Swarm Code Set-Up – June 12, 2013

Necessary Packages and Software:

* SAP2000 – activated
* Python 3.2.x (<http://www.python.org/download/>)
* pyWin32 module for Python (<http://sourceforge.net/projects/pywin32/files/pywin32/>)
  + Make sure you obtain the version for whichever python you’re running
* VPython for visualization (<http://www.vpython.org/contents/download_windows.html>)
* !important -> Make sure to install the appropriate versions of pyWin32 and Python 3.3.x (this means that even if you’re on a 64-bit machine, if you have installed 32-bit python, then you MUST install pyWin32 32-bit. Most of the time this error will be caught by the pyWin32 installer (it will tell you that you have no version of Python installed). Instead of attempting to re-install python (as two versions of python can be a pain to handle), make sure you’re using the right pyWin32 installed.
* Install the tablib library. Again, this can more easily be accomplished if the code is simple pulled from the github reposity as the tablib library is already incorporated into it.
* Instead of tablib library, install xlsxwriter (<https://xlsxwriter.readthedocs.org/en/latest/>)
  + Make sure you install the xlsxwriter for the right python version

Quick Notes:

* In order to gain access to the SAP2000 COM, the name is “sap2000v15” instead of “sap2000” (as most online tutorials have written). This is probably because most of the online tutorials tend to be older and therefore aren’t direct to this version of sap2000 (presumably, v15).
* Along this same line, make sure to modify the pySAP2000 library provided (unless you’ve downloaded the latest version from the correct repository). This modification is necessary for this version of SAP2000. Instead of using the EnsureDispatch method which attempts to make the program compatible with makepy, use the Dispatch functions directly. In other words, replace gencache.EnsureDispatch with Dispatch in swarm.sap2000.sap2000.start.

Steps Taken to Set Up:

* Git clone the pySAP2000 repository into a new folder locally. The repository can be found on bitbucket (<https://bitbucket.org/pmav99/pysap2000/src/81269a3e0a093c104066005e3d23892e910c8d10?at=default>). Code seems to provide basic Python access to most of the functions available through the SAP2000’s OAPI.
* Received CLSIDtoClass KeyError and spend hours deciphering it.
  + Found makepy.py in C:\Python33\Lib\site-packages\win32com\client
  + Ran python makepy.py –i to obtain code for the generation of .py support manually. Attempting debug found in (<http://osdir.com/ml/python.windows/2005-04/msg00101.html>).
    - Results: gencache.EnsureModule("DD021D4E-29D9-467F-8CAB-9038B20FB0B4",0,3,3)
    - Did not change the error message. Was simply doing the same thing in a longer fashion. Still struggling.
  + Solution to Problem:
    - Instead of starting the program with win32comm.client.gencache.EnsureDispatch(“SAP2000v15.sapobject), the program needs to be started with win32comm.client.gencache.Dispatch(“SAP2000v15.sapobject”). I’m not 100% sure why this is the case. Regardless, the current code has been fixed. This can be done if you look inside the file sap2000.py and change the appropriate line inside of the class SAP2000().
      * Aside: Problem might have to do with connecting through VPN. I am not entirely sure if this is the case of if something else was causing it. Might also be because this is a different version of SAP2000 for which makepy might not be supported. Again, this is a conjecture. There was not enough time to solve the issue, though if you backtrack all of the errors, the seem to occur during the creation of the cache used by makepy, which leads me to the assumption that makepy is incompatible with SAP2000v15. Furthermore, the library tends to have small errors throughout it. Make sure to change the code as specified below. Or, otherwise, download the latest version from the github repository linked below.
* There is also a private repository with all of the code set up on: http://github.com/kandluis/sap2000
* Documentation of the functions themselves can be found inside of sap2000.py (the comments are exhaustive and most of the basic functions are self-explanatory). The code is mostly the same as that found with pySAP2000 but there are multiple modifications that were made. The way a system saves a file, for example. Additionally, there is a new initializeModel method that allows the initialization of model. For more information, take a look at the comments in sap2000.py, inout.py, helpers.py, and \*.py (all python source files).

Swarm Code Testing – June 13, 2013

Proposed Directory Structure:

Swarm\

Sap2000\

Sap2000\

Constants.py

Readme.txt

Sap2000.py

Sap\_analysis.py

Sap\_areas.py

Sap\_base.py

Sap\_groups.py

Sap\_points.py

Sap\_properties.py

Commandline.py

Helpers.py

Inout.py

Readme.md

Python module: swarm.helpers

Created helpers.py with a helpful functions that checks whether a directory exists, and if not, creates the directory. Was running into the issue where the definition of the output file might throw an exception if the folder that contained it did not exist. This resolves that issue and makes the commandline program much simpler to use. There exists a simpler way to do this, but they are only available for Python3.0+. Since the library is so far compatible Python 2.7+, I decided to maintain the compatibility by writing my own function.

Python module: swarm.inout

Created inout.py which provides the input/output function for the program. Starts SAP2000, creates a model if none is specified, and saves it at the specified locations.

Python module:swarm.commandline

Provides simple command-line control of the SAP2000 program. Basically allow for specifying input file (on which an analysis is run) and output file.

Further Library Documentation:

Methods and Properties of SapBase:

* \_sap provides access to the sap\_com\_objects (often referred to as SapModel in online tutorials)
* \_obj provides access to the specific object, such as:
  + .SapModel.PropArea for SapAreaProperties
  + .SapModel.Analyze for SapAnalysis
  + Some classes have objects associated with them, others are specified at instantiation of class
* get \_names to return the names of all defined elements/objects, depending on class
* count to count all of the defined elements/objects, depending on class

SapPropertiesBase is currently the same as SapBase (no changes)

SapPointsBase inherits SapBase and adds the following methods:

* get\_cartesian(name,csys = “Global”) to obtain the coordinates of the point object of this class in the Present Units. The coordinate system is specified by Csys.
* get\_all to return a dictionary with the coordinates of all point

SapPointObjects inherits SapPointsBase and simply names \_obj as SapModel.PointObj

SapPointElements inherits SapPointsBase and simply names \_obj as SapModel.PointElm

SapAreaProperties inherits SapBase and simply sets \_obj as SapModel.PropArea

SapAreasBase inherits SapBase and adds the following methods:

* get\_property(name) to return ((( NEEDS EDITING)
* get\_points (name) to return the points that define an area element/object

SapAreaElements inherits SapAreasBase, naming \_obj as SapModel.AreaElm. Also adds following methods:

* get\_area\_objects(area\_element\_name) to return the name of the area object from which the area element was created

SapAreaObjects inherits SapAreaBase and simply sets \_obj to SapModel.AreaObj

SapGroups inherits SapBase, setting \_obj to SapModel.GroupDef and adding the following methods:

* get\_one(group\_name) to

sap\_base.SapAnalysis inherits from SapBase. Though this seems unnecessary and might be taken out. There’s another in sap\_analysis which does not inherit.

Swarm Code Interfacing and Familiarization – June 14, 2013

This section covers familiarization concepts. To begin, currently running the commandline script provides direct access to the API. This access allows all of the library functions detailing for the SAP 2000 OAPI to be used.

Notes of Interest:

* When the program returns an error about SAFEARRAYS, this means that, all though the type is Array, you should use either tuples or lists (ie, arrays with a predefined and unchanging state). This is what is meant by SAFEARRAYS. In VBA, this is often written as Offset(0) or some ByRef NAME(). As always, preferred that tuples are used. Still experimenting with program.

Changes:

* Adding spherical and cylindrical coordinates to the python definitions. Should make creating the beam element for construction much simpler.
* Still attempting to decipher program at what types of materials to use for the construction (simulation). Links appear unfeasible, so the original idea of creating point objects and connecting them with links is probably not good. Line Elements exists onto which certain properties could be added, but I have yet to test this out.
* Was Able to add Solid Objects, though, which appears to be good enough.
* Began creating the basic skeleton for the robots, even if I’m still not sure how we will manage. The idea will be to define some type of “beam” which each robot carries (each robot has a certain amount which will be defined in variables.py). The beam will more likely result from a point object and a link (or line element, if that ends up working correctly). Still don’t have enough information to resolve.
* Will probably settle on creating the beams out of frames. This sounds like the only logical solution to the problem. There will be an inherent limit to the length of the objects, which I assume, cannot be changed. We’ll see how it all works out.
* There are two new additional files. sap\_points.py, which adds the ability to access Point Elements and Objects, in addition to sap\_lines.py, which allows access to Line Elements. Furthermore, the file sap\_frames.py is under construction.

**NO IDEA WHAT HAPPENED TO MY NOTES ON JUNE 17th**

Swarm Code Interfacing and Familiarization – June 18, 2013

Creating the structure to store the model in python.

This is the initial plan. The structure will divide the model into a 3-dimensional array (in a sense, we are dividing space into small rectangular boxes) so that checking for nearby beams can be done more quickly. This division will mostly take palce inside of structure.py. The main part of the structure will be a 3d array holds a dictionary indexed by the name of the beams. Each beam will contain the coordinates of its endpoints. This seems like the most straightforward approach.

Considerations:

* Boxes will only lie in the positive x,y, and z axis. That seems to be the simplest solution to this. This means that robots will have to construct in the first octant and nowhere else. This should be fairly interesting.
* Have found a function called coordinaterange which might do just what we want. It selects all elements within a range of a certain point. We won’t need to keep track of the entire structure any longer.
* Decided against using the simpler function and will instead implement the entire functionality from scratch. Will break up the coordinate system into 1000 boxes, each of 5 units length. Under the default settings, this should give a 5m by 5m by 5m box on which the robots can construct.

Information on the 3D Array:

* All of the code for what I’ve termed the structure can be found inside of structure.py. This file contains the class definition for the structure.
* The structure is designed so that the structure is built starting from the origin, and only along the first octant. This was made to make the division of the structures easier.

Error Corrections:

* Added multiple functions inside of helpers.py so as to allow for error correction in possible inaccuracies that might arise because of floating point arithmetic errors. These errors are corrected through the following procedure.
  + Every time the get\_directions function is called, it looks at the location of the robot. In addition to a location, the robot also keeps track of the beam it is currently crawling on. With this information, the program checks to make sure that the robots location agrees (within a specified epsilon) with the points spanned by the beam. If the robot is not on the beam, then the location of the robot is reset by repositioning it onto the beam (this is done by simple geometric projection which is take care of by the function in helpers.py named correct.
  + Furthermore, by updating the location, hopefully small errors won’t add up over multiple iterations and movements. This should allow the robots to continue working. It had to be done because we don’t want to continually query the SAP2000 program about the specific location of beams and whether or not a load can be assigned to specific points.
* More functions were added to Robots.py. One particular function added is a private function which changes the location of a robot. This function (change location, actually moves the load of the robot on the SAP2000 program in addition to changing the self.location instance variable)

Afternoon:

Most if not all of the syntax errors are now solved. Here is the general structure of the program.

* The main file is called main.py. This file imports all of the code from the commandline.py
  + The commanline.py code has been updated as follows:
    - If the code is run from the commandline (ie, \_\_name\_\_ == “\_\_main\_\_”), then a separate branch of code is executed. This code executes as follows.
      * This code remains basically the same as before. Commandline arugments are parsed for information on the output and inputfile, and then the program is opened. The opening, though, occurs through a definition of a new function called run(). This was done so that the file could be imported and used in main.py. Might factor even further later on.
    - If the code is not run from the command-line (ie, imported as is the case in main.py), then nothing occurs automatically. Instead, access to the run function is provided.
* Main.py then goes on to construct a specific output filename for the this specific run. A folder is created inside of C:\SAP 2000\ based on the month and day (MON-DA), and then a file inside that folder is created with the following structure – “HOUR-MINUTE-SECOND.sdb”
* As the next step, we use the run function to initialize the SAP2000 program. Hopefully (though I have not verified), this will allow all of the following classes to have access to the program. If not, the solution is as follows:
  + When an automaton is created, the model (which provides access to all of the SAP2000 functions) will be passed in. This will allow each robot to have access to the program,though it must be remembered, that the program is mutable (there exists only one, and a change caused by one robot changes the program for all of the robots)
* Continuing, we import Worker from robots. This gives us access to our Worker bots so they can climb in the structure. (In the future, we will import Swarm from swarm.py, since the swarm will keep track of hundreds of robots (an arbitrary number, really)).

Robots.py Notes

* Need to really focus on \_\_change\_location function, as this is the meat of the program. Each robot will have a location on the structure. Figure out how to unload, load the robot. Fast.

Sidenote: Unused code removed for simplicity’s sake

'''

## Some TEST code we decided not to use

range = variable.local\_radius/2

x,y,z = self.\_\_location

selectObject = model.SelectObject

# calculate box coordinates

xmin,xmax = x - range, x + range

ymin,ymax = y - range, y + range

zmin,zmax = z - range, y + range

# select box using program

selectObject.CoordinateRange(xmin,xmax,ymin,ymax,zmin,zmax,IncludeIntersections=True)

return\_value, number, types, names = selectObject.GetSelected()

helpers.check(return\_value, "An error occurred when obtaining the direction. GetSelected returned non-zero")

'''

Swarm Code Interfacing and Familiarization – June 19, 2013

Ideas for creating the robots in the SAP Program:

* Either, create new point objects which are the robots. This will allow mass to be added to different locations along the beams (ie, not just onto the beam itself, but at specific points instead).
* Simply add the mass the frame object that the robot is standing on. This is the simplest way to create it.
* This means that we will be creating a loadcase for the robots walking on the structure, and then a load case called DEAD for simply the structure standing on its own. The robot loadcase is named “ROBOTS” as specified in sap2000.variables

Swarm Code Interfacing and Familiarization – June 20, 2013

* Possible errors with the movement that still need to be fixed:
  + Robots are not capable of switching beams. They can only switch from one beam to another if they happen to land precisely at the intersection point of the beams. This is can probably be fixed by calculating all the intersection points of the robots, and moving the right length around each. This should be interesting to try and program.
  + The code for wandering around when not on the structure is still not implemented. The function is called “wander()”, but I don’t yet have a clear idea as to how to make the robots wonder. Here are a couple of proposed solutions:
    - The robot wanders around randomly. This is difficult because get\_directions\_info is programmed assuming that the robot is on the structure (the directions it returns only have to do with the directions allowed by the beam.)
      * Possible solutions: If get\_directions\_info returns empty, then chose a random direction that still maintains the robot within the positive octant.
    - The robot wanders towards the nearest beam. This doesn’t take into account the possibility that there might not be a nearest beam anywhere around. This is very interesting.
  + Currently, the robots only build at the endpoints and only transfer to other beams at the endpoints. I honestly have no idea how to get line intersection in 3D going here. Will try later
* Solutions:
  + The robots will only switch beams at endpoints. The problem can be reduces by making the time-step as small as possible and by adding some sort of radius around each point. This might be taken care of by epsilon in the compare function. Hopefully these errors don’t add up and spiral out of control.
  + The robot wanders around randomly, except when there is a beam within one timestep of it’s location. When there is, (and this beam must have a zero z-component), the robot has a 50% chance of moving onto the beam and off

Swarm June 21, 2013

Major revision to the codebase in order to incorporate the ability for beams to contain the joints. This is how it goes down, as of now.

* New file: beams.py
  + This file contains the class Beam. A beam is basically what we had before, except that it also contains additional information. Beams store information pertaining to the locations where other beams have a junction. This information is contained inside of the self.joints. Additionally, the class is broken down as follows. This class replaces the dictionary we previously used to store information on beams (ie, beam[name] = name, beam[endpoints] = endpoints) with the following structure.
    - Beam.name = name
    - Beam.endpoints = (i=i-end coordinates, j=j-end coordinates)
      * We are using named tuples in case we eventually need to order these coordinates in some way or another.
    - Beam.joints which is a dictionary indexed by the coordinates of the joints. Each coordinate contains a list of the names of the respective beams that intersect the current beam at that point

Swarm June 24, 2013

Beginning work on construction. Seems like a lot of functions from movable are going to be overwritten in order to add the tendency to move upwards. Unless I just edit the code directly in movable, which is always a nice possibility. Might factor it out into more classes. (MovableBase) and (MovableUp)

* MovableBase will contain all of the helper functions which I think are ready for completion. MovableUp will
* Have decided against it.
* New changes:
  + Moving in a random direction now means that the robot moves in a straight line until it hits the ends of the box (then it changes direction to another random direction), unless it finds a beam somewhere near it.

Useful Functions Found in API (Can only be used if we “Start toe steel frame design”):

* SapObject.SapModel.DesignSteel.GetSummary Results
  + Controlling stress or capacity ratio type:
    - PPM
    - Major Shear
    - Minor Shear
    - Major beam-column capacity ratio
    - Minor beam-column capacity raio
    - Other
* SapObject.SapModel.DesignSteel.VerifySections
  + Returns the names of frame objects that have different analysis and design sections

Swarm June 25, 2013

Worked mostly on deciding how to get the construction down. The robot will wiggle around the beam, starting from a vertical direction, and position it as soon as it hits another beam. This is the basic idea of construction. Since the robot can only build upwards, it means that it has to climb back down if it needs to reinforce and area somewhere.

Back to the building. The steps taken are as follows.

* Robot decides whether to build at its current position or not
  + This function is not yet implemented. This will more than take in the inputs from the analysis model, along with a short-term memory, and if everything fits correctly, tell the robot to construct as opposed to moving again.
  + Additionally, we will need to keep track of the location of the robot for a while (or give it some sense other than “upwards” and “downwards”
    - Speaking of, the robot now has an “upwards” and “downwards” variable which means that it not only moves upwards when it has a beam and downwards when it doesn’t, but it also can move in either direction depending on how the variable is set.
* The robot then wiggles a beam vertically up until it matches another beam (or if it doesn’t find a beam within a suitable angle, it just sets the beam down as specified in construction.beam. Here is a more detailed explanation of the process:
  + We obtain all of the beams in the local box, and check to see if the beam crosses into the next box. If it does, then we obtain the beams in that box
  + We run through the beams and we find the ones that might intersect with our current beam. We calculate that intersection point.
  + We find the closes point of each beam to the vertical beam
  + From this data, we calculate the angle at which the beam would be if placed there.
  + The robot always places the beam with the smallest ration (ie, largest angle)

Swarm June 26, 2013

* Finishing the intersection of the sphere with the beams in a box. Turns out to be more complicated than expected. The goal, therefore, is as follows:
  + Calculating the intersection of the beam with the sphere will be done with vectors. There’s a formula online for it. This formula is for calculating the intersection between a line and a sphere, which means we will have to do some additional checks once we do find the intersection:
    - Check to make sure that the point of intersection is on the beam.
    - Should probably generalize this function as we will more than likely use it later (for finding the closest point)
  + Furthermore, we also need to calculate the closest point on all other beams to the vertical beam. This will let us take into account the fact that you can have a beam that leans down onto another. Process to find these two closes points in documented inside the helpers.py file, but here’s a short overview:
    - Using the two endpoints, we construct a line in vector form (initial point + direction\_vector \* constant). We do this for both lines.
    - Take the cross product of the two lines in order to find a plane on which both of the lines *could* lie
    - This plane is initially un-positioned, so we position it so that it contains at least one of the aforementioned lines. (In our code, this is the first line passed in)
    - We project the other line onto this place. There are a couple of possibilities when doing this:
      * The line is already on the plane, which means the two lines are coplanar. If this is the case, then we know that the closest distance is either zero, or that the distance is constant throughout both lines (intersection and parallel).
    - We find the intersection of these two lines (if it exists)
    - Using this intersection point and our normal, we create another line (intersection point + normal) and we find the intersection of this line with our original l2 (the second line which was projected onto the place.
    - The two points (intersection point on the plane, and intersection point on the second line) are the two closest points!
    - Return them
    - In the cases where the lines are parallel, we return None. The hope is that the line will be caught when calculating the intersection with the sphere.
    - In the cases where the lines do intersect, depending on how the Boolean “segment” is set, either the two points are returned or the two closest endpoints are returned. Or something like that. Hopefully this isn’t the case often.

Swarm June 27, 2013

Intersection of the lines with the sphere has proved more difficult than previously expected. Still working on it.

Adding the properties of the material. Moved around a lot of the code inside of main.py so that we now have a way to run the simulation multiple times without relaunching the SAP 2000 Program. A few issues that have been fixed:

* Added a decide() function to the colony (which implies that a “decide()” function was also added to the robots)
  + Reasons:
    - SAP 2000 Program locks the model after it has been analyzed. This means that the robots can’t both gather the information needed (from the Analysis model) *and* act on that decision. Therefore, the functions that decide whether to construct or not, on in which direction to move, have been rewritten to run first (when the colony’s decide() method is called) and store that information within the robot.
    - The analysis model is the unlocked (which deletes the analysis results for this time-step, so there is *no* way to to access them again).
    - The colony’s act() method is called, which calls each of the robot’s do\_action() method.
    - do\_action() looks at the information stored by decide() and performs whatever it tells it to. This includes moving (adding a new load and removing a load) which can *only* be done when the model is unlocked.
    - Rinse and repeat

Swarm June 28, 2013

* Finished the code for the simulation and spent most of time debugging everything. Added more functionality to main:
  + Created the class Simulation
  + Allowed settings to be reset without necessarily restarting the SAP 2000 program. This means that everything can be tested multiple times. Adding a few other abilities to the running function:
    - Run now writes out:
      * Locations.txt which contains the locations of the robots
      * Sap\_failures.txt which contains any failures that occur with the SAP 2000 Program.
      * Opens these files safely and closes them out even when the program crashes. This allows us to view a lot of information on the current state of the simulation and therefore debug more effectively when something does go wrong.
  + Added reset functions to structure, colony, and sap2000. In order, this is what they do.
    - Structure.reset() simply starts up a new self.model – basically resets all of the boxes to be empty. Additionally, this also resets the number of tubes that the structure has 0.
    - Colony.reset() creates a new swarm of robots. This means that the old robots are left on the model exactly where they were before they were reset. But a new swarm is created at the home location. This works in conjunction with the next reset function.
    - Sap2000.reset() resets the SAP Program. In order to do this, it basically saves the current model and creates a new one, saved to a different folder (depending on the time at which the reset() was made).

Swarm July 1 2013

Debugging finally finished except for a few errors:

* The robot continues to try and place a beam in the same location as before. This is unacceptable as it causes the SAP Program to return a non-zero value (and therefore the simulation fails). In order to fix this, I added the following function to structure:
  + Exists(e1,e2):
    - This checks to make sure that no beam currently exists between the points specified by e1 and e2. This way, the robot won’t construct somewhere where a beam already exisits.
  + Modifications to the build function:
    - The build function has been modified so that it now checks whether or not a beam exists before it tries to build it. If there are ratios to be had, then it moves to the next eligible ratio – since we have already constructed one at the current level.
    - If there are no ratios, the it places a beam randomly (still with a tendency to point upwards, but not necessary at the same level as before)
  + The robot also seems to be stuck at one joint. It seems to be incapable of climbing onto the beams, so…

Swarm July 2 2013

* Improvement on the exists function. Additionally to checking whether or not the two points contain a beam, it now also checks to see whether or not the beam that will be placed is going to overlap with an already existing beam. The function has been renamed structure.available(), considering its additional functionality.
* Added more abilities to the main function, in addition to adding a current\_state function for the robots (will probably add one to the beams if found necessary)
  + Used this to write out the data onto the corresponding folder for each run. This will allow us to analyze the results of the run more closely.

Swarm July 3 2013

* Refactoring code in worker and moving it into builder.py. This will allow me to keep all of the code for the decision making of construction to stay together inside of worker.py. It turns out that builder.py has become too lengthy.
* Added function:
  + Climb\_off which returns whether or not a robot should climb off the structure (this will allow us to change this functionality in future classes)
  + Fixed mistake in \_\_path function. We used to move the closest cube face (measure in distance), but in fact it needed to be measure in time\_distance (distance/velocity). This has been fixed and the program no longer crashes.
* Implementing testing suite:
  + Makes testing simpler. Code can be found inside of run\_test.py. File has been pushed to git.
* Current\_state() added to beam. This allows for more in depth debugging.
* Planning on adding export functionality. Allow for a report to be created at the end of the run. This should make things easier.
  + Also planning on exporting my python structure and robot\_data to excel files. This will probably make more readable -> will use the tablib library
  + Finally set up the excel exporting of location data
    - Trimming the values (rounding to two decimal places)
    - It still seems to have some sort of bug, but moving on from it

Swarm July 5 2013

* Colony.py:
  + Added the ability to add more robots to the swarm while the simulation is occurring. This should make everything a little more interesting and provides something else with which can tweak
  + Added the ability to remove robots from the swarm. I
    - If the robot is on the structure, its load is removed before the robot’s reference is deleted from the swarm. This makes it so that we can model the breakdown of certain robots every now and again.
  + Added the ability to delete robots by name.
    - Pass in a list of names and it deletes each of those robot’s references. Not particularly sure why this would be useful, except for possibility providing a lifetime for robots (or something along those lines)
* Builder.py:
  + Modifying the get\_direction() function so that it can actually set the start\_construction variable. Additionally, it will be able to tell the robot to travel in certain directions (down, up, positive x, positive y, etc). This will allow the robot to travel down to another location.
    - Ideas: adding a “time\_steps\_to\_construction” to the memory of the robot so that when it climbs down, after a certain number of steps, it attempts to construct a new beam which hopefully hits something (of course, this will only happen if the beam it is currently on will be able to support the additional weight).
  + Walking off the structure
    - The robot, when it walks off the structure, now points in the direction of the home. This could be modeled by some sort of signal sent from the home which tells it in which direction to travel when it needs more material.
* Worker.py
  + Redefining the construction algorithm to take into account the results from the analysis. Will probably have to check whether or not the analysis mode is available before we take the information into account.
  + Redefining the filter\_directions function in order to either tell the robot to construct (if there no directions are available.
    - Function will filter directions based on the variables specified. Self.upwards. self.memory[pos\_x], self.momery[pos\_y], etc. These variables will be either True (move in the positive variable direction), False (move in the negative directions) or None (direction does not matter).
    - Had to rewrite a few other functions. Will document later today.
* Main.py:
  + Fixing excel writing so that the floats are trimmed.
    - Did this inside of current\_state() function for robot.
* Warning:
  + If you change the filter\_directions() anonymous functions to being inclusive, due to the way the move function is written, the goes into an infinite loop. Should be careful with the filtering.
    - Adding a failsafe to move.
* Will be using VPython for the visualization of the structure. Make sure to install Python 3.2.2
  + VPython

Swarm July 8 2013

* Visualization is up and running. It is set up to launch right after each test, or whenever the function go is run from inside the Simulation class.
* Started work on the idea of allowing robots to rebuild parts of the structure that are too weak. Added the variables necessary to calculate our moment limits to variables.py
* Reworked the filtering functions as to allow for more options when moving.
* Added an at\_top function so that the robot has a way of telling whether or not it really is at the top
* Tested everything out with multiple robots. Created a way for only one robot to set down the first beam by separating the construction rules into the simple ones (those which are logical), and those which will require a deeper analysis of the model.
* There seems to be something wrong with the build function for ratios. It seems that when two beams are parallel, the closest point returned is on endpoint and then the beam is laid down horizontally. This makes it nearly impossible to reinforce when at the bottom of the structure

Swarm July 9 2013

* Reformed the program structure for easier management. The structure is now as follows.

swarm/ Top-level swarm package

\_\_init\_\_.py

helpers/ Subpackage containing helper files

\_\_init\_\_.py

commandline.py

errors.py

helpers.py

inout.py

vectors.py

robots/ Subpackage for robot swarm

\_\_init\_\_.py

automaton.py

builder.py

colony.py

movable.py

worker.py

sap2000/ Subpackage for communication with SAP2000

\_\_init\_\_.py

constants.py

sap2000.py

sap\_analysis.py

sap\_areas.py

sap\_base.py

sap\_frames.py

sap\_groups.py

sap\_lines.py

sap\_points.py

sap\_properties.py

structure/ Subpackage for python structure

\_\_init\_\_.py

beams.py

structure.py

construction.py Constants for construction (limits,etc)

main.py

run\_test.py

variables.py Constants for the program

vis\_test.py

visualization.py

Swarm July 9 2013

* Fixed bug when moving in single direction. Was passing the unit direction inside the function (pick\_direction). This meant that the robot was only ever moving by an amount of one. This has a few implications.
  + The robot was not moving onto the joint, so sometimes it would skip the joint without ever having had a choice to move onto the other beams. This would make it so that when it would acquire new directions, the only available ones would be up
    - It uses self.beam to acquire non-joint directions
  + The robot would get stuck going back and forth around one endpoint of its previous beam (since it was always moving 1). This only occurred after several hundred timesteps because that was the only time when the robot came close enough to an end of a beam for it to obtain a direction that mattered.
* In feasible\_directions, the robot goes ahead and stores information about broken beams. This information is then used by no\_directions\_available(). In a sense, this means that the robot wants to climb higher, not immediately repair.
  + Climbing higher is a priority.
  + If it can’t climb higher, then it looks through the beams that are in need of repair (as far as it is concerned)
    - Out of those, it picks the beam that needs the most repair and uses the information from that beam to climb down (a certain number of steps)
* Repair Mode:
  + Climb down x steps (determined by the angle of the beam that needs repairing and the angle at which we want to set a support beam).
  + While climbing down, look for another support beam set by some other robot. If you find it, climb out in that direction (recalculate the number of steps needed for construction), and once you are a point from which you can reach the beam in need, build)
    - Still need to decide if we want to build randomly, or if the robot is going to need to know how far out to build.’
  + If you do not find a support beam after the number of steps, set the steps to construction to 1, move up, and construct a support beam in the right angle and in the direction of the beam that needs supporting (so that another robot later on can hopefully use that beam for the repairs that are necessary)
    - In using the beam that needs repairing, we always use the j-end of the beam. The assumption is that the i-end is connected to something that supports it well enough, and the j-end (the end moving up), is the that needs reinforcing.