Assignment 2

Exercise 1

See code

Exercise 2

See code

Exercise 3

- V1: x = 2.506184145588769, iterations = 27
- V2: x = 3.141592653589793, iterations = 6
- V3: x = error (division by zero), stopped at iteration = 4. Derivative is very close to zero as the algorithm advances.
- V4: x = error (max iterations exceeded/no convergence). The first derivative is not well behaved so the algorithm diverges or overshoots. It can be seen from the log that the value of x jumps back and forth from zero.
- V5: x = error (division by zero), stopped at iteration = 1. This is due to there is a stationary point at x =0, the starting point is a poor choice. If the function is run with a different starting point, it gives the correct solution

```
Out[37]:
                                fsolve         optimize.newton  \
x^x-10              2.5061841455887692      1.0002000162925946
sin(x)               3.141592653589793       3.141592653589793
log2(x)-128                         NA  3.4028236692093695e+38
8*x^6 + 1                           NA    -0.03834488213760778
2*x^3 -2*x^2 + 3                    NA    9.99866669191606e-05

                     optimize.fsolve
x^x-10                           1.0
sin(x)                 3.14159265359
log2(x)-128         3.40282366921e+38
8*x^6 + 1           0.00175915720807
2*x^3 -2*x^2 + 3                 0.0
```

- For V1: optimize functions give value of 1.0, which is not correct.
- For V2: same results
- For V3: correct solutions
- For V4: incorrect solution since should be complex number (no real solutions). RuntimeWarning (for fsolve): The iteration is not making good progress, as measured by the improvement from the last ten iterations.
- For V5: incorrect solution since should be -0.89070. This is because there is a stationary point at x =0, the starting point is a poor choice. If the function is run with a different starting point, it gives the correct solution

Differences might be explained by more advanced implementations by the functions in optimize package.

To test fprime parameter, the different functions were tested with a poor initial guess:

```
# the same
fsolve(lambda x: x**4, -100)
fsolve(lambda x: x**4, -100,lambda x: 4*x**3)

# with fprime converges in 100 iterations
optimize.newton(lambda x: x**4, -100,maxiter=100)
optimize.newton(lambda x: x**4, -100, fprime = lambda x: 4*x**3,maxiter=100)

# the same
optimize.fsolve(lambda x: x**4, -100)
optimize.fsolve(lambda x: x**4, -100, fprime = lambda x: 4*x**3)
```

- For the fsolve functions, no difference was observed when using fprime parameter option
- For the newton function, without fprime the function failed to converge in 100 iterations, while with fprime the function converged in 100 iterations.
- With a good initial guess, no difference was observed when using frpime parameteroption
- For the fsolve runtime error occurs too, reaching maximum iteration with and without fprime.
- So the advantage of fprime is that it provides the exact approximation of the derivative so faster converge might be possible in cases where approximations are not accurate

Exercise 4

See code for function. Test function:

Out[142]:

	math.sqrt	sqrt
0	0.000000	1.490116e-08
9	3.000000	3.000000e+00
10	3.162278	3.162278e+00
50	7.071068	7.071068e+00
100	10.000000	1.000000e+01
150	12.247449	1.224745e+01
400	20.000000	2.000000e+01
900	30.000000	3.000000e+01
1500	38.729833	3.872983e+01

- test with initial guess funcion non fprime vs fprime:

Out[145]:

	w/ fprime (val)	w/o fprime (val)	w/ fprime (iter)	w/o fprime (iter)
0	1.396984e-08	1.396984e-08	32	32
9	3.000000e+00	3.000000e+00	9	9
10	3.162278e+00	3.162278e+00	9	9
50	7.071068e+00	7.071068e+00	8	8
100	1.000000e+01	1.000000e+01	7	7
150	1.224745e+01	1.224745e+01	7	7
400	2.000000e+01	2.000000e+01	6	6
900	3.000000e+01	3.000000e+01	6	6
1500	3.872983e+01	3.872983e+01	5	5

- test with poor initial guess non fprime vs fprime

Out[146]:

	w/ fprime (val)	w/o fprime (val)	w/ fprime (iter)	w/o fprime (iter)
0	1.164157e-08	1.164157e-08	33	33
9	3.000000e+00	3.000000e+00	10	10
10	3.162278e+00	3.162278e+00	10	10
50	7.071068e+00	7.071068e+00	9	9
100	1.000000e+01	1.000000e+01	8	8
150	1.224745e+01	1.224745e+01	8	8
400	2.000000e+01	2.000000e+01	7	7
900	3.000000e+01	3.000000e+01	7	7
1500	3.872983e+01	3.872983e+01	6	6

- results show that number of iterations **do not change** (regardless of good or poor initial estimate) if fprime parameterof fsolve function is used explicitly instead of using the default

Code

```
# -*- coding: utf-8 -*-
"""
Assignment2
Date: 4/4/15
"""

import math
import numpy as np
import matplotlib.pyplot as plt
from scipy import optimize
from scipy import misc
import pandas
#%% Exercise 1 #####################################################################

def createPolynomial(coefficients):
    """
    Creates polynomial function given coefficients

    @ param coefficients:  A tuple of numbers, representing the coefficients
    of a polynomial, such that the i-th entry in the tuple is the coefficient
    of xi.
    @  return a callable (for today, another function), which will have one
    parameter x (anumber) and will return another number resulting in
    evaluating the polynomial p at x
    """

    def polyFunction(x):

        f = 0 # variable to hold polynomial expression

        # iterate through all coefficients
        for n in range(0,len(coefficients)):

            # add up the coefficient times x to the corresponding power
            f += coefficients[n]*math.pow(x,n)

        return f

    return polyFunction


# numpy version
def createPolynomial2(coefficients):
    """
    Creates polynomial function given coefficients

    @ param coefficients:  A tuple of numbers, representing the coefficients
    of a polynomial, such that the i-th entry in the tuple is the coefficient
    of xi.
    @  return a callable (for today, another function), which will have one
    parameter x (anumber) and will return another number resulting in
    evaluating the polynomial p at x
    """

    def polyFunction(x):

```

```
        exponents = range(0,len(coefficients))
        x = [x]*len(coefficients)
        f = np.multiply(np.power(x,exponents),coefficients)

        return sum(f)

    return polyFunction

#%% Exercise 1 test

# example
p = createPolynomial( (2, -4, 3) )
y = p(5)
print(y) # answer should be 57

# example
p = createPolynomial2( (2, -4, 3) )
y = p(5)
print(y) # answer should be 57


#%% Exercise 2 #####################################################################


def findDerivative(f,d):
    """
    Estimates the derivative of a function
    ((f(x+d)-f(x-d)) / (2*d))

    @ param f: function
    @ param d: parameter to estimate derivative
    @  return a callable (for today, another function), which will have one
          parameter x (anumber) and will return th estimated derivative at x

          """
          def derivFunction(x):
              return ((f(x+d)-f(x-d)) / (2*d))

          return derivFunction


      def fsolve(f, x0, fprime=None, xtol=1.49012e-08, maxiter=100):
          """
          Write a function fsolve(f, x0, fprime=None, xtol=1.49012e-08, maxiter=100)
          that returns one real solution of the equation defined by f(x) = 0 given
          a starting estimate x0 using Newton's method.

          @f : callable(x)
          A function that takes exactly one number argument.

          @x0 : float
          The starting estimate for the real root of f(x) = 0.

          @fprime : callable(x), optional
          A function to compute the derivative of func. If the user does not supply
          an argument, estimate the derivative as (f(x+d)-f(x-d)) / (2*d) using a
          small value of d (relative to xtol).

          @xtol : float, (optional)
          The calculation will terminate if the relative error between two consecutive
```

```
            iterates is at most xtol.

            @maxiter : int, (optional)
            The maximum number of iterations. The function should raise RuntimeError if
    this
            occurs.

            @returns one real solution of the equation defined by f(x) = 0

            Raises RuntimeError exception if exceeds maxiterations

            """

            # estimate second derivative if not supplier by user
            if (fprime == None):
                d = xtol/10      # small value of d (relative to xtol)
                fprime = findDerivative(f,d)

            # initialize variables
            error = 1   # difference between x and updated x (large value to start)
            i = 0       # iteration counter
            x = x0      # initial value for estimated root

            # iterate until error in x within tolerance
            while (error > xtol) :

                i += 1

                xnew = x - f(x)/fprime(x)  #compute new x based on previous x
                error = abs(xnew - x)      #compute error
                x = xnew                   #update x

                # raise exception if exceed maxiterations
                if (i > maxiter):
                    raise RuntimeError('Maximum Iterations ' + str(maxiter) +
                    ' Exceeded')

                print('Iter: {:<2}, x = {:<20}, error: {:<40}'.format(i,xnew,error))

            return x


        #%% Exercise 2 test
        f = createPolynomial( (-4 ,0,1) )  #x^2 - 4
        df = createPolynomial( (0 ,2) )    #2*x
        x0 = 7
        maxiter = 5
        xtol = 1e-12
        fsolve(f,x0)                          # should give 2
        fsolve(f,x0, df, xtol)                # should give 2
        fsolve(f,x0, df, xtol, maxiter)       # should give run time error

        y = lambda x: math.cos(x)-x
        x0 = 0.5
        fsolve(f,x0)                          # should give 0.7391


        #%% Exercise 3 #######################################################################
        '''''
        Test the functions you wrote in the previous two exercises. In particular.
        What does your fsolve return with the following five inputs?
```

```
    How many iterations did it take?:
    '''

    def aSillyFunction(x):
        return x**x-10

    # x = 2.506184145588769, iterations = 27
    v1 = fsolve(aSillyFunction, 1)

    x = np.linspace(-5,5,100)
    y = aSillyFunction(x)
    plt.plot(x, y)

    # x = 3.141592653589793, iterations = 6
    v2 = fsolve(math.sin, 2)

    x = list(np.linspace(0,5,100))
    y = np.sin(x)  # use numpy functions instead of math functions
    plt.plot(x, y)

    # x = error (division by zero), stopped iteration = 4
    # first derivative ~ zero for almost all big values
    # solution should be a very big number (x=2^(128) = 3.4*10^(38))
    v3 = fsolve(lambda x: math.log2(x)-128, 1)

    x = np.linspace(1,100000,100)
    y = np.log2(x)-128 # use numpy functions instead of math functions
    y2 = np.gradient(y)
    plt.plot(x, y)
    plt.plot(x, y2)


    # 8*x^6 + 1
    # x = error (division by zero), stopped iteration 96
    # first derivative ~ zero when x ~ 0
    # solution should be complex number (no real solutions)
    v4 = fsolve(createPolynomial( (1,0,0,0,0,0,8) ) , 1)

    x = np.linspace(-8,8,100)
    y = np.polyval((1,0,0,0,0,0,8),x)
    y2 = np.gradient(y)
    plt.plot(x, y)
    plt.plot(x, y2)

    #  2*x^3 -2*x^2 + 3
    # x = error (division by zero), stopped iteration 1
    # first derivative ~ zero when x ~ 0 (starting point)
    # solution should be -0.89070
    v5 = fsolve(createPolynomial( (3,0,-2,2) ), 0)

    x = np.linspace(-2,2,100)
    y = np.polyval((3,0,-2,2),x)
    y2 = np.gradient(y)
    plt.plot(x, y)
    plt.plot(x, y2)


    # create list of functions and initial values
    funcs = [aSillyFunction,
             math.sin,
             lambda x: math.log2(x)-128,
```

```
                createPolynomial( (1,0,0,0,0,0,8) ),
                createPolynomial((3,0,-2,2))]

    x0s = [1,2,1,1,0]

    xs1 = [v1, v2, 'NA','NA','NA']
    xs2 =[]
    xs3 =[]
    # How does it compare to what newton in the scipy optimization module returns?
    # How about fsolve in the same module?
    for f,x0 in zip(funcs, x0s):
        xs2.append(optimize.newton(f, x0))
        xs3.append(optimize.fsolve(f, x0)[0])

    rownames = ['x^x-10', 'sin(x)','log2(x)','8*x^6 + 1','2*x^3 -2*x^2 + 3']
    colnames = ['fsolve','optimize.newton','optimize.fsolve']

    data = np.array([xs1,xs2,xs3])
    data = np.transpose(data)

    pandas.DataFrame(data, rownames, colnames)

    # What is the advantage of using the fprime parameter?
    # (hint: try it with the function lambda x: x**4).

    # misc.derivative(lambda x: x**4,1,dx=1e-6)

    x = np.linspace(-8,8,100)
    y = np.polyval((0,0,0,0,1),x)
    y2 = np.gradient(y)
    plt.plot(x, y)
    plt.plot(x, y2)

    # the same
    fsolve(lambda x: x**4, -100)
    fsolve(lambda x: x**4, -100,lambda x: 4*x**3)

    # with fprime converges in 100 iterations
    optimize.newton(lambda x: x**4, -100,maxiter=100)
    optimize.newton(lambda x: x**4, -100, fprime = lambda x: 4*x**3,maxiter=100)

    # the same
    optimize.fsolve(lambda x: x**4, -100)
    optimize.fsolve(lambda x: x**4, -100, fprime = lambda x: 4*x**3)

    #%% Exercise 4 ########################################################################


    def initialGuess(x):
        """
        Compute the initial guess for finding the square root

        @param x: number to find the square root for
        @returns: initial guess
        """
        d = len(str(x)) # number of digits

        # number of digits is even
        if(d%2==0):
            k = (d-2)/2
            return 6*10**k
```

```
            #otherwise odd
            k = (d-1)/2
            return 2*10**k

        def sqrt(x):
            """
            Computes the (positive) square root of a positive number.

            @param x: positive number
            @returns square root

            """
            f = createPolynomial( (-x, 0, 1) ) # solve for z: z^2 -x =0

            # initial value
            x0 = initialGuess(x)

            # this will most likely results in non negative because of initial guess
            # but to be safe output positive value
            return abs(fsolve(f,x0))


        #%% Exercise 4 test

        # compare to math sqrt function to see if it is working

        xs = [0, 9, 10, 50, 100, 150, 400, 900, 1500]
        sqrts = []
        for x in xs:
            sqrts.append([math.sqrt(x),sqrt(x)])

        rownames = xs
        colnames = ['math.sqrt','sqrt']
        pandas.DataFrame(sqrts,rownames,colnames)

        #%% modifiy fsolve function to output number of iterations
        '''''
        Does the number of iterations change if you use the fprime parameter of
        your function fsolve explicitly instead of using the default
        '''
        def fsolve2(f, x0, fprime=None, xtol=1.49012e-08, maxiter=100):
            """
            Write a function fsolve(f, x0, fprime=None, xtol=1.49012e-08, maxiter=100)
            that returns one real solution of the equation defined by f(x) = 0 given
            a starting estimate x0 using Newton's method.

            @f : callable(x)
            A function that takes exactly one number argument.

            @x0 : float
            The starting estimate for the real root of f(x) = 0.

            @fprime : callable(x), optional
            A function to compute the derivative of func. If the user does not supply
            an argument, estimate the derivative as (f(x+d)-f(x-d)) / (2*d) using a
            small value of d (relative to xtol).

            @xtol : float, (optional)
            The calculation will terminate if the relative error between two consecutive

            iterates is at most xtol.
```

```
370.
371.            @maxiter : int, (optional)
372.            The maximum number of iterations. Rraise RuntimeError if this  occurs.
373.
374.            @returns one real solution of the eq. defined by f(x) = 0 and iterations
375.
376.            Raises RuntimeError exception if exceeds maxiterations
377.
378.            """
379.
380.            # estimate second derivative if not supplier by user
381.            if (fprime == None):
382.                d = xtol/10      # small value of d (relative to xtol)
383.                fprime = findDerivative(f,d)
384.
385.            # initialize variables
386.            error = 1   # difference between x and updated x (large value to start)
387.            i = 0       # iteration counter
388.            x = x0      # initial value for estimated root
389.
390.            # iterate until error in x within tolerance
391.            while (error > xtol) :
392.
393.                i += 1
394.
395.                xnew = x - f(x)/fprime(x)  #compute new x based on previous x
396.                error = abs(xnew - x)      #compute error
397.                x = xnew                   #update x
398.
399.                # raise exception if exceed maxiterations
400.                if (i > maxiter):
401.                    raise RuntimeError('Maximum Iterations ' + str(maxiter) +
402.                    ' Exceeded')
403.
404.                print('Iter: {:<2}, x = {:<20}, error: {:<40}'.format(i,xnew,error))
405.
406.            return (x,i)
407.
408.        #%% modifiy sqrt function to output number of iterations and
409.        #   have inputs for the initial guess and fprime
410.        def sqrt2(x, x0=initialGuess(x), fprime=None):
411.            """
412.            Computes the (positive) square root of a positive number.
413.
414.            @param x: positive number
415.            @param x0: initial guess (optional, default = initial guess function)
416.            @param fprime: first derivative function (optional, default = none)
417.            @returns square root and iterations
418.
419.            """
420.            f = createPolynomial( (-x, 0, 1) ) # solve for z: z^2 -x =0
421.
422.
423.            # this will most likely results in non negative because if initial guess
424.            # function is used, but to be safe output positive value
425.
426.            if (fprime==None):
427.                results =  fsolve2(f,x0)
428.                return (abs(results[0]),results[1])
429.
430.            results =  fsolve2(f,x0,fprime = lambda x: 2*x)
```

```
431.             return (abs(results[0]),results[1])
432.
433.
434.         #%%  test with initial guess funcion non fprime vs fprime
435.         sqrts = []
436.         for x in xs:
437.             results1 = sqrt2(x)
438.             results2 = sqrt2(x,fprime=None)
439.             sqrts.append([results1[0],results2[0],results1[1],results2[1]])
440.
441.         rownames = xs
442.         colnames = ['w/ fprime (val)','w/o fprime (val)',
443.                     'w/ fprime (iter)','w/o fprime (iter)']
444.         pandas.DataFrame(sqrts,rownames,colnames)
445.
446.         #%%  test with poor initial guess non fprime vs fprime
447.
448.         sqrts = []
449.         for x in xs:
450.             results1 = sqrt2(x,x0=-100)
451.             results2 = sqrt2(x,x0=-100, fprime=None)
452.             sqrts.append([results1[0],results2[0],results1[1],results2[1]])
453.
454.         rownames = xs
455.         colnames = ['w/ fprime (val)','w/o fprime (val)',
456.                     'w/ fprime (iter)','w/o fprime (iter)']
457.         pandas.DataFrame(sqrts,rownames,colnames)
```