SESG6025 Coursework: Fractals and Integration

Due: Wednesday 14th December 2011 (12 midday). 20% weighting.

Aims: This coursework uses Newton-Raphson iteration on a non-linear equation to improve the accuracy of an initial guess of the solution. From some initial guesses, the Newton-Raphson method may not converge to a real root of a non-linear equation: this leads to a fractal. We then explore methods for numerical integration.

Objectives: Produce a plot of a simple fractal using Matlab or Python. Use Numerical Integration routines in in Matlab or Python and write a Monte-Carlo integration routine.

Requirements and hand-in: You should email two files (together as attachments to a single email) to sesg6025@soton.ac.uk with the email subject "**Coursework**"

File 1: (Question 1). The file name must be "**fract.m**" or "**fract.py**"

File 2: (Question 2). The filename must be "**integ.m**" or "**integ.py**"

These files, when executed within Matlab/Python, give the results required for each part of questions (1) and (2) on screen. Plots should be in separate figure windows.

Before attempting the questions, read the Matlab/ Python `help` on:

```
for, format, i, meshgrid, shading, surf, colormap, view, feval, exp, trapz, quad, function, for, rand, mean, std, abs, imshow
```

Similar commands exist in Python for plotting using Matlab commands- see course notes on Python for numpy and matplotlib, for example on http://www.soton.ac.uk/~sesg2006 and http://matplotlib.sourceforge.net/

1) Non-linear equation solving and Fractals

a) Using the starting value of z = 1.4, write a piece of Matlab/Python code to perform 5 Newton-Raphson steps to find a better solution of the equation

$$z^3 - 1 = 0 \qquad (1)$$

The output should be to the screen when the .m / .py file is executed.

b) In general there is no restriction on whether we start with a real value or a complex value as the initial guess. Give the 5 values which you obtain if you start with z = -1.0 – 1.5 i, where $i = \sqrt{-1}$. Output results to the screen.

c) For equation (1) there are three roots at z = 1, -0.5 ± 0.866 i. You can check this using Matlab:

```
» roots([1,0,0,-1])
ans =
 -0.50000000000000 + 0.86602540378444i
 -0.50000000000000 - 0.86602540378444i
  1.00000000000000
```

In Python this becomes

```
>>> numpy.roots([1,0,0,-1])
array([-0.5+0.8660254j, -0.5-0.8660254j,  1.0+0.j        ])
```

In this section we repeat (b), but instead focus on which complex starting values converge to the root z = –1 to the equation in (1). The solution will be in the form of a map.

Set up a *grid* of 100 equally spaced x and y values in the range [-2, 2]. You should use the `linspace` and `meshgrid` functions (or equivalents). Set up a new array `z = x + i*y`, where i = $\sqrt{-1}$ and for each point in this array perform 10 iterations of the iteration scheme you derived in Q1. Form a second array, `colour`, as follows:

$$\texttt{colour(j,k)} = \begin{cases} 1 & |z_{10} - 1| \le 1 \times 10^{-6} \\ 0 & \text{otherwise} \end{cases} \qquad (2)$$

where z_0`(j,k) = x(j,k) + i* y(j,k)` is the initial z value (you may find section 1.8 in the Matlab handout book useful for an easy way to set up the array `colour`).

Produce a surface whose height is given by the entries in the `colour` array at each `x` and `y` value. The shading for this plot should be interpolated using the 'cool' colormap in Matlab, or a suitable Python colormap. The output figure should be viewed from the top (`view(2)`). You may also use `imshow`. Python provides matplotlib.imshow(), see section 15.1.9 in http://www.soton.ac.uk/~sesg2006/textbook/Python-for-Computational-Science-and-Engineering.pdf

[For information only: The picture which you have produced is a fractal. The points which are shaded in are those which when used as starting values for the Newton-Raphson iteration converge to the root at 1. Other fractals, such as the Mandelbrot set, use a different iteration scheme and the colouring of the point at (x, y) is dependent on how quickly (if at all) the iteration scheme diverges from an initial guess $z_0 = x + i \times y$.

A fractal is a shape which has self-similar structure on all scales of magnification. Whilst it is *not* part of the coursework, you can zoom in on a small region of the set (e.g. centred near to the edge of the shaded region) and repeat the iterations to reveal further structure. You can also use a much finer resolution to produce the plots and more detail will be visible. For more information on fractals see: Peitgen, Jürgens, and Saupe (1992) "Chaos and Fractals. New Frontiers of Science" (Springer-Verlag).]

2) Numerical Integration

a) Plot the function e^x using 10 equally spaced points between 0 and 4. Show circles for the points and join up the points with a straight line.

b) Using the trapezium rule find the integral of the function e^x using 10 points over the range 0 to 4.

c) The exact answer is given by:

d) $$\int_0^4 e^x = e^x\Big|_0^4 = e^4 - e^0 = \texttt{exp(4) - exp(0)}$$

e) Evaluate the integral as in (2) using N = 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, and 200 equally spaced points over the range and plot the absolute percentage value of the error = $\dfrac{\left|I_{exact} - I_{trapezium}\right|}{I_{exact}} \times 100\%$ as a function of the number of points used, N.

f) Evaluate the integral in (4) using the '`quad`' function and a suitable tolerance. Use `scipy.integrate.quad` with default settings in Python

g) Monte Carlo integration. It is possible to estimate an integral using the following formula:

$$I = \int_{x=a}^{b} f(x)\,\mathrm{d}x \approx (b-a)\left\langle f(x_i)\right\rangle \pm \frac{(b-a)}{\sqrt{N}}\sqrt{\left\langle f(x_i)^2\right\rangle - \left\langle f(x_i)\right\rangle},$$

where the x_i are chosen randomly between a and b, N is the number of sample points, and:

$$\left\langle f(x_i)\right\rangle = \texttt{mean(f)} \text{ and } \sqrt{\left\langle f(x_i)^2\right\rangle - \left\langle f(x_i)\right\rangle} = \texttt{std(f)}.$$

Using for x_i 10000 random numbers in the range 0 to 4, estimate the integral calculated in (4) and give the one standard deviation error bound for it (this is just the the ± part).

(Hint: To scale random numbers in the range 0 to 1 to be in the range a to b, you should use `(b - a)*rand(N,1) + a`, where $b > a$ and N is the number of points to generate. In Python, use `numpy.random.rand(N)`)

Why might you ever want to use this method? For multi-dimensional integrals in d dimensions, the error for the trapezium rule falls off as $N^{(-2/d)}$, where N is the number of points used/ function evaluations made. However, the $N^{(-1/2)}$ dependence of the error for the Monte Carlo integration is independent of the number of dimensions; hence for large d, the Monte Carlo method is actually more accurate for a given number of function evaluations. For Simpson's rule the error goes as $N^{(-4/d)}$ in d dimensions. By using quasi-random sequences it is possible to arrange for the error to drop of as $N^{(-1)}$ independent of d.

Please ask if you need help.Prof Simon Cox, sjc@soton.ac.uk. Building 25/2037 Phone Ext 23116.