Computational Physics HW1

Yue (Fred) Shi

October 2019

1 Answers

(a) q1:$\epsilon' = -w\epsilon + (1+w)f'(x^*)$

q2:We rewrite the expression for $\epsilon' = \epsilon[(1+w)f'(x^*)-w]$, then $\epsilon = \frac{\epsilon'}{[(1+w)f'(x^*)-w]}$, and as $x^* = x' + \epsilon' = x + \epsilon$, we can rewrite $x - x' = \epsilon' - \epsilon = \epsilon'[1 - \frac{1}{(1+w)f'(x^*)-w}]$ such that $\epsilon' = \frac{x-x'}{1-\frac{1}{(1+w)f'(x^*)-w}} = \frac{x-x'}{1-\frac{1}{(1+w)f'(x)-w}}$ when x is near x^*

(b) Fig 1 is the code for the relaxation method and it takes 14 recursion to converge below tolerance

(c) Fig 2 is the code for the over-relaxation method and it takes 5 recursion to converge below tolerance for w=0.5 for my initial condition. It also takes 5 recursions for w=0.7 and takes more time when w is smaller than 0.5. For initial guesses that are more distant from the answer, however, big w can cause overflow and smaller w (eg. w=0.1 for initial guess x_0=3) can speed up this process by a factor of two.

(d) Setting $w < 0$ can help solve the cases where normal relaxation method cannot solve. For example, for the function $f(x) = e^{1-x^2}$, the normal relaxation cannot solve it as $\epsilon' > \epsilon$ due to its first derivative at x^*. Thus with negative w, we can make $\epsilon' = -w\epsilon + (1+w)f'(x^*) < \epsilon$ so that it converges and hence solve the equation. This is helpful particularly when we cannot re-write the equations that normal relaxation can solve.

2 Answers

(a)We first calculate the derivative:

$$\frac{dI(\lambda)}{d\lambda} = (-5)\frac{2\pi hc^2\lambda^{-6}}{e^{hc/\lambda K_b T}-1} + (2\pi hc^2\lambda^{-5})(\lambda^{-2})\frac{hc}{K_b T}\frac{e^{hc/\lambda K_b T}}{(e^{hc/\lambda K_b T}-1)^2} \tag{1}$$

As we are given $\frac{dI(\lambda)}{d\lambda} = 0$, then we multiply $\frac{e^{hc/\lambda K_b T}-1}{2\pi hc^2\lambda^{-6}}$ to the both side:

$$0 = (-5) + \frac{hc}{\lambda K_b T}\frac{e^{hc/\lambda K_b T}}{(e^{hc/\lambda K_b T}-1)} \tag{2}$$

Again, times $\frac{(e^{hc/\lambda K_b T}-1)}{e^{hc/\lambda K_b T}}$ to both side and we get:

$$0 = 5e^{-hc/\lambda K_b T} - 5 + \frac{hc}{\lambda K_b T} \tag{3}$$

In [2]:
```
#Q1 b
def inputfun(x):
    return 1-np.exp(-2*x)

def relax(x,func):
    e = 1
    c = 0
    while e>10**(-6):
        c+=1
        x_old = x
        x = func(x)
        e = abs(x_old - x)
    print(c)
    return x
```

In [3]:
```
relax(1,inputfun)
```

```
14
```

Out[3]: 0.7968126311118457

Figure 1: Code for relaxation method with counted iterations

Denote $x = \frac{hc}{\lambda K_b T}$, we get

$$0 = 5e^{-x} - 5 + x \tag{4}$$

As we get the maximum intensity $I(\lambda)$ when x is the root of eqn. 4, we denote $b = \frac{hc}{k_B x}$ such that $\lambda = b/T$ for λ to be the wavelength of the maximum intensity. Thus it obeys the Wien displacement law.
(b) Fig 3 is the code for the bisection method as well as the answer for x
(c) After substitute the constants and the value for the displacement constant into the Wien displacement law, we get $T = 10^4 \frac{6.626*3}{1.3787*4.965*5.02} = 5785K$, which is quite close to the actual temperature of the sun.

3 Answers

Fig4, Fig5 and Fig6 is my gradient descend code for two and three variables and their utility functions respectively. For the two variables one, I have a fixed step size for 0.01 and gamma also 0.01. For the three variables, as the input Schechter function is exponential and pretty sensitive, I set the step as the difference between the difference between each step and gamma 0.0001. Also, I used central difference method for two variable cases and forward difference method for three variables as I found that with the given problem, forward difference method somehow works better.

Fig 7 is the value of the test function under gradient descent, you can see that it is decreasing initially and become more and more flat(reaches local minimal) as iterations going on.

In [4]:
```
#Q1c
def overrelax(x, func, w):
    e = 1
    c = 0
    while (abs(e)>10**(-6)):
        c+=1
        x_old = x
        x = (1+w)*func(x_old)-w*x_old
        e = x - x_old
        print(x)
    print(c)
    return x
```

In [20]:
```
overrelax(1, inputfun, 0.5)
```

```
0.796997075145081
0.7968323724281932
0.7968143476015649
0.7968123729832619
0.7968121566399141
5
```

Out[20]: 0.7968121566399141

Figure 2: Code for over-relaxation method with counted iterations

Fig8 is the best fit results I got. Fig9, Fig10, and Fig11 are different results with different initial guess, with the orange line the best fit result (Fig7). As you can see, although they are not perfect match with the orange line, they are close to it. Note here in my code the initial guess for the $LogM^*$ must be bigger than 11, otherwise the function will have overflow issue.

Fig 12, 13 and 14 are the plots for the chi square function of step under the initial guesses in Fig9,10,11 respectively. You can see that they converge quickly first and then slows down to get a more accurate answer.

```
In [2]: def bisect(fun,rang):
            x1 = rang[0]
            x2 = rang[1]
            #check if the input is correct
            if fun(x1)*fun(x2) > 0:
                print("Error")
                return 0
            else:
                while abs(fun(x1))>10**(-6):
                    x3 = (x1+x2)/2
                    if fun(x3)*fun(x2)>0:
                        x2 = x3
                    else:
                        x1 = x3
                return x1

        def equation(x):
            return 5*np.exp(-x)+x-5

In [5]: bisect(equation,[0.5,7])

Out[5]: 4.965113580226898
```

Figure 3: Code for bisection method and answer for x

```
In [6]: def central_difference(func, x, h):
            f_prime = ((func(x+h) - func(x-h))/(2*h))
            return f_prime

        def test1(x, y):
            return (x-2)**2+(y-2)**2

        def gradient_descend(fun, x, y):
            h = 0.01
            xp = 1
            yp = 1
            while abs(xp)>10**(-6):
                xp = central_difference(lambda x:fun(x, y), x, h)
                yp = central_difference(lambda y:fun(x, y), y, h)
                x = x - xp*h
                y = y - yp*h
            return x, y

In [7]: gradient_descend(test1, -100, 100)

Out[7]: (1.9999995189989725, 2.000000462138241)
```

Figure 4: Code for two variable gradient descend

```
:
def sche(phi,M_s,a,M):
    return phi*np.log(10)*np.exp(-(10**M)/(10**M_s))*((10**M/10**M_s)**(a+1
def cost_fun(phi, M_s, a, M, out):
    return sum((out-sche(phi,M_s,a,M))**2/sche(phi,M_s,a,M))
def inputfun(x1,x2,x3):
    return cost_fun(x1,x2,x3,logM,y)
def forward_difference(func,x,h):
    f_prime = (func(x+h) - func(x))/h
    return f_prime
def central_difference(func,x,h):
    f_prime = ((func(x+h) - func(x-h))/(2*h))
    return f_prime
```

Figure 5: Code for three variable gradient descend

```
In [98]:
def gradient_descend(fun,x,y,z,tol):
    c = 0
    maxite = 100
    xp = 1
    yp = 1
    zp = 1
    step = 1
    h = 0.0001
    hx = 0.0001
    hy = 0.0001
    hz = 0.0001
    x2=x+h;
    y2=y+h;
    z2=z+h;
    chi2 = []
    while step>tol and c<maxite:
        xp = forward_difference(lambda x:fun(x,y,z),x,hx)
        yp = forward_difference(lambda y:fun(x,y,z),y,hy)
        zp = forward_difference(lambda z:fun(x,y,z),z,hz)
        x3 = x2 - h*xp
        y3 = y2 - h*yp
        z3 = z2 - h*zp
        hx = x2-x
        hy = y2-y
        hz = z2-z
        x = x2
        x2 = x3
        y = y2
        y2 = y3
        z = z2
        z2 = z3
        chi2.append(fun(x,y,z))
        step = max(abs(hx),abs(hy),abs(hz));
        c+=1

    return x,y,z,chi2
```

Figure 6: Code for utility functions

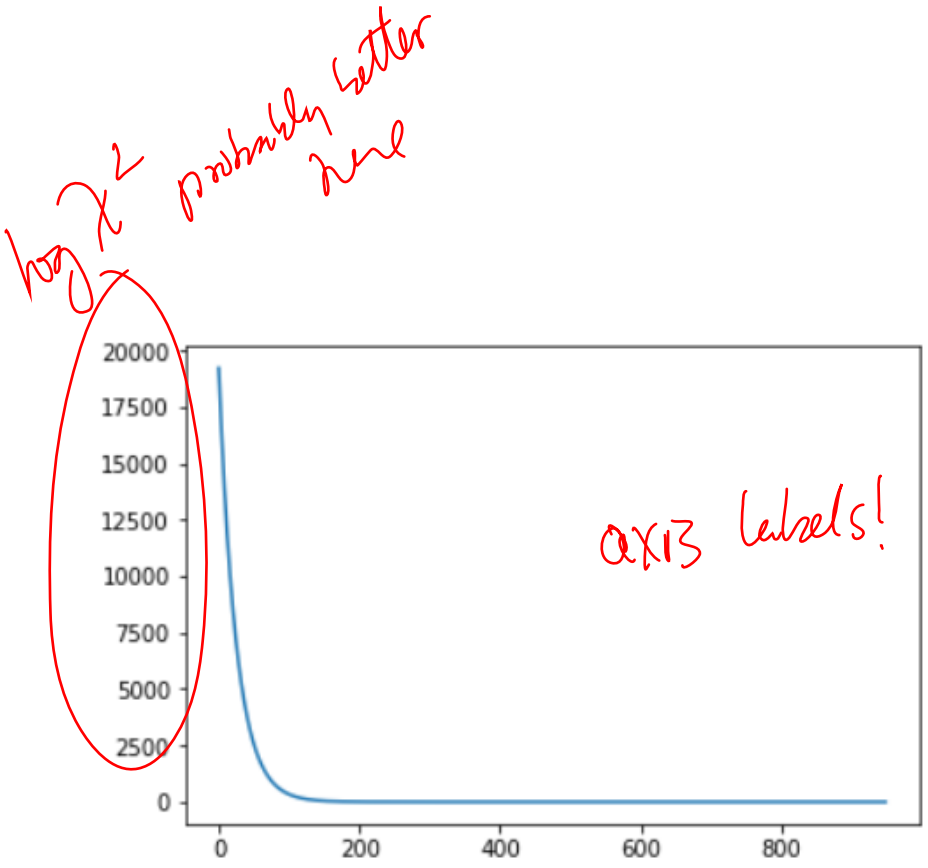

Figure 7: value for test function of two variables gradient descend

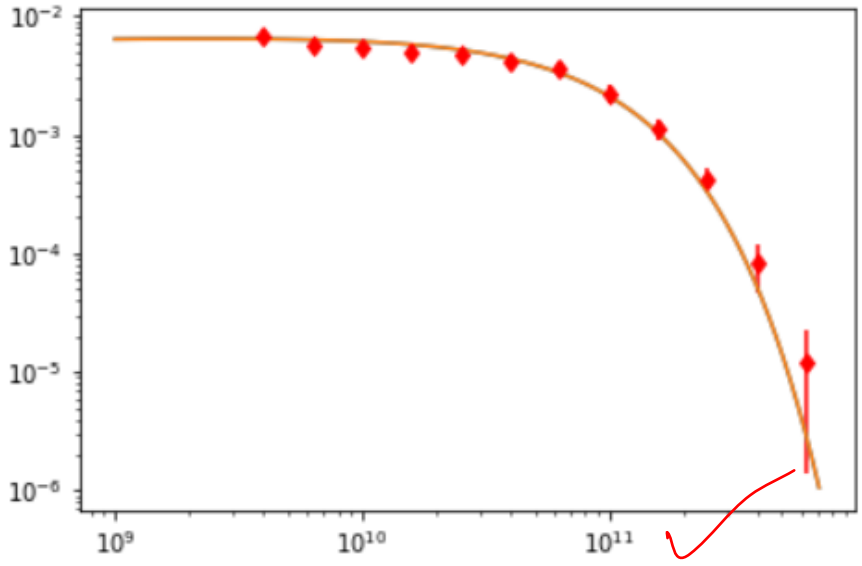

Figure 8: Best fit for the Schehster Function

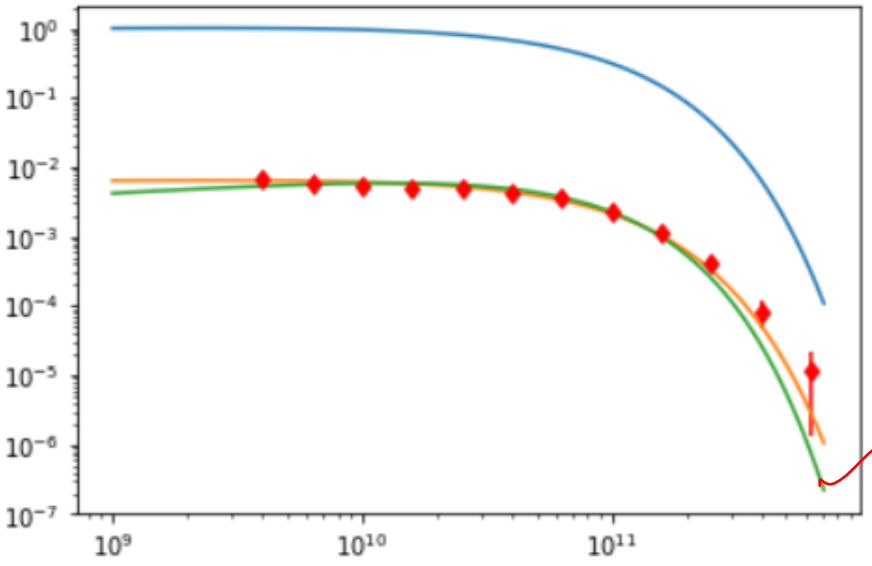

Figure 9: fit1 for the Schehster Function with $\phi = 0.5$,$M^* = 10.88$,$a = -0.97$

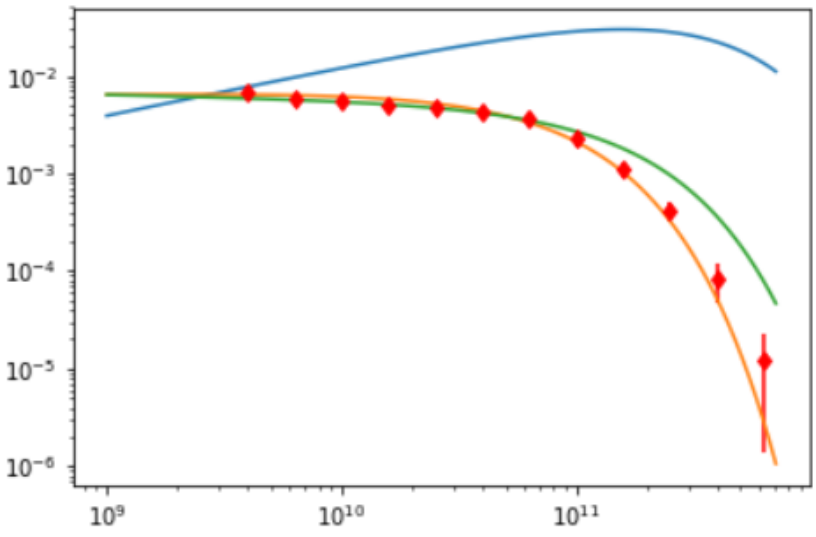

Figure 10: fit2 for the Schehster Function with $\phi = e^{-3.5}$,$M^* = 11.5$,$a = -0.5$

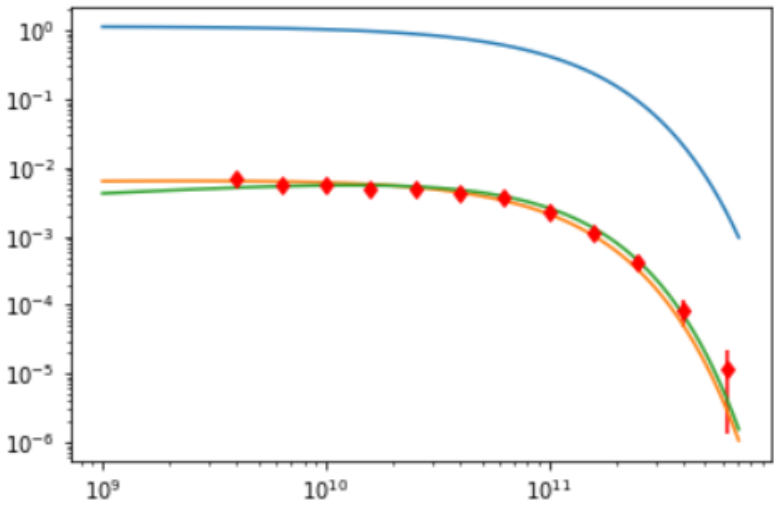

Figure 11: fit3 for the Schehster Function with $\phi = 0.5$,$M^* = 11$,$a = -1$

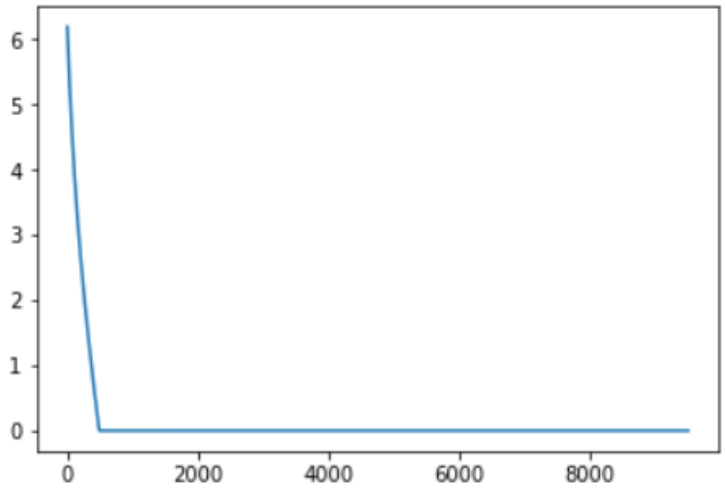

Figure 12: chi2 as function of iteration for fit1

```
plt.plot(chi)
```

```
[<matplotlib.lines.Line2D at 0x7f2a12823860>]
```

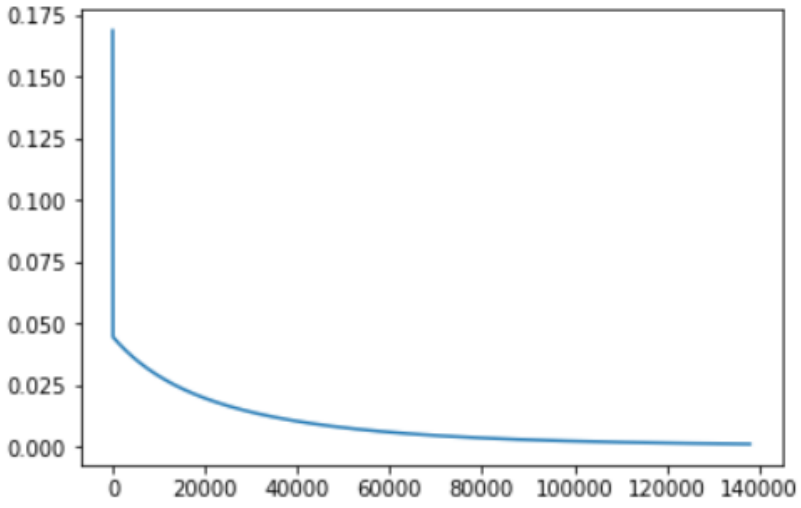

Figure 13: chi2 as function of iteration for fit2

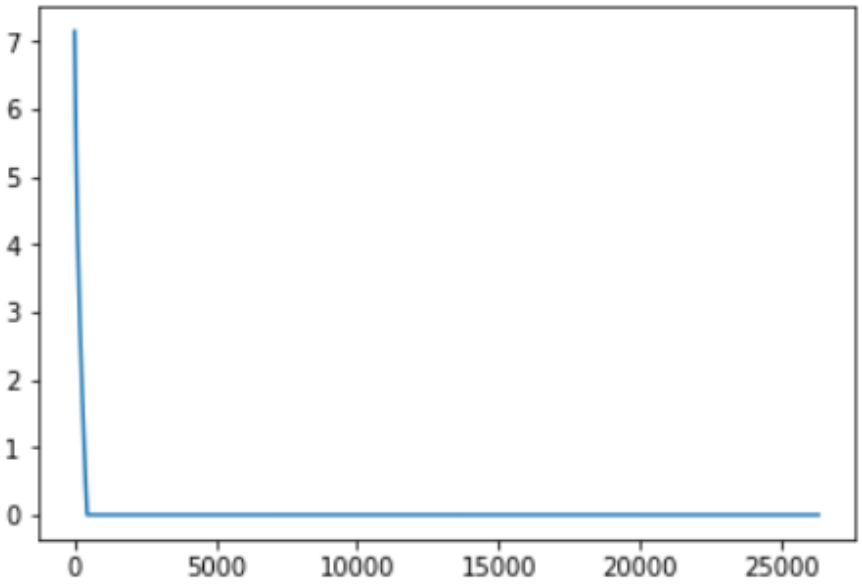

Figure 14: chi2 as function of iteration for fit3