Scientific Computing Examination: Friday 9th January 2015

Writeup

Candidate 117458

Note: this document does not include verbatim all of the code used to solve the problems: what is quoted is only a set of extracts.

Question 1 – Root solving

A. Write a program to solve this equation for x using the relaxation method for the case $c = 2$. Calculate to an accuracy of at least 10^{-6}.

Because $x = 1 - e^{-cx}$, to find x, we can solve for x by finding the roots of the equation – i.e. where $1 - e^{-cx} - x = 0$.

After importing the libraries required, the next step is to define the equation and the constants in use. Because we want an accuracy of at least 10^{-6}, the argument `xtol=1e-7` is used. The initial guess in use 1; this is far enough away from 0 that fsolve will not converge on the solution where $x = 0$.

```
# Constants
c = 2.

# Equation
def f(x):
      eqn = (1-np.exp((-c*x))-x)
      return eqn
sol = optimize.fsolve(f,1.,xtol=1e-7) # 1 is an initial guess

print "Solution calculated as " + str('%e' %sol) + " with an error of less
than 1e-7 for c = 2."
```

Which returns the following:

```
Solution calculated as 7.968121e-01 with an error of less than 1e-7 for c =
2.
```

In other words, $x = 0.07968121$ for this value of c.

B. Modify your program to calculate the solution for values of c from 0 to 3 in steps of 0.01 and make a plot of x as a function of c.

First, we can define the range to integrate over: this will be `c_range`. For this, we can use `linspace`, remembering that it does not include the last value in its range (so we supply it with $c_{\max} + 1$).

```
c_range = np.arange(0,3.01,0.01)
```

We're going to create an array (called `x_plot`) of points in order to plot $x(c)$; for that, we can use `zeros(len(c_range))`. The easiest way to fill in that array is using a function, based upon what was already used to find the solution for $c = 2$:

```
def x_filler(c_i):
      eq = lambda x : (1-np.exp((-c_i*x))-x)
      eq_sol = optimize.fsolve(eq,1.,xtol=1e-7)
```

```
        return eq_sol
```

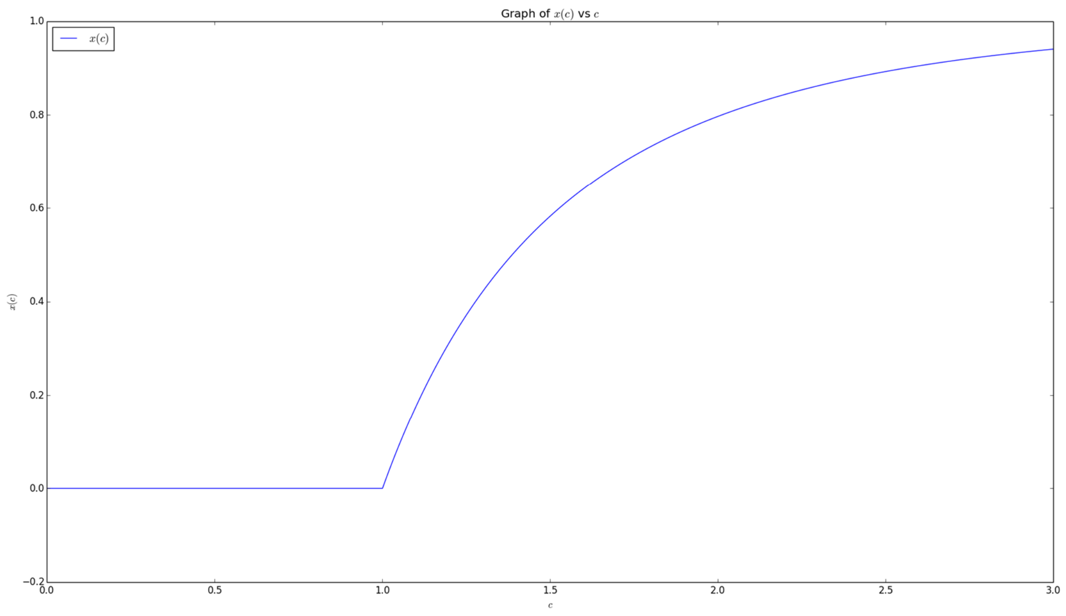
This function can then be called within a for loop, which was used to fill in `x_plot`:

```
    x_plot = np.zeros(len(c_range))
        for i in range(len(c_range)):
        x_plot[i] = x_filler(c_range[i])
```

With the array filled in, plotting is easy.

```
    plt.figure(1)
    plt.plot(c_range,x_plot,label=r'$x(c)$')
    plt.legend(loc=0)
    plt.title(r'Graph of $x(c)$ vs $c$')
    plt.xlabel(r'$c$')
    plt.ylabel(r'$x(c)$')
    plt.plot()
```

This yields the following graph:



From this graph, we can see a very clear transition from the regime where $x = 0$ (this holds for $c = 0 \rightarrow 1$), to where x is non-zero ($x > 1$). The transition is the *percolation transition*.

Question 3 – Numerical integration

Using a method of your choice and tolerance of 1.49×10^{-8} *compute* $g(u)$ *for* $u = 0.05$ *to* 1.0 *in intervals of* 0.05*. and plot the results.*

After importing the libraries and tools that we want to use, the next step is to define the constant in use and the integrand (with the integrand = $\int_0^{\frac{1}{u}} \frac{x^4 e^x}{e^x - 1}\, dx$), along with the range of points which we want to find $g(u)$ for:

```
# Define constants
k = 1.3807e-23

# Define integrand and range of points to integrate over
integrand = lambda x: (((x**4)*(np.exp(x)))/(((np.exp(x))-1)**2))
urange = np.arange(0.05,1.05,0.05)
```

Then we can define the function `g(u)`, which we use to evaluate $g(u)$ at the range of points for $u$:

```
def g(u):
      intres, interror = integrate.quad(integrand,0,(1/u),epsabs=1.49e-08)
      res = (u**3)*intres
      return res
```

(The `epsabs=1.49e-08` argument given to quad specifies the maximum error - 1.49×10^{-8}.) I have chosen to use `quad()` since it is fast, given that it is a wrapper around the highly optimised FORTRAN library `quadpack`.

The next step is to construct an empty array, which will then be filled with data points from `g(u)`.

```
gu_plot = np.zeros(len(urange))
```

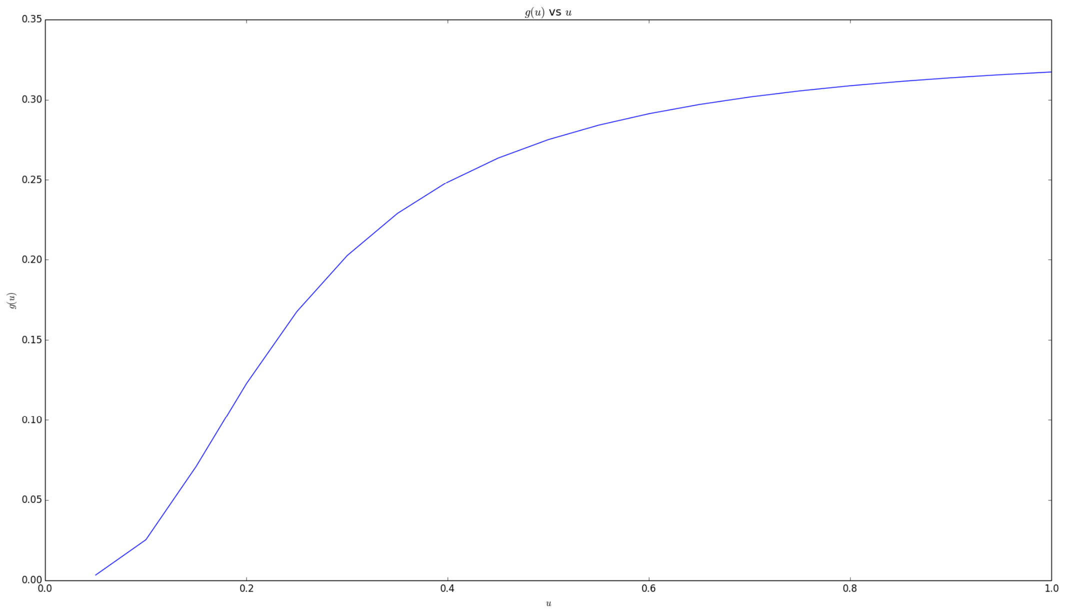
A `for` loop can then be used to fill in the `gu_plot` array, by evaluating the `g(u)` function at the points from the range `urange`.

```
for i in range(0,(len(urange))):
      gu_plot[i] = g(urange[i])
```

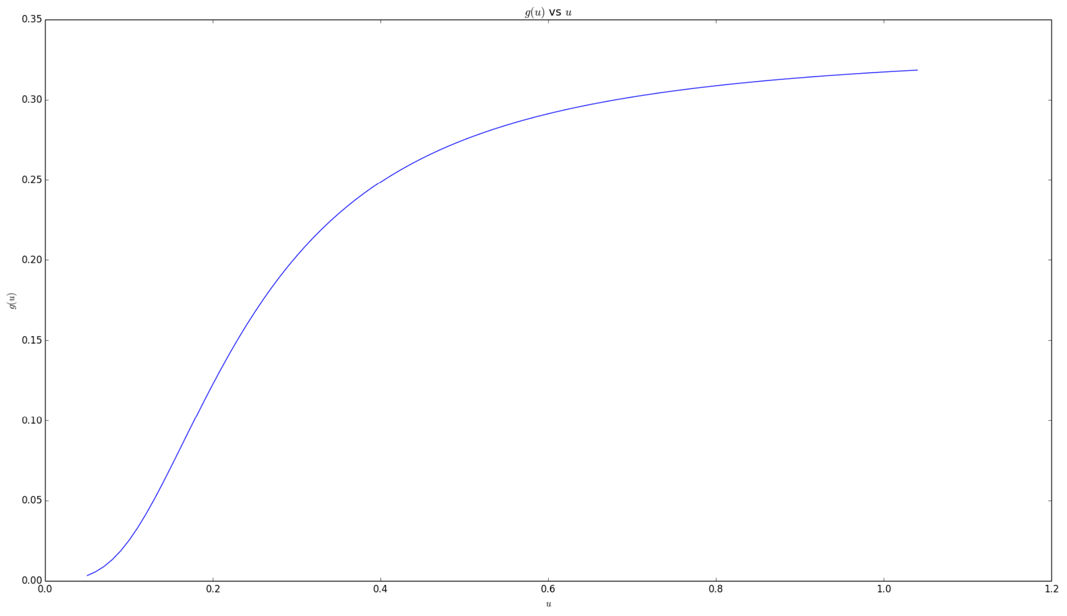
Plotting is then trivial, using `matplotlib`:

```
plt.figure(1)
plt.plot(urange,gu_plot)
plt.title(r'$g(u)$ vs $u$')
plt.xlabel(r'$u$')
plt.ylabel(r'$g(u)$')
plt.show()
```

This yields the following graph:



The slight ‘kink’ in the curve is down to the relatively large interval used for the integration range. BY reducing the interval from 0.05 the curve would be more smooth. The curve below, for example was plotted with the interval of 0.01.



QUESTION 2

A. Convert the above system of equations to 4 first-order equations

By introducing the variable $a = x'$ and $b = y'$, it can be written that $x'' = a'$ and $y'' = b'$. Hence:

$$x' = a$$

$$a' = -c\, a\, \sqrt{a^2 + b^2}$$

$$y' = b$$

$$b' = -c\, b\, \sqrt{a^2 + b^2} - g$$

B. Solve the equations over the time interval [0, 30] for each launch angle. Plot the resulting trajectories.

First, we convert the degrees to radians, using the following:

```
a_r10 = np.deg2rad(10)
a_r20 = np.deg2rad(20)
a_r80 = np.deg2rad(80)
```

I ran out of time to complete this question.