Week 4 FIT3139 Lab – Please read me!

Hi there, Cedric Liang here, student number 29674662. My lab is the Thursday 2pm session. Unfortunately I'm not able to make today's lab because I've had the flu over the past week and still have symptoms.

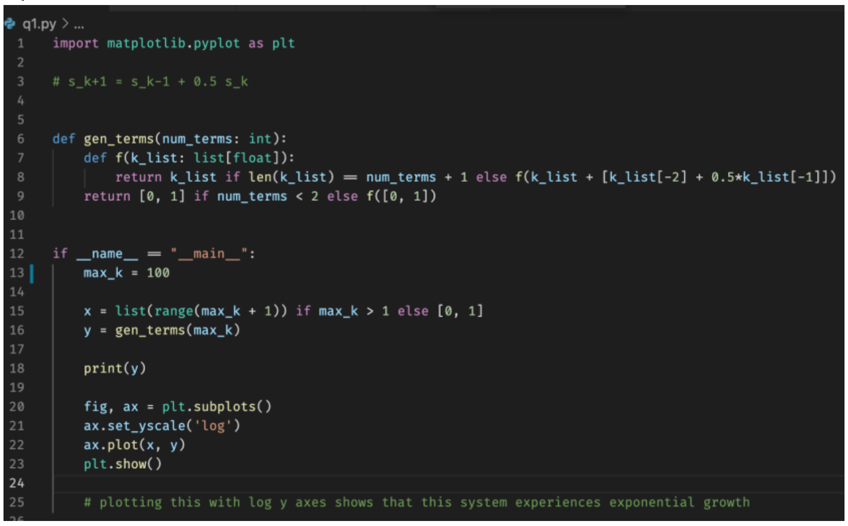
I've included this document to act as an interview in 'pdf' form in the hopes that I might be able to get marked for this week, although I understand if this is not possible. Still, I intend to demonstrate with these explanations of my solutions that the code is my own work, and that I understand my solutions.

You can also check the development history of my solutions on my Github repo, demonstrating the iterative process I followed when coming to my solutions.

https://github.com/cedliang/FIT3139/tree/main/Labs/week4

You can also check that the code style and structure of the files are similar to my Week 2 and 3 solutions, for which I *did* interview.

Question 1



q1.py > ...

```python
import matplotlib.pyplot as plt

# s_k+1 = s_k-1 + 0.5 s_k


def gen_terms(num_terms: int):
    def f(k_list: list[float]):
        return k_list if len(k_list) == num_terms + 1 else f(k_list + [k_list[-2] + 0.5*k_list[-1]])
    return [0, 1] if num_terms < 2 else f([0, 1])


if __name__ == "__main__":
    max_k = 100

    x = list(range(max_k + 1)) if max_k > 1 else [0, 1]
    y = gen_terms(max_k)

    print(y)

    fig, ax = plt.subplots()
    ax.set_yscale('log')
    ax.plot(x, y)
    plt.show()

    # plotting this with log y axes shows that this system experiences exponential growth
```

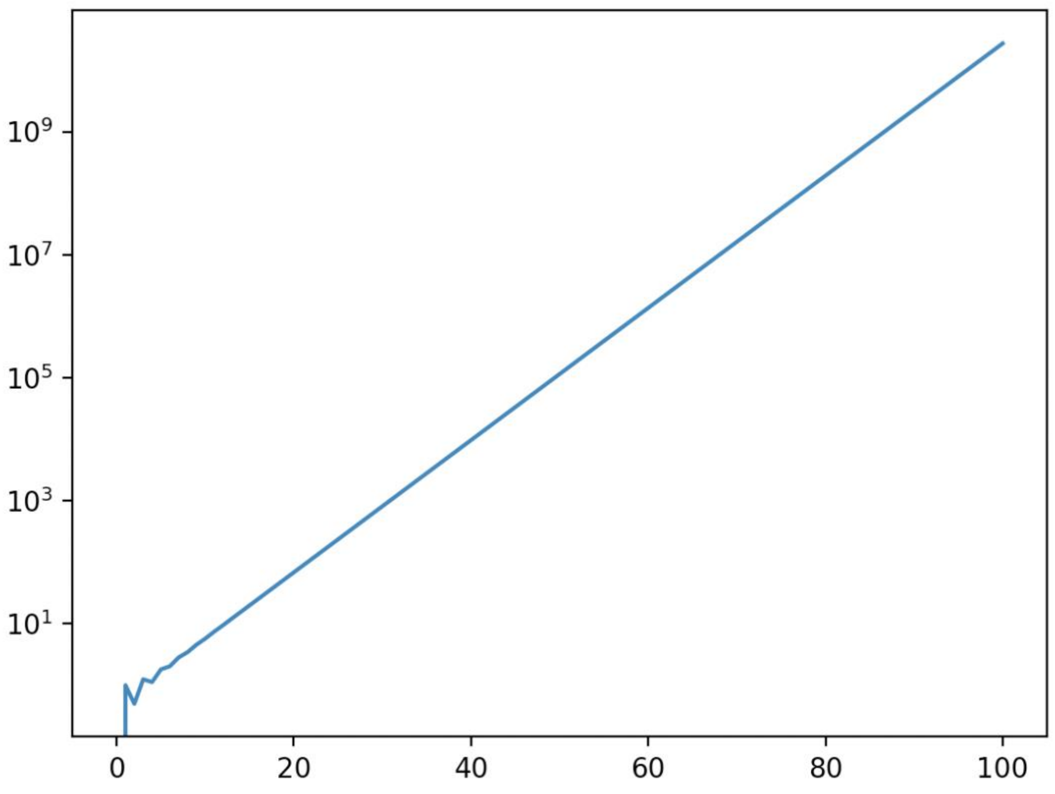
This question asks for the calculation of the rate of growth of the system in the question sheet.

I implement a function called gen_terms that acts as the entry point into the recursive function. The number of terms required is passed in as a parameter to the function.

I implement a *constructive* tail recursive function in order to generate the terms because this is necessary to implement recursion in a way that includes memoisation, so that I'm not computing overlapping recursive calls.

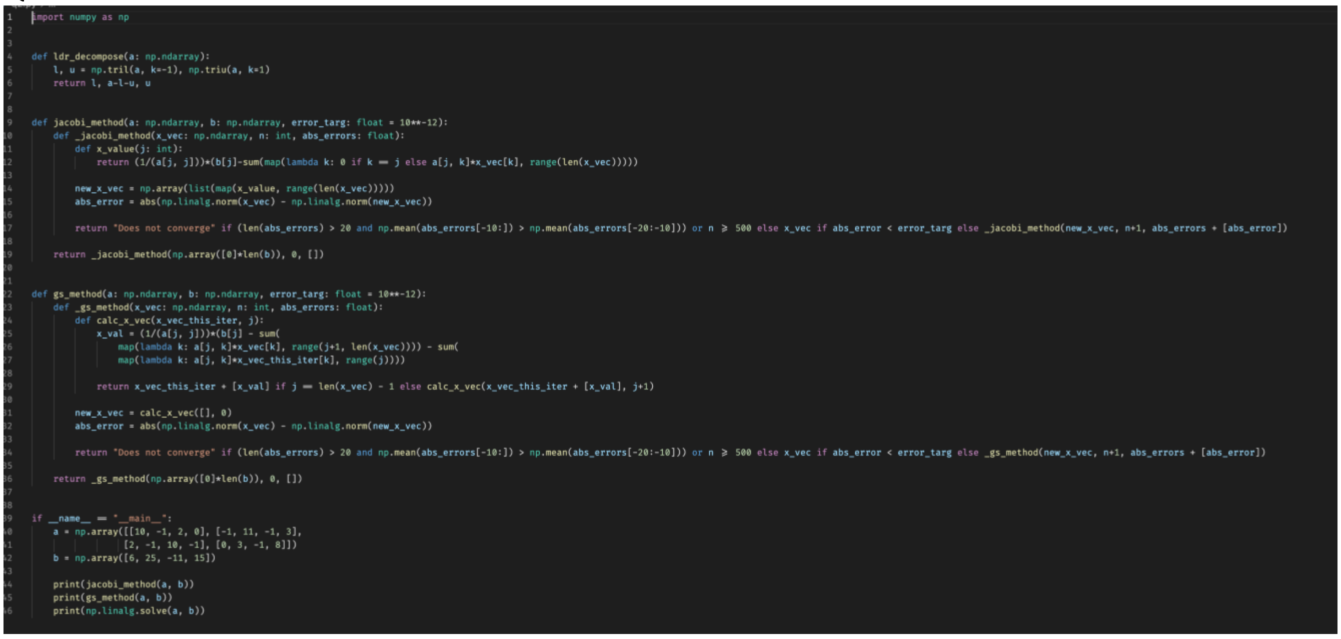
Here, I also initialise the function with initial values of [0, 1], which are the defined first terms in the system.

In the __main__ section of the code, I set my number of terms, generate my x and ys, then plot them to yield this graph (for a max_k value of 100):



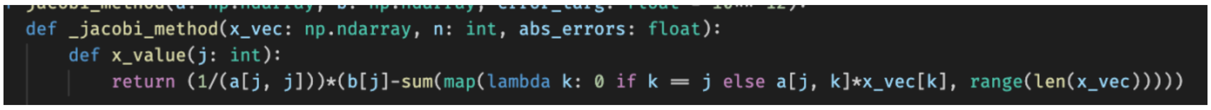
Noting that I’ve set the y axis on a log scale, it follows that the system experiences exponential growth.

Question 2



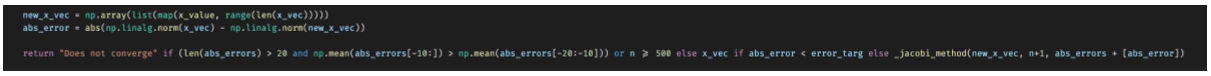
```python
import numpy as np


def ldr_decompose(a: np.ndarray):
    l, u = np.tril(a, k=-1), np.triu(a, k=1)
    return l, a-l-u, u


def jacobi_method(a: np.ndarray, b: np.ndarray, error_targ: float = 10**-12):
    def _jacobi_method(x_vec: np.ndarray, n: int, abs_errors: float):
        def x_value(j: int):
            return (1/(a[j, j]))*(b[j]-sum(map(lambda k: 0 if k == j else a[j, k]*x_vec[k], range(len(x_vec)))))

        new_x_vec = np.array(list(map(x_value, range(len(x_vec)))))
        abs_error = abs(np.linalg.norm(x_vec) - np.linalg.norm(new_x_vec))

        return "Does not converge" if (len(abs_errors) > 20 and np.mean(abs_errors[-10:]) > np.mean(abs_errors[-20:-10])) or n >= 500 else x_vec if abs_error < error_targ else _jacobi_method(new_x_vec, n+1, abs_errors + [abs_error])

    return _jacobi_method(np.array([0]*len(b)), 0, [])


def gs_method(a: np.ndarray, b: np.ndarray, error_targ: float = 10**-12):
    def _gs_method(x_vec: np.ndarray, n: int, abs_errors: float):
        def calc_x_vec(x_vec_this_iter, j):
            x_val = (1/(a[j, j]))*(b[j] - sum(
                map(lambda k: a[j, k]*x_vec[k], range(j+1, len(x_vec)))) - sum(
                map(lambda k: a[j, k]*x_vec_this_iter[k], range(j))))

            return x_vec_this_iter + [x_val] if j == len(x_vec) - 1 else calc_x_vec(x_vec_this_iter + [x_val], j+1)

        new_x_vec = calc_x_vec([], 0)
        abs_error = abs(np.linalg.norm(x_vec) - np.linalg.norm(new_x_vec))

        return "Does not converge" if (len(abs_errors) > 20 and np.mean(abs_errors[-10:]) > np.mean(abs_errors[-20:-10])) or n >= 500 else x_vec if abs_error < error_targ else _gs_method(new_x_vec, n+1, abs_errors + [abs_error])

    return _gs_method(np.array([0]*len(b)), 0, [])


if __name__ == "__main__":
    a = np.array([[10, -1, 2, 0], [-1, 11, -1, 3],
                  [2, -1, 10, -1], [0, 3, -1, 8]])
    b = np.array([6, 25, -11, 15])

    print(jacobi_method(a, b))
    print(gs_method(a, b))
    print(np.linalg.solve(a, b))
```

Here I’ve defined both my Jacobi method and my Gauss-Seidel method using the element-wise definitions. Both functions follow a similar structure: a tail recursive function that has access to the current x_vec (representing the x_vector passed in from the previous iteration), an n (a counter for the level of iteration), and the list of absolute errors (it will become apparent soon why the list is necessary, but in short, I store the last 20 absolute errors and compare the average of the first 10 terms to the average of the last 10 terms in order to check for convergence).



```
    def _jacobi_method(x_vec: np.ndarray, n: int, abs_errors: float):
        def x_value(j: int):
            return (1/(a[j, j]))*(b[j]-sum(map(lambda k: 0 if k == j else a[j, k]*x_vec[k], range(len(x_vec)))))
```

In the Jacobi Method implementation, I have a method (x_value) that the x value for that iteration given the nonlocal variable x_vec, which is the x vector from the last iteration, and the *j*, which is the element in the column that is to be computed. The return value here is just a mapping of an anonymous function containing the mathematics of each term, over range(len(x_vec)), which represents the indices of the x vector. Note here that I also have logic that returns that term to be 0 if *j* if *k* matches *j* – to ensure that the term is discarded in the sum computed later.



```
new_x_vec = np.array(list(map(x_value, range(len(x_vec)))))
abs_error = abs(np.linalg.norm(x_vec) - np.linalg.norm(new_x_vec))

return "Does not converge" if (len(abs_errors) > 20 and np.mean(abs_errors[-10:]) > np.mean(abs_errors[-20:-10])) or n >= 500 else x_vec if abs_error < error_targ else _jacobi_method(new_x_vec, n+1, abs_errors + [abs_error])
```

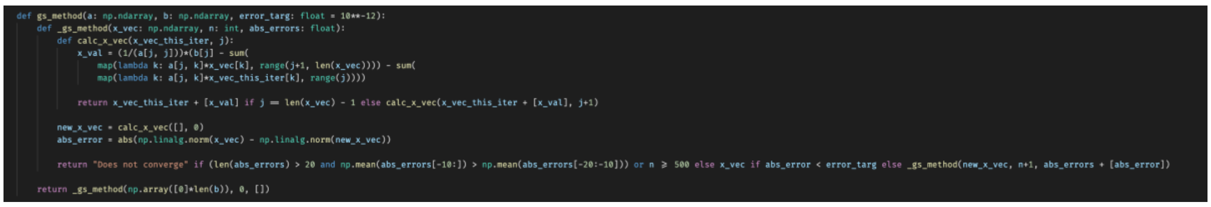
The x vector is computed with a mapping over the indices of x_vec, and the absolute error is calculated. I then have the return function, which includes branching logic for the recursive call.

The last 20 absolute errors are stored in the abs_errors array – this is useful because it helps give me an indication of whether the problem is diverging or not. Looking at the individual absolute errors, I notice that even in convergent problems, it can be the case that the absolute error increases locally from one iteration to the next, even if the overall trend of the absolute error is downwards. This means that it's not sufficient to call a problem divergent if the absolute error at one iteration is greater than the absolute error at the iteration before.

As such, I compare the first 10 absolute errors with the last 10 absolute errors using np.mean, with the idea that over this number of iterations, a much more accurate picture of the divergence properties is provided.

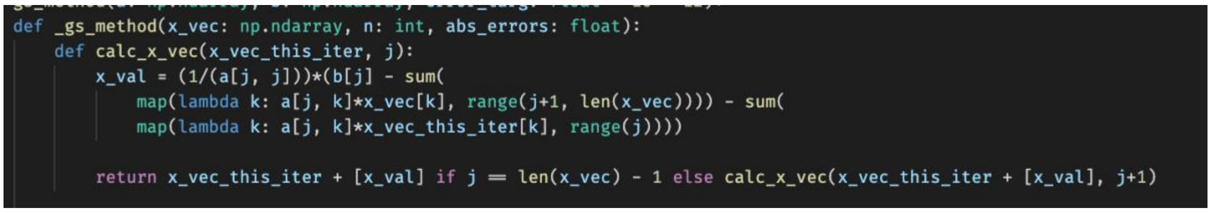
I also need a safeguard for n > 500 to ensure I don't infinite loop. I'm aware that there's a subclass of problems where the size of the absolute error oscillates between two fixed values, which may not necessarily be caught by the previous logic. Nevertheless, if it doesn't hit our absolute error target in 500 iterations, we'll call it non-divergent. It could converge very slowly, but it'll be up to the user to ensure they pick an appropriate error target for the problem.

I then have a recursive call conditional upon whether the absolute error target is reached – if so, return x_vec, if not, recurse.



```
def gs_method(a: np.ndarray, b: np.ndarray, error_targ: float = 10**-12):
    def _gs_method(x_vec: np.ndarray, n: int, abs_errors: float):
        def calc_x_vec(x_vec_this_iter, j):
            x_val = (1/(a[j, j]))*(b[j] - sum(
                map(lambda k: a[j, k]*x_vec[k], range(j+1, len(x_vec)))) - sum(
                map(lambda k: a[j, k]*x_vec_this_iter[k], range(j))))

            return x_vec_this_iter + [x_val] if j == len(x_vec) - 1 else calc_x_vec(x_vec_this_iter + [x_val], j+1)

        new_x_vec = calc_x_vec([], 0)
        abs_error = abs(np.linalg.norm(x_vec) - np.linalg.norm(new_x_vec))

        return "Does not converge" if (len(abs_errors) > 20 and np.mean(abs_errors[-10:]) > np.mean(abs_errors[-20:-10])) or n >= 500 else x_vec if abs_error < error_targ else _gs_method(new_x_vec, n+1, abs_errors + [abs_error])

    return _gs_method(np.array([0]*len(b)), 0, [])
```

The implementation for Gauss-Seidel is similar – the error threshold mechanism is the same. The difference is the calculation of the x terms:



```
def _gs_method(x_vec: np.ndarray, n: int, abs_errors: float):
    def calc_x_vec(x_vec_this_iter, j):
        x_val = (1/(a[j, j]))*(b[j] - sum(
            map(lambda k: a[j, k]*x_vec[k], range(j+1, len(x_vec)))) - sum(
            map(lambda k: a[j, k]*x_vec_this_iter[k], range(j))))

        return x_vec_this_iter + [x_val] if j == len(x_vec) - 1 else calc_x_vec(x_vec_this_iter + [x_val], j+1)
```

Here I use the property for Gauss-Seidel:

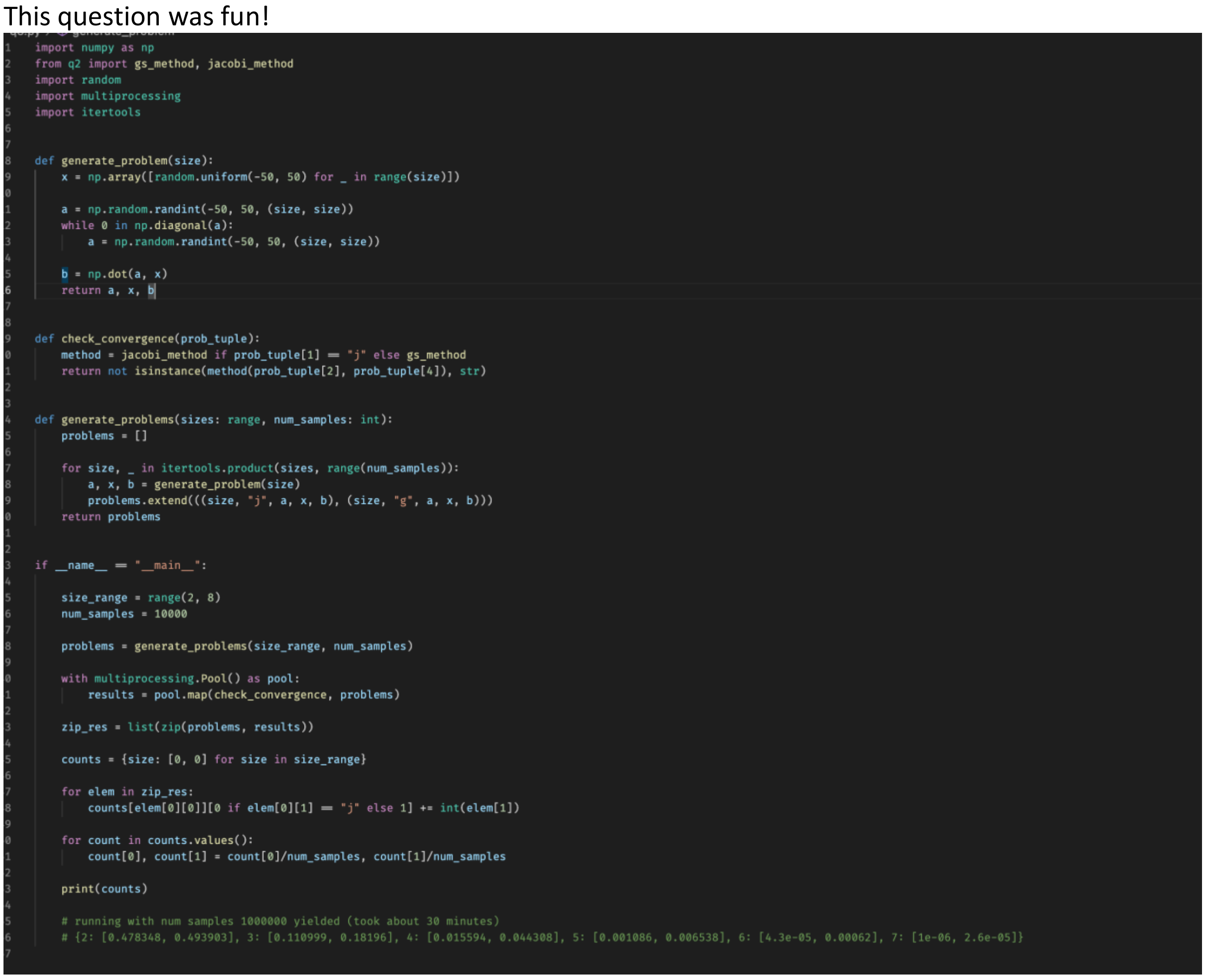
$$\vec{x}_j^{(i)} = \frac{1}{a_{jj}}\left(\vec{b}_j - \sum_{k=1}^{j-1} a_{jk}\vec{x}_k^{(i)} - \sum_{k=j+1}^{n} a_{jk}\vec{x}_k^{(i-1)}\right)$$

I have the implementation here where I've calculated each of the sums by mapping over the relevant *k* range, and also ensured that I have a reference to the x_vec_this_iter, which represents the x_vec in this iteration that I am building.

As such, note that this time, instead of mapping an 'independent' function over the indices of x_vec as we did in the Jacobi solution, here calc_x_vec itself is a recursive function that carries the accumulated x_vec so far this iteration, calculates the term using that, and makes a recursive call with the new value appended.

Question 3

This question was fun!



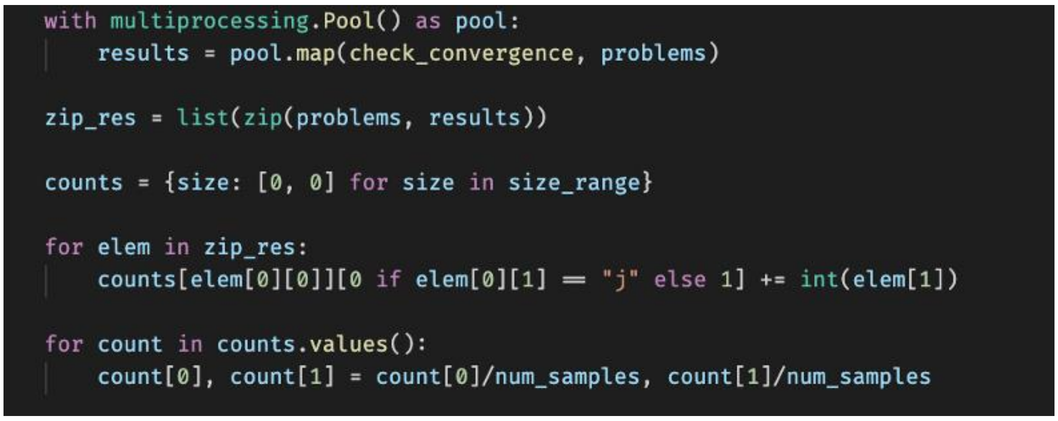
```python
import numpy as np
from q2 import gs_method, jacobi_method
import random
import multiprocessing
import itertools


def generate_problem(size):
    x = np.array([random.uniform(-50, 50) for _ in range(size)])

    a = np.random.randint(-50, 50, (size, size))
    while 0 in np.diagonal(a):
        a = np.random.randint(-50, 50, (size, size))

    b = np.dot(a, x)
    return a, x, b


def check_convergence(prob_tuple):
    method = jacobi_method if prob_tuple[1] == "j" else gs_method
    return not isinstance(method(prob_tuple[2], prob_tuple[4]), str)


def generate_problems(sizes: range, num_samples: int):
    problems = []

    for size, _ in itertools.product(sizes, range(num_samples)):
        a, x, b = generate_problem(size)
        problems.extend(((size, "j", a, x, b), (size, "g", a, x, b)))
    return problems


if __name__ == "__main__":

    size_range = range(2, 8)
    num_samples = 10000

    problems = generate_problems(size_range, num_samples)

    with multiprocessing.Pool() as pool:
        results = pool.map(check_convergence, problems)

    zip_res = list(zip(problems, results))

    counts = {size: [0, 0] for size in size_range}

    for elem in zip_res:
        counts[elem[0][0]][0 if elem[0][1] == "j" else 1] += int(elem[1])

    for count in counts.values():
        count[0], count[1] = count[0]/num_samples, count[1]/num_samples

    print(counts)

    # running with num samples 1000000 yielded (took about 30 minutes)
    # {2: [0.478348, 0.493903], 3: [0.110999, 0.18196], 4: [0.015594, 0.044308], 5: [0.001086, 0.006538], 6: [4.3e-05, 0.00062], 7: [1e-06, 2.6e-05]}
```

I kind of went all out for this one, implementing multiprocessing in order to run simulations of large volumes of problems.

I also wanted to check the proportion of convergent problems for matrices of difference sizes, so I've made my 'generate_problems' method take sizes (a range) as a parameter. Here, I generate 10000 problems each for matrices of 2x2 up to 7x7.

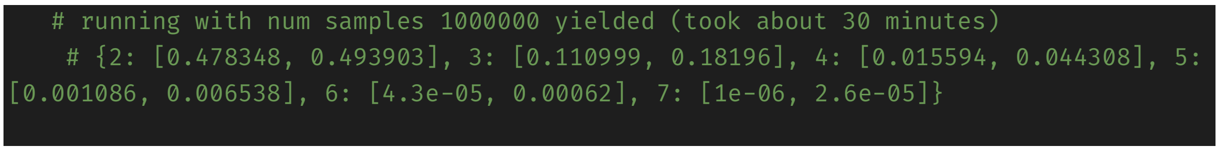
I then assign ensure that two copies of each problem is stored, one with ‘g’ attached, one with ‘j’ attached, so that their solutions can be computed asynchronously by whichever process is assigned it.

I also have a check_convergence method that runs the algorithm on the problem. If the problem does not converge, the output is a type string (as per the previous problem), so I check for conversion with an isinstance.



```
with multiprocessing.Pool() as pool:
    results = pool.map(check_convergence, problems)

zip_res = list(zip(problems, results))

counts = {size: [0, 0] for size in size_range}

for elem in zip_res:
    counts[elem[0][0]][0 if elem[0][1] == "j" else 1] += int(elem[1])

for count in counts.values():
    count[0], count[1] = count[0]/num_samples, count[1]/num_samples
```

Finally, I spawn a pool of processes and use pool.map to compute the solutions to all my problems. I then parse these to check for convergence for each method with a counter for each matrix size.



```
    # running with num samples 1000000 yielded (took about 30 minutes)
     # {2: [0.478348, 0.493903], 3: [0.110999, 0.18196], 4: [0.015594, 0.044308], 5:
[0.001086, 0.006538], 6: [4.3e-05, 0.00062], 7: [1e-06, 2.6e-05]}
```

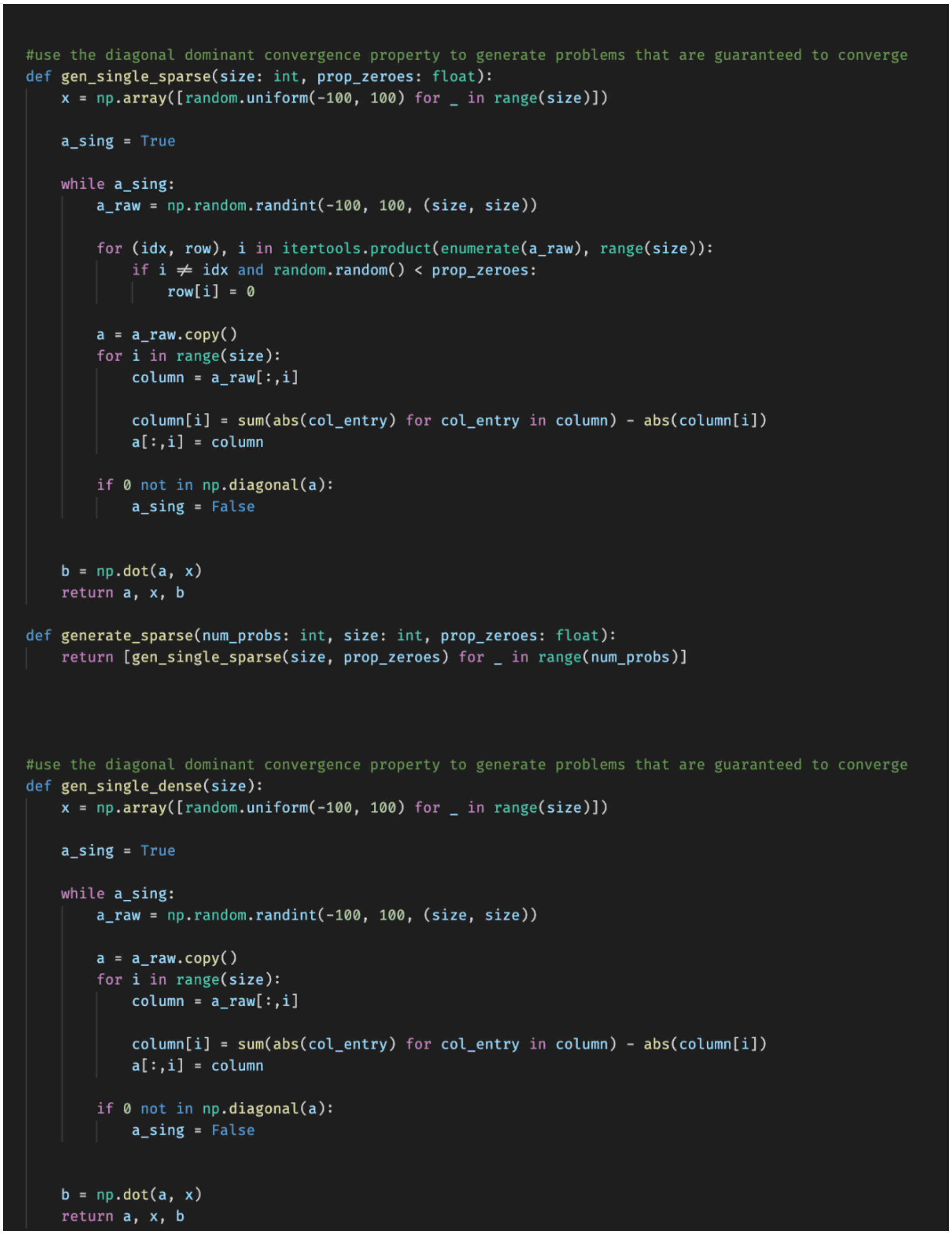
As per this comment, I ran the script on 1,000,000 problems for each matrix size between (and including) 2x2 and 7x7 – the pairs are the results, with the first element being the proportion of problems that converge under Jacobi iteration, and the second element being the that value for Gauss Seidel.

Indeed, for all problem sizes, Gauss-Seidel converges for more problems than Jacobi. Interestingly, the proportion of problems that converge under J decreases much faster than the proportion of problems that converge under GS.

Question 4
Here, I’m computing the time it takes to find solutions to convergent problems with iterative and direct methods.

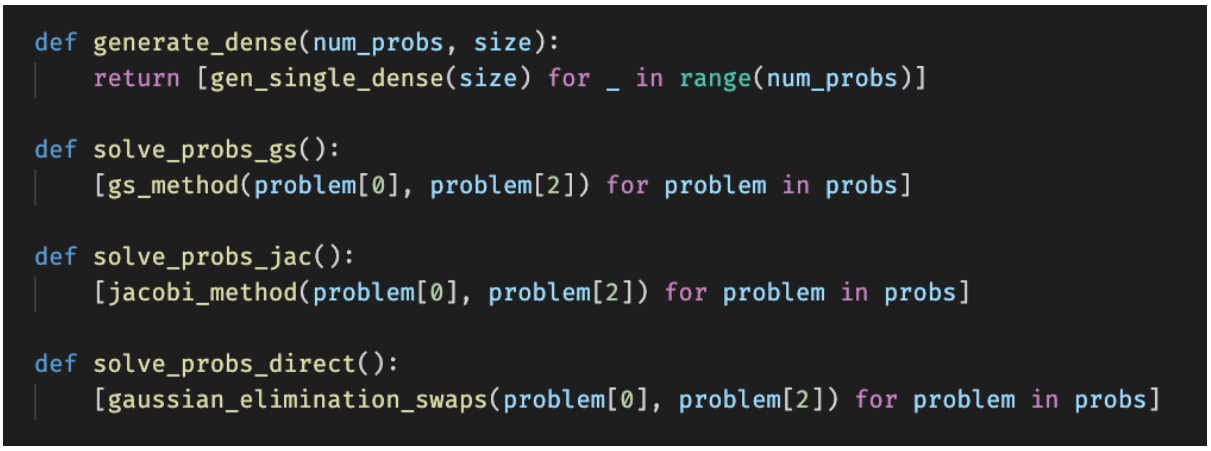
Two things – first is that I generate problems that are guaranteed to converge under iteration by generating matrices that are only diagonally dominant.

Second, I make two methods for generating problems (although I really only needed one) – one to generate problems with all elements with a random value as usual, and the second where I overwrite a certain proportion of the elements in the *A* matrix with zeros, in order to get a certain *sparcity.*



```python
#use the diagonal dominant convergence property to generate problems that are guaranteed to converge
def gen_single_sparse(size: int, prop_zeroes: float):
    x = np.array([random.uniform(-100, 100) for _ in range(size)])

    a_sing = True

    while a_sing:
        a_raw = np.random.randint(-100, 100, (size, size))

        for (idx, row), i in itertools.product(enumerate(a_raw), range(size)):
            if i ≠ idx and random.random() < prop_zeroes:
                row[i] = 0

        a = a_raw.copy()
        for i in range(size):
            column = a_raw[:,i]

            column[i] = sum(abs(col_entry) for col_entry in column) - abs(column[i])
            a[:,i] = column

        if 0 not in np.diagonal(a):
            a_sing = False


    b = np.dot(a, x)
    return a, x, b

def generate_sparse(num_probs: int, size: int, prop_zeroes: float):
    return [gen_single_sparse(size, prop_zeroes) for _ in range(num_probs)]



#use the diagonal dominant convergence property to generate problems that are guaranteed to converge
def gen_single_dense(size):
    x = np.array([random.uniform(-100, 100) for _ in range(size)])

    a_sing = True

    while a_sing:
        a_raw = np.random.randint(-100, 100, (size, size))

        a = a_raw.copy()
        for i in range(size):
            column = a_raw[:,i]

            column[i] = sum(abs(col_entry) for col_entry in column) - abs(column[i])
            a[:,i] = column

        if 0 not in np.diagonal(a):
            a_sing = False


    b = np.dot(a, x)
    return a, x, b
```

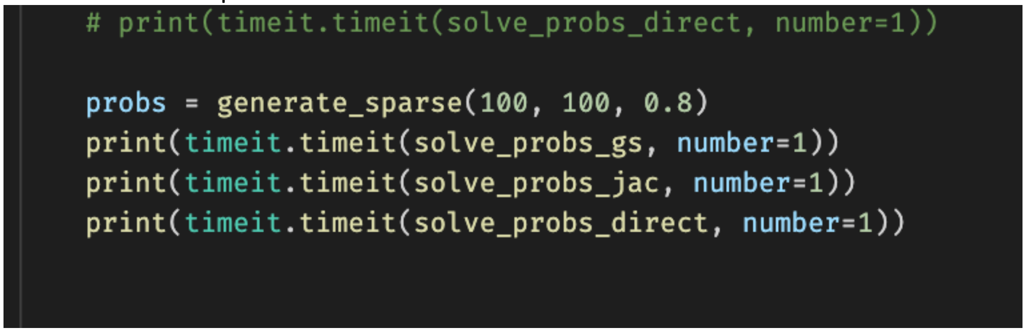
For example, here I can generate an *A* matrix with 60% of the cells being 0 if I pass 0.6 as an argument to prop_zeroes. Of course, size is also an argument as usual.



```python
def generate_dense(num_probs, size):
    return [gen_single_dense(size) for _ in range(num_probs)]

def solve_probs_gs():
    [gs_method(problem[0], problem[2]) for problem in probs]

def solve_probs_jac():
    [jacobi_method(problem[0], problem[2]) for problem in probs]

def solve_probs_direct():
    [gaussian_elimination_swaps(problem[0], problem[2]) for problem in probs]
```

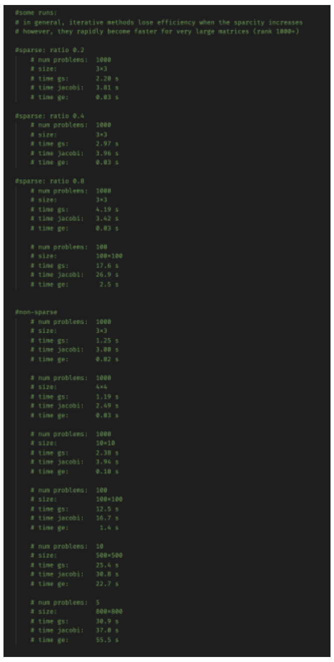
Here, I provide caller methods for solving the problems, that I can use as an argument to timeit.

Finally, I generate some problems with some params, create the problems, and print the time it takes to solve the problems.



```python
    # print(timeit.timeit(solve_probs_direct, number=1))

    probs = generate_sparse(100, 100, 0.8)
    print(timeit.timeit(solve_probs_gs, number=1))
    print(timeit.timeit(solve_probs_jac, number=1))
    print(timeit.timeit(solve_probs_direct, number=1))
```

I've attached the runtimes for sets of parameters below.



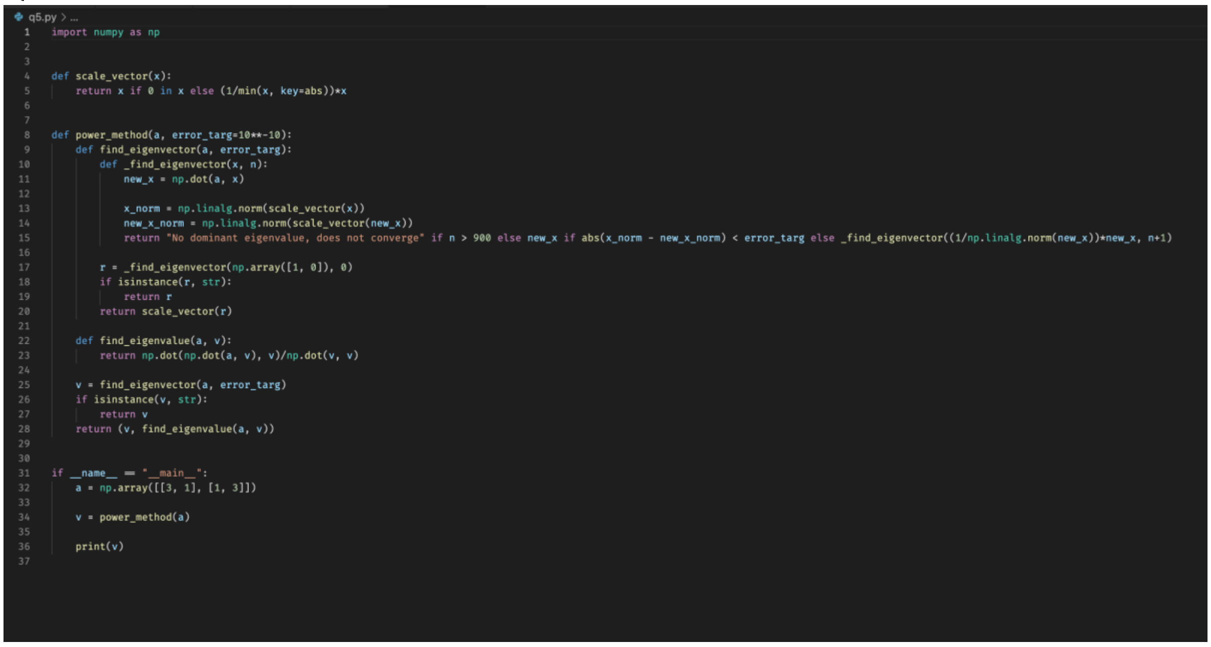
```
#some runs:
# in general, iterative methods lose efficiency when the sparcity increases
# however, they rapidly become faster for very large matrices (rank 1000+)

#sparse: ratio 0.2
    # num problems:  1000
    # size:          3×3
    # time gs:       2.20 s
    # time jacobi:   3.81 s
    # time ge:       0.03 s

#sparse: ratio 0.4
    # num problems:  1000
    # size:          3×3
    # time gs:       2.97 s
    # time jacobi:   3.96 s
    # time ge:       0.03 s

#sparse: ratio 0.8
    # num problems:  1000
    # size:          3×3
    # time gs:       4.19 s
    # time jacobi:   3.42 s
    # time ge:       0.03 s

    # num problems:  100
    # size:          100×100
    # time gs:       17.6 s
    # time jacobi:   26.9 s
    # time ge:        2.5 s


#non-sparse
    # num problems:  1000
    # size:          3×3
    # time gs:       1.25 s
    # time jacobi:   3.00 s
    # time ge:       0.02 s

    # num problems:  1000
    # size:          4×4
    # time gs:       1.19 s
    # time jacobi:   2.49 s
    # time ge:       0.03 s

    # num problems:  1000
    # size:          10×10
    # time gs:       2.38 s
    # time jacobi:   3.94 s
    # time ge:       0.10 s

    # num problems:  100
    # size:          100×100
    # time gs:       12.5 s
    # time jacobi:   16.7 s
    # time ge:        1.4 s

    # num problems:  10
    # size:          500×500
    # time gs:       25.4 s
    # time jacobi:   30.8 s
    # time ge:       22.7 s

    # num problems:  5
    # size:          800×800
    # time gs:       30.9 s
    # time jacobi:   37.0 s
    # time ge:       55.5 s
```

Worthy to note here is that for smaller problems, a direct method is *much* faster than an iterative method. However, the size and sparcity of the problem affects not just the amount of time it takes to find the solution using each method – both parameters scale the time taken differently for direct and iterative methods.

For example, from the data of our non-sparce runs, we can see that the time taken to solve matrices with direct methods increases with a much higher proportionality as the size of the matrix increases. For 3x3 matrices, the iterative methods took on the order of 100 times longer to compute the solutions. As we gradually increase the size of the matrix, we can see that ratio drops until there's approximate parity at a matrix size of 500x500, and superiority for iterative methods beyond there, such as at matrices of size 800x800.

Sparcity is the other parameter. For the iterative methods, we can see that a matrix being sparse *increased* the amount of time required to compute the solution. I can't see why there would be a difference in algebraic computation time, which leads me to suspect that the reason why it's slower is because sparse matrices somehow converge slower (they're working on less 'data' I guess?), and take more iterations to hit an abolute error target.

Question 5

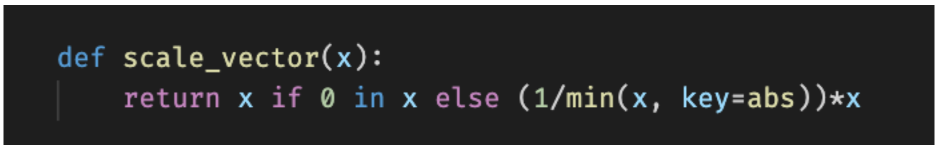


q5.py > ...

```python
import numpy as np


def scale_vector(x):
    return x if 0 in x else (1/min(x, key=abs))*x


def power_method(a, error_targ=10**-10):
    def find_eigenvector(a, error_targ):
        def _find_eigenvector(x, n):
            new_x = np.dot(a, x)

            x_norm = np.linalg.norm(scale_vector(x))
            new_x_norm = np.linalg.norm(scale_vector(new_x))
            return "No dominant eigenvalue, does not converge" if n > 900 else new_x if abs(x_norm - new_x_norm) < error_targ else _find_eigenvector((1/np.linalg.norm(new_x))*new_x, n+1)

        r = _find_eigenvector(np.array([1, 0]), 0)
        if isinstance(r, str):
            return r
        return scale_vector(r)

    def find_eigenvalue(a, v):
        return np.dot(np.dot(a, v), v)/np.dot(v, v)

    v = find_eigenvector(a, error_targ)
    if isinstance(v, str):
        return v
    return (v, find_eigenvalue(a, v))


if __name__ == "__main__":
    a = np.array([[3, 1], [1, 3]])

    v = power_method(a)

    print(v)
```

This question is pretty self explanatory basically – performs matrix multiplication repeatedly, extracts eigenvalue.

Note here that not only have I normalised after each step, I've normalised to 1 using this function:



```python
def scale_vector(x):
    return x if 0 in x else (1/min(x, key=abs))*x
```

Here, I normalise according the minimum element in x, so that all other elements can be scaled relative to 1.