Tiffany Wang

260684152

ECSE 543 – Assignment 1

The assignment was fully completed in python.

Question 1 3

a) Choleski 3

b) Constructing real, symmetric, positive-definite matrices *A* 3

c) Solving with Choleski Decomposition 4

d) Solving circuit from a circuit file 4

Question 2 6

a) Finding *Req* 6

b) Program’s runtime 7

c) Sparse Matrix 8

d) *R(N) vs. N* 9

Question 3 10

a) Solving with Sucessive Over-Relaxation method 10

b) SOR with fixed *h* and varying *w* 11

c) SOR with fixed *w* and varying *h* 12

d) Solving with Jacobi with varying *h* 12

e) Non-uniform spacing grid 13

Appendix 15

1) Matrix Manipulation 15

Matrix Multiplication: 15

Matrix – Vector Multiplication: 15

Vector Subtraction: 16

Vector Comparator: 16

Choleski Decomposition to LowerMatrix L: 16

Solves L🞄 y =b: 16

Solves LT🞄 x =y 16

Choleski Decomposition solving A🞄 x =b 17

Check if A🞄 x =b is true: 17

2) Sparse Matrix Manipulation 17

Sparse Matrix Manipulation 17

Sparse Matrix – Vector Multiplication: 17

Sparse Choleski Decompose to LowerMatrix L: 18

Solves L🞄 y =b where L is a sparse lower matrix: 18

Solves LT🞄 x =y where LT is a sparse upper matrix: 18

Solving A🞄 x = b where A is a banded matrix 19

3) Circuit Analysis 19

Find Node Voltage: 19

Find Node Voltage using sparse properties of matrices: 19

Find Req: 19

Find Req using sparse properties of matrices: 19

Parse circuit file to circuit network object: 20

Resistor Network circuit file generator: 20

f) Iterative Finite Difference Methods 21

Uniform Spacing Successive Over Relaxation Method: 21

Non-Uniform Spacing Successive Over Relaxation Method: 22

Uniform Jacobi Method: 22

Mapping Grid Over Planes of Symmetry: 23

# Question 1

1. Choleski

Several helper methods were used to program choleski(). These methods include the following:

* choleskiDecomposition(matrix): this function takes in the real, symmetric, positive-definite matrix *A* and modified it into a lower triangle matrix *L*, where *L 🞄 LT = A*. The “look-ahead modification” method was implemented in order to save runtime.
* solvingLowerMatrix(matrix, vector): taking in a lower matrix *L* and a vector *b*, the function returns the solution to *L🞄 y =b*.
* solvingTransposeLwerMatrix(matrix, vector): similar to solverLowerMatrix(), this function solves the equation *LT🞄 x = b*, where *LT* is an upper triangle matrix.

Within choleski(): the read, symmetric, positive definite matrix *A* is first decomposed into its lower triangle matrix *L.* Then using solving solvingLowerMatrix(), we find the corresponding vector *y* to *L🞄 y =b*. Finally, the solution *A🞄x = b*, it found by solving *LT🞄 x = y* with solvingTransposeLwerMatrix().

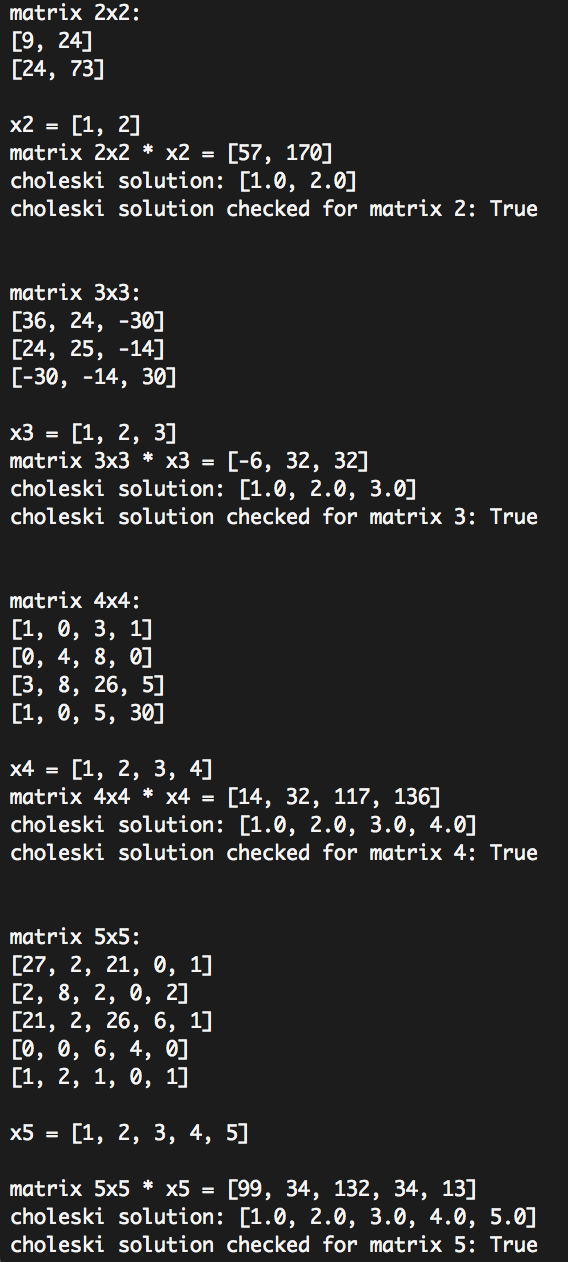
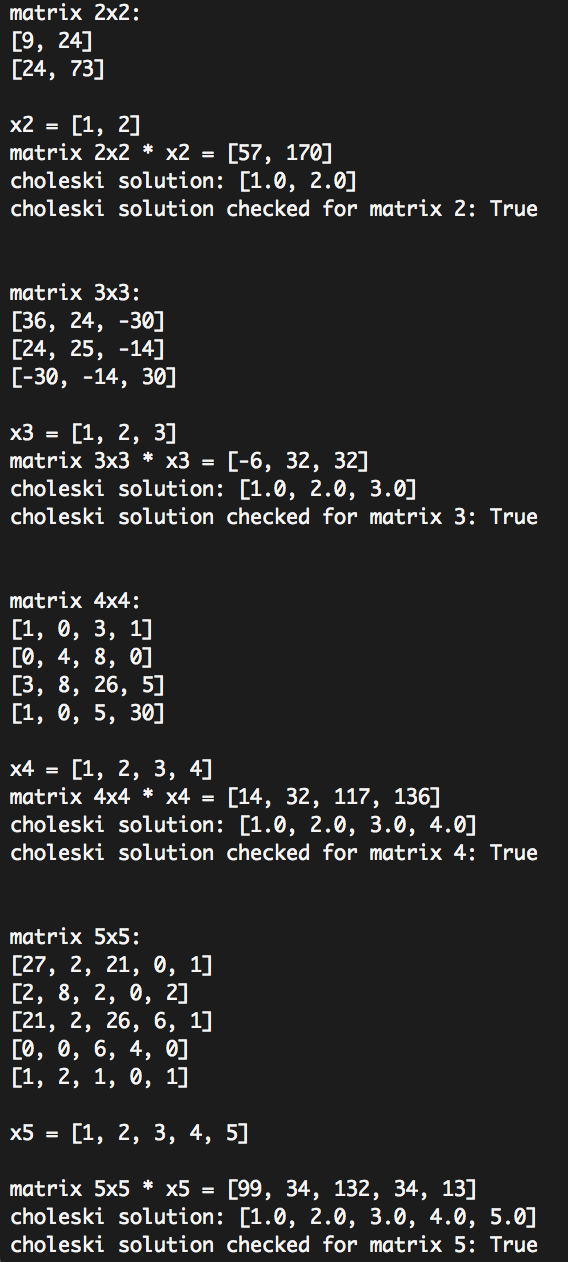
1. Constructing real, symmetric, positive-definite matrices *A*

The real, symmetric, positive-definite matrices were built through computing *A = L 🞄 LT*, where *L*, is a singular lower matrix*.*  I made sure that *A* satisfied the condition *ZT 🞄 A 🞄 Z ≠ 0*. As the results do not already satisfy the conditions, I tweaked the values and found real, symmetric, positive-definite matrices through trial and error (especially for the larger size matrices).

5

1. Solving with Choleski Decomposition

I wrote a vector comparator function to compare the choleski() result and the original x vector.



1. Solving circuit from a circuit file

In this part, I assumed a standard format of the circuit text file of my choice. Below is an example with an explanation of the deployed format:

consider ‘#’ demarks the beginning of a comment.

|  |
| --- |
| 0 0 0 0 0 0 #first line: J – each entry is separated by a space  20 10 10 30 30 30 #second line: R  10 0 0 0 0 0 #third line: E  #empty line  -1 1 1 0 0 0 #incident matrix A  0 -1 0 1 1 0  0 0 -1 -1 0 1 |

The function parseCircuit() reads the lines of the circuit files, parse the data into J, R, E, A tuples, and returns an array containing all these circuit network information. Then, it is easy to retrieve the data by simply calling:

circuitNetwork = parseCircuit(filename)

J = circuitNetwork[0]

R = circuitNetwork[1]

E = circuitNetwork[2]

A = circuitNetwork[3]

I implemented my matrix manipulation codes to solve following equation to find the node voltages *vn.*

where A is the incidence matrix, Y is the admittance matrix, J is the branch current vector, and E is the branch voltage vector.

The results are as follows:

|  |  |  |
| --- | --- | --- |
| Circuit diagram | circuit file | Program Result |
| ../../../../../../Desktop/Screen%20Shot%202017-10-18%20at%208.21.11 | ../../../../../../Desktop/Screen%20Shot%202017-10-18%20at%208.20.51 | ../../../../../../Desktop/Screen%20Shot%202017-10-18%20at%208.28.40 |
| ../../../../../../Desktop/Screen%20Shot%202017-10-18%20at%208.29.51 | ../../../../../../Desktop/Screen%20Shot%202017-10-18%20at%208.32.36 | ../../../../../../Desktop/Screen%20Shot%202017-10-18%20at%208.28.11 |
| ../../../../../../Desktop/Screen%20Shot%202017-10-18%20at%208.30.07 | ../../../../../../Desktop/Screen%20Shot%202017-10-18%20at%208.33.44 | ../../../../../../Desktop/Screen%20Shot%202017-10-18%20at%208.28.07 |
| ../../../../../../Desktop/Screen%20Shot%202017-10-18%20at%208.30.12 | ../../../../../../Desktop/Screen%20Shot%202017-10-18%20at%208.34.09 | ../../../../../../Desktop/Screen%20Shot%202017-10-18%20at%208.28.02 |
| ../../../../../../Desktop/Screen%20Shot%202017-10-18%20at%208.30.17 | ../../../../../../Desktop/Screen%20Shot%202017-10-18%20at%208.34.30 | ../../../../../../Desktop/Screen%20Shot%202017-10-18%20at%208.27.51 |

The results of the choleski circuits analysis are the same as what’s expected (results from hand analysis). Therefore, I conclude that my program works properly.

# Question 2

In this problem, I considered a regular *N* x *2N* finite difference mesh circuit to be the following:

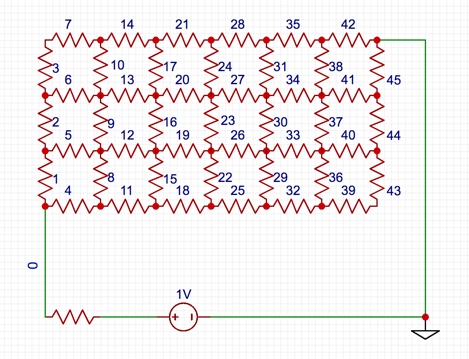
e.g. with N = 1



1. Finding *Req*

In order to calculate the equivalent resistance between the top right and bottom corner, I added an additional mesh connecting the two nodes with a test voltage of 1V and an input resistance of 1kΩ as well.

For a N = 3 circuit, the test circuit would be



Once we find the node voltage at the bottom left corner of the grid, we can use voltage divider to find *Req* of the grid.

The meshes are numbered in the left to right, bottom to top fashion. The vertical meshes are numbered first, then are the horizontal ones. The nodes are numbers in the same way, the grid would for this example would be:

This will render a total of *(N+1) 🞄 (2N + 1)* nodes and *4N2 + 3N + 1* meshes.

These parameters are taken into account when writing the circuit files. As for the incidence matrix, the way the nodes and meshes are numbers make it easier to determine the current flow with regards to each node. In this analysis, the current flow is always from left to right, bottom to top. Please refer to the code in FDMatrixGenerator() function in circuit.py for more detailed understanding.

Still using the same circuit example, the generated incidence matrix *A* would be:

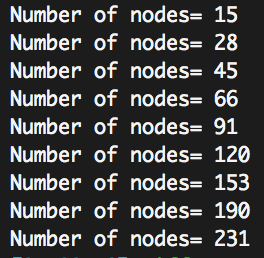


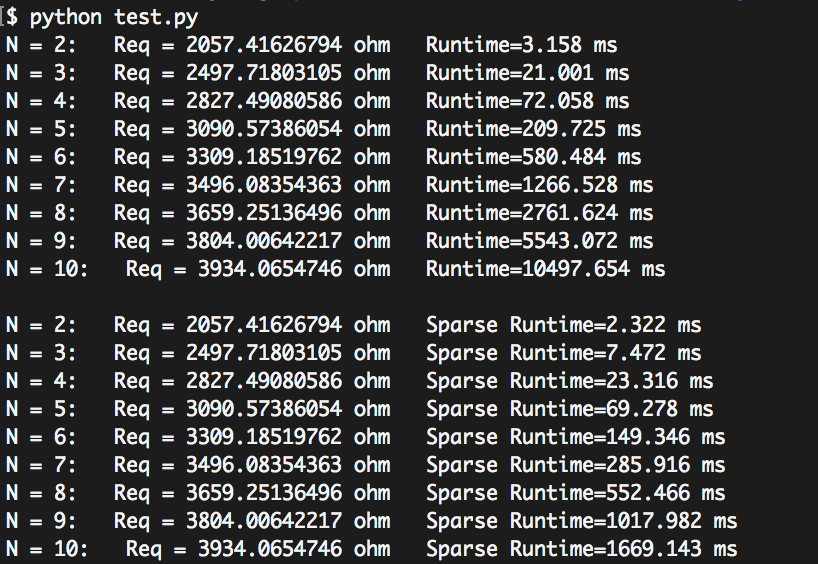
as expected, it is a sparse banded matrix with a maximum of 4 non-zero entries per row.

Please refer to the appendix to all incidence matrices from N = 2 to 10.

1. Program’s runtime

The *Req* for N = 2 …10 and the runtime required are the following:





In theory, the computing time for this operation is *O(n3)* with *n* being the number of nodes.

Therefore, the expected runtime of the program for each N would be the following:

When comparing the outcomes, the runtime of my program is consistent with the theoretical results. They are of the same order, with an error of at most 16% (at N = 10). However, we can note that the practical implementation runtime diverges from the theoretical trend and becomes faster a N grows larger.

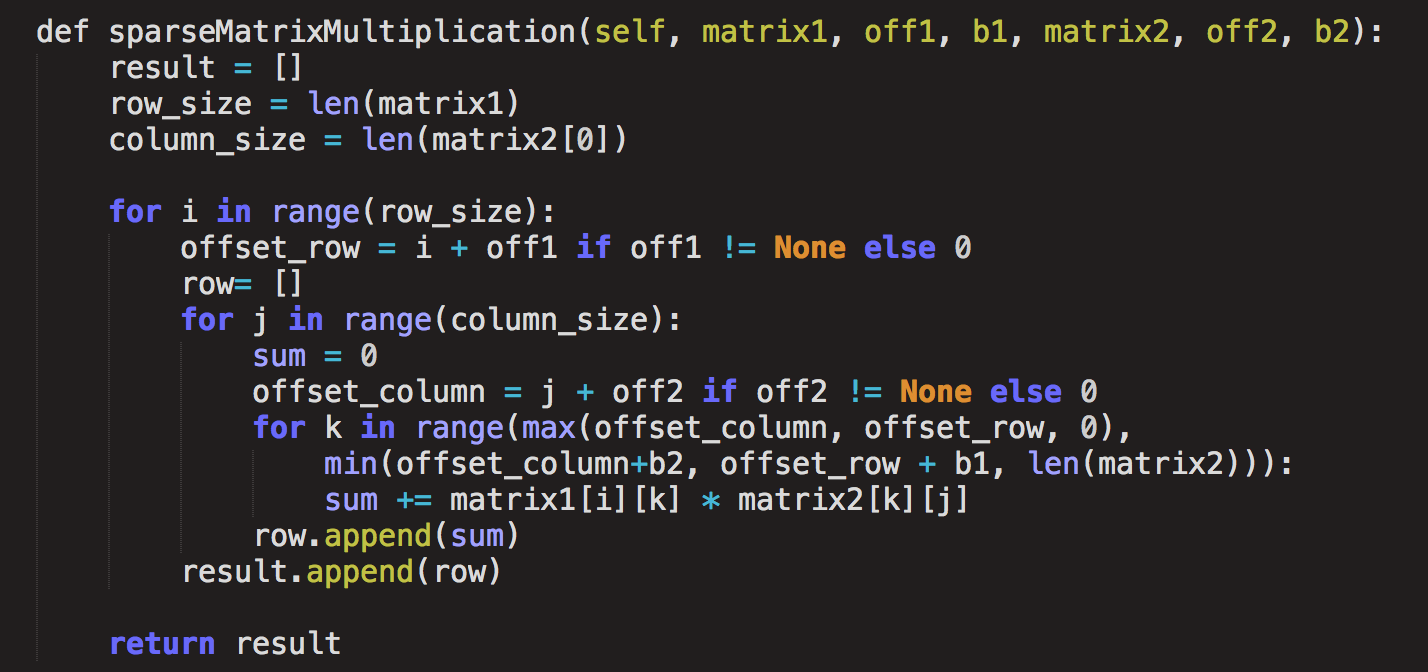
|  |  |  |
| --- | --- | --- |
| N | nodes | O(n3) |
| 2 | 15 | 3 375 |
| 3 | 28 | 21 952 |
| 4 | 45 | 91 125 |
| 5 | 66 | 287 496 |
| 6 | 91 | 753 571 |
| 7 | 120 | 1 728 000 |
| 8 | 153 | 3 581577 |
| 9 | 190 | 6 859 000 |
| 10 | 231 | 12 326 391 |

1. Sparse Matrix

The incidence matrices *A* and the admittance matrices *Y* are greatly sparse and banded.

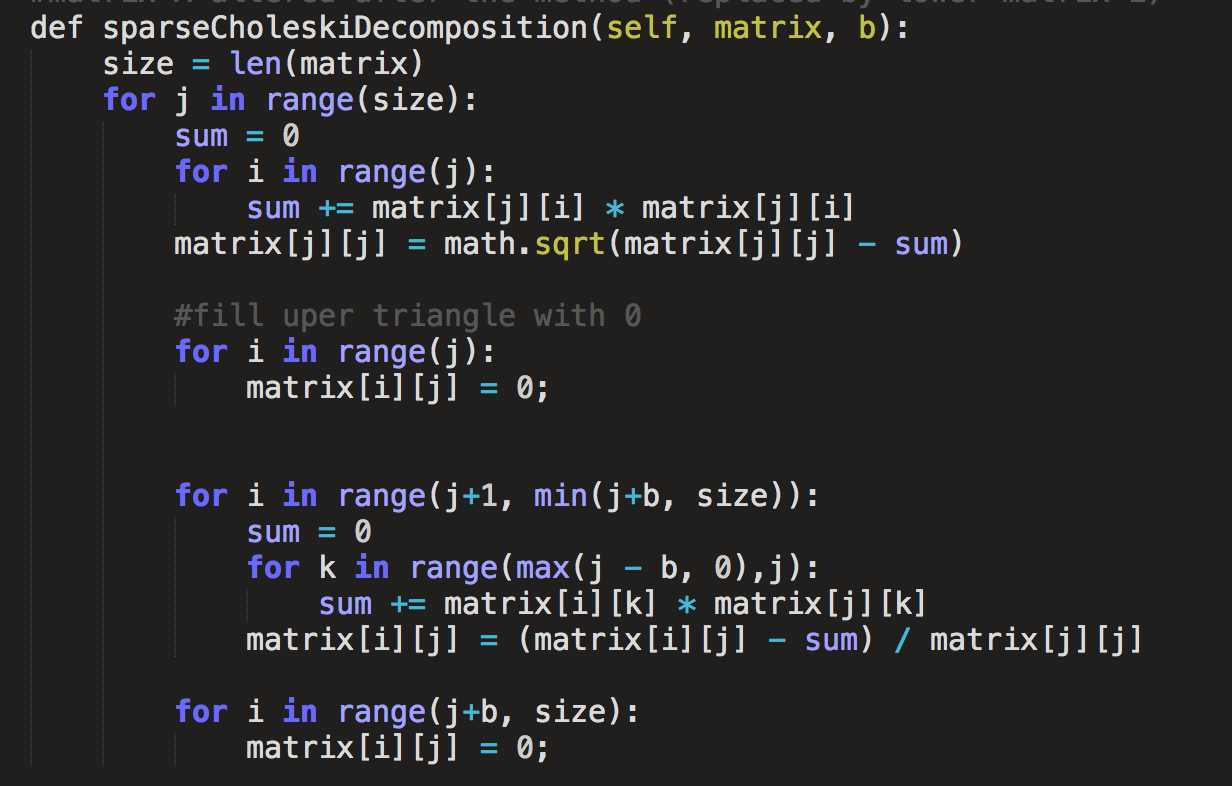
The band of the incidence matrix *An*x*m* is: *bandA = m-n,* and that of *Y* is just 1. Therefore, a lot of runtime can be saved in functions including matrix multiplication and choleski decomposition.

* Matrix multiplication



The function only multiplies and adds the components that are within the band of the both matrices. In the case of multiplying *Y🞄AT*, the sparseness comes in handy because the function only computes one multiplication in each row.

* Decomposition of matrix *A* to lower matrix *L*

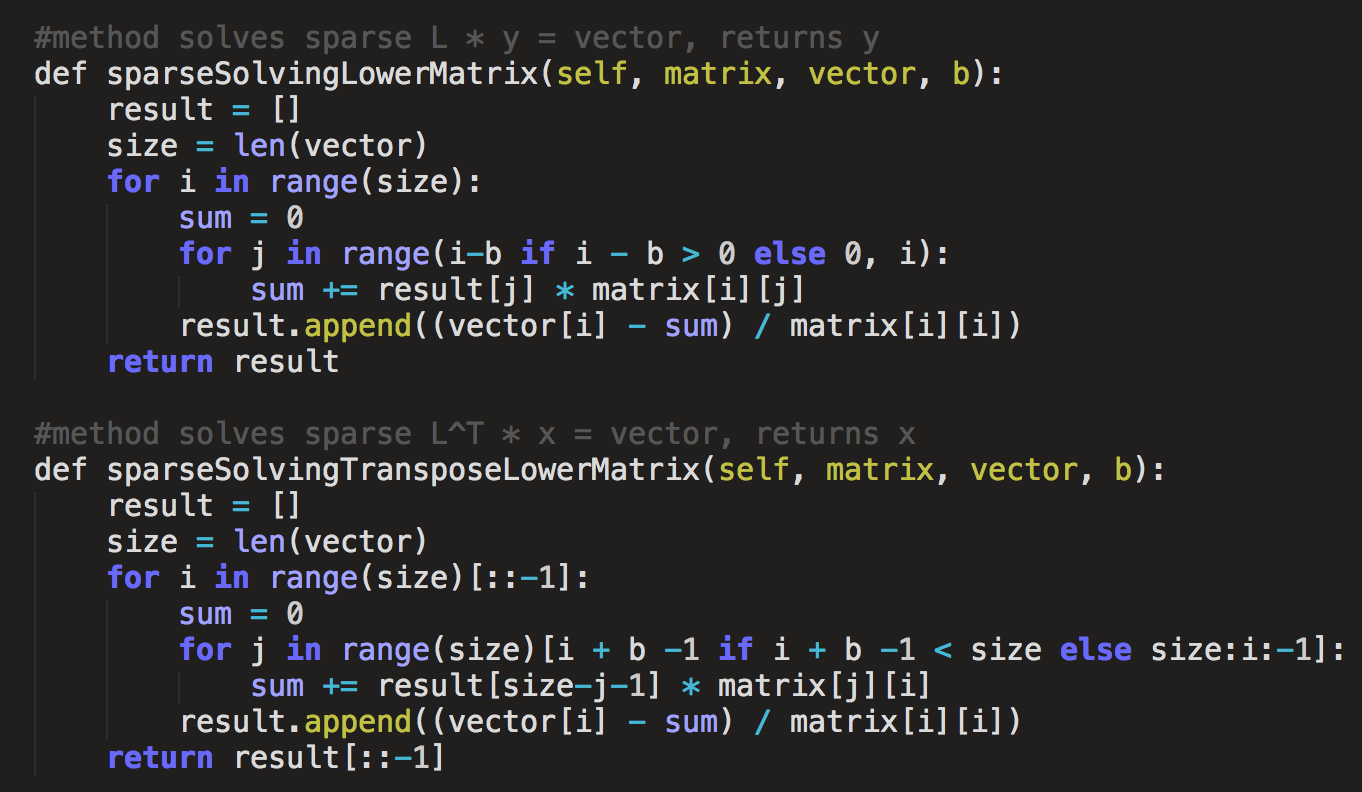


When solving the equation

using choleski decomposition, we aim to decompose into its corresponding lower matrix *L,* which share the same zero values.

In the specific case of this assignment, the half-bandwidth *b* = *N+1.* Omitting the calculation of zero values become very efficient as the size of the incidence matrix grow larger.

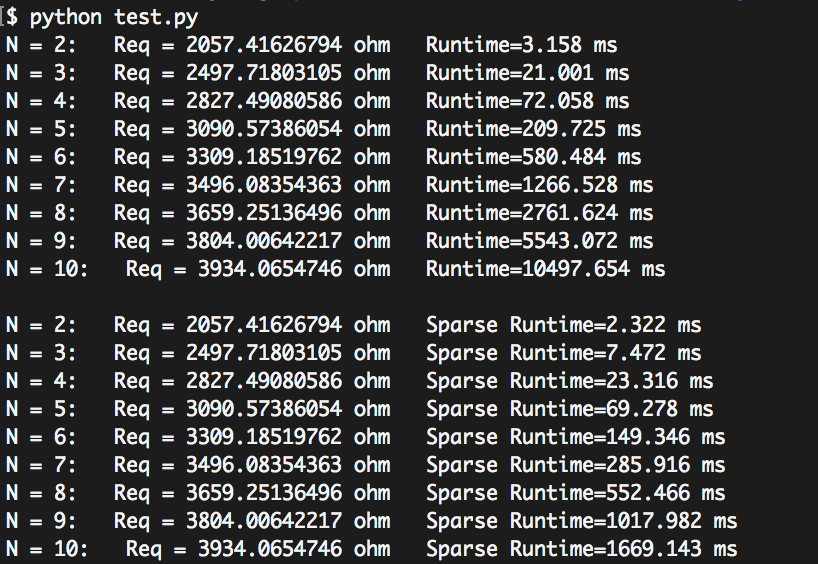
* solving L 🞄 y = b and LT 🞄 x = y



Similarly, the zeroes values of the lower matrices are not included in the calculations.

As a result, the runtimes of the program have reduced significantly, especially as N gets larger. We observe a decrease of 84ms runtime when N = 10.

|  |  |
| --- | --- |
| N | O(b2n) |
| 2 | 240 |
| 3 | 700 |
| 4 | 1 620 |
| 5 | 3 234 |
| 6 | 5 824 |
| 7 | 9 720 |
| 8 | 15 300 |
| 9 | 22 990 |
| 10 | 33 264 |



Theoretically, banded Choleski decomposition has a time complexity of *O(b2n).*

The results of my practical implementation of the sparse Choleski decomposition is not consistent with the theoretical expectations. The rate at which the program’s runtime increases with N is much lower than expected. I think it is due to the time saved in the matrix multiplication when computing that is not taken into account in the theoretical calculations of *O(b2n)*. As the size of the incidence matrix increases, the sparseness of the *Y* matrix contributes to reducing the runtime by almost *O(n2).* Hence why, there is such a big difference between the two results.

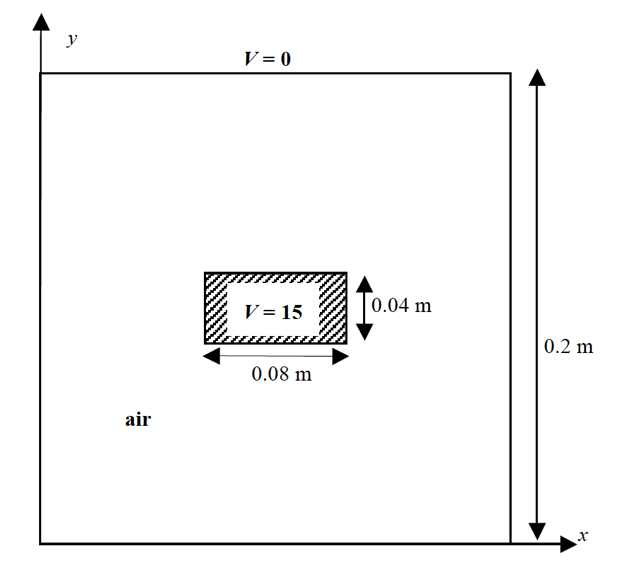
1. *R(N) vs. N*

The equivalent resistance *Req* of the resistor network seem to fit a logarithmic function with respect to N.

As the N goes to infinity, *Req* goes to infinity.

# Question 3

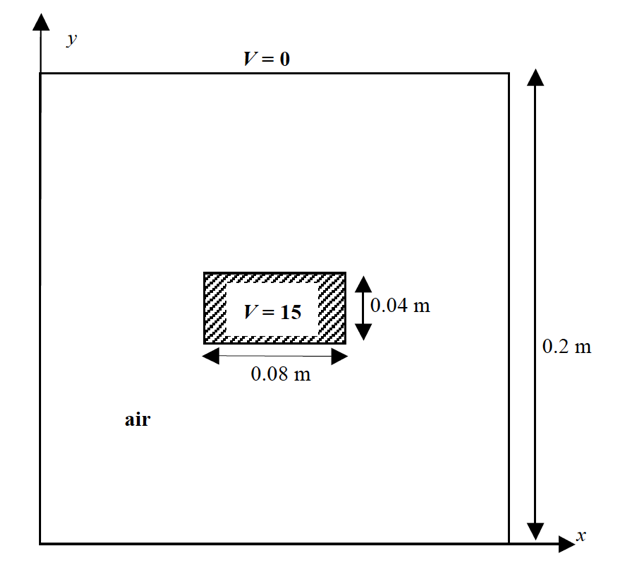
1. Solving with Sucessive Over-Relaxation method



The goal of the problem is to find the potential a node in air within the coaxial cable by the method of finite difference. There are two planes of symmetry in this problem:

*x = 0.1m and y = 0.1m*

Therefore, I focused on solving for the potential of the nodes laying in the bottom left corner of the square grid.



solid lines = fixed nodes / boundary conditions

dashed lines = planes of symmetry

My function takes in several different parameters: matrix grid, booleans determining whether each border is a plane of symmetry, boolean determining if there is a corner with fixed values (in this case the top left corner’s values are fixed), and *x, y* determining the size of fixed valued corner.

The grid holds the potential values at each node, which are updated throughout the method. It has an extra column and row for each plane of symmetry in order to store the false nodes.

The potential of each node is calculated iteratively using the formula:

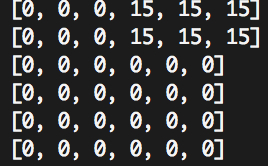
until at each node is less than 10-5.

The order of computation is important. In my program, the potentials are updated from left to right, top to bottom. It is important to note that only the free nodes are updated, that is the not the fixed 15V corner and the boundaries. A special case is considered near the plane of symmetry: the potentials of the false nodes are updated once the new potentials of the corresponding symmetric node is calculated.

Once the Successive Over-Relaxation method is completed. I input the grid into the mapGrid() function, which maps the potentials of all nodes within the whole coaxial cable according to the planes of symmetry.

Example: results of the program with *h = 0.02m*

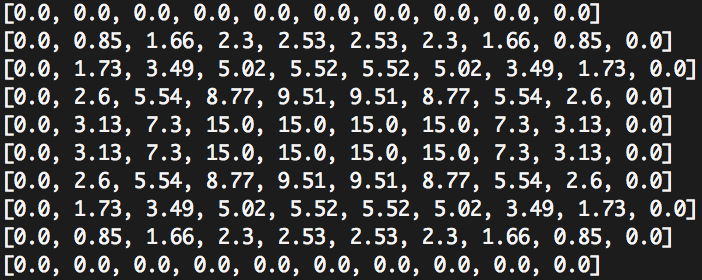
* Construct initial grid with all boundary conditions

All nodes start at 0V potential, except for the fixed boundaries nodes. The extra top row and right column respect the top and right symmetries feature of the grid.

* compute node potentials using SOR method with *w = 10-5*

We can observe that the fixed nodes at at the top right corner and 0V contour remains untouched. As well, rows *i=0* and *i=3* are symmetric over *i=2*, and columns *j=3* and *j=5* are symmetric over *j=4*.

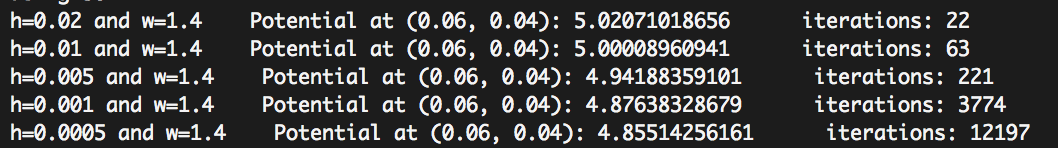
* map the whole grid with respect to its planes of symmetry (top and right in this case)

We can observe that the outer and inner conductor’s potentials are fixed at 0V and 15V respectively. The grid has two plane of symmetry that is between *i=4* and *i=5*, and between *j=4* and *j=5.*

1. SOR with fixed *h* and varying *w*

 The result converges the fastest when *w = 1.3* or *w = 1.4* with 22 iterations.

1. SOR with fixed *w* and varying *h*



The number of iterations is exponentially proportional to *1/h*.

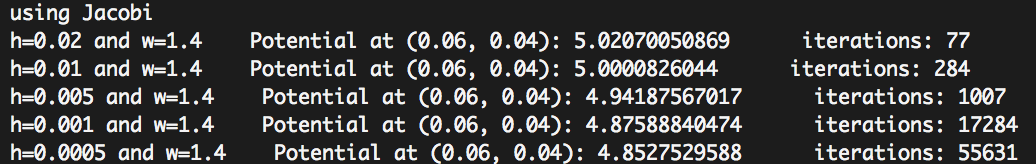
The potential approximation is inversely proportional to the *1/h*.

In general, as the distance between the nodes decrease (and the number of nodes increase), the potential approximation becomes more accurate at the expense of more iterations and computation runtime. The potential at (0.06, 0.04) seems to converge to 4.85V.

1. Solving with Jacobi with varying *h*

The set up to solve the problem with Jacobi method is the same as the one for SOR. However, the iterative formula is now:

The results of the Jacobi iteration method are the following:



The number of iterations is exponentially proportional to *1/h*.

The potential approximation is inversely proportional to the *1/h*.

|  |  |  |
| --- | --- | --- |
| 1/h | SOR iterations | Jacobi iterations |
| 50 | 22 | 77 |
| 100 | 63 | 284 |
| 200 | 221 | 1007 |
| 1000 | 3774 | 17284 |
| 2000 | 12197 | 55631 |

The accuracy of the potential approximation is similar to the results obtained through SOR method; the difference is less than 1%. However, it is importantly to note that in order to obtain the same accuracy, the Jacobi method requires many more iterations. As h gets small, the computation time required by the Jacobi method is significantly higher than that of SOR.

1. Non-uniform spacing grid

In order to obtain a more accurate potential at (0.06, 0.04), I constructed the grid such that the spacing between the nodes are much smaller near the point of interest. I stored the *x* and *y* coordinates of each row and column lists for easier reference. The set up of the Jacobi function is similar to that of SOR, with the iteration formula being:

with = distance between *i-1* and *i*

= distance between *i* and *i+1*

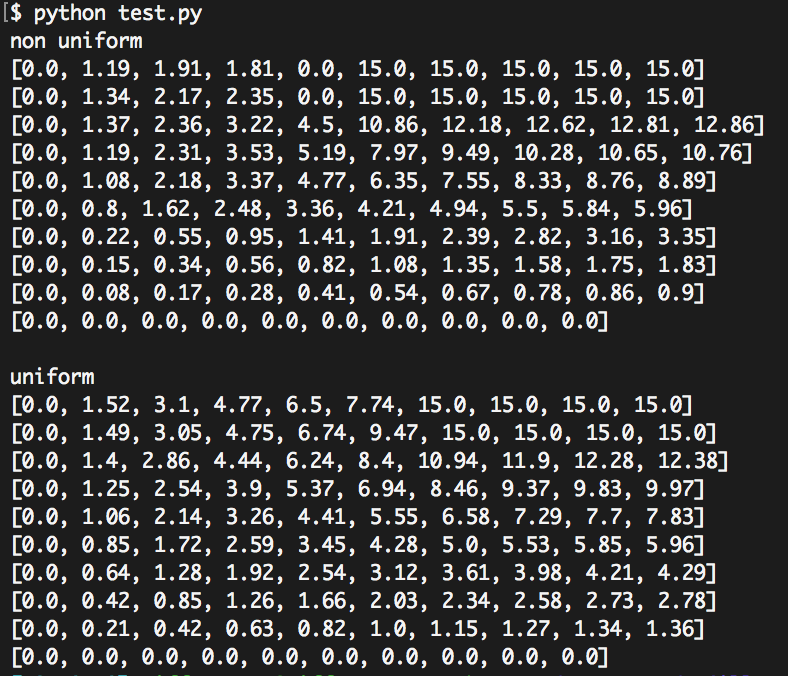
= distance between *j-1* and *j*

= distance between *j* and *j+1*

The outcome of the non-uniform spacing grid using the following coordinates:

*i\_coord* = [0, 0.025, 0.035, 0.04, 0.045, 0.05, 0.06, 0.075, 0.09, 0.1, 0.11] #focus on y

*j\_coord* = [0, 0.02, 0.04, 0.05, 0.055, 0.06, 0.065, 0.07, 0.085, 0.1, 0.115] #focus on x



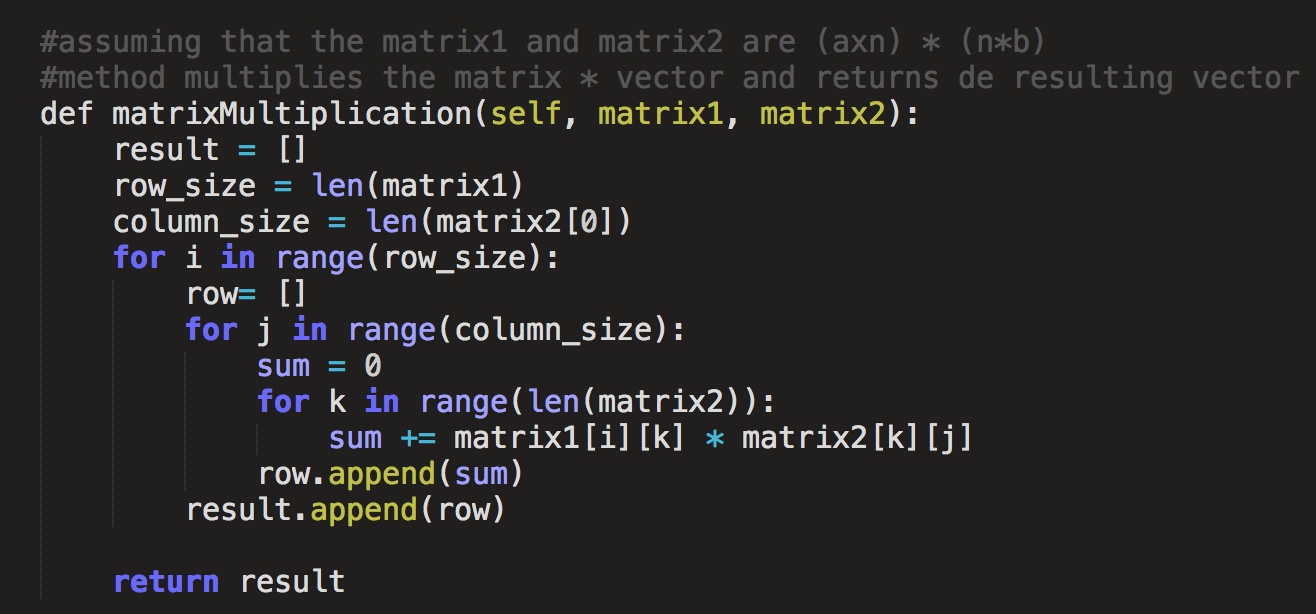
As found in part c) the potential at (0.06, 0.04) converges to 4.85V. The output of the non uniform spacing grid shows a more accurate approximation of the potential, 4.94V *vs.* 5.0V.

Considering the same amount of nodes, the non-uniform spacing method can approximate the potential of a specific point at a higher accuracy, whereas the uniform spacing method finds the potentials of all nodes on the map with equal accuracy.

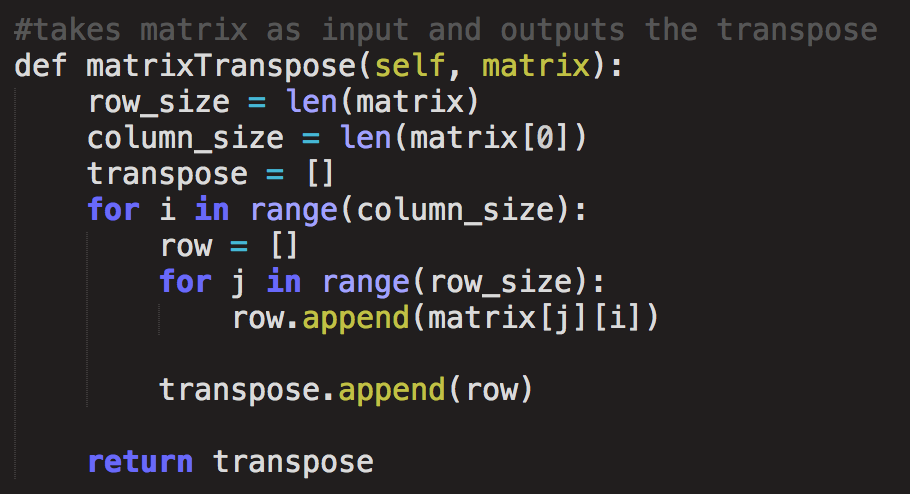
# Appendix

1. Matrix Manipulation

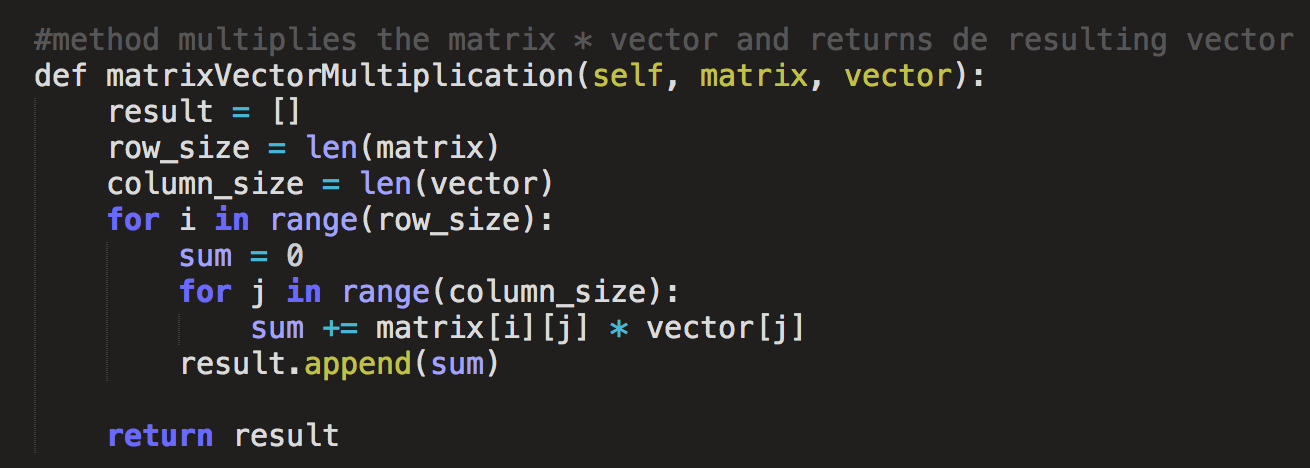
## Matrix Multiplication:



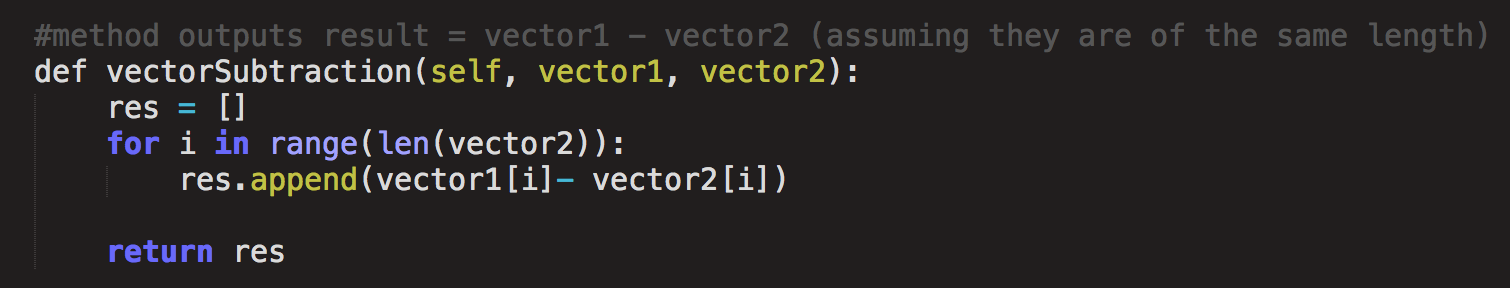
Matrix Transpose:



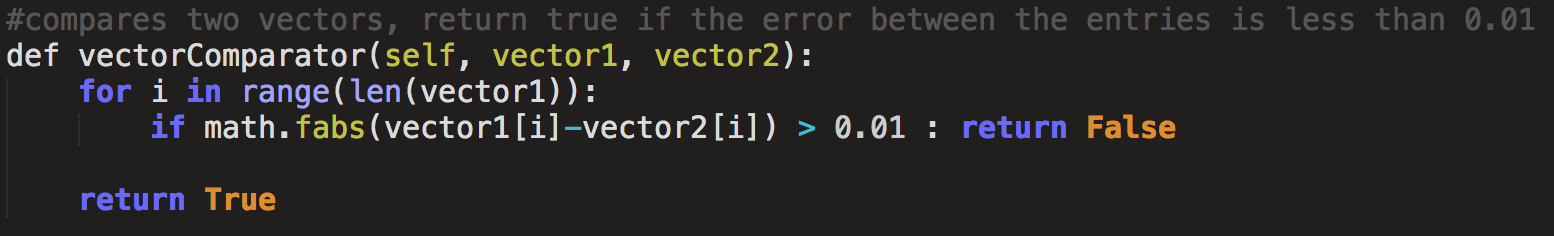
## Matrix – Vector Multiplication:



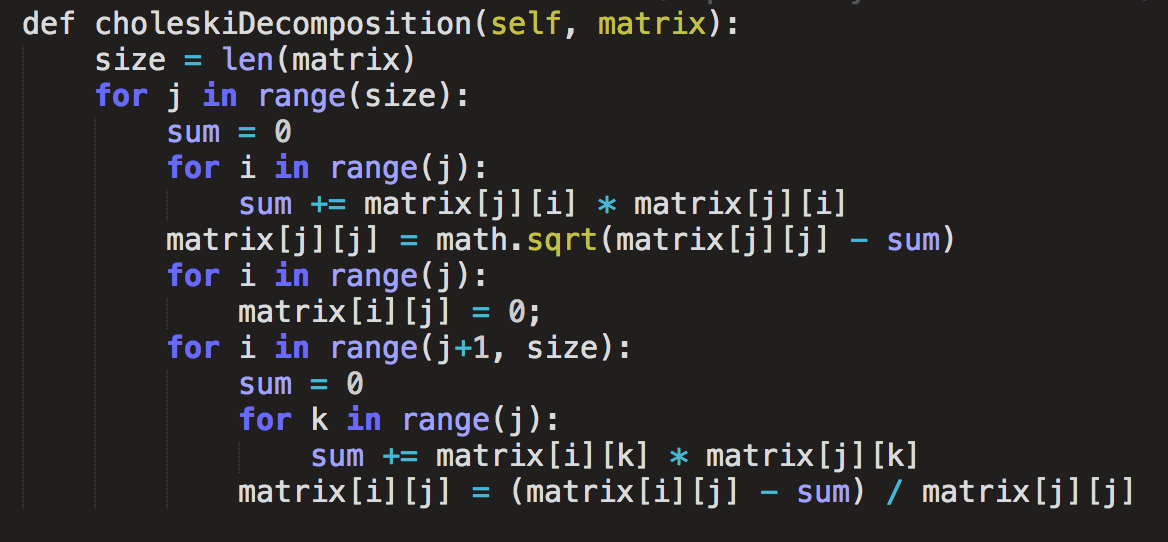
## Vector Subtraction:



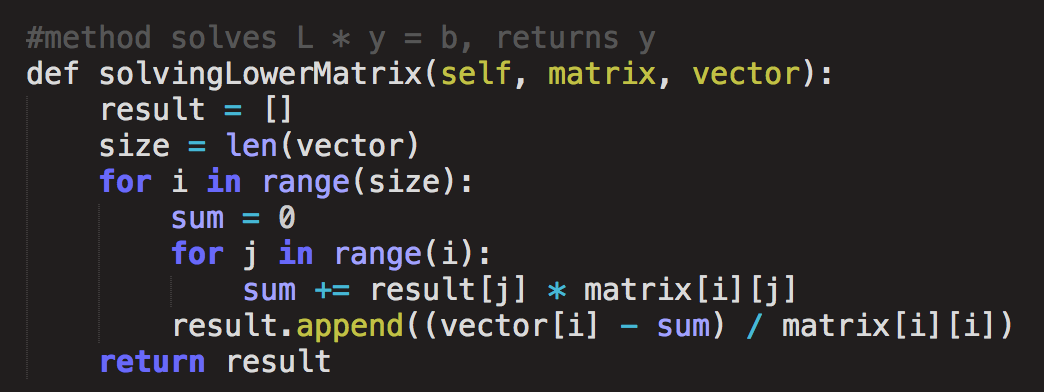
## Vector Comparator:



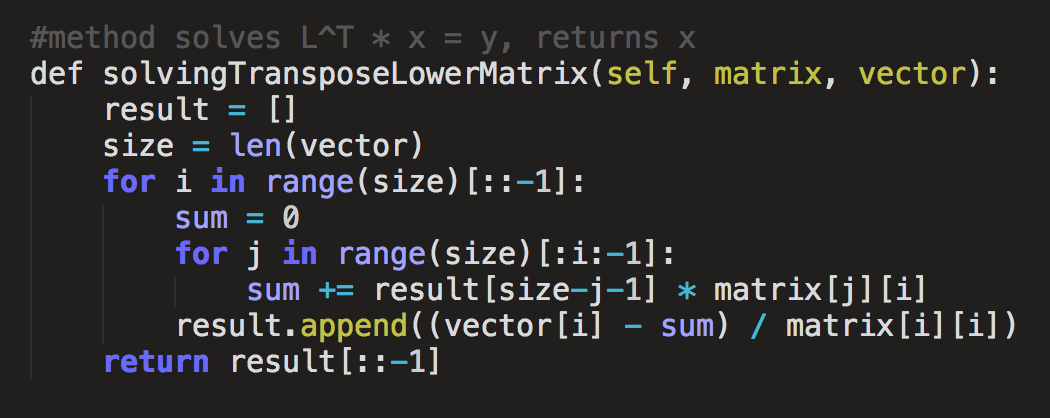
## Choleski Decomposition to LowerMatrix *L*:



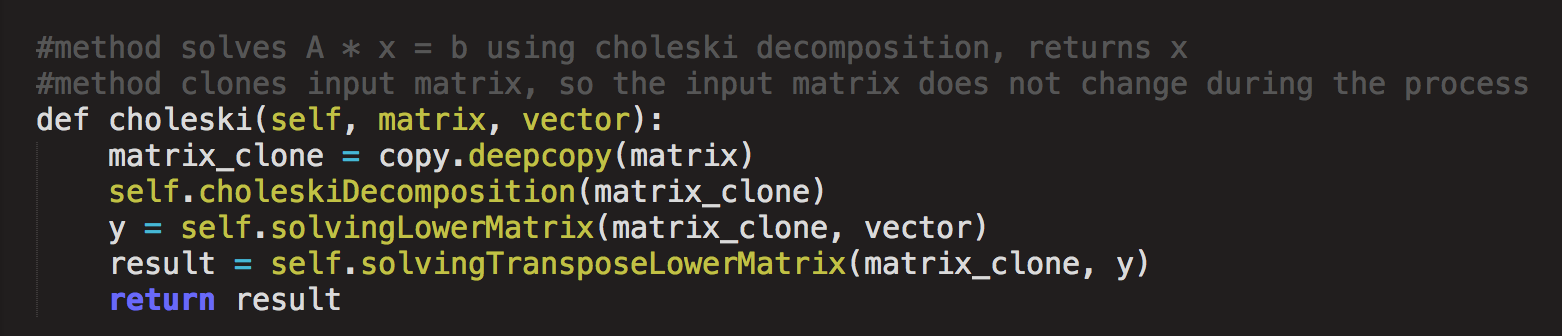
## Solves *L🞄 y =b*:



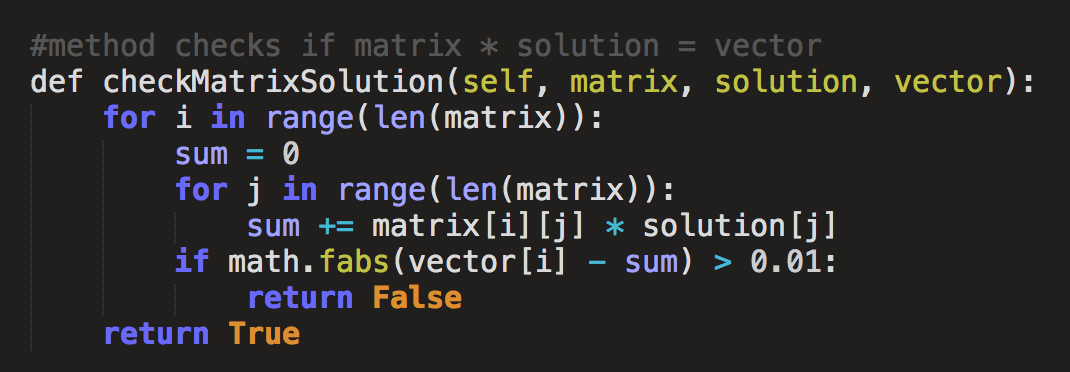
## Solves *LT🞄 x =y*



Choleski Decomposition solving *A🞄 x =b*:

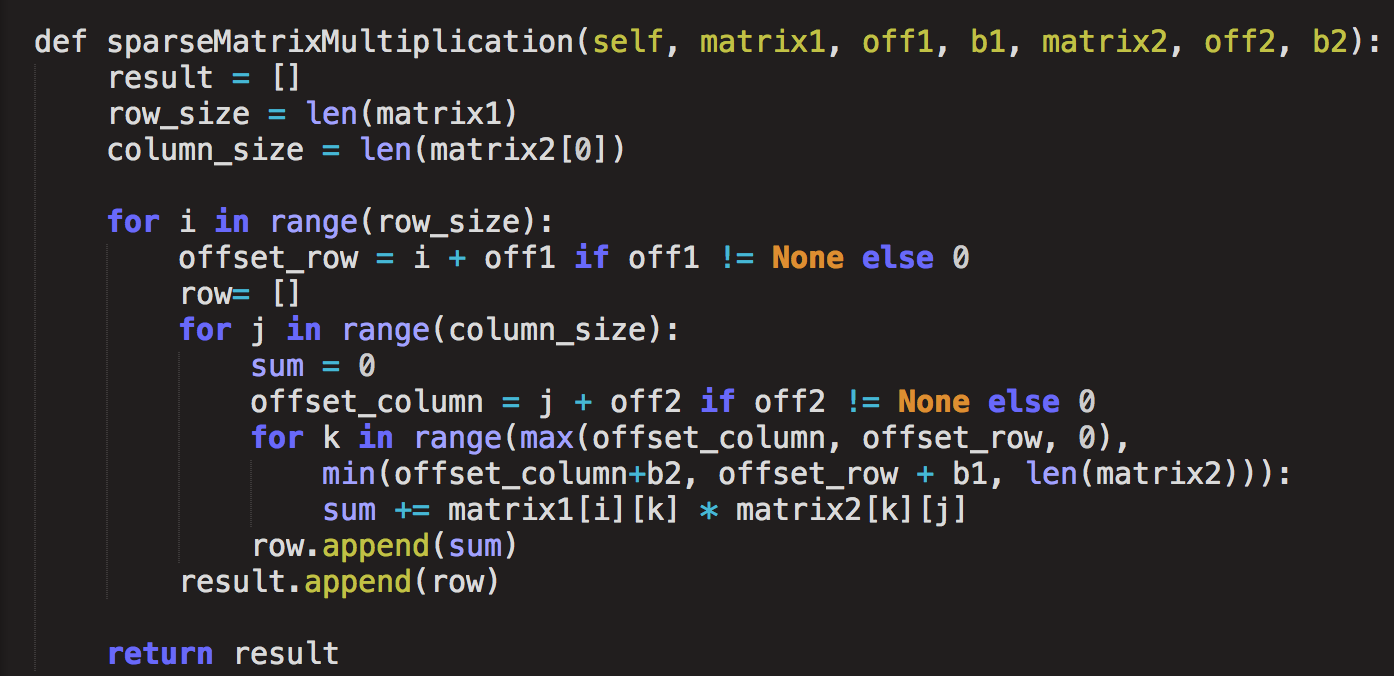


## Check if *A🞄 x =b* is true:

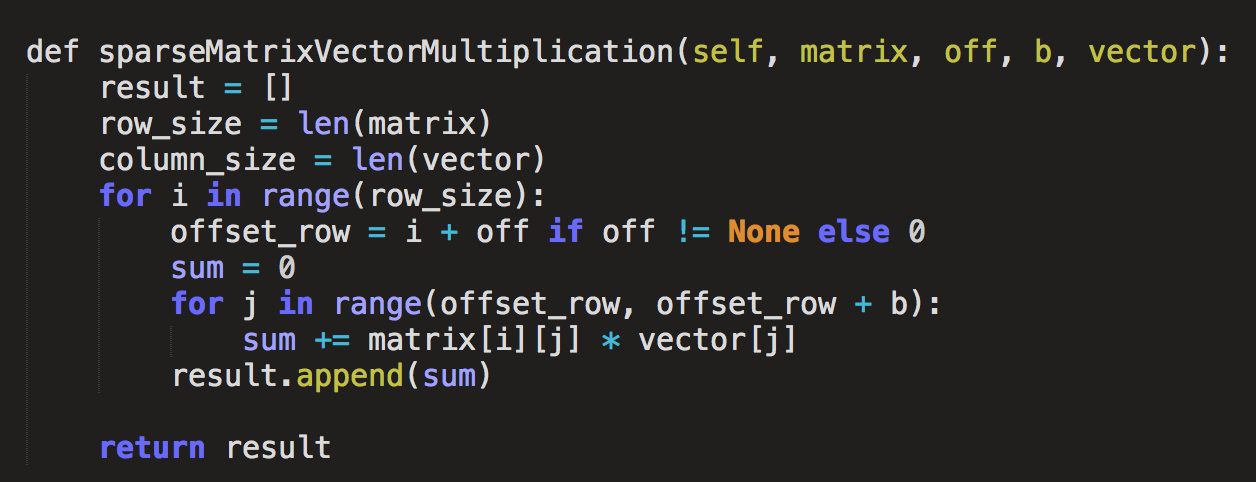


1. Sparse Matrix Manipulation

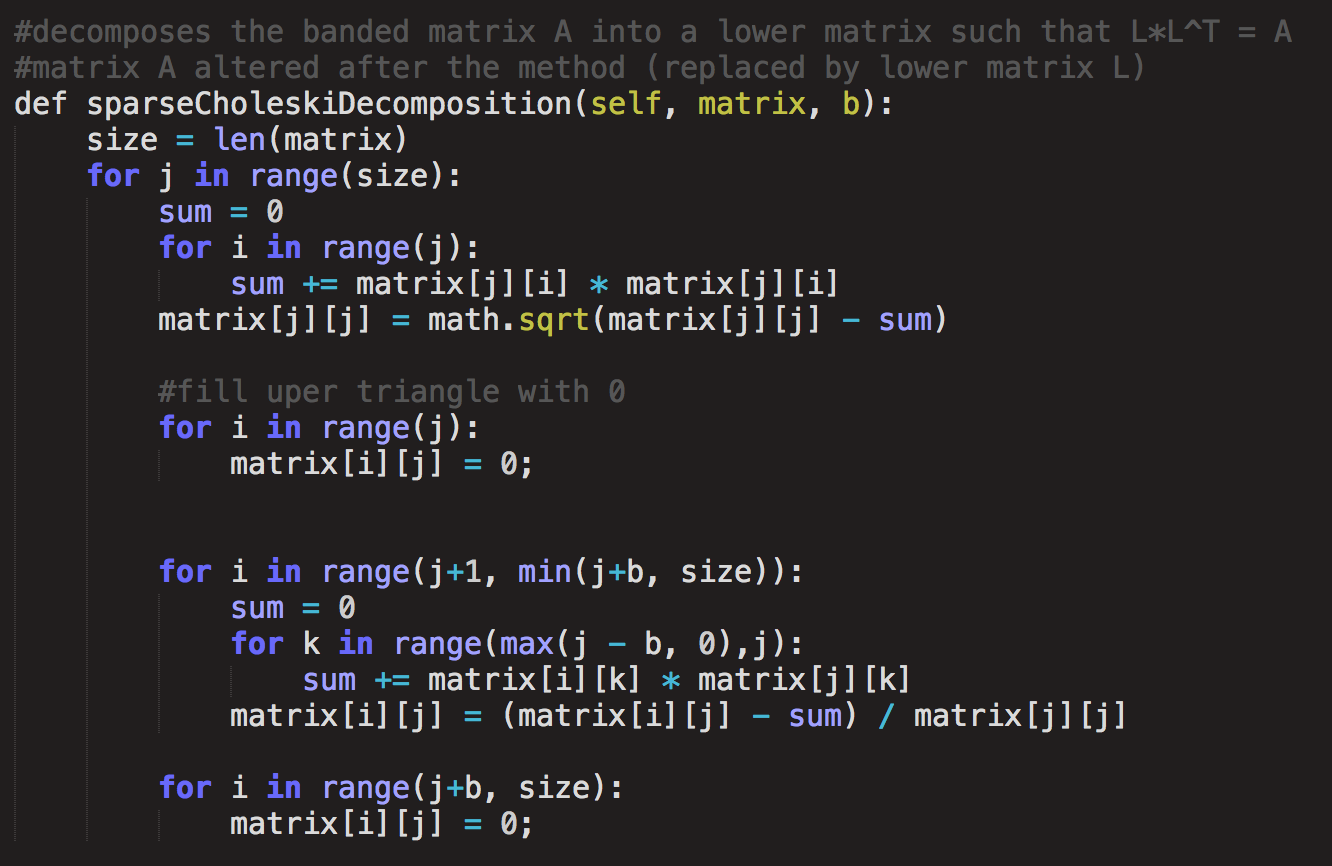
Sparse Matrix Manipulation:



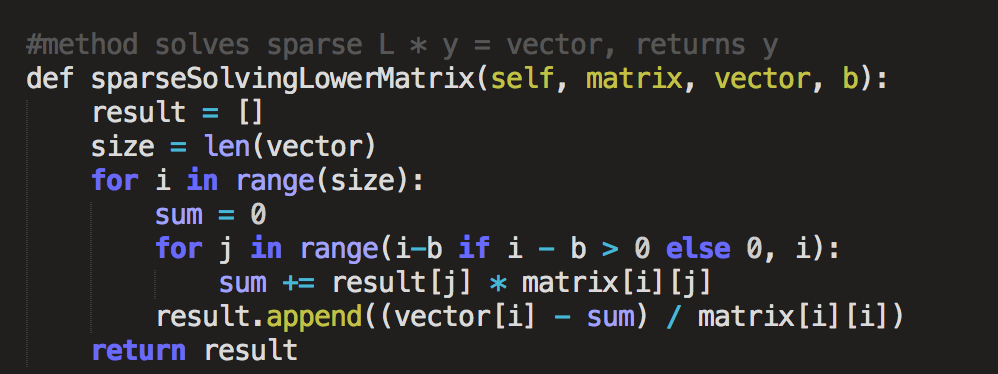
## Sparse Matrix – Vector Multiplication:



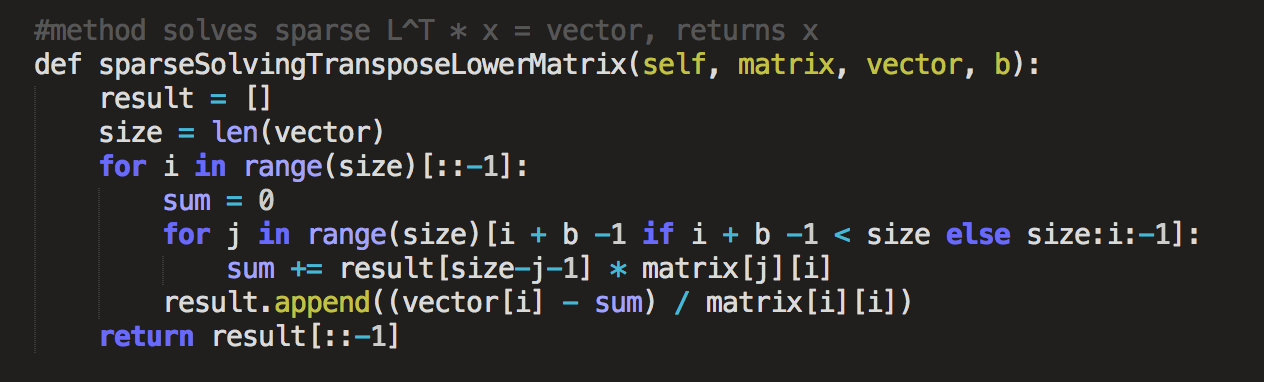
## Sparse Choleski Decompose to LowerMatrix *L*:



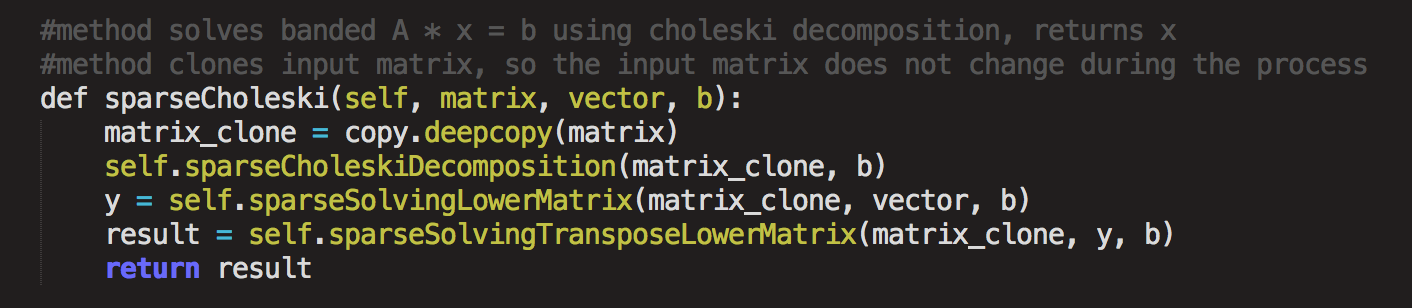
## Solves *L🞄 y =b* where L is a sparse lower matrix:



## Solves *LT🞄 x =y* where *LT* is a sparse upper matrix:

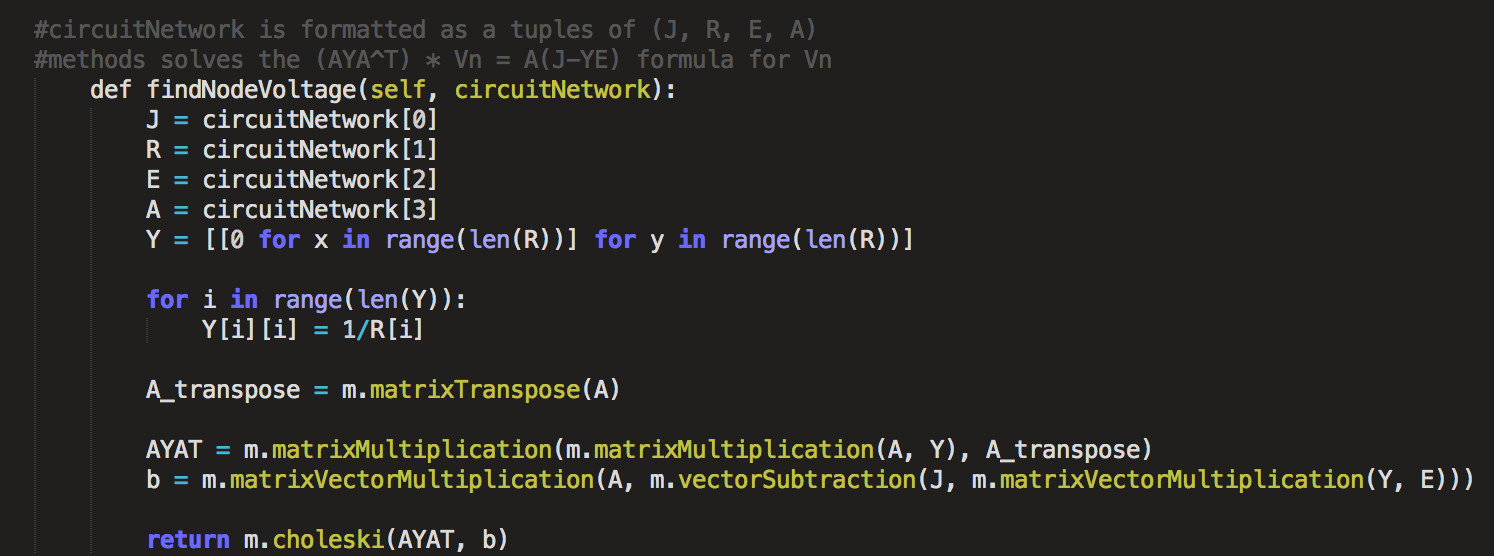


## Solving *A🞄 x = b* where *A* is a banded matrix

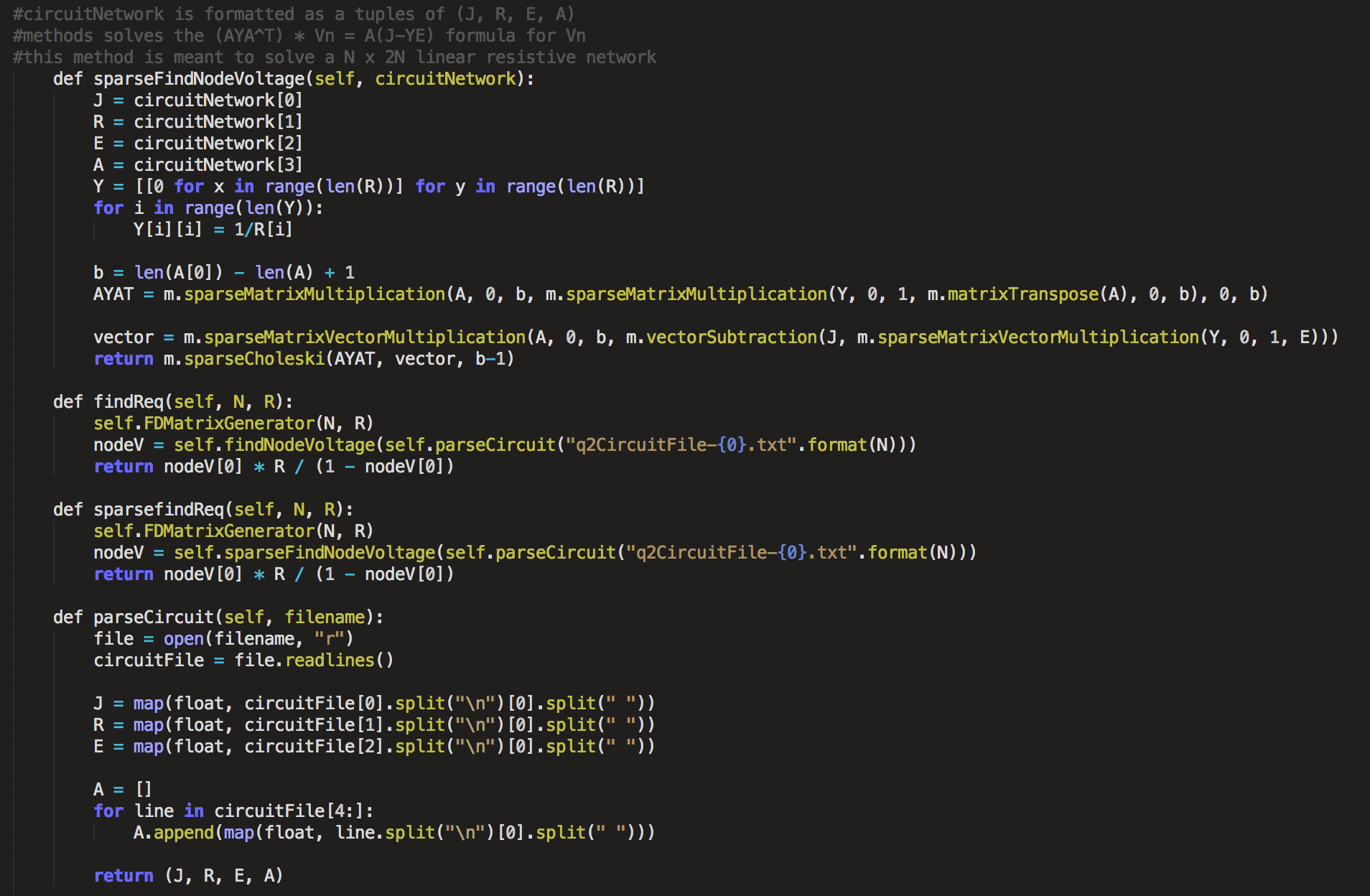


1. Circuit Analysis

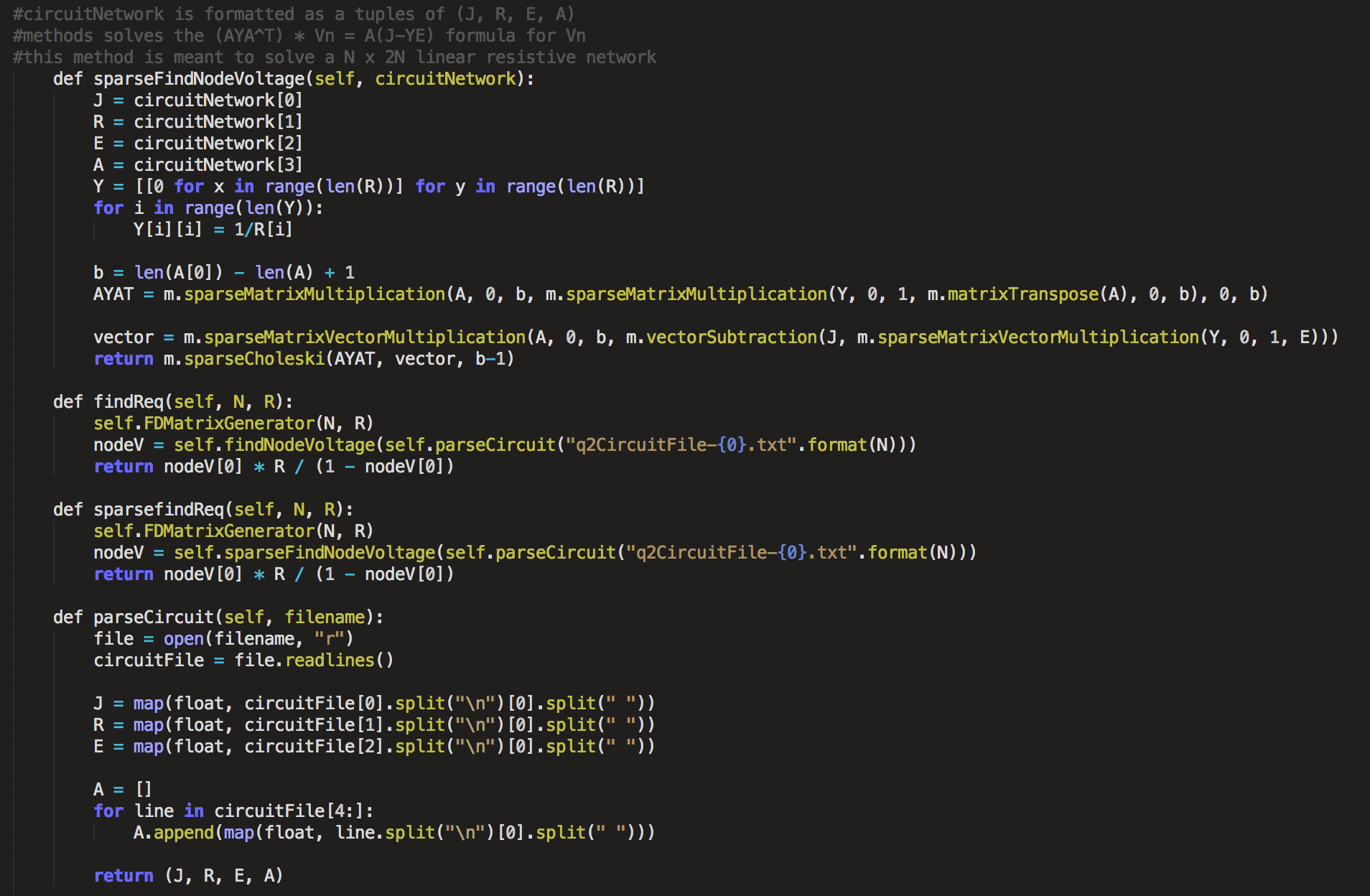
## Find Node Voltage:



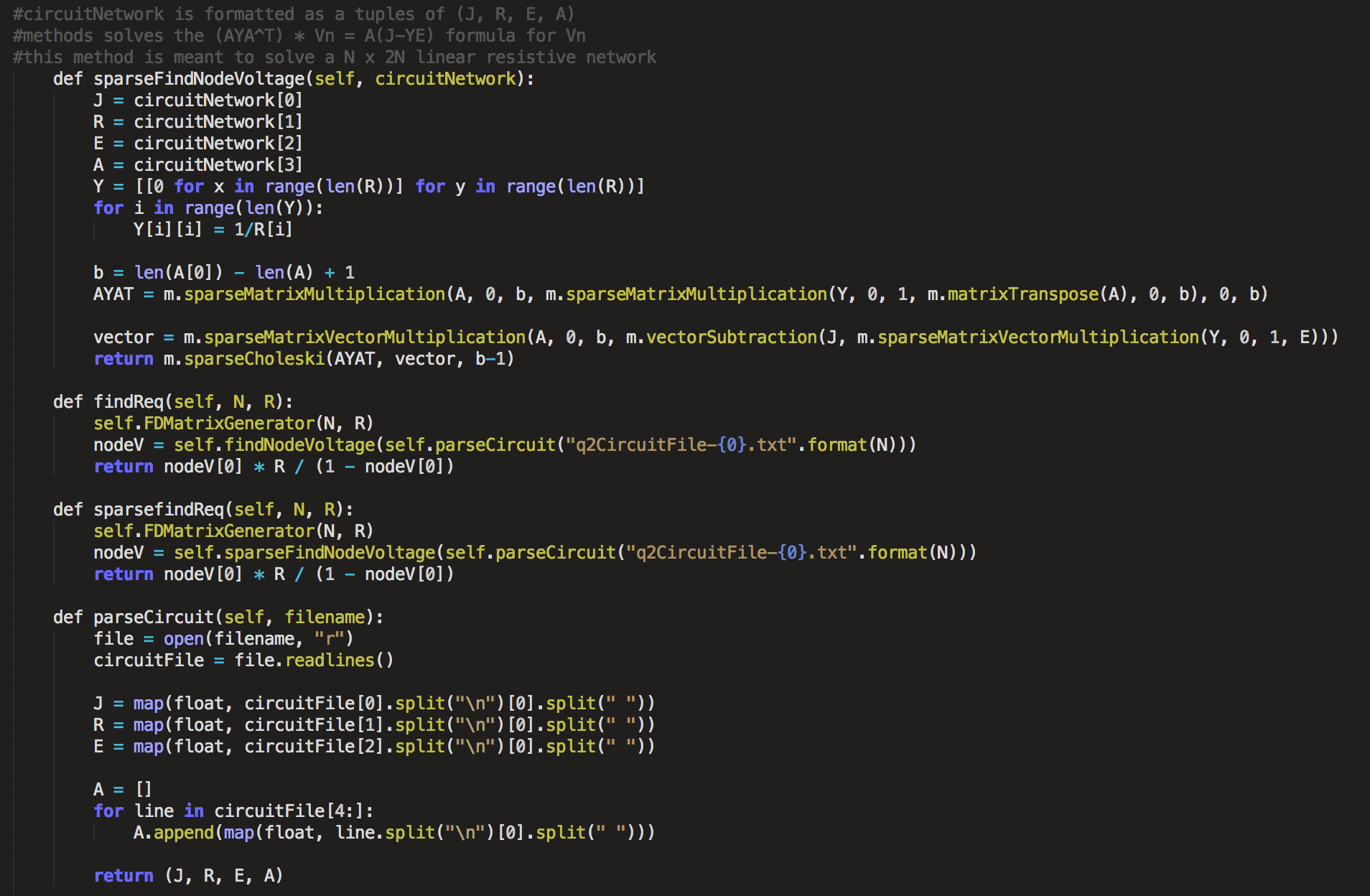
## Find Node Voltage using sparse properties of matrices:



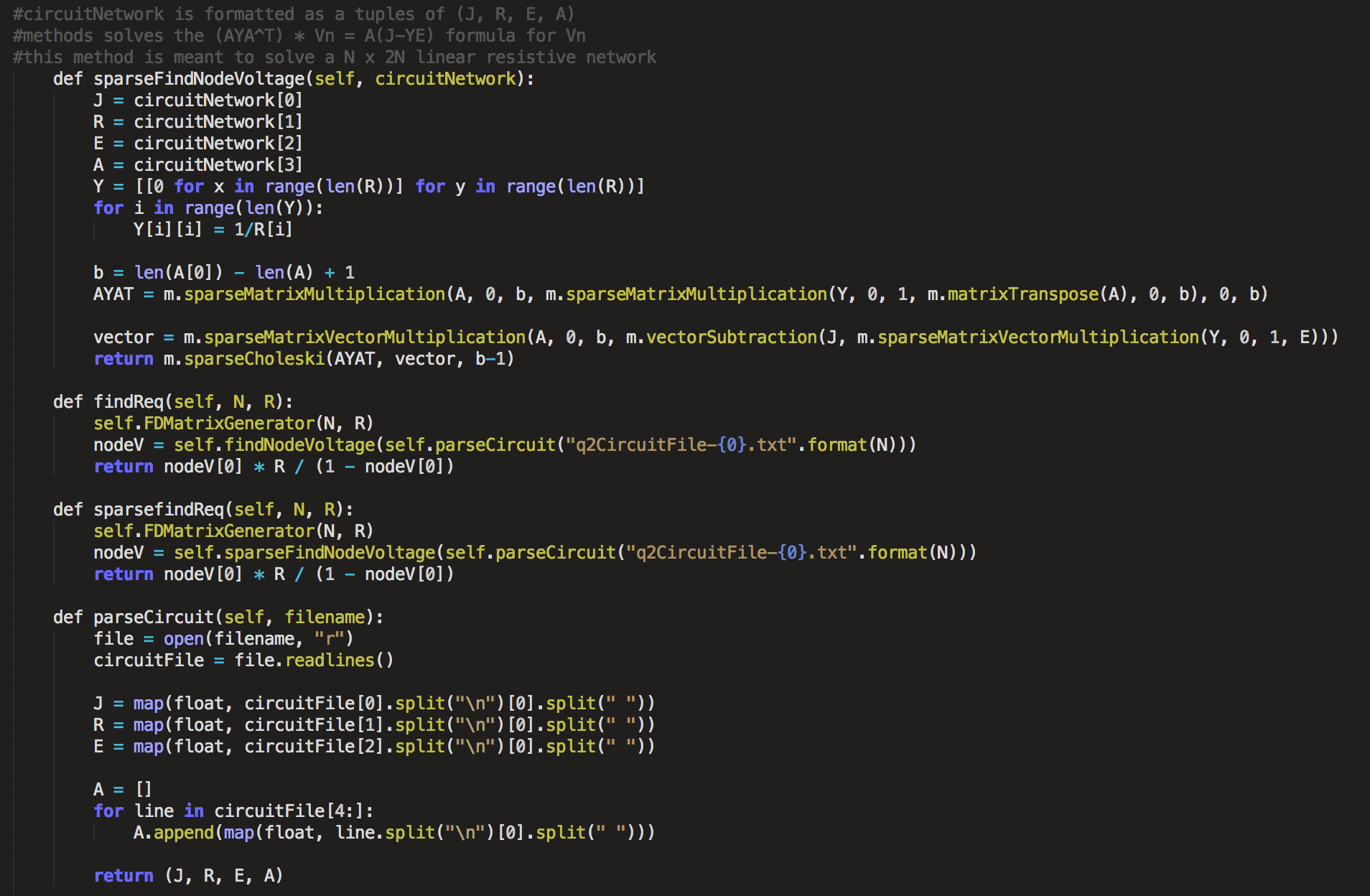
## Find *Req*:



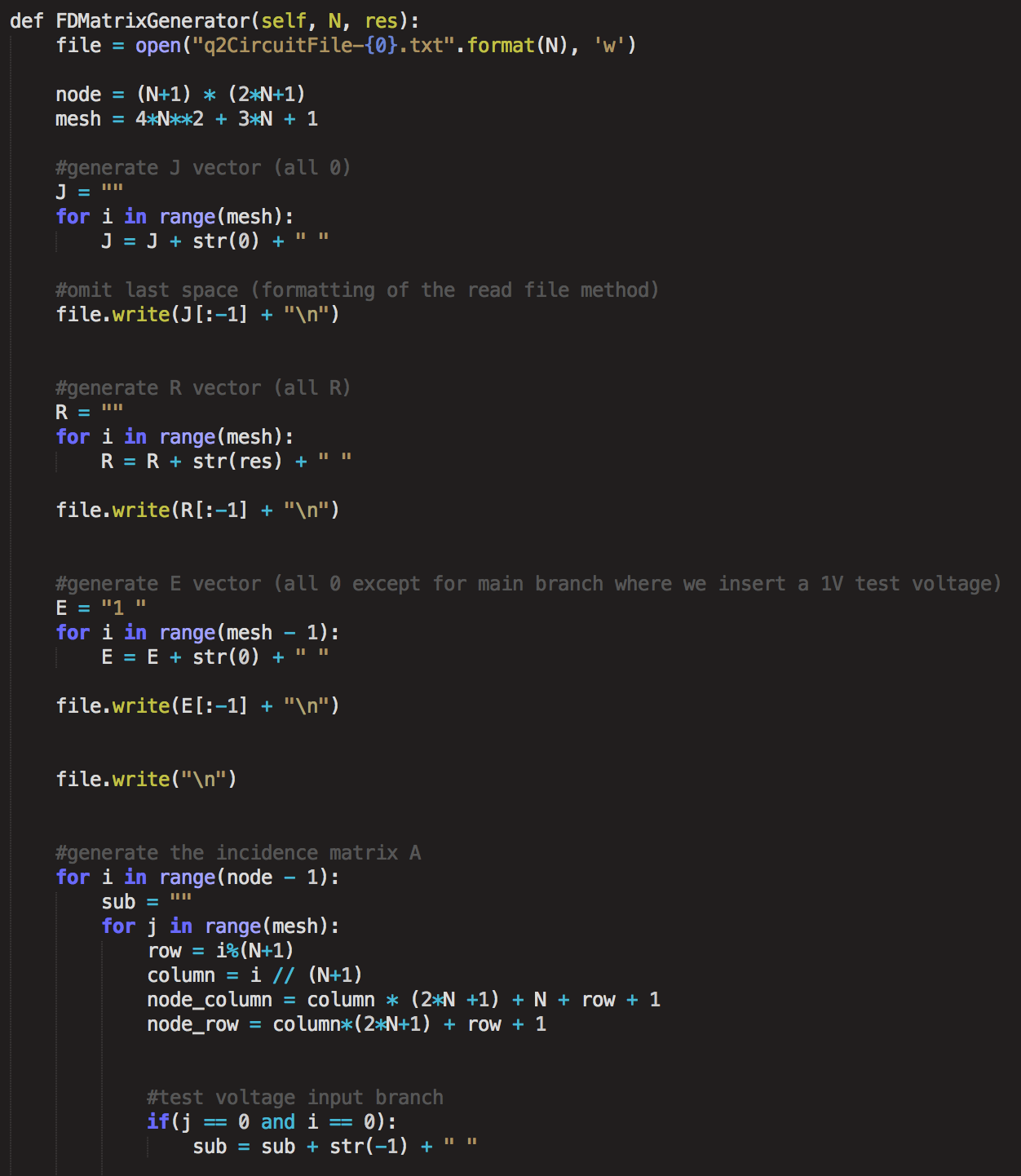
## Find *Req* using sparse properties of matrices:

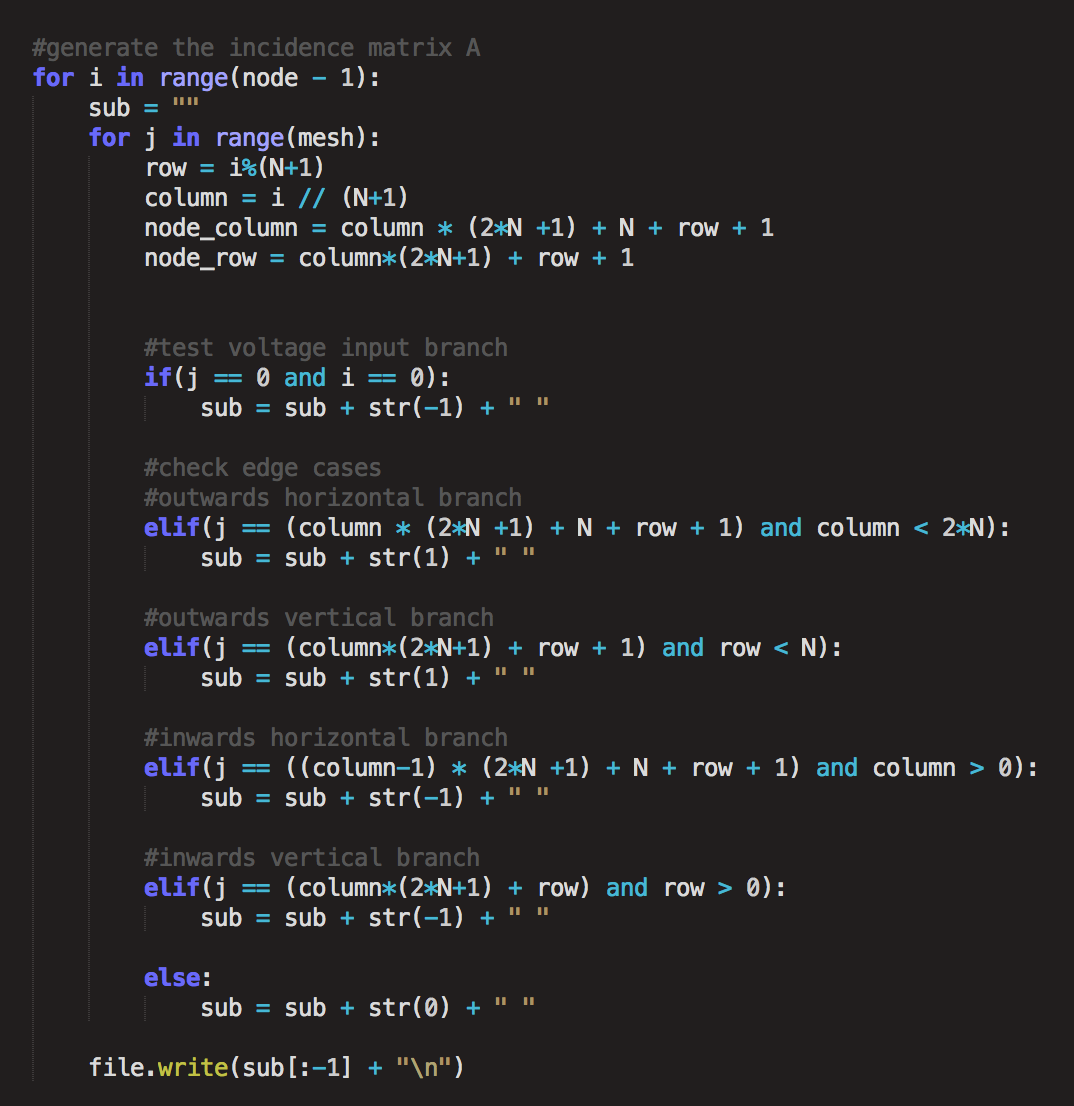


## Parse circuit file to circuit network object:



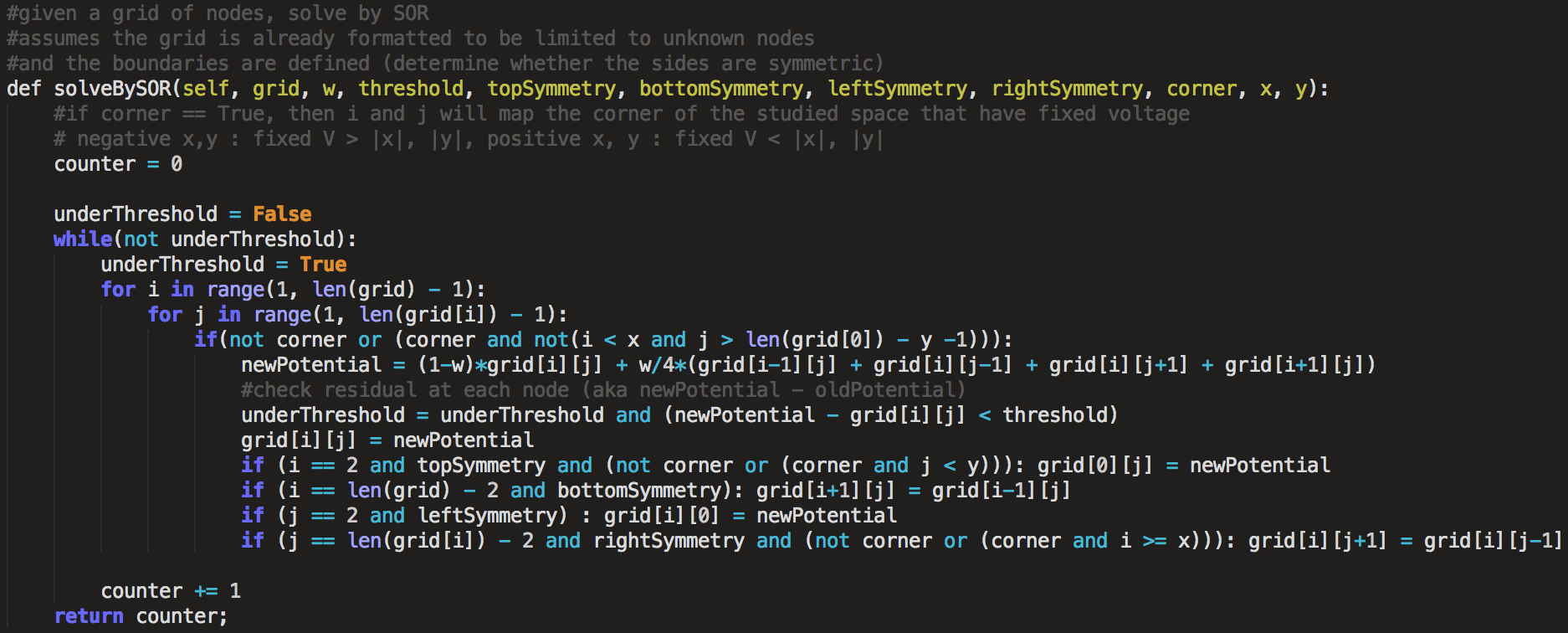
## Resistor Network circuit file generator:

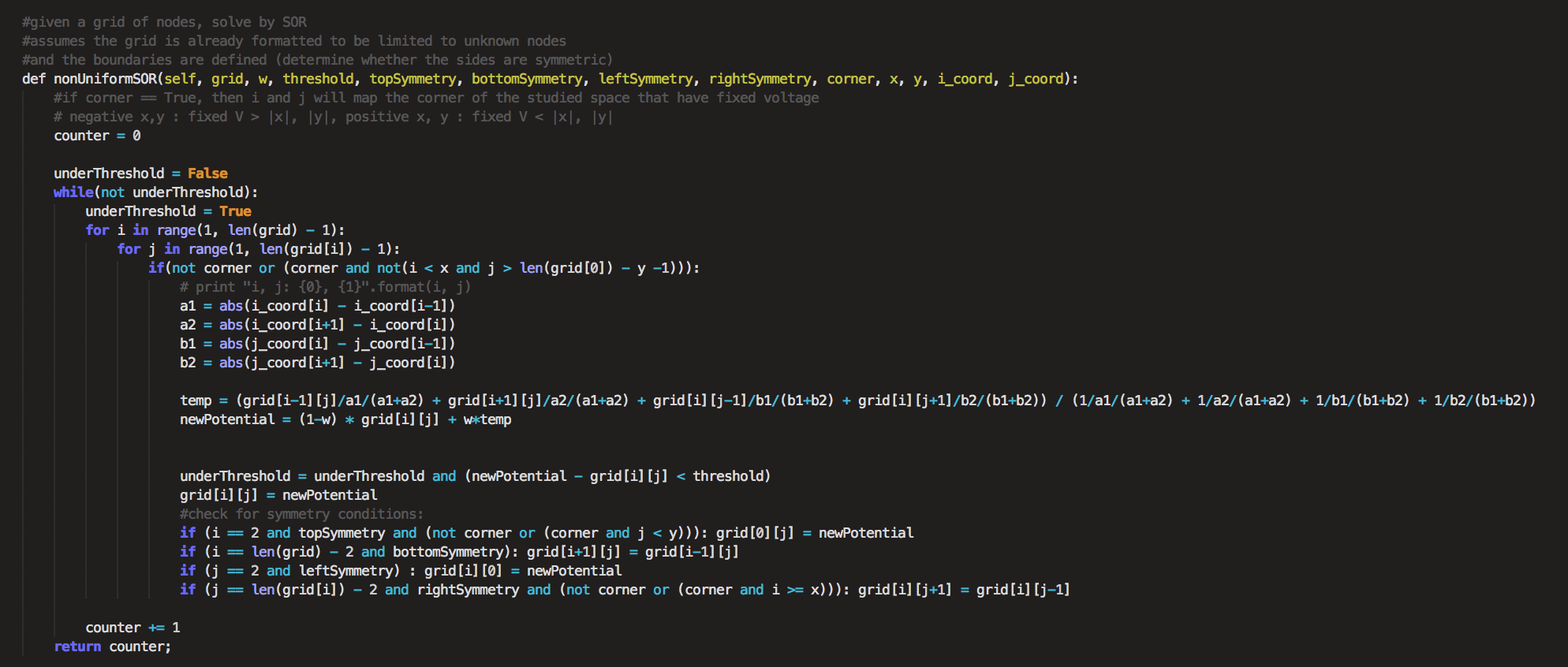




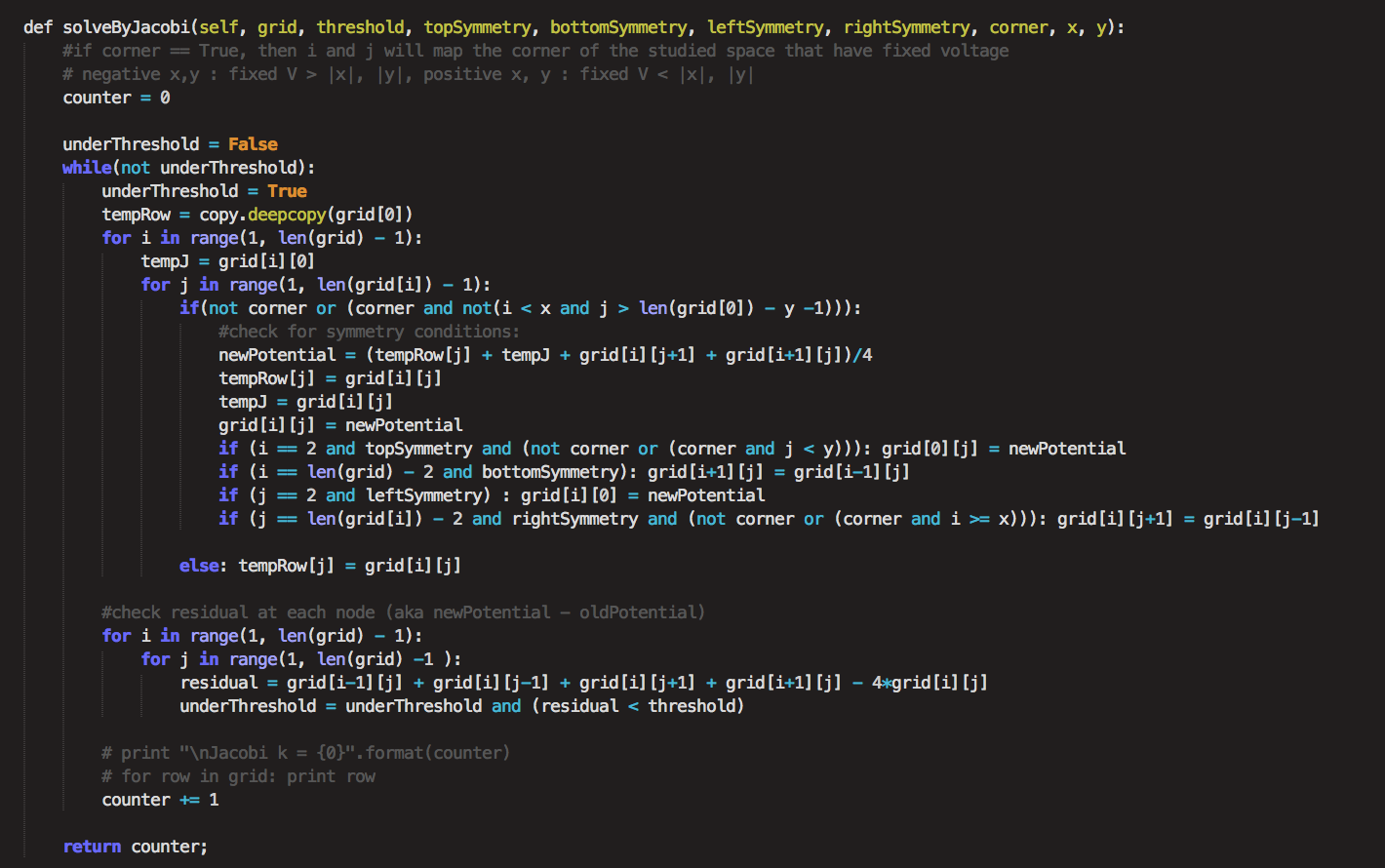
1. Iterative Finite Difference Methods

## Uniform Spacing Successive Over Relaxation Method:



Non-Uniform Spacing Successive Over Relaxation Method: 

## Uniform Jacobi Method:



## Mapping Grid Over Planes of Symmetry:

