Scientific Computing Examination: Friday 9th January 2015

Writeup

Candidate 117458

Question 1

A. *"Convert the above system of equations to 2 first-order equations."*

We can introduce a new variable x, defining it as $x = y'$. Thus,

$$x = y'$$
$$x' = g - \frac{C_D}{m}x^2$$

B. *Determine the time of a 5000-m fall. Use $g = 9.80665 m/s^2$, $c_D = 0.2028 kg/m$, and $m = 80$ kg. You can use a method of your choice, e.g. `scipy.integrate.odepack` with its default tolerance.*

After importing the relevant libraries, we define a function to encapsulate the derivatives:

```
def sys(q,t):
      yi = q[0]
      xi = q[1]
      f0 = xi
      f1 = (g - ((cD/m)*(xi**2)))
      return [f0,f1]
```

We can then define the initial conditions and a time period to integrate over:

```
g = 9.80665
cD = 0.2028
m = 80
x0 = 0
y0 = 0
ics = [y0,x0]
t = np.linspace(0,100,10000)
```

Next, we perform the integration, using `odeint`. I have chosen to use odeint because it is fast (because it is a wrapper to the fast linpack libraries), and it is accurate.

```
sol = integrate.odeint(sys,ics,t)
y = sol[:,0]
x = sol[:,1]
```

The solutions can then be plotted.

```
plt.figure(1)
plt.plot(t,y, label='Position')
plt.plot(t,x, label='Velocity')
plt.ylim(0,5000)
```

```
plt.legend(loc=0)
plt.plot()
```

To find the time of the 5000m fall, we can use `np.where()` to find the two times between which the value of distance fallen becomes greater than 5000m. `nanmin()` is used since otherwise we get an array of values – we just want the first (smallest) of them.

```
print "Time to fall 5000m is between:"
t5000_b1 = np.nanmin(np.where(y>5000))-1
t5000_b2 = np.nanmin(np.where(y>5000))
```

Then we print the bracketing values to the screen.

```
print str((t5000_b1/100)) + " seconds; here y = " + str(y[t5000_b1])
print "and"
print str((t5000_b2/100)) + " seconds; here y = " + str(y[t5000_b2])
```

This returns:

```
Time to fall 5000m is between:
84 seconds; here y = 4999.55943887
and
84 seconds; here y = 5000.18147382
```

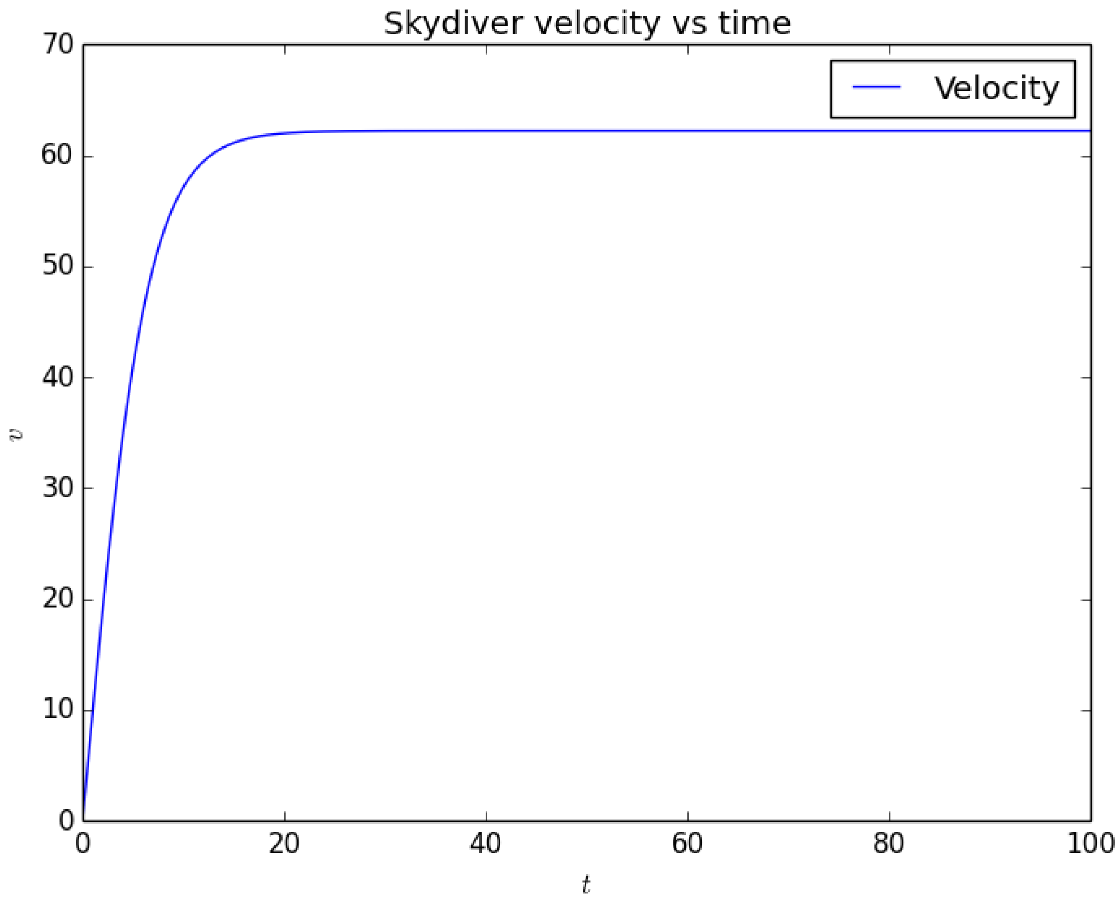
The result is that it takes ≈84 seconds to fall 5000m. The exact time is bracketed as being between 84.77 and 84.78 seconds.

C. Plot the skydiver's velocity vs. time. What is the terminal velocity of the diver (i.e. the velocity when he stops accelerating)?

Plotting is trivial:

```
plt.figure(2)
plt.plot(t,x, label='Velocity')
plt.legend(loc=0)
plt.title('Skydiver velocity vs time')
plt.xlabel(r'$t$')
plt.ylabel(r'$v$')
plt.plot()
```

The plot returned is:



The skydiver will stop accelerating when he reaches the maximum speed. The maximum speed can be found using `np.max()`

```
xmax = np.max(x)
print "maximum speed is " + str(xmax)
```

The maximum speed was thus found as 62.1972747764 ms^{-1}.

QUESTION 2

A. *Use the interpolating polynomial of maximum degree to estimate the value of ρ at h = 2,4, and 8 km. What degree polynomial did you use?*

The polynomial will be of degree six, since there are seven data points in all. You use a polynomial of $n-1$ to fit to n points.

`interpolate.barycentric_interpolate()` was used to interpolate the data. This function was chosen out of conviencience, as with one call it can create the interpolation function and evaluate it at a given point.

```
print "At 2km, the polynomial, interpolated value is:"
print(str(interpolate.barycentric_interpolate(h,rho,2.0)))

print "At 4km, the polynomial, interpolated value is:"
print(str(interpolate.barycentric_interpolate(h,rho,4.0)))

print "At 8km, the polynomial, interpolated value is:"
print(str(interpolate.barycentric_interpolate(h,rho,8.0)))
```

Returning:

```
At 2km, the polynomial, interpolated value is:
0.821776765804
And at 4km, the interpolated value is:
0.668838720134
And at 8km, the interpolated value is:
0.428931828448
```

Thus, the interpolated values are: for $h = 2$ km, $\rho = 0.821776765804$; for $h = 4$ km, $\rho = 0.668838720134$; for $h = 8$ km, $\rho = 0.428931828448$.

B. *Use a cubic spline to estimate the value of ρ at h = 2,4, and 8 km. Are your results different from a)? Explain.*

First, we set up a function that contains the cubic spline fit.

```
fit = interpolate.interp1d(h,rho,kind='cubic')
```

Then, we can evaluate it.

```
print "At h=2, h=4 and h=8 respectively, using a cubic spline, rho="
print fit(2.0), ", ", fit(4.0), ", ",fit(8.0)
```

Great stuff. The result is

```
At h=2, h=4 and h=8 respectively, using a cubic spline, rho=
0.821767808776 ,  0.668840822907 ,  0.428944803738
```

These results vary to the polynomial interpolation, albeit only at high levels of precision. The cubic spline is more likely to be accurate here, because a polynomial interpolation is more likely to over-emphasise anomalous data points: the polynomial fit, by definition, will have construct a fit which passes through every point, even if one point is anomalous. There is also the potential issue with Runge phenomena – i.e. spurious, extreme oscillations in the fit, rendering the fit inaccurate.

C. If the actual value at h = 4 km were 0.67, what are the relative and absolute errors of your estimates for ρ(h = 4km) in (a) and (b)?

First, we define the actual value.

```
rho_actual = 0.67
```

Then, we can go ahead and calculate the errors. The absolute error is defined as $|\rho_{\text{approx}} - \rho_{\text{actual}}|$, and the relative error is defined as $\frac{|\rho_{\text{approx}} - \rho_{\text{actual}}|}{\rho_{\text{actual}}}$.

```
print "Absolute error for polynomial interpolation:"
err_abso_poly = np.abs(interpolate.barycentric_interpolate(h,rho,4.0)-rho_actual)
print err_abso_poly
print "Relative error for polynomial interpolation:"
print np.abs((err_abso_poly)/rho_actual)
```

This returns:

```
Absolute error for polynomial interpolation:
0.00116127986617
Relative error for polynomial interpolation:
0.00173325353159
```

We can then do the same for the cubic spline.

```
print "\nAbsolute error for cubic spline:"
err_abso_spline = np.abs((fit(4.0)-rho_actual))
print err_abso_spline
print "Relative error for cubic spline:"
print np.abs((err_abso_spline)/rho_actual)
```

Returning:

```
Absolute error for cubic spline:
0.0011591770928
Relative error for cubic spline:
0.00173011506388
```

From these two figures, we can see that both interpolations are accurate to a reasonable precision (i.e. 3 decimal places), but the spline is the more accurate of the interpolations as its relative error (along with absolute error) is smaller.

QUESTION 3

A. Using the data $i_0 = 100$ A, $R = 0.5\ \Omega$, and $t\ 0 = 0.01$ s, plot $[i(t)]^2$ for $t = 0$ to 1 s using linear x-axis and logarithmic y-axis.

After importing the relevant libraries (`scipy.integrate`, `numpy` and `matplotlib.pyplot`), the next step is to define the function $\left(i(t)\right)^2$ and the values of the constants in use.

```
def i_squared(x):
      q = (((i0)*np.exp(-x/t0)*(np.sin((2*x)/t0))))
      return (q**2)

i0 = 100.0
R = 0.5
t0 = 0.01
```

Functions cannot be plotted directly with these libraries, so we create a series of vectors which contain the values for t and the corresponding values of $\left(i(t)\right)^2$ which we wish to plot.

```
t_range = np.linspace(0,1,100000)
i_squared_plot = np.zeros(len(t_range))
```

This creates an array with the t values (with 100000 equally separated points – to smooth the curve, a high number is chosen. In this case it isn't too computationally expensive), and create an empty array for the $\left(i(t)\right)^2$ values. The empty array is then filled in by this:

```
for n in range(len(t_range)):
      i_squared_plot[n] = i_squared((t_range[n]))
```

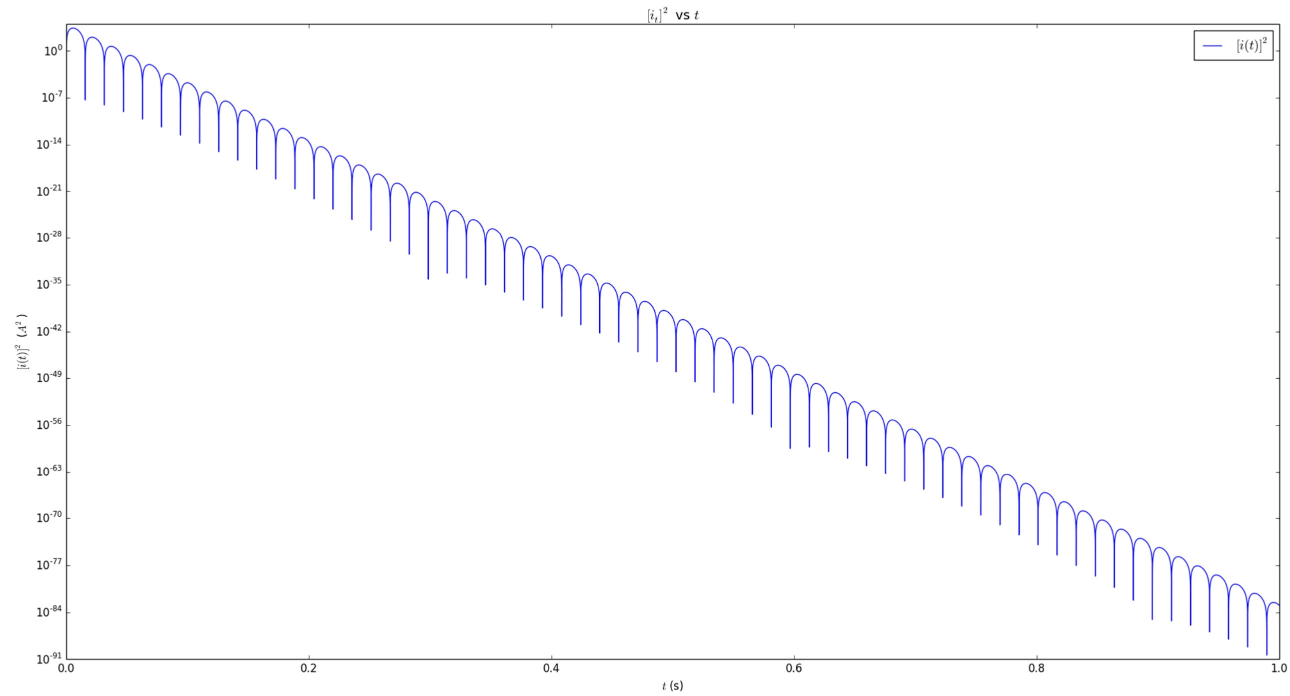
We loop through and fill in the array sequentially using the `i_squared()` function defined previously.

Plotting is then a trivial affair. `semilogy()` is used to make the y-axis a log scale.

```
plt.figure(1)
plt.semilogy(t_range,i_squared_plot,label=r'$[i(t)]^2$')

plt.title(r'$[i_t]^2$ vs $t$')
plt.xlabel(r'$t$ (s)')
plt.ylabel(r'$[i(t)]^2$ (${A^2}$)')
plt.legend(loc=0)
plt.show()
```

This returns:



B. *Find* E

To find E, we simply define a function (we can use a lambda as the function only contains one variable to be integrate) which contains the equation, then call `integrate.quad()`.

```
integrand = lambda t: R*(((i0)*np.exp(-t/t0)*(np.sin((2*t)/t0)))**2)
E, error = integrate.quad(integrand,0,np.inf)
print "E = " + str(E) + " with an error of " + str(error)
```

`quad` automatically returns the error associated with the integration. The result is:

```
E = 10.0 with an error of 1.90924170202e-08
```

So $E = 10$ J.

We can actually disregard the error in this case: not only is it extremely small, but the result of $E = 10$ J is exactly correct. If the integration is performed symbolically (and the limits then computed), we arrive at the same answer.