In [92]:
```python
import math
from sympy import *
```

§2.3

Problem 1

Let $f(x) = x^2 - 6$ and $p_0 = 1$. Use Newton's method to find p_2.

In [93]:
```python
# Variable for sympy diff function
x = Symbol('x')
```

In [94]:
```python
# Function for newton method
# Manually input functions - this function changes return values
def f(z):
    return z**2 - 6
```

In [95]:
```python
# Function for equation to input into symbol diff function
def fdiff():
    return x ** 2 - 6
```

In [96]:
```python
# Newton method algorithm
def newton(p0, N, f, TOL=None):
    i = 1 # Counter for number of iterations

    while (i <= N):
        fprime = diff(fdiff(),x) # Getting f' using sympy
        fprime = lambdify(x,fprime) # Turning it into a function
        p = p0 - f(p0) / fprime(p0) # Compute p

        # Check if p is close enough
        if TOL != None:
            if abs(p - p0) < TOL:
                return p

        print(f"p_{i} =", p) # Print out each iteration

        i += 1 # Next iteration

        p0 = p # Update p0

    return f"Failed after {N} iterations."
```

In [97]:
```python
print(newton(1,2,f))
```

```
p_1 = 3.5
p_2 = 2.607142857142857
Failed after 2 iterations.
```

Solution: Using Newton's method, we get that $p_2 = 2.607142857142857$.

Problem 3

Let $f(x) = x^2 - 6$. With $p_0 = 3$ and $p_1 = 2$, find p_3.

(a) Use the Secant method.

In [98]:

```
def secant(p0, p1, N, f, TOL=None):
    i = 2

    q0 = f(p0)
    q1 = f(p1)

    while (i <= N):
        p = p1 - q1*(p1-p0)/(q1-q0) # Compute p

        if TOL != None:
            if abs(p - p1) < TOL: # Check if p within our tolerance
                return p # Return p if so

        print(f"p_{i} =", p) # Print each iteration

        i += 1 # Next iteration

        # Update to new positions
        p0 = p1
        q0 = q1
        p1 = p
        q1 = f(p)

    return f"Failed after {N} iterations."
```

In [99]:

```
print(secant(3,2,3,f))
```

```
p_2 = 2.4
p_3 = 2.4545454545454546
Failed after 3 iterations.
```

Solution: The Secant method gives $p_3 = 2.4545454545454546$.

(b) Use the method of False Position

In [100…

```
def false_pos(p0, p1, N, f, TOL=None):
    i = 2

    q0 = f(p0)
    q1 = f(p1)

    while (i <= N):
        p = p1 - q1*(p1-p0)/(q1-q0) # Compute p

        if TOL != None:
            if abs(p - p1) < TOL: # Check if p within our tolerance
                return p # Return p if so
```

```python
        print(f"p_{i} =", p) # Print each iteration

        i += 1 # Next iteration

        q = f(p) # Compute f(p)

        if q * q1 < 0: # q and q1 have different signs
            p0 = p1
            q0 = q1

        # Update to new positions
        p1 = p
        q1 = q

    return f"Failed after {N} iterations."
```

In [101…

```python
print(false_pos(3,2,3,f))
```

```
p_2 = 2.4
p_3 = 2.4444444444444444
Failed after 3 iterations.
```

Solution: The False Position method gives $p_3 = 2.4444444444444444$.

(c) Which of (a) or (b) is closer to $\sqrt{6}$?

In [102…

```python
abs(2.4444444444444444 - math.sqrt(6))
```

Out[102… 0.005045298338733684

In [103…

```python
abs(2.4545454545454546 - math.sqrt(6))
```

Out[103… 0.0050557117622767045

Solution: We see that $2.44444\ldots$ gives a lower error than $2.454545\ldots$, so (b) is closer to $\sqrt{6}$.

Problem 5

Use Newton's method to find solutions accurate to within 10^{-4} for the following problems.

(a) $x^3 - 2x^2 - 5 = 0, \quad [1, 4]$

In [104…

```python
# Function for computing in Newton's method
def f(z):
    return z**3 - 2*z**2 - 5

# Function for equation in sympy diff func
def fdiff():
    return x**3 - 2*x**2 - 5
```

In [105…

```
f(1), f(4)
```

Out[105…

```
(-6, 27)
```

We see that f has a solution in $[1, 4]$. A good guess might be $z = 2$.

In [106…

```
print(newton(2, 10, f, 0.0004))
```

```
p_1 = 3.25
p_2 = 2.811036789297659
p_3 = 2.697989502468529
p_4 = 2.6906771528603617
2.690647448517619
```

Solution: We see that $p_4 = 2.6906771528603617$ is accurate within 10^{-4}.

(b) $x^3 + 3x^2 - 1 = 0, \quad [-3, -2]$

In [107…

```
# Function for computing in Newton's method
def f(z):
    return z**3 - 3*z**2 - 1

# Function for equation in sympy diff func
def fdiff():
    return x**3 - 3*x**2 - 1
```

In [108…

```
f(-3), f(-2)
```

Out[108…

```
(-55, -21)
```

By the Intermediate Value Theorem (IVT), there are no solutions for this equation in the interval $[-3, -2]$ because both endpoints have the same sign.

In [109…

```
print(newton(-2.5,100,f,0.0004))
```

```
p_1 = -1.451851851851852
p_2 = -0.7611882918800966
p_3 = -0.25697205800872513
p_4 = 0.441371322735 2529
p_5 = -0.2846881453917618
p_6 = 0.36423007608469904
p_7 = -0.39087694196166356
p_8 = 0.1505932161049699
p_9 = -1.123600353621353
p_10 = -0.5341870324377
p_11 = -0.039627324816151 94
p_12 = 4.10419582947992 2
p_13 = 3.4248891529206533
p_14 = 3.1527702080782065
p_15 = 3.105212888825903
p_16 = 3.103804621117199
3.103803402736447 4
```

Solution: We see that a solution exists at $p_{16} = 3.103804621117199$ with

$p_0 = -2.5$ given our tolerance. However, $p_{16} \notin [-3, -2]$.

(c) $x - \cos x = 0$, $[0, \pi/2]$

In [110…

```
# Function for computing in Newton's method
def f(z):
    return z - math.cos(z)

# Function for equation in sympy diff func
def fdiff():
    return x - cos(x)
```

In [111…

```
f(0), f(math.pi/2)
```

Out[111…

```
(-1.0, 1.5707963267948966)
```

By IVT, this function has a solution in the interval $[0, \pi/2]$. A good guess is $\pi/4$.

In [112…

```
print(newton(math.pi/4,100,f,0.0004))
```

```
p_1 = 0.7395361335152383
p_2 = 0.7390851781060102
0.739085133215161
```

Solution: $p_2 = 0.7390851781060102$.

(d) $x - 0.8 - 0.2\sin x = 0, \quad [0, \pi/2]$

In [113…

```
# Function for computing in Newton's method
def f(z):
    return z - 0.8 - 0.2*math.sin(z)

# Function for equation in sympy diff func
def fdiff():
    return x - 0.8 - 0.2*sin(x)
```

In [114…

```
f(0), f(math.pi/2)
```

Out[114…

```
(-0.8, 0.5707963267948966)
```

By IVT, we have a solution. We start with the guess $\pi/4$.

In [115…

```
print(newton(math.pi/4,100,f,0.0004))
```

```
p_1 = 0.9671208209237235
p_2 = 0.9643346085485506
0.9643338876952708
```

Solution: $p_2 = 0.9643346085485506$.

Problem 29. A.

In [116…

```
# Helper functions for computing main function in Newton's method
def A(l,b1):
    return l*math.sin(b1)

def B(l,b1):
    return l*math.cos(b1)

def C(h,d,b1):
    return (h+0.5*d)*math.sin(b1) - 0.5*d*math.tan(b1)

def E(h,d,b1):
    return (h+0.5*d)*math.cos(b1) - 0.5*d

#l = 89, h = 49, D = 55, b1 = 11.5
# Main function for Newton's method
def f(a):
    return A(89,11.5*math.pi/180)*math.sin(a)*math.cos(a) + B(89,11.5*math.pi/18
```

In [117…

```
f(33*math.pi/180)
```

Out[117…

```
0.025411305811587 68
```

Solution: We see that $f(33^o) \approx 0$, which means that $\alpha \approx 33^o$ gives $f(\alpha) = 0$ and therefore is a root.

Problem 29. B.

In [118…

```
# Helper functions for computing main function in Newton's method
def A(l,b1):
    return l*math.sin(b1)

def B(l,b1):
    return l*math.cos(b1)

def C(h,d,b1):
    return (h+0.5*d)*math.sin(b1) - 0.5*d*math.tan(b1)

def E(h,d,b1):
    return (h+0.5*d)*math.cos(b1) - 0.5*d

#l = 89, h = 49, D = 55, b1 = 11.5
# Main function for Newton's method
def f(a):
    return A(89,11.5*math.pi/180)*math.sin(a)*math.cos(a) + B(89,11.5*math.pi/18

# Function for equation in sympy diff func
def fdiff():
    return A(89,11.5*math.pi/180)*sin(x)*cos(x) + B(89,11.5*math.pi/180)*sin(x)*
```

In [119…

```
print(newton(30*math.pi/180,10,f))
```

```
p_1 = 0.5815592326046858
```

```
p_2 = 0.5789113452693035
p_3 = 0.5789065809579442
p_4 = 0.5789065809424
p_5 = 0.5789065809423998
p_6 = 0.5789065809423999
p_7 = 0.5789065809423998
p_8 = 0.5789065809423999
p_9 = 0.5789065809423998
p_10 = 0.5789065809423999
Failed after 10 iterations.
```

In [120…

```
0.5789065809423999*(180/math.pi)
```

Out[120…

```
33.16890382034809
```

Solution: $\alpha = 33.16890382034809.$

§2.4

Problem 1

Use Newton's method to find solutions accurate to within 10^{-5} to the following problems.

(a) $x^2 - 2xe^{-x} + e^{-2x} = 0$, for $0 \leq x \leq 1$

In [121…

```
# Function for computing in Newton's method
def f(z):
    return z**2 - 2*z*math.e**(-z) + math.e**(-2*z)

# Function for equation in sympy diff func
def fdiff():
    return x**2 - 2*x*math.e**(-x) + math.e**(-2*x)
```

Let's choose $p_0 = 0.5$.

In [122…

```
print(newton(0.5,100,f,0.00001))
```

```
p_1 = 0.5331555015986092
p_2 = 0.5500438056228882
p_3 = 0.5585669565504401
p_4 = 0.5628484514220423
p_5 = 0.5649941998805688
p_6 = 0.5660683270089566
p_7 = 0.5666057041281631
p_8 = 0.5668744711177544
p_9 = 0.567008874225304
p_10 = 0.5670760806826248
p_11 = 0.5671096851375983
p_12 = 0.5671264876717672
0.5671348890156375
```

Solution: $p_{12} = 0.5671264876717672.$

(d) $e^{6x} + 3(\ln 2)^2 e^{2x} - (\ln 8)e^{4x} - (\ln 2)^3 = 0$, for $-1 \le x \le 0$

In [123…

```
from math import e, log

# Function for computing in Newton's method
def f(z):
    return e**(6*z) + 3*log(2)**2*e**(2*z) - log(8)*e**(4*z) - log(2)**3

# Function for equation in sympy diff func
def fdiff():
    return e**(6*x) + 3*log(2)**2*e**(2*x) - log(8)*e**(4*x) - log(2)**3
```

Let's choose $p_0 = -0.5$.

In [124…

```
print(newton(-0.5,100,f,0.00001))
```

```
p_1 = -0.35263843577271403
p_2 = -0.2854364225630798
p_3 = -0.24764648794627175
p_4 = -0.22473983414938858
p_5 = -0.21032222095717718
p_6 = -0.201051649286204 4
p_7 = -0.19501309988653298
p_8 = -0.19104778392223762
p_9 = -0.18843033562173553
p_10 = -0.186696756659131
p_11 = -0.18554603692266078
p_12 = -0.18478109465495024
p_13 = -0.184272107577899 5
p_14 = -0.1839332143942448
p_15 = -0.18370747695799367
p_16 = -0.18355707024831425
p_17 = -0.18345683691912198
p_18 = -0.18339003157432046
p_19 = -0.18334550135427216
p_20 = -0.18331581925532686
p_21 = -0.18329603011259576
p_22 = -0.18328284949636225
-0.18327404212566822
```

Solution: $p_{22} = -0.18328284949636225.$

Problem 5

Use Newton's method and the modified Newton's method described in Eq. (2.13) to find a solution accurate to within 10^{-5} to the problem

$$e^{6x} + 1.441e^{2x} - 2.079e^{4x} - 0.3330 = 0, for - 1 \le x \le 0.$$

This is the same problem as 1(d) with the coefficients replaced by their four-digit approximations. Compare the solutions to the results in 1(d) and 2(d).

In [125…

```
# Modified Newton method
def mod_newton(p0, N, f, TOL=None):
    i = 1 # Counter for number of iterations
```

```python
    while (i <= N):

        fp = diff(fdiff(),x) # Get f' by sympy
        fpp = diff(fp,x)

        fp = lambdify(x,fp) # Turn fp into a function
        fpp = lambdify(x,fpp) # Turn fpp into a function

        p = p0 - ( (f(p0) * fp(p0)) / (fp(p0)**2 - f(p0)*fpp(p0)) ) # Compute p

        # Check if p is close enough
        if TOL != None:
            if abs(p - p0) < TOL:
                return p

        print(f"p_{i} =", p) # Print out each iteration

        i += 1 # Next iteration

        p0 = p # Update p0

    return f"Failed after {N} iterations."
```

In [126…

```python
from math import e, log

# Function for computing in Newton's method
def f(z):
    return e**(6*z) + 1.441*e**(2*z) - 2.079*e**(4*z) - 0.3330

# Function for equation in sympy diff func
def fdiff():
    return e**(6*x) + 1.441*e**(2*x) - 2.079*e**(4*x) - 0.3330
```

Let's choose $p_0 = -0.5$.

In [127…

```python
print(newton(-0.5,100,f,0.00001))
```

```
p_1 = -0.352418390794109
p_2 = -0.28498026416326533
p_3 = -0.24681352821302127
p_4 = -0.22323285347614585
p_5 = -0.20750945835308304
p_6 = -0.19540082353533894
p_7 = -0.1810351782180801
p_8 = -0.1525930918117031
p_9 = -0.16201454173472227
p_10 = -0.16736111400529263
p_11 = -0.1693411335734311
p_12 = -0.16960232053199342
-0.16960654689895904
```

In [128…

```python
print(mod_newton(-0.5,100,f,0.00001))
```

```
p_1 = -0.2648968499447546
p_2 = -0.18557875399632764
p_3 = -0.20275906145866215
```

```
p_4 = -0.44218269478800387
p_5 = -0.2394289355121628
p_6 = -0.17873578568860854
p_7 = -0.17365776818821568
p_8 = -0.17062242171578237
p_9 = -0.16967055911224765
p_10 = -0.16960679874406792
-0.16960654799492844
```

Solution: Newton's method gives $p_{12} = -0.16960232053199342$ while modified Newton's method gives $p_{10} = -0.16960232053199342$.

Problem 6

Show that the following sequences converge linearly to $p = 0$. How large must n be before $|p_n - p| \leq 5 \times 10^{-2}$?

(a) $p_n = \dfrac{1}{n}, \quad n \geq 1$

$$\lim_{n\to\infty} \frac{p_{n+1} - 0}{p_n - 0} = \lim_{n\to\infty} \frac{\frac{1}{n+1}}{\frac{1}{n}} = \lim_{n\to\infty} \frac{n}{n+1} = 1$$

So the convergence is linear.

In [129…

```
# Function for calculating n for second part of question
def error(f, p, TOL):
    n = 1 # Counter for n
    while (abs(f(n) - p) > TOL): # The error inequality given
        n += 1
    return n

# Function to be used in error
# Defined by p_n
def f(n):
    return 1/n
```

In [130…

```
print(error(f,0,0.05))
```

```
20
```

$n = 20$

(b) $p_n = \dfrac{1}{n^2}, \quad n \geq 1$

$$\lim_{n\to\infty} \frac{p_{n+1} - 0}{p_n - 0} = \lim_{n\to\infty} \frac{\frac{1}{(n+1)^2}}{\frac{1}{n^2}} = \lim_{n\to\infty} \frac{n^2}{(n+1)^2} = \lim_{n\to\infty} \frac{n^2}{n^2 + 2n + 1} = 1.$$

So the convergence is linear.

In [131…

```
# Function for calculating n, defined by p_n
```

```python
def f(n):
    return 1 / n**2
```

In [132…

```python
print(error(f,0,0.05))
```

5

$n = 5$

Problem 8. A.

Show that the sequence $p_n = 10^{-2^n} = \dfrac{1}{10^{2^n}}$ converges quadratically to 0.

$$\lim_{n\to\infty} \frac{|p_{n+1} - 0|}{|p_n - 0|^2} = \lim_{n\to\infty} \frac{|\frac{1}{10^{2^{n+1}}}|}{|\frac{1}{10^{2^n}}|^2} = \lim_{n\to\infty} \frac{10^{2^{n+1}}}{10^{2^{n+1}}} = 1.$$

So the convergence is quadratic.

Problem 8. B.

Show that the sequence $p_n = 10^{-n^k}$ does not converge to 0 quadratically, regardless of the size of the exponent $k > 1$.

$$\lim_{n\to\infty} \frac{|p_{n+1}|}{|p_n|^2} = \lim_{n\to\infty} \frac{|10^{-(n+1)^k|}}{|10^{-n^k}|^2} = \lim_{n\to\infty} \frac{10^{2n^k}}{10^{(n+1)^k}} = \lim_{n\to\infty} \frac{10^{2n^k}}{10^{n^k+\cdots}} = \infty$$

So this sequence does not converge to 0 quadratically.

$\S 3.1$

Problem 1

For the given functions $f(x)$, let $x_0 = 0, x_1 = 0.6$, and $x_2 = 0.9$. Construct interpolation polynomials of degree at most one and at most two to approximate $f(0.45)$, and find the absolute error.

(a) $f(x) = \cos x$

In [133…

```python
# Function for getting L_{n,k}
def lagrange(n,x,xlist):
    L1 = [] # List for each (x-x_i) / (x_k - x_i) in L_{n,k}
    L2 = [] # List for L_{n,k}
    for k in range(n+1):
        for i in range(n+1):
            if k != i: # Makes sure we don't divide by zero
                L1.append((x-xlist[i])/(xlist[k]-xlist[i])) # Append (x-x_i) / (
        L2.append(prod(L1)) # Append the product of all (x-x_i) / (x_k - x_i) in
        L1 = [] # Resets for new term
    return L2
```

```python
# Function for polynomial P(x)
def P(f,n,x,xlist):
    poly = [] # List for each term in P(x)
    L = lagrange(n,x,xlist) # Get list of all L_{n,k}
    for i in range(n+1):
        poly.append(f(xlist[i]) * L[i]) # Multiply L_{n,k} with f(x_k)
    return sum(poly) # Sum them up

# Given function
def f(x):
    return math.cos(x)
```

In [134…

```python
print(P(f,1,0.45,[0,0.6,0.9]))
```

```
0.8690017111822588
```

In [135…

```python
f(0.45)
```

Out[135…

```
0.9004471023526769
```

In [136…

```python
f(0.45)-P(f,1,0.45,[0,0.6,0.9])
```

Out[136…

```
0.031445391170418l1
```

$P_1(0.45) = 0.8690017111822588$. And the absolute error is $|0.9004471023526769 - 0.8690017111822588| = 0.031445391170418l1$.

In [137…

```python
print(P(f,2,0.45,[0,0.6,0.9]))
```

```
0.8981000747057218
```

In [138…

```python
f(0.45)-P(f,2,0.45,[0,0.6,0.9])
```

Out[138…

```
0.0023470276469550466
```

$P_2(0.45) = 0.8981000747057218$. And the absolute error is $|0.9004471023526769 - 0.8981000747057218| = 0.0023470276469550466$.

(d) $f(x) = \tan x$

In [139…

```python
def f(x):
    return math.tan(x)
```

In [140…

```python
print(P(f,1,0.45,[0,0.6,0.9]))
```

```
0.5131026062562692
```

In [141…

```python
f(0.45)
```

Out [141... 0.4830550656165784

In [142...
```
abs(f(0.45)-P(f,1,0.45,[0,0.6,0.9]))
```
Out [142... 0.03004754063969084

$P_1(0.45) = 0.5131026062562692$. And the absolute error is $|0.4830550656165784 - 0.5131026062562692| = 0.03004754063969084$.

In [143...
```
print(P(f,2,0.45,[0,0.6,0.9]))
```
0.45461435499681907

In [144...
```
abs(f(0.45)-P(f,2,0.45,[0,0.6,0.9]))
```
Out [144... 0.028440710619759335

$P_2(0.45) = 0.45461435499681907$. And the absolute error is $|0.4830550656165784 - 0.45461435499681907| = 0.028440710619759335$.

Problem 2

For the given functions $f(x)$, let $x_0 = 1, x_1 = 1.25$, and $x_2 = 1.6$. Construct interpolation polynomials of degree at most one and at most two to approximate $f(1.4)$, and find the absolute error.

(a) $f(x) = \sin \pi x$

In [145...
```
def f(x):
    return math.sin(math.pi*x)
```
In [146...
```
print(P(f,1,1.4,[1,1.25,1.6]))
```
-1.1313708498984756

In [147...
```
f(1.4)
```
Out [147... -0.9510565162951535

In [148...
```
abs(f(1.4)-P(f,1,1.4,[1,1.25,1.6]))
```
Out [148... 0.18031433360332205

$P_1(1.4) = -1.1313708498984756$. And the absolute error is $|-0.9510565162951535 - (-1.1313708498984756)| = 0.1803143336033220$

.

In [149…

```
print(P(f,2,1.4,[1,1.25,1.6]))
```

```
-0.9182280617406016
```

In [150…

```
abs(f(1.4)-P(f,2,1.4,[1,1.25,1.6]))
```

Out[150…

```
0.03282845455455197
```

$P_2(1.4) = -0.9182280617406016$. And the absolute error is $|-0.9510565162951535 - (-0.9182280617406016)| = 0.032828454554551 9$

.

(b) $f(x) = \sqrt[3]{x-1}$

In [151…

```
def f(x):
    return (x-1)**(1/3)
```

In [152…

```
print(P(f,1,1.4,[1,1.25,1.6]))
```

```
1.0079368399158983
```

In [153…

```
f(1.4)
```

Out[153…

```
0.7368062997280773
```

In [154…

```
abs(f(1.4)-P(f,1,1.4,[1,1.25,1.6]))
```

Out[154…

```
0.271130540187821
```

$P_1(1.4) = 1.0079368399158983$. And the absolute error is $|0.7368062997280773 - 1.0079368399158983| = 0.271130540187821.$

In [155…

```
print(P(f,2,1.4,[1,1.25,1.6]))
```

```
0.8169446700381561
```

In [156…

```
abs(f(1.4)-P(f,2,1.4,[1,1.25,1.6]))
```

Out[156…

```
0.08013837031007875
```

$P_2(1.4) = 0.8169446700381561$. And the absolute error is $|0.7368062997280773 - 0.8169446700381561| = 0.08013837031007875.$

Problem 3

Use Theorem 3.3 to find an error bound for the approximations in Exercise 1.

(a) $f(x) = \cos(x)$

For the first degree polynomial, we have

$$\left|\frac{f''(\xi)}{2}(0.45-0)(0.45-0.6)\right| = \left|-\cos(\xi)\cdot -0.03375\right| \leq 0.03375.$$

For the second degree polynomial, we have

$$\left|\frac{f'''(\xi)}{6}(0.45-0)(0.45-0.6)(0.45-0.9)\right| = \left|sin(\xi)\cdot 0.0050625\right| \leq 0.783326909627\cdot 0.005$$

We take $\xi = 0.9$.

(d) $f(x) = \tan(x)$

For the first degree polynomial, we have

$$\left|\frac{f''(\xi)}{2}(0.45-0)(0.45-0.6)\right| = \left|2\sec^2(\xi)\tan(\xi)\cdot -0.03375\right| \leq \left|2.00868474112\cdot -0.03375\right|$$

We take $\xi = 0.6$.

For the second degree polynomial, we have

$$\left|\frac{f'''(\xi)}{6}(0.45-0)(0.45-0.6)(0.45-0.9)\right| = \left|\left[2\sec^4(\xi)+4\sec^2(\xi)\tan^2(\xi)\right]\cdot 0.0050625\right| \leq 2$$

In [157…

```
diff(diff(tan(x)))
```

Out[157…

$$\left(2\tan^2(x)+2\right)\tan(x)$$

In [158…

```
diff(diff(diff(tan(x))))
```

Out[158…

$$\left(\tan^2(x)+1\right)\left(2\tan^2(x)+2\right)+2\left(2\tan^2(x)+2\right)\tan^2(x)$$

Problem 4

Use Theorem 3.3 to find an error bound for the approximations in Exercise 2.

(a) $f(x) = \sin(\pi x)$.

The first degree polynomial has the error form

$$\left|\frac{f''(\xi)}{2}(1.4-1)(1.4-1.25)\right| = \left|-\pi^2\sin(\pi\xi)\cdot 0.03\right| \leq 6.97886419964\cdot 0.03 = 0.209365925$$

We take $\xi = 1.25$.

The second degree polynomial has the error form

$$\left|\frac{f'''(\xi)}{6}(1.4-1)(1.4-1.25)(1.45-1.6)\right| = \left|-\pi^3\cos(\pi\xi)\cdot -0.0015\right| \leq \pi^3 \cdot 0.0015 = 0.0465$$

We take $\xi = 1$.

(b) $f(x) = \sqrt[3]{x-1}$.

The first degree polynomial has the error form

$$\left|\frac{f''(\xi)}{2}(1.4-1.6)(1.4-1.25)\right| = \left|-\frac{2}{9(\xi-1)^{\frac{5}{3}}}\cdot -0.015\right| \leq 2.23985964426 \cdot 0.015 = 0.0335$$

We take $\xi = 1.25$.

The second degree polynomial has the error form

$$\left|\frac{f'''(\xi)}{6}(1.4-1.25)(1.45-1.6)\right| = \left|\frac{10}{27(\xi-1)^{\frac{8}{3}}}\cdot -0.005\right| \leq 14.9323976284 \cdot 0.005 = 0.0746$$

We take $\xi = 1.25$.

Problem 11

Use the following values and four-digit rounding arithmetic to construct a third Lagrange polynomial approximation to $f(1.09)$. The function being approximated is $f(x) = \log_{10}(\tan x)$. Use this knowledge to find a bound for the error in the approximation.

$$f(1.00) = 0.1924 \quad f(1.05) = 0.2414 \quad f(1.10) = 0.2933 \quad f(1.15) = 0.3492$$

In [159…

```
# Lagrange function with rounding arthmetic
def lagrange_round(n,x,xlist,sig_fig):
    L1 = [] # List for each (x-x_i) / (x_k - x_i) in L_{n,k}
    L2 = [] # List for L_{n,k}
    for k in range(n+1):
        for i in range(n+1):
            if k != i: # Makes sure we don't divide by zero
                L1.append( round((x-xlist[i])/(xlist[k]-xlist[i]),sig_fig) ) # A

        # Get the product of all (x-x_i) / (x_k - x_i) in L1 with rounding arthm
        product = 1
        for L in L1:
            prod = round(product * L, sig_fig)
        L2.append(product) # Append the product of all (x-x_i) / (x_k - x_i) in
        L1 = [] # Resets for new term

    return L2

# Function for polynomial P(x) with rounding arthmetic
# Added an f(x) list and removed function f, since the f(x) are provided
def P_round(n,x,xlist,flist,sig_fig):
```

```
    poly = [] # List for each term in P(x)
    L = lagrange(n,x,xlist) # Get list of all L_{n,k}
    for i in range(n+1):
        poly.append(round(flist[i] * L[i], sig_fig)) # Multiply L_{n,k} with f(x

    # Sum them up with rounding arthmetic
    s = 0
    for term in poly:
        s = round(s + term, sig_fig)

    return s

# Given function
def f(x):
    return math.log10(math.tan(x))
```

In [160…

```
print(P_round(3,1.09,[1.00,1.05,1.10,1.15],[0.1924,0.2414,0.2933,0.3492],4))
```

```
0.2825
```

$$f^4(x) = \frac{2\sec^2(x)\left(4\tan^6(x) + 8\sec^4(x)\tan^2(x) - 3\sec^6(x) - 6\tan^4(x)\sec^2(x)\right)}{\ln(10)\tan^4(x)}.$$

The third degree polynomial has the error form

$$\left|\frac{f^4(\xi)}{4!}(1.09-1)(1.09-1.05)(1.09-1.10)(1.09-1.15)\right| = \left|f^4(\xi)\cdot 9\times 10^{-8}\right| \leq 81.508\cdot 9\times$$

We choose $\xi = 1.15$.

$P_3(1.09) = 0.2825$ with 4-digit rounding arthmetic.

Problem 18 A.

From $\S1.1$ 24, Maclaurin series : $\text{erf}(x) = \frac{2}{\sqrt{\pi}}\sum_{k=0}^{\infty}\frac{(-1)^k x^{2k+1}}{(2k+1)k!}$.

0.0	0.2	0.4	0.6	0.8	1.0
0	0.22267	0.42844	0.60381	0.74217	0.84271
1	1	2	3	4	6

In [161…

```
# erf function as given by Section 1.1 Exercise 24
def erf(n, x):
    sum_list = []
    for k in range(n+1):
        sum_list.append(((-1)**k * x**(2*k+1)) / ((2*k+1)*math.factorial(k)))

    return (2 / math.sqrt(math.pi)) * sum(sum_list)

# Find the number of terms needed for erf to be accurate within TOL
def erf_n(x, TOL):
    n = 1
```

```
    while abs((erf(n,x) - math.erf(x))) > TOL:
        n += 1
    return n
```

In [162…

```
erf_n(0.2, 0.0001), erf(1,0.2), math.erf(0.2)
```

Out[162…

```
(1, 0.2226668223068478, 0.22270258921047847)
```

In [163…

```
erf_n(0.4, 0.0001), erf(2, 0.4), math.erf(0.4)
```

Out[163…

```
(2, 0.4284350382072733, 0.42839235504666845)
```

In [164…

```
erf_n(0.6, 0.0001), erf(3,0.6), math.erf(0.6)
```

Out[164…

```
(3, 0.6038063957951938, 0.6038560908479259)
```

In [165…

```
erf_n(0.8, 0.0001), erf(4,0.8), math.erf(0.8)
```

Out[165…

```
(4, 0.7421682570974268, 0.7421009647076605)
```

In [166…

```
erf_n(1.0, 0.0001), erf(6,1), math.erf(1)
```

Out[166…

```
(6, 0.8427142223810101, 0.8427007929497148)
```

Problem 18 B.

We have erf(0) = 0, erf(0.2) = 0.22269, erf(0.4) = 0.42838, etc. The rest are shown above. So we have use our P function from before, but with an f(x) list instead of function f, to get $P_1(1/3)$ and $P_2(1/3)$.

In [167…

```
# Function for polynomial P(x) with flist
def P_flist(flist,n,x,xlist):
    poly = [] # List for each term in P(x)
    L = lagrange(n,x,xlist) # Get list of all L_{n,k}
    for i in range(n+1):
        poly.append(flist[i] * L[i]) # Multiply L_{n,k} with f(x_k)
    return sum(poly) # Sum them up
```

In [168…

```
print(P_flist([0.0,math.erf(0.2),math.erf(0.4),math.erf(0.6),math.erf(0.8),math.
```

```
0.37117098201746407
```

In [169…

```
print(P_flist([0.0,math.erf(0.2),math.erf(0.4),math.erf(0.6),math.erf(0.8),math.
```

```
0.3617194134761927
```

In [170…

```
math.erf(1/3)
```

Out[170…

```
0.3626481117660628
```

Solution: The real value of erf(1/3) = 0.3626481117660628. We have $P_1(1/3) = 0.37117098201746407$ and $P_2(1/3) = 0.3617194134761927$. This means that quadratic interpolation is more feasible.

Problem 23 A.

In [171…

```python
# Helper functions
def n_choose_k(n,k):
    return math.factorial(n) / ( math.factorial(k) * math.factorial(n-k) )

def f(x):
    return x

# Function for Bernstein polynomial of degree n
def bernstein(n, f, x):
    s = 0
    for k in range(n+1):
        s += n_choose_k(n,k) * f(k/n) * x**k*(1-x)**(n-k)
    return s
```

(i) $f(x) = x$.

$$\begin{aligned} B_3(x) &= \binom{3}{0} \cdot \frac{0}{3} \cdot x^0(1-x)^3 + \binom{3}{1} \cdot \frac{1}{3} \cdot x^1(1-x)^2 + \binom{3}{2} \cdot \frac{2}{3} \cdot x^2(1-x)^1 + \binom{3}{3} \cdot \frac{3}{3} \cdot x \\ &= 0 + x(1-x)^2 + 2x^2(1-x) + x^3 \\ &= x^3 - 2x^2 + x + 2x^2 - 2x^3 + x^3 \\ &= x \end{aligned}$$

(ii) $f(x) = 1$.

$$\begin{aligned} B_3(x) &= \binom{3}{0} \cdot x^0(1-x)^3 + \binom{3}{1} \cdot x^1(1-x)^2 + \binom{3}{2} \cdot x^2(1-x)^1 + \binom{3}{3} \cdot x^3(1-x)^0 \\ &= (1-x)^3 + 3x(1-x)^2 + 3x^2(1-x) + x^3 \\ &= -x^3 + 3x^2 - 3x + 1 + 3x^3 - 6x^2 + 3x + 3x^2 - 3x^3 + x^3 \\ &= 1 \end{aligned}$$

Problem 23 B.

$$\begin{aligned} \binom{n-1}{k-1} &= \frac{(n-1)!}{(k-1)!(n-1+k-1)!} \\ &= \frac{(n-1)!}{(k-1)!(n-k)!} \\ \frac{k}{n}\binom{n}{k} &= \frac{k}{n} \cdot \frac{n!}{k!(n-k)!} \\ &= \frac{(n-1)!}{(k-1)!(n-k)!} \end{aligned}$$

Therefore $\binom{n-1}{k-1} = \frac{k}{n} \cdot \binom{n}{k}$.

Problem 23 C.

$$
\begin{aligned}
B_n(x) &= \sum_{k=0}^{n} \binom{n}{k} f(\frac{k}{n}) x^k (1-x)^{n-k} \\
&= \sum_{k=0}^{n} \binom{n}{k} \left(\frac{k}{n}\right)^2 x^k (1-x)^{n-k} \\
&= \sum_{k=1}^{n} \binom{n}{k} \left(\frac{k}{n}\right)^2 x^k (1-x)^{n-k} \\
&= \sum_{k=1}^{n} \binom{n}{k} \left(\frac{k}{n}\right) \left(\frac{k}{n}\right) x^k (1-x)^{n-k} \\
&= \sum_{k=1}^{n} \binom{n-1}{k-1} \left(\frac{k}{n}\right) x^k (1-x)^{n-k} \\
&= \frac{n-1}{n} \sum_{k=1}^{n} \binom{n-1}{k-1} \left(\frac{(k-1)+1}{n-1}\right) x^k (1-x)^{n-k} \\
&= \frac{n-1}{n} \left[\sum_{k=1}^{n} \binom{n-1}{k-1} \left(\frac{k-1}{n-1}\right) x^k (1-x)^{n-k} + \sum_{k=1}^{n} \binom{n-1}{k-1} \left(\frac{1}{n-1}\right) x^k (1-x)^{n-}\right. \\
&= \frac{n-1}{n} \left[\sum_{k=2}^{n} \binom{n-1}{k-1} \left(\frac{k-1}{n-1}\right) x^k (1-x)^{n-k} + \sum_{k=1}^{n} \binom{n-1}{k-1} \left(\frac{1}{n-1}\right) x^k (1-x)^{n-}\right. \\
&= \frac{n-1}{n} \left[\sum_{k=2}^{n} \binom{n-2}{k-2} x^k (1-x)^{n-k}\right] + \frac{1}{n} \left[\sum_{k=1}^{n} \binom{n-1}{k-1} x^k (1-x)^{n-k}\right] \\
&= \frac{n-1}{n} x^2 \left[\sum_{k=2}^{n} \binom{n-2}{k-2} x^{k-2} (1-x)^{n-k}\right] + \frac{x}{n} \left[\sum_{k=1}^{n} \binom{n-1}{k-1} x^{k-1} (1-x)^{n-k}\right]
\end{aligned}
$$

Let $i = k-2$ and $j = k-1$.

$$
\begin{aligned}
B_n(x) &= \frac{n-1}{n} x^2 \left[\sum_{i=0}^{n-2} \binom{n-2}{i} x^i (1-x)^{(n-2)-i}\right] + \frac{x}{n} \left[\sum_{j=0}^{n-1} \binom{n-1}{j} x^j (1-x)^{(n-1)-j}\right] \\
&= \left(\frac{n-1}{n}\right) x^2 + \frac{1}{n} x.
\end{aligned}
$$

Problem 23 D.

In [172…

```
def B_n_error(z):
    n = 1
    while abs( (((n-1)/n)*z**2 + (1/n)*z) - z**2 ) > 0.000001:
        n += 1
    return n
```

$$\begin{aligned}
|B_n(x) - x^2| &= \frac{n-1}{n}x^2 + \frac{1}{n}x - x^2 \\
&= \frac{-1}{n}x^2 + \frac{1}{n}x \\
\frac{d}{dx}\frac{-1}{n}x^2 + \frac{1}{n}x &= -\frac{1}{n}2x \cdot \frac{1}{n} \\
\implies \frac{1}{n} &= \frac{2x}{n} \\
\implies x &= \frac{1}{2}.
\end{aligned}$$

So at $x = 1/2$, we have the maximum value of $B_n(x) - x^2$.

In [173…

```
print(B_n_error(0.5))
```

```
250000
```

Solution: $n \geq 250,000$.