HomeWork 2 Computational Physics

10/10

Giorgi Arsenadze (ga1348)

Due to: 11 OCT 2019

1

1.1

In this part we had to derive formula for the error of the overrelaxation method. Same formula was derived in the book (Newman-Computational Physics), but for the relaxation method. Using same approach it is relatively easy to derive formula for the errors. First lets start with the error in x' which is the measurement before the real value x^*

$$x^* = x' + \epsilon'$$

Now error for the measurement before x' which we call x

$$x^* = x + \epsilon$$

now using overrelaxation equation and expanding it in the point x^* we will get the following:

$$x' = (1+\omega)f(x) - \omega x$$

$$x' = (1+\omega)(f(x^*) + (x-x^*)f'(x^*)) - \omega x$$

now with small mathematical manipulations and remembering that $x^* = (1+\omega)f(x^*) - \omega x^*$ We can write the following:

$$x' = (1+\omega)f(x^*) - \omega x^* - \omega(x-x^*) + (1+\omega)(x-x^*)f'(x^*)$$

$$x' - x^* = ((1+\omega)f'(x^*) - \omega)(x-x^*)$$

the last equation directly gives us relationship between ϵ' and ϵ

$$\epsilon' = ((1+\omega)f'(x^*) - \omega)\epsilon$$

Now going back to the definitions of errors we can write

$$x^* = x' + \epsilon' = x + \epsilon = x + \epsilon'/((1+\omega)f'(x^*) - \omega)$$

$$x' - x = \epsilon'(1/((1+\omega)f'(x^*) - \omega) - 1)$$

And after straightforward simplifications we finally get new equation for errors

$$\epsilon' = \frac{x - x'}{1 - \frac{1}{(1+\omega)f'(x^*) - \omega}}$$

1.2

Now using relaxation method we have to solve equation:

$$x = 1 - e^{-2x}$$

The accuracy we need is $\epsilon = 10^{-6}$. We also have to print the number of iterations it takes. For this problem I used MATLAB as my main coding software. The code for this section could be found in the Appedinx A. MATLAB just took $c = 12$ interactions to find the answer for the starting point $x_0 = 1$. Answer given was:

$$x = 0.796812$$

1.3

For this section we have to implement the overrelaxation method and solve the same equation using ovverrelaxation. I used $\omega = 0.5$ and my starting point was still $x_0 = 1$. The interactions MATLAB took this time was $c = 3$ to converge to the answer:

$$x = 0.796813$$

1.4

For this section it is helpful to remember in which cases does the relaxation method diverge. We talked in the lecture that when $|f'(x_+)| < 1$ the relaxation method converges. It will diverge if $|f'(x_+)| > 1$. So the way we treat this kind of situation is that we try to get new equation for the same problem, analytically, but with different function which will not have $|f'(x_+)| > 1$ and try to solve the equation using different function. But Another way of doing this I think would be to try under-relaxation. which will get closer to root with much smaller steps and might help us avoid divergence.

2

2.1

In this problem we wanted to find value of λ which minimizes the function:

$$I(\lambda) = \frac{2^2\lambda^-5}{e^{\frac{-hc}{k_BT\lambda}} - 1}$$

derivative of this function must be equal to 0 if we want to find critical point. After derivation we get:

$$5e^{\frac{-hc}{k_BT\lambda}} + \frac{hc}{\lambda k_BT} - 5 = 0$$

After we substitute $x = \frac{hc}{\lambda}$ we have to solve:

$$5e^{-5} + x - 5 = 0$$

and after finding x we can use it in the equation:

$$\lambda = \frac{b}{T}$$

Where $b = \frac{hc}{k_Bx}$

2.2

In this section I wrote a program which solves above equation. Accuracy we wanted is $\epsilon = 10^{-6}$. The method I use is simple binary search. Initial starting points were $x_1 = 4$ and $x_2 = 6$. Code for this Method could be found in the Appendix B. After running the program I found that the solution of our equation is:

$$x = 4.965114$$

2.3

And finally, in this section we have to calculate surface temperature of the Sun using our solution and the constants. Therefore, to estimate surface temperature we have to use following constants (note that every constant is given up to 10^{-6} precision cause it does not make sense to use better precision than x has) :

$$h = 6.62607004 * 10^{-34}$$

$$c = 299792458$$

$$k_B = 1.38064852 * 10^{-23}$$

$$\lambda = 502 * 10^{-9}$$

Remember that b is defined following way:

$$b = \frac{hc}{k_B x}$$

And finally the temperature will be:

$$T = b\lambda = 5.772456 * 10^3$$

3

In this problem we had to implement the Gradient Descent Algorithm and using that we had to find three parameters in Schechter function. The Schechter function can be written following way:

$$n(M_{gal}) = \Phi^*(\frac{M_{gal}}{M^*})^{\alpha+1} exp(-\frac{M_{gal}}{M^*}) ln10$$

Our mission in this section is to find parameters Φ^*, M^*, α so that this function can be good fit to the measurements of the galaxy stellar mass function. The measurements are given. We have to fit Schechter function to the given Data using χ^2 function, which reads:

$$\chi^2(\Phi^*, \alpha, M^*) = \sum (n_{data}(M^i_{gal}) - n_{Schechter}(M^i_{gal}))^2 / e_i^2$$

where $n_{data}(M^i_{gal})$ is galaxy stellar mass function evaluated for the M^i_{gal}. $n_{Schechter}(M^i_{gal})$ is Schecher's function evaluated for the M^i_{gal}. e_i is the error of the given data.

Our goal is to find parameters Φ^*, M^*, α so that χ^2 function is minimal. Therefore, using that parameters Schechter's function will be good description of our data.

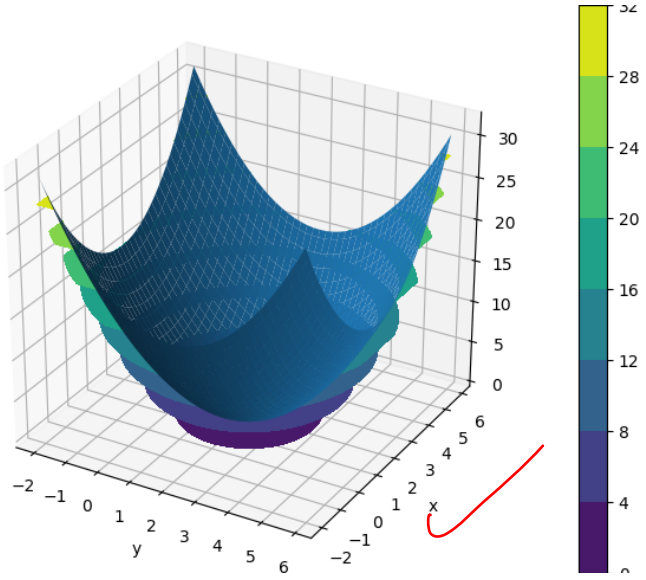
3.1

Before we start minimizing the χ^2 function lets implement the Gradient Descent method and test it for the functions:

$$f(x, y) = (x-2)^2 + (y-2)^2$$
$$g(x, y, z) = (x-2)^2 + (y-3)^2 + (z-4)^2$$

$f(x, y)$ was required by the problem, but as minimizing χ^2 function is equivalent to minimizing some function of three variables, I decided that testing the

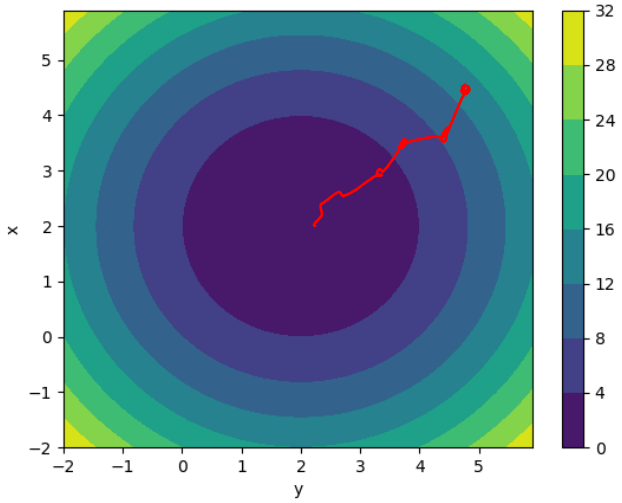
code for the $g(x, y, z)$ would be a good idea. I will provide plots for the $f(x, y)$ only, but same happens for $g(x, y, z)$. First of all lets plot the $f(x, y)$ to have an idea where the minimal point is. Obviously $x = 2$ and $y = 2$ minimizes the function but still good to see the behaviour on the plot.



nice, but probably overkill.

Figure 1:

In the first plot (Fig.1) you see the 3D figure of the given function. Colors show how low is the given level of the function in z direction. Next plot will provide more accurate placement of the minimal value.



hm... this doesn't really show me anything about your algorithm's performance.
Show points.

Figure 2:

Here we have the projection of $f(x, y)$ on the xy plane and still colors show approximately what is the value of the function on top of the points. Obviously, function must have minimum on top of the $(2, 2)$, therefore we see that colors get darker as we get closer to the point $(2, 2)$. Now lets check our implementation of the Gradient Descent algorithm in action. When I run the algorithm for $f(x, y)$ and plot the behaviour of $x_i - x_{i-1}$ I get the following plot (Fig.3). Which means that our answer is converging and obviously answers given were near $x = 2$ and $y = 2$ ($x = y = 2.000000140$ to be exact, and I used tolerance 10^{-7}).

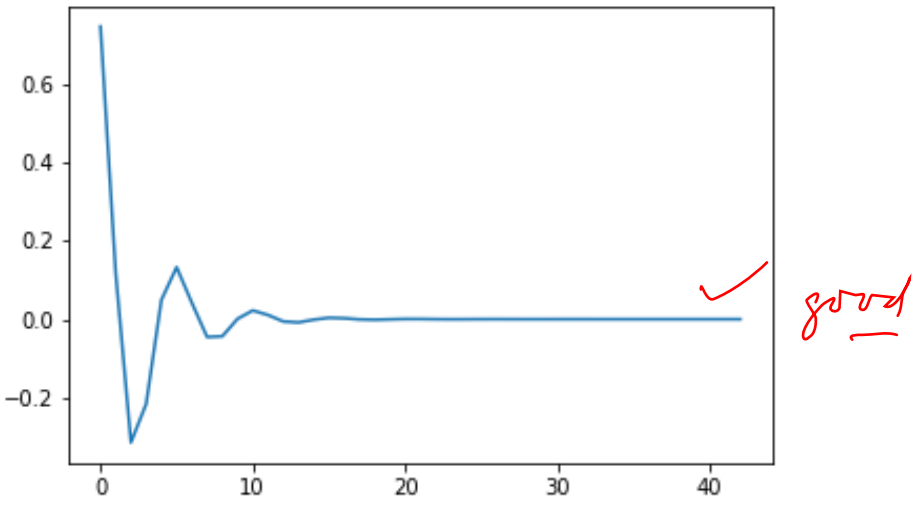


Figure 3: Dependence of $x_i - x_{i-1}$ on number of iterations

My implementation of the Gradient Descent algorithm for three variable function can be found in the Appendix C.

3.2

In this section we will start to find above mentioned three parameters, which minimizes the χ^2 function. First of all, lets change variables (parameters) a little bit, so that we have to work on same range of numbers for all the parameters. Our new variables are:

$$Log\Phi^* \equiv x$$

$$LogM_{gal} \equiv M_n$$

$$LogM^* \equiv m_n$$

$$\alpha \equiv \alpha$$

In this new variables Schechter function will look like the following:

$$n = 10^x(10^{M_n - m_n})^{\alpha+1} exp(-10^{M_n - m_n}) ln10$$

Using new variables its way faster to find the values which minimizes the χ^2 function, because it does not have to go up to the 10^{11} for the M^* and it stays

within the range of $-10-20$ which makes it way easier to find desired values quickly. Using the Gradient Descent method for three variable functions, lets see the algorithm in action for the χ^2 function. Code for this one can be found in the Appendix C. As it is hard to plot three variable function, I decided to plot how the difference between new and old points behaves and as it goes to zero that means that the Gradient Descent algorithms converges. One of the best results were when I used initial guesses: $x = -3.5; \alpha = 0.5; m_n = 11.5$ and used step size $\delta = 0.0001$ for all the variables.Iused $\gamma = 0.0001$ and my tolerance was 10^{-6}. In that case behaviour of $x_i - x_{i-1}$ difference for all the parameter could be seen on the Fig.4. As the plot shows all of them converge.

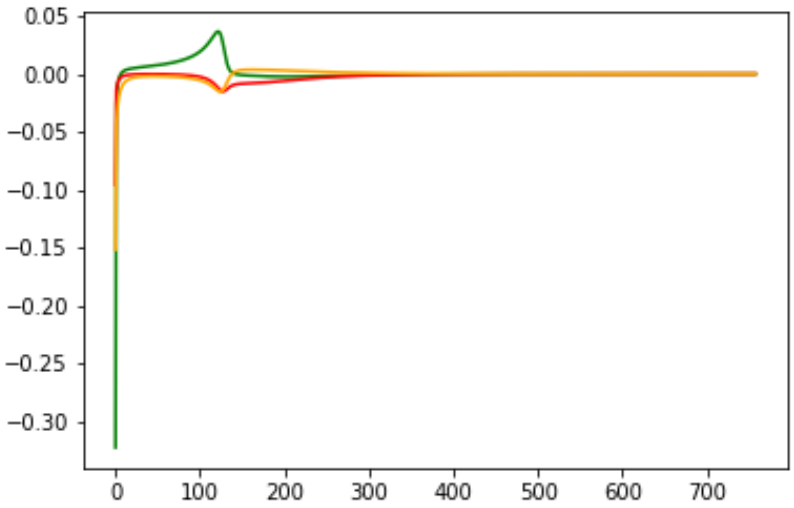


Figure 4: $x_i - x_{i-1}$-green, $y_i - y_{i-1}$-red $z_i - z_{i-1}$-orange

Now we can try different initial guesses and see how final answer will fit the data. For example, for the initial guesses: $x = -3.0; \alpha = 1.2; m_n = 11.0$ and used step size $\delta = 0.0001$ for all the variables. My tolerance was 10^{-6}. The result can be seen on the Fig.5. Fig.5 shows what initial guesses look like and what it becomes after fitting.

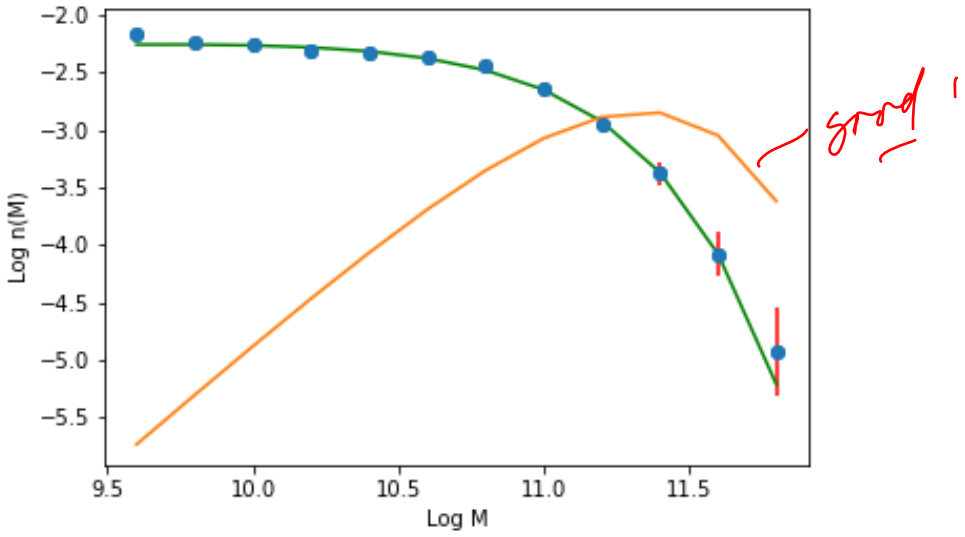


Figure 5: Orange-Schechter function for the initial guess, Green-final fit. Blue dots-data

So on the Fig.5 it can be seen how old Schechter function becomes new one, which fits the data. Actual fit can be properly seen on the Fig.6. For this plot my final values of new variables (parameters) were:

$$Log\Phi^* = -2.523353$$

$$LogM^* = 10.947206$$

$$\alpha = -0.944311$$

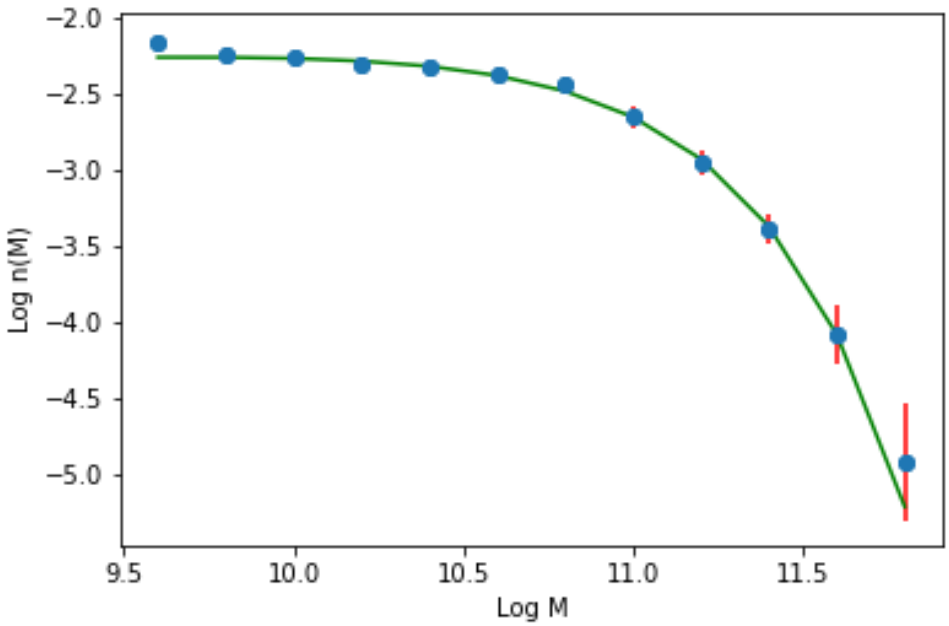


Figure 6: reen-final fit. Blue dots-data

Now to make sure that χ^2 minimizes, lets plot how it behaves during the whole computational process. For the above initial guesses the behaviour of χ^2 can be seen on Fig.7.

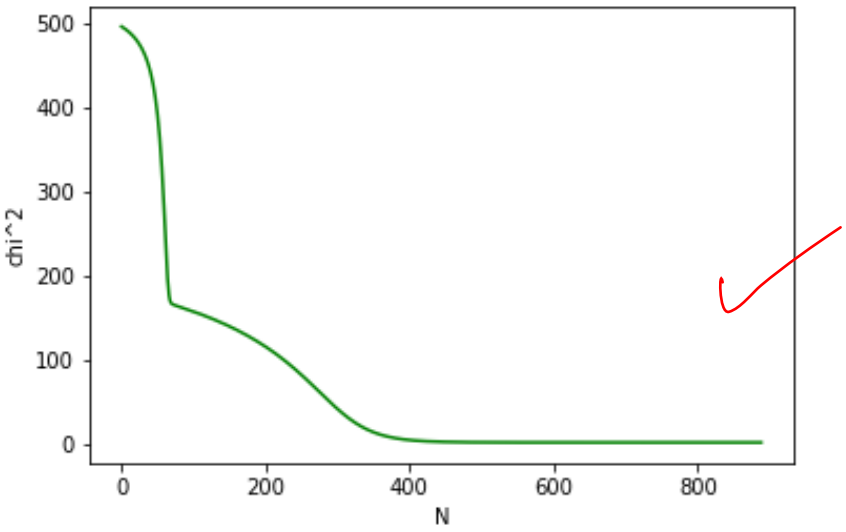
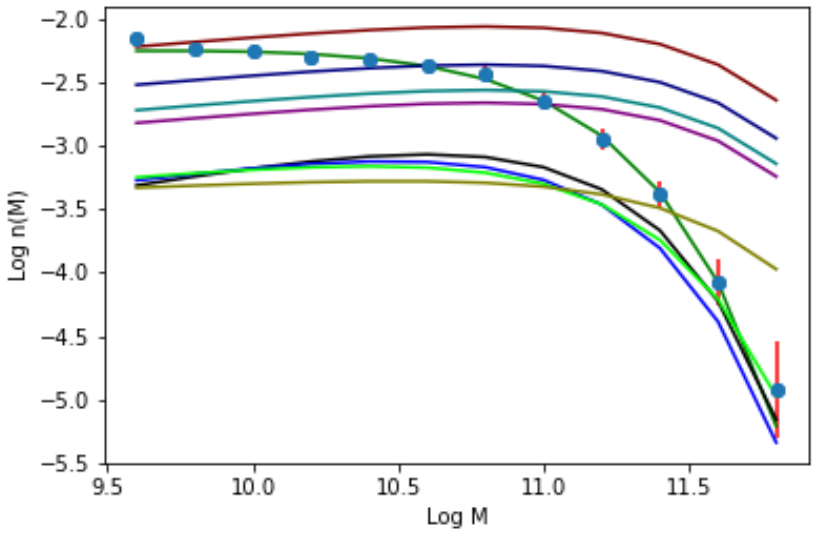


Figure 7: Dependence of χ^2 on iterations

And finally, I wanted to show how our function fits the data for the different starting points. Therefore, on the Fig.8 you will see bunch of plots of initial schechter functions and then the final plot. All the given initial plots converged to the green line. Initial points will be shown in the description of the Fig.



very thorough!

Figure 8: (parameters given respectfully: $Log\Phi^*$, α, $LogM^*$) Green-fitted line $-2.522771, -0.9436232, 10.946810$ (it was impossible to differentiate fitted lines for each of the initial guesses, therefore I decided to just provide one of the fits), blue dots-data, black-$-3.1, -0.6, 11$, blue- $-3.2, -0.7, 11$, lime- $-3.3, -0.8, 11.1$, olive- $-3.5, -0.9, 11.5$, purple- $-2.8, -0.8, 11.5$, teal- $-2.7, -0.8, 11.5$, navy- $-2.5, -0.8, 11.5$, maroon- $-2.2, -0.8, 11.5$

4 Appendix A

```
function [x] = overrelaxation(w,f,e,x0)
%UNTITLED5 Summary of this function goes here
%   Detailed explanation goes here
    c=0;
    xo=x0;
    xn=eval((1+w)*f(xo)-w*xo);
    while abs(xn-f(xn))>e
        c=c+1;
        xo=xn;
        xn=eval((1+w)*f(xo)-w*xo);
    end
    c
    x=xn;

end
```

Figure 9:

```
function [x] = relaxation(f,e,x0)
%UNTITLED7 Summary of this function goes here
%   Detailed explanation goes here
    c=0;
    xo=x0;
    xn=eval(f(xo));
    while abs(xn-f(xn))>e
        c=c+1;
        xo=xn;
        xn=eval(f(xo));
    end
    c
    x=xn;
end
```

Figure 10:

5 Appendix B

```
function [xm] = binary(a,b,f,e)
%UNTITLED3 Summary of this function goes here
%   Detailed explanation goes here
    x1=a;
    x2=b;
    while abs(x1-x2)>e
        xm=1/2*(x1+x2)
        if f(xm)==0
            break;
        elseif sign(f(xm))==sign(f(x1))
            x1=xm;
        else
            x2=xm;
        end
    end
    xm=1/2*(x1+x2);


end
```

Figure 11:

6 Appendix C

```
def dec(x0,dx,y0,dy,z0,dz,w,r1,r2,r3):

    x=x0+dx-w*(chi(x0+dx,y0,z0)-chi(x0,y0,z0))/dx;
    y=y0+dy-w*(chi(x0,y0+dy,z0)-chi(x0,y0,z0))/dy;
    z=z0+dz-w*(chi(x0,y0,z0+dz)-chi(x0,y0,z0))/dz;

    while abs(x-x0)>r1 or abs(y-y0)>r2 or abs(z-z0)>r3:

        x0=x;
        y0=y;
        z0=z;

        x=x0+dx-w*(chi(x0+dx,y0,z0)-chi(x0,y0,z0))/dx;
        y=y0+dy-w*(chi(x0,y0+dy,z0)-chi(x0,y0,z0))/dy;
        z=z0+dz-w*(chi(x0,y0,z0+dz)-chi(x0,y0,z0))/dz;
        #fp=x
        #fa=y
        #fm=z
        #chip.append(chi(x,y,z))
    return x,y,z;
```

Figure 12:

```
def schechter(pn,a,mn,Mn):
    ratio=10**(Mn-mn);
    exp=nm.exp(-ratio);
    return (10**pn)*(ratio)**(a+1)*exp*nm.log(10);
```

```
def chi(p1,a1,m1):
    I=0;
    for i in range(1,N):
        M1=logMg[i];
        e=er[i];
        n=n0[i];
        I=I+(n-schechter(p1,a1,m1,M1))**2/e**2;
    return I;
#chi(0.001,1,112441000000)
```

Figure 13:

7 References

1. Mark E. J. Newman- Computational Physics.
2. Steven Chapra -Applied Numerical Methods with MATLAB for Engineers and Scientists.