MATH 320(!) - preliminaries

1 Prerequisite knowledge

The following techniques taught in MATH 222 should be natural to you (to the extent that you can write out a sample problem and solve it on the spot):

- Integration techniques (integration by parts, partial fractions, trig substitution, ...)
- Taylor series
- Vectors
- Slope fields
- Solving a separable first-order differential equation
- Solving a linear first-order differential equation using the integrating factor method

You should be very comfortable with algebraic manipulation of trigonometric functions, exponentials, logarithms, etc.; if you have trouble solving algebraic equations, this is not the course for you.

2 Preliminary exam

You should not find these questions particularly challenging.

1. $\lim_{f \to \infty} f \ln(f)$

2. What is the fifth-order Taylor expansion of $f(x) = x^3 \cos(x)$ about $x = \pi/2$? (Use "the fast" way...)

3. Give an equation for the plane that is tangent to the surface $f(x, y) = xy \cos(xy)$ at the point $(x, y) = (\pi/2, 0)$.

4. $\int_{\pi/6}^{\pi/3} \cos^2 \theta - \sin^2 \theta \, d\theta$

5. $\int \frac{x}{x^2 - 1} \, dx$

6. $\int \frac{1}{t^2 - 1} \, dt$

7. $\int \frac{1}{z^3 + 1} \, dz$

8. $\frac{d^2}{dt^2} \int_t^1 \frac{\sin(x)}{1 + \log(x)} \, dx$

9. Solve $y' + 2 \tan(t) y = 1$, $y(0) = 0$.

10. Solve $y' = t^2 y^2$, $y(0) = 1$.

3 Python

In this course we will learn and use the Python computing language, which is free. Before taking this course, you are expected to install a version of Python on your laptop (or to find a computer on which this can be done) and to go through the following short training program. The more exposure you have to Python before class begins, the more you will enjoy and learn in the course.

3.1 Step 1: Installation

Gain access to Python. I suggest installing Enthought Canopy (free with an academic license: https://www.enthought.com/academic-subscriptions/) which comes with everything packaged together at once and has a built in IDE (editor). Alternatively, many people use the Anaconda distribution and then Spyder to edit. Both distributions are free. Google around, you'll figure it out.

Anaconda/Spyder have been installed on all the computers on the labs on campus, so you can simply visit one such lab instead if you prefer: https://www.doit.wisc.edu/services/computer-labs-infolabs/.

Be sure that you can run the two codes: Euler.py and MatrixPlayground.py (see Canvas). The first should output a figure, the second should output the results of a few matrix operations. Read through each code line by line; play around a bit, make some minor changes and run again.

3.2 Step 2: The basics

Go through the wonderful notes written by Will Mitchell which are attached below. You are not expected to know what's going on in the differential equation and linear algebra sections of course!

Separately, a very nice website is: http://www.scipy-lectures.org/intro/index.html. I suggest spending some hours there...

AN INTRODUCTION TO PYTHON FOR SCIENTIFIC COMPUTING

WILL MITCHELL

Python is a flexible and very widely used computer language. Python is:

- Freely available – no software keys to buy and remember
- High-level, compared to, say, C – this means that your code is easier to write and slower to execute

You can learn a lot from reading general-purpose Python tutorials, but these frequently present a lot of material that is not relevant for numerical stuff– manipulating strings, for example. The NumPy tutorial assumes some background in Python. This document introduces both together; it is what would have been most useful to me when I began coding. The first nine sections cover the following:

(1) Installation
(2) Python as a Calculator
(3) Modules
(4) Vectors and Matrices
(5) Plotting
(6) Directing Traffic
(7) Exercise on (1) - (6)
(8) Functions
(9) Exercise (ODEs)

Any mildly ambitious computational project in Python will use all of sections 1-6,8. The remaining sections cover material that is frequently but not always useful: linear algebra, saving data, clocks, strings, movies, and classes.

1. Installing Python/pylab, in Five Minutes

All of the software we will use is free. You can get python installations from many places online. On Ubuntu I prefer to use `ipython`. For Windows and Apple systems a convenient source is Enthought Canopy Express: www.enthought.com. While you wait for it to install, try this literature quiz: who or what is rocinante?

2. Python as a Desktop Calculator

2.1. **Integer and Floating-Point Arithmetic.** Variable names can include numbers and underscores, but they cannot start with a number: `A2` and `var_j` are OK, but not `3e`. **Do not** start variable names with underscores; these are usually reserved for built-in commands that you should not redefine. Try typing this, hitting return after each line:

```
a = 20
b = 3
print a, b, a+b, a-b, a*b, a/b
```

Next try the following:

Date: May 10, 2016.

```
a = 20.0
b = 3.0
print a, b, a+b, a-b, a*b, a/b
```

This should illustrate that Python is a *typed* language. Numbers are either integers or double-precision floating point numbers, and variables are initialized as one or the other depending on whether you include a decimal point. **Danger:** Python treats integer arithmetic differently from floating arithmetic, and you should keep this in mind. It is good programming practice to use floats for everything except things that really must be integers, such as the index of an entry in a vector.

Questions:

- What happens if you try to add an integer to a floating point number?
- Investigate the frequently useful fifth arithmetic operation `%`.

To compute a^b the syntax is `a**b` or `pow(a,b)`.

Calculus functions are as you'd expect: `exp, sin, cos, cot, arcsin, log, log10, tanh`, et cetera– but these are not included in basic Python. You need to load them from a module, as we'll discuss below.

The square root of -1 is the engineers' `1j`. Note that `j` does not work by itself; you always need a number preceding the `j`. Example: typing `exp(pi*1j)` gives `(-1+1.2246063538223773e-16j)`, which is quite close to -1; note the scientific notation for the imaginary part. This is not the moment for introducing floating-point arithmetic, but if you are new to machine computing you should be told not to trust anything smaller than 10^{-15}.

2.2. **On Spaces.** Consider the three lines:

```
a=20
a   =        20
   a = 20
```

The first two are equivalent, but the third gives an *indentation error*. In Python, *spaces at the beginning of a line are important*. They serve to denote the nesting of blocks of code. We will see examples when we cover traffic flow. In contrast, *spaces after the beginning of the line are decorative*. I think `a = b + 2` is easier to read than `a=b+2`, but it's just style.

2.3. **Initializing Multiple Variables; Modifying a Variable.** It's perfectly OK to write

```
a = 20.0
b = 3.0
a = a * 2.
```

but the equivalent statements

```
a,b = 20.0, 3.0
a  *= 2.
```

save some space and are certainly cooler.

3. Modules

Python is a general-purpose programming language. It does not have native support for doing interesting math, so we need to extend it by importing modules. Some relevant ones are listed here:

- `numpy`, for general numerical work including the wonderful `array` class
- `matplotlib`, for 2D plotting (includes some 3D also, but this isn't great)
- `mayavi`, for 3D plotting
- `pylab`, a combination of `numpy, scipy, matplotlib` – usually all you need
- `sympy`, for symbolic manipulations

- `scipy`, which includes all of numpy as well as useful submodules such as:
  - `scipy.integrate`, for numerical integration
  - `scipy.linalg`, for dense linear algebra
  - `scipy.sparse.linalg`, for sparse and iterative linear algebra routines
  - `scipy.spatial`, for geometry methods *e.g.* Delaunay triangulation, nearest neighbor searching
- `dolfin`, my favorite finite element software package
- `Pickle`, for reading/writing data from/to files
- `time`, for clocking your code.

Sample code for importing these modules is given below. Start by opening a Python window. Typing `pi` and hitting return should get you an error; the value of π is not needed frequently enough to be included in the basic Python language. We will load the 16-digit approximation of π from the `numpy` package. Five ways to do this and then display the decimal approximation are given on the five lines below:

```
import numpy;                print numpy.pi
import numpy as np;          print np.pi
from numpy import pi;        print pi
from numpy import pi as PI; print PI
from numpy import *;         print pi
```

The simplest is the last line; this imports every class, function, and constant (like `pi`) from the numpy module. For many computational programs it is fine to just include the line `from pylab import *` at the top of the file and go on with your life.

If you read professional Python code, you may notice that people avoid using `from X import *`, usually preferring versions of the second or third lines above to the fifth. One reason is efficiency; if you really only need a value for π, it is excessive to load the whole `numpy` module. A second reason to avoid the $*$ version arises when you are using multiple packages. For example, the `numpy` and `dolfin` modules both have functions named `solve`, and you may need both; you will want to `import .. as ..` so you can call one using `np.solve` and the other using `dolfin.solve`. Otherwise, the most recent import wins.

4. Vectors and Matrices

Vectors and matrices are ubiquitous. Creating and manipulating them are essential parts of pretty much every computational program you will write. I assume we have already typed `from pylab import *` in this section.

4.1. **Vectors.** The basic tools for producing vectors are the following:

- `ones(N)` is a vector of length N whose entries are all 1..
- `zeros(N)` is a vector of length N whose entries are all 0..
- `arange(N)` is a vector of length N whose entries are $0, 1, \cdots, N-1$. Note that the entries are integers iff N is, so `arange(3)` and `arange(3.)` are not identical.
- `arange(a,b,s)` is the vector whose entries are $a, a+s, \cdots a+ks$ where k is the largest integer such that $a + ks < b$, so the last entry is always less than b. Again the entries are integers iff all of a, b, s are integers. Use `linspace` instead if the step s is not an integer.
- `linspace(a,b,N)` is a vector of length N whose entries are evenly spaced and range from a to b. The last entry is b by default; to exclude b use `linspace(a,b,N,endpoint=False)`.
- `rand(N)` is a vector of length N with entries chosen uniformly from $(0, 1)$.
- `randn(N)` is a vector of length N with normally distributed entries with mean 0, standard deviation 1.

Once you have a vector, you can modify its entries individually: to create $[0., 1., 0., 1., 0., 1.]$ you can type

```
a = ones(6)
a[0] = 0
a[2] = 0
a[4] = 0
```

or, equivalently,

```
a = ones(6)
a[[0,2,4]]=0
```

noting the zero-based indexing. We can modify all entries starting from (or before) a certain index:

```
a = arange(6); a[2:]=0; a[:2]=4
```

gives $[4, 4, 0, 0, 0, 0]$. You can also count backwards from the end;

```
a = arange(7); a[[-3,-1]]=0
```

gives $[0, 1, 2, 3, 0, 5, 0]$. Finally you can even modify based on complicated conditions. A third way to get $[0., 1., 0., 1., 0., 1.]$ is to type

```
a,b = ones(6), arange(6)
a[b%2==0] = 0
```

to zero all entries with indices equal to 0, modulo 2. As a last resort, type the entries individually:

```
a = array([0,18,1,6,12,-9,0.1])
```

noting that the presence of one float will make all of the entries into floats.

You can extract subvectors as well:

```
a = array([0,1,8,3,8,5])
b = a[[0,3,5,2]]; print b
```

gives $[0, 3, 5, 8]$. The "subvector" can be longer than the original vector, too:

```
a = array([2,4,5,6,1,-3])
b = a[[0,0,1,2,2,5,2]]; print b
```

gives $[2, 2, 4, 5, 5, -3, 5]$.

Once you have created a vector, you can call on many routines to extract information about it. Most of the commands `sum, abs, max, min, len, sort, norm` are transparent; we will just clarify that the Euclidean length is `norm`, the number of entries is `len`, and the vector of elementwise absolute values is `abs`. Other occasionally useful commands include `argsort` and `setdiff1d`, which we illustrate by noting that `argsort([-2,-4,3,2,8])` gives `[1,0,3,2,4]` and `setdiff1d([1,2,3,10],[2,3,5]) = [1,10]`. Syntax deserves a few comments. Some of these commands, like `sum, max, min`, can be called either as `a.sum()` or `sum(a)`; both return a number. However, the commands `a.sort()` and `sort(a)` are different: the first changes `a` by reordering the entries in ascending order, while the second returns a sorted copy of `a` without modifying `a`. Finally `norm, len, abs` are called only by the form `abs(a)`; the command `a.abs()` returns an error. This will all make more sense once we learn something about classes and object-oriented programming; for now it is fine to use any syntax.

Vector arithmetic is fairly convenient. For example,

```
a = arange(5.)
b = ones(5.)
print 3.0 * a + 17.1* b
```

produces

```
[ 17.1  20.1  23.1  26.1  29.1]
```

Note that $*$ and / and % are the elementwise versions. Try `dot(a,b)` and `cross(a,b)` for vector dot and cross products – of course, `sum(a*b)` is the same as `dot(a,b)`.

Danger: be careful about initializing vectors from other vectors. If you type

```
a = arange(4); b=a; a[2] = -1; print a, b
```

you will see that both a and b have -1 in the third position (2 is the third number counting from 0). If you intend to modify one variable but not the other, you should not write `b=a` but instead `b = a.copy()` or `b = a+0`.

4.2. **Matrices and Higher-Dimension Arrays.** Generation of matrices is similar. Examples:

```
a = ones((3,3))
b = zeros((3,3)); b[0,2] = 4
c = -2*eye(3)
d = arange(9).reshape((3,3))
print a; print b; print c; print d
```

gives

```
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
[[ 0.  0.  4.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
[[-2. -0. -0.]
 [-0. -2. -0.]
 [-0. -0. -2.]]
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

You can extract columns and rows; for example, with the above `a,b,c,d` the command `print d[1,-1], b[:,2], b[0,:],c[2,1:]` returns 5 [4. 0. 0.] [0. 0. 4.] [−0. − 2.]. The main diagonal of the matrix A is available through `diag(A)`; note that if A is instead a vector this will return a square diagonal matrix with the entries of A on the diagonal.

To construct a block matrix $[A\ B]$ from A and B, use `hstack((A,B))` or "horizontal stack." The command `vstack` is similar. For dimensions greater than 2, use `ones((2,2,2))`, etc.

Many of the functions we call for vectors also work on higher-dimensional objects, e.g. the maximum entry of a matrix is `A.max()`. Note that `dot(A,B)` is matrix multiplication while `A*B` is elementwise multiplication (Schur product). For arrays of dimension ≥ 2, the `len` command is replaced by `size` and `shape`:

```
b = randn(4,8)
print shape(b)
print size(b)
print size(b,0)
print size(b,1)
```

gives the output

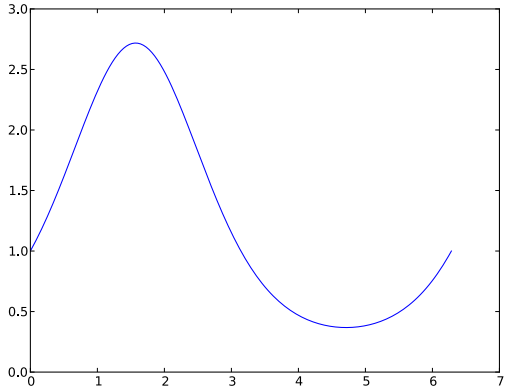
```
(4, 8)
32
4
8
```

Finally, we mention that `spy(A); show()` gives a plot of the nonzero entries of A; this is useful when you deal with sparse matrices. See the linear algebra section for details.

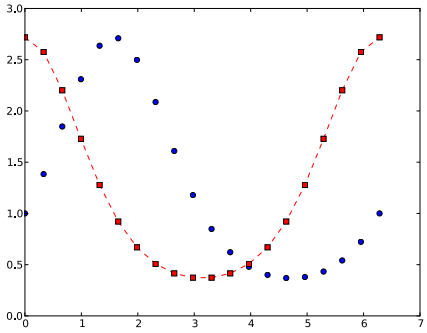
5. Plotting

We can now plot some data. For 2D plotting this means we first assemble a vector x containing the values of the independent variable and a vector y containing those of the dependent variable. We then call the plot() command. A few samples:

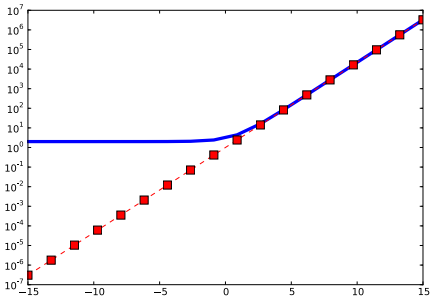
```
from pylab import *
x = linspace(0,2*pi,150)
y = exp(sin(x))
plot(x,y)
show()
```



```
from pylab import *
x = linspace(0,2*pi,20)
y = exp(sin(x))
z = exp(cos(x))
plot(x,y,'bo')
plot(x,z,'rs--')
show()
```



```
from pylab import *
x = linspace(-15,15,18)
y = 2+exp(x)
z = exp(x)
semilogy(x,y,'b-', linewidth = 4)
semilogy(x,z,'rs--', markersize = 10)
show()
```



The third argument of the plot command is a string containing a color, marker style, and line style. `'bo'` gives blue circles with no connecting lines, while `'rs--'` gives red squares connected by dashed red lines. We can take control of these features at a much more detailed level to get publication-quality figures; try `plot?` to get started.

We can save the figures (in a variety of file formats) using the GUI or by inserting the command

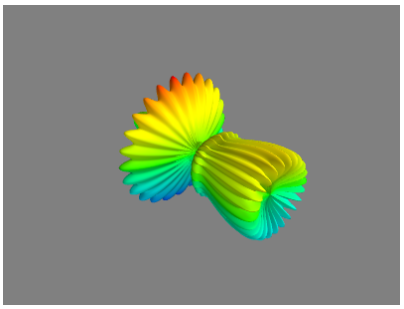
```
savefig('FILENAME.pdf')
```

before or instead of the `show()` command.

5.1. **3D plotting.** For 3D plotting, I recommend starting with the `mlab` module in the `mayavi` package. Here is an example from the page `http://docs.enthought.com/mayavi/mayavi/mlab.html`:

```
# Create the data.
from numpy import *
phis, thetas = linspace(0,pi,251), linspace(0,2*pi,501)
[phi,theta] = meshgrid(phis,thetas)
m0,m1,m2,m3,m4,m5,m6,m7 = 4,3,2,3,6,2,6,4
r = sin(m0*phi)**m1 + cos(m2*phi)**m3
r +=sin(m4*theta)**m5 + cos(m6*theta)**m7
x = r*sin(phi)*cos(theta)
y = r*cos(phi)
z = r*sin(phi)*sin(theta)

# View it.
from mayavi import mlab
s = mlab.mesh(x, y, z)
mlab.savefig("view.png")
```



6. Traffic Flow

6.1. **Booleans.** A Boolean variable can take only two values, `True` and `False`. Most Booleans occur when two numbers are compared: try typing

```
a = 6==7
b = 4. < 5.
c = 5 != 7
d = 3<5<1
e = a and b and c and d
print a,b,c,d,e
```

Note that the `=` sign in each line is used for variable assignment, while `==` tests equality. This gives `False True True False False`; do you see why? The commands `<=` and `>=` for $\leq$ and $\geq$ are occasionally useful too.

Danger: Never use `a==b` to test equality of floating point numbers. Instead, use

```
abs(a-b) < 1e-14
```

or something similar. To see why, note that `0.1 * 3 == 0.3` returns `False`. This is because 0.1 has no exact representation as a floating point number, so you have errors in the 16th digit. In fact, `0.1*3` gives `0.30000000000000004`. From a numerical perspective, this is the same as 0.3. From an algebraic perspective it isn't, and `==` takes the algebraic view.

6.2. **Flow.** Directing traffic is easy. Here are three programs to print out a list of the first fifteen integers, then print 'done'. Note that *indentation is meaningful.*

```
j = 1
while j < 16:
    print j
    j += 1
print 'done'
```

```
j = 1
while j < 30:
    if j <= 15:
        print j
    elif j > 50:
        print 'Gneiss.'
    else:
        pass
    j += 1
print 'done'
```

```
for j in 1+arange(15): print j
print 'done'
```

These examples should illustrate the popular `while`, `if`, `if-elif-else`, and `for-in` constructions. The `pass` is a placeholder doing nothing; you'll get an error if you write `else:` and then

fail to include any indented lines. Note that in the case of a one-line "block" of code you can put it on the same line as the colon, as in the third and tersest example using `for` above.

A fourth way to print out the first fifteen integers followed by `done` is:

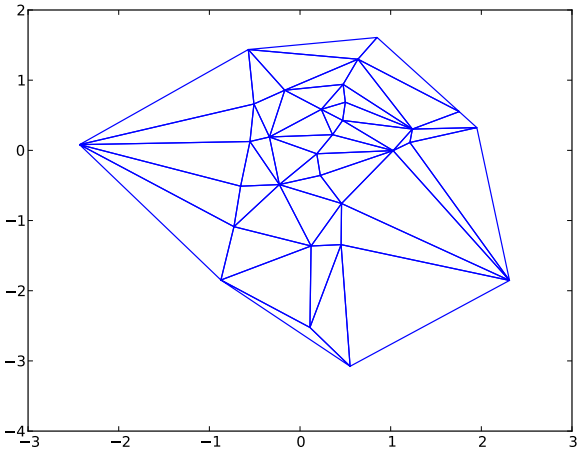
```
for j in 1+arange(15):
    try:
        assert j == 100
    except:
        if j < 16:
            print j
print 'done'
```

The `try X`, `except Y` combination in this last example is quite useful for debugging. First the code inside X is executed. If this produces an error, the code in Y runs. If X produces no error then Y is skipped. In this example we used try-except in combination with the `assert` command. Typing `assert Z` produces an error if Z is false and does nothing if Z is true. Filling your code with assert commands is a good way to make it understandable to others.

7. An Exercise

The following program plots a Delaunay triangulation of 30 randomly chosen points whose x- and y-coordinates are normally distributed about 0 with standard deviation 1.

```
from pylab import *
from scipy.spatial import Delaunay
P = randn(30,2)
T = Delaunay(P).vertices
figure(1)
clf()
for j in arange(size(T,0)):
    x = P[T[j,[0,1,2,0]],0]
    y = P[T[j,[0,1,2,0]],1]
    plot(x,y,'b-')
axis("equal")
show()
```



Modify this program so that it also displays a red circle at the center of each triangle.

Bonus: decide what makes a triangle skinny, and only indicate the centers of nonskinny triangles. To figure out what's going on, you might want to reduce the number of points, then display `P` and `T` and `P[T[0,[0,1,2,0]],0]` to see how they are related.

8. Functions

Defining your own functions is easy. A few examples suffice to indicate the syntax:

```
def square_me(x):
     return x**2
def HelloWorld():
     print 'hello world'
def spectral_radius_and_diagonal_entries(A,k):
     assert len(A.shape) == 2 and A.shape[0] == A.shape[1]
     w,v = eig(A)
     sr = abs(w).max()
     fe = diag(A)[:k]
```

```
    return sr, fe
```

The first function squares anything you feed it and returns the result; the second prints a rather dumb message and returns nothing; the third accepts two inputs `A` and `k` and returns two outputs: (1) the absolute value of the largest eigenvalue of A and (2) a vector containing the first k entries of the diagonal of A. Note that the third function uses an `assert` statement to check that the first input is a 2D square array. Having defined these you could continue with:

```
n = 5
n2 = square_me(n)
HelloWorld()
C = eye(12)
s, f = spectral_radius_and_diagonal_entries(C,7)
print n2 - sum(f) + s
```

to obtain the output

```
hello world
19.0
```

The subtraction in the last line works out to be 25 - 7 + 1 since the spectral radius of the identity map is 1 along with all of the diagonal entries. Effective use of functions can make your code very tidy.

8.1. **Default Argument Values.** An advanced feature is to give the function arguments default values. This makes them optional arguments which the user can choose not to specify. For example, the `plot` command has an optional argument called `linewidth` determining the thickness of any graphed lines. Most of the time we don't care and it would be annoying to have to specify the line thickness; occasionally we do care, and want to take over. The syntax is as follows:

```
def power(x, N=8):  return x**N
```

Here the user must supply a value for x and may optionally specify N; if no N is given, we take $N = 8$. If you write long and complicated functions, default values can make your code much more flexible and general.

9. An Exercise on ODEs

Write a short script beginning with the lines

```
from pylab import *
from scipy.integrate import odeint
```

to numerically simulate the Lokta-Volterra model of predation:

$$x' = x(\alpha - \beta y)$$
$$y' = -y(\gamma - \delta x)$$

Here take $\alpha = 2, \beta = 0.3, \gamma = 0.3, \delta = 0.01$ and use initial conditions $(x, y) = (80, 7)$. Solve the equations out to time $t = 40$. Then plot the values of x and y against t. Make a second plot of x versus y. Below I explain how to use `odeint` to solve a related problem:

Suppose we want to solve the nonlinear 2nd-order initial value problem

$$y'' + 2yy' + y = \sin(x); \quad y(0) = 0, y'(0) = 2.$$

We can do this with off-the-shelf methods in Python. We first have to write it as a system of first-order ODE's. Setting $w(x) = y'(x)$, we have

$$y'(y, w, x) = w$$
$$w'(y, w, x) = \sin(x) - 2yw - y$$

together with initial conditions $y(0) = 0, w(0) = 2$. To solve this numerically we have to define a function accepting two arguments: (1) a vector `V` whose first and second components `V[0]` and `V[1]` are the values of y and w, and (2) the value of the independent variable x. We call this function `F`; it should return a 2-vector containing the derivatives y', w'. We type:

```
def F(V,x): return array([ V[1], sin(x) - 2*V[0]*V[1] - V[0] ])
```

We then define the initial condition and the x-values where we wish to know the solution:

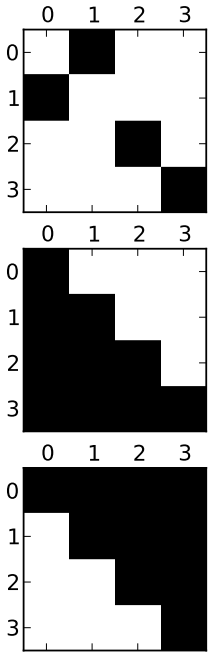
```
IC = array([0.,2.])
x  = linspace(0,3*pi/2,200)
```

Finally, we type `Y = odeint(F, IC, x)`. This returns a 200×2 vector Y containing the values of y in the first column and those of w in the second column. Note: the vector x is different from the set of points where the integrator evaluates F in the quadrature process; there is a variable-step method under the hood here. You do not need to specify a closely-spaced vector x in order to get accurate results.

If you want a few bells and whistles, use the help function to learn about the optional arguments of `odeint`. For example, you can supply the Jacobian of F for faster computation.

10. Linear Algebra

Pylab already has the built-in methods `svd, eig, solve`. Other linear algebra routines include `from scipy.linalg import lu, qr` for the A=PLU and A=QR matrix factorizations. Iterative methods are also available: `from scipy.sparse.linalg import cg, gmres` loads the conjugate gradient and GMRES functions. To get the figure at right:




```
from pylab import *
from scipy.linalg import lu
A = randn(4,4)
P,L,U = lu(A)
assert norm(A - dot(P,dot(L,U))) < 1e-14
figure(1)
subplot(311); spy(P)
subplot(312); spy(L)
subplot(313); spy(U)
savefig('PALU.pdf',bbox_inches='tight',dpi=300)
```

This illustrates that the matrices L and U returned by `lu` really are lower- and upper-triangular. P should be a permutation matrix, *i.e.* should consist of zeros except for one 1 in each row and column.

Other tricks: `subplot(abc)` is the c-th subplot in an array with a rows and b columns. The `bbox_inches` argument governs the thickness of white space around the sides of the figure.

11. Reading and Writing Files

Suppose you have code which produces a 50×50000 matrix `A` after an hour of computing time. You might want to save such an expensive object for future use! The following will save it to a file named MyFile:

```
import pickle
f = file('MyFile','w')
pickle.dump(A,f)
```

The following will extract your data from MyFile and store it as `A` again.

```
import pickle
f = file('MyFile','r')
A = pickle.load(f)
```

Yes, the module is named in analogy to preserving cucumbers, and the 'r' and 'w' arguments refer to reading and writing. If you want to save many objects, combine them in an array before you save:

```
a_s = array([a0,a1,a2,a3,a4,a5,a6])
pickle.dump(a_s, f)
...
a0,a1,a2,a3,a4,a5,a6 = pickle.load(f)
```

12. Time

Three methods for finding the sum of the first 999 integers, in order of decreasing cleverness, are:

- `T = 999*1000/2`
- `T = arange(1000).sum()`
- ```
  T = 0
  for j in arange(1000):  T+= j
  ```

We might want to see how fast these run. One way to do this for the third version is the following:

```
import timeit
tic = timeit.default_timer()
T = 0
for j in arange(1000): T+= j
toc = timeit.default_timer()
print toc - tic
```

Running this third version as a script (not from the command line) I get about 0.001 seconds over several trials. This is clock time, not CPU time, so it is affected by other programs your machine is running. Similarly the first version, where we compute 999*1000/2, takes about 5×10^{-6} or even 5×10^{-7} seconds, while the `sum` version takes $5 \cdot 10^{-5}$ seconds. Notice that, of the three, the `for` loop is by far the slowest. The relatively good performance of `sum` occurs because the real work is done off-stage, by the built-in command, which is heavily optimized. This is what is meant by *vectorizing* code. Raw Python code is relatively fast to write and slow to execute compared to compiled languages like C or Fortran; vectorized code can erase a lot of this execution disadvantage.

13. Strings

Python has a built-in class of string objects. These are not used heavily in computational work, but we mention some basic tools. If you type

```
a = 'Hello, my name is'
b = 'abcdefghijk'
```

both a and b are string instances. You can print them and concatenate them; `print a+b` gives `Hello, my name isabcdefghijk`. To print out the value of a variable, use the fancy syntax involving the percent sign:

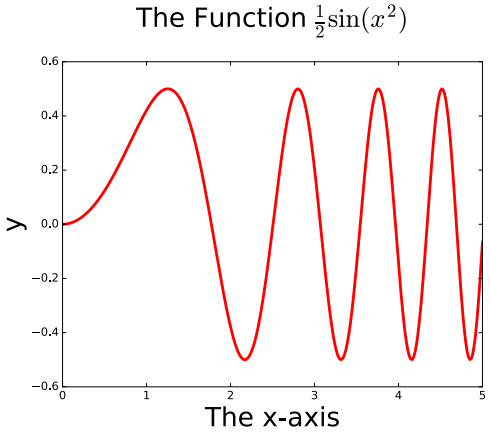
```
a = 12.4
print 'A reasonable number is %s, wouldn\'t you say?' % a
```

yields `A reasonable number is 12.4, wouldn't you say?` The 12.4 was substituted in for the %s. Note the backslash before the apostrophe in *wouldn't*; otherwise the apostrophe would be interpreted as ending the string. For several values:

```
print 'Two numbers are %s and %s, OK?' % (2,4.0)
```
```

I use strings most frequently to title plots and label axes. You can put LaTeX code in these, using $ signs. Example:

```
from scipy import *
from matplotlib import pyplot as plt
x = linspace(0,5,300)
y = sin(x**2) / 2
plt.clf()
plt.plot(x,y,'r-',linewidth = 3)
plt.title('The Function $\\frac{1}{2} \\sin(x^2)$\n',
plt.xlabel('The x-axis', fontsize = 28)
plt.ylabel('y', fontsize = 28)
plt.savefig('f5.pdf',bbox_inches = 'tight')
plt.show()
```



Here the double backslash \\ is necessary to get one backslash of text, and the \n signifies a new line of text (I had to do this to make the figure title taller in the plot, since otherwise the denominator of the fraction was partly cut off by the graph). There are many string tricks!

14. Movies

While matplotlib and mayavi have internal methods for making animations, I've found that it can be easier to save many still images and then use some other software to assemble the frames into a movie. One free possiblity is `mencoder`. A sample code is the following:

```
from pylab import *
from scipy.spatial import Delaunay
P = randn(30,2)
m = abs(P).max()
for j0 in arange(100):
    P+= 0.015*randn(size(P,0),size(P,1))
    P[:,0] += 0.004
    T = Delaunay(P).vertices
    figure(1)
    clf()
    axis("equal")
    P_n = P.copy()

    for j in arange(size(T,0)):
        P = P_n.copy()
        x = P[T[j,[0,1,2,0]],0]
        y = P[T[j,[0,1,2,0]],1]
        rx,ry = sum(P[T[j,:],0])/3,sum(P[T[j,:],1])/3
        plot(x,y,'b-')
        plot([rx,rx],[ry,ry],'r.')
        ds = sqrt(sum((P[T[j,:],0]-rx)**2+(P[T[j,:],1]-ry)**2))
        if j0%5 == 0: alf = 0.98*(ds>0.1)
        else: alf = 1.0
        P_n[T[j,:],0] = alf*P_n[T[j,:],0]+(1-alf)*rx
        P_n[T[j,:],1] = alf*P_n[T[j,:],1]+(1-alf)*ry
    xlim([-m,m])
```

```
    ylim([-m,m])

    savefig('tmp_%03d.png'%j0,bbox_inches = 'tight')

import os
os.system("mencoder 'mf://tmp_*.png' -mf type=png:fps=10 \
  -ovc lavc -lavcopts vcodec=wmv2 -oac copy -o animation.mpg")
```

It is only the last two lines that make the movie. Without the last two lines, this program will create 100 images named tmp_000.png, tmp_001.png, tmp_002.png, $\cdots$, tmp_099.png. The final command takes all png images in the current folder whose titles begin with "tmp_" and assembles them in alphabetical order into a movie named `animation.mpg` at 10 frames per second.

15. Object-Oriented Programming

Python is a heavily object-oriented language, so you should know a little about this topic even if you prefer to write functions for everything. If you didn't really understand the line

```
T = Delaunay(P).vertices
```

in the Exercise above, a bit of insight into OOP will be useful. To illustrate this, suppose we are writing programs that track the course of a party where various cakes are served to guests. To keep track of which slices of cake are available, it is efficient to define a new kind of object called a `cake`.

```
class cake:
    def __init__(self, n):
        assert type(n) == int
        self.num_slices = n
        self.unserved_slices = arange(n)

    def serve(self, sli):
        self.unserved_slices = setdiff1d(self.unserved_slices, sli)

    def first_unserved_slice(self):
        return array([self.unserved_slices.min()])

    def inspect(self):
        figure(1)
        clf()
        L = 2*pi/self.num_slices
        for j in self.unserved_slices:
            text(0.6*cos(L*(j+0.5)),0.6*sin(L*(j+0.5)),'%s'%j)
        axis('equal')
        xlim([-1,1])
        ylim([-1,1])
        show()
```

Each class definition must include an `__init__` function, which is run automatically whenever you create an instance of the class. It can also contain functions which can only be called using an instance of the class; these are caled *methods*. The code above defines the cake class and gives us three cake methods, `serve` and `first_unserved_slice` and `inspect`. Note that the initialization method checks to see that an integer is supplied as the number of slices and produces an assertion error otherwise. A given cake instance has two *attributes*, `num_slices` and `unserved_slices`. These consist of an integer and a vector that follow the cake instance around.

The class definition above can be part of our script if we don't intend to use it elsewhere. Another approach is to save it in a file called CakeFile.py and then load the definition where it is needed with the line `from CakeFile import cake`. We could then write the following to describe a scenario where two cakes are served:

```
C1, C2 = cake(8), cake(9)
C1.serve([0,4])
for j in arange(5): C1.serve(C1.first_unserved_slice())
print C1.unserved_slices
C2.serve([0,1,4,7])
C2.inspect()
```

At this party there were apparently two cakes, one divided into eight slices and the other into nine slices. The first and fourth slices of the first cake were the first to be eaten, and then five people took slices from the first cake. Next, the slices numbered 0, 1, 4, and 7 were taken from the second cake, and finally we get to see a plot of the remaining slices of the second cake.

In practice, writing your own class does not make sense unless you want to track something larger and more complicated than the desserts in this example. The syntax above should help you understand Python, however; for example, the main benefit of importing `numpy` is its very useful `array` class. Typing `A.max()` for the largest value of a matrix (an instance of the `array` class) is exactly analogous to finding the number of the first available slice of the cake C1 by calling `C1.first_unserved_slice()`. Returning to `Delaunay(P).vertices`, we can now say that `Delaunay` is a class imported from the `scipy.spatial` module, and each `Delaunay` instance has an array attribute called `vertices`.

16. Tips

Don't assume you will make no mistakes and that no one else will ever read your code. Assume you will make mistakes and assume that others will want to figure out what you've done. When you read others' code, or your own from last year, you can tell which attitude the writer adopted.

- COMMENT extensively. # makes the rest of the line into a comment, while ''' ... comment...''' can be used for multi-line comments.
- Make your assumptions explicit using `assert` statements. This helps you avoid errors and helps others (and you, months later) figure out how your code works.
- Test small pieces of your code separately before combining them.
- Don't optimize too early. Get a working version, then fine-tune just the parts that you want to run a million times.

17. Valediction

Your feet should now be wet! Have fun!