Simulating Language: some notes on graphs

When you run a simulation, it's really useful to be able to visualise the results, and finding the right visualisation can sometimes make the difference between understanding what's going on in a particular model and being completely clueless. Plotting graphs is quite easy in python: you have all already plotted various graphs of simulation results, and you will be plotting graphs as part of your assessment for this course. This is a quick tutorial on making those graphs a bit easier to read, by adding axis labels, legends etc. Then there are some notes on adding plotting commands into your code (instead of doing it at the prompt), and some final comments on **what** you should plot.

Nicer plots

It's very easy to quickly plot a graph in python, and the code you are working with is designed so that it generates nice, easily-plottable lists. But the basic graphs you get aren't always exactly what you want - python doesn't label the axes for you, and chooses a different colour for every line when this might not be what you want. You can get some ideas about what is possible by googling for "matplotlib" and looking at example graphs. But I have included some basic thoughts below, plotting results from the simulation in evolution1.py. I am also working at the prompt initially, then later I'll show you how to get your code to plot these graphs for you automatically.

One of the things we talked about in class and in labs was replicating Oliphant's result that communication evolves when there is mutual benefit to hearer and speaker, but doesn't when only the hearer benefits. We can get that result using our code, and plot it quite nicely. First, I want to do some simulation runs in each condition, and build up a list of results:

```
>>> n_runs_per_condition=5
>>> mutual_benefit_results=[]
>>> for r in range(n_runs_per_condition):
     this_pop,this_pop_fitness=simulation(2000)
     mutual_benefit_results.append(this_pop_fitness)


>>> send_weighting=0
>>> receive_weighting=20
>>> receiver_benefit_results=[]
>>> for r in range(n_runs_per_condition):
     this_pop,this_pop_fitness=simulation(2000)
     receiver_benefit_results.append(this_pop_fitness)
```

Now I have my results, I would like to plot them. It's quite fiddly to get nice interactive plotting at the prompt in IDLE, but it is possible with matplotlib's interactive mode, and a somewhat obscure "pause" function to help the plotting catch up. Here's one attempt to plot the results this way:

```
>>> import matplotlib.pyplot as plt
>>> plt.ion()
>>> for r in range(n_runs_per_condition):
     plt.plot(mutual_benefit_results[r])
     plt.pause(1)
     plt.plot(receiver_benefit_results[r])
     plt.pause(1)
```

Try that out - as you can see, it's not very illuminating! There are a bunch of lines there, but I have no idea which lines belong to which condition, so I can't see what's going on. Also I can't tell by looking at the graph what this is - is it the results of a run of a genetic algorithm, or the results of a monte carlo communicative accuracy assessment, or what?

The first thing to do is to plot the two conditions using two different colours. I have 5 runs per condition, but I don't actually care about the details of individual runs, just the overall patterns, so in this case I think it's OK (and in fact helpful) to use the same colour for each simulation run in a given condition. I am also going to put this plot in a new window, so I can compare it to my old plot - I open a new window using `figure()`.

```
>>> plt.figure()
>>> for r in range(n_runs_per_condition):
     plt.plot(mutual_benefit_results[r],color='b')
     plt.pause(1)
     plt.plot(receiver_benefit_results[r],color='r')
     plt.pause(1)
```

That's better, now I can see that the blue lines, the "mutual benefit" runs, are fairly reliably hitting the maximal value of communicative accuracy (21) whereas the red lines, the "receiver benefit" lines, aren't. I can also see, if I peer very closely, that the lines are drawn on top of each other - that is OK here since I can still see what is going on, but it can be problematic if you have one set of runs that are completely obscured by something else plotted on top - watch out for that, and if necessary change the order in which you add the lines to the figure.

This is already much better, but I'd really like to add labels to the axes. It would also be nice to rescale the plot so that it goes from the minimum value of communicative accuracy (which I know is 1) to the maximum (which I know is 21). That's easy enough to do:

```
>>> plt.xlabel("Generations")
>>> plt.ylabel("Fitness")
>>> plt.ylim(1,21)
>>> plt.pause(1)
```

Actually I'm not sure I like way it looks with the y-axis going all the way down to 1 - while it conveys some extra information (even in the receiver benefit condition communicative accuracy isn't as low as it could be), I think there is too much whitespace on the graph. So I will tweak it,

using the `ylim` command, till I am happy - for my data, it looks OK with the y-axis running from 6 to 21.

Now I would like to set the values on the x- and y-axis to something sensible - the default values are OK, but I don't need that much information on generation number, and I don't like the way the graph looks with the y-axis ending at an unlabelled point. So I will change the tick marks on the x and y axis to more sensible values, using `xticks` and `yticks` - I specify a list of values, and that's where the tick marks will be.

```
>>> plt.xticks([0,500,1000,1500,2000])
>>> plt.yticks([6,11,16,21])
>>> plt.pause(1)
```

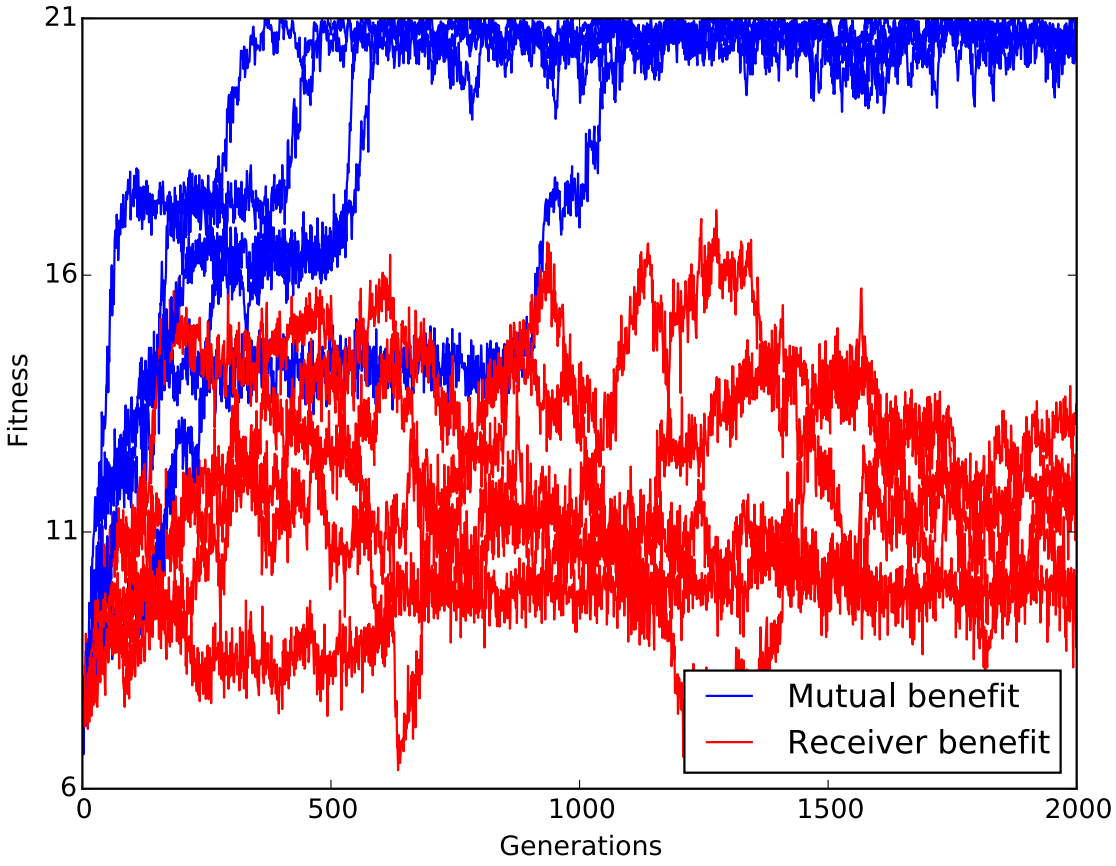
Now I am pretty happy with it - I can see the difference between the two conditions, it's nice and clear what each axis stands for, and the tick marks are in sensible places. The only thing I don't like is that there is no legend - unless I know that blue is the mutual benefit condition and red is the receiver benefit condition, I can't interpret the graph. I'd like that information in there too. It's actually quite easy to do legends in python, but it involves plotting the data in a slightly different way, so I'll do that in a new figure, add in the labels and tick marks and stuff again, then explain how the legend works.

```
>>> plt.figure()
>>> for r in range(n_runs_per_condition):
     plt.plot(mutual_benefit_results[r],color='b',
              label="Mutual benefit" if r == 0 else "")
     plt.pause(1)
     plt.plot(receiver_benefit_results[r],color='r',
              label="Receiver benefit" if r == 0 else "")
     plt.pause(1)

>>> plt.xlabel("Generations")
>>> plt.ylabel("Fitness")
>>> plt.ylim(6,21)
>>> plt.xticks([0,500,1000,1500,2000])
>>> plt.yticks([6,11,16,21])
>>> plt.legend(loc=4)
>>> plt.savefig("my_figure.pdf")
```

You will notice that when I plot my results, I have added another argument to the plot command, like this: `label="Mutual benefit" if r == 0 else ""`). The `label` argument associates a label with each line - then, later on when I use the `legend` command, it adds a legend for each line with a label attached. But I have 5 sets of results in each condition, and I don't need a label for each one, so I use a little extra bit of `if ... else` syntax to make sure that only the first line in each condition gets a label (and the rest get empty labels) - that's what the `if r == 0 else ""` does (remember I am calling this in a for loop for `r=0` to `r=4`, where r is the run number). Then right at the end I call `legend` to add the legend. `loc=4` sticks the legend in the

bottom right corner - for my results that looks OK, but if you find the legend is on top of some of the data, put it somewhere else. And now I have a figure I am reasonably happy with, so I save it using the save function. Here it is:



Plotting directly from your code

Producing figures by typing stuff at the prompt is fine, and it makes it easy to tweak things so you are happy with how they look. But sometimes you want to add some code to your .py file to plot the figure for you immediately, for instance if you are doing a long simulation run and want a figure but don't want to sit looking at the prompt until the results are ready. You can of course add plotting commands to your code (i.e. to the .py file you are using), but there are some differences:

1. We don't switch into interactive mode, so all the plotting gets created in the background before being shown, and we don't need to use the weird "pause" trick.
2. We always have to use figure() to create a new figure.
3. Once the figure is ready, you have to explicitly state whether it should be shown or saved - if you don't do this, the figure will be invisible, which is not a good feature for a figure to have.

If you add the following text to the evolution1.py you should (eventually) get a nice PDF figure.

```
import matplotlib.pyplot as plt #load plotting library, call it plt for short

n_runs_per_condition = 5

send_weighting = 10    # weighting factor for send score
receive_weighting = 10 # weighting factor for receive score
mutual_benefit_results = []
for r in range(n_runs_per_condition):
    this_pop,this_pop_fitness = simulation(2000)
    mutual_benefit_results.append(this_pop_fitness)

send_weighting = 0
receive_weighting = 20
receiver_benefit_results = []
for r in range(n_runs_per_condition):
    this_pop,this_pop_fitness = simulation(2000)
    receiver_benefit_results.append(this_pop_fitness)

#now for plotting
plt.figure()
for r in range(n_runs_per_condition):
    plt.plot(mutual_benefit_results[r],color='b',
       label="Mutual benefit" if r == 0 else "")
    plt.plot(receiver_benefit_results[r],color='r',
       label="Receiver benefit" if r == 0 else "")
plt.xlabel("Generations")
plt.ylabel("Fitness")
plt.ylim(6,21)
plt.xticks([0,500,1000,1500,2000])
plt.yticks([6,11,16,21])
plt.legend(loc=4)
plt.savefig("my_figure.pdf") #this saves the figure as pdf
#plt.show() #this would show the figure instead of saving it
```

What should I plot?

It's very tempting just to plot whatever output the code gives you - our code tends to produce trial-by-trial or generation-by-generation data, so we are plotting lots of timecourses. But depending on the message you are trying to convey, maybe timecourses aren't required. For example, in cases where you don't actually care about how the system evolves, but only about where it ends up, it might be more informative to plot the endpoint of a bunch of simulations, rather than a set of timecourses - we could reduce each simulation run to a single number, the average of the last 50 generations say, then plot those. That's what this code will do (if you add it in at the end of evolution1.py, after the code above):

```
plt.figure()
#cheeky use of [1]*5 and [2]*5 to generate x values of appropriate length
plt.plot([1]*5,mutual_benefit_averages,'bo',label="Mutual benefit")
plt.plot([2]*5,receiver_benefit_averages,'ro',label="Receiver benefit")
plt.xlim(0.5,2.5)
plt.ylim(6,21.5)
plt.xticks([]) #don;t want any ticks or labels on the x axis
plt.ylabel("Fitness (average of last 50 generations)")
plt.yticks([6,11,16,21])
plt.legend(loc=1,numpoints=1) #see what it looks like without numpoints=1
plt.savefig("my_next_figure.pdf")
```

This is what that produces - it's probably a bit excessive in terms of whitespace, maybe I should have made my points bigger, but you can't deny it's simpler than the previous figure! In general, think about what makes sense for what you're trying to convey with your data. For example, if we had lots of runs, perhaps a histogram of fitness values would tell us even more? If you look on the matplotlib homepage you'll find loads and loads of details on how to plot different types of graphs.