AST 4930/7939: Computational Astrophysics **Desika Narayanan**

Project 1: Cryptography

All work is due Tuesday Nov 7th at the beginning of class. Please name the assignment 'lastname_project1.py' and use all lowercase letters. Please also include lastname_project1.pdf as your write up. The project is to be turned in by pushing it to your bitbucket repository. Please also bring a printed copy of your source code to class, alongside your pdf of your project report: if you don't, then you will not receive written record of where points were taken off.

1 Executive Summary

I'm going to give you a random jumbling of the phrase: JACK AND JILL WENT UP THE HILL TO FETCH A PAIL OF WATER (all lower case). You are to develop a Markov Chain Monte Carlo technique to decrypt this message.

The basic plan will be to read in a very long text (*War and Peace*), and use the ordering of the letters in that text to find out how often particular letters in the English language follow one another.

Then, like the traveling salesperson problem, propose a solution, test the liklihood with the above in mind, and use that information to decide whether or not to accept.

2 Details

1. First, familiarize yourself with the data type 'dictionaries'. These will be critical to your success in this project. Do not skip this step. You can find some information here:
   - https://docs.python.org/3/tutorial/datastructures.html#dictionaries
   - https://www.python-course.eu/dictionaries.php
2. Second: familiarize yourself with np.savez and np.load
   - https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.savez.html
   - https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.load.html
3. Now, from the class respository, download wp.npz: This is the text for *War and Peace* in npz file format. You could, for example, open it via:

```
import numpy as np
data = np.load('wp.npz')
text = data['text']
```

4. Create a probability matrix that's 26×26 that represents a-z on both axes. Fill this with the liklihood that one letter is followed by another sequentially. (You're welcome to do 27×27 if you want to include spaces). Note, there are special characters in the text – you can just pass over those.

5. Generate a random solution key where every letter maps to another randomly generated letter. You might find

```
import random
x = list_of_letters
random.shuffle(x)
```

   or some similar code to be useful.

6. Calculate the value P from this sequence of letters from the probability matrix. There are a number of possible ways to do this. You could sum the values; take their product; work in log space, etc. You'll need to experiment to figure out what works best for you.

7. Choose two indices in your solution key to swap at random, and swap them.

8. Evaluate if the probability from this proposed solution is better or worse than the previous probability. If yes, accept. If no, accept under some condition.

9. Continue on like this for some pre-determined number of tries.

10. At the end, calculate the percentage of correct letters in your solution phrase (at the right position) compared to the true phrase. How does this compare to if you were to randomly draw a letter for each position?

11. Calculate the rate at which you approach your converged solution (i.e. you aren't getting any *better* of an answer) as a function of iteration number for each of the simulation variations you attempt (below).

12. Your jumbled phrase is: **zywdynfzmbboanxjrxiaimbbxpgaxwiyrymbpgoyxal**. This said, you are welcome to create your own jumbled phrase if you'd like. The way I did it was to do something like:

```
def generate_key(list_of_letters):
#dictionary mapping random letter to true letter
    x = list_of_letters
    random.shuffle(x) #now the orders are different.
the

    key = dict(zip(x,string.ascii_lowercase))
    reverse_key =  dict(zip(string.ascii_lowercase, x))

    return key,reverse_key


def generate_random_from_phrase(phrase,reverse_key):

    jumbled_phrase = ""

    for i in range(len(phrase)):
    jumbled_phrase += reverse_key[phrase[i]]
    return jumbled_phrase
```

Note, this is different than our previous homeworks in that it's a *project*. Your goal is to build the best cryptograph that you can. Note, you probably won't get to 100% accuracy. You might do much

worse. You aren't being graded on how well this performs, but rather how well you've done in getting the code to perform at it's maximum potential. In other words, what's important is the *scientific investigation* part, as you learn what works and what doesn't. I want you to invest significant effort in understanding how you can make your MCMC cryptograph the best that it can be.

You should test the effects of:

- Annealing (with different annealing rates)
- Only accepting positive proposals
- Varying the probability by which you accept negative proposals
- Varying how you calculate the Probability matrix of the solution key that you're trying.

Of course there are other avenues you might test, but at a minimum test these.

3 To Turn in

The turn in for this project will be different than homeworks. I want you to turn in:

- Your code
- A written up project report

The latter is the more important part. It won't be possible (nor, necessary) for your code to demonstrate in one single code all of the tests that you did. That's totally fine. What I want is to be able to tell from the code *roughly* what you did – the real meat is in the report.

In the report, include the following (with contribution to total grade next to it):

- A description of your code architecture (25%)
- A description of how you tested the aforementioned parameters. (25%)
- Diagnostic plots of the impact of varying each of the aforementioned parameters. That is, how does varying these impact the fraction of correct letters that you obtain? (50%)

4 References

Some useful references include:

1. http://probability.ca/jeff/ftpdir/decipherart.pdf
2. http://statweb.stanford.edu/~cgates/PERSI/papers/MCMCRev.pdf
3. https://www.math.utah.edu/~ethier/5040-MCMC-Lecs1-5.pdf (The last few slides)