Justin Sage

Groundwater Modeling Documentation

OVERVIEW

GWGrid class

This class is stored in the GWGrid.py file and acts as the main driver for the operations. There are multiple functions within this class that are used in order to perform the calculations such as fraction through. Flow data can be read in using either text files or binary files. This data is then stored in a 3d array of classes. Each element of the array contains multiple values designating flow, node naming, and calculation values. This class interacts with the ADJlist class by feeding it the 3d array of flows as an adjacency list.

ADJlist class

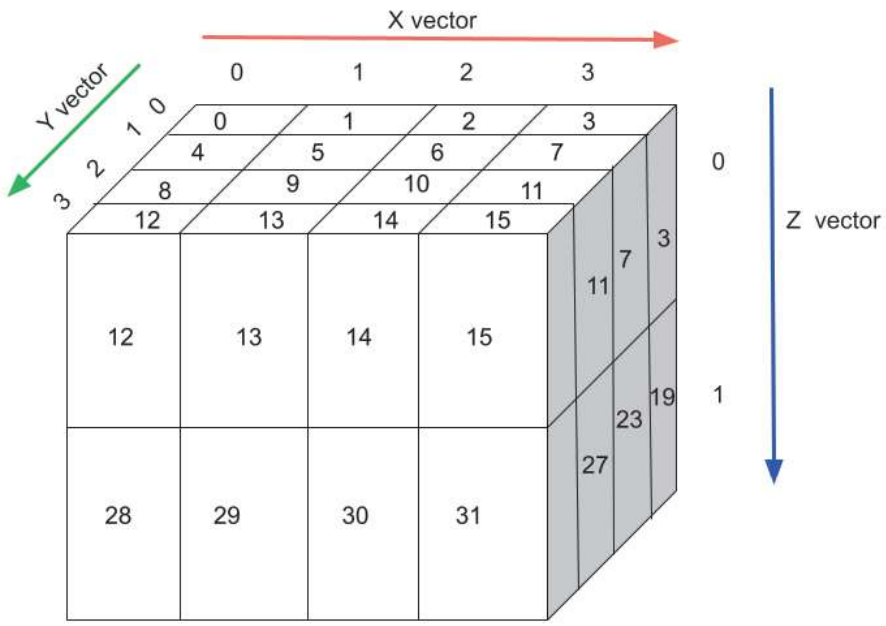
This class is stored in ADJlist.py. This class is primarily used as a helper class for GWGrid. This class uses a given adjacency list to perform topsort and return the ordering of the nodes. This class also has the ability to find all nodes reachable by a specified node.

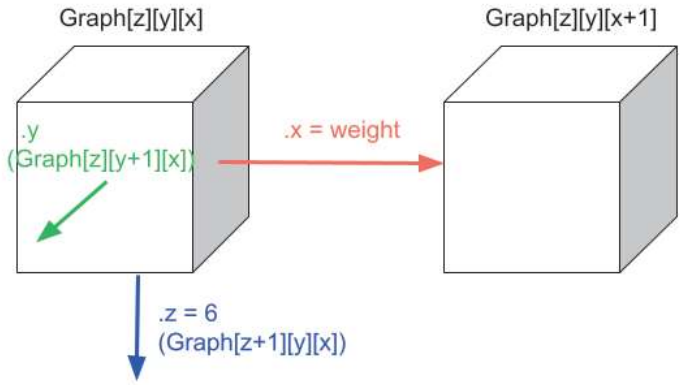
GWGRID

Variables

The main variables of this class are:

- GWGraph
    - The 3d array that is to be populated with values of class Node
    - Each node contains the flow in x,y,z directions, the nodes name(an int representing the node starting at 0), and the node_frac_through (calculated value for fraction through, initialized to 0).





- adjlist
    - An object that is initialized to none, but will contain the initialized object representing the adjlist class after running the create_adjlist or create_adjlist2 functions (see function descriptions below)
- topsortList
    - This is a list initialized to none and will contain the node names in topological sorted order after running the ADJlist classes topological_sort. This list is populated in the GWGrid's topsort function which just executes the ADJlist's topological_sort and stores the result.
- depth

    - The depth variable represents the depth in layers of the flows. This value represents the length of the z vector in the above 3d array graphic
- length
    - The depth variable represents the length in layers of the flows. This value represents the length of the x vector in the above 3d array graphic
- width
    - The depth variable represents the width in layers of the flows. This value represents the length of the y vector in the above 3d array graphic
- totalNodes
    - Total number of nodes inserted into the 3d array. This is primarily used to help with node naming and using topsort

Functions

- read_data(path)
    - This is one of the first functions that should be run. This function takes in the path to data that needs to be read in. This function expects to find text files in the specified path in the format of r0.txt, r1.txt, ..., rN.txt. Where N is the maximum depth of the flows. The first character of the file name should be either r,d, or f, which represents right, down, or forward flows.
    - The main loop of this function loops through the text input and populated the 3d array of flow values.
    - After executing this function you will have a populated 3d array of flows that can be used for future operations.
- read_data_bin(path, modelname)
    - This function is similar to above; however, it reads in data from the binary files created by modflow.
    - This function expects to have a path to the binary files and the modelname which will be the naming of the binary files.
    - We then use bf.HeadFile(f"{directory}/{modelname}.hds") and headobj.get_times() to get a list of all time frames for the specified flow files.
    - We then use the time frame to get the flows for each time frame
        - cbb = bf.CellBudgetFile(f"{directory}/{modelname}.cbc")
        - frf = cbb.get_data(text='FLOW RIGHT FACE', totim=times[x])[0]
        - This code gets the right facing flows from the binary for the specified time and returns a 3d array of said flows. We then take this and integrate it into our 3d array.
        - The specified facing strings are ('FLOW RIGHT FACE' (right), 'FLOW FRONT FACE' (front), 'FLOW LOWER FACE' (down))
- print_graph()

    - This is a simple function that loops through the 3d array of flows and prints said flows.
- create_adjlist() and create_adjlist2()
    - These functions creates a list of edges using the 3d array and feeds it to the adjlist class to create the adjlist object.
    - The edges are created by taking the nodes name, the x,y,or z value, and the node that those values are pointing at.
        - The x value points to the node that is at gwgraph[z][y][x+1]
        - The x value points to the node that is at gwgraph[z][y+1][x]
        - The x value points to the node that is at gwgraph[z+1][y][x+1]
    - create_adjlist() will feed zero values into the adjacency list and create_adjlist2() will not. This was done to compare the effects of a zero flow between nodes. Hypothetically, if there is 0 flow between 2 nodes then their connection should not be added to the graph.
- print_adjlist()
    - This function simply runs the adjlists printing function to print the adjacency list and confirm that it was populated correctly.
- topsort()
    - This function runs ADJlists's topological_sort function to get the topological sorted nodes. This function populates the topsortList list with the topologically sorted names of the nodes.
- fraction_through(nodename)
    - This function calculates the fraction of water passing through each cell which is extracted at a user specified location
    - Using the topologically sorted list we backtrack through nodes in the graph and calculate the fraction through using flow and the already calculated neighbors.
- print_frac_through()
    - Loops through the graph and prints the calculated fraction through values.
- coords_from_name(name)
    - Given the name of a node, this function calculates the values to find the node in the 3d graph

ADJLIST

Variables

The main variables of this class are:

- adjList
    - An adjacency list of the edges received on initialization.

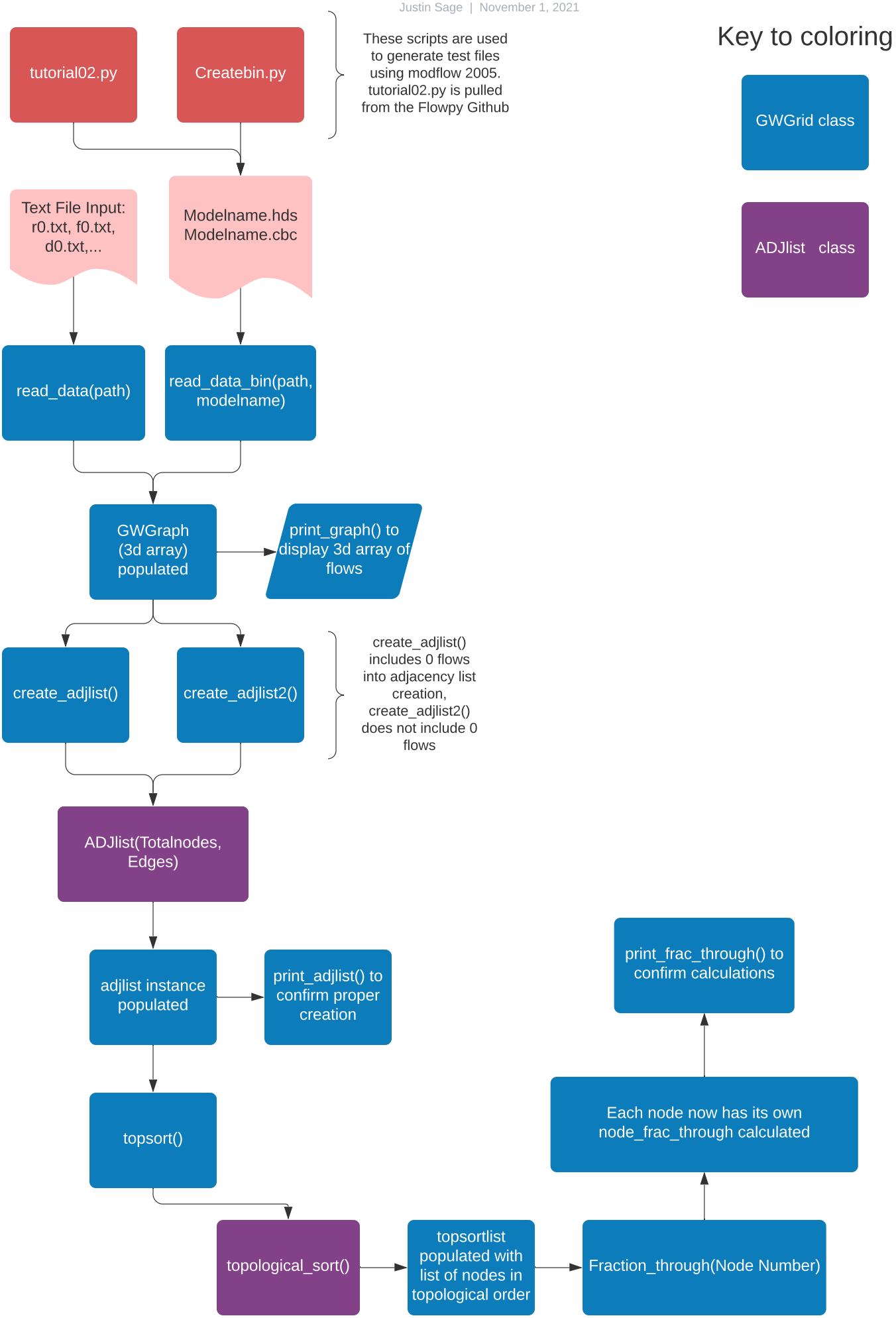
- totalNodes
    - Total number of nodes

Functions

- dfs(self, v, discovered, departure, time)
    - Performs dfs on the specified nodes
- topological_sort()
    - Returns a list of node names in topologically sorted order
- Dfs_target(revadglist, v, discovered))
    - Dfs designed to work with the following function to find all nodes reachable by the target node v
- upstream_target(self, target)
    - Reverses the adjacency list and looks upstream to find all nodes reachable by the target node. This returns a list of nodes to help with the fraction through calculations, so that we only look at nodes that reach our target.

Groundwater Flow

Justin Sage | November 1, 2021



TESTING

Testing can be done manually by creating the text files. These files need to follow a certain format as shown below.

R0.txt:

1 0 0 0

7 8 5 0

0 0 5 0

This is the right flows txt file. This is layer 0 from the file naming convention, each value in a row is space delimited to represent different columns. The last value always needs to be a zero for the right faced flows because that is the edge of the container. Similarly for down flows the bottom row needs to be all zeros for the edge of the container. Finally, for front flows the deepest level needs to be all zeros for the bottom of the container.

It is best to think of your flows in the context of a tub. You can't have right flows going through the wall of a tub and similar with down and front. The values are changed to 0 to represent the edge of the tub that your water is in.

Binary files can be created using the Createbin.py (modified version of Flopy tutorial 2 to create binary files of models, original tutorial and a modified version can be found in modflowTutorial folder). More information on Flopy can be seen in the documentation referenced below. This requires the mf2005 executable to run. The mf2005 executable in the source code is made to run on windows, the linux version can be found on the Modflow USGS website(see references below) . This will essentially create a mock flow using the variables in the file over different time frames. There is a bit of a learning curve with modflow so I recommend contacting Dr. Steffen Mehl for more information on creating binary files. You can also look at the modflow tutorial files included with this project to see WHere I derived the Createbin.py from.

Example from Test Files

In this section, I plan to document the flow of the program and how the different variables are populated or managed. For this example, I am going to use flows from the "TestData/txtfiles/test02" folder. I will show how this data is affected and modified by each function. I chose to use .txt files for this demonstration because it is easier to set up simple test flows this way. Normally data would be fed in through the binary files created by modflow. Example code is shown below.

```
from Helpers.GWGrid import GWGrid


def main():
    # initialize class for graph
    x = GWGrid()
    # reading in data files from specified folders
    x.read_data("TestData/txtfiles/test02")
    x.print_graph()
    x.create_adjlist2()
    x.print_adjlist()
    x.topsort()
    x.fraction_through(27)
    x.print_frac_through()
    return 0


if __name__ == "__main__":
    main()
```

x = GWGrid()

This is the first call. It essentially sets up an empty object representing the GWGrid that we will populate with the data.

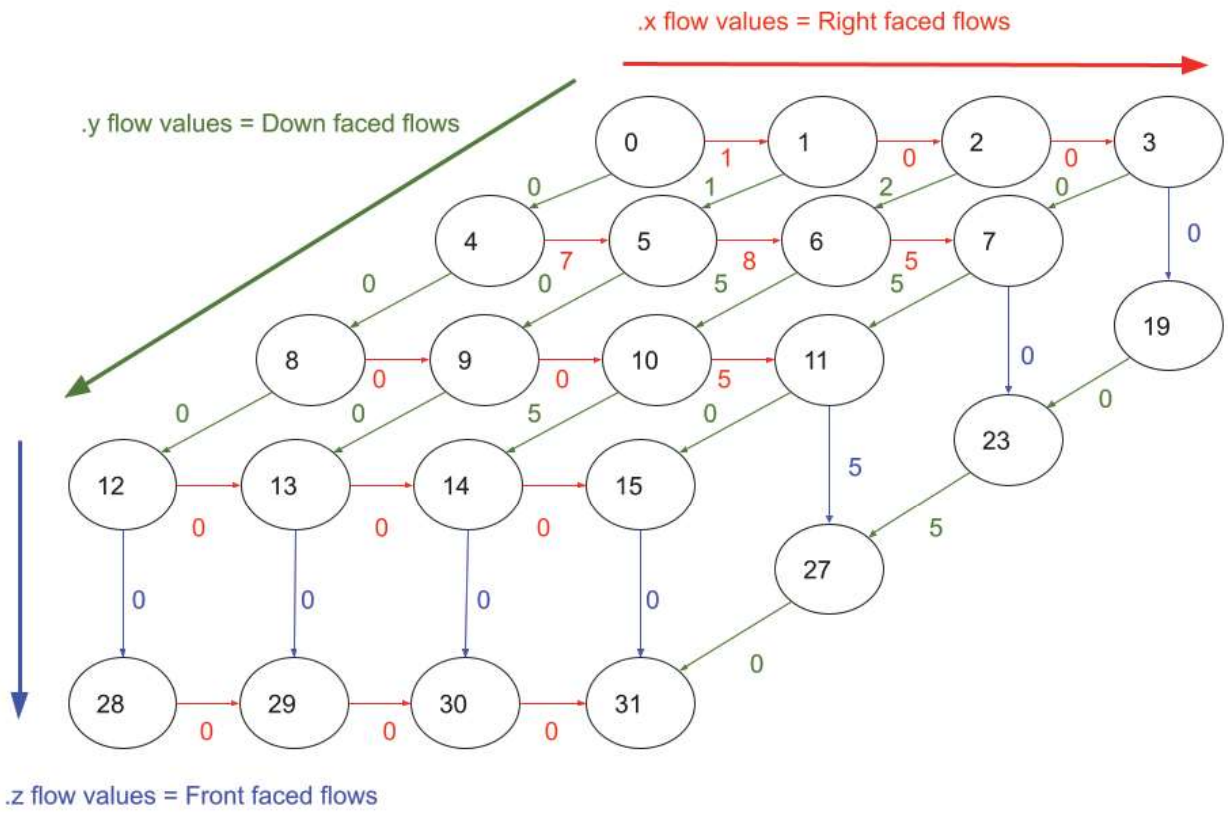
x.read_data("TestData/txtfiles/test02")

We then call the read data function to populate the 3d arrays values based on the data in the text files from the test02 directory. This function starts at the highest level, level 0, and proceeds to the deepest level, level 1.

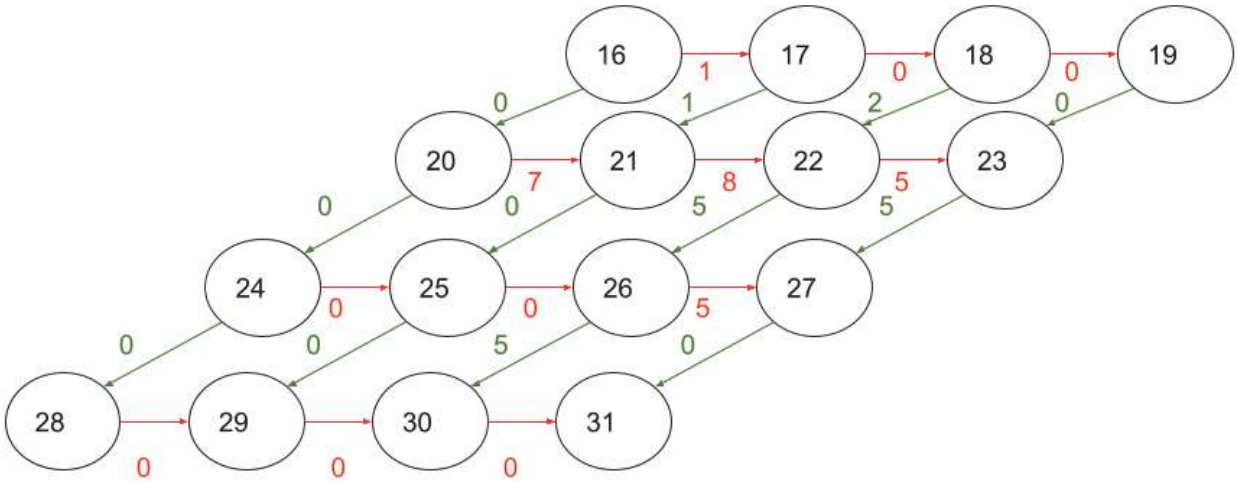
Files from TestData/txtfiles/test02: r0.txt, r1.txt, d0.txt, d1.txt, f0.txt, f1.txt

The flows in these files look like this in a node format:

0th level of flows



1st level of flows



While reading these flows, we populate a 3d list of nodes. Each node takes in 4 values:

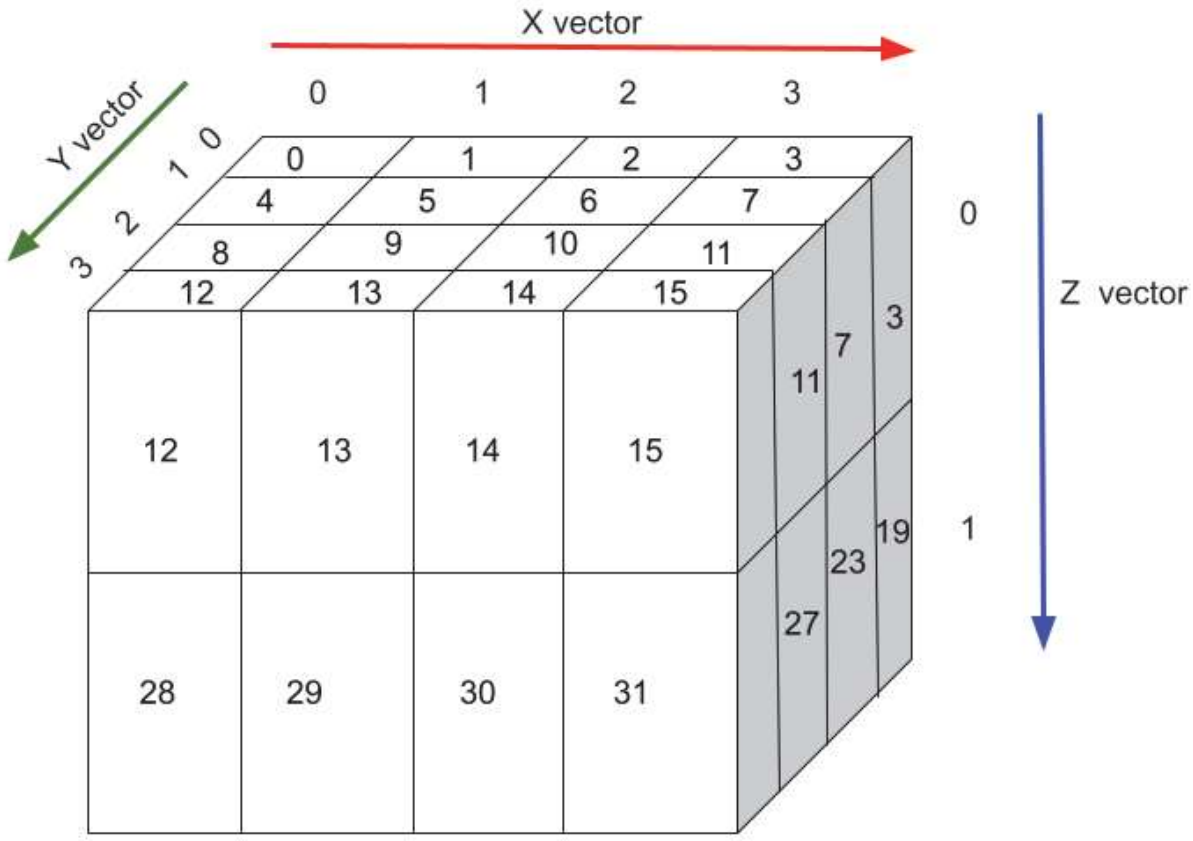
Name: the naming of the node starting at 0 and ending with the total number of nodes minus 1

X: flow to cell at right shown as [z][y][x+1]

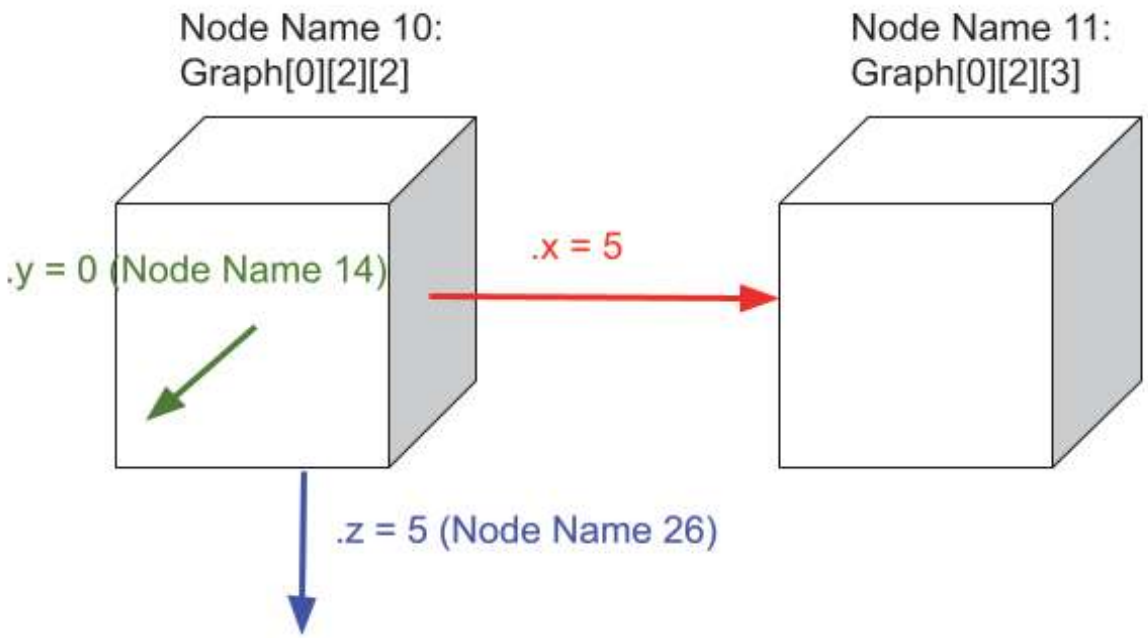
Y: flow to cell below [z][y+1][x]

Z: flow to cell in next layer [z+1][y][x]

The following graphic shows the 3d list with the naming of the nodes.



Lets use node 10 as a reference to see how the variables are stored



x.print_graph()

Now that we have the graph set up with the flows, we can print out the flows at each level to confirm that they were read in correctly.

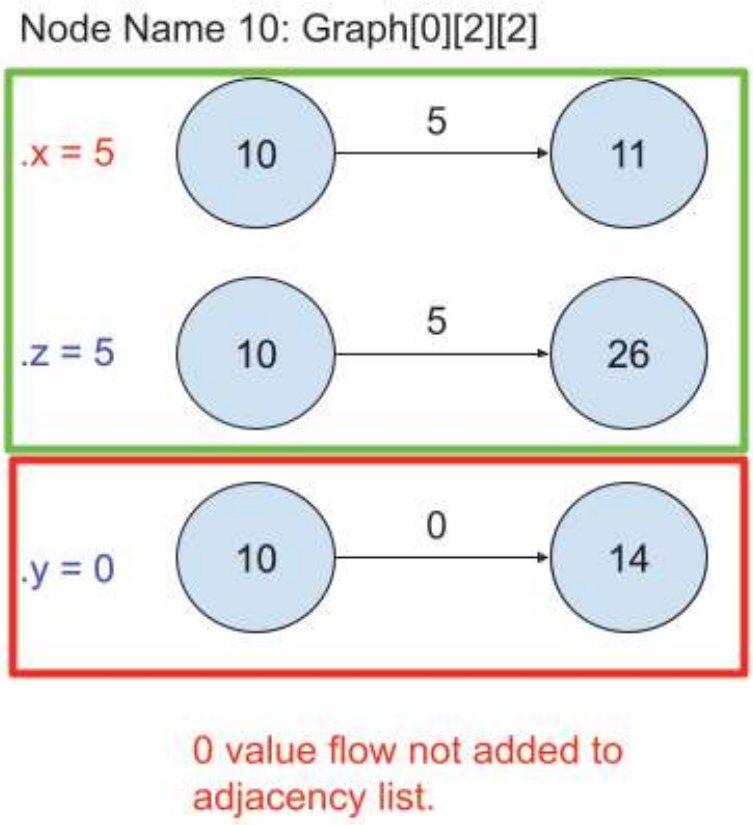
```
############Printing GWGraph#############
Total Nodes: 32
-----------Printing Level: 0------------
Right weights:
1 0 0 0
7 8 5 0
0 0 5 0
0 0 0 0
Down weights:
0 1 2 0
0 0 5 5
0 0 0 0
0 0 0 0
Forward weights:
0 0 0 0
0 0 0 0
0 0 5 5
0 0 0 0
-----------Printing Level: 1------------
```

```
Right weights:
1 0 0 0
7 8 5 0
0 0 5 0
0 0 0 0
Down weights:
0 1 2 0
0 0 5 5
0 0 5 0
0 0 0 0
Forward weights:
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
```

x.create_adjlist2()

Now we go about feeding our 3d array to the adjlist class as a list of node edges. This is where the naming of the edges comes into play. The adjlist class creates an adjacency list of the nodes based on the integer values used as names. The adj list is indexed by the name of the source node. Each x,y, and z weight is converted into an edge (name of source node, destination node, flow value). Edges with a 0 flow value are not added to the adjacency list using this function, this is done because a 0 flow technically represents no interaction between 2 nodes. If you wish to include 0 flows in topsort, You can use the x.create_adjlist() function.

Example edges using node 10 (GWGraph[0][2][2]):



x.print_adjlist()

Now that we have created the adjacency list. We print it in order to confirm that it was created successfully. We can see that the edges shown in the previous graphic are correctly added to the adjacency list. These edges are shown as (10 --> 11, 5) (10 --> 26, 5).

```
############Printing ADJlist#############
(0 --> 1, 1)
(1 --> 5, 1)
(2 --> 6, 2)
(4 --> 5, 7)
(5 --> 6, 8)
(6 --> 7, 5) (6 --> 10, 5)
(7 --> 11, 5)
(10 --> 11, 5) (10 --> 26, 5)
(11 --> 27, 5)
(16 --> 17, 1)
(17 --> 21, 1)
(18 --> 22, 2)
(20 --> 21, 7)
(21 --> 22, 8)
```

```
(22 --> 23, 5) (22 --> 26, 5)
(23 --> 27, 5)
(26 --> 27, 5) (26 --> 30, 5)
```

x.topsort()

Now that we have the adjacency list created, We are able to run topsort. After this function is run, the GWGrid class will have a list of topologically sorted node names. We will use this list in future calculations. We print the list of node names to standard out. Every node on the right depends on flow from the nodes to the left.

```
#######Printing Topological Sort########
31 29 28 25 24 20 19 18 16 17 21 22 23 15 14 13 12 9 8 4 3 2 0 1 5 6 10 26
30 7 11 27
```

Before storing this list in GWGrid we reverse the ordering. This is done because calculations, such as Fraction Through, require a node to be calculated before its dependencies can be calculated. This will be further explained in the next section.

x.fraction_through(27)

With Top Sorting complete, we can now perform one of the main calculations for this project, Fraction Through. To do this, we specify a downstream node. In this case, we are specifying node 27. We then set that nodes fraction through as 1, since the targeted node can only have 100% flow coming though itself. We then use the topsort to move upstream of this node and calculate what fraction of water that is leaving this node is making it to the specified node. A nodes neighbors should be calculated before itself, hence why we use topsort to guide calculations. We use our neighbors fraction through to calculate our own fraction through.

coords_from_name(self, name)

We utilize a helper function for fraction through calculations called coords_from_name. This function takes in the name of the node and returns the z,y,x values([z][y][x]) to access that node in the 3d list. This is done to get the nodes from node names returned by topsort The calculations to find the coordinates look like this:

```
z = int(name / nodes_per_level)
y = int((name - (nodes_per_level * z)) / self.length)
x = int(name - ((nodes_per_level * z) + (self.length * y)))
```

Lets do this calculation with node 27:

Length = 4

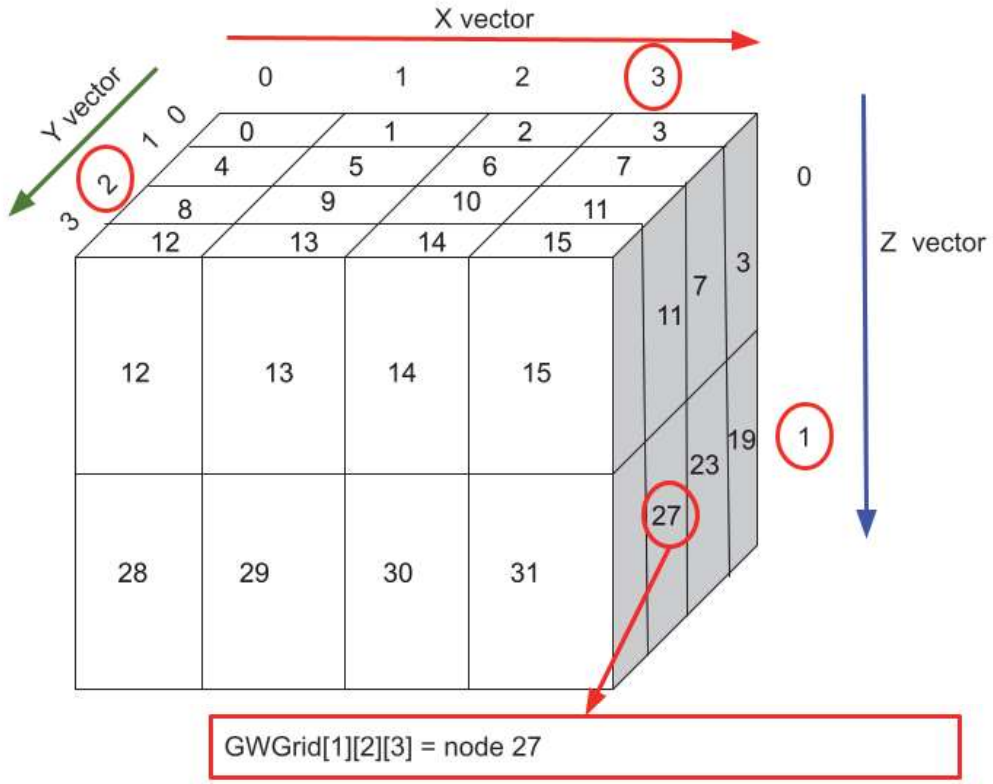
Nodes_per_level = 16 (4 width x 4 length)

Z = 1 (27/16)

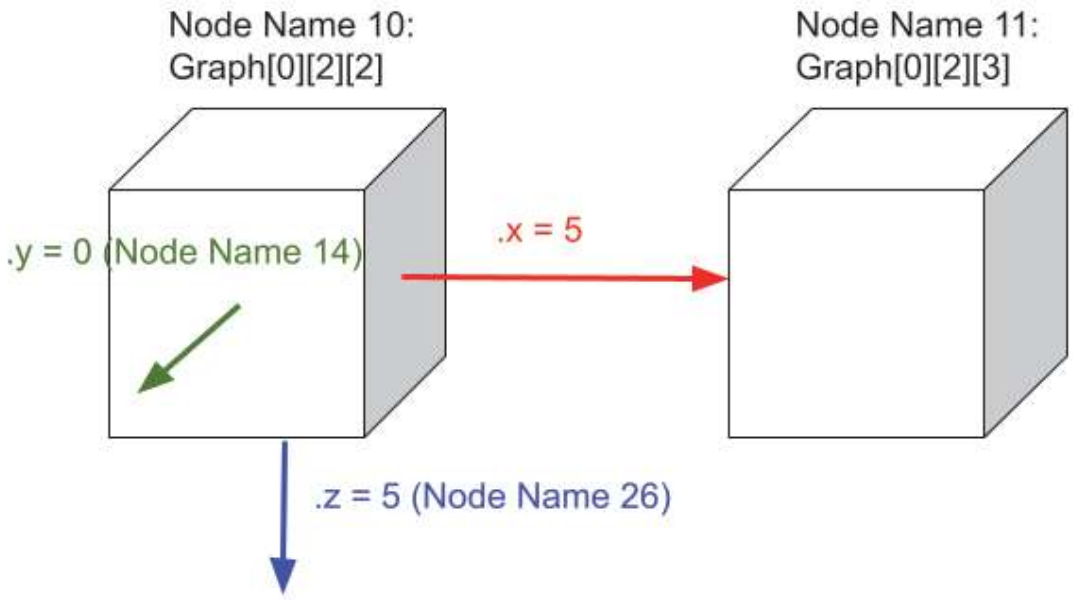
Y = 2 [(27-16*1)/ 4]

X = 3 [27-((16*1) + (4*2))]

Now we know that node 27 is at GWGrid[1][2][3]



Now that we have discussed how we get to a node from its name in the tops sorted list. Lets look at how the actual fraction through calculations are completed. First we set the specified starting node as 1 to represent 100% flow going through itself. We then start going through the adjacency list starting at the specified node to perform the calculations. Lets take a look at node 10 as an example. The neighbors values should be calculated before node 10 thanks to following topological order and as you will see node 14's fraction through is equal to 0. This means that this node does not have flow that makes it to the specified node 27.



GWGrid[0][2][2+1](node 11)	.node_fraction_through = 1
GWGrid[0][2+1][2](node 14)	.node_fraction_through = 0
GWGrid[0+1][2][2](node 26)	.node_fraction_through = 0.5

With the values shown above, we can complete the calculations needed to find fraction through for the node 10 before moving to more upstream nodes in the top sort list. The calculations is completed as follows:

```
x_downstream = self.GWGraph[z][y][x].x
x_downstream_mod = float(x_downstream) * self.GWGraph[z][y][x+1].node_frac_through

y_downstream = self.GWGraph[z][y][x].y
y_downstream_mod = float(y_downstream) * self.GWGraph[z][y+1][x].node_frac_through

z_downstream = self.GWGraph[z][y][x].z
z_downstream_mod = float(z_downstream) * self.GWGraph[z+1][y][x].node_frac_through
```

$$\text{GWGrid[z][y][x].node\_fraction\_through} = \frac{\text{X\_downstream\_mod} + \text{y\_downstream\_mod} + \text{z\_downstream\_mod}}{\text{X\_downstream} + \text{y\_downstream} + \text{z\_downstream}}$$

Lets put in the calculations for node 10:

GWGrid[0][2][2].node_fraction_through = $((5 * 1) + (0 * 0) + (5 * 0.5))/(5 + 0 + 5)$

GWGrid[0][2][2].node_fraction_through = $7.5/10$

GWGrid[0][2][2].node_fraction_through = 0.75

x.print_frac_through()

Now that we have completed the calculations for fraction through, we can print out the values to confirm our calculations. Below we can see that the output for node 10 shows as 0.75, confirming our calculations(node 10 is in level 0, 3 rows down, the third number to the right)

```
 #######Printing fraction through########
Total Nodes: 32
-----------Printing Level: 0------------
0.875 0.875 0.875 0
0.875 0.875 0.875 1.0
0 0 0.75 1.0
0 0 0 0
-----------Printing Level: 1------------
0.75 0.75 0.75 0
0.75 0.75 0.75 1.0
0 0 0.5 1.0
0 0 0 0
```

REFERENCES

Foley, Christopher & Black, Alastair. (2013). Efficiently delineating volumetric capture areas and flow pathways using directed acyclic graphs and MODFLOW; description of the algorithms within FlowSource.

“Beginner's Guide to MODFLOW.” *Online Guide to MODFLOW-NWT*, https://water.usgs.gov/ogw/modflow-nwt/MODFLOW-NWT-Guide/index.html?beginners_guide_to_modflow.htm.

“What Is Flopy¶.” What Is FloPy - Flopy Documentation 3.3.2 Documentation, https://flopy.readthedocs.io/en/3.3.2/introduction.html.