Exercises

6.5

(1) To determine where the critical points are located, we need to determine the points where both partial first order derivatives, Fx and Fy, are equal to zero.

(2) To determine whether the critical points are saddle points we need to check whether, Fxx*Fyy - (Fxy)^2 < 0 for the point (a, b). Otherwise the point is a minima/maxima.

(3) To determine if a point is a maxima, we need to check whether both Fxx < 0 and Fyy < 0. Otherwise the point is a minima.

(a)

The function is: F = x^2 - 4xy + y^2

(1)
The first order partial derivatives are:

1. Fx = 2x - 4y
2. Fy = 2y - 4x

The points where both partial derivatives are equal to zero are (x, y) such that x = y.
So, all pairs such that x = y are critical points.

(2)
The second order partial derivatives are:

1. Fxx = 2
2. Fyy = 2
3. Fxy = -4
4. Fyx = -4

The value of the expression at (x, y) such that x = y:

- Fxx*Fyy - Fxy^2 = 2*2 - (-4)^2 = 2 - 16 = -14 < 0

Since the expression above is always less than zero all of these critical points must be saddle.
Therefore, all pairs (x, y) such that x = y are saddle points.

(b)

The function is: F = x^4 - 4xy + y^4

(1)
The first order partial derivatives are:

1. Fx = 4x^3 - 4y
2. Fy = 4y^3 - 4x

The points where both partial derivatives are equal to zero are (0, 0) and (1, 1).
So, the points (0, 0) and (1, 1) are critical points.

(2)
The second order partial derivatives are:

1. Fxx = 12x^2
2. Fyy = 12y^2
3. Fxy = -4
4. Fyx = -4

The value of the expression at (0, 0):

- Fxx*Fyy - Fxy^2 = 12x^2*12y^2 - (-4)^2 = 12x^2*12y^2 - 16 = -16 < 0

Since the expression above is less than zero (0, 0) must be saddle.
Therefore, (0, 0) is a saddle point.

The value of the expression at (1, 1):

- Fxx*Fyy - Fxy^2 = 12x^2*12y^2 - (-4)^2 = 12x^2*12y^2 - 16 = 144 - 16 = 128 > 0

Since the expression above is greater than zero (1, 1) must be a minima or maxima.

(3)
Note that both Fxx and Fyy are greater than zero at (1, 1), mean the point must be a minima
Therefore, (1, 1) is a minima.

(c)

The function is: F = 2x^3 - 3x^2 - 6xy(x - y - 1)

(1)
The first order partial derivatives are:

1. Fx = 6x^2 - 6x - 12yx + 6y^2 + 6y
2. Fy = -6x^2 + 12xy + 6x

The points where both partial derivatives are equal to zero are (-1, -1), (0, -1), (0, 0), and (1, 0)
So, the points (-1, -1), (0, -1), (0, 0), and (1, 0) are critical points.

(2)
The second order partial derivatives are:

1. Fxx = 12x - 6 -12y
2. Fyy = 12x
3. Fxy = -12x + 12y + 6
4. Fyx = -12x + 12y + 6

The value of the expression at (0, 0):

- Fxx*Fyy - (Fxy)^2 = (12x - 6 -12y)12x - (-12x + 12y + 6)^2 = -(-6)^2 = -36 < 0

Since the expression above is less than zero (0, 0) must be saddle.
Therefore, (0, 0) is a saddle point.

The value of the expression at (0, -1):

- Fxx*Fyy - (Fxy)^2 = (12x - 6 -12y)12x - (-12x + 12y + 6)^2 = -(6)^2 = -36 < 0

Since the expression above is less than zero (0, -1) must be saddle.
Therefore, (0, -1) is a saddle point.

The value of the expression at (-1, -1):

- Fxx*Fyy - (Fxy)^2 = (12x - 6 -12y)12x - (-12x + 12y + 6)^2 = -12(-12 - 6 + 12) - (12 - 12 + 6)^2 = 72 - 6^2 = 36 > 0

Since the expression above is greater than zero (-1, -1) must be a minima or maxima.

(3)
Fxx = 12x - 6 - 12y = -12 - 6 + 12 = -6 < 0
Fyy = 12x = -12 < 0
Since both second partial derivatives are less than zero at (-1, -1) the point must be a maxima.
Therefore, (-1, -1) is a maxima.

The value of the expression at (1, 0):

- Fxx*Fyy - (Fxy)^2 = (12x - 6 -12y)12x - (-12x + 12y + 6)^2 = 12(12 - 6) - (-12 + 6)^2 = 12*6 - (-6)^2 = 72 - 36 = 36 > 0

Since the expression above is greater than zero (1, 0) must be a minima or maxima.

(3)
Fxx = 12x - 6 - 12y = 12 - 6 = 6 > 0
Fyy = 12x = 12 > 0
Since both second partial derivatives are greater than zero at (1, 0) the point must be a minima.
Therefore, (1, 0) is a minima.

(d)

The function is: F = (x - y)^4 + x^2 - y^2 - 2x + 2y + 1

(1)
The first order partial derivatives are:

1. Fx = 4(x - y)^3 + 2x - 2
2. Fy = -4(x - y)^3 - 2y + 2

The point where both partial derivatives are equal to zero is (1, 1)
So, the point (1, 1) is a critical point.

(2)
The second order partial derivatives are:

1. Fxx = 12(x - y)^2 + 2
2. Fyy = 12(x - y)^2 - 2
3. Fxy = -12(x - y)^2
4. Fyx = -12(x - y)^2

The value of the expression at (1, 1):

- Fxx*Fyy - (Fxy)^2 = (12(x - y)^2 + 2)(12(x - y)^2 - 2) - (-12(x - y)^2)^2 = (2)(-2) = -4 < 0

Since the expression above is less than zero (1, 1) must be saddle.
Therefore, (1, 1) is a saddle point.

6.8

(a)

The function is: F = 0.5(x^2 - y)^2 + 0.5(1 - x)^2

(1)
The first order partial derivatives are:

1. Fx = 2x(x^2 - y) - (1 - x) = 2x^3 - 2xy - 1 + x
2. Fy = -(x^2 - y) = -x^2 + y

The point where both partial derivatives are equal to zero is (1, 1)
So, the point (1, 1) is a critical point.

(2)
The second order partial derivatives are:

1. Fxx = 6x^2 - 2y + 1
2. Fyy = 1
3. Fxy = -2x
4. Fyx = -2x

The value of the expression at (1, 1):

- Fxx*Fyy - (Fxy)^2 = 6x^2 - 2y + 1 - (-2x)^2 = 6 - 2 + 1 - (-2)^2 = 5 - 4 = 1 > 0

Since the expression above is greater than zero (1, 1) must be a minima or maxima.

(3)
Fxx = 6x^2 - 2y + 1 = 6 - 2 + 1 = 5 > 0
Fyy = 1 > 0
Since both second partial derivatives are greater than zero at (1, 1) the point must be a minima.
Therefore, (1, 1) is a minima.

(b)

The Hessian matrix is,

```
H_f(x) = [ (6x - 2y + 1)     -2x ]
         [ -2x                 1  ]
```

We need to obtain H(x_k)s_k = -f'(x_k),

```
[21  -4][x] = [-9]
[-4    1][y]    [ 2]
```

The can obtain the inverse of the matrix and find that,

```
H^-1 = 1/5 [1     4]
                  [21   4]
```

So,

```
dx_2 = [1    4][-9] = 1/5 [-1]
           [21  4][ 2]           [ 6]
```

Therefore we obtain,
dx_2 + x_1 = [9/5 16/5]^T

(c)

This step is good in a sense because the derivative is near to zero at the new point.

(d)

This step is bad in a sense because the new point is further away from the actual solution.

7.1

(a)

Monomial basis

```
[1 -1  1]    [x0]    [1]
[1  0  0] = [x1] = [0]
[1  1  1]    [x2]    [1]
```

After gaussian elimination we obtain

```
[x0]    [0]
[x1] = [0]
[x2]    [1]
```

Therefore, f(x) = x^2
(b)
Lagrange basis
The lagrange equation using degree two is as follows.

```
f(x) = y1 (x-x2)(x-x3)/(x1-x2)(x1-x3)
         + y2 (x-x1)(x-x3)/(x2-x1)(x2-x3)
         + y3 (x-x1)(x-x2)/(x3-x1)(x3-x2)
```

f(x) = 1*(x-x2)(x-x3)/(x1-x2)(x1-x3)
+ 0*(x-x1)(x-x3)/(x2-x1)(x2-x3)
+ 1*(x-x1)(x-x2)/(x3-x1)(x3-x2)
= 1*(x-0)(x-1)/(-1-0)(-1-1) + 1*(x+1)(x-0)/(1+1)(1-0)
= t(t - 1)/2 + t(t + 1)/2
= 2t^2/2 = t^2

(c)

Newton basis
The linear system for newton interpolation is as follows.

[1 0 0]   [x0]   [1]
[1 1 0] = [x1] = [0]
[1 2 2]   [x2]   [1]

Using forward substitution it is easy to see that,

[x0]   [1]
[x1] = [-1]
[x2]   [1]

Then the polynomial is.
f(x) = 1 - (t + 1) + t(t + 1) = 1 - t - 1 + t^2 + t = t^2.

Therefore, each method gives the same polynomial representation of t^2.

7.4

(a)

Monomial Basis
With Horner's method, Pn-1(t) = x1 + t(x2 + t(.....(xn-1 + xnt)...)).
This requires (n - 1) multiplications.

(b)

Lagrange Basis
Pn-1(t) =
=y1((t-t2)(t-t3)...(t-tn))/((t1-t2)(t1-t3)...(t1-tn)) +...+ yn ((t-t1)(t-t2)...(t-tn-1))/((tn-t1)(tn-t2)...(tn-tn-1))

For a given polynomial where the denominator are constants we can obtain:

Pn-1(t)= c1(t-t2)(t-t3)...(t-tn)+...+cn(t-t1)(t-t2)...(t-tn-1)

Where each basis function requires n multiplications.
Since there are n basis function then a total of (n*n = n^2) multiplications.

(c)

Newton Basis
With Horner's method, Pn-1(t) = x1 + (t - t1)(x2 + (t - t2)(.....(xn-1(t - tn-1)).....))
This requires (n - 1) multiplications.

7.5

(a)

Monomial Basis
The monomial matrix is:

```
[1   1    1     1][a0]    [ 11 ]
[1   2    4     8][a1] = [ 29 ]
[1   3    9   27][a2]    [ 65 ]
[1   4   16  64][a3]    [125]
```

Solving this system we obtain,

```
[1   0    0    0][5]
[0   1    0    0][2]
[0   0    1    0][3]
[0   0    0    1][1]
```

So, a = [5 2 3 1]^T and the equation is

f(t) = 5 + 2t + 3t^2 + t^3

(b)

Lagrange Basis

L(t) = y0*l0(t) + y1*l1(t) + y2*l2(t) + y3*l3(t)
li(t) = Product{i != j} ((t - tj)/(ti - tj))

So,
l0(t)
=> ((t - t1)(t - t2)(t - t3))/((t0 - t1)(t0 - t2)(t0 - t3))
=> ((t - 2)(t - 3)(t - 4))/((1 - 2)(1 - 3)(1 - 4))
=> ((t - 2)(t - 3)(t - 4))/(-6)

l1(t)
=> ((t - t0)(t - t2)(t - t3))/((t1 - t0)(t1 - t2)(t1 - t3))
=> ((t - 1)(t - 3)(t - 4))/(2)

l2(t)
=> ((t - t0)(t - t1)(t - t3))/((t2 - t0)(t2 - t1)(t2 - t3))
=> ((t - 1)(t - 2)(t - 4))/(-2)

l3(t)
=> ((t - t0)(t - t1)(t - t2))/((t3 - t0)(t3 - t1)(t3 - t2))
=> ((t - 1)(t - 2)(t - 3))/(6)

After simplifying the numerators above we obtain,

f(t) = 5 + 2t + 3t^2 + t^3

(c)

Newton Basis
f(t) = a0 + a1(t - 1) + a2(t-1)(t-2) + a2(t-1)(t-2)(t-3)

The lower triangular method the matrix is,

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & t1-t0 & 0 & 0 \\ 1 & t2-t0 & (t2-t0)(t2-t1) & 0 \\ 1 & t3-t0 & (t3-t0)(t2-t1) & (t3-t0)(t3-t1)(t3-t2) \end{bmatrix}\begin{bmatrix} a0 \\ a1 \\ a2 \\ a3 \end{bmatrix} = \begin{bmatrix} y0 \\ y1 \\ y2 \\ y3 \end{bmatrix}$$

Now substituting in,

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 2-1 & 0 & 0 \\ 1 & 3-1 & (3-1)(3-2) & 0 \\ 1 & 4-1 & (4-1)(3-2) & (4-1)(4-2)(4-3) \end{bmatrix}\begin{bmatrix} a0 \\ a1 \\ a2 \\ a3 \end{bmatrix} = \begin{bmatrix} 11 \\ 29 \\ 65 \\ 125 \end{bmatrix}$$

Simplifying,

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 2 & 2 & 0 \\ 1 & 3 & 3 & 6 \end{bmatrix}\begin{bmatrix} a0 \\ a1 \\ a2 \\ a3 \end{bmatrix} = \begin{bmatrix} 11 \\ 29 \\ 65 \\ 125 \end{bmatrix}$$

After reducing the matrix we obtain,

a = [11 18 9 1]^T

After substituting into the equation above we obtain,

f(t) = 5 + 2t + 3t^2 + t^3

8.1

(a)

Midpoint Rule:
The midpoint of 0 and 1 is 0.5.
f(x) = x^3 at x = 0.5 is equal to 0.125
The interval length is 1
So the value of the integral rule using the midpoint rule is 1*0.125 = 0.125

Trapezoid Rule:
First we calculate at the end points of 0 and 1
f(0) = 0 and f(1) = 1
This is just a right triangle with base 1 and height 1.
Therefore the value of the integral is 1*1*0.5 = 0.5

(b)

The true value of the integral is 0.25 and the error of an approximation is |approx - true| / true.
So,

Midpoint Rule:
|0.125 - 0.25| / 0.25 = 0.125 / 0.25 = 0.5
Therefore the % error is 50%

Trapezoid Rule:
|0.5 - 0.25| / 0.25 = 0.25 / 0.25 = 1
Therefore the % error is 0%

(c)

Using simpson's rule we find that the solution is as follows,
.5/3*(f(0) + 4*f(.5) + f(1)) = .5/3*(0 + 4*0.125+1) = 0.5/3(.5+1) = 0.5/3*(1.5) = .5*0.75 = 0.375

(d)

No, we do not expect the results of this problem to be more accurate since the problem is monotonically increasing. Since the simpson's rule uses parabolas to approximate the sum it will over estimate the solution in this case.

8.7

The quadrature rule for the open two point newton cotes formula on (a, b) is as follows,

Since the data is open we know that

- x_i = a + i*(b - a)/(n+1)

Since n = 2, we can obtain

- x_1 = a + 1*(b - a)/3 = (2a + b)/3
- x_2 = a + 2*(b - a)/3 = (a + 2b)/3

To solve for the weight we look at:

- w_1*f(x_1) + w_2*f(x_2) = integral{a to b} f(x) dx

First we let f(x) = 1 and obtain

- w_1 + w_2 = b - a

Second we let f(x) = x and obtain

- w_1*x_1 + w_2*x_2 = b^2/2 - a^2/2

By substituting the points obtained above into the second equation we obtain

- w_1*(2a + b)/3 + w_2*(a + 2b)/3 = b^2/2 - a^2/2
- w_1*(2a + b) + w_2*(a + 2b) = 3*b^2/2 - 3*a^2/2
- a(2*w_1 + w_2) + b(w_1 + 2*w_2) = a(-1.5*a) + b(1.5*b)

Since we were able to isolate the equation into like terms we obtain two new equations and using the first equation obtained we get:

1. 2*w_1 + w_2 = -1.5*a
2. w_1 + 2*w_2 = 1.5*b
3. w_1 + w_2 = b - a

By moving terms in equation 3 we get:

- w_1 = b - a - w_2

By substituting that into equation 2 we get:

- b - a - w_2 + 2*w_2 = 1.5*b
- w_2 = 0.5*b + a

Now by substituting that result into equation 1 we get:

- 2*w_1 + 0.5*b + a = -1.5*a
- 2*w_1 = -2.5*a - 0.5*b
- w_1 = -1.25*a - 0.25*b

Therefore,
The points are:

- x_1 = a + 1*(b - a)/3 = (2a + b)/3
- x_2 = a + 2*(b - a)/3 = (a + 2b)/3

The weights are:

- w_1 = -1.25*a - 0.25*b
- w_2 = 0.5*b + a

The equation is:

- (-1.25*a - 0.25*b) * f((2a + b)/3) + (0.5*b + a)*f((a + 2b)/3)

Since we are only using two points the degree of the resulting rule is 2.

8.11

(a)

The interval is [0, 3]
The function is f(x) = x^2
True integral value = 9

Trapezoid:
approx = (f(1) * dx)/2 + ((f(1) * dx) + ((f(2)-f(1)) * dx)/2)
= (1*1)/2 + ((1*1) + ((4-1)*1)/2)
= 0.5 + (1 + 1.5) = 3

Error = (9 - 3)/(9) * 100% = 66%

Simpson's rule:
approx = 3/8[f(0) + 3*f(1) + 3*f(2) + f(3)]
= 3/8[0 + 3*1 + 3*4 + 9]
= 3/8[24] = 9

Error = (9 - 9)/(9) * 100% = 0%

This shows that the simpsons rule gives a more accurate approximation

(b)

The interval is [0, 4]
The function is f(x) = x^3
True integral value = 64

Simpson's rule:
approx = 1/8[f(0) + 3*f(0.67) + 3*f(1.34) + f(2)] + 1/8[f(2) + 3*f(2.67) + 3*f(3.34) + f(4)]
= 1/8[0 + 3*0.3 + 3*2.4 + 8] + 1/8[8 + 3*19 + 3*37.3 + 64]
= 1/8[0.9 + 7.2 + 8] + 1/8[8 + 57 + 111.9 + 64]
= 1/8[16.1] + 1/8[240.9]
= 8.1 + 30.1 = 38.2

Error = (64 - 38.2)/(64) * 100% = 40.3%

Open Newton-Cotes formula of degree 5:
approx = 1/6[11*f(0.8) + f(1.6) + f(2.4) + 11*f(3.2)]
= 1/6[11*0.5 + 4.1 + 13.8 + 11*32.8]
= 1/6[5.5 + 4.1 + 13.8 + 360.4]
= 1/6[383.8] = 63.97 ~ 64

Error = (64 - 64)/(64) * 100% = 0%

This shows that the open Newton-Cotes formula gives a more accurate approximation.

8.12

Averaged function:
$f'(x) = f(x + h) - f(x - h) / 2h$

So we can do some error analysis on both part of the numerator using Taylor series

1. $f(n + h) = f(n) + f'(n)*h + f''(n)h^2/2 + f'''(n)h^3/3! + ...$
2. $f(n - h) = f(n) - f'(n)*h + f''(n)h^2/2 - f'''(n)h^3/3! + \ldots$

Now by subtracting we obtain
$f(x + h) - f(x - h) = 2*f'(n)*h + 2*f'''(n)*h^3/3! + \ldots.$
$f(x + h) - f(x - h) / 2h = f'(n) + f'''(n)*h^2/3! + f'''''(n)*h^4/5! + \ldots.$

Now we can subtract the derivative

$f(x + h) - f(x - h) / 2h - f'(n) = f'''(n)*h^2/3! + f'''''(n)*h^4/5! + \ldots. = b1*h^2 + b2*h^4 + \ldots$

Hence the Error $E = O(h)$
Since,
$E = b1*h^2 + b2(h^2)^2 + b3(h^2)^3$

Therefore the accuracy of the averaged function is order 2.

Computer Problems

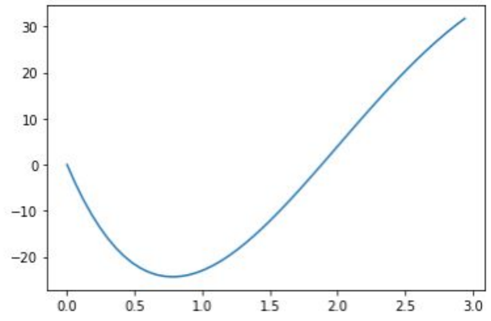
6.3

(a)

```
[2] # function for math function
    def derive_seta(n=50, a=0, b=3):
      # step size and list declarations
      step_size = (b - a) / n
      x_list = list()
      y_list = list()

      # loop through each step in the domain
      for i in range(n):
        # calculate the current position and its function value
        temp_x = a + i*step_size
        temp_y = temp_x**4 - 14*temp_x**3 + 60*temp_x**2 - 70*temp_x
        # append current x and y to their lists
        x_list.append(temp_x)
        y_list.append(temp_y)

      # determine the minimum y value
      min_value = min(y_list)
      print("The minimum of this function is", min_value)

      # plot the function
      plt.plot(x_list, y_list)
      print("Since the function only has one local minima on the closed interval [0, 3], it is unimodal")
```

```
derive_seta()
```

```
The minimum of this function is -24.369577439999997
Since the function only has one local minima on the closed interval [0, 3], it is unimodal
```

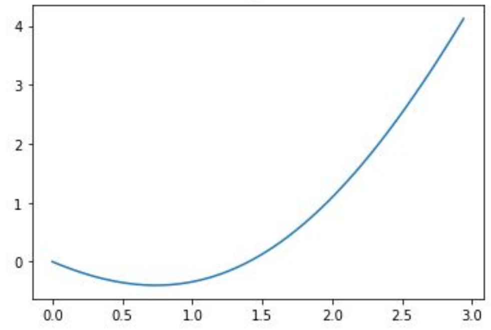


(b)

```
[4] # function for math function
    def derive_setb(n=50, a=0, b=3):
      # step size and list declarations
      step_size = (b - a) / n
      x_list = list()
      y_list = list()

      # loop through each step in the domain
      for i in range(n):
        # calculate the current position and its function value
        temp_x = a + i*step_size
        temp_y = 0.5*temp_x**2 - math.sin(temp_x)
        # append current x and y to their lists
        x_list.append(temp_x)
        y_list.append(temp_y)

      # determine the minimum y value
      min_value = min(y_list)
      print("The minimum of this function is", min_value)

      # plot the function
      plt.plot(x_list, y_list)
      print("Since the function only has one local minima on the closed interval [0, 3], it is unimodal")
```

```
derive_setb()
```

```
The minimum of this function is -0.4001846719714731
Since the function only has one local minima on the closed interval [0, 3], it is unimodal
```

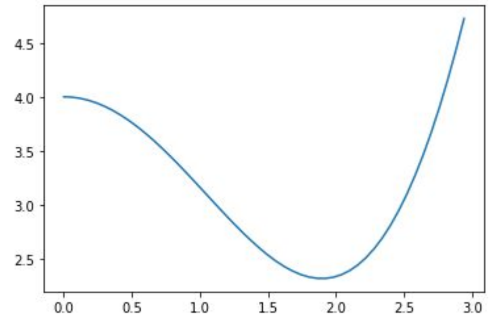


(c)

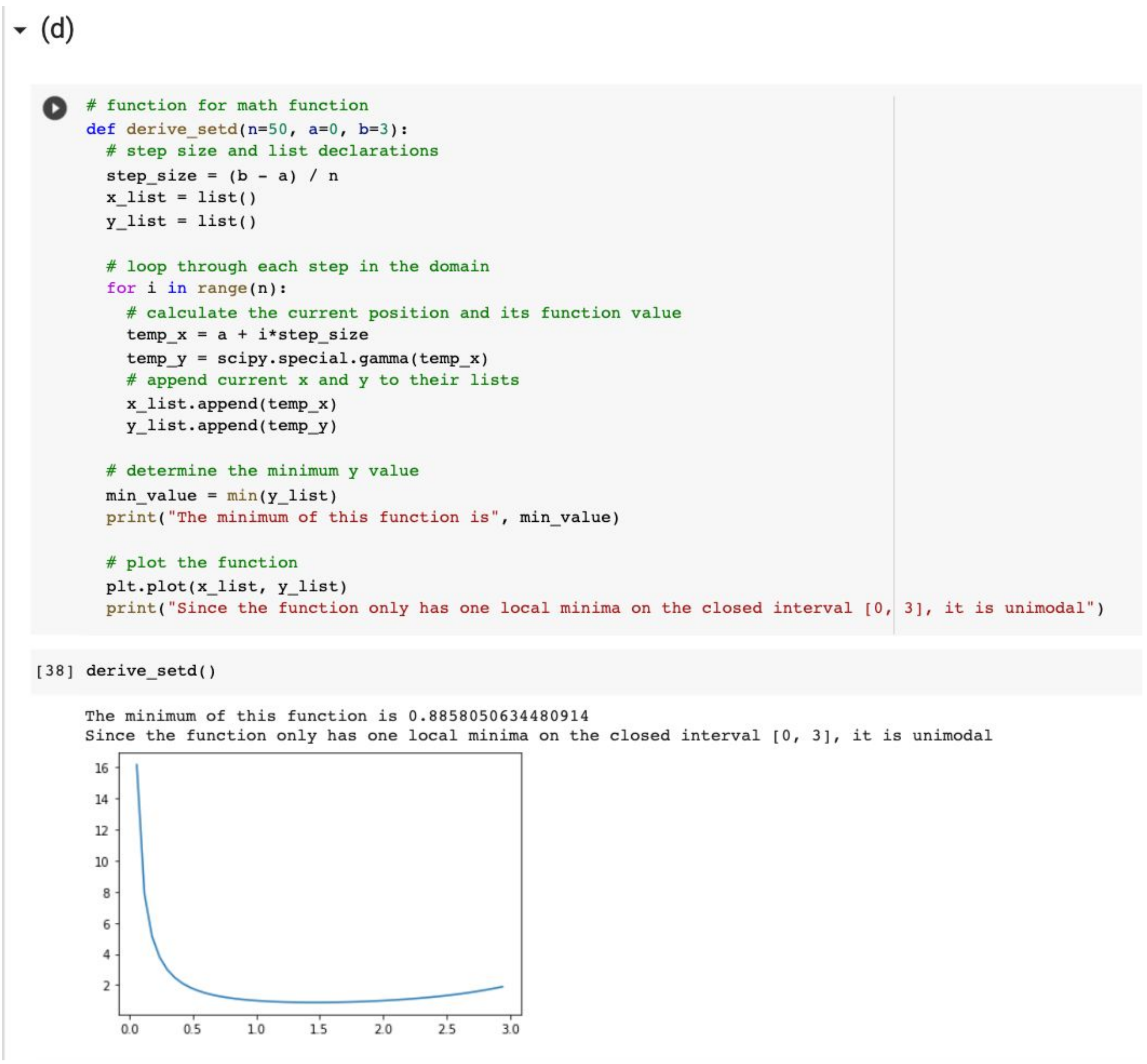
```
# function for math function
def derive_setc(n=50, a=0, b=3):
  # step size and list declarations
  step_size = (b - a) / n
  x_list = list()
  y_list = list()

  # loop through each step in the domain
  for i in range(n):
    # calculate the current position and its function value
    temp_x = a + i*step_size
    temp_y = temp_x**2 + 4*math.cos(temp_x)
    # append current x and y to their lists
    x_list.append(temp_x)
    y_list.append(temp_y)

  # determine the minimum y value
  min_value = min(y_list)
  print("The minimum of this function is", min_value)

  # plot the function
  plt.plot(x_list, y_list)
  print("Since the function only has one local minima on the closed interval [0, 3], it is unimodal")
```

```
[7] derive_setc()
```

```
The minimum of this function is 2.3178013953964074
Since the function only has one local minima on the closed interval [0, 3], it is unimodal
```



(d)



```
# function for math function
def derive_setd(n=50, a=0, b=3):
  # step size and list declarations
  step_size = (b - a) / n
  x_list = list()
  y_list = list()

  # loop through each step in the domain
  for i in range(n):
    # calculate the current position and its function value
    temp_x = a + i*step_size
    temp_y = scipy.special.gamma(temp_x)
    # append current x and y to their lists
    x_list.append(temp_x)
    y_list.append(temp_y)

  # determine the minimum y value
  min_value = min(y_list)
  print("The minimum of this function is", min_value)

  # plot the function
  plt.plot(x_list, y_list)
  print("Since the function only has one local minima on the closed interval [0, 3], it is unimodal")
```

```
[38] derive_setd()
```

```
The minimum of this function is 0.8858050634480914
Since the function only has one local minima on the closed interval [0, 3], it is unimodal
```

6.7

(a)

The function is: F = 2x^3 - 3x^2 - 6xy(x - y - 1)

(1)
The first order partial derivatives are:

3. Fx = 6x^2 - 6x - 12yx + 6y^2 + 6y
4. Fy = -6x^2 + 12xy + 6x

The points where both partial derivatives are equal to zero are (-1, -1), (0, -1), (0, 0), and (1, 0)
So, the points (-1, -1), (0, -1), (0, 0), and (1, 0) are critical points.

(b)

(2)
The second order partial derivatives are:

5. Fxx = 12x - 6 -12y
6. Fyy = 12x
7. Fxy = -12x + 12y + 6
8. Fyx = -12x + 12y + 6

The value of the expression at (0, 0):

- Fxx*Fyy - (Fxy)^2 = (12x - 6 -12y)12x - (-12x + 12y + 6)^2 = -(-6)^2 = -36 < 0

Since the expression above is less than zero (0, 0) must be saddle.
Therefore, (0, 0) is a saddle point.

The value of the expression at (0, -1):

- Fxx*Fyy - (Fxy)^2 = (12x - 6 -12y)12x - (-12x + 12y + 6)^2 = -(6)^2 = -36 < 0

Since the expression above is less than zero (0, -1) must be saddle.
Therefore, (0, -1) is a saddle point.

The value of the expression at (-1, -1):

- Fxx*Fyy - (Fxy)^2 = (12x - 6 -12y)12x - (-12x + 12y + 6)^2 = -12(-12 - 6 + 12) - (12 - 12 + 6)^2 = 72 - 6^2 = 36 > 0

Since the expression above is greater than zero (-1, -1) must be a minima or maxima.

(3)
Fxx = 12x - 6 - 12y = -12 - 6 + 12 = -6 < 0
Fyy = 12x = -12 < 0
Since both second partial derivatives are less than zero at (-1, -1) the point must be a maxima.
Therefore, (-1, -1) is a maxima.

The value of the expression at (1, 0):

- Fxx*Fyy - (Fxy)^2 = (12x - 6 -12y)12x - (-12x + 12y + 6)^2 = 12(12 - 6) - (-12 + 6)^2 = 12*6 - (-6)^2 = 72 - 36 = 36 > 0

Since the expression above is greater than zero (1, 0) must be a minima or maxima.

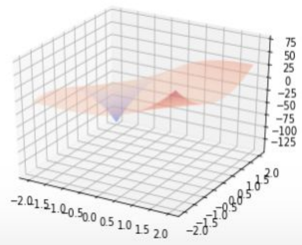
(3)
Fxx = 12x - 6 - 12y = 12 - 6 = 6 > 0
Fyy = 12x = 12 > 0
Since both second partial derivatives are greater than zero at (1, 0) the point must be a minima.
Therefore, (1, 0) is a minima.

(c)

```python
# definition of function
def f(xy):
    x = xy[0]
    y = xy[1]
    val = 2.0*x**3 - 3.0*x**2 - 6.0*y*x**2 + 6.0*x*y**2 + 6.0*x*y
    return val

# definition of derivative
def df(xy):
    x = xy[0]
    y = xy[1]
    val1 = 6.0*x**2 - 6.0*x - 12.0*y*x + 6.0*y**2 + 6.0*y
    val2 = -6.0*x**2 + 12.0*x*y + 6.0*x
    return np.array([val1, val2])

# function to find the minima
def find_minima(starting_point, f, df):
  # initialize threshold, learning rate, and initial guess
  threshold = np.array([1e-8, 1e-8])
  lr = 0.001
  loop_cnt = 0
  max_loops = 1000
  guesses = [starting_point]

  # loop until convergence is found
  while True:
    # grab the initial guess
    x = guesses[-1]

    # solve for the new gradient and append it to the list
    s = x - lr*df(x)
    guesses.append(s)

    # break if the guesses are the same
    if np.all((guesses[-1] - guesses[-2]) < threshold):
      break

    # break if infinitly looping
    loop_cnt+=1
    if loop_cnt > max_loops:
      break

  # print the minima
  min_point = guesses[-1]
  min_val = f(min_point)

  if loop_cnt > max_loops and min_val < 0:
    print("The minimum starting at point [", starting_point[0], ",", starting_point[1], "] is -inf")
  elif loop_cnt > max_loops and min_val > 0:
    print("The minimum starting at point [", starting_point[0], ",", starting_point[1], "] is +inf")
  else:
    print("The minimum starting at point [", starting_point[0], ",", starting_point[1], "] is", min_val)
```

[40]
```python
fig = plt.figure()
ax = fig.gca(projection="3d")

xmesh, ymesh = np.mgrid[-2:2:50j,-2:2:50j]
fmesh = f(np.array([xmesh, ymesh]))
ax.plot_surface(xmesh, ymesh, fmesh, alpha=0.4, rstride=1, cstride=1, cmap=plt.cm.coolwarm, linewidth=0.5, antialiased=True, zorder = 0.5)
```

```
<mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7f26d4f477f0>
```



(d)

```
[41] # define starting points
     starting_point_1 = np.array([1, 1])
     starting_point_2 = np.array([-1, 1])
     starting_point_3 = np.array([1, -1])
     starting_point_4 = np.array([-1, -1])
     starting_point_5 = np.array([0, 0])
     starting_point_6 = np.array([1, 0])
     starting_point_7 = np.array([-1, 0])
     starting_point_8 = np.array([0, 1])
     starting_point_9 = np.array([0, -1])

     # find the minima from each starting point
     find_minima(starting_point_1, f, df)
     find_minima(starting_point_2, f, df)
     find_minima(starting_point_3, f, df)
     find_minima(starting_point_4, f, df)
     find_minima(starting_point_5, f, df)
     find_minima(starting_point_6, f, df)
     find_minima(starting_point_7, f, df)
     find_minima(starting_point_8, f, df)
     find_minima(starting_point_9, f, df)
```

```
The minimum starting at point [ 1 , 1 ] is 4.856864
The minimum starting at point [ -1 , 1 ] is -inf
The minimum starting at point [ 1 , -1 ] is -inf
The minimum starting at point [ -1 , -1 ] is 1.0
The minimum starting at point [ 0 , 0 ] is 0.0
The minimum starting at point [ 1 , 0 ] is -1.0
The minimum starting at point [ -1 , 0 ] is -inf
The minimum starting at point [ 0 , 1 ] is -0.14529945600000002
The minimum starting at point [ 0 , -1 ] is 0.0
```

6.9

(a) Steepest Descent

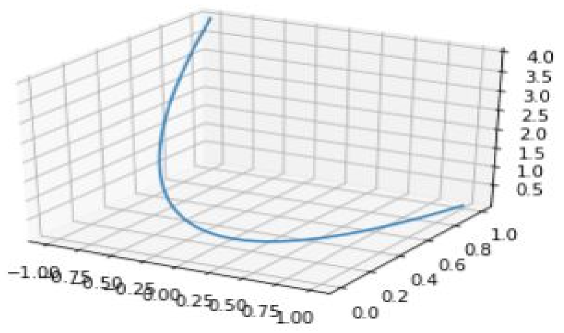
[13]

```
# definition of Rosenbrock's function
def f(xy):
    x = xy[0]
    y = xy[1]
    val = 100.0 * (y - x**2)**2 + (1.0 - x)**2
    return val

def df(xy):
    x = xy[0]
    y = xy[1]
    val1 = -400.0 * x * (y - x**2) - 2 * (1 - x)
    val2 = 200.0 * (y - x**2)
    return np.array([val1, val2])
```

[14]

```
# initialize threshold, learning rate, and initial guess
threshold_sd_1 = np.array([1e-8, 1e-8])
lr_sd_1 = 0.0001
guesses_sd_1 = [np.array([-1, 1])]

# loop until convergence is found
while True:
  # grab the initial guess
  x_sd_1 = guesses_sd_1[-1]

  # solve for the new gradient and append it to the list
  s_sd_1 = x_sd_1 - lr_sd_1*df(x_sd_1)
  guesses_sd_1.append(s_sd_1)

  # break if the guesses are the same
  if np.all((guesses_sd_1[-1] - guesses_sd_1[-2]) < threshold_sd_1):
    break

# plot the path
X_sd_1 = [guess[0] for guess in guesses_sd_1]
Y_sd_1 = [guess[1] for guess in guesses_sd_1]
Z_sd_1 = [f(guess) for guess in guesses_sd_1]
fig = plt.figure()
ax = fig.gca(projection="3d")
ax.plot(X_sd_1, Y_sd_1, Z_sd_1)
```

```
[<mpl_toolkits.mplot3d.art3d.Line3D at 0x7f26d156c438>]
```

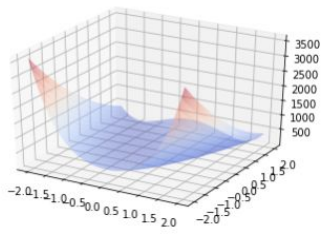


(b) Newton's Method

```
[17] # definition of Rosenbrock's function
     def f(xy):
         x = xy[0]
         y = xy[1]
         val = 100.0 * (y - x**2)**2 + (1.0 - x)**2
         return val

     def df(xy):
         x = xy[0]
         y = xy[1]
         val1 = 400.0 * (y - x**2) * x - 2 * x
         val2 = 200.0 * (y - x**2)
         return np.array([val1, val2])

     def ddf(xy):
         x = xy[0]
         y = xy[1]
         val11 = 400.0 * (y - x**2) - 800.0 * x**2 - 2
         val12 = 400.0
         val21 = -400.0 * x
         val22 = 200.0
         return np.array([[val11, val12], [val21, val22]])
```

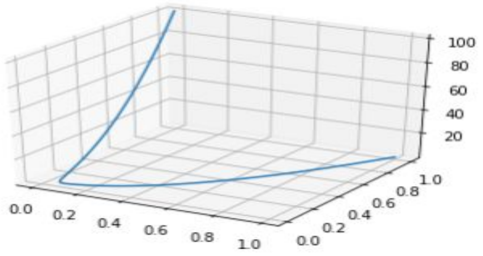
```
fig = plt.figure()
ax = fig.gca(projection="3d")

xmesh, ymesh = np.mgrid[-2:2:50j,-2:2:50j]
fmesh = f(np.array([xmesh, ymesh]))
ax.plot_surface(xmesh, ymesh, fmesh, alpha=0.4, rstride=1, cstride=1, cmap=plt.cm.coolwarm, linewidth=0.5, antialiased=True, zorder = 0.5)
```

```
<mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7f26ce8d4208>
```



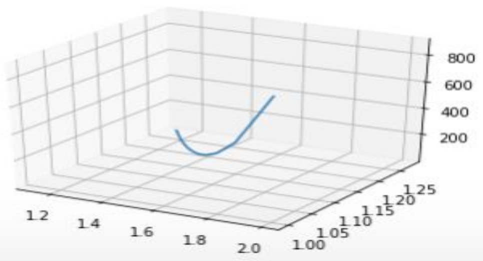
```
[15] # initialize threshold, learning rate, and initial guess
     threshold_sd_2 = np.array([1e-8, 1e-8])
     lr_sd_2 = 0.0001
     guesses_sd_2 = [np.array([0, 1])]

     # loop until convergence is found
     while True:
       # grab the initial guess
       x_sd_2 = guesses_sd_2[-1]

       # solve for the new gradient and append it to the list
       s_sd_2 = x_sd_2 - lr_sd_2*df(x_sd_2)
       guesses_sd_2.append(s_sd_2)

       # break if the guesses are the same
       if np.all((guesses_sd_2[-1] - guesses_sd_2[-2]) < threshold_sd_2):
         break

     # plot the path
     X_sd_2 = [guess[0] for guess in guesses_sd_2]
     Y_sd_2 = [guess[1] for guess in guesses_sd_2]
     Z_sd_2 = [f(guess) for guess in guesses_sd_2]
     fig = plt.figure()
     ax = fig.gca(projection="3d")
     ax.plot(X_sd_2, Y_sd_2, Z_sd_2)
```

```
[<mpl_toolkits.mplot3d.art3d.Line3D at 0x7f26d072ff98>]
```



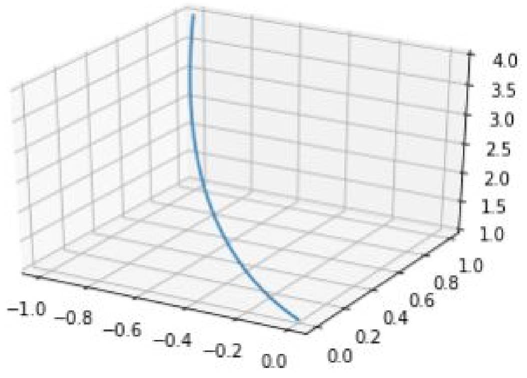
```
# initialize threshold, learning rate, and initial guess
threshold_sd_3 = np.array([1e-8, 1e-8])
lr_sd_3 = 0.0001
guesses_sd_3 = [np.array([2, 1])]

# loop until convergence is found
while True:
  # grab the initial guess
  x_sd_3 = guesses_sd_3[-1]

  # solve for the new gradient and append it to the list
  s_sd_3 = x_sd_3 - lr_sd_3*df(x_sd_3)
  guesses_sd_3.append(s_sd_3)

  # break if the guesses are the same
  if np.all((guesses_sd_3[-1] - guesses_sd_3[-2]) < threshold_sd_3):
    break

# plot the path
X_sd_3 = [guess[0] for guess in guesses_sd_3]
Y_sd_3 = [guess[1] for guess in guesses_sd_3]
Z_sd_3 = [f(guess) for guess in guesses_sd_3]
fig = plt.figure()
ax = fig.gca(projection="3d")
ax.plot(X_sd_3, Y_sd_3, Z_sd_3)
```

```
[<mpl_toolkits.mplot3d.art3d.Line3D at 0x7f26d072fe80>]
```



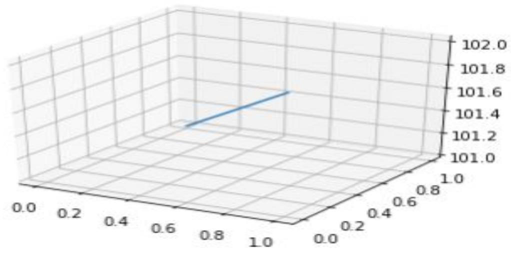
```python
# initialize threshold and initial guess
threshold_new_1 = np.array([1e-8, 1e-8])
guesses_new_1 = [np.array([-1, 1])]

# loop until convergence is found
while True:
  # grab the initial guess
  x_new_1 = guesses_new_1[-1]

  # solve for the new gradient and append it to the list
  s_new_1 = la.solve(ddf(x_new_1), df(x_new_1))
  next_guess_new_1 = x_new_1 - s_new_1
  guesses_new_1.append(next_guess_new_1)

  # break if the guesses are the same
  if np.all((guesses_new_1[-1] - guesses_new_1[-2]) < threshold_new_1):
    break

# plot the path
X_new_1 = [guess[0] for guess in guesses_new_1]
Y_new_1 = [guess[1] for guess in guesses_new_1]
Z_new_1 = [f(guess) for guess in guesses_new_1]
fig = plt.figure()
ax = fig.gca(projection="3d")
ax.plot(X_new_1, Y_new_1, Z_new_1)
```

```
[<mpl_toolkits.mplot3d.art3d.Line3D at 0x7f26ce710438>]
```



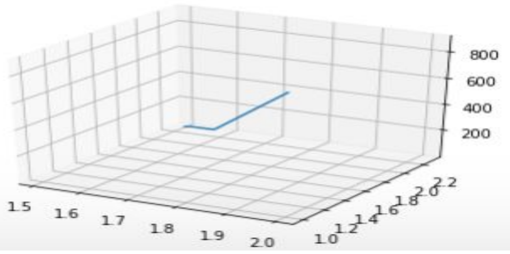
```
# initialize threshold and initial guess
threshold_new_2 = np.array([1e-8, 1e-8])
guesses_new_2 = [np.array([0, 1])]

# loop until convergence is found
while True:
  # grab the initial guess
  x_new_2 = guesses_new_2[-1]

  # solve for the new gradient and append it to the list
  s_new_2 = la.solve(ddf(x_new_2), df(x_new_2))
  next_guess_new_2 = x_new_2 - s_new_2
  guesses_new_2.append(next_guess_new_2)

  # break if the guesses are the same
  if np.all((guesses_new_2[-2] - guesses_new_2[-2]) < threshold_new_2):
    break

# plot the path
X_new_2 = [guess[0] for guess in guesses_new_2]
Y_new_2 = [guess[1] for guess in guesses_new_2]
Z_new_2 = [f(guess) for guess in guesses_new_2]
fig = plt.figure()
ax = fig.gca(projection="3d")
ax.plot(X_new_2, Y_new_2, Z_new_2)
```

```
[<mpl_toolkits.mplot3d.art3d.Line3D at 0x7f26d06dbda0>]
```



```
[21] # initialize threshold and initial guess
threshold_new_3 = np.array([1e-8, 1e-8])
guesses_new_3 = [np.array([2, 1])]

# loop until convergence is found
while True:
  # grab the initial guess
  x_new_3 = guesses_new_3[-1]

  # solve for the new gradient and append it to the list
  s_new_3 = la.solve(ddf(x_new_3), df(x_new_3))
  next_guess_new_3 = x_new_3 - s_new_3
  guesses_new_3.append(next_guess_new_3)

  # break if the guesses are the same
  if np.all((guesses_new_3[-1] - guesses_new_3[-2]) < threshold_new_3):
    break

# plot the path
X_new_3 = [guess[0] for guess in guesses_new_3]
Y_new_3 = [guess[1] for guess in guesses_new_3]
Z_new_3 = [f(guess) for guess in guesses_new_3]
fig = plt.figure()
ax = fig.gca(projection="3d")
ax.plot(X_new_3, Y_new_3, Z_new_3)
```

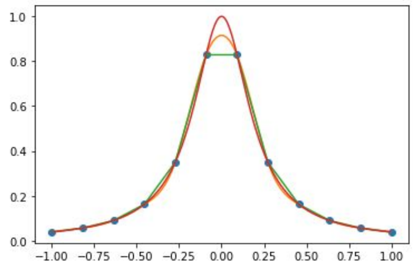
```
[<mpl_toolkits.mplot3d.art3d.Line3D at 0x7f26ce84f5f8>]
```



7.4

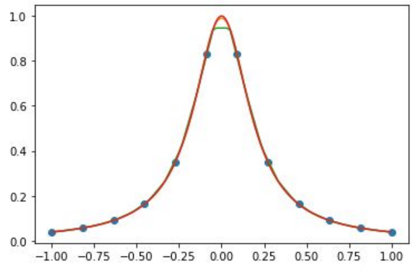
```
[22] # define data for 11/21 equally spaced regions
     x_11 = np.linspace(-1, 1, num=12, endpoint=True)
     y_11 = 1 / (1 + 25*x_11**2)
     x_21 = np.linspace(-1, 1, num=22, endpoint=True)
     y_21 = 1 / (1 + 25*x_21**2)

     # define new domain to plot on
     new_domain = np.linspace(-1, 1, num=100, endpoint=True)

     # determine polynomial spline data for 11/21 equally spaced regions
     ps_11 = interp1d(x_11, y_11, kind='slinear')
     y_ps_11_new = ps_11(new_domain)
     ps_21 = interp1d(x_21, y_21, kind='slinear')
     y_ps_21_new = ps_21(new_domain)

     # determine cubic spline data for 11/21 equally spaced regions
     cs_11 = interp1d(x_11, y_11, kind='cubic')
     y_cs_11_new = cs_11(new_domain)
     cs_21 = interp1d(x_21, y_21, kind='cubic')
     y_cs_21_new = cs_21(new_domain)

     # true function values
     true_x = np.linspace(-1, 1, num=10000, endpoint=True)
     true_y = 1 / (1 + 25*true_x**2)
```

```
[23] # plot the spline
     plt.plot(x_11, y_11, 'o', new_domain, y_cs_11_new, '-', new_domain, y_ps_11_new, '-', true_x, true_y, '-')
     plt.show()
     print("As we can see from the graph above the red line, which represents")
     print("the true function is most closely approximated by the cubic spline")
     print("and the slinear/polynomial spline is below the cubic spline.")
```



```
As we can see from the graph above the red line, which represents
the true function is most closely approximated by the cubic spline
and the slinear/polynomial spline is below the cubic spline.
```

```
[24] # plot the spline
     plt.plot(x_11, y_11, 'o', new_domain, y_cs_21_new, '-', new_domain, y_ps_21_new, '-', true_x, true_y, '-')
     plt.show()
     print("As we can see from the graph above an increase in the number of")
     print("regions increased the approximation of the interpolation methods.")
     print("Although the cubic spline still interpolates better.")
```



```
As we can see from the graph above an increase in the number of
regions increased the approximation of the interpolation methods.
Although the cubic spline still interpolates better.
```

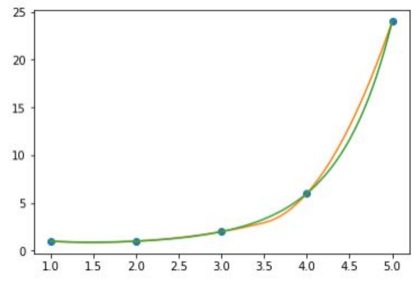
7.7

```
[25] # define data
     x = [1, 2, 3, 4, 5]
     y = [1, 1, 2, 6, 24]
     new_domain = np.linspace(1, 5, num=100, endpoint=True)

     # true function values
     true_x = np.linspace(1, 5, num=10000, endpoint=True)
     true_y = [gamma(x) for x in true_x]
```

(a) Quadratic

```
[26] # quadratic spline
     qs = interp1d(x, y, kind='quadratic')
     y_qs = qs(new_domain)
```

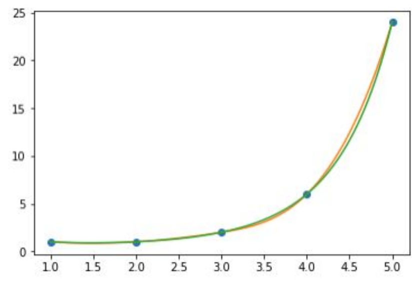
```
[27] # plot the data
     plt.plot(x, y, 'o', new_domain, y_qs, '-', true_x, true_y, '-')
     plt.show()
```



(b) Cubic

```
[28] # quadratic spline
     cs = interp1d(x, y, kind='cubic')
     y_cs = cs(new_domain)
```

```
[29] # plot the data
     plt.plot(x, y, 'o', new_domain, y_cs, '-', true_x, true_y, '-')
     plt.show()
```



(c)

```
[30] print("Based on both graphs above, I can accurately say that the cubic spline is more accurate overall.")
```

```
Based on both graphs above, I can accurately say that the cubic spline is more accurate overall.
```

(d)

```
[31] print("Based on both graphs above, it seems as though the quadratic spline is more accurate between 1 and 2.")
```

```
Based on both graphs above, it seems as though the quadratic spline is more accurate between 1 and 2.
```

8.18

```
[32] # define data
     x = [0, 1, 2, 3, 4, 5]
     y = [1.0, 2.7, 5.8, 6.7, 7.5, 9.99]
     new_domain = np.linspace(0, 5, num=100, endpoint=True)
```

(a)

```
[48] def poly_val(poly_fit, x_val):
       poly_degree = len(poly_fit)
       y_val = 0
       for i in range(poly_degree):
         y_val = y_val + poly_fit[i]*(x_val**(poly_degree-i))
       return y_val

     def poly_val_dx(poly_fit, x_val):
       poly_degree = len(poly_fit)
       y_dx = 0
       for i in range(poly_degree - 1):
         y_dx = y_dx + poly_fit[i]*(poly_degree-i)*(x_val**(poly_degree-i-1))
       return y_dx

     def print_derivative(der_vals, x_vals, degree):
       # print derivatives
       for i in range(len(x_vals)):
         print("The derivative of the polyfit for degree", degree, "at t =", x_vals[i], "is", der_vals[i], "\n")
```

```
[49] # fit the data to to polynomials of degrees 0-5
     p0 = np.polyfit(x, y, 0)
     p0_y = [poly_val(p0, tmp_x) for tmp_x in new_domain]
     p0_dx = [poly_val_dx(p0, tmp_x) for tmp_x in x]

     p1 = np.polyfit(x, y, 1)
     p1_y = [poly_val(p1, tmp_x) for tmp_x in new_domain]
     p1_dx = [poly_val_dx(p1, tmp_x) for tmp_x in x]

     p2 = np.polyfit(x, y, 2)
     p2_y = [poly_val(p2, tmp_x) for tmp_x in new_domain]
     p2_dx = [poly_val_dx(p2, tmp_x) for tmp_x in x]

     p3 = np.polyfit(x, y, 3)
     p3_y = [poly_val(p3, tmp_x) for tmp_x in new_domain]
     p3_dx = [poly_val_dx(p3, tmp_x) for tmp_x in x]

     p4 = np.polyfit(x, y, 4)
     p4_y = [poly_val(p4, tmp_x) for tmp_x in new_domain]
     p4_dx = [poly_val_dx(p4, tmp_x) for tmp_x in x]

     p5 = np.polyfit(x, y, 5)
     p5_y = [poly_val(p5, tmp_x) for tmp_x in new_domain]
     p5_dx = [poly_val_dx(p5, tmp_x) for tmp_x in x]
```

```
[50] print_derivative(p0_dx, x, 0)
     print_derivative(p1_dx, x, 1)
     print_derivative(p2_dx, x, 2)
     print_derivative(p3_dx, x, 3)
     print_derivative(p4_dx, x, 4)
     print_derivative(p5_dx, x, 5)
```

```
The derivative of the polyfit for degree 0 at t = 5 is 0

The derivative of the polyfit for degree 1 at t = 0 is 0.0

The derivative of the polyfit for degree 1 at t = 1 is 3.442857142857142

The derivative of the polyfit for degree 1 at t = 2 is 6.885714285714284

The derivative of the polyfit for degree 1 at t = 3 is 10.328571428571426

The derivative of the polyfit for degree 1 at t = 4 is 13.771428571428569

The derivative of the polyfit for degree 1 at t = 5 is 17.21428571428571

The derivative of the polyfit for degree 2 at t = 0 is 0.0

The derivative of the polyfit for degree 2 at t = 1 is 4.099107142857144

The derivative of the polyfit for degree 2 at t = 2 is 7.635714285714288

The derivative of the polyfit for degree 2 at t = 3 is 10.609821428571433

The derivative of the polyfit for degree 2 at t = 4 is 13.021428571428576

The derivative of the polyfit for degree 2 at t = 5 is 14.870535714285717

The derivative of the polyfit for degree 3 at t = 0 is 0.0

The derivative of the polyfit for degree 3 at t = 1 is 4.737764550264541

The derivative of the polyfit for degree 3 at t = 2 is 7.40608465608465

The derivative of the polyfit for degree 3 at t = 3 is 9.72718253968253

The derivative of the polyfit for degree 3 at t = 4 is 13.423280423280389

The derivative of the polyfit for degree 3 at t = 5 is 20.216600529100425

The derivative of the polyfit for degree 4 at t = 0 is 0.0

The derivative of the polyfit for degree 4 at t = 1 is 4.737764550264557

The derivative of the polyfit for degree 4 at t = 2 is 9.010251322751314

The derivative of the polyfit for degree 4 at t = 3 is 7.802182539682515

The derivative of the polyfit for degree 4 at t = 4 is 9.573280423280327

The derivative of the polyfit for degree 4 at t = 5 is 36.25826719576689

The derivative of the polyfit for degree 5 at t = 0 is 0.0

The derivative of the polyfit for degree 5 at t = 1 is 5.475499999999941

The derivative of the polyfit for degree 5 at t = 2 is 8.649333333333365

The derivative of the polyfit for degree 5 at t = 3 is 6.676500000000118

The derivative of the polyfit for degree 5 at t = 4 is 13.192000000000956

The derivative of the polyfit for degree 5 at t = 5 is 22.250833333336452
```

(b)

```
[51] # determine cubic spline
     cs = CubicSpline(x, y).derivative()
     y_cs = cs(x)
```

```
[52] # print derivatives
     for i in range(len(y_cs)):
       print("The derivative of the cubic spline at t = ", x[i], " is ", y_cs[i], "\n")
```

```
The derivative of the cubic spline at t =  0  is  -0.7135555555555531

The derivative of the cubic spline at t =  1  is  3.2567777777777764

The derivative of the cubic spline at t =  2  is  2.086444444444445

The derivative of the cubic spline at t =  3  is  0.3974444444444444

The derivative of the cubic spline at t =  4  is  1.4237777777777783

The derivative of the cubic spline at t =  5  is  3.777444444444444
```

(c)

```
[53] # determine cubic spline
     chs = PchipInterpolator(x, y).derivative()
     y_chs = chs(x)
```

```
[54] # print derivatives
     for i in range(len(y_chs)):
       print("The derivative of the cubic spline at t = ", x[i], " is ", y_chs[i], "\n")
```

```
The derivative of the cubic spline at t =  0  is  1.0000000000000004

The derivative of the cubic spline at t =  1  is  2.1958333333333333

The derivative of the cubic spline at t =  2  is  1.3950000000000005

The derivative of the cubic spline at t =  3  is  0.8470588235294118

The derivative of the cubic spline at t =  4  is  1.2109422492401214

The derivative of the cubic spline at t =  5  is  3.335000000000001
```