To start a SV model within the simvascular GUI you first need to make sure that you open a new sv project. (I don’t think this is completely necessary, but it helps during prototyping because users will probably be working within the GUI to make sure that their commands are being properly translated into the models they want to see.)

In the python interpreter

**>>>** import os, sys

**>>>** from sv import \*

Below are some of the most commonly used commands, their inputs, and outputs:

***Creating Paths:***

Path.pyPath() ← This is a path module that will end up being used to define and create all paths within the model

I’ve usually assigned the Path.pyPath() to a variable for easier calling. This does not take any inputs.

Ex. **>>>** p = Path.pyPath()

\*\***I’ll refer to the Path.pyPath() by this newly assigned “p” variable for the rest of this document**

p.NewObject(str) ← This function creates a new path within your model with the given name assigned by the input string. The output is the new path line. All subsequent path functions called will affect the most recently opened path object. (So keep that in mind while working!)

p.AddPoint(list) ← This function adds control points which will eventually define the path that is created by simvascular (the default is a spline). The list contains three numbers (either integers or floats) that give the point’s 3D coordinate [X,Y,Z].

Ex. **>>>** p.AddPoint([1.0, 0.0, 0.0])

p.CreatePath() ← This function takes no input; instead it creates a pyPath and adds it to the working repository under the name given in the most recent path object (so either the string given to the most recent p.NewObject() or a reopened object **this will be mentioned next**)

p.GetObject(str) ← This function can be used to recall previously made objects so that they can be adapted or deleted.

Ex. >>>p.GetObject(‘path1’)

\*\*in the example above, the p.GetObject() will only call the path if it exists in the repository already. In this case, ‘path1’ becomes the current working path object and any adaptation to the points will refer to the points in this path.

***Creating Contours:***

Contour.pyContour() ← Just like the overarching Path.pyPath() module, contouring has an analogous module used to define all the contours within a model. A good practice is to reassign the module to a short, simple variable name.

Ex. >>> c = Contour.pyContour()

\*\* **I’ll refer to Contour.pyContour() as “c” for the rest of this document**

Contour.SetContourKernel(str) ← This defines the type of contour slices that will be created. The string input specifies this. The available kernel options are:

· ‘Circle’

· ‘Threshold’

· ‘LevelSet’

· ‘Polygon’

· ‘SplinePolygon’

Ex. >>> Contour.SetContourKernel(‘Circle’)

\*\* For example, the above command will set the contour slice to be a circle, and, unless other

wise specified, all subsequent contours will be assumed to be a circle.

c.NewObject(str,str,int) ← This function creates a new slice given the name specified by the first string. This contour belongs to the path which is specified by the second string and the point within that path which is given by the integer (the third input).

Ex. >>> c.NewObject(‘ct1’,’path1’,0)

This example prescribed the first contour (i.e. at the first index of the point list defining the path) of ‘path1’. Note that this function does not prescribe a radius or actual geometry to the contour itself. It merely creates the contour about a specific location relative to a specific path.

c.SetCtrlPtsByRadius(list,int or float) ← this function takes a list of three integers or floats which specify the center of the contour and an additional integer or float which specifies the radius value. The contour will be normal to the path automatically.

Ex. >>> c.SetCtrlPtsByRadius([0,0,0], 20)

This example creates a contour at the point specified by the path with center [0,0,0]. The radius of this contour will be 20.

c.Create() ← this function takes no inputs and it creates the contour object and adds it to the repository. Note that single contour slices cannot be imported into the GUI and instead must be in a list form.

c.GetPolyData(str) ← this function takes a string which becomes the name of an individual contour polydata file. This polydata is used later when creating a lofted surface during solid creation. Note that this function will generate the polydata for the contour object that is ***currently*** being constructed. Thus, if you want to get the polydata for ***any other*** contour then you will have to retrieve that contour via the c.GetObject() command (which will be explained later).

Ex. >>> c.GetPolyData(‘ctp’)

In the example the current polydata is generated and saved in the working repository under the name “ctp”. This data can be called at any point later.

c.GetObject(str) ← just like the p.GetObject function in the path module. This function accepts a string argument that is the name of a created contour object. When this contour is retrieved it becomes the new contour that is under construction. Thus, any subsequent adaptations that are made will be applied to this object unless a new contour object is created or the c.GetObject() function is called again.

Ex. >>> c.GetObject(‘ct1’)

This example calls the “ct1” contour (assuming that such a contour exists under this name) and makes “ct1” the current working contour.

***Moving things to and from the GUI***

GUI.ImportPathFromRepos(str,\*optional string) ← This function moves created paths into the simvascular GUI. The first string should specify a path name that is supposed to be moved to the GUI. In the case that only a path name is given then this path will be moved under the “Repository” folder within the SV data manager window on the GUI. After this is done you should be able to visualize the path you have created within the SV GUI. This is a good way to debug any code you creating as well as ensure that the model you are creating is accurate. Note that even if you save the simvascular project, any data within this folder will not be saved; thus if you want to make sure that your data is not lost you should store any paths by instead storing them in the “Paths” folder within the SV data manager. To do this you will have to include an additional string input “Paths” in the function.

Ex. >>> GUI.ImportPathFromRepos(‘path1’,’Paths’)

This example moves the created ‘path1’ to the “Paths” folder. From here you can easily save the path if you want using the GUI.

GUI.ImportContoursFromRepos(str,list,str,\*optional string) ←This function moves created contour into the simvascular GUI. The first string should specify a name that will be given to the contour or group of contours being added to the GUI. The list should contain one or more individual contour slices names (given as strings) that have been made. The last required input is the path name that the contour group will be associated with; this string should specify a path that exists within the repository. If no optional string is given then the contour group will be added to the “Repository” folder in the SV data manager section of the GUI. Just as in the case of importing paths, the repository folder is not saved. If the contour groups need to be saved, “Segmentations” can be given in the optional string argument. This will store the contour group in the “Segmentations” folder in the SV data manager. This folder can be saved via the GUI.

Ex. >>> GUI.ImportContoursFromRepos(‘contour\_example’,[‘ct1’,’ct2’],’path1’,’Segmentations’)

This example creates the contour group in the GUI called “contour\_example” which has the contour slices “ct1” and “ct2” which were already created using the contour functions previously discussed. This contour group is associated with the path called “path1” and is stored in the “Segmentations” folder under the SV data manager section.

***Working with the Geometry Module***

In general the geometry module contains the vast amount of model manipulation functions and works in unison with the solid module to create a desired solid model for the user. Thus, it some functions within the geometry module depend upon and solid data and vice versa. The functionality described here will cover the basic functions needed to create a uniform, simple geometry. (For now concepts such as smoothing, warping, scaling, translation, surface mathematics, etc. will be omitted).

Geom.SampleLoop(str,int,str) ← This function creates a discrete sample of points using the polydata from the individual contour slices. This creates points that can later be aligned and used to loft an initial surface for the solid model. The second integer input specifies the number of points to discretize the contour slice space. The last string input is the name of the output file that will be generated and put into the working directory.

Ex. >>> Geom.SampleLoop(‘ctp’,100,’ctps’)

This example reads the polydata file “ctp” in the working repository and creates a discrete sample of 100 points (based on the second integer input) and writes this into the output file titled “ctps” (based on the third string input).

Geom.AlignProfile(str,str,str,int) ← the first two string inputs specify the two alignment groups of data generated by the SampleLoop() function. The third string specifies the name that will be given to the aligned data. The last integer input gives the index of the aligned segment.

Ex. >>> Geom.AlignProfile(‘ctps1’,’ctps2’,’ctpsa12’,0)

The example above takes the two discretized contours “ctps1” and “ctps2” from the working directory and aligns the second to the first. This newly aligned data is then given the name “ctpsa12” and the index 0. The aligned data is useful to ensure effective lofting.

Geom.LoftSolid(list,str,int,int,int,int,int,int) ←