Mario L. Gutierrez Abed
364009832
mlg3843@rit.edu

Problem Set 6
Computational Astrophysics

12-02-2021

Perform the numerical and analytical exercises below. Write up your results (preferably using LaTeX or a Jupyter notebook) in a document. Include all relevant figures and explanations. Include your codes at the end of the document.

Problem 1. *Use* `scipy`*'s built-in scalar optimization functions to compare the performance of Brent's method and the golden section method for a realistic test function (i.e., not a pure quadratic). Figure out a way of determining the number of function calls that are required to obtain the minimum.*

Solution. We shall use the function $g\colon \mathbb{R} \to \mathbb{R}$ given by

$$g(x) = x^2 \cos x \tag{1}$$

on the interval $x \in [0, 8]$ (see Fig. 1). The following Python snippet finds the local minimum using both Brent's method and the Golden Section method:

```python
import numpy as np
from scipy.optimize import minimize_scalar
from matplotlib import pyplot as plt

'''
    User-defined function
'''

def g(x):
    return x*x * np.cos(x)

brent  = minimize_scalar(g, bracket=(0,4,8), method='brent')
golden = minimize_scalar(g, bracket=(0,4,8), method='golden')

'''
    Get the location of the minimum and
    the number of function calls from both methods
'''
min_brent   = brent.x
nfev_brent  = brent.nfev
min_golden  = golden.x
nfev_golden = golden.nfev

print(
      f'Brent\'s Method took {nfev_brent} function evaluations to',
      f'find the minimum, located at x = {min_brent}.'
     )
print(f'The Golden Method took {nfev_golden} function evaluations',
      f'to find the minimum, located at x = {min_golden}.')

'''
    Plot the function and location of the minimum
'''
xgrid = np.linspace(0,8,100)
plt.plot(xgrid, g(xgrid), 'y-')
plt.plot(brent.x, brent.fun, marker="X", color='limegreen',
         markersize=9, label='Local minimum (Brent)')
plt.plot(golden.x, golden.fun, marker="*", color='cyan',
         markersize=9, label='Local minimum (Golden)')
plt.xlabel(r'$x$')
plt.ylabel(r'$g(x)$')
plt.legend(fancybox=True, framealpha=1, borderpad=1, shadow=True)
plt.show()
plt.close()
```

As the output shows, Brent's method is quite more efficient than the Golden Section method (15 function evaluations, compared to 43 function evaluations, respectively).

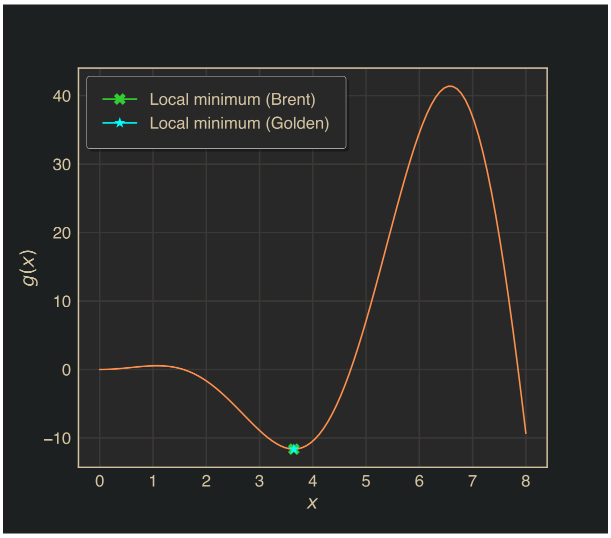

Figure 1: The function (1) evaluated on the interval $x \in [0, 8]$. The minimum found by both Brent's method and the Golden Section method is also shown.

♠

Problem 2. *For this problem, you will continue to use the* `scipy.minimize_scalar` *functions but provide your own extension to multiple dimensions. In particular, implement Powell's method (from 10.7.2) and the Conjugate gradient method. Don't use the Numerical Recipes code; rather develop your own code based on the formulas in the text. Finally, test your two implementations by minimizing the function*

$$f(x, y) = \frac{\rho^2 \sin[2(\theta - \rho)]}{1 + \rho^3}, \tag{2}$$

where $\rho = \sqrt{x^2 + y^2}$ *and* $\tan\theta = y/x$. *Start with a* guess *for the root of about* $(x, y) = (50, -50)$.

Solution. The codes for both Powell's quadratically-convergent set method and the conjugate gradient method are provided in the attached files.

```
import numpy as np
from scipy.optimize import minimize_scalar

'''
    Powell's Set Method
'''

def powell(p, func, tol  = 1.e-16, it_max = 10000):
    '''
    Initial parameters:
        p      (initial position)
        func   (function of interest)
        tol    (user-defined tolerance)
        it_max (max number of iterations allowed)

    Choose initial direction vector sets u = [e0, e1],
    where e0, e1 are the unit coordinate vectors

    Then perform 1D optimization over the line
        func(p[0] + s * e0[0], p[1] + s * e0[1])
```

```
    Take the parameter s that yields the minimum of func
    along the above line and set a new point
       p1 = p + s * e0

    From p1 repeat the above process along the direction e1
    to arrive at new point p2. Then define a new direction
    p2 - p0, and optimize again along this direction to
    arrive at another point p3.

    Rinse and repeat until a minimum has been found within
    set tolerance.

    Output:
        p_f    (found local minimum of the function)
        f(p_f) (value of the function at p_f)
        it     (number of iterations needed to find p_f)
    '''

    # define unit vectors
    e0 = np.array((1., 0.))
    e1 = np.array((0., 1.))

    # initialize direction sets u_i
    u = [e0, e1]

    diff = 1.
    it   = 0

    while diff > tol:

        it += 1
        if it == it_max:
            print(f'Aborting after {it} iterations...Unable to find minimum.')
            break

        p_old = p
        f_old = func(p[0],p[1])

        for ui in u:
            # perform 1D optimization
            f_1dim = lambda s : func(p[0] + s * ui[0], p[1] + s * ui[1])
            brent  = minimize_scalar(f_1dim, method='brent')
            # update to new point p0 -> p0 + s*u
            s = brent.x
            p = p + s * ui

        # update direction sets u_i
        u[0] = u[1]
        u[1] = p - p_old

        # perform a further 1D optimization, along the new direction of u[1]
        f_1dim = lambda s : func(p[0] + s * u[1][0], p[1] + s * u[1][1])
        brent  = minimize_scalar(f_1dim, method='brent')
        s      = brent.x
        p      = p + s * u[1]

        # evaluate function at the new location and compare with previous value
        f_new = func(p[0],p[1])
        diff  = np.abs(f_old - f_new)

    return p, func(p[0], p[1]), it



'''
    Conjugate Gradient Method
'''

def conj_grad(p, func, grad, tol = 1.e-16, it_max = 10000):
    '''
    Initial parameters:
        p      (initial position)
        func   (function of interest)
        grad   (gradient of func)
```

```
        tol     (user-defined tolerance)
        it_max (max number of iterations allowed)

    Initial direction vector g0:
        g_0 = - grad(p)

    Set h_0 = g_0

    Then
        g_{i+1} = - grad(p_{i+1})
        h_{i+1} = g_{i+1} + \gamma_i h_i

    where
        gamma_i = dot(g_{i+1} - g_i, g_{i+1}) / dot(g_i, g_i)

    Output:
        p_f     (found local minimum of the function)
        f(p_f) (value of the function at p_f)
        it      (number of iterations needed to find p_f)
    '''

    # here we need to use '[:,0]' because of the way sympy's lambdify works
    # if not using lambdify, just remove '[0,:]'
    g = - grad(p[0], p[1])[:,0]
    h = g

    diff = 1.
    it   = 0

    while diff > tol:

        it += 1
        if it == it_max:
            print(f'Aborting after {it} iterations...Unable to find minimum.')
            break

        f_old = func(p[0],p[1])
        g_old = g

        # perform 1D optimization
        f_1dim = lambda s : func(p[0] + s * h[0], p[1] + s * h[1])
        brent  = minimize_scalar(f_1dim, method='brent')
        # update to new point p0 -> p0 + s*h
        s = brent.x
        p = p + s * h

        # evaluate function at the new location and compare with previous value
        f_new = func(p[0],p[1])
        diff  = np.abs(f_old - f_new)

        # if diff > tol not yet satisfied continue loop...
        g      = - grad(p[0], p[1])[:,0]
        gamma  = np.dot(g - g_old, g) / np.dot(g_old, g_old)
        h      = g + gamma * h

    return p, func(p[0], p[1]), it
```

Before applying the conjugate gradient method, we need to calculate the gradient of the function (2):

```
from sympy import *

''' Write the function in symbolic form '''
var(('x', 'y', 'rho', 'theta'))
f_symb = rho**2 * sin(2*(theta - rho)) / (1 + rho ** 3)

''' Get its gradient '''
g = f_symb.subs({rho : sqrt(x**2 + y**2), theta : atan(y/x)})

sym_grad_f_x = diff(g,x)
sym_grad_f_y = diff(g,y)
sym_grad_f   = Matrix([
                    [sym_grad_f_x],
                    [sym_grad_f_y]
                  ])
```

```python

''' Convert from symbolic to Python expression'''
grad_f =lambdify((x,y), sym_grad_f)
```

Now we call the methods to find the minimum of the function (2):

```python
'''
    Find the minimum using both Powell's Method and CG
'''
# starting guess
p0 = np.array((50., -50.))

min_powell   = powell(p0, f)[0]
f_min_powell = powell(p0, f)[1]
it_powell    = powell(p0, f)[2]

min_CG   = conj_grad(p0, f, grad_f)[0]
f_min_CG = conj_grad(p0, f, grad_f)[1]
it_CG    = conj_grad(p0, f, grad_f)[2]


print('Using Powell\'s method, '
      f'the minimum f = {f_min_powell} was found at p = {min_powell}',
      f'after {it_powell} iterations.')


print('\nUsing the Conjugate Gradient method, '
      f'the minimum f = {f_min_CG} was found at p = {min_CG}',
      f'after {it_CG} iterations.')
```

As the output shows, both methods find the minimum at $p = (-1.12071341, -0.5756757)$ with a function value of $f(p) = -0.5291336839893999$. The rather surprising bit here is that the conjugate gradient took longer than Powell's method to find the minimum within the specified user-set tolerance of 10^{-16}; the former took 3970 iterations, while the latter took 2146 iterations. ♠