Each implementation of MatrixGen calculated the proportions for how a company’s ownership should be divided and stored the results in a list. The matrix was then rotated before it was returned so that each list within the 2d array now represented how much the company controlled of others.

The first implementation of MatrixGen() used some randomness but spread shares out relatively evenly to create a densely populated matrix but resulting in ownership that converged on 1/n as n increased – this was not good because corporate control becomes improbable as n increases.

The second implementation MatrixGen2() used a different approach, ensuring that each column’s sum < 1 by tracking how much of a company’s share had already been claimed. This approach has the advantage that there is 50% probability that the first share apportioned will be greater than 0.5.

The third implementation MatrixGen3() used the same foundation as MatrixGen(2) but included an additional step which added the unclaimed share of a company to another (randomly selected ) company’s controlling share. The function was adjusted to divide it’s shares between only n – 1 companies (because 0 would be inserted in after in the diagonal position).

When it came to testing scalability I realised that representing the problem using a conventional matrix would be impossible at sizes over 10^6. Even when using numpy.zeros() to create an empty matrix where each element in the matrix was the smallest possible memory size (8-bit unsigned integer) I measured that a matrix of size 10^7 was 100,000,000 gigabytes. This meant that any work on the matrix would consume all available RAM on a computer and grind matrix creation /computation to a halt.

The premise of the problem implies that we want to make a company control allocation algorithm that routinely generates controlling shares (>0.5), given that a company’s shares must sum to <= 1, this means that our matrix generator is going to allocate shares for each company to a small number of companies and this number will remain constant even as the number of companies grows.

Ownership allocation was done using a simple loop as seen below:

share = 100  
*while* share > 0:  
 share -= randint(1, share)

This results in each company having approximately 5.2 share holders on average, resulting in an average of 5.2 non-zero entries for each column on the matrix (for any matrix where n > 12). A more realistic model of corporate control would not have columns summing to 1 (because individual shareholders with tiny fractional percentages would not be accounted for) and controlling shares would be much more rare. However, the allocation algorithm chosen here strikes a balance between modelling reality and creating matrices that represent relatively frequent occurrences of corporate control.

The result is that in larger matrices the column values will be mostly zeroes. Instead of using memory to store all of those zeroes I decided to represent the matrix using a sparse matrix where the information stored is:

1. the shape of the matrix
2. it’s non-zero values and
3. the coordinates for each non zero value.

The data of the matrix took the form of a 2d numpy array. It contained 3 arrays of equal length where:

1st array = row indexes

2nd array = column indexes

3rd array = values stored

Memory usage was further optimised by converting the ownership values into percentages that could be stored as 8-bit unsigned integers.

**Explanation of how the corporate control algorithm works:**

Start with a set which initially only contains the row index of **s**.

1. Create a set **control\_positions** this will contain the column index of each company where **s** has (indirect or direct) control (initially this will contain only **s**.)
2. Create a list **col** this will contain the column index of each company where **s** owns a share (initially this will contain only **s**.).
3. Create a list **data** this will contain a list of percentage shares (directly or indirectly) controlled by **s**.
4. Pass into a function **control\_adder()** the sparse matrix representing all cross ownership, the set **control\_positions,** and the lists **col** and **data**
5. **control\_adder()** looks at the column indexes which have been newly-added to **col** (ie those companies that are controlled by **s** but which have not yet been checked)it then iterates through the corresponding row for each of these column indexes in **col.** It appends to **col** and **data** the index of the company that it has shares in and the percentage amount of the share.
6. **control\_adder()** then creates a new single-row sparse matrix where the ownership shares for each company stored in **data** are summed together. This creates a single row matrix where each company is represented by a column and the ownership value is the sum of all companies controlled by **s.**
7. **control\_adder()** builds a new set **new\_fifty\_set,** this set contains column indexes for companies where the control > 50 and the company is not already included in the **control\_positions** set.
8. **new\_fifty\_set** nowcontains companies newly-discovered to be controlled by **s**, these companies are added to **control\_positions.**
9. **new\_fifty\_set** is passed back into **control\_adder()** which loops back to step 5, adding additional share control to **col** and **data** (for each company which has shares held by the companies in **new\_fifty\_set**)
10. Steps 5 to 10 are repeated until the function completes an iteration where no new companies are added to **new\_fifty\_set**.

Look at numbers in row > 0.5

For each control, take column and look at numbers in row

Recursively:

Have set of controlpositions

Add to the set by searching row, if found add to set\

Consider using dtype to optimise the matrix size in memory

: a = np.arange(2708000000, dtype=np.int8)