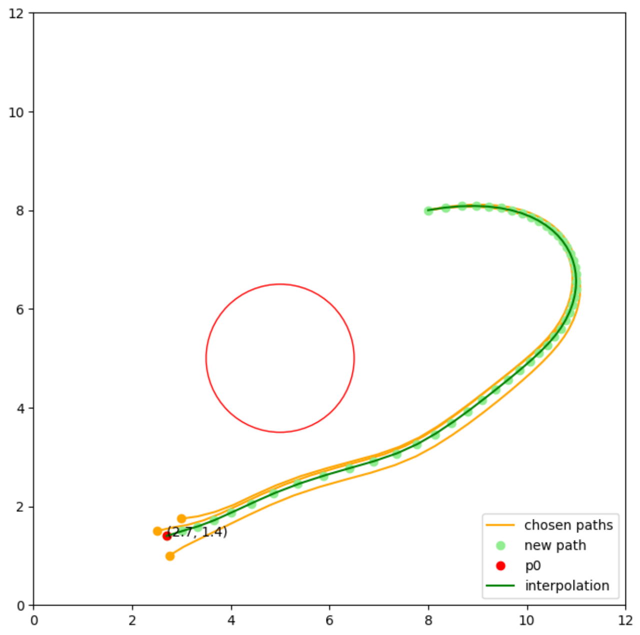


(d)

Output for new path and for continuous t's interpolation:







0
2
4
6
8
10
12
(2.7, 1.4)
chosen paths
new path
p0
interpolation

(e)

Current algorithm, under simple situations, ensure that every new point mostly fall within the triangle at each timestep, allowing the new path to follow the chosen paths. Plus, the piece-wise linear interpolation ensures that points not on the recorded timesteps still stay close to the approximated path.

Therefore, this algorithm can be generally applicable as long as

a. we can identify 3 chosen paths that avoid obstacles in the same way, meaning, if there are more obstacles, all 3 chosen paths should avoid every obstacle in the same direction.

b. the chosen paths don't significant intersect with and diverge from each other, the approximation can be sufficiently accurate.

This is because the weights are initialized only once, calculated based on the relative locations of the 3 starting points. If in a certain timestep t, there is a significant change in the relative locations between the 3 points, the initial weights might not approximate well the p(t) anymore. Because of the scaling and flipping of the 3 points' relative location, the approximated new point might no longer fall inside the triangle.

What need to to be modified when obstacles increase:

1. We need a new method for splitting the paths into batches. It is no longer a simple binary separation anymore, as there is a decision tree of which direction to take when avoiding multiple obstacles down the way.

2. We need to deal with the situation when we cannot successfully identify 3 paths for interpolation, because there might not be enough data for each batch of paths once the obstacle number increases. The initial point might not fall with any triangle at the initial timestep.

3. Increase of obstacles might lead to the data paths becoming more complex with more intersections and divergence. Approximating the weights from the initial point might not be accurate enough for generating a whole path. A new approximation strategy should be employed. There might be a need for multiple approximations for each new path.