Linn Boldt-Christmas

Scientific Computing
Assignment 2

Candidate no. 117418
Due in November 23rd, 2015

University of Sussex

Physics & Astronomy

QUESTION 1

The speed v *of a Saturn V rocket in vertical flight near the surface of the Earth can be approximated by*

$$v = u \ln\left(\frac{M_0}{M_0 - \dot{m}t}\right) - gt$$

where

u = *2510 m/s = velocity of exhaust relative to rocket,*

M_0 = *2.8 x 10 ^{6} kg = mass of rocket at liftoff*

m' = *13.3 x 10 ^{3} kg/s = rate of fuel consumption*

g = *9.81 m/s^2 = acceleration due to gravity*

t = *time measured from liftoff*

(a) Determine the time when the rocket reaches the speed of sound (335 m/s) using the Ridder and Newton methods, as well as the scipy.optimize.fsolve. You will need to choose a suitable initial guess value for t, *e.g. by plotting the function.*

We first want to plot the function so that we can make a guess for where t will approximately lie. First, after importing the necessary modules, we define the constants:

```
import numpy as np
import matplotlib.pyplot as plt
import ridder as ri
import newtonRaphson as nR
from scipy.optimize import fsolve


# part a


# constants:
u=2510 #[m/s], velocity of exhaust relative to the rocket
Mo=2.8e+6 #[kg], mass of rocket at liftoff
dmdt=13.3e+3 #[kg/s], rate of fuel consumption
g=9.81 #[m/s^2], gravitational acceleration
```

After that, we want to define the actual function so that we can tell Python what to plot. Our variable in this is *t*, so we need to define a range of *t* in which *t* will lie. We will accomplish this by using the linspace that we are using from numpy

```
# speed of Saturn V rocket:

t = np.linspace(0,200,500) # our range of t (lower, upper, iterations)
```

We are creating this linspace from zero to 200, going up in graduations of 500. Next, we need to define the actual function. Instead of calling this function *v(t)* as one would intutively, I am calling it *v(p)* since we already have something called *t* and we don't want to confuse Python.

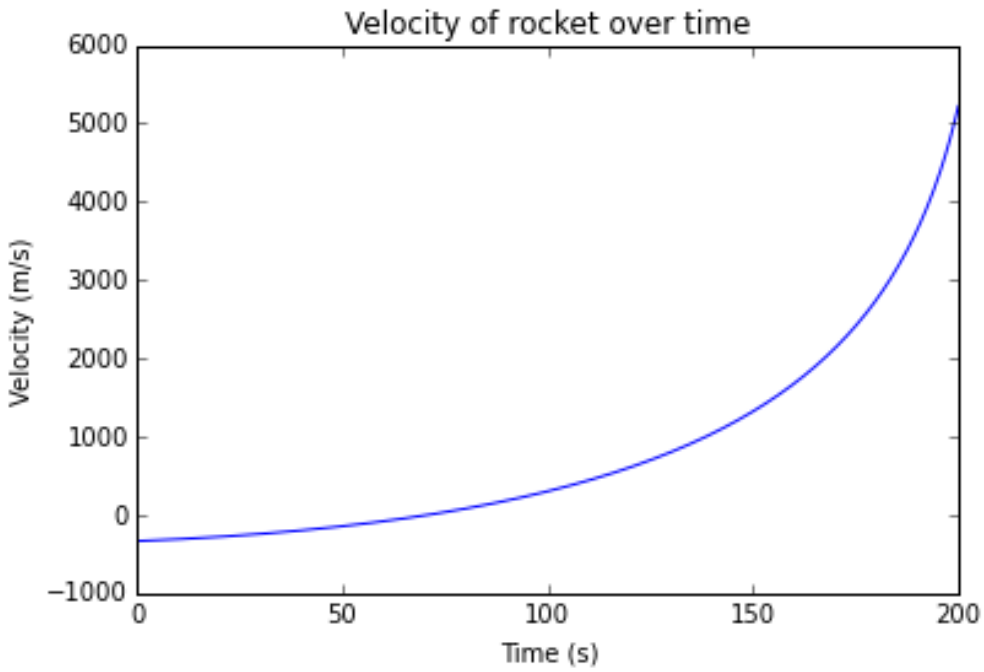
We also want it to cross the x-axis, so in order to do that, we are saying that we want velocity to equal 335 m/s (speed of sound) and moving that term over. Therefore, we end up with a -335 term in our velocity equation.

```
def v(p):

    return (u*(np.log(Mo/(Mo-(dmdt*p)))))-(g*p)-335  # this equation has a -335 term because we want to find out when v=335 so we moved the 335 term over
```

Now, we want to plot our *v(t)* function against our range of *t.* Using the pyplot module imported earlier, we can write:

```
plt.plot(t,v(t))

plt.title('Velocity of rocket over time')

plt.xlabel('Time (s)')

plt.ylabel('Velocity (m/s)')

plt.show()
```

And the resultant program gives us:



This plot indicates that the velocity has a negative velocity at $t = 0$ which is obviously not the case; this is just because we have added our -335 term in order to make the plot have a root. If we didn't have this term, we would not be able to find a root but $v(t = 0) = 0$

Looking at our plot, we can interpolate that velocity will equal 335 m/s somewhere within the 50-100 second interval. Typing in v(n) into the terminal gives us the value of the function at time n, so using the knowledge gained from the plot, I did that and I found that v(60) gave a value below 335 m/s and v(80) gave one above. Therefore, I used 60-80 seconds as my interval.

I differentiated the *v(t)* equation and defined that, as we need it for the Newton method. I then gave the Ridder, Newton, and fsolve solutions to two decimal places.

```
def dvdt(p):

    return (((dmdt*u)/(Mo-(dmdt*p))-g))

risol = ri.ridder(v,60,80)

print "The Ridder method tells us that the rocket will reach
the speed of sound at time =",round(risol,2),"s."

nRsol = nR.newtonRaphson(v,dvdt,60,80,tol=1e-09)

print "The Newton Raphson method tells us that the rocket will
reach the speed of sound at time =",round(nRsol,2),"s."

fs = fsolve(v,0.1)

print "The scipy.optimize.fsolve method tells us that the
rocket will reach the speed of sound at time
=",round(fs,2),"s."
```

Running this code produces:

```
The Ridder method tells us that the rocket will reach the speed of sound at time = 70.88 s.
The Newton Raphson method tells us that the rocket will reach the speed of sound at time = 70.88 s.
The scipy.optimize.fsolve method tells us that the rocket will reach the speed of sound at time = 70.88 s.
```

(b) What are the limits of validity of this equation? (Hint: what happens as time increases?) Discuss.

This equation describes approximately rocket velocity near the surface of the Earth. Once a rocket is higher up, *g* will decrease as 9.81 m/s^2 is *surface* gravity and there will also be a change in *u* as the velocity of the exhaust relative to the rocket changes. There will be less air resistance as the air gets thinner higher up and so the rocket will need less exhaust to continue moving at the same speed. Also, looking at the equation, we have *t* terms as the denominator in ln and as a minus term. ln(1/∞) - ∞ -> - ∞ which does not make sense as a velocity, so, this equation would break down before that.

QUESTION 2

The Fourier law of heat conduction states that the unit time rate of heat transfer through a material is proportional to the negatvie gradient in the temperature. In its simplest form, it can be expressed as:

$$Q_x = -k \frac{dT}{dx}$$

where x *is the distance (in m) along the path of heat flow,* T *is the temperature (in degrees),* k *is the thermal conductivity, and* Q_x *is the heat flux (W/m²).*

Given that

T(x=0) = 15

T(x=0.1) = 10

T(x=0.2) = 5

T(x=0.3) = 3

Compute k *as precisely as you can if* Q_x *at* x = 0 *is 40 W/m². What order approximation did you use?*

This question is very mathematical, and so there is really no code needed. Instead, I have included my derivation in # commentary in my script.

```
import numpy as np


# Fourier law of heat conducion: Q_x = -k * (dT/dx)


# data from table in Q

x = np.array([0.0,0.1,0.2,0.3]) # distance in m along path of
heat flow

T = np.array([15,10,5,5],dtype=float) # temperature in degrees


# Taylor expansion:

# T(x+h) = T(x) + hT'(x) + [h^2 / 2!] T''(x) + [h^3 / 3!]
T'''(x)

# T(x+h) = T(x) + hT'(x) + [h^2 / 2] T''(x) + [h^3 / 6]
T'''(x)
```

```
# T(x+2h) = T(x) + 2hT'(x) + [ (2h)^2 / 2!] T''(x) + [ (2h)^3 / 3! ] T'''(x)

# T(x+2h) = T(x) + 2hT'(x) + [ 4h^2 / 2!] T''(x) + [ 8h^3 / 3! ]T'''(x)

# T(x+2h) = T(x) + 2hT'(x) + [2h^2] T''(x) + [(4/3)h^3]T'''(x)


# T(x+3h) = T(x) + 3hT'(x) + [ (3h)^2 / 2!] T''(x) + [ (3h)^3 / 3! ] T'''(x)

# T(x+3h) = T(x) + 3hT'(x) + [ (9h^2) / 2!] T''(x) + [ (27h^3) / 3! ] T'''(x)

# T(x+3h) = T(x) + 3hT'(x) + [(9/2)h^2] T''(x) + [(9/2)h^3] T'''(x)


# Name the equations:


# A = T(x+h) = T(x) + hT'(x) + [h^2 / 2] T''(x) + [h^3 / 6] T'''(x)

# B = T(x+2h) = T(x) + 2hT'(x) + [2h^2] T''(x) + [(4/3)h^3]T'''(x)

# C = T(x+3h) = T(x) + 3hT'(x) + [(9/2)h^2] T''(x) + [(9/2)h^3] T'''(x)


# Rearrange A for T'''(x):

# T(x+h) - T(x) - hT'(x) - [h^2 / 2] T''(x) = [h^3 / 6] T'''(x)

# T'''(x) = [6 / h^3] * [T(x+h) - T(x) - hT'(x) - [h^2 / 2] T''(x)]


# Rearrange B for T''(x)

# T(x+2h) = T(x) + 2hT'(x) + [2h^2] T''(x) + [(4/3)h^3]T'''(x)

# [2h^2] T''(x) = T(x+2h) - T(x) - 2hT'(x) - [(4/3)h^3]T'''(x)

# T''(x) = [1/(2h^2)] * [ T(x+2h) - T(x) - 2hT'(x) - [(4/3)h^3]T'''(x)


# Substitute T'''(x) into our T''(x) equation
```

```
# T''(x) = [1/(2h^2)] * [ T(x+2h) - T(x) - 2hT'(x) -
[(4/3)h^3] * [6 / h^3] * [T(x+h) - T(x) - hT'(x) - [h^2 / 2]
T''(x)]

# T''(x) = [1/(2h^2)] * [ T(x+2h) - T(x) - 2hT'(x) - 8 *
[T(x+h) - T(x) - hT'(x) - [h^2 / 2] T''(x)]

# T''(x) = [1/(2h^2)] * [ T(x+2h) - T(x) - 2hT'(x) - 8*T(x+h)
- 8*T(x) - 8*hT'(x) - (4*h^2) T''(x)]

# T''(x) = [1/(2h^2)] * T(x+2h) - [1/(2h^2)] * T(x) - [1/h] *
T'(x) - [4/h^2] * T(x+h) -[4/h^2] * T(x) - [4/h] * T'(x) - 2
T''(x)


#Plug in values to rearrange

# T''(x) + 2 T''(x) = [1/(2*(0.1^2))] * [5] - [1/(2*(0.1^2)] *
T(x) - [1/0.1] * T'(x) - [4/(0.1^2)] * [10] -[4/(0.1^2)] *
T(x) - [4/0.1] * T'(x)

# T''(x) = (1/3) * [ (250) - (50)*T(x) - 10*T'(x) - (4000) -
400*T(x) - 40*T'(x)]

# T''(x) = (1/3) * [ (250-4000) - (50-400)*T(x) - (10-
40)*T'(x) ]

# T''(x) = (1/3) * [ (-3750) - (350)*T(x) - (30)*T'(x) ]

# T''(x) =  -(350/3)*T(x) - 10*T'(x) - 1250

# T''(x) =  -(350/3)*(15)- 10*T'(x) - 1250

# T''(x) =  -1750 - 10*T'(x) - 1250

# T''(x) =  -3000 - 10*T'(x)


# Rearrange C for T''(x) and plug the above into it

# [(9/2)h^2] T''(x) = T(x+3h) - T(x) - 3hT'(x) - [(9/2)h^3]
T'''(x)

# [(9/2)h^2] * [ -3000 - 10*T'(x) ] = T(x+3h) - T(x) - 3hT'(x)
- [(9/2)h^3] T'''(x)


# Plug values in and rearrange for T'(x)

# [(9/2)*(0.1^2)] * [ -3000 - 10*T'(x) ] = (3) - (15) -
(3*0.1)T'(x) - [(9/2)(0.1^3)] T'''(x)
```

[0.045] * [-3000 - 10*T'(x)] = (3) - (15) - (0.3)*T'(x) - [0.0045]* T'''(x)

- 135 - 0.45*T'(x) = (-12) - (0.3)*T'(x) - [0.0045]* T'''(x)

- 0.45*T'(x) = (-12+135) - (0.3)*T'(x) - [0.0045]* T'''(x)

- 0.45*T'(x) = (123) - (0.3)*T'(x) - [0.0045]* T'''(x)

T'(x) = (123/0.45) - (0.3/0.45)*T'(x) - [0.01]* T'''(x)

Plug in T'''(x)

T'(x) = (123/0.45) - (0.3/0.45)*T'(x) - [0.01]*([6 / h^3] * [T(x+h) - T(x) - hT'(x) - [h^2 / 2] T''(x)])

T'(x) = (123/0.45) - (0.3/0.45)*T'(x) - [0.01]*([6 / (0.1^3)] * [10 - 15 - (0.1)*T'(x) - [(0.1^2) / 2] * T''(x)])

T'(x) = (123/0.45) - (0.3/0.45)*T'(x) - [0.01]*([6000] * [- 5 - (0.1)*T'(x) - [0.005] * T''(x)])

T'(x) = (123/0.45) - (0.3/0.45)*T'(x) - [60]* [- 5 - (0.1)*T'(x) - [0.005] * T''(x)]

T'(x) = (123/0.45) - (0.3/0.45)*T'(x) - [- 300 - (6)*T'(x) - [0.3] * T''(x)]

T'(x) = (123/0.45) - (0.3/0.45)*T'(x) + 300 + (6)*T'(x) + [0.3] * T''(x)]

T'(x) = [(123/0.45)+300] - [(0.3/0.45)+6]*T'(x) + [0.3] * T''(x)]

T'(x) = (123/0.45) - (0.3/0.45)*T'(x) + 300 + (6)*T'(x) + [0.3] * T''(x)]

T'(x) = [(123/0.45)+300] - [(0.3/0.45)+6]*T'(x) + [0.3] * T''(x)

Plug in T''(x)

T'(x) = [(123/0.45)+300] - [(0.3/0.45)+6]*T'(x) + [0.3] * [-3000 - 10*T'(x)

T'(x) = [(123/0.45)+300] - [(0.3/0.45)+6]*T'(x) - 900 - 3*T'(x)]

```
# T'(x) +[(0.3/0.45)+6]*T'(x) + 3*T'(x)]   = [(123/0.45)+300] -  900

# [1 + [(0.3/0.45)+6] + 3] * T'(x) = [(123/0.45)+300] -  900

# T'(x) = ( [(123/0.45)+300] -  900 ) / [1 + [(0.3/0.45)+6] + 3]


# T'(x) = - 30.625


# So if Q_x = 40 and Q_x = -k*(T'(x))

# Then - k = (Q_x)/(T'(x))


Qx = 40

dTdx = -30.625

k = -(Qx/dTdx)

print "The thermal constant k is equal to",round(k,3)
```

This gives us the thermal constant k to 3 decimal places:

```
The thermal constant k is equal to 1.306
```

QUESTION 3

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

This part is again very mathematical, and so again, I have included my derivation in # commentary in my script.

```
import numpy as np


# part (a)


# from lecture notes (week 6)

# G = g(h1)+E(h1) = g(h1)+ch1^p

# G = g(h2)+E(h2) = g(h2)+ch2^p
```

```
# >> Eliminate c and solve for G

# G = [ (h1/h2)^p * g(h2) - g(h1) ] / [ (h1/h2)^p - 1 ]


# >> Assume h2 = h1/2

# G = [ 2^p * g(h1/2) - g(h1) ] / [ 2^p - 1 ]


# >> Use Taylor expansion

# f(x+h) = f(x) + hf'(x) + (h^2/2!)f''(x) + (h^3/3!)f'''(x) +
(h^4/4!)fiv(x) + (h^5/5!)fv(x) + ...

# f(x-h) = f(x) - hf'(x) + (h^2/2!)f''(x) - (h^3/3!)f'''(x) +
(h^4/4!)fiv(x) - (h^5/5!)fv(x) + ...


# >> Sum and subtract the two expansions

# f(x+h)+f(x-h) = 2f(x) + (h^2)f''(x) + (h^4/12)fiv(x)

# f(x+h)-f(x-h) = 2hf'(x) + (h^3/3)f'''(x) + (h^5/60)fv(x)


# >> Rearrange for f'(x). The higher orders can be considered
errors and not part of the actual solution.

# Rearranging is only possible using f(x+h)-f(x-h) because
f'(x) is cancelled out in f(x+h)+f(x-h)

# f(x+h)-f(x-h) - (h^3/3)f'''(x) - (h^5/60)fv(x) = 2hf'(x)

# [ f(x+h)-f(x-h) ] / 2h = centered approximation

# [ (h^3/3)f'''(x) - (h^5/60)fv(x) ] / 2h = error associated
with centered approximation

# Refer to equation G(h)=g(h)+E(h) where E(h) is the error

# [ f(x+h)-f(x-h) ] / 2h = g(h)

# [ f(x+{h/2})-f(x-{h/2}) ] / h = g(h/2)


# Substitute g(h) and g(h/2) into our equation for G

# G = [ 2^p * ( [ f(x+{h1/2})-f(x-{h1/2}) ] / h ) - ([ f(x+h)-
f(x-h) ] / 2h ]) ] / [ 2^p - 1 ]
```

(b) Use the approximations in (a) to estimate the derivative of $f(x) = x + e^x$ at $x = 1$ using $h = 0.5$. What are the error for each of the two approximations? Do they behave as expected?

In this part, we have to define x at 1 and h at 0.5. We also define the order of the error, p.

```
# part (b)

x=1.0

h=0.5

p=2.0 # this is the order of the error
```

Next, we have to define $f(x) = x + e^x$ since that is the function that we are dealing with in this question. We also have to define the differential of this function since the derivative is what we are estimating.

```
def f(x): # equation given in Q

    return x+(np.exp(x))

def dfdx(p, x, h): # central difference estimation of a
differential

    return (p(x+h/2)-p(x-h/2))/h

g1 = dfdx(f, 1.0, 0.5) # plugging in constants given

g2 = dfdx(f, 1.0, 0.25)

# say h = h1 --> h2 = h1/2, then:

G = (((2**p)*g2)-g1)/((2**p)-1)
```

We also want to obtain the errors on the approximations in order to compare them. For that, we need to find out how far the estimations g1 and G are from the derivative.

```
# the error would be the difference between the derivative of
f and the estimations g1 and G

def fi(x): # derivative of f(x), so, f'(x)

    return 1+(np.exp(x))

trueValue=fi(1)

err1=G-trueValue

err2=g1-trueValue
```

In order to get the results of these produced in a nice readable manner, I created a print statement that includes the results with an appropriate number of significant figures.

```
print "The Richardson extrapolation equals",round(G,3),"when x is equal to ",round(x,4)," and h is equal to",h,"with an error of",round(err1,8)

print "The central difference extrapolation equals",round(g1,3),"when x is equal to ",x," and h is equal to",h,"with an error of",round(err2,3)
```

And in order to comment on whether or not these approximations behave as expected with one another, we add a third statement:

```
print "The central difference and Richardson techniques give values that are within each others' error ranges, which is expected."
```

Running this code, we get:

```
The Richardson extrapolation equals 3.718 when x is equal to  1.0  and h is equal to 0.5 with an error of -2.216e-05
The central difference extrapolation equals 3.747 when x is equal to  1.0  and h is equal to 0.5 with an error of 0.028
The central difference and Richardson techniques give values that are within each others' error ranges, which is expected.
```

QUESTION 4

A commonly occuring function in physics calculations, with applications in e.g. quantum physics, astrophysics and fluid dynamics, is the gamma function Γ(a), *which is defined by the integral*

$$\Gamma(a) = \int_0^\infty x^{a-1} e^{-x}\, dx$$

There is no closed-form expression for the gamma function for a general value of a, *but one can calculate its value for given* a *by performing the integral above numerically. You have to be careful how you do it, however, if you wish to get an accurate answer.*

(a) *Write a program to make a graph of the value of the integrand* $x^{(a-1)}e^{-x}$ *as a function of* x *from* x = 0 *to* x = 5, *with three separate curves for* a = 2, 3, *and* 4, *all on the same axes. You should find that the integrand starts at zero, rises to a maximum, and then decays again for each curve.*

In order to plot the three different integrand curves (where a = 2, 3, 4), we need to define the three curves. We also need to give Python a range of x-values to plot these curves against.

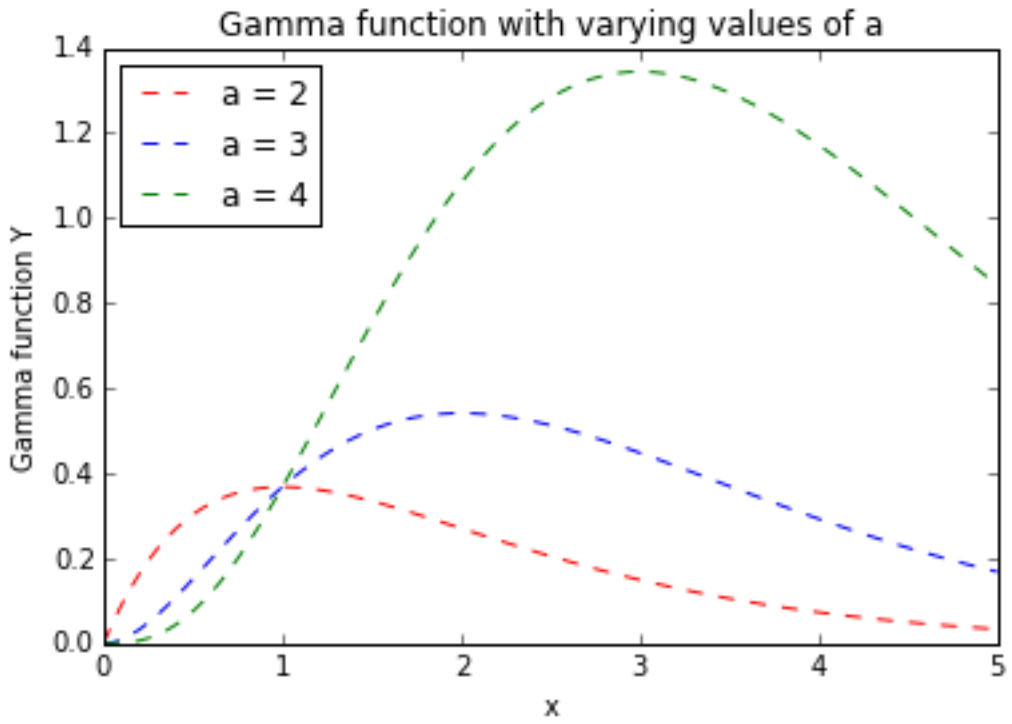
```
import numpy as np

import matplotlib.pyplot as plt

from scipy.integrate import quad

from scipy.integrate import romberg
```

```
# part a


x = np.linspace(0,5,100)


Y1 = (x**(1))*(np.exp(-x)) # a = 2
Y2 = (x**(2))*(np.exp(-x)) # a = 3
Y3 = (x**(3))*(np.exp(-x)) # a = 4
```

Then we want to put this information onto a graph, so we use matplotlib.pyplot

```
plt.plot(x,Y1,'r--',label='a = 2') # first line where a = 2 is
red dashes
plt.plot(x,Y2,'b--',label='a = 3') # second line where a = 3
is blue dashes
plt.plot(x,Y3,'g--',label='a = 4') # third line where a = 4 is
green dashes
plt.legend(loc=0) # adding a legend
plt.title('Gamma function with varying values of a')
plt.xlabel('x')
plt.ylabel('Gamma function Y')
plt.show()
```

Running this code, we get this graph:



(b) Show analytically that the maximum falls at x = a-1

This part required us to differentiate the integrand given to us in the main section of the question that we have just plotted. After obtaining the differential, when plugging in *x = a -1,* it is possible to see how the differential equals zero. A differential is essentially showing the rate of change of gradient, so, when the change in gradient is zero, this must be a stationary point.

```
# part b
# gamma function integrand (denoted as y for this derivation):


# y = x^(a-1) * e^(-x) where a = real constant
# y' = dy/dx = d/dx (u*v)
# when u = x^(a-1) and u' = (a-1)*x^(a-2)
# and v = e^(-x) and v' = -e^(-x)


# d/dx (u*v) = u'v+uv'


# u'v+uv' = [e^(-x)]*[(a-1)*x^(a-2)]+[x^(a-1)]*[-e^(-x)]
```

```
# = [e^(-x)]*[(a-1)*x^(a-2)]-[x^(a-1)]*[e^(-x)]
# = [e^(-x)]*[(a-1)*x^(a-2)-x^(a-1)]

# plug in x = a-1

# u'v+uv' = [e^(1-a)]*[(a-1)*(a-1)^(a-2)-(a-1)^(a-1)]
# u'v+uv' = [e^(1-a)]*[(a-1)^(a-1)-(a-1)^(a-1)] = 0

# derivative of a f(x) equals zero when x = stationary point
# so, x = a-1 gives stationary point (maximum)
```

(c) Calculate Γ(a) *for* a = 3/2 *using both* sci.integrate.quad *and Romberg integration. The value of* Γ(3/2) *is known to be equal to* √π / 2 ≈ 0.886.

For this section, we needed to define the Gamma function using the def command, and plugging in 3/2 for *a* as instructed by the question. Since the function gives x to the power of a-1, we are now putting the x to the power of ½ since 3/2 – 1 = ½.

```
def Y(x): # defining gamma equations integrand

    return (x**(0.5))*(np.exp(-x)) #  here, a = 3/2
```

Now we want to produce the integrations that give us values for the integrand of our function that we defined as Y(x). For this, we are using the sci.integrate.quad and Romberg functions as instructed with the two modules we imported earlier, i.e.

```
from scipy.integrate import quad
from scipy.integrate import romberg
```

We want to produce an easy to read result when we run the code, so we incorporate this calculation within a printed message explaining what we are doing.

```
print "The value of the Gamma function with its error
is",quad(Y, 0, np.inf),"at a = 3/2 when we use the built in
sci.integrate.quad module. This value is supposed to be around
1/2*(sqrt(pi) which is ~0.886 so this value is good."

print "The value of the Gamma function
is",romberg(Y,0,1e+1),"at a = 3/2 when we use the Romberg
method. This value is also around the accepted value of ~0.886
so this value is also good."
```

The upper limit of the Romberg method cannot be infinity because the module does not allow for it. When we run the code, we get this:

```
The value of the Gamma function with its error is (0.8862269254536111,
9.077554263825505e-10) at a = 3/2 when we use the built in sci.integrate.quad module. This
value is supposed to be around 1/2*(sqrt(pi) which is ~0.886 so this value is good.
The value of the Gamma function  is 0.886010248691 at a = 3/2 when we use the Romberg
method. This value is also around the accepted value of ~0.886 so this value is also good.
```