Lecture 3.2

Competence 2: Review

```
x = np.array([3,4,5]) # These are supposed to be the unit vectors
y = np.array([6,7,8]) # This does not represent the unit vectirs
z = np.array([1,2,9]) # That's the point.  To represent something physical in code.

def f(x,m):
    return m*x

guess = [2] # not guess = [2,3]!
            # i.e. one paramater function; therefore one parameter guess

popt,pcov = curve_fit(f, xs, ys, p0=guess)
m = popt[0] # best-fit value m comes from popt
sigma = np.sqrt(pcov[0,0]) # sigma is the sqrt of diagonal element(s) of the covariance matrix
```

Note that quad gives both answer and error:

```
ans,err = quad(f, 0, 1)
```

Assignment 2: Tips

This module is about representing something physical in code. There also has to be some slightly more abstract coding (A2Q2).

A2Q1

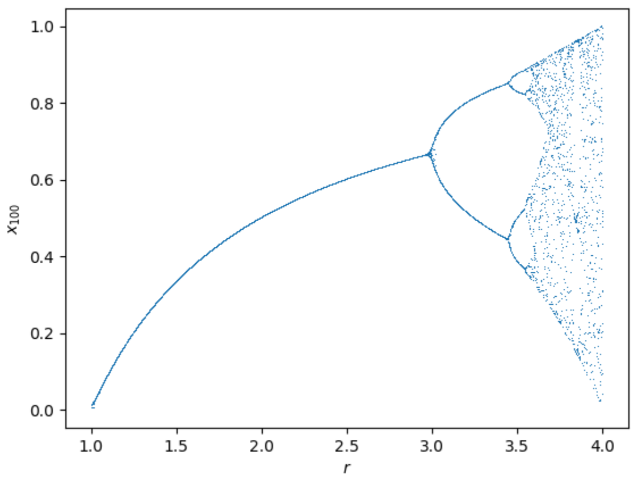
- I meant that vector v is at 45 degrees to both x and y.
- Point is to take something physical and represent it in code.
- Make variable names closely match the mathematics
- e.g. for velocity use `v = ...` rather than `a`; use `E` for electric field; etc.

A2Q3

- we want to make a function $g_n(x)$ i.e. `def g(n,x):`
- *inside this function* define a function to represent the integrand $f(\tau) = \cos(n\tau - x\sin(\tau))$
- calculate the integral using `ans,err = quad` and remember to return just `ans`

A2Q2

- Plot x_{100} as a function of r, therefore suggested starting with `def x100(r):`
- Inside this function,
  - pick a random value for x_0
  - calculate x_{100} for the given value or r
- Iterate over this using a for loop or list comprehension; it won't take an array in one go
- Result should look something like



A2Q4

- dealing with real experimental data; can check result for mass against known value
- interpolating function made with `interp1d` can be used with `quad`

A2Q5

- dealing with lots of experimental data is where programming is most valuable
- calculate temperature T and luminosity L for all the stars;
- then plot (approximately) a Hertzsprung–Russell diagram.
- this is intended to be authentic, not deliberately difficult.

Suggested structure for A2Q5:

```
Ts = list() # make an empty list; do the same for Luminosity and the uncertainty in T
for n in range(1,...): # loop over each star
    freq = data[:,0]    # load the frequency axis
    spec = data[:,n]    # load the spectrum for the n th star

    opt,cov = curve_fit(...)

    # extract T from opt and uncertainty in T from cov
    Ts.append(T) # and similarly for uncertainty in T

    # The most straightforward estimate of something proportional to luminosity
    # is the sum of all elements in the spectrum, since every data set has the
    # same number of points, and we only want something proportional.
    # Calculate this and append it to a list

    plt.plot(...) # plt data points and the best-fit curve

    plt.savefig("spectrum_{}.png".format(n)) # this makes an image of the plot
                                              # called "spectrum_1.png" etc
    plt.close("all") # this closes the plot so we start a new one next loop
```

To save a CSV file, us `np.savetxt` which expects a filename (e.g. "output.csv") and a 2D array. Make an empty 2D array using `out=np.zeros((N,M))` where `N,M` are the number of rows and columns. Populate this empty 2D array using slicing e.g. `out[:,0] = Ts`.
To plot a log-log scatter plot with error bars, do something like

```
plt.errorbar(x,y,xerr=...,fmt='.'); plt.xscale('log'); plt.yscale('log')
```

Functional: `lambda`, `map`, `filter`, `reduce` [Optional]

Some people may like `lambda`. This is discouraged by the BDFL but may make more sense to some:

```
f = lambda x: x**2
```

Read `lambda` as "function". i.e. `f` is a function of `x` which gives `x**2`.
"lambda" functions can be used in e.g. `quad` for integration, or `euler` for solving ODEs.
Also `map`, a "lazy" alternative to list comprehension.

```
ns = [1,2,3,4,5]
ms = [f(n) for n in ns] # explicit list
mg = map(f, ns)         # generator!
```

Also `filter`, to select elements according to a condition:

```
ns = range(1000)
ps = [n for n in ns if isprime(n)] # explicit list of primes < 1000
pg = filter(isprime, ns)           # generates primes < 1000 when asked
```

See also `reduce` from `functools` library. This was removed from in-builts.

```
from functools import reduce
reduce(lambda x,y: x+y, range(10))
```

```
45
```

This kind of "functional" programming can be really powerful, but is a little beyond the scope of an introductory course.

Dot product infix notation

Forgot to mention! Use `@` as dot product e.g.

```
import numpy as np
A = np.array([[1,2],[3,4]])
v = np.array([5,6])
A @ v
```

```
[17 39]
```

Linear algebra

Many linear algebral operations you know and love are available from `scipy.linalg`. Here's just a few:

```
from scipy.linalg import inv,det,eig,expm
```

```
inv(A) # matrix inverse of A, defined above
```

```
[[-2.   1. ]
 [ 1.5 -0.5]]
```

We cannot possibly cover all of `scipy`. If you encounter a numerical problem, there's probably a routine for it in `scipy` or `numpy`.

Solving ODEs

Reminder: generators are like functions, but have `yield` instead of `return` and make results when needed, e.g. a trivial generator which makes the squared numbers up to some specified maximum:

```
def gen(N):
    for n in range(N):
        yield n**2

a = gen(10)
next(a), next(a), next(a), next(a), next(a)
```

```
(0, 1, 4, 9, 16)
```

Euler's method for solving differential equations of the form $\dot{x} = f(t, x)$ makes the approximation

$$x_{n+1} \approx x_n + \delta t\, f(t_n, x_n).$$

We encode this as a generator:

```
def euler(f, x0, dt):
    xn = x0
    tn = 0
    while True:
        yield tn, xn
        xn = xn + f(tn, xn)*dt
        tn = tn + dt
```

An example: $\dot{x} = -0.1x + \sin(t)$ with $x(0) = 10$:

```
def f(t,x):
    from numpy import sin
    return -0.1*x + sin(t)

solver = euler(f, 10, 0.1)

ts = list()
xs = list()
for t,x in solver:
    ts.append(t)
    xs.append(x)
    if t > 10:
        break
```

Other methods exist, e.g. Leapfrog or 4th order Runge–Kutta (used in `projectile.py` from Assignment 1). These can be used as drop-in replacements for Euler:

```
solver = rk4(f, 10, 0.1)
```

Solving n-th order ODEs

1. We can solve $\dot{x} = f(t,x)$.
2. This works for n coupled first order ODEs.
3. We can convert any n^{th} order ODE into n coupled first order ODEs
    - Often called "Companion Form".
    - Companion Form is extremely useful; we'll only treat it briefly here.

Simple example: $\ddot{x} + a\dot{x} + bx = 0$ (the harmonic oscillator).
Introduce $v = \dot{x}$. Hence $\dot{v} + av + bx = 0$.
Rearrange so that anything with a dot is on the left hand side: $\dot{x} = v$ and $\dot{v} = -av - bx$.
Group x, v into one **state vector** $X = [x, v]$ which we represent as a `numpy.array`.
Then we can write as

$$\dot{X} = \begin{pmatrix} \dot{x} \\ \dot{v} \end{pmatrix} = \begin{pmatrix} v \\ -av - bx \end{pmatrix}.$$

This has the form $\dot{X} = f(t,X)$ and hence we identify $f(t,X) = [v, -av - bx]$.
Here's a complete implementation. It is very similar to the previous examples except the function `f` must now accept and return a `numpy.array` representing the state vector X:

```
def f(t,X):                  # function to embody the damped oscillator
    x,v = X
    xdot = v
    vdot = -a*v-b*x
    Xdot = np.array([xdot,vdot])
    return Xdot

a, b = 0.1, 1                # parameters of the damped oscillator
X0 = np.array([1,0])         # initial conditions: displaced, at rest
solver = euler(f,X0,0.01)    # glue it all togther

ts = []
Xs = []
for t,X in solver:
    ts.append(t)
    Xs.append(X)
    if t > 100:
        break

Xs = np.array(Xs) # convert the list of state vectors into a 2D array
Xs.shape          # it has a large number of rows and 2 columns
```

```
(10001, 2)
```

Procedure

1. Introduce variables to make into coupled first order ODEs
2. Group variables into one state-vector X.
3. Write function which computes **rate of change** for every element of X
4. Glue together with appropriate solver: e.g. `euler`, `rk4`, etc