Maths 6141 - Numerical Methods

Computer Lab 2

Black box methods

A lot of key algorithms have been robustly implemented by others, either within python or in packages python can easily call. These can be used as a black box. The aim of this lab is to gain familiarity with some standard commands relating to matrices, linear systems, quadrature and solving differential equations.
The key packages to investigate are `numpy` (`np`) and `scipy`.

Tasks

Matrices

We can rapidly create certain simple matrices, using commands such as `zeros`, `ones`, `rand` (`np.random.randn`), and manipulate it to find determinant (`det`), trace (`trace`) and eigenthings (`eig`). Check that you know these commands, as well as how to manipulate a diagonal matrix (`diag`). You should be able to:

1. Create a vector of size 8 whose entries are $1+\delta R$ where R is a uniformly distributed random number and δ is a small real number, say 10^{-4}. This should take one line of commands.

2. Create a diagonal matrix whose entries along the diagonal are the entries of the vector produced in the previous part. Use the same command to create matrices containing non-zero entries only on a single diagonal line that is not the principal diagonal.

Sometimes you need to force a collection of numbers (an array) to have a different shape than python believes it should. Key commands for this include `reshape`, `hstack`, and `vstack`.

3. Take the random vector created earlier and repack into an array with two rows and four columns, four rows and two columns, and a three dimensional $2 \times 2 \times 2$ array.

Quadrature

Unsurprisingly python contains a simple command for quadrature, which is the numerical integration of a function, implementing a highly accurate adaptive routine. The command is `scipy.integrate.quad`, which takes a *function*, an interval, and optional arguments including the tolerance through an options table. There are two ways of defining a function. The standard way is to create a function, either inline or in a separate file, that has one argument. An example would be

```
def f1(x):
  import numpy as np
  return np.sin(x)**2
```

This can easily be done using the editor. To integrate this function between 0 and π you would call

```
> scipy.integrate.quad(f1, 0, np.pi)
```

An alternative that works for simple functions is to create an *anonymous* or *lambda* function. This can be typed directly into the command window (or in a script), using

```
> f1 = lambda x: np.sin(x)**2
```

This can then be used in the quadrature function.

1. Create a function $f_2(x) = \exp(-x^2 \cos^2(2\pi x))$ and integrate it over the interval $[0, 1]$.
2. Repeat this using an anonymous function, checking that you get the same answer.
3. Modify the tolerance to *improve* the accuracy of the integration to see how much it makes a difference; you may have to increase the output precision.
4. Investigate the help to see other types of black-box quadrature methods, such as `quadrature`, `romberg`.

Differential equations

All the black box methods for solving ODEs assume that the problem is written as a first order system

$$\mathbf{y}'(x) = \mathbf{f}(\mathbf{y}, x).$$

As we will see in lectures there is a big difference between methods for *Initial Value* problems, where $\mathbf{y}(a)$ is given, and *Boundary Value* problems where values are fixed at the boundaries of the interval $[a, b]$. There are a number of python IVP solvers. The default is `scipy.integrate.odeint`.
Consider the differential equation

$$y'(x) = -y(x), \quad y(0) = 1, \quad x \in [0, 10].$$

Clearly the answer is $y(x) = e^{-x}$. To compute the solution using `odeint` we can use

```
> # Create an anonymous function defining the IVP.
> f = lambda y, x: -y
> x = np.linspace(0, 10)
> # Solve using odeint.
> solivp = odeint(f, [1.0], x);
```

The first argument is the function; the second is the initial value; the third are the points at which the solution is required.

1. Solve the differential equation

   $$y'(x) = -\cos(y(x)), \quad y(0) = 1, \quad x \in [0, 10].$$

   Use `odeint`. Plot the result.
2. Look at the help for `odeint` to work out how to set the tolerance for the solve. Reduce the accuracy by a few orders of magnitude to see how much difference it makes.
3. Investigate the `scipy.integrate.ode` command to see what alternatives there are.
4. Solve the differential equation

   $$y'(x) = -C(x)y(x), \quad y(0) = 1, \quad x \in [0, 10].$$

   Define C as

   $$C = 1 + \int_0^x \sin^2(s)\, ds.$$

   Solve for C using the in-built quadrature routine `quad`.
5. Solve the system

   $$\begin{pmatrix} x(t) \\ y(t) \end{pmatrix}' = \begin{pmatrix} -y(t) \\ x(t) \end{pmatrix}, \quad t \in [0, 500], \quad \begin{pmatrix} x(0) \\ y(0) \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

   Plot the solution in three different ways: x, y vs. t, x vs. y, and r vs. t, where $r^2 = x^2 + y^2$.

Nonlinear roots

Finding the roots of nonlinear equations is an important part of the course; it consists of finding s such that

$$f(s) = 0$$

for an arbitrary function $f(x)$.
There are many python black box solvers for this problem; for scalar problems the standard is `scipy.optimize.brentq`, and for systems is `scipy.optimize.fsolve`. The simplest way to use it is to provide a function and an initial guess for the root, such as

```
s = scipy.optimize.brentq(lambda x: np.cos(x) - x, 0.0, 1.0)
```

The `fsolve` function will solve for systems (that is where f returns a real vector given a real vector), but the input guess must also be an appropriate vector.
As with previous black box solvers there are additional options to control the tolerance of the algorithm, and additional output is also possible.