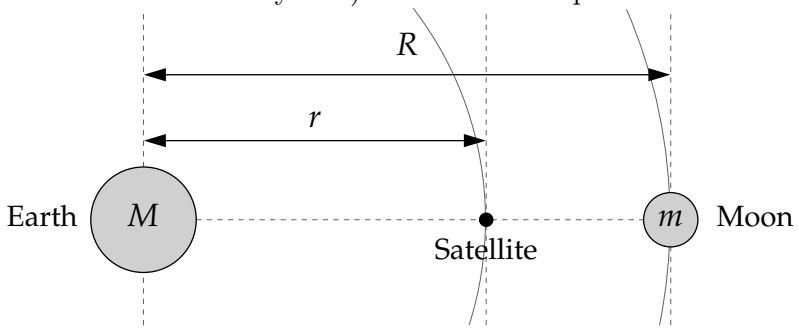
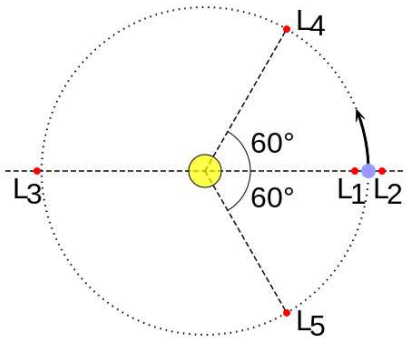
UNIVERSITY OF SUSSEX
Scientific Computing
Tutor: Dr. Ilian Iliev, Office: Pev III 4C5

Assignment 2
Deadline: 12pm on Thursday, November 20th, 2014.
Penalties will be imposed for submissions beyond this date.
Final submission date: Friday, November 21nd, 2014
No submissions will be accepted beyond this date.

1. **The Lagrange point:** There is a magical point between the Earth and the Moon, called the L_1 Lagrange point, at which a satellite will orbit the Earth in perfect synchrony with the Moon, staying always in between the two. This works because the inward pull of the Earth and the outward pull of the Moon combine to create exactly the needed centripetal force that keeps the satellite in its orbit (this could also be done for the Earth-Sun system). Here's the setup:



There are also 4 more Lagrange points - L_2 to L_5, similarly quite useful for parking satellites into orbit. E.g. the Planck and Hershel satellites were send to L_2 for Earth-Sun, in order to always be on Earth's shadow, see figure.



(a) Assuming circular orbits, and assuming that the Earth is much more massive than either the Moon or the satellite, show that the distance r from the center of the Earth to the L_1 point satisfies

$$\frac{GM}{r^2} - \frac{Gm}{(R-r)^2} = \omega^2 r,$$

where M and m are the Earth and Moon masses, G is Newton's gravitational constant, and ω is the angular velocity of both the Moon and the satellite.

(b) The equation above is a fifth-order polynomial equation in r (also called a quintic equation). Such equations cannot be solved exactly in closed form, but it's straightforward to solve them numerically. Write a program that uses either Newton's method, the Ridder method or the internal `fsolve` routine to solve for the distance r from the Earth to the L_1 point. Compute a solution accurate to at least four significant figures.

The values of the various parameters are:

$$\begin{aligned} G &= 6.674 \times 10^{-11}\,\mathrm{m^3 kg^{-1} s^{-2}}, \\ M &= 5.974 \times 10^{24}\,\mathrm{kg}, \\ m &= 7.348 \times 10^{22}\,\mathrm{kg}, \\ R &= 3.844 \times 10^{8}\,\mathrm{m}, \\ \omega &= 2.662 \times 10^{-6}\,\mathrm{s^{-1}}. \end{aligned}$$

You will also need to choose a suitable starting value for r, or two starting values if you use the secant method. [30]

Solution: (a) This is done by simply writing the force equation for the satellite - the difference between the Earth and Moon gravities should balance the centripetal force $\omega^2 r$. [10]

(b) We first import the needed modules and set the values of the constants:

```
import numpy as np
import pylab as pl
import scipy as sci
from scipy import optimize
from newtonRaphson2 import *
from ridder import *

G=6.674e-11
M=5.974e24
msat=7.348e22
Rad=3.844e8
omega=2.662e-6
```

Next, we define the function which determines the equation to be solved ($f(x) = 0$), by moving all terms to one side:

```
def f(r):
    return G*M/r**2-G*msat/(Rad-r)**2-omega**2*r
```

with initial guess given by (determined by e.g. plotting the function, see Figure).

```
r0=np.array([3e8])
```

Note that it is important for the initial guess to be smaller than R (and of course larger than zero), otherwise we might find the other Lagrange points, instead (or find no solution at all). We set up the initial guess as a vector to avoid calculating the derivative of $f(x)$ for the Newton-Raphson method.

We then call the Newton-Raphson and `fsolve` solvers directly:

```
a=newtonRaphson2(f,r0)
```

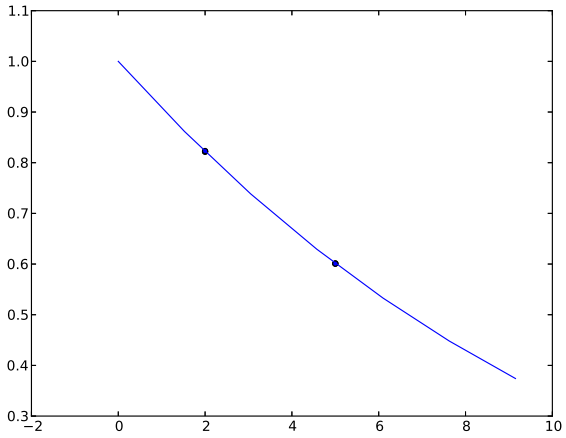


Figure 1: Plot of the function in Problem 2.

```
print 'Newton-Raphson solution is: r=',a[0]

b=sci.optimize.fsolve(f,r0)

print 'fsolve solution is: r=',b[0]
```

which gives:

```
Newton-Raphson solution is: r= 326045071.804
fsolve solution is: r= 326045071.665
```

Setting up Ridder's method requires bracketing the root between two values, which could be 10^8 cm and 4×10^8 cm:

```
c=ridder(f,1e8,4e8,1e-4)

print 'Ridder solution is: r=',c
```

which yields:

```
Ridder solution is: r= 326045071.658
```

[10]

Clearly, all three methods give consistently the same solution, which with 4 significant digits is $r = 3.260 \times 10^8$ cm. This value makes sense physically, since it is clearly between the Earth and the Moon and closer to the (much smaller) Moon, as it should be in order for the two gravity forces to balance out. [10]

2. (a) Given the values of $f(x)$ at the points x, $x - h_1$, and $x + h_2$, where $h_1 \neq h_2$, derive the most accurate possible finite difference approximations for $f'(x)$ and $f''(x)$. What is the order of the truncation error in each case? Assume that h_1 and h_2 are of the same order.

   (b) Estimate $f'(1)$ and $f''(1)$ from the following data:

x	0.97	1.00	1.05
$f(x)$	0.85040	0.84147	0.82612

[20]

Solution: In order to evaluate the required derivative, we start from the Taylor expansions at the two points given:

$$f(x - h_1) = f(x) - h_1 f'(x) + \frac{h_1^2}{2} f''(x) + O(h^3)$$

$$f(x + h_2) = f(x) + h_2 f'(x) + \frac{h_2^2}{2} f''(x) + O(h^3)$$

We then use the above equations to eliminate the $f''(x)$ terms by multiplying the first equation by h_2^2 and the second by h_1^2 and subtract them, finding:

$$-h_2^2 f(x-h_1)+h_1^2 f(x+h_2) = (h_1^2-h_2^2)f(x)+h_1 h_2(h_1+h_2)f'(x)+O(h^3)$$

Then we solve for $f'(x)$, finding the required estimate:

$$f'(x) = \frac{-h_2^2 f(x - h_1) + h_1^2 f(x + h_2) - (h_1^2 - h_2^2) f(x)}{h_1 h_2 (h_1 + h_2)} + \frac{O(h^3) h_2^2 + O(h^3) h_1^2}{h_1 h_2 (h_1 + h_2)}$$

As a quick check, when $h_1 = h_2$ this reduces to the standard central finite-difference approximation. Using that h_1 and h_2 are of the same order (though not equal to each other), we see that the last term is

of order $O(h^2)$, i.e. it is of the same (second) order as the standard central finite-difference approximation.

In order to derive the approximation for the second derivative we start from the same two Taylor expansions as above, but now we eliminate the $f'(x)$ terms, instead. This is achieved by multiplying the first equation by h_2 and the second by h_1 and adding them, which gives:

$$h_1 f(x+h_2)+h_2 f(x-h_1) = (h_1+h_2)f(x)+\frac{h_1 h_2(h_1+h_2)}{2}f''(x)+O(h^2)$$

Solving for $f''(x)$ gives us the desired expression:

$$f''(x) = 2\frac{h_2 f(x-h_1) - (h_1+h_2)f(x) + h_1 f(x+h_2)}{h_1 h_2(h_1+h_2)} + O(h)$$

As a quick check, when $h_1 = h_2$ this reduces to the standard central finite-difference approximation for the second derivative. Again, above we used that h_1 and h_2 are of the same order (though not equal to each other), to derive that the last term is of order $O(h)$, i.e. it is of the same (first) order as the standard central finite-difference approximation. [13]

(b) We can use the expressions we derived in (a) to evaluate the required derivatives, either by hand or using Python, as follows:

```
import numpy as np
import math

x=np.array([0.97,1.00,1.05])
f=np.array([0.85040,0.84147,0.82612])

h1=x[1]-x[0]
h2=x[2]-x[1]

dfdx=(h1**2*f[2]-(h1**2-h2**2)*f[1]-h2**2*f[0])/(h1*h2*(h1+h2))

d2fdx2=2.0*(h1*f[2]+(h1+h2)*f[1]+h2*f[0])/(h1*h2*(h1+h2))

print dfdx,d2fdx2
```

This gives:

```
-0.301166666667 -0.233333333333
```

i.e. the first derivative is negative and small, as expected for a slowly decreasing function as the one given. The second derivative is also small and negative, indicating that the function is curving downwards (negative curvature). [7]

3. (a) Derive central difference approximations for $f'(x)$ accurate to $O(h^4)$ by applying the Richardson extrapolation, starting with the central difference approximation of $O(h^2)$ (given in the lecture notes).

   (b) Use the approximations in (a) to estimate the derivative of $f(x) = tanh(x)$ at $x = 1$ using $h = 0.5$ and 0.1. What are the errors for each of the two approximations with respect to the exact answer? Do the errors behave as expected? Why do you think that is? [20]

Solution: [(a)] We start from the central finite-difference approximation to $f'(x)$ given in the lecture notes:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

We apply Richardson's extrapolation formula for $h_2 = h$ and $h_1 = 2h$, finding

$$f''(x) = \frac{2^2 g(h) - g(2h)}{2^2 - 1},$$

where $g(h)$ is given by the second-order approximation above.

Expanded, this gives

$$f'(x) = \frac{1}{3}\left[4\frac{f(x+h) - 2f(x-h)}{2h} - \frac{f(x+2h) - f(x-2h)}{4h}\right] + O(h^4)$$

or

$$f'(x) = \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h} + O(h^4)$$

Which is a fourth-order central finite-difference approximation to the first derivative. [10]

(b) The numerical estimate required could be done either by hand, or using Python, e.g. as follows:

```
import numpy as np

def f(x):
```

```
    f=np.tanh(x)
    return f

def df(x):
    df=1-np.tanh(x)**2
    return df

h=0.5
x=1.

dfdx2=(f(x+h)-f(x-h))/2./h
dfdx4=(8.*f(x+h)-8.*f(x-h)-f(x+2.*h)+f(x-2.*h))/12./h

error2=df(x)-dfdx2
error4=df(x)-dfdx4

print 'values= ',dfdx2,dfdx4,'exact=',df(x)
print 'errors=',error2,error4
print 'error ratio=',error2/error4
```

i.e. we first set up the function to be differentiated and its derivative (for checking the results), and then directly evaluate the expression derived in (a). Once the approximation is obtained we find the (absolute) errors by comparing to the exact result.

This yields:

```
values=  0.443031096385 0.430036865167 exact= 0.419974341614
errors= -0.0230567547708 -0.0100625235531
error ratio= 2.29134914806
```

for $h = 0.5$ and

```
values=  0.421005757808 0.419992813124 exact= 0.419974341614
errors= -0.001031416194 -1.84715097537e-05
error ratio= 55.8382183023
```

for $h = 0.1$. Clearly, the fourth order finite-difference approximation is much more precise than the second-order one in either case, although the improvement is somewhat less than generally expected,

which should be better a factor of approximately $1/h^2 = 4$ or 100, respectively,while in reality we get 2.3 and 56 times better for $h = 0.5$ and $h = 0.1$. The reson for this is that the derivatives of the `tanh` function change rapidly in that interval, thus a smaller step h is required for accurate results. Nevertheless, it is much better to use the higher order scheme, where we do just a little more work, but obtain much better results. [10]

4. **The Stefan–Boltzmann constant:** The Planck theory of thermal radiation (you will study this in detail later) tells us that in the (angular) frequency interval ω to $\omega + d\omega$, a perfect black body of unit area radiates electromagnetically an amount of thermal energy per second equal to $I(\omega)d\omega$, where

$$I(\omega) = \frac{\hbar}{4\pi^2 c^2} \frac{\omega^3}{(e^{\hbar\omega/k_B T} - 1)} \tag{1}$$

Here $\hbar$ is Planck's constant divided by 2π, c is the speed of light, and k_B is Boltzmann's constant.

   (a) Show that the total energy per unit area radiated by a black body is

$$W = \frac{k_B^4 T^4}{4\pi^2 c^2 \hbar^3} \int_0^\infty \frac{x^3}{e^x - 1} dx \tag{2}$$

   (b) Write a program to evaluate the integral in this expression from zero to infinity using a method of your choice (e.g. internal `scipy.integrate.quad`, or the provided Romberg code). Explain what method you used, and how accurate you think (or know) your answer is.

   (c) Even before Planck gave his theory of thermal radiation around the turn of the 20th century, it was known that the total energy W given off by a black body per unit area per second followed Stefan's law: $W = \sigma T^4$, where σ is the Stefan–Boltzmann constant. Use your value for the integral above to compute a value for the Stefan–Boltzmann constant (in SI units). Check your result against the known value, which you can find in books or on-line. You should get good agreement. What is the relative error? [30]

Solution:

(a) We change the variable to the dimensionless one $x = \hbar\omega/k_BT$ (thus $\omega = xk_BT/\hbar$), from which we directly obtain the desired expression. [7]

(b) First, we import the needed modules and define the function to be integrated:

```
import scipy as sci
from scipy import integrate
import numpy as np

def f(x):
    return x**3/(np.exp(x)-1.0)
```

Then, if we use the internal `quad` routine the calculation of the integral is done by a direct call to it:

```
int1,errh1=sci.integrate.quad(f,0,sci.integrate.Inf)

print int1,errh1
```

Note that one can directly set the limit to infinity, which is done through the `sci.integrate.Inf` function. The result provided by `quad` is the value of the integral and an estimate of the error (done through adaptive time-stepping).

If the provided Romberg code is used one cannot use infinity as an upper limit. Instead, one can use a sufficiently large number as an upper limit, taking care that the integral is converged, independent of that limit. Since the integrant decreases exponentially this is quickly achieved. The lower limit of zero similarly creates probems due to division by zero, so has to be replaced by a very small, but nonzero number. The call in this case could be e.g.:

```
Ir1,n=romberg(f,1e-8,30.,tol=1.e-6)

print Ir1,n
```

The result is:

```
6.49393940227 2.62847002893e-09
6.4939394662 128
```

for the two methods. Clearly, both give the same result within the requested precision. [13]

(c) Then, we define the required constants and calculate the value of the Stefan-Boltzmann constant σ by:

```
kB=1.3806488e-23 #m^2 kg/s^2/K
h=6.62606957e-34 #m^2 kg/s
c=299792458 #m/s
hbar=h/2/np.pi

sigma=kB**4*int1/(4.0*np.pi**2*hbar**3*c**2)
```

and we can compare to the actual value of σ and calculate the error by:

```
sigma_actual=5.670373e-8

print 'sigma= ',sigma,' rel. error =', (sigma-sigma_actual)/sigma_actual
```

which gives:

```
sigma=  5.67037262259e-08  rel. error = -6.65579963218e-08
```

Therefore, the calculated value of the constant was indeed quite precise, with 7 significant digits (machine precison). [10]