Ph22.0 Finding Roots Assignment

LIM SOON WEI DANIEL

1. Let $x^{(j)}$ be close to the root position x. We replace $x^{(j)}$ with its deviation from the root position:

$$x^{(j)} = x + \epsilon_j$$

and replace $f(x^{(j)})$ with the second order Taylor approximation:

$$f(x^{(j)}) = f(x) + \epsilon_j f'(x) + \frac{1}{2}\epsilon_j^2 f''(x) + \ldots = \epsilon_j f'(x) + \frac{\epsilon_j^2}{2} f''(x)$$

Then the secant method primary step can be written as:

$$x^{(j+2)} = x^{(j+1)} - f(x^{(j+1)})\frac{x^{(j+1)} - x^{(j)}}{f(x^{(j+1)}) - f(x^{(j)})}$$
$$\implies \epsilon_{j+2} = \epsilon_{j+1} - (\epsilon_{j+1} f'(x) + \frac{\epsilon_{j+1}^2}{2} f''(x))\frac{\epsilon_{j+1} - \epsilon_j}{\epsilon_{j+1} f'(x) + \frac{\epsilon_{j+1}^2}{2} f''(x) - \epsilon_j f'(x) - \frac{\epsilon_j^2}{2} f''(x)}$$

$$\begin{aligned}
\implies \epsilon_{j+2} &= \epsilon_{j+1} - \left(\epsilon_{j+1} f'(x) + \frac{\epsilon_{j+1}^2}{2} f''(x)\right)\frac{1/f'(x)}{1 + \frac{\epsilon_{j+1}^2 - \epsilon_j^2}{\epsilon_{j+1} - \epsilon_j}\frac{f''(x)}{2f'(x)}} \\
&= \epsilon_{j+1} - \left(\epsilon_{j+1} f'(x) + \frac{\epsilon_{j+1}^2}{2} f''(x)\right)\frac{1/f'(x)}{1 + (\epsilon_{j+1} + \epsilon_j)\frac{f''(x)}{2f'(x)}} \\
&\approx \epsilon_{j+1} - \left(\epsilon_{j+1} f'(x) + \frac{\epsilon_{j+1}^2}{2} f''(x)\right)\frac{1}{f'(x)}\left(1 - (\epsilon_{j+1} + \epsilon_j)\frac{f''(x)}{2f'(x)}\right) \\
&= \epsilon_{j+1} - \epsilon_{j+1} - \frac{\epsilon_{j+1}^2}{2}\frac{f''(x)}{f'(x)} + \epsilon_{j+1}(\epsilon_{j+1} + \epsilon_j)\frac{f''(x)}{2f'(x)} + O(\epsilon^3) \\
&= \frac{f''(x)}{2f'(x)}(\epsilon_j \epsilon_{j+1})
\end{aligned}$$

Making the substitution $\epsilon_{j+1} = C\epsilon_j^r$ and $\epsilon_{j+2} = C\epsilon_{j+1}^r = C^2(\epsilon_j^r)^r = C^2\epsilon_j^{r^2}$, we obtain:

$$C^2\epsilon_j^{r^2} = \frac{f''(x)}{2f'(x)}\left(C\epsilon_j^{r+1}\right)$$

for this recursion relation to hold, we require that:

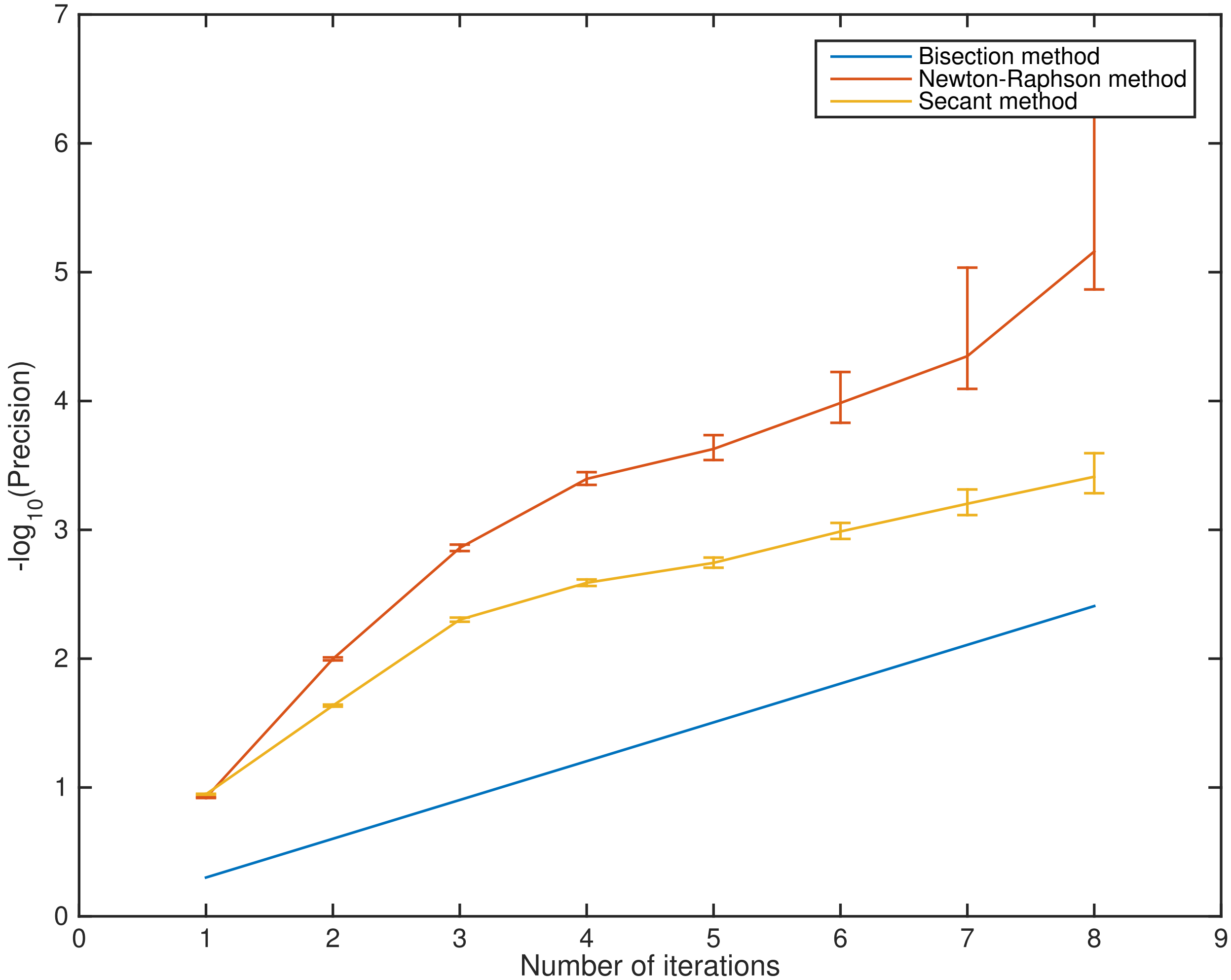
$$C = \frac{f''(x)}{2f'(x)}$$
$$r^2 = r + 1$$

The latter is a quadratic equation with roots at $r = \frac{1}{2} \pm \frac{\sqrt{5}}{2}$. Since we require that the error vanishes as ϵ vanishes, we reject the negative value and obtain that $r = \frac{1}{2} + \frac{\sqrt{5}}{2}$, which is the golden ratio.

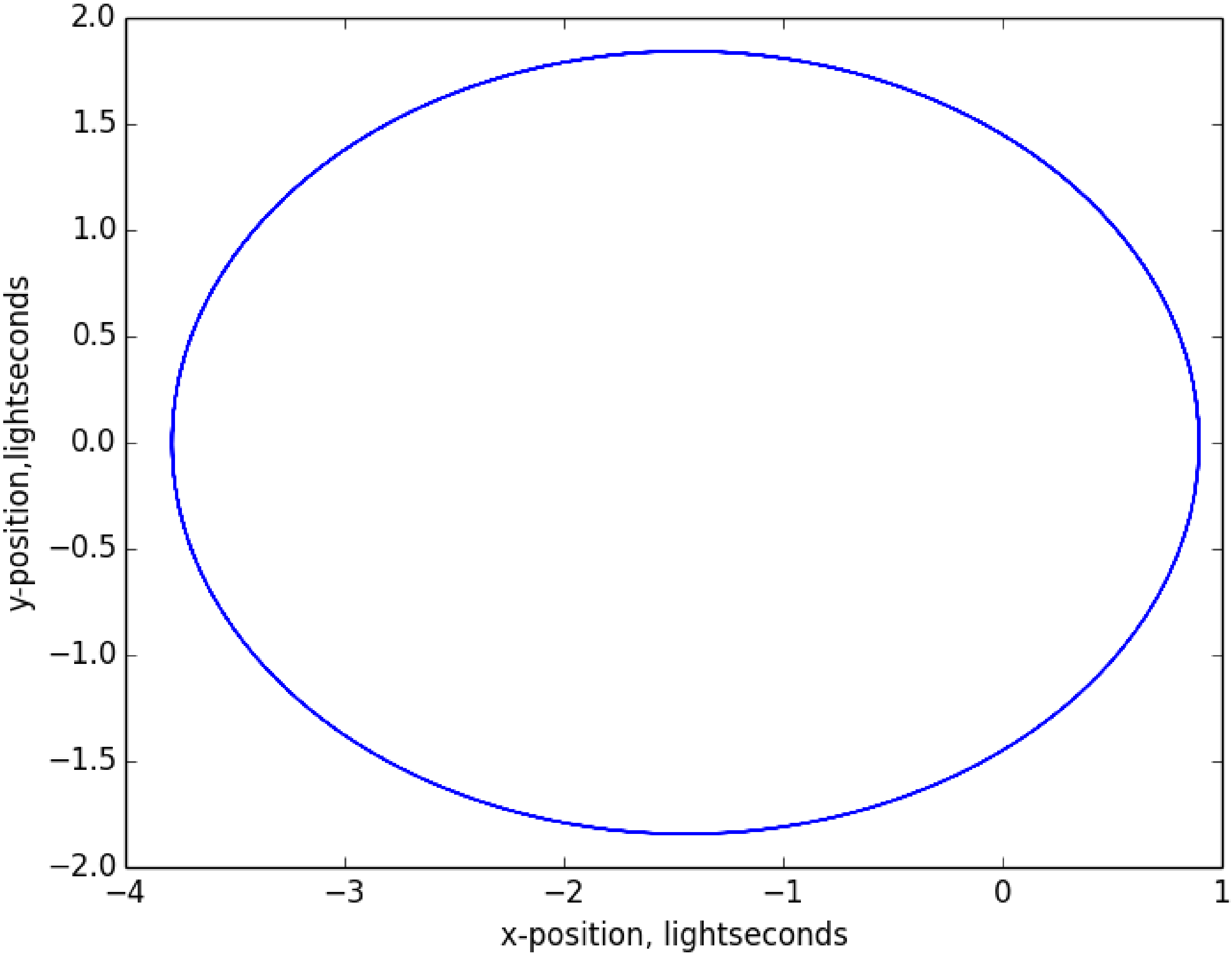
2. The bisection method halves the precision each time (where we define precision to be the length of the interval in which the zero is contained), hence the dependence of the precision on the number of iterations is known exactly. For the Newton-Raphson method, the dependence of the precision on the number of iterations is contingent on the function and the initial guess, hence we examine a number of possible parameters and take the mean (with associated standard errors). In the Newton-Raphson model, we consider the iterated solutions to the system $\sin x - c = 0$ with c varying uniformly from 0 to 1 (10000 samples, including 0 and excluding 1). The guess for all these systems was drawn from a uniform random distribution using the random.random() function, and the precision was measured at each iteration step and normalised to the initial precision before any iterations. The final value is the arithmetic average of all the precisions obtained at each iteration, and the error bars are the standard errors (N-1 convention) of the mean at each iteration. The standard error bars are not symmetrical because the logarithm to base 10 is plotted.

For the secant method, the values of c was also varied uniformly from 0 inclusive to 1 exclusive (10000 samples) and the two initial guesses were drawn using the random.random() function. The precision was measured using the analogue of the Newton-Raphson method, $p = f(x^{(j)}) \frac{x^{(j)} - x^{(j-1)}}{f(x^{(j)}) - f(x^{(j-1)})}$, and was evaluated at each iteration step.

The logarithm to base 10 of the precision of each method, which represents the number of correct digits obtained at each iteration, was plotted against the iteration number, and is shown below.

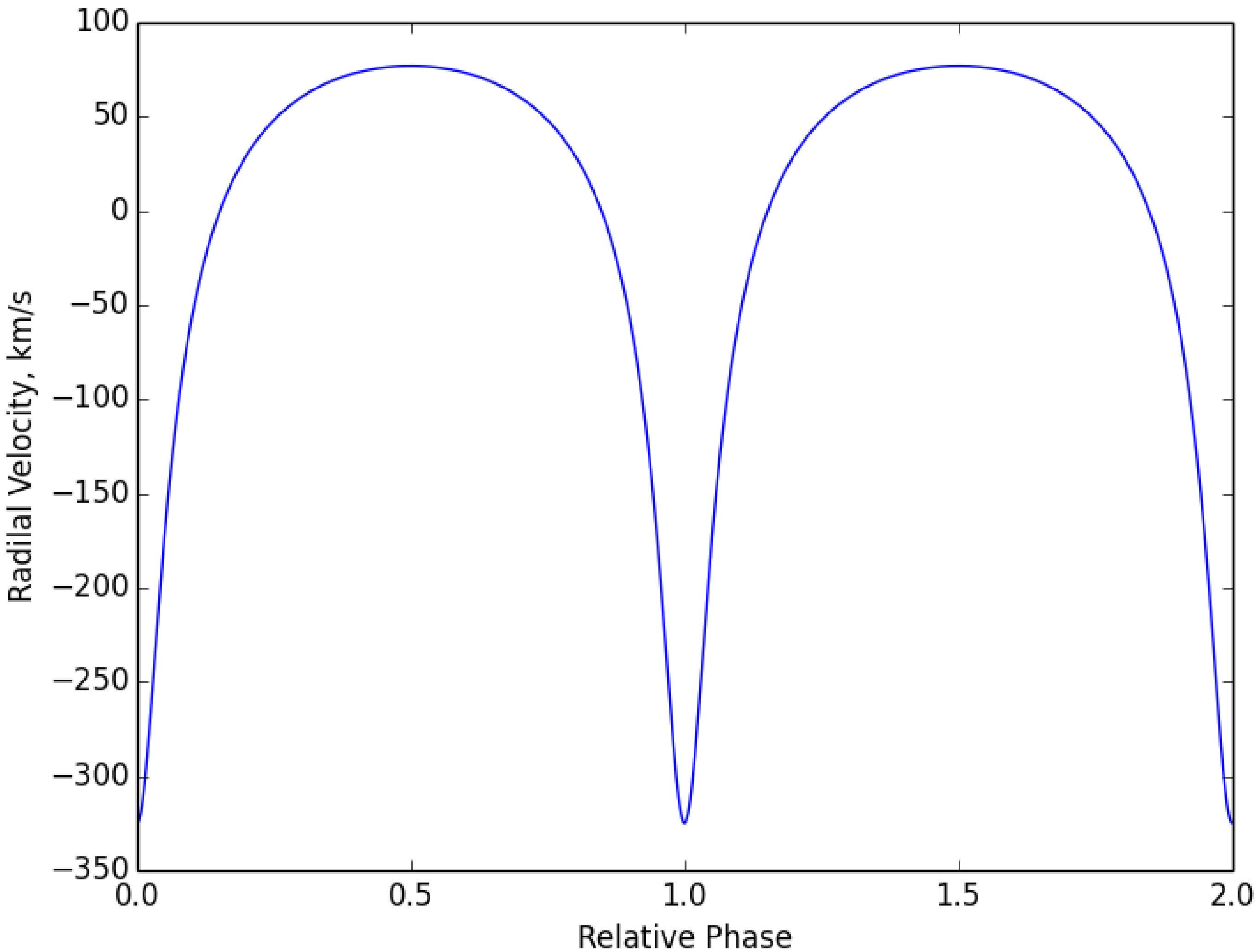


3. The orbit is plotted below:



4. Command: radvelplot(-pi/2,0,2000)

The angle ϕ appears to be $\pi/2$.



Appendix: Question 2 code

```
"""
Root Finder
Author: Soon Wei Daniel Lim
Updated: 19 Dec 2014
"""
from math import *
def bisection(f,a,b,p):
    # Bisection Root Finder
    # f = Function of x, string
    # a = Guess, lower bracket
    # b = Guess, upper bracket
    # p = Precision (desired value of b-a)
    # Output = [a0,b0], bracket containing root
    a = float(a)
    b = float(b)
    fx = lambda x: eval(f)
    p = abs(float(p))
    # Error checks
    print 'f(a)=%f, f(b)=%f' %(fx(a), fx(b))
    if b <= a:
        return 'Error: choose proper bracket ordering.'
```

```
    if fx(a) == 0:
        if fx(b) == 0:
            return '%f and %f are roots' % (a,b)
        else:
            return '%f is a root.' % a
    elif fx(b) == 0:
        return '%f is a root.' % b
    elif fx(b)*fx(a) > 0:
        return 'Error: guesses have same sign'
    N = 0
    plist = []
    while b-a >= p:
        if fx((a+b)/2)*fx(a)>0: #midpoint same sign as f(a)
            a = (a+b)/2
            N = N + 1
            plist = plist + [b-a]
            #print '%d iterations, precision %e' %(N,b-a)
        elif fx((a+b)/2)*fx(b)>0: #midpoint same sign as f(b)
            b = (a+b)/2
            N = N + 1
            plist = plist + [b-a]
            #print '%d iterations, precision %e' %(N,b-a)
        else: #midpoint is a zero
            print type((a+b)/2.0)
            m = (a+b)/2.0
            return '%f is a root.' % m
    print plist
    return 'The root is in the interior of [%f,%f] with %d iterations' %(a,b,N)
def newton(f,df,a,t):
    # Newton-Raphson Method
    # f = Function of x, string
    # df = Derivative of f, string
    # a = Initial guess
    # t = Tolerance
    savefile = open('Newton-Rapshson-Output.csv','a')
    a = float(a)
    a0 = a
    t = abs(float(t))
    fx = lambda x: eval(f)
    dfx = lambda x: eval(df)
    if fx(a) == 0:
        return '%f is a root' % a
    N = 0
    if dfx(a) == 0:
        return 'Error: local extrema nearby'
    p0 = abs(float(fx(a))/float(dfx(a))) #initial precision (to normalize later)
    savefile.write('%s,%s,%f,%d,%f,%e\n'%(f,df,a0,0,a,1))
    while abs(fx(a)) >= t:
        a = a - fx(a)/dfx(a)
        if dfx(a) == 0:
            savefile.close()
            return 'Error: local extrema nearby'
        N = N + 1
        p = abs(fx(a)/dfx(a))/p0
        savefile.write('%s,%s,%f,%d,%f,%e\n'%(f,df,a0,N,a,p))
    p = abs(fx(a)/dfx(a))
    savefile.close()
    return 'The root is near %f with precision %e after %d iterations' %(a,p,N)
def secant(f,a,b,t):
    # Secant method
```

```
    # f = Function of x, string
    # a = Initial guess 1
    # b = Initial guess 2
    # t = Tolerance
    a = float(a)
    b = float(b)
    a0 = a
    b0 = b
    t = abs(float(t))
    t0 = t
    fx = lambda x: eval(f)
    if fx(a) == 0:
        if fx(b) == 0:
            return '%f and %f are roots' %(a,b)
        else:
            return '%f is a root.' %a
    if fx(b) == 0:
        return '%f is a root.' %b
    if (a == b) | (fx(a)==fx(b)):
        return 'Error: Equal guess or value at guess'
    df = (b - a)/(fx(b) - fx(a))
    p0 = abs(df*fx(b)) #initial precision for normalization
    p = p0
    N = 1
    savefile = open('Secant-Output.csv','a')
    c = b - fx(b)*df
    savefile.write('%s,%f,%f,%e,%d,%f,%e\n' %(f,a0,b0,t0,N,c,1))
    while (abs(fx(c)) >= t) & (b != c) & (fx(b) != fx(c)):
        N = N + 1
        a = b
        b = c
        df = (b - a)/(fx(b) - fx(a))
        c = b - fx(b)*df
        p = abs(df*fx(b))
        savefile.write('%s,%f,%f,%e,%d,%f,%e\n' %(f,a0,b0,t0,N,c,p/p0))
    savefile.close()
    return 'The root is near %f with precision %e after %d iterations' %(c,p,N)
def newtonsimple(f,df,a,t):
    # Newton-Raphson Method
    # f = Function of x, string
    # df = Derivative of f, string
    # a = Initial guess
    # t = Tolerance
    a = float(a)
    t = abs(float(t))
    fx = lambda x: eval(f)
    dfx = lambda x: eval(df)
    if fx(a) == 0:
        return a # a is a root
    if dfx(a) == 0:
        return 'Error: local extrema nearby'
    while abs(fx(a)) >= t:
        a = a - fx(a)/dfx(a)
        if dfx(a) == 0:
            return 'Error: local extrema nearby'
    return a
```

Appendix: Question 3 code

```
from math import *
from RootFinder import *
from numpy import *
from pylab import *
T = 27906.98161 #seconds
ecc = 0.617139
a = 2.34186 #lightseconds
zetalist = []
N = 1000 # number of steps
for i in range(N):
    zeta = newtonsimple('(%f/(2*pi))*(x-%f*sin(x))-%f' %(T,ecc,100.0*i),'(%f/(2*pi))*(1-%f*cos(x
    zetalist = zetalist + [zeta]
    #print i
#tlist = [100.0*i for i in range(N)]
xlist = [a*(cos(zetalist[i])-ecc) for i in range(N)]
ylist = [a*sqrt(1-ecc**2)*sin(zetalist[i]) for i in range(N)]
plot(xlist,ylist)
xlabel('x-position, lightseconds')
ylabel('y-position,lightseconds')
show()
```

Appendix: Question 4 code

```
from math import *
from RootFinder import *
from numpy import *
from pylab import *
def radvelplot(phi,phase,N):
    # Plots the radial velocity as a function of fraction of orbit period.
    # phi = Angle of line-of-sight to the Earth
    # phase = Initial phase of the binary (radians)
    # N = Number of steps
    T = 27906.98161
    ecc = 0.617139
    a = 2.34186*299792.458 #km/s
    zetalist = []
    N = abs(int(N))
    deltat = 2.0*T/N #two cycles modelled
    for i in range(N):
        zeta = newtonsimple('(%f/(2*pi))*(x-%f*sin(x))-%f' %(T,ecc,deltat*i+phase*T/(2*pi)),'(%f/
        zetalist = zetalist + [zeta]
    xlist = [a*(cos(zetalist[i])-ecc) for i in range(N)]
    ylist = [a*sqrt(1-ecc**2)*sin(zetalist[i]) for i in range(N)]
    vx = [(xlist[i+1]-xlist[i])/deltat for i in range(N-1)]
    vy = [(ylist[i+1]-ylist[i])/deltat for i in range(N-1)]
    tlist = [(2.0/N)*i for i in range(N-1)]
    radvel = [vx[i]*cos(phi)+vy[i]*sin(phi) for i in range(N-1)]
    plot(tlist,radvel)
    xlabel('Relative Phase')
    ylabel('Radilal Velocity, km/s')
    show()
```