Lecture 4.2

Overview

This lecture illustrates use of Python `class` to define an object. Everything in Python is an object, and `class` is how we define our own.

We will also meet `sympy`, for symbolic computing, some surprising behaviours of which make sense when seen as Python objects.

Objects have properties (values associated with them) and methods (functions associated with them).

For example,

```
import numpy as np
A = np.array([[1,2,3],[4,5,6]])
A.shape # a property; this is stored as part of the array object
A.mean() # a method; this is calculated based on the array contents
```

```
3.5
```

Please review the `Fork` example from last time.

Ising Model

The Ising model of a ferromagnet is a classic of computational physics. It is the simplest model to exhibit a phase transition, and phase transitions are important all across physics: from down-to-earth solids and liquids, to Cosmology and Quantum Chromodynamics.

Consider a 2D grid. At each site there is a "spin" which can be either up + or down -.

Aligned spins e.g. ++ have a lower-energy than anti-aligned e.g. +-.

The energy associated with a given site is the sum of this interaction energy with all 4 nearest neighbours: up, down, left, right.

Thermal fluctuations induce random spin flips. At high temperature, the thermal energy is far higher than the interaction energy, and spins flip randomly, with little dependence on their neighbours. At low temperature, we should generally expect neighbouring spins to align. When cooling quickly, domains will form and they will be frozen in place.

For even a tiny grid it is impractical to evaluate the total energy for every possible combination of spins: there are $2^{N\times N}$ combinations, which is $\sim 10^{30}$ for $N = 10$. The world's fastest supercomputer can do around 10^{17} operations per second (a single CPU can do around 10^9), so optimistically this would take 400 thousand years.

Metropolis is a class of algorithms for exploring this kind of impractically large parameter space.

For the Ising model, we have the **Metropolis single spin-flip algorithm**. There are other, much faster algorithms. Whether an algorithm captures the behaviour you're interested in is an interesting branch of theoretical physics.

The **Metropolis single spin-flip algorithm** is:

1. pick a site at random
2. compute the energy change ΔE if this spin were to flip
3. accept the change if
   (a) it would reduce the energy
   (b) with probability $e^{-\Delta E/k_BT}$ if it increases the energy

We parameterise the coupling and the temperature as a single parameter J where $J = J_{nn}/k_BT$ and

$$\Delta E_{i,j} = -J_{nn}\sum_{nn} s_{i,j}s_{nn}$$

where i, j is the index of the particular spin and the summation is taken over nearest neighbours only.

We use **periodic boundary conditions** to give meaning to the notion of nearest neighbour for sites on the edge of the grid. This is usually a sensible choice for emulating an infinite system on a finite grid.

It is convenient to encode this behaviour as a Python object. This allows us to consider different specific *instances* of this general model. Here is the class definition. A full working example is attached to this PDF and available on the course website.

```
class IsingModel:
    def __init__(self, N, J):
        self.N = N
        self.J = J
        self.state = (np.random.random((N,N)) < 0.5) * 2 - 1

    def __str__(self):
        return 'Ising model with {}x{} sites'.format(self.N,self.N)

    def __repr__(self):
        return "IsingModel({}, {})".format(self.N, self.J)

    def mag(self):
        return self.state.sum()/(self.N*self.N)

    def site_energy(self,n,m):
        site  = self.state[n,m]
        left  = self.state[n-1,m]
        right = self.state[(n+1)-self.N,m]
        up    = self.state[n,m-1]
        down  = self.state[n,(m+1)-self.N]
        return -self.J*site*(left+right+up+down)

    def flip(self, n, m):
        energy_before = self.site_energy(n,m)
        self.state[n,m] *= -1
        energy_after = self.site_energy(n,m)
        self.state[n,m] *= -1
        energy_change = energy_after - energy_before
        if energy_change < 0:
            self.state[n,m] *= -1
        else:
            if np.random.random() < np.exp(-energy_change):
                self.state[n,m] *= -1

    def iterate(self, iterations=1):
        for i in range(iterations):
            n,m = randint(0,self.N-1),randint(0,self.N-1)
            self.flip(n,m)
```

The method `site_energy` exploits negative indexing of the `numpy.array` to implement periodic boundary conditions.

The method `flip` implements the Metropolis algorithm for a given site; it could be shortened a little. `iterate` uses `flip` a number of times.

We can then make an instance of this object, iterate it using the in-built method, and plot the result using e.g. `plt.pcolormesh`:

```
thing = IsingModel(100,0.5)
thing.iterate(1000000)
```

It takes a long time to reach equilibrium, even for modest sized grids. There is a critical coupling constant J_C above which there is spontaneous magnetisation: a single domain forms and all spins point in the same direction. To find this is computationally intensive, so we use `multiprocessing` to consider different values of J in parallel. We do this by writing a function which takes in J, makes an instance of `IsingModel`, iterates it, and returns the result. We then compute the magnetisation for each of these iterated Ising models. See the attached for a full, working example.

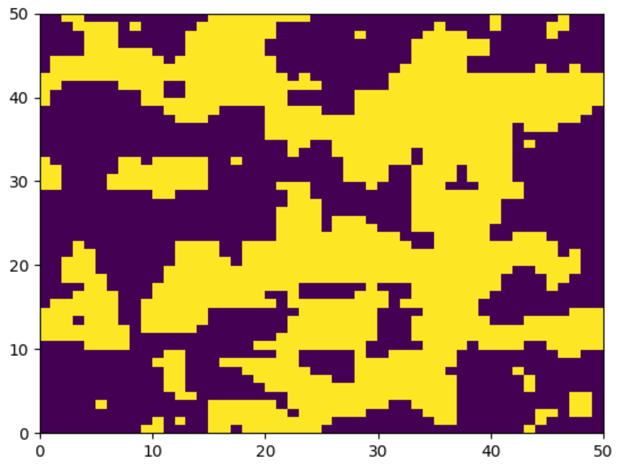
```
def make_and_iterate(J):
    A = IsingModel(30, J)
    A.iterate(1000000)
    return A

if __name__=='__main__':
    Js = np.linspace(0.1,0.8,32)
    with Pool() as p:
        As = p.map(make_and_iterate, Js)
    mag = [abs(A.mag()) for A in As]
```

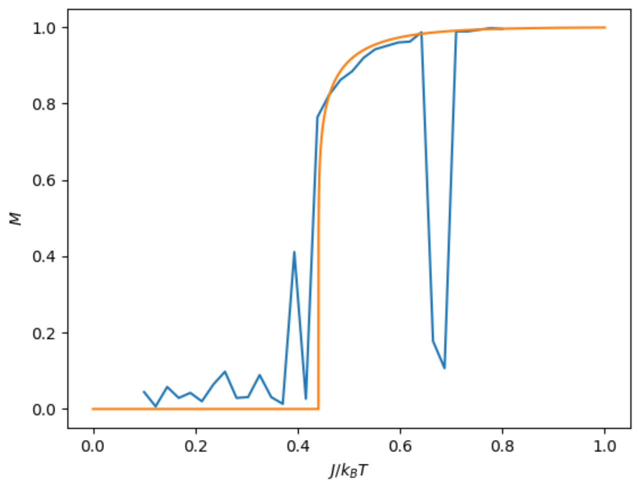
The analytical "Onsager" solution is a famous result from statistical physics. Critical coupling J_C and magnetisation M are:

$$J_C = \frac{\ln(1+\sqrt{2})}{2}\,; M(J) = \left(1 - \sinh(2J)^{-4}\right)^{1/8}.$$

Typical intermediate state of Ising model below critical temperature



Magnetisation with coupling constant. Note that for one model it appears two large domains have formed, giving net magnetisation of approximately zero.



Symbolic computing: algebra

We now meet **Symbolic** computing: algebra, calculus, differential equations, etc. Official tutorial

Some of this is very different to how we've used Python so far.

(Tutorials often import all of the library into the current namespace i.e. `from sympy import *`. I find this messy and will instead work exactly analogously to how we've used `numpy`.)

```
import sympy as sp
x,y,z = sp.symbols('x y z')
```

`x,y,z` are now sympy objects. This means, amongst other things, that the human-readable name of the symbol is built by the `__str__` method. A consequence of this is that if we do `a=x` then `a` **is** `x` i.e. they point to the same Python object, and by typing `a` we get the result `x`, which is perhaps not what we would want but makes sense when we understand how Python handles objects.

You may wish to use the following to have a more human-readable display of mathematics:

`sp.init_printing(use_unicode=True)`

Let's define `a,b,c` and then define an *expression* to represent a quadratic:

```
a,b,c = sp.symbols('a b c')
f = a*x**2 + b*x + c
```

We can *solve* this quadratic for x:

```
sp.solve(f,x)
```

```
[(-b + sqrt(-4*a*c + b**2))/(2*a), -(b + sqrt(-4*a*c + b**2))/(2*a)]
```

There are two solutions, and these are stored in a Python list.

Although this only reproduces a result you will all recognise, I hope you realise the power of this approach. Now, if you see **any** expression where the quantitity you want appears as x^2, x, and a constant, then you can use `sympy` to solve it, almost regardless of how messy the expression is. You can spend your time thinking about more abstract problems rather than grinding through the mechanical process of solving an equation!

Equality checking also behaves Pythonically rather than mathematically:

```
f = (x-1)**2
g = x**2 - 2*x + 1
```

We know these are different representations of the same mathematical expression, but Python just sees that they're different:

```
f == g
```

```
False
```

We can test whether they are mathematically equal by taking the difference and seeing whether that *simplifies* to zero:

```
sp.simplify(f-g)
```

```
0
```

The `simplify` function is extremely useful!

Symbolic computing: calculus

sympy also does calculus. Tutorial
Differentiation:

```
f = x**2 + 2*x
f.diff(x)
```

```
2*x + 2
```

Integration:

```
f.integrate(x)
```

```
x**3/3 + x**2
```

Definite integrals:

```
f.integrate((x,0,1))
```

```
4/3
```

Plugging in numbers to expressions is acomplished with the subs method. Pass a dict containing the replacement rules e.g. for $x \to 2$:

```
f.subs({x: 2})
```

```
8
```

Operations can be chained together like so

```
sp.sin(x).diff(x)
```

```
cos(x)
```

You can take limits:

```
sp.limit(sp.sin(x)/x,x,0)
```

```
1
```

and do series expansion:

```
f = sp.exp(x)
f.series(x,0,4)
```

```
1 + x + x**2/2 + x**3/6 + O(x**4)
```

Next time

We will treat complex numbers and differential equations next time, as well as a brief correspondence with Mathematica.