APPM 4600 — HOMEWORK # 4

1. In laying water mains, utilities must be concerned with the possibility of freezing. Although soil and weather conditions are complicated, reasonable approximations can be made on the basis of the assumption that soil is uniform in all directions. In that case the temperature in degrees Celsius $T(x,t)$ at a distance x (in meters) below the surface, t seconds after the beginning of a cold snap, approximately satisfies

$$\frac{T(x,t)-T_s}{T_i-T_s} = \operatorname{erf}\left(\frac{x}{2\sqrt{\alpha t}}\right),$$

where T_s is the constant temperature during a cold period, T_i is the initial soil temperature before the cold snap, α is the thermal conductivity (in meters^2 per second), and

$$\operatorname{erf}(t) = \frac{2}{\sqrt{\pi}}\int_0^t \exp(-s^2)ds$$

Assume that $T_i = 20$ [degrees C], $T_s = -15$[degrees C], $\alpha = 0.138\cdot 10^{-6}$ [meters^2 per second]. For parts (b) and (c), run your experiments with a tolerance of $\epsilon = 10^{-13}$.

   (a) We want to determine how deep a water main should be buried so that it will only freeze after 60 *days* exposure at this constant surface temperature.
   Formulate the problem as a root finding problem $f(x) = 0$. What is f and what is f'? Plot the function f on $[0,\bar{x}]$, where $\bar{x}$ is chosen so that $f(\bar{x}) > 0$.

   (b) Compute an approximate depth using the Bisection Method with starting values $a_0 =$ 0[meters] and $b_0 = \bar{x}$ [meters].

   (c) Compute an approximate depth using Newton's Method with starting value $x_0 = 0.01$[meters]. What happens if you start with $x_0 = \bar{x}$?

Soln:

(a) Solving for $T(x,t)$, we find

$$T(x,t) = T_s + (T_i - T_s)\operatorname{erf}\left(\frac{x}{2\sqrt{\alpha t}}\right).$$

In Celsius freezing occurs when $T(x,t) = 0$.
Now for a fixed t, let $f(x) = T(x,t)$. To apply Newton's method, we need

$$f'(x) = (T_i - T_s)\frac{\partial}{\partial x}\left(\operatorname{erf}\left(\frac{x}{2\sqrt{\alpha t}}\right)\right).$$

By the Fundamental Theorem of Calculus,

$$\frac{\partial}{\partial z}\Big(\operatorname{erf}(z)\Big) = \frac{2}{\sqrt{\pi}}e^{-z^2}.$$

To get the derivative we need, we must use chain rule, i.e.

$$\frac{\partial}{\partial x}\Big(\mathrm{erf}(z(x))\Big) = \frac{\partial}{\partial z}\Big(\mathrm{erf}(z)\Big)\frac{\partial z}{\partial x}.$$

Thus

$$f'(x) = (T_i - T_s)\frac{1}{\sqrt{\pi\alpha t}}e^{-\left(\frac{x}{2\sqrt{\alpha t}}\right)^2}.$$

We are now in a position to code. See figure 0.1 for the plot.

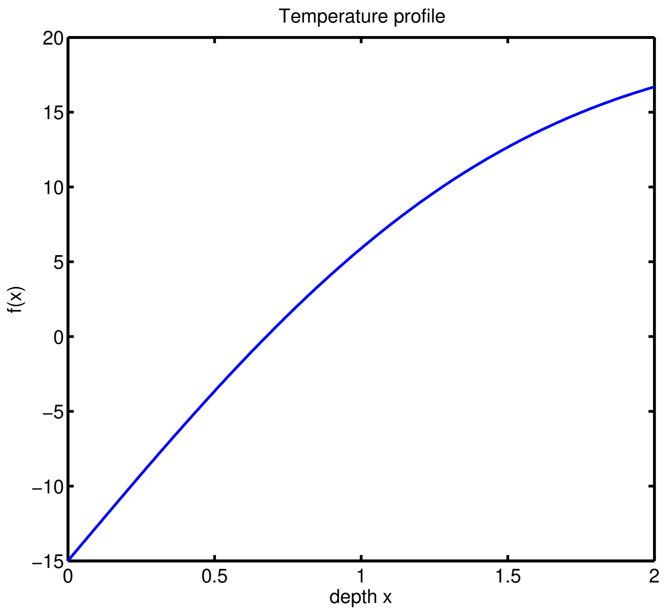


Figure 0.1: Plot of $f(x) = T(x,t)$ for t set at 60 days.

(b)
```
Bisection: the approximate root is: 6.7696185448193091e-01
Bisection: Number of iterations: 44
```

(c)
```
  Newton: the initial guess is: 2.0000000000000000e+00
Newton: the approximate root is: 6.7696185448193658e-01
Newton: Number of iterations: 6
```

Newton's converges much fasters hence I prefer it.
The code for this problem reads

```
import mypkg.prini as prini
import mypkg.my2DPlotB
import scipy
import numpy as np
import math

def driver():
```

```
tol = 1e-13

Ti = 20.0
Ts= -15.0
alpha = 0.138*1e-6

''' we need to turn 60 days into seconds
 there are 60sec in a min
 60 min in a hour and 24 hours in a  day.
So there are 24*60*60 seconds in a day.'''
t = 60*24*60*60.0

f = lambda x: Ts+(Ti-Ts)*scipy.special.erf(x/(2*math.sqrt(alpha*t)))
fp = lambda x: (Ti-Ts)*math.exp(-(x**2)/(4*alpha*t))/math.sqrt(math.pi*alpha*t)

xtest = np.linspace(0,2,200)
ftest = f(xtest)

plt = mypkg.my2DPlotB(xtest,ftest)
plt.labels('depth x', 'f(x)')
plt.savefig('hw4prob1.pdf')
plt.show()

Nmax = 100
[c,ier,its] = bisection(f,0,2,tol,Nmax)
tmp = prini('real','Bisection: the approximate root is:',c)
tmp.print()
tmp = prini('inter','Bisection: Number of iterations:',its)
tmp.print()


p0 = 0.01
[p,c,info,it] = newton(f,fp,p0,tol, Nmax)

tmp = prini('real','Newton: the initial guess is:',p0)
tmp.print()
tmp = prini('real','Newton: the approximate root is:',c)
tmp.print()
tmp = prini('inter','Newton: Number of iterations:',it)
tmp.print()


p0 = 2;
[p,c,info,it] = newton(f,fp,p0,tol, Nmax)
```

```
    tmp = prini('real','Newton: the initial guess is:',p0)
    tmp.print()
    tmp = prini('real','Newton: the approximate root is:',c)
    tmp.print()
    tmp = prini('inter','Newton: Number of iterations:',it)
    tmp.print()




def bisection(f,a,b,tol,Nmax):
    '''
    Inputs:
      f,a,b       - function and endpoints of initial interval
      tol, Nmax   - bisection stops when interval length < tol
                  - or if Nmax iterations have occured
    Returns:
      astar - approximation of root
      its - number of iterations to converge
      ier   - error message
            - ier = 1 => cannot tell if there is a root in the interval
            - ier = 0 == success
            - ier = 2 => ran out of iterations
            - ier = 3 => other error ==== You can explain
    '''

    '''     first verify there is a root we can find in the interval '''
    fa = f(a); fb = f(b);
    if (fa*fb>0):
       ier = 1
       astar = a
       its = 0
       return [astar, ier,its]

    ''' verify end point is not a root '''
    if (fa == 0):
      astar = a
      ier = 0
      its = 0
      return [astar, ier,its]

    if (fb ==0):
      astar = b
      ier = 0
      its = 0
      return [astar, ier,its]

    count = 0
    while (count < Nmax):
      c = 0.5*(a+b)
      fc = f(c)

      if (fc ==0):
        astar = c
        ier = 0
        its = count
        return [astar, ier,its]

      if (fa*fc<0):
         b = c
      elif (fb*fc<0):
        a = c
        fa = fc
      else:
        astar = c
        ier = 3
        its = count
        return [astar, ier,its]

      if (abs(b-a)<tol):
        astar = a
        ier =0
        its = count
        print(count)
        return [astar, ier,its]

      count = count +1

    astar = a
    ier = 2
    its = count
    return [astar,ier,its]


def newton(f,fp,p0,tol,Nmax):
  """
  Newton iteration.

  Inputs:
    f,fp - function and derivative
    p0   - initial guess for root
    tol  - iteration stops when p_n,p_{n+1} are within tol
    Nmax - max number of iterations
  Returns:
    p     - an array of the iterates
    pstar - the last iterate
    info  - success message
          - 0 if we met tol
          - 1 if we hit Nmax iterations (fail)
  """
  p = np.zeros(Nmax+1)
  p[0] = p0
  for it in range(Nmax):
      p1 = p0-f(p0)/fp(p0)
      p[it+1] = p1
      if (abs(p1-p0) < tol):
          pstar = p1
          info = 0
          return [p,pstar,info,it]
      p0 = p1
  pstar = p1
  info = 1
  return [p,pstar,info,it]


if __name__ == '__main__':
  # run the drivers only if this is called from the command line
  driver()
```

2. Let $f(x)$ denote a function with root α of multiplicity m.

   (a) Write down a formal mathematical definition of what it means for α to be a root of multiplicity m of $f(x)$.

   (b) Show that Newton's method applied to $f(x)$ only converges linearly to the root α.

   (c) Show that the fixed point iteration applied to $g(x) = x - m\frac{f(x)}{f'(x)}$ is second order convergent.

   (d) What does part (c) provide for Newton's method in the case of roots with multiplicity greater than 1?

Soln:

(a) $f(x) = (x-\alpha)^m q(x)$ where $q(\alpha) \neq 0$.

(b) Newton's method is a fixed point method with $g(x) = x - \frac{f(x)}{f'(x)}$. Fixed point method's converges linearly to the fixed point p when $0 < |g'(p)| < 1$.

$f(x) = (x-\alpha)^m q(x)$ and $f'(x) = m(x-\alpha)^{m-1}q(x) + (x-\alpha)^m q'(x)$
Thus

$$g(x) = x - \frac{(x-\alpha)^m q(x)}{m(x-\alpha)^{m-1}q(x) + (x-\alpha)^m q'(x)} = x - \frac{(x-\alpha)q(x)}{mq(x) + (x-\alpha)q'(x)}$$

Thus

$$g'(x) = 1 - \left[\frac{-(x-\alpha)q(x)(mq'(x) + q'(x) + (x-\alpha)q''(x)}{(mq(x) + (x-\alpha)q'(x))^2} + \frac{(x-\alpha)q'(x) + q(x)}{mq(x) + (x-\alpha)q'(x)}\right]$$

Plugging in $x = \alpha$, we get $g'(\alpha) = 1 - \frac{q(\alpha)}{mq(\alpha)} = 1 - \frac{1}{m} < 1$.

(c) here are several ways to do this problem. This is one option.

We want to show that $\frac{|e_{n+1}|}{|e_n|^2} \leq M$ where M is a constant independent of α and n.
We know the following

$$
\begin{aligned}
e_{n+1} = p_{n+1} - \alpha = p_n - \alpha - m\frac{f(p_n)}{f'(p_n)} \\
&= e_n - m\frac{(p_n-\alpha)q(p_n)}{mq(p_n)+(p_n-\alpha)q'(p_n)} \\
&= e_n\left(1 - m\frac{q(p_n)}{mq(p_n)+(p_n-\alpha)q'(p_n)}\right) \\
&= e_n\left(\frac{mq(p_n)+(p_n-\alpha)q'(p_n)-mq(p_n)}{mq(p_n)+(p_n-\alpha)q'(p_n)}\right) \\
&= e_n\left(\frac{(p_n-\alpha)q'(p_n)}{mq(p_n)+(p_n-\alpha)q'(p_n)}\right) \\
&= e_n^2\left(\frac{q'(p_n)}{mq(p_n)+(p_n-\alpha)q'(p_n)}\right)
\end{aligned}
$$

Now we are in a position to look at the ratio.

$$\frac{|e_{n+1}|}{|e_n|^2} = \left|\frac{q'(p_n)}{mq(p_n)+(p_n-\alpha)q'(p_n)}\right| \le M$$

where $M = \max_{x\in(\alpha-\epsilon,\alpha+\epsilon)}\left|\frac{q'(x)}{mq(x)+(x-\alpha)q'(x)}\right|$. Since $q(\alpha)\neq 0$, this a number less than infinity.

(d) Part (c) provides a way to restore second order convergence of Newton's method without requiring any additional derivative information about $f(x)$.

3. Beginning with the definition of order of convergence of a sequence $\{x_k\}_{k=1}^{\infty}$ that converges to α, derive a relationship between the $\log(|x_{k+1}-\alpha|)$ and $\log(|x_k-\alpha|)$. What is the order p in this relationship?

4. You now have two ways of improving the convergence of Newton's method when a root has multiplicity greater than 1: one from class, the other is in problem 2.

   In this problem consider finding the root of the function $f(x) = e^{3x} - 27x^6 + 27x^4e^x - 9x^2e^{2x}$ in the interval $(3,5)$.

   Explore the order of convergence when applying (i) Newton's method, (ii) the modified Newton's method from class and (iii) the modified Newton's method in Problem 2. Which method do you perfer and why?

   Soln: I ran my codes with $p_0 = 3.5$. Below is the output from the codes.

```
Newton first
the approximate root is 3.7330582544115503e+00
the error message reads: 0
number of iterations: 42
[ 0.57093306  0.61887367  0.63880033  0.64938281  0.65554918  0.65928997
```

```
  0.66157228  0.66290981  0.66357952  0.66371491  0.66334601  0.66241035
  0.66074295  0.65804085  0.65379779  0.6470884   0.63680215  0.61854539
  0.59358804  0.53177374  0.41255546  0.36626294  1.43098516  5.03054223
  0.60640669 59.17022618  0.66012054  0.65704875  0.65223796  0.64468829
  0.63292131  0.61327724  0.57889616  0.51715975  0.34923665  0.32920539
  3.98904     1.48931569  0.40025777  0.17312383  6.40002783]
Class version
the approximate root is 3.7330790286328139e+00
the error message reads: 0
number of iterations: 5
[2.29940095e-01 5.02095042e-02 2.48415965e-03 6.16628741e-06]
[0.98653275 0.93684561 0.92315721 0.92244487]
Homework version
the approximate root is 3.7330725623116194e+00
the error message reads: 0
number of iterations: 12
[2.33072562e-01 6.69202117e-02 3.86225831e-03 2.01240850e-05
 1.21366634e-06 2.03675081e-04 6.59888869e-06 2.26486748e-01
 6.25934354e-02 3.39424916e-03 1.70139974e-05 1.87626180e-05]
[1.23189872e+00 8.62435658e-01 1.34906698e+00 2.99686389e+03
 1.38273605e+08 1.59072464e+02 5.20117169e+09 1.22023421e+00
 8.66335563e-01 1.47679062e+00 6.48157740e+04 0.00000000e+00]
```

The version of Newton's method from the homework is running into some stability problems in evaluating the iterates. This is why we go from quadratic convergence to stall then nonsense.

The version from class is the best in my opinion. Here is the code:

```
import numpy as np
import math
import time


def driver():

    f = lambda x: math.exp(3*x)-27*x**6+27*x**4*math.exp(x)-9*x**2*math.exp(2*x)
    fp = lambda x:   3*(math.exp(x) - 6*x)*(math.exp(x) - 3*x**2)**2

    mu = lambda x: (math.exp(x) - 3*x**2)/(3*math.exp(x) - 18*x)
    mup = lambda x: (6*x**2 + math.exp(x)*(2 - 4*x + x**2))/(math.exp(x) - 6*x)**2

    p0 = 3.5
    Nmax = 100
    tol = 1.e-14

    print('Newton first')
    [p,pstar,info,it] = newton(f,fp,p0,tol, Nmax)
    print('the approximate root is', '%16.16e' % pstar)
    print('the error message reads:', '%d' % info)
    print('number of iterations:', '%d' % it)
    err = abs(p-pstar)
    rat = err[1:it]/err[0:it-1]
    print(rat)

    print('Class version')
    [p,pstar,info,it] = newton(mu,mup,p0,tol, Nmax)
    print('the approximate root is', '%16.16e' % pstar)
    print('the error message reads:', '%d' % info)
    print('number of iterations:', '%d' % it)
    err = abs(p-pstar)
    rat = err[1:it]/err[0:it-1]
    print(rat)
    rat = err[1:it]/(err[0:it-1]**2)
    print(rat)

    print('Homework version')
    m = 3.
    [p,pstar,info,it] = scaled_newton(m,f,fp,p0,tol, Nmax)
    print('the approximate root is', '%16.16e' % pstar)
    print('the error message reads:', '%d' % info)
    print('number of iterations:', '%d' % it)
    err = abs(p-pstar)
    print(err[0:it])
    rat = err[1:it+1]/(err[0:it]**2)
    print(rat)



def scaled_newton(m,f,fp,p0,tol,Nmax):
  """
  scaled Newton iteration. for roots of multiplicity m

  Inputs:
    f,fp - function and derivative
    p0   - initial guess for root
    tol  - iteration stops when p_n,p_{n+1} are within tol
    Nmax - max number of iterations
    m - multiplicity of the root
  Returns:
    p     - an array of the iterates
    pstar - the last iterate
    info  - success message
          - 0 if we met tol
          - 1 if we hit Nmax iterations (fail)

  """
  p = np.zeros(Nmax+1);
  p[0] = p0
  for it in range(Nmax):
      p1 = p0-m*f(p0)/fp(p0)
      p[it+1] = p1
      if (abs(p1-p0) < tol):
          pstar = p1
          info = 0
          return [p,pstar,info,it]
      p0 = p1
  pstar = p1
  info = 1
  return [p,pstar,info,it]




def newton(f,fp,p0,tol,Nmax):
  """
  Newton iteration.

  Inputs:
    f,fp - function and derivative
    p0   - initial guess for root
    tol  - iteration stops when p_n,p_{n+1} are within tol
    Nmax - max number of iterations
  Returns:
    p     - an array of the iterates
    pstar - the last iterate
    info  - success message
          - 0 if we met tol
          - 1 if we hit Nmax iterations (fail)

  """
  p = np.zeros(Nmax+1);
  p[0] = p0
  for it in range(Nmax):
      p1 = p0-f(p0)/fp(p0)
      p[it+1] = p1
      if (abs(p1-p0) < tol):
          pstar = p1
          info = 0
          return [p,pstar,info,it]
      p0 = p1
  pstar = p1
  info = 1
  return [p,pstar,info,it]

if __name__ == '__main__':
  # run the drivers only if this is called from the command line
  driver()
```

5. Use Newton and Secant method to approximate the largest root of

$$f(x) = x^6 - x - 1.$$

Start Newton's method with $x_0 = 2$. Start Secant method with $x_0 = 2$ and $x_1 = 1$.

(a) Create at table of the error for each step in the iteration. Does the error decrease as you expect?

(b) Plot $|x_{k+1} - \alpha|$ vs $|x_k - \alpha|$ on log-log axes where α is the exact root for both methods. What are the slopes of the lines that result from this plot? How does this relate to the order?

Soln:

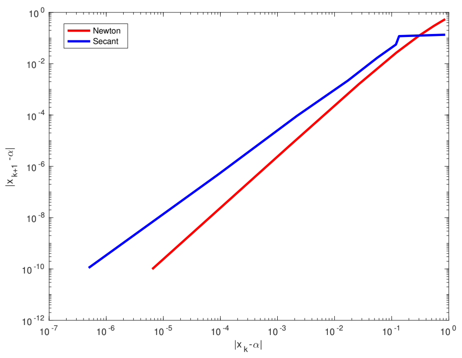
The code for this problem is below

(a)

Iteration	Newton	Secant
0	0.865275861598481	0.865275861598481
1	0.545904133849789	0.134724138401519
2	0.296014849837543	0.118595106143455
3	0.120246817707917	0.0558536302751180
4	0.0268142943717937	0.0170683074599678
5	0.00162913576898593	0.00219258818538637
6	6.38994210966359e-06	9.26696033334284e-05
7	9.87017134690404e-11	4.92452814526700e-07
8	-	1.10303544076373e-10

Newton's method is decreasing quadratically while Secant is decreasing faster than linearly but not quite quadratically.

(b) Below is the requested figure. As expected there are two lines. The Newton line has a steeper slope than the secant line. This is because $2 > 1.6$.



```
import mypkg.prini as prini
import mypkg.my2DPlotB
import scipy
import numpy as np
import math

def driver():

    f = lambda x: x**6-x-1
    fp = lambda x: 6*x**5-1

    p0 = 2
    Nmax  =100
    tol = 1e-14

    [p,pstar,info,it1] = newton(f,fp,p0,tol,Nmax)
    tmp = prini('real','Newton: the approximate root is:',pstar)
    tmp.print()
    tmp = prini('inter','Newton: Error message reads:',info)
    tmp.print()
    tmp = prini('inter','Newton: Number of iterations:',it1)
    tmp.print()

    err1 = abs(p[1:it1]-pstar);

    p0 = 2
    p1 = 1
    [p,pstar,info,it2] = secant(f,p0,p1,tol,Nmax)
    tmp = prini('real','Secant: the approximate root is:',pstar)
    tmp.print()
    tmp = prini('inter','Secant: Error message reads:',info)
    tmp.print()
    tmp = prini('inter','Secant: Number of iterations:',it2)
    tmp.print()

    err2 = abs(p[1:it2]-pstar)

    plt = mypkg.my2DPlotB(err1[0:it1-2],err1[1:it1])
    plt.labels('|x_k-alpha|', '|x_{k+1}-alpha|')
    plt.addPlot(err2[0:(it2-2)],err2[1:it2])
    plt.logy()
    plt.logx()
    plt.savefig('hw4prob5.pdf')
    plt.show()



def newton(f,fp,p0,tol,Nmax):
  """
  Newton iteration.

  Inputs:
    f,fp - function and derivative
    p0   - initial guess for root
    tol  - iteration stops when p_n,p_{n+1} are within tol
    Nmax - max number of iterations
  Returns:
    p     - an array of the iterates
    pstar - the last iterate
    info  - success message
          - 0 if we met tol
          - 1 if we hit Nmax iterations (fail)
  """
  p = np.zeros(Nmax+1)
  p[0] = p0
  for it in range(Nmax):
      p1 = p0-f(p0)/fp(p0)
      p[it+1] = p1
      if (abs(p1-p0) < tol):
          pstar = p1
          info = 0
          return [p,pstar,info,it]
      p0 = p1
  pstar = p1
  info = 1
  return [p,pstar,info,it]


def secant(f,p0,p1,tol,Nmax):
  """
  Secant iteration.

  Inputs:
    f,fp - function and derivative
    p0   - initial guess for root
    p1  - a second guess for the root
    tol  - iteration stops when p_n,p_{n+1} are within tol
    Nmax - max number of iterations
  Returns:
    p     - an array of the iterates
    pstar - the last iterate
    info  - success message
          - 0 if we met tol
          - 1 if we hit Nmax iterations (fail)
  """
  p = np.zeros(Nmax+1)
  p[0] = p0
  p[1] = p1
  fp0 = f(p0)
  fp1 = f(p1)
  for it in range(1,Nmax):
      p2 = p1-fp1*(p1-p0)/(fp1-fp0)
      p[it+1] = p2
      if (abs(p1-p2) < tol):
          pstar = p2
          info = 0
          return [p,pstar,info,it]
      p0 = p1
      fp0 = fp1
      p1 = p2
      fp1 = f(p2)
  pstar = p2
  info = 1
  return [p,pstar,info,it]



if __name__ == '__main__':
  # run the drivers only if this is called from the command line
  driver()
```