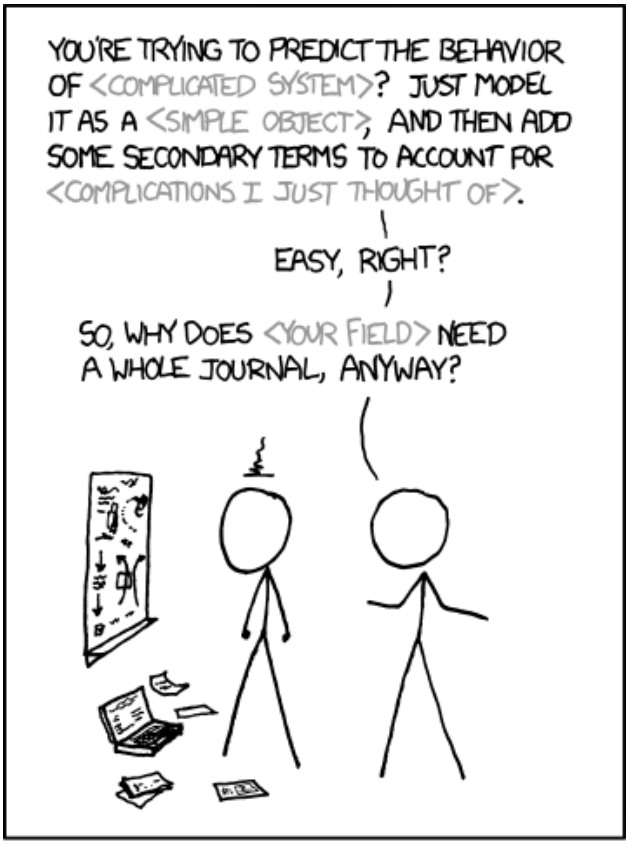
Project 2

Due July 3



`https://xkcd.com/793/`

1 Description

A number of 'data files' are included in `data.zip`; all data files are JSON arrays representing data from some (hypothetical) experiment/study. The x-values of the data sets are just the array indices (indexed from zero). The y-values are the actual array values. For example, the `oscillator.json` data set has:

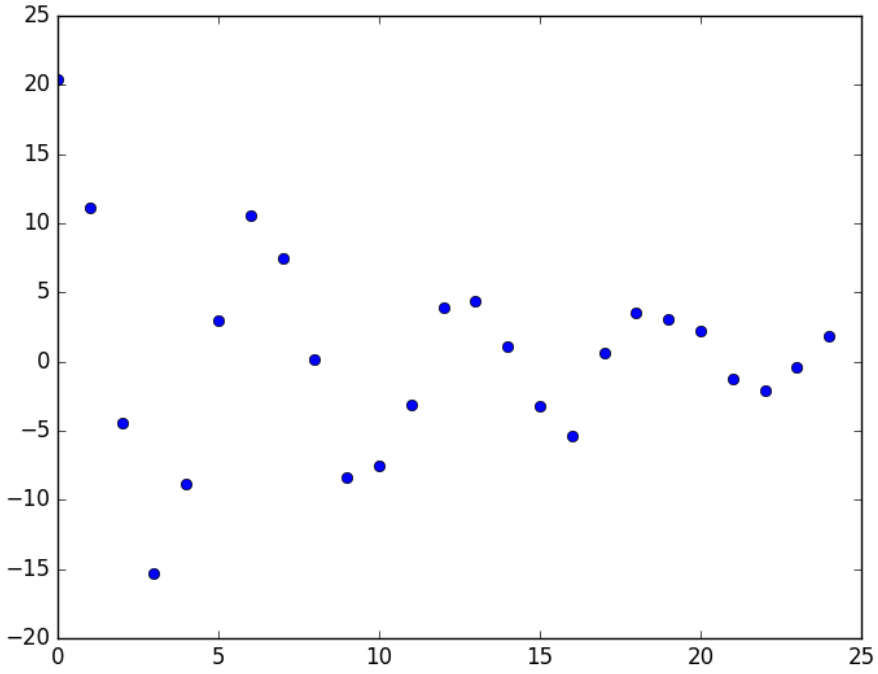
```
[20.4388, 11.1238, -4.39306, -15.2666, -8.83603, 2.95402, 10.6062,
7.45964, 0.1444, -8.34118, -7.56731, -3.08952, 3.9061, 4.388,
1.12161, -3.26213, -5.39157, 0.578177, 3.57381, 3.07006, 2.1935,
-1.25449, -2.06437, -0.404703, 1.84]
```

This is easy to visualize (optional, but it may be helpful) in python3, if you have matplotlib installed (install with `$ sudo pip3 install matplotlib` if you don't have matplotlib):

```
import json
import matplotlib.pyplot as plt

data = []
with open(oscillator.json) as data_file:
    data = json.load(data_file)

plt.plot(data, 'o')
plt.show()
```



Your assignment is to implement a search/optimization program that finds a real-valued function that fits the data well. For this assignment, a good fit is a function, $f(x)$, that has small *sum of squared error*:

$$\sum_{i=0}^{n-1}(y_i - f(x_i))^2$$

where y_i is the i^{th} value of the data set and x_i is just i (to make things simpler, all of the data sets have x values {0, 1, 2, ...})
Traditionally, this kind of search/optimization problem is done with s-expressions. For example, the function:

$$f(x) = 2 * (x + 1) + 3$$

as an s-expression would be:

```
(+ (* 2 (+ x 1)) 3)
```

s-expressions are not difficult to parse, but it's still more work than I want you to have to do for this project. Instead, we'll be using JSON-expressions (which can be easily parsed by any standard JSON library):

```
["+", ["*", 2, ["+", "x", 1]] 3]
```

JSON-expressions (like s-expressions) represent expression trees. The first element of any array is the operator, all elements after that are arguments to that operator; note that arguments can be the result of another function. JSON-expressions support the following binary operators:

- `"+"` - addition
- `"-"` - subtraction
- `"*"` - multiplication

And the following unary operators:

- `"e"` - exponentiation (`["e", "x"]` is equivalent to e^x)
- `"sin"` - sine
- `"cos"` - cosine

Arguments can either be any real number or `"x"`.

2 Required files

You must provide the following executable files:

- `$ ./initial` (no arguments) should print a valid, *random* JSON-expression to standard output; repeatedly running `initial` should produce expressions of varying sizes with varying arguments (random real numbers and "x") and operators. The trees should not always be be complete/full (i.e. many of the random trees should be unbalanced). Tree height should be no greater than 8.

- `$ ./mutate '[JSON-expression]'` should replace (and print to standard output) some subtree of the JSON-expression with a new, random subtree. Like with `initial`, repeatedly running `mutate` should produce varying random 'mutations'.

- `$ ./crossover '[JSON-expression]''[JSON-expression]'` should calculate the result of 'swapping' one random subtree of the first JSON-expression with a random subtree of the second JSON-expression and print the result to standard output (one JSON-expression per line). Like with `initial` and `mutate`, repeatedly running `crossover` should produce varying results. Every possible crossover should have a non-zero probability of ocurring.

- `$ ./error data_file.json '[JSON-expression]'` should calculate (and print to standard output) the sum of squared error for a given JSON-expression and data file. For example:

  ```
  ./error line.json '["+", "x", 1]'
  33.85309903600001
  ```

The above files are required and are the 'individual components' of a complete genetic programming implementation. For (up to) 10% extra credit, also implement:

- `$ ./optimize data_file.json` which should run genetic programming (create initial random population, calculate fitness, select fit memebers of the population for crossover, mutate, repeat) to find a good fitting function for the data in `data_file.json` Your `optimize` program should print out the best scoring JSON-expression, its sum of squared

error and its fitness (if different than sum of squared error; see the next section)

Submit your files to the submit server (`https://submit.cs.umd.edu/`)

2.1 A note on 'bloat'

Genetic programming often suffers from 'bloat', where the trees become very large over time. One way to mitigate this problem is to modify the fitness function to include a 'penalty' for large trees.
There are many parameters to 'tune' in a genetic programming implementation. For full extra credit, your genetic programming implementation should find JSON-expressions that are close to the actual functions used to model the original data sets. (This can be difficult to achieve. You'll still get full regular credit, even if your `optimize` doesn't work especially well, as long as you correctly implement the other requirements.)

3 Other details

- **Start early. Do not put this project off until the last minute.**
- All code submitted must be your own. You may use standard library functions/modules and are encouraged to use an existing JSON library.
- This project is *possible* to do in Java, but that is likely making your life more difficult then it needs to be. Python or Ruby is recommended.
- Your files must be executable; if you're using a scripting language (recommended), the first line should be an appropriate shebang (`https://en.wikipedia.org/wiki/Shebang_(Unix)`), e.g.

```
#!/usr/bin/env python3
```

- The following programming languages are 'officially sanctioned'. Talk to us before using anything different (remember, we do need to be able to execute your code on our machines):
  - Java

- Python 2 or 3 (make sure you use the correct shebang)
- Ruby
- OCaml (make sure `$ ./solution ...` works)
- Bash (to 'wrap' your code files, if necessary)