Contents

1 Simple Visualisation

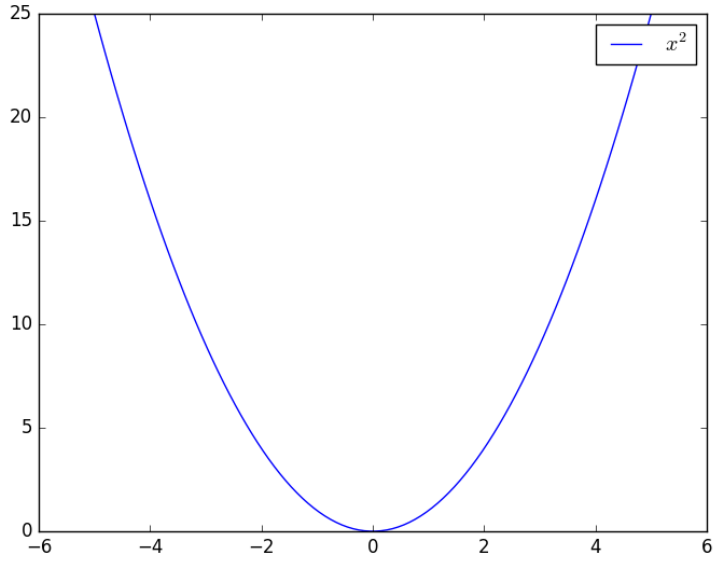
1.1 matplotlib

- The `matplotlib` python package is terribly good but cannot do Big Data as it is **not** distributed
  - has extensive documentation at matplotlib homepage
- It's also not properly parallel so it can often be slow
- But it is
  - easy
  - interactive
  - if you only need to plot a subset of your data (e.g. 2D slice of 3D data) it might scale well enough
- please note that interactivity over the network will be laggy; we show how it works anyway
- the following "ipython magic" is only needed to embed the output in the ipython/jupyter notebook
  - it needs to be done *once* per python session, so please always execute this cell even if you only want to look at a single later example

```
%matplotlib notebook
```

1.1.1 A Simple Example: a parabola

```
import pylab, numpy
x = numpy.mgrid[-5:5:100j]
pylab.plot(x, x**2, "b-", label=r"$x^2$")
pylab.legend()
```



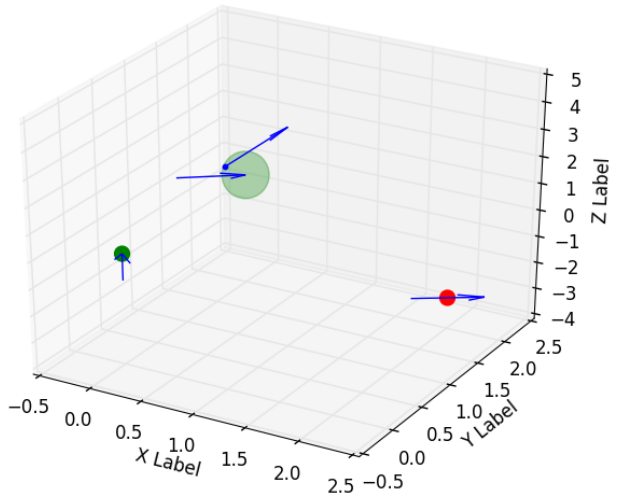
1.1.2 Plotting a Saved File: a simple 3D example

- in this example we use the file we created earlier: `files/genfromtxt_example_data.txt` and save it to another called `files/genfromtxt_example_data.png`

```
infile = "files/genfromtxt_example_data.txt"
oufile = "files/genfromtxt_example_plot.png"
import numpy
import matplotlib
import matplotlib.pyplot
from mpl_toolkits.mplot3d import Axes3D

def randrange(n, vmin, vmax):
    return (vmax - vmin)*numpy.random.rand(n) + vmin
```

```
data = numpy.genfromtxt(infile, comments="#", delimiter="\t", skip_header=3)
fig = matplotlib.pyplot.figure()
ax = fig.add_subplot(111, projection='3d')
n = data.shape[0]
# plot a sphere for each particle
# colour charged particles red (charge>0), blue (charge<0) and neutrals green
blues = data[data[:,7]<0]
reds = data[data[:,7]>0]
greens=data[numpy.logical_not(numpy.logical_or(data[:,7]<0,data[:,7]>0))]
ax.scatter(blues[:,0], blues[:,1], blues[:,2], c="b", edgecolors="face",
           marker="o", s=blues[:,6])
ax.scatter(reds[:,0], reds[:,1], reds[:,2], c="r", edgecolors="face",
           marker="o", s=greens[:,6])
ax.scatter(greens[:,0], greens[:,1], greens[:,2], c="g", edgecolors="face",
           marker="o", s=greens[:,6])
ax.quiver(blues[:,0], blues[:,1], blues[:,2], blues[:,3], blues[:,4],
          blues[:,5], pivot="tail")
ax.quiver(reds[:,0], reds[:,1], reds[:,2], reds[:,3], reds[:,4],
          reds[:,5], pivot="middle")
ax.quiver(greens[:,0], greens[:,1], greens[:,2], greens[:,3], greens[:,4],
          greens[:,5], pivot="tip")
ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')
matplotlib.pyplot.savefig(oufile)
print(oufile, end="")
```



1.1.3 Advanced Features

- we did not use anything advanced, except matplotlib's builtin latex capability, but it provides a full control of the whole canvas and image window

1.1.4 Animation Using matplotlib

- matplotlib has a rudimentary animation capability as well
  - ParaView is better in this, and matplotlib will not be able to create beautiful complex animations
  - but it can do simple ones
  - and it can be used to generate lots of frames for a video
    - but unless you use matplotlib-frontend specific, just using file-write backend directly, without plotting on screen is much faster
    - in both cases you can convert to video like

```
ffmpeg -f image2 -pattern_type glob -framerate 25 -i\
 'testanimationsaveframe_*.png' -s 800x600 foo.mkv
```

  - or illustrate how an algorithm works, see exercises!

- here's an example with all the important bits:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
plt.ion()

def data_gen(t=0):
    '''A generator function, which must use "yield" as all generators do,
    to produce results one frame at a time. In this example, the "run"
    function will actually remember/save data for previous frames so
    we get away with generating just the new data. Whatever we return
    will be passed as the sole argument to "run".'''
    cnt = 0
    while cnt < 1000:
        cnt += 1
        t += 0.1
        yield t, np.sin(2*np.pi*t) * np.exp(-t/10.)

def init():
    '''A setup function, called before the animation begins.'''
    ax.set_ylim(-1.1, 1.1)
    ax.set_xlim(0, 100)
    del xdata[:]
    del ydata[:]
    line.set_data(xdata, ydata)
    return line,

fig, ax = plt.subplots()
line, = ax.plot([], [], lw=2)
ax.grid()
xdata, ydata = [], []

def run(data, args):
    '''This is called by the animator for each frame with new data from
    "data_gen" each time. What we do here is up to us: we could even
    write the plot to disc (see the commented-out line) or we could do
    something completely unrelated to matplotlib!  The present code
```

```
    will append new data to its old (global variable) data and
    generate a new animation frame. Note that matplotlib holds a copy
    of our old data so we could fish it out from the depths of its
    internal representation and append to that but that's a bit
    complicated for our example here.  We have been passed "args" but
    we ignore that.'''
    t, y = data
    xdata.append(t)
    ydata.append(y)
    xmin, xmax = ax.get_xlim()
    if t >= xmax:
        ax.set_xlim(xmin, 2*xmax)
        ax.figure.canvas.draw()
    line.set_data(xdata, ydata)
    return line,

ani = animation.FuncAnimation(fig, run, data_gen, blit=False, interval=10,
                              fargs=("arguments",), repeat=False, init_func=init)
plt.show()
```

1.2 Exercise

Use your Game of Life from earlier on and animate it using `FuncAnimation`. You have already written the stepper in such a way that it is easy to wrap into a small "run" function which generates frames one at a time. Hint: easiest way to plot is probably matplotlib's `imshow` function.

1.2.1 Solution

Available in the repo.

2 Parallel Visualisation: ParaView (very quick intro)

- ParaView, as the name suggests, runs in (distributed) parallel: no data is too big if you managed to create it in the first place
- Some complications in getting the proper distributed parallel version up and running:
  - ParaView is split into a client and a server

- normal `paraview` command runs client with a local server, but not in parallel
- not what you want anyway: you can run ParaView this way on your supercomputer, but the UI will be **very** slow as all plotting data and interaction need to go over the network
- you need to run `pvserver` on the "big" machine and connect `paraview` frontend to that
- so far so simple, but `paraview` needs to be able to connect to `pvserver` and this is usually blocked by a firewall
- need to punch a hole to the firewall in a three step process:
  * `ssh` to the machine you want to run `pvserver` on and start `pvserver`
  * from `pvserver`'s output, find the port it listens on (`Connection URL:`); it will look like `cs://vega:11111`
  * now punch a hole in firewall with `ssh -NL 11111:localhost:PortYouJustFound PVServerMachineName`
  * then start `paraview` locally and connect it to the local server at port `11111`
  * not really paraview's fault here: blame the criminals whose activities enforce everyone to firewall off their computers

2.1 A simple example with HDF5 without remote `pvserver`

- first we write the HDF5 file using `h5py`, one of the many python HDF5 interfaces

```
import numpy
import tempfile
import h5py
file=tempfile.NamedTemporaryFile(
    dir="files/",
    prefix="hdf5_visualisation_example",
    suffix=".h5",
    delete=False)
file.close()
xmin, xmax, ymin, ymax, zmin, zmax = -5,+5,-5,+5,-5,+5
xpts, ypts, zpts = 101, 101, 101
cutoff1, cutoff2, cutoff3 = 1.0, 3.0, 4.0
```

```
dsname="mydataset"
m = numpy.mgrid[xmin:xmax:xpts*1j,ymin:ymax:ypts*1j,zmin:zmax:xpts*1j]
r = (m**2).sum(axis=0)**0.5
mydata = cutoff2/(cutoff2-cutoff3)**2*(r-cutoff3)**2
mydata[r<cutoff2] = r[r<cutoff2]
mydata[r<cutoff1] = 0.0
mydata[r>cutoff3] = 0.0
h5file = h5py.File(file.name,"w")
h5file.create_dataset(dsname, data=mydata)
print("Wrote data to file {f}.".format(f=file.name))
```

- note that we did not close the file yet
- but HDF5 is too generic for paraview to have a generic import module, we need to tell paraview what the HDF5 file looks like
  - do not ask me why this information cannot be in the file itself
- note the numpy-like access to the dataset in the HDF5 file

```
str="""<?xml version="1.0" ?>
<!DOCTYPE Xdmf SYSTEM "Xdmf.dtd" []>
<Xdmf Version="2.0">
  <Domain>
    <Grid Name="{meshname}" GridType="Uniform">
      <Topology TopologyType="3DCoRectMesh" NumberOfElements="{Nx} {Ny} {Nz}"/>
       <Geometry GeometryType="ORIGIN_DXDYDZ">
        <DataItem DataType="Float" Dimensions="3" Format="XML">
          {xmin} {ymin} {zmin}
        </DataItem>
        <DataItem DataType="Float" Dimensions="3" Format="XML">
          {dx} {dy} {dz}
        </DataItem>
      </Geometry>
      <Attribute Name="mydata" AttributeType="Scalar" Center="Node">
        <DataItem Dimensions="{Nx} {Ny} {Nz}" NumberType="Float"
         Precision="{precision}" Format="HDF">
          {filename}:/{datasetname}
        </DataItem>
      </Attribute>
    </Grid>
```

```
  </Domain>
</Xdmf>
""".format(meshname="mymesh",
           Nx=h5file[dsname].shape[0], Ny=h5file[dsname].shape[1],
           Nz=h5file[dsname].shape[2],
           xmin=xmin, ymin=ymin, zmin=zmin,
           dx=(xmax-xmin)*1.0/(xpts-1), dy=(ymax-ymin)*1.0/(ypts-1),
           dz=(zmax-zmin)*1.0/(zpts-1),
           precision=h5file[dsname].dtype.itemsize,
           filename=h5file.filename,
           datasetname=dsname)
xdmffilen=h5file.filename.replace(".h5",".xdmf")
xdmffile=open(xdmffilen,"w")
xdmffile.write(str)
xdmffile.close()
h5file.close()
```

- now to paraview which we unfortunately cannot do in Jupyter
  - that's a lie! we could, but we would need to install ParaView first and it would be terribly slow to use and would just open another window like we are doing now