Peter A. Jacobs, Rowan J. Gollan and Ingo Jahn

The Eilmer 4.0 flow simulation program:

Guide to the geometry package

for construction of flow paths.

June 2, 2019

Technical Report 2017/25
School of Mechanical & Mining Engineering
The University of Queensland

Abstract

The geometry package supports the construction of geometric elements such as surface patches and volumes within the Eilmer4 compressible-flow simulation program. Elements of the package are available to build a description of the gas-flow and solid domains that can be discretized and then passed onto the flow solver. The description of the flow domain is constructed as a Lua script which defines the locations and extents of the elements bounding the domain. For a 2D domain, you will use edges to bound patches in the (x,y)-plane and, for a 3D domain, you will define surfaces that meet at edges to define parametric volumes in (x,y,z)-space.

The flow solver expects the gas-flow and solid domains to be specified as meshes of finite-volume cells, so you will then need to discretize the 2D patches or 3D volumes. The StructuredGrid class is available to build these meshes by interpolating points within the patches and volumes. There is also an UnstructuredGrid class for when structured-grids are too difficult to generate nicely.

Eilmer4 is available as source code from https://bitbucket.org/cfcfd/dgd/ and is related to the larger collection of compressible flow simulation codes found at http://cfcfd.mechmining.uq.edu.au/.

Acknowledgement
This document was prepared while PAJ was on Special Studies Program at Oxford University.

Contents

1

Introduction

The geometry package supports the construction of geometric elements such as surface patches and volumes within the `Eilmer` flow simulation program[1]. The flow solver expects the gas-flow and solid domains to be specified as meshes of finite-volume cells. You may prepare these meshes in your favourite grid generation program and then import them into your simulation or you may prepare a description of the domain using the elements described in this report and then discretize it using one of the grid generators that are also included in the geometry package. The procedures described here are most convenient for meshing relatively simple domains but, with sufficient effort, can be applied to arbitrarily complex situations. They have the advantage that your simulation description is completely self-contained and you will not be dependent on external programs to get your simulation going. This report is a companion to the user's guide for the overall simulation program[2].

For structured meshes, the top-level geometric elements that are given to the grid generator are "patches" for 2D flow and "parametric volumes" for 3D flow. These are regions of space that may be traversed by a set of parametric coordinates $0 \leq r < 1$, $0 \leq s < 1$ in 2D and with the third parameter $0 \leq t < 1$ in 3D. These patches or volumes can be imported as VTK structured grids or they can be constructed as a "boundary representation" from lower-dimensional geometric entities such as paths and points.

For unstructured meshes[1], the boundary representation is provided to the grid generator as a set of discretized boundary edges in 2D or as a surface mesh in 3D. It is also possible to construct an unstructured mesh from a structured mesh or to import the unstructured mesh in VTK or SU2 format.

1.1 Some advice

Before describing the details of the geometric elements that you will use to build a description of your flow domain, we would like to offer some advice on the process of building that description. The process is one of programming the the geometry-building program to construct an encoded description of your flow domain. With this in mind, we advise the following procedure:

[1]Functions for unstructured-mesh generation are a work in progress.

1. Start with a rough sketch of your flow domain on paper, labelling key features.
2. Start small, building a script to describe a very simple element from your full domain.
3. Process this script with e4shared options `--prep` or `--custom-post` to produce either a rendered artifact or a grid.
4. View this artifact to see that it is what you wanted, debugging as required.
5. Proceed in small steps to complete your domain description.

We believe that this procedure will result in a far more satisfactory experience than coding your entire description in one pass. You might be lucky, but chances are that your mistakes will overpower your luck.

2

Geometric elements

The end goal of working with elements from the geometry package is to construct a representation of the flow domain that can be passed to the flow solver. For a two-dimensional flow simulation, this domain will be defined as one or more patches in the x,y-plane. Within the program code, these patches are represented as surface objects which are topologically quadrilateral and have boundary edges labelled north, east, south and west. The patches will be body-fitted, meaning that the domain boundaries take on the shape one or more bodies that contain the gas flow. For a three-dimensional flow simulation, the gas-flow domain will be described using body-fitted volume objects. These volumes and surfaces are constructed from lower-dimensional geometric objects, specifically points and paths.

2.1 Points

The most fundamental class of geometric object is the `Vector3` which represents a point in 3D space and has the usual behaviour of a geometric vector. The code for creating and manipulating these objects is in the Vector3 class of the underlying D-language module. This D code has been wrapped in such a way that the class methods are accessible from your Lua script.

In your Lua script, you might construct a new Vector3 object as:

```
Vector3:new{x=0.1, y=0.2, z=0.3}
```

When building models of 2D regions, you can omit the z-component value and it will default to zero. Note that we are using the object-oriented convention described in the Programming in Lua book [3]. In the Lua environment that has been set up by our program, `Vector3` is a table and `new` is a function within that table. In this particular case, we will think of the `new` function as our constructor function for `Vector3` objects. The use of the colon rather than a dot before the word `new` tells the interpreter that we want to include the object itself in the arguments passed to the called function. Finally, note the use of braces to construct a table of parameters that are given to the function. In the binding code for the `new` function, we are expecting all of the arguments to be passed on the stack to be contained in a single table. When constructing other objects within the input script, we have usually chosen a notation that has all of the elements as named attributes in a single table. We believe that this provides some benefit by making the order of arguments not significant. It also makes your script a little more self-documenting at the cost of being a bit more verbose.

It is possible to 'get' and 'set' values of attributes within a geometric element. For example, to create a node, extract the x-component of that node, change the y-component, or to use the components to construct a new point, you could use the following script.

```
1 -- example-1.lua
2
3 a = Vector3:new{x=0.5, y=0.8}
4 x_value = a.x
5 a.y = 0.4
6 b = Vector3:new{x=a.x+0.4, y=a.y+0.2}
7 print("a=", a, "b=", b)
8 print("x_value=", x_value)
9
10 dofile("sketch-example-1.lua")
```

On line 3, we have constructed a point and bound it to the name `a`. Lines 4 and 5 show examples of getting and setting components and line 6 constructs a new point from expressions involving point `a`. Line 10 calls up another file to make an SVG rendering of the points. We will discuss the content of that file in a later section. It is somewhat detailed and unimportant for the moment.

The transcript of running this script through the custom-postprocessing mode of the Eilmer4 program is:

```
1 $ e4shared --custom-post --script-file="example-1.lua"
2 Eilmer4 compressible-flow simulation code.
3 Revision: 0c2021bb8eb9 713 default tip
4 Begin custom post-processing using user-supplied script.
5 Start lua connection.
6 a=      Vector3([0.5, 0.4, 0])  b=      Vector3([0.9, 0.6, 0])
7 x_value=        0.5
8 Done custom postprocessing.
```

and the resulting points are displayed graphically in Figure 2.1. Note the output from the `print` function calls on lines 6 and 7 of the transcript. The format of the displayed Vector3 objects is that of the underlying D-language module rather than the Lua construction format used on line 3 of the input script. Internally, the coordinates are stored in fixed-size array, hence the printed format as list of coordinate values delimited by square brackets.

If you look into the file `dgd/src/geom/geom.d`, you will see that the `Vector3` objects support the usual vector operations of addition, subtraction and the like. Also, you can construct one Vector3 object from another and there are a small number of built-in transformations that might be useful for programmatically constructing your flow domain. For example, to create a point and its mirror image in the (y,z)-plane going through x=0.5, you could use:

```
1 -- example-2.lua
2
3 a = Vector3:new{x=0.2, y=0.5}
```

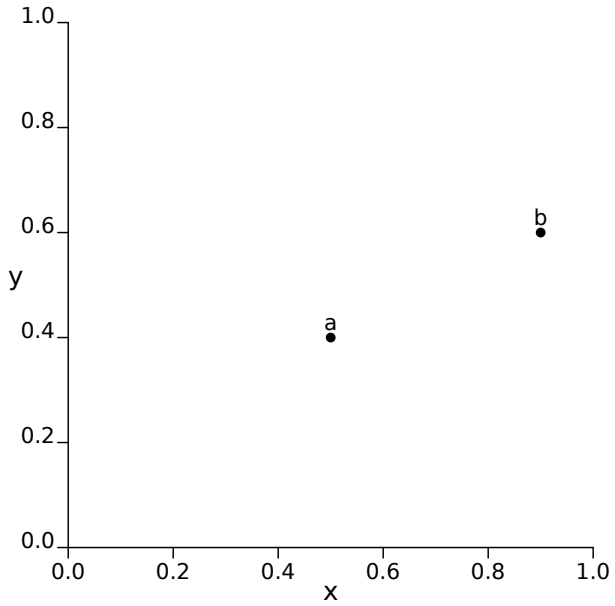


Figure 2.1: A couple of points defined as `Vector3` objects and rendered to SVG.

```
4 b = Vector3:new{a}
5
6 mi_point = Vector3:new{x=0.5} -- mirror-image plane through this point
7 mi_normal = Vector3:new{x=1.0} -- normal of the mirror-image plane
8 b:mirrorImage(mi_point, mi_normal)
9
10 print("a=", a, "b=", b)
11 print("abs(b)=", b:abs())
12
13 c = a + b
14 print("c=", c)
```

which has the corresponding transcript:

```
1 $ e4shared --custom-post --script-file="example-2.lua"
2 Eilmer4 compressible-flow simulation code.
3 Revision: 0c2021bb8eb9 713 default tip
4 Begin custom post-processing using user-supplied script.
5 Start lua connection.
6 a=      Vector3([0.2, 0.5, 0])  b=      Vector3([0.8, 0.5, 1.11022e-16])
7 abs(b)= 0.94339811320566
8 c=      Vector3([1, 1, 1.11022e-16])
9 Done custom postprocessing.
```

Note that, on line 8 of the example script, the `mirrorImage` method call passes the defining points for the mirror-image plane as unnamed but ordered arguments. These arguments are delimited by parentheses rather than braces, as would have been used for building a table. Note also, on line 11 of the script, that the method call for `abs` uses a pair of empty parentheses to make the method call with no argu-

ments. Finally, on line 6 of the transcript, note the round-off error appearing in the z-component of point `b`. This error is propagated into point `c`.

A list of operators and methods provided to the Lua environment is given in Table 2.1.

Table 2.1: Operators and methods for `Vector3` objects. In the table, Vector3 quantities are indicated as $\vec{a}$. Scalar quantities are double (or 64-bit) floating-point values. Some of the methods also appear as unbound functions in the global name space.

Lua expression	Result of evaluation
$-\vec{a}$	Vector3 object with negated components. $\vec{a}$ will not be changed.
$\vec{a} + \vec{b}$	Vector3 object with summed components
$\vec{a} - \vec{b}$	Vector3 object with subtracted components
$\vec{a} * s$	Vector3 object with components scaled by s
$s * \vec{b}$	Vector3 object with components scaled by s
$\vec{a}/s$	Vector3 object with components divided by s
$\vec{a}$`:normalize()`	make $\vec{a}$ into unit vector. $\vec{a}$ *will* be changed.
$\vec{a}$`:dot(`$\vec{b}$`)`	scalar dot product
$\vec{a}$`:abs()`	scalar magnitude of vector. $\vec{a}$ will not be changed.
$\vec{a}$`:unit()`	Vector3 object of unit magnitude in direction of $\vec{a}$. $\vec{a}$ will not be changed.
$\vec{a}$`:mirrorImage(`$\vec{p}$`,` $\vec{n}$`)`	Vector3 object that is the mirror image of $\vec{a}$ in the plane through $\vec{p}$ with normal $\vec{n}$. Note that $\vec{a}$ *will* be changed.
$\vec{a}$`:rotateAboutZAxis(`θ`)`	rotate $\vec{a}$ about the z-axis by θ radians. Note that $\vec{a}$ *will* be changed.
`dot(`$\vec{a}, \vec{b}$`)`	scalar dot product, as above
`vabs(`$\vec{a}$`)`	scalar magnitude, as above
`unit(`$\vec{a}$`)`	Vector3 object with unit magnitude and direction of $\vec{a}$. $\vec{a}$ will not be changed.
`cross(`$\vec{a}, \vec{b}$`)`	Vector3 cross product $\vec{a} \times \vec{b}$
`quadProperties{p0=`$\vec{p}_0$`, p1=`$\vec{p}_1$`, p2=`$\vec{p}_2$`, p3=`$\vec{p}_3$`}`	returns table `t` with named elements `centroid`, `n`, `t1`, `t2`, and `area`
`hexCellProperties{p0=`$\vec{p}_0$`, p1=`$\vec{p}_1$`, p2=`$\vec{p}_2$`, p3=`$\vec{p}_3$`, p4=`$\vec{p}_4$`, p5=`$\vec{p}_5$`, p6=`$\vec{p}_6$`, p7=`$\vec{p}_7$`}`	returns table `t` with named elements `centroid`, `volume`, `iLen`, `jLen`, and `kLen`

2.2 Paths

The next level of dimensionality is the `Path` class. A path object is a parametric curve in space, along which points can be specified via the single parameter $0 \leq t < 1$. `Path` is a base class in the D-language domain and a number of derived types of paths are available. Evaluating a path for a parameter value t results in the corresponding Vector3 value. A path may also be copied or printed as a string.

2.2.1 Simple Paths

There are several simple path objects that can be constructed from points. The constructors typically accept their arguments in a single table, with the items being named, as shown below:

- `Line:new{p0=`$\vec{a}$`, p1=`$\vec{b}$`}`: a straight line between points $\vec{a}$ and $\vec{b}$.
- `Arc:new{p0=`$\vec{a}$`, p1=`$\vec{b}$`, centre=`$\vec{c}$`}`: a circular arc from $\vec{a}$ to $\vec{b}$ around centre, $\vec{c}$. Be careful that you don't try to make an `Arc` with included angle of 180^o or greater. For such a situation, create two circular arcs and join as a `Polyline` path.
- `Arc3:new{p0=`$\vec{a}$`, pmid=`$\vec{b}$`, p1=`$\vec{c}$`}`: a circular arc from $\vec{a}$ through $\vec{b}$ to $\vec{c}$. All three points lie on the arc.
- `Bezier:new{points=`$\{\vec{b}_0, \vec{b}_1, ..., \vec{b}_n\}$`}`: a Bezier curve of order $n-1$ from $\vec{b}_0$ to $\vec{b}_n$. You can specify as many points as you like for the curve, however, 4 points for a third-order curve is very commonly used. A convenient feature of the Bezier curve is that you get direct control of the end points and the tangents at those end points. Note that the table of points is labelled and contained within the single table passed to the constructor. The individual points are not labelled and, although the subscripts shown start from zero (to be consistent with the representation within the core D-language code), the default numbering of items in a Lua table starts at 1. We have side-stepped the issue here by providing a literal construction of the points table, however, there are times when you need to account for this difference in indexing.

Some examples of constructing these simple paths is shown here:

```
1 -- path-example-1.lua
2
3 -- An arc with a specified pair of end-points and a centre.
4 c = Vector3:new{x=0.4, y=0.1}
5 radius = 0.2
6 a0 = Vector3:new{x=c.x-radius, y=c.y}
7 a1 = Vector3:new{x=c.x, y=c.y+radius}
8 my_arc = Arc:new{p0=a0, p1=a1, centre=c}
9
10 -- A Bezier curve of order 3.
11 b0 = Vector3:new{x=0.1, y=0.1}
12 b3 = Vector3:new{x=0.4, y=0.7}
13 b1 = b0 + Vector3:new{y=0.2}
```

```
14 b2 = b3 - Vector3:new{x=0.2}
15 my_bez = Bezier:new{points={b0, b1, b2, b3}}
16
17 -- A line between the Bezier and the arc.
18 my_line = Line:new{p0=b0, p1=a0}
19
20 -- Put an arc through three points.
21 a2 = Vector3:new{x=0.8, y=0.2}
22 a3 = Vector3:new{x=1.0, y=0.3}
23 my_other_arc = Arc3:new{p0=a1, pmid=a2, p1=a3}
24
25 dofile("sketch-path-example-1.lua")
```

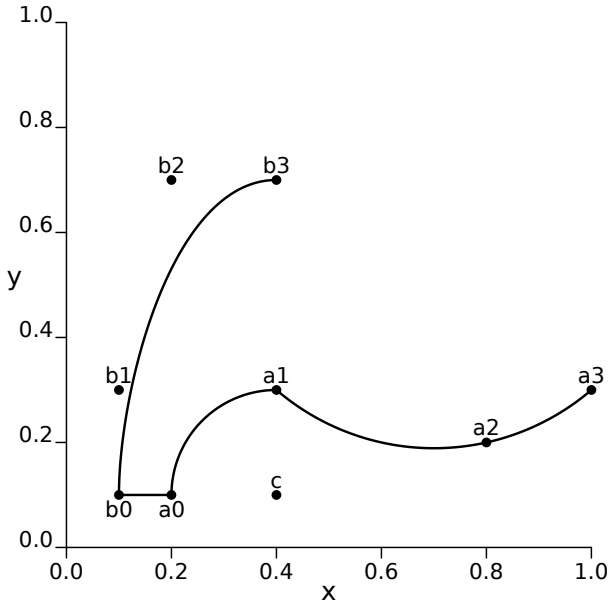


Figure 2.2: Some simple paths defined from `Vector3` objects and rendered to SVG.

2.2.2 User-defined-function path

For the ultimate in flexibility of path definition you can supply a Lua function to convert from parameter, t, to physical space. The following script, with geometry definition extracted from the sharp-body example for the simulation code, shows how to do this for a two-dimensional case. On line 23, a `LuaFnPath` is constructed by supplying the name of the Lua function as a string. The function `xypath`, defined starting on line 13, accepts values of the parameter, t, (in the range 0.0 to 1.0) and returns a table with `x`, `y` (and maybe `z`) named components that represent the point in space for parameter value t. The function `y(x)`, defined starting on line 4, is a helper function that is used to make the script look more like the formula in the textbook [4] that originally described the exercise. It is not necessary to split up the functions in this manner. Figure 2.3 shows a rendering of the path.

```
1 -- path-example-2.lua
2 -- Extracted from the 2D/sharp-body example.
3
4 function y(x)
5    -- (x,y)-space path for x>=0
6    if x <= 3.291 then
7       return -0.008333 + 0.609425*x - 0.092593*x*x
8    else
9       return 1.0
10   end
11 end
12
13 function xypath(t)
14   -- Parametric path with 0<=t<=1.
15   local x = 10.0 * t
16   local yval = y(x)
17   if yval < 0.0 then
18      yval = 0.0
19   end
20   return {x=x, y=yval}
21 end
22
23 my_path = LuaFnPath:new{luaFnName="xypath"}
24
25 dofile("sketch-path-example-2.lua")
```

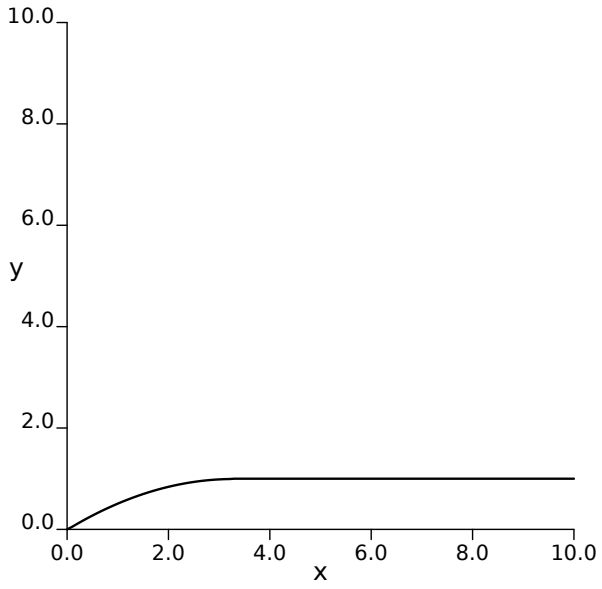


Figure 2.3: A path defined via a Lua function and rendered to SVG. This particular path is used to represent the profile of the body in the 2D/sharp-body simulation exercise for Eilmer4.

2.2.3 Compound Paths

Sometimes you wish to have a single parametric path object to pass into a grid-generation function but it is convenient to define that part in smaller pieces. The following path subtypes may be constructed:

- `Polyline:new{segments={`$p_0, p_1, ..., p_n$`}}`: a composite path made up of the segments p_0, through p_n, each being a previouly defined `Path` object. The individual segments are reparameterised, based on arc length, so that the composite curve parameter is $0 \leq t < 1$.
- `Spline:new{points={`$\vec{b}_0, \vec{b}_1, ..., \vec{b}_n$`}}`: a cubic spline from $\vec{b}_0$ through $\vec{b}_1$, to $\vec{b}_n$. A `Spline` is actually a specialized `Polyline` containing $n - 1$ cubic Bezier segments.
- `Spline2:new{filename="something.dat"}`: a spline constructed from a file containing x y z coordinates of the interpolation points, one point per line. The coordinate values are expected to be space separated. If the y or z values are missing, they are assumed to be zero.

The following script builds a combined path object that looks similar to the first example:

```
1 -- path-example-3.lua
2
3 -- An arc with a specified pair of end-points and a centre.
4 c = Vector3:new{x=0.4, y=0.1}
5 radius = 0.2
6 a0 = Vector3:new{x=c.x-radius, y=c.y}
7 a1 = Vector3:new{x=c.x, y=c.y+radius}
8 my_arc = Arc:new{p0=a0, p1=a1, centre=c}
9
10 -- A Bezier curve of order 3.
11 b3 = Vector3:new{x=0.1, y=0.1}
12 b0 = Vector3:new{x=0.4, y=0.7}
13 b1 = b0 - Vector3:new{x=0.2}
14 b2 = b3 + Vector3:new{y=0.2}
15 my_bez = Bezier:new{points={b0, b1, b2, b3}}
16
17 -- A line between the Bezier and the arc.
18 my_line = Line:new{p0=b3, p1=a0}
19
20 -- Put an arc through three points.
21 a2 = Vector3:new{x=0.8, y=0.2}
22 a3 = Vector3:new{x=1.0, y=0.3}
23 my_other_arc = Arc3:new{p0=a1, pmid=a2, p1=a3}
24
25 one_big_path = Polyline:new{segments={my_bez, my_line, my_arc,
26                                       my_other_arc}}
27
28 dofile("sketch-path-example-3.lua")
```

Note, that to make a single path that can be traversed, we needed to define the Bezier curve using points in the opposite order to the earlier example.

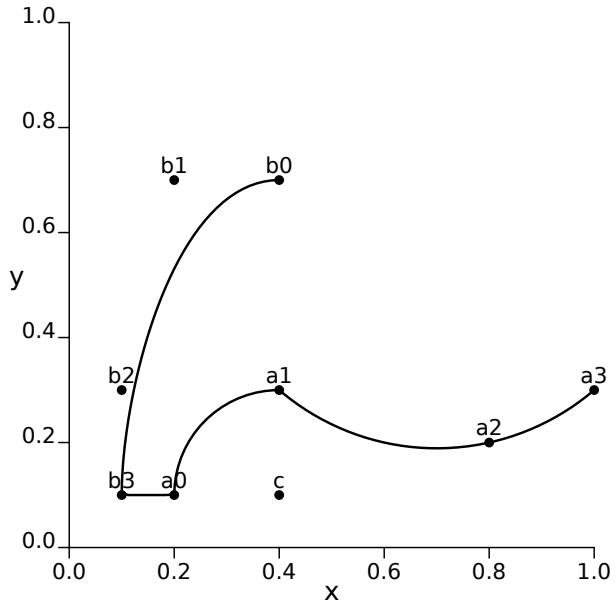


Figure 2.4: A single compound path defined from a set of path objects and rendered to SVG.

If we take the same data points as used to construct the Polyline, we can see the difference it make to use a Spline constructor in Figure 2.5. Sometimes, it is just what you want, and other times, not so.

```
1 -- path-example-4.lua
2 -- A single spline through the same data points.
3 c = Vector3:new{x=0.4, y=0.1}
4 radius = 0.2
5 a0 = Vector3:new{x=c.x-radius, y=c.y}
6 a1 = Vector3:new{x=c.x, y=c.y+radius}
7
8 b3 = Vector3:new{x=0.1, y=0.1}
9 b0 = Vector3:new{x=0.4, y=0.7}
10 b1 = b0 - Vector3:new{x=0.2}
11 b2 = b3 + Vector3:new{y=0.2}
12
13 a2 = Vector3:new{x=0.8, y=0.2}
14 a3 = Vector3:new{x=1.0, y=0.3}
15
16 my_spline = Spline:new{points={b0,b1,b2,b3,a0,a1,a2,a3}}
17
18 dofile("sketch-path-example-4.lua")
```

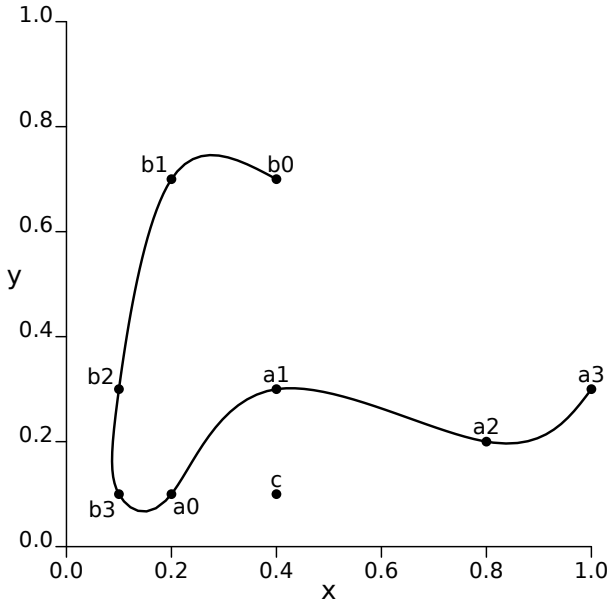


Figure 2.5: A Spline path defined on the same set of points that we used for the Polyline.

2.2.4 Derived Paths

We can construct new path objects as derivatives of previously defined paths. There are a number of constructors with differing transformations:

- `ArcLengthParameterizedPath:new{underlying_path=`pth`}`: Sometimes a Bezier curve, Spline or Polyline may have its defining points distributed such that the grid built upon it is not clustered in a good way. To fix this, it may be useful to specfy the curve to be parameterized by arc length.
- `SubRangedPath:new{underlying_path=`pth`, t0=`t_0`, t1=`t_1`}`: The new path is a subset of the original. If $t_0 > t_1$, the new path is traversed in the reverse direction.
- `ReversedPath:new{underlying_path=`pth`}`: is useful for building patches with common edges. For example, if the east edge of the first patch is common with the south edge of a second patch, the direction of traverse is reversed.
- `TranslatedPath:new{original_path=`pth`, shift=`$\vec{a}$`}`: Points on the new path are translated by $\vec{a}$ from corresponding points on the original path.
- `MirrorImagePath:new{original_path=`pth`, point=`$\vec{p}$`, normal=`$\vec{n}$`}`: The point $\vec{p}$ is the anchor point for the plane through which the image is made and $\vec{n}$ is the unit normal of that plane.
- `RotatedAboutZAxisPath:new{original_path=`pth`, angle=`θ`}`: Useful for building bodies of revolution in 3D.

Here are some examples of building derived paths, with the results shown in Figure 2.6:

```
1 -- path-example-5.lua
2 -- Transformed Paths
3
4 b0 = Vector3:new{x=0.1, y=0.1}
5 b1 = b0 + Vector3:new{x=0.0, y=0.05}
6 b2 = b1 + Vector3:new{x=0.05, y=0.1}
7 b3 = Vector3:new{x=0.8, y=0.1}
8 path0 = Bezier:new{points={b0,b1,b2,b3}}
9
10 path1 = TranslatedPath:new{original_path=path0, shift=Vector3:new{y=0.5}}
11 path1b = MirrorImagePath:new{original_path=path1,
12                              point=Vector3:new{x=0.5, y=0.6},
13                              normal=Vector3:new{y=1.0}}
14
15 path2 = TranslatedPath:new{original_path=path0, shift=Vector3:new{y=0.5}}
16 path2b = ArcLengthParameterizedPath:new{underlying_path=path2}
17
18 dofile("sketch-path-example-5.lua")
```

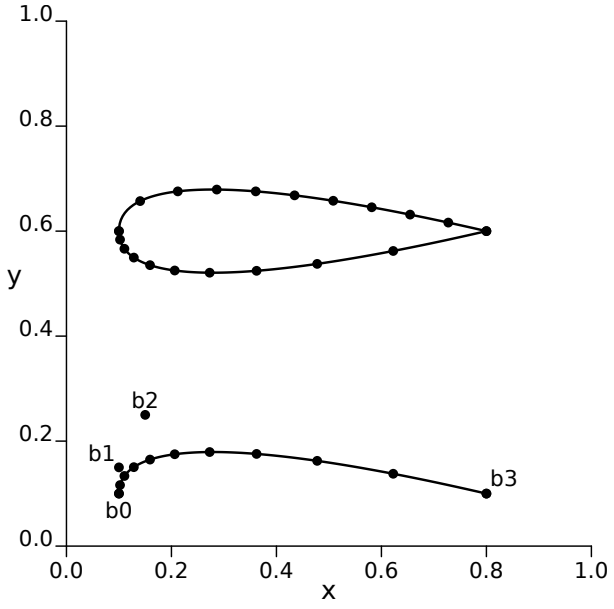


Figure 2.6: The original Bezier curve defined with points b0, b1, b2 and b3 in the lower part of the figure. A pair of derived paths is shown above it. The dotted points on all of the paths indicate equal increments in parameter t. Note the more uniform distribution in x,y-space for the ArcLengthParameterizedPath.

2.3 Surfaces

The `ParametricSurface` class represents two-dimensional objects which can be constructed from `Path` objects. These can be used as the `ParametricSurface` objects that are passed to the `StructuredGrid` constructor (Sec. 3.2) or they can be used to form the bounding surfaces of a 3D `ParametricVolume` object (Sec. 2.4). ParametricSurface objects can be evaluated as functions of parameter pair (r, s) to yield corresponding Vector3 values, representing a point in modelling space. Here $0 \leq r \leq 1$ and $0 \leq s \leq 1$.

Examples of the most commonly used surface patches are:

- `CoonsPatch:new{south=`$path_S$`, north=`$path_N$`, west=`$path_W$`, east=`$path_E$`}`: a transfinite interpolated surface between the four paths. It is expected that the paths join at the corners of the patch, such that $path_S(0) = path_W(0) = p_{00}$, $path_S(1) = path_E(0) = p_{10}$, $path_N(0) = path_W(1) = p_{01}$ and $path_N(1) = path_E(1) = p_{11}$. See the left part of Figure 2.7 for the layout of this surface. Although the order of specifying the named edges is not significant, it is important to be careful with the orientation of the `Path` elements that form the patch boundaries. The `north` and `south` boundaries progress `west` to `east` with parameter r increasing from 0 to 1. The `west` and `east` boundaries progress `south` to `north` with parameter s increasing from 0 to 1. If the preparation stage of the program complains that the corners of your patch are "open", that may be a symptom of having one, or more, of your bounding paths having incorrect orientation.

- `CoonsPatch:new{p00=`$\vec{p}_{00}$`, p10=`$\vec{p}_{10}$`, p11=`$\vec{p}_{11}$`, p01=`$\vec{p}_{01}$`}`: a quadrilateral surface defined by it corners. Straight line segments (implicitly) join the corners. This is convenient for building simple regions that can be tiled with straight edged patches, since you don't need to explicitly generate Line objects to form the edges of each patch. Note that the order for specifying the corners is not significant.

- `AOPatch:new{south=`$path_S$`, north=`$path_N$`, west=`$path_W$`, east=`$path_E$`, nx=10, ny=10}`: an interpolated surface, bounded by four paths. When constructed, this surface sets up a background mesh with resolution specified by `nx` and `ny`. The construction method, based on an elliptic grid generator [5], tries to keep the background mesh orthogonal near the edges and also tries to keep equal cell areas across the surface. The background mesh is retained within the `AOPatch` object and, if the `AOPatch` is later passed to the grid generator, the final grid is produced by interpolating within this background mesh. The default background mesh of 10×10 seems to work fairly well in simple situations. If the bounding paths have strong curvature, it may be beneficial to increase the resolution of the background mesh, so that the final interpolated mesh does not cut across the boundary paths. This is especially important if the final grid lines are clustered close to the boundaries. Setting a higher resolution of the background mesh will require more iterations to reach convergence and you may actually see a warning message that the iteration did not converge. Fortunately, an unconverged background mesh is usually fine for use, because it

starts as `CoonsPatch` mesh and each iteration should just improve the metrics of orthogonality and distribution of cell areas.

- `AOPatch:new{p00=`$\vec{p}_{00}$`, p10=`$\vec{p}_{10}$`, p11=`$\vec{p}_{11}$`, p01=`$\vec{p}_{01}$`, nx=10, ny=10}`: a quadrilateral surface defined by it corners. Straight line segments (implicitly) join the corners. As shown in Figure 2.7, the difference with the corresponding CoonsPatch is that the background mesh, here, tries to be orthogonal to the edges and maintain equal cell areas across the surface.

Here are two examples of setting up simple patches:

```
1 -- surface-example-1.lua
2
3 -- Transfinite interpolated surface
4 a = Vector3:new{x=0.1, y=0.1}; b = Vector3:new{x=0.4, y=0.3}
5 c = Vector3:new{x=0.4, y=0.8}; d = Vector3:new{x=0.1, y=0.8}
6 surf_tfi = CoonsPatch:new{north=Line:new{p0=d, p1=c},
7                           south=Line:new{p0=a, p1=b},
8                           west=Line:new{p0=a, p1=d},
9                           east=Line:new{p0=b, p1=c}}
10
11 -- Knupp's area-orthogonality surface
12 xshift = Vector3:new{x=0.5}
13 p00 = a + xshift; p10 = b + xshift;
14 p11 = c + xshift; p01 = d + xshift;
15 surf_ao = AOPatch:new{p00=p00, p10=p10, p11=p11, p01=p01}
16
17 dofile("sketch-surface-example-1.lua")
```

As well as the constructors mentioned above, there is a convenience function `makePatch{south=`$path_S$`, north=`$path_N$`, west=`$path_W$`, east=`$path_E$`, gridType="TFI"}` which returns an interpolated surface. It returns either a CoonsPatch (TFI, by default) or an AOPatch object (`gridType="AO"`). The convenience is really limited to allowing your scripts to be a little more like the Eilmer3 input script, if you happen to be porting an old example to Eilmer4.

As listed below, there are more surface patch constructors, however, you will need to refer to their source code for documentation.

- `ChannelPatch:new{south=`$path_S$`, north=`$path_N$`, ruled=false, pure2D=false}`: an interpolated surface between two paths. By default, cubic Bezier curves are used to bridge the region between the defining curves, resulting in a grid that is orthogonal to those curves. Providing a `true` value for the `ruled` parameter results in the bridging curves being straight line segments. If you are integrating this patch into a larger construction, it may be convenient to obtain the `west` and `east` bounding curves by calling its `make_bridging_path` method with arguments of `0.0` and `1.0`, respectively. The argument for `pure2D` may be set to `true` to set all z-coordinate values to zero for purely two-dimensional constructions.

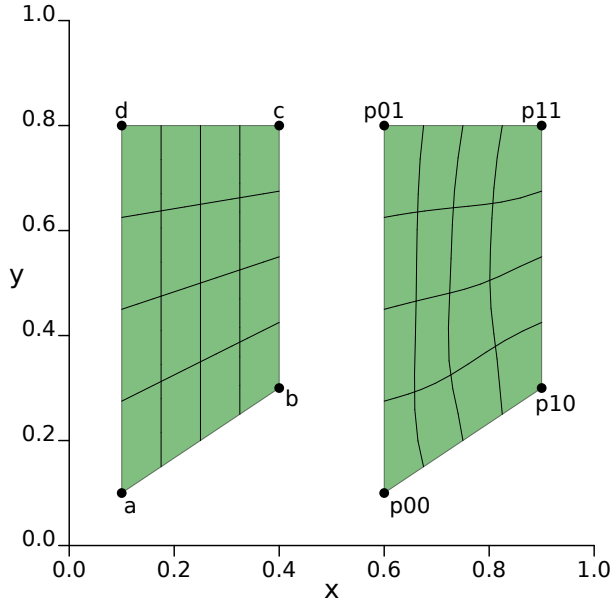


Figure 2.7: An example of a CoonsPatch and an AOPatch. The south-to-north lines drawn over the patches are for constant values or r and the west-to-east lines are for constant values of s.

- `SweptPathPatch:new{west=`$path_W$`, south=`$path_S$`}`: a surface, anchored by the south path, and generated by sweeping the west path across the south path.
- `LuaFnSurface:new{luaFnName="`$myLuaFnName$`"}`: a surface defined by the user-supplied function, $f(r, s)$. The user function returns a table of three labelled coordinates, representing the point in 3D space for parameter values r and s. If you are trying to build a 2D simulation, just set the z-coordinate to zero.
- `MeshPatch:new{sgrid=`$myStructuredGrid$`}`: a surface defined over a previously-defined structured mesh of quadrilateral facets. The structured mesh may have been imported (in VTK format) from an external grid generator.
- `SubRangedSurface:new{underlying_surface=`$mySurf$`, r0=0.0, r1=1.0, s0=0.0, s1=1.0}`: can select a section of a surface by setting suitable values of r0, r1, s0 and s1. Default values for the full parametric range are shown.

Here are examples of setting up the two flavours of ChannelPatches:

```
1 -- surface-example-2.lua
2
3 path1 = Arc3:new{p0=Vector3:new{x=0.2, y=0.3},
4                  pmid=Vector3:new{x=0.5, y=0.25},
5                  p1=Vector3:new{x=0.8, y=0.35}}
6 p = Vector3:new{y=0.2}; n = Vector3:new{y=1.0}
7 path2 = MirrorImagePath:new{original_path=path1,
8                             point=p, normal=n}
9 channel = ChannelPatch:new{north=path1, south=path2,
```

```
                              ruled=false, pure2D=true}
bpath0 = channel:make_bridging_path(0.0)
bpath1 = channel:make_bridging_path(1.0)

yshift = Vector3:new{y=0.5}
path3 = TranslatedPath:new{original_path=path1, shift=yshift}
path4 = TranslatedPath:new{original_path=path2, shift=yshift}
channel_ruled = ChannelPatch:new{north=path3, south=path4,
                                 ruled=true, pure2D=true}

dofile("sketch-surface-example-2.lua")
```

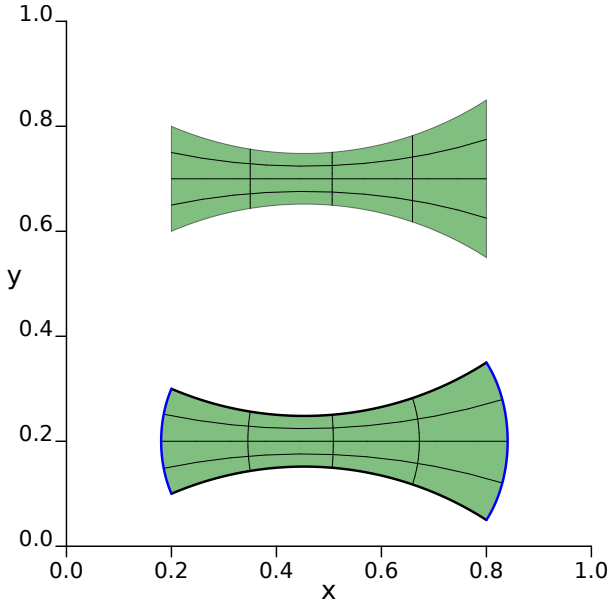


Figure 2.8: Example of ChannelPatch objects. The south-to-north lines drawn over the patches are for constant values or r and the west-to-east lines are for constant values of s. The lower patch uses the (default) Bezier curves bridging the north and south defining paths. The south and north paths are rendered as thick black lines and the $r = 0$ and $r = 1$ bridging paths are shown in blue. The upper example uses straight-line bridging paths.

The following script shows the construction of a SweptPathPatch, where the south path anchors the surface and the west path is swept across it. Note, in Figure 2.9, that the path constructed for the west path is not actually part of the final surface. The LuaFnSurface example maps the unit square in (r,s)-space to the region between two circles in the (x,y)-plane.

```
-- surface-example-3.lua

path1 = Arc3:new{p0=Vector3:new{x=0.2, y=0.7},
                 pmid=Vector3:new{x=0.5, y=0.65},
                 p1=Vector3:new{x=0.8, y=0.75}}
```

```
path2 = Line:new{p0=Vector3:new{x=0.1, y=0.7},
                 p1=Vector3:new{x=0.1, y=0.9}}
swept = SweptPathPatch:new{south=path1, west=path2}

function myFun(r, s)
   -- Map unit square to the region between two circles
   local theta = math.pi * 2 * r
   local bigR = 0.05*(1-s) + 0.2*s
   local x0 = 0.5; local y0 = 0.3
   return {x=x0+bigR*math.cos(theta), y=y0+bigR*math.sin(theta), z=0.0}
end
funSurf = LuaFnSurface:new{luaFnName="myFun"}

dofile("sketch-surface-example-3.lua")
```

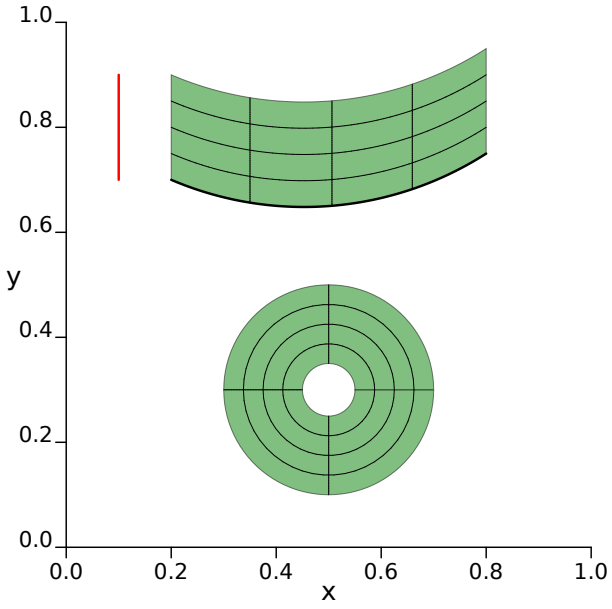


Figure 2.9: Example of a SweptPathPatch (upper) and a LuaFnSurface (lower). The south-to-north lines drawn over the patches are for constant values or r and the west-to-east lines are for constant values of s.

2.3.1 Paths on Surfaces

In Eilmer3, a subclass of Path was available for constructing a path over a parametric surface. As well as the underlying surface, it consisted of a pair of function objects for evaluating $r(t)$ and $s(t)$. These were limited to linear functions of t. In Eilmer4, the way to make a path on a parametric surface is to construct a LuaFnPath. This is more flexible in that any function can now be used for $r(t)$ and $s(t)$.

2.4 Volumes

Working in three dimensions significantly increases the amount of detail needed to specify a flow domain. In 2D, a simple region may be represented as a single surface patch with 4 bounding surfaces but, in 3D, the simple volume is now bounded by 6 surfaces, each with 4 edges. With an edge being shared by two surfaces, there will be 12 distinct edges on the volume. The ParametricVolume object, as shown in Figure 2.10 maps r,s,t parametric coordinates to x,y,z physical-space coordinates within the volume. To assist in understanding the orientation of the corners, surfaces and indices, you can build a model block from the development plan in Appendix A. This should bring back fond memories of kindergarten and primary school, at least it did for us.

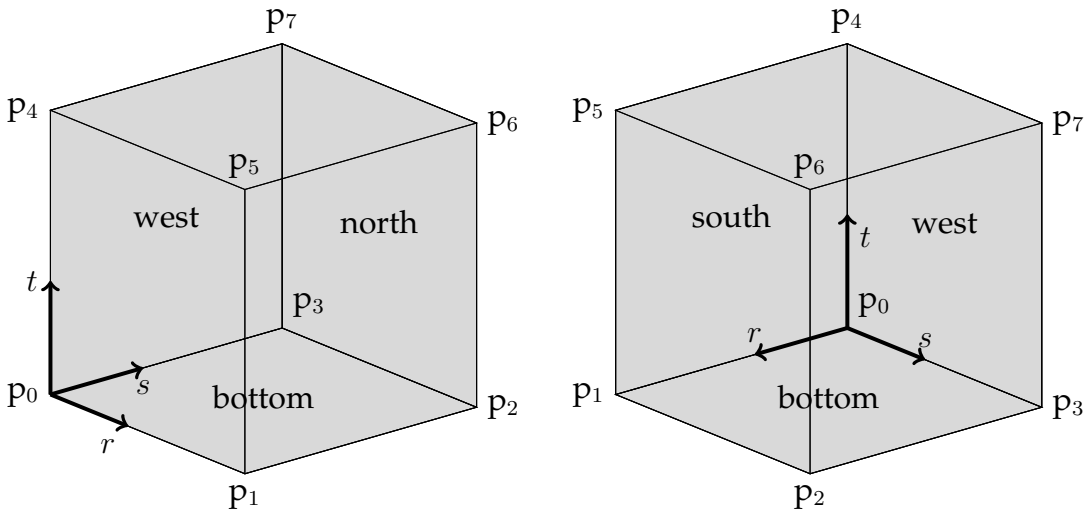


Figure 2.10: Two views of the parametric volume defined by its six bounding surfaces. These figures are ambiguous but each is supposed to show a hollow box with the *far* surfaces in each view being labelled. The *near* surfaces are transparent and unlabelled. On the left view, the p_1–p_5 edge is closest to the viewer. On the right view, the p_2–p_6 edge is closest to the viewer.

There are a number of constructors:

- `TFIVolume:new{north=`$surf_N$`, east=`$surf_E$`, south=`$surf_S$`, west=`$surf_W$`, top=`$surf_T$`, bottom=`$surf_B$`}` constructs the parametric volume from a set of six parametric surfaces to form a body-fitted hexahedral volume. It is assumed that the surfaces are consistent, in that they align appropriately at their edges, leaving no gaps in the bounding surface. Minimal checks are done to be sure that the corner locations (at least) are consistent.

- `TFIVolume:new{vertices=}`$\{\vec{p}_0, \vec{p}_1, \vec{p}_2, \vec{p}_3, \vec{p}_4, \vec{p}_5, \vec{p}_6, \vec{p}_7\}$: defines the volume by its vertices. Straight-line edges are assumed between these points. The subscripts correspond to the labelled points in Figure 2.10.

- `SweptSurfaceVolume:new{face0123=`$surf$`,edge04=`$path$`}`: constructs the volume by extruding the surface along the path. When evaluating points, the

calculation is actually done as $path(t) + surf(r, s) - surf(0, 0)$ such that the path effectively anchors the volume in x,y,z-space.

- `TwoSurfaceVolume:new{face0123=`$surf_{bottom}$`, face4567=`$surf_{top}$`}`: constructs the volume by linearly interpolating between bottom and top surface. When evaluating points, the calculation is actually done as $(1.0 - t) \times surf_{bottom}(r, s) + t \times surf_{top}(r, s)$ resulting in a straight line between corresponding points on the two surfaces.

- `LuaFnVolume:new{luaFnName="`$myLuaFunction$`"}`: constructs the volume from a user-supplied Lua function. For example, the simple unit cube could be defined with:

```
function myLuaFunction(r, s, t)
   return {x=r, y=s, z=t}
end
```

- `SubRangedVolume:new{underlying_volume=`$pvolume$`, r0=0, r1=1, s0=0, s1=1, t0=0, t1=1}`: constructs the volume as a subregion of a previously-defined volume. Any of the `r, s, t` limits that are not specified assume the default values shown.

Just about the simplest ParametricVolume that can be defined is the regular hexahedral block defined by its vertices. Although the example is simple, there is the same subtle indexing issue to consider, as we have seen for the construction of Bezier paths (see page 7). On line 21 of the script, the table of vertices is assembled with numbering of the points being implicit. Be aware that, by default, Lua will start numbering the table items from 1. This is a continuing point of friction with the core program written in the D programming, where array indices start at zero, so be careful if you explicitly number the vertices. The Lua interface code for the constructor is expecting to receive the points with default Lua numbering, even though we have tried to consistently use D-language indexing in our documentation. In this example, we hide the indexing issue by using a literal expression for the table.

```
1 -- volume-example-1.lua
2 -- Transfinite interpolated volume
3
4 Lx = 0.2; Ly = 0.1; Lz = 0.5 -- size of box
5 x0 = 0.3; y0 = 0.1; z0 = 0.0 -- location of vertex p000/p0
6
7 -- There is a standard order for the 8 vertices that define
8 -- a volume with straight edges.
9 p000 = Vector3:new{x=x0, y=y0, z=z0}          -- p0
10 p100 = Vector3:new{x=x0+Lx, y=y0, z=z0}       -- p1
11 p110 = Vector3:new{x=x0+Lx, y=y0+Ly, z=z0}    -- p2
12 p010 = Vector3:new{x=x0, y=y0+Ly, z=z0}       -- p3
13 p001 = Vector3:new{x=x0, y=y0, z=z0+Lz}       -- p4
14 p101 = Vector3:new{x=x0+Lx, y=y0, z=z0+Lz}    -- p5
15 p111 = Vector3:new{x=x0+Lx, y=y0+Ly, z=z0+Lz} -- p6
16 p011 = Vector3:new{x=x0, y=y0+Ly, z=z0+Lz}    -- p7
```

```
17
18 -- Assemble a table literal to avoid the issue
19 -- of numeric indices starting at 1 in Lua code
20 -- whilst starting at 0 in D code.
21 vList = {p000, p100, p110, p010, p001, p101, p111, p011}
22
23 my_vol = TFIVolume:new{vertices=vList}
24
25 dofile("sketch-volume-example-1.lua")
```

Let's build the same box volume with a Lua function that is defined within the input script. The function must return a single table with the x, y and z coordinates labelled as such. The script is actually shorter than the previous example but the computational machinery that calls back into the Lua interpreter to get the coordinate values has a cost. For complex geometries with large grids, this might make the preparation stage run slowly. For simple geometries, this cost is not an issue and your time preparing scripts is much more valuable, so you should choose the approach that you find convenient.

```
1 -- volume-example-2.lua
2 -- Volume defined on a Lua function.
3
4 Lx = 0.2; Ly = 0.1; Lz = 0.5 -- size of box
5 x0 = 0.3; y0 = 0.1; z0 = 0.0 -- location of vertex p000/p0
6
7 function myLuaFn(r, s, t)
8    return {x=x0+Lx*r, y=y0+Ly*s, z=z0+Lz*t}
9 end
10
11 my_vol=LuaFnVolume:new{luaFnName="myLuaFn"}
12
13 dofile("sketch-volume-example-2.lua")
```

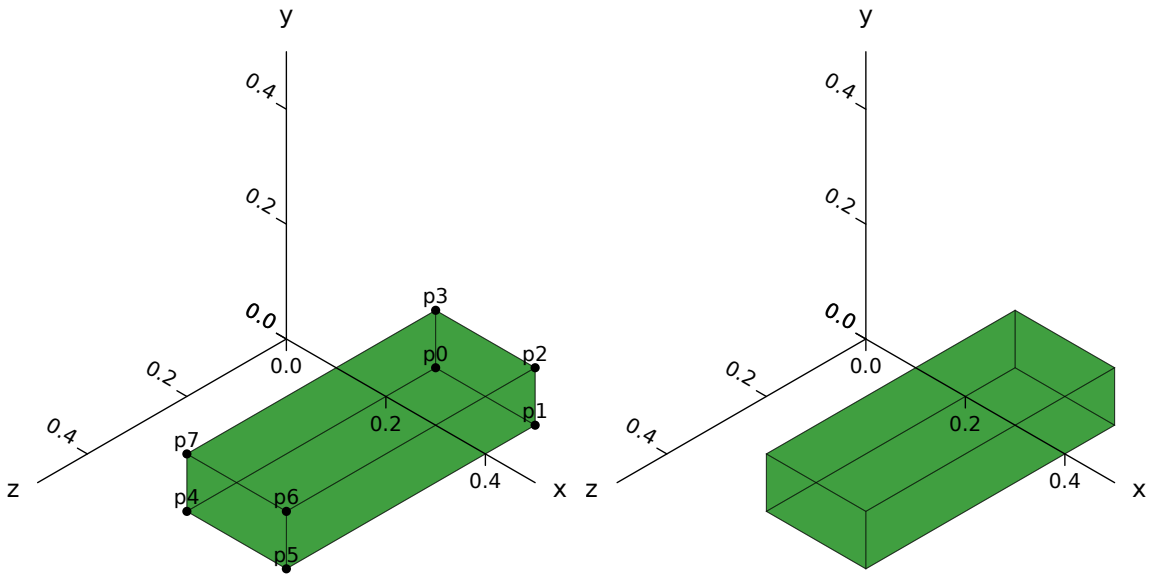


Figure 2.11: Isometric views of a regular hexahedral volume. The left view shows the volume defined by defined by its vertices. The right view shows the same volume defined by a Lua function.

A very commonly studied 3D configuration is the cylinder with cross flow. The following script constructs a flow volume around part of a cylindrical surface. We first construct the (future) bottom surface in the z=0 plane and then sweep out the volume by following the line b0-d parallel to the z-axis.

```
-- volume-example-3.lua
-- Parametric volume construced by sweeping a surface..

L = 0.4 -- cylinder length
R = 0.2 -- cylinder radius

-- Construct the arc along the edge of the cylinder
a0 = Vector3:new{x=R, y=0}; a1 = Vector3:new{x=0, y=R}
c = Vector3:new{x=0, y=0}
arc0 = Arc:new{p0=a0, p1=a1, centre=c}

-- Use a Bezier curve for the edge of the outer surface.
b0 = Vector3:new{x=1.5*R, y=0}; b1 = Vector3:new{x=b0.x, y=R}
b3 = Vector3:new{x=0, y=2.5*R}; b2 = Vector3:new{x=R, y=b3.y-R/2}
bez0 = Bezier:new{points={b0, b1, b2, b3}}

-- Construct the bottom surface
surf0 = CoonsPatch:new{west=bez0, east= arc0,
                       south=Line:new{p0=b0, p1=a0},
                       north=Line:new{p0=b3, p1=a1}}
-- Construct the edge along which we will sweep the surface.
d = Vector3:new{x=b0.x, y=b0.y, z=b0.z+L}
line0 = Line:new{p0=b0, p1=d}

my_vol = SweptSurfaceVolume:new{face0123=surf0, edge04=line0}

dofile("sketch-volume-example-3.lua")
```

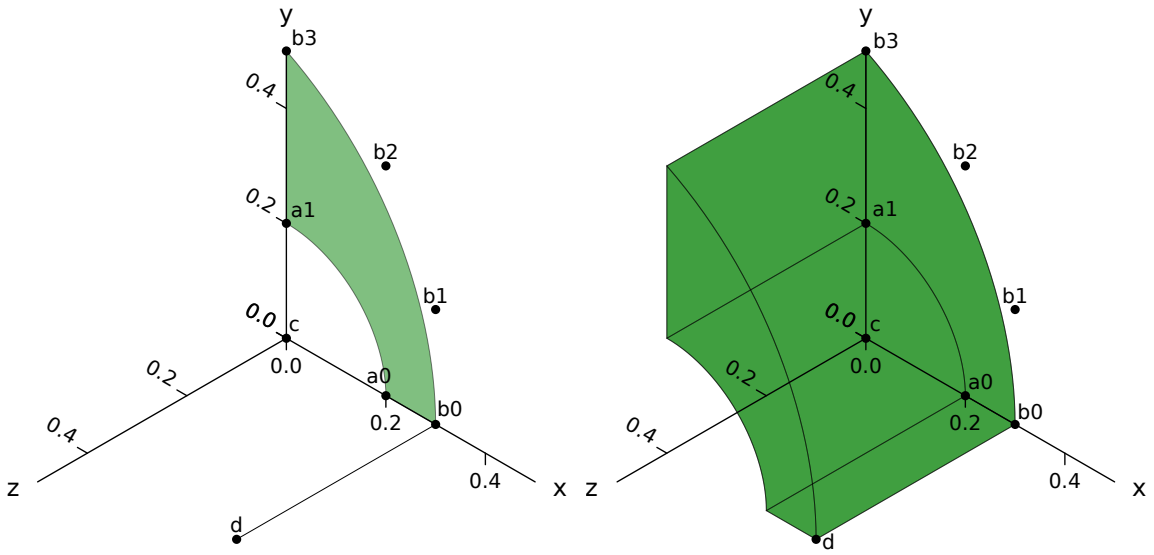


Figure 2.12: A volume constructed over part of a cylinder. The left view shows the surface that will be swept along the line b0-d and the right view shows the constructed volume.

2.5 Manipulating elements

A major advantage of defining geometries using a programming language is that the same very programing language can then be used to parameterise and automatically manipulate your geometry and mesh. This section illustrates a few examples of how to use this capability.

2.5.1 Parametric geometry

In the simplest form a number of variables, such as a scaling factor or specific angles for walls can be defined a-priori. Shown below is a parameterised example for the sharp-cone simulation as presented in the flow-solver user's guide. In the first few lines, several Lua variables are initialized to particular values, either with literals or algebraic expressions. These variables are then used to define the elements that define the flow domain in a fully-parametric manner.

```
1 -- sharp-cone-parameterised-example.lua
2
3 -- Parameters defining cone and flow domain.
4 theta = 32 -- cone half-angle, degrees
5 L = 0.8    -- axial length of cone, metres
6 rbase = L * math.tan(math.pi*theta/180.0)
7 x0 = 0.2   -- upstream distance to cone tip
8 H = 2.0    -- height of flow domain, metres
9
10 -- Set up two quadrilaterals in the (x,y)-plane by first defining
11 -- the corner nodes, then the lines between those corners.
12 a = Vector3:new{x=0.0, y=0.0}         --       f----e---------d
13 b = Vector3:new{x=x0, y=0.0}          --       |    |         |
14 c = Vector3:new{x=x0+L, y=rbase}      --       |quad|   quad  |
15 d = Vector3:new{x=x0+L, y=H}          --       |  0 |     1   |
16 e = Vector3:new{x=x0, y=H}            --       |    |   ___---c
17 f = Vector3:new{x=0.0, y=H}           --       a----b---
18
19 -- Define the two surfaces
20 quad0 = makePatch{p00=a, p10=b, p11=e, p01=f}
21 quad1 = makePatch{p00=b, p10=c, p11=d, p01=e, gridType="ao"}
```

2.5.2 The eval function

One of the most useful functions is the `eval` function. This function evaluates the current path, surface or volume element at the parameteric location defined by (t), (s, t), or (r, s, t) and returns a new point. The function is executed by calling the eval method on a particular geometry object as per the Lua object-oriented convention described in Section 2.1: `my_path:eval(t)`, `my_surf:eval(s,t)` or `my_vol:eval(r,s,t)`. Alternatively one may call the geometry object directly as, for example, `my_path(t)`. See following sections for how the `eval` function can be used to manipulate a geometry.

2.5.3 Subdividing a path

A common task in geometry generation is to sub-divide an existing line and to add a new point at the break. Particularly when using parameteric curves, such as Bezier curves or splines, simply defining two lines that meet at a point is not a desirable option. As illustarted in the example below, also shown in Figure 2.13, the gradients (and higher order derivatves) of the lines do not match at the interface and the overall line shape can change. A better approach is to use a single line, and to split this into two lines using `SubRangedPath` and then creating a new point at the interface.

Here are some examples of splitting a Bezier curve. The results are shown in Figure 2.13.

```
1 -- split-path-example.lua
2
3 -- Single Bezier curve of order 4.
4 b={}
5 b[1] = Vector3:new{x=0.1,  y=0.7}
6 b[2] = Vector3:new{x=0.15, y=0.8}
7 b[3] = Vector3:new{x=0.5,  y=0.9}
8 b[4] = Vector3:new{x=0.6,  y=0.7}
9 b[5] = Vector3:new{x=0.9,  y=0.7}
10 single_bez = Bezier:new{points={b[1], b[2], b[3], b[4], b[5]}}
11
12 -- offset points in y-direction for 2nd and 3rd line
13 c = {}
14 d = {}
15 for i,v in ipairs(b) do
16    c[i] = Vector3:new{x=v.x,  y=v.y-0.35} -- offset points by -0.35
17    d[i] = Vector3:new{x=v.x,  y=v.y-0.7} -- offset points by -0.7
18 end
19
20 -- create two Bezier curves
21 two_bez_0 = Bezier:new{points={c[1], c[2], c[3]}}
22 two_bez_1 = Bezier:new{points={c[3], c[4], c[5]}}
23
24 -- create split line
25 t = 0.5  -- define parameterised location of split
26 bez = Bezier:new{points={d[1], d[2], d[3], d[4], d[5]}}
27 C = bez:eval(t)
28 line_d1_C = SubRangedPath:new{underlying_path=bez, t0 = 0.0, t1 = t}
29 line_C_d5 = SubRangedPath:new{underlying_path=bez, t0 = t, t1 = 1.0}
30
31 dofile("sketch-split-path-example.lua")
```

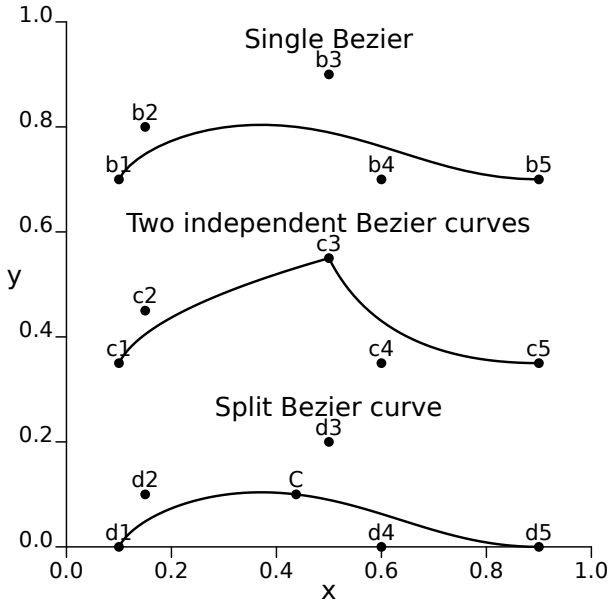


Figure 2.13: Effect of using different approaches to split a Bezier curve into two paths elements.

2.5.4 Creating a line normal to a path

When creating a refined grid that conforms to a wall, a common requirement is to have lines and grid construction points positioned normal to the path that represents the wall. Using the `eval` function it is easy to automate this process.

In the following code, point `A` is created at normalised position t=0.3. Next the tangent vector (which equals the gradient of the path) at point `A`, `gradient` is determined using the finite difference method. From this a wall normal vector, `normal` is created by swapping the x- and y-components and negating the sign of the x-component. Finally, the position of point `B` is determined by adding a scaled version of the normal vector to point `A`. The results are shown in Figure 2.14.

```
1 -- point-normal-to-wall-example.lua
2
3 -- Single Bezier curve of order 4.
4 b={}
5 b[1] = Vector3:new{x=0.1,  y=0.1}
6 b[2] = Vector3:new{x=0.15, y=0.2}
7 b[3] = Vector3:new{x=0.5,  y=0.3}
8 b[4] = Vector3:new{x=0.6,  y=0.1}
9 b[5] = Vector3:new{x=0.9,  y=0.1}
10 single_bez = Bezier:new{points={b[1], b[2], b[3], b[4], b[5]}}
11
12 -- Create point along wall
13 t = 0.3
14 A = single_bez:eval(t)
15
16 -- Calculate gradient vector by evaluating path at t +/- dt
17 dt = 0.01
18 gradient = single_bez:eval(t+dt) - single_bez:eval(t-dt)
19
20 -- create wall normal vector and normalise
21 normal = Vector3:new{x=-gradient.y,  y=gradient.x}
22 normal:normalize()
23
24 -- Create point B at distance L
25 L = 0.2
26 B = A + L*normal -- add vectors
27
28 -- Create line connecting A and B
29 mypath = Line:new{p0=A, p1=B}
30
31 dofile("sketch-point-normal-to-wall-example.lua")
```

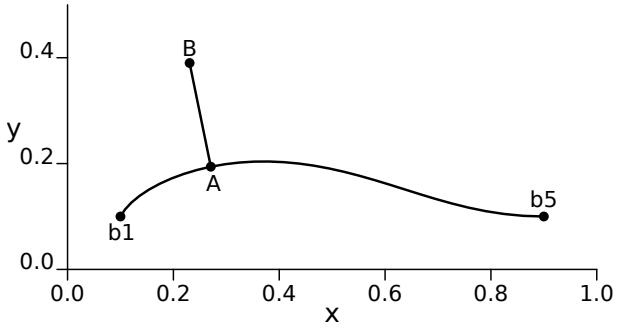


Figure 2.14: Constructing a point normal to a wall.

3

Grids

Because the flow solver expects the gas-flow and solid domains to be specified as meshes of finite-volume cells, we now need to consider the discretizion of the 2D patches and 3D volumes. Typically, we pass the surface and volume objects, along with cluster-function objects, to the constructor for the StructuredGrid class and get back a mesh of points that define the vertices of our finite-volume cells. This mesh, when combined with boundary conditions and an initial gas state (as described in the main solver user guide [2]), then defines a flow block.

As a motivational example, especially for MECH4480 students of CFD, consider the construction of a two-dimensional grid in the region around a bottle of *James Boag's Premium*. Figure 3.1 shows the final block arrangement with the bottle lying on its side. You can see the profile of the bottle in the curves from x=0 to x=0.2 metres. We model only the upper half plane, with the gas domain being the region around the bottle. Also, we'll do the modelling in stages, starting with a single block defining a limited subregion.

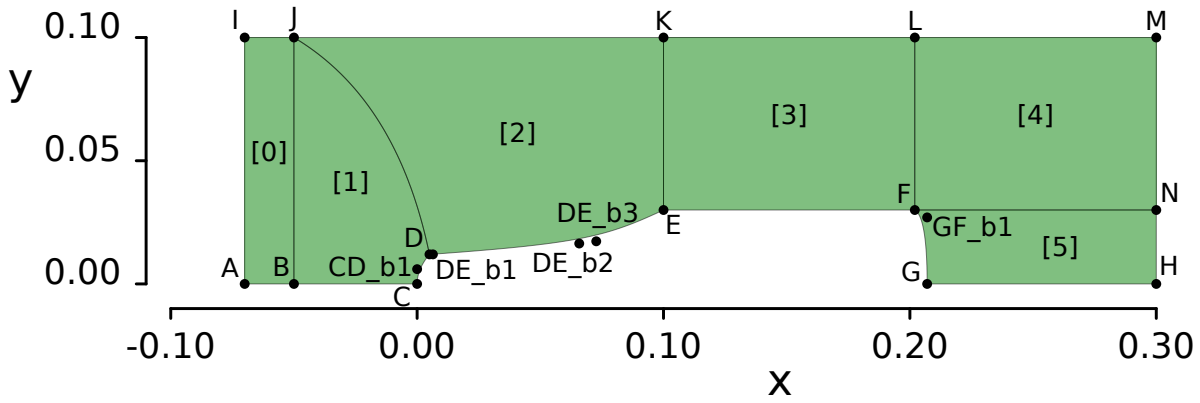


Figure 3.1: Schematic diagram of Ingo's beer bottle aligned with the x-axis. This PDF figure was generated from the SVG file with some edits to move the point and block labels to nicer positions.

3.1 Making a simple 2D grid

We start with just block [3] above the main body of the bottle and define just the 4 nodes E,F,K and L that mark the corners of our region of interest (Figure 3.2). A simple way to define the region is to make a CoonsPatch object with the four sides specified as staight-line paths. The `StructuredGrid` constructor is given this surface object, the number of vertices in each parametric direction and, possibly, the list of cluster functions along each edge. The parametric directions of the surface, r and s, are aligned with the grid index directions, i and j, respectively. We will come back to clustering the points later.

```
-- the-minimal-grid.lua

-- Create the nodes that define key points for our geometry.
E = Vector3:new{x=0.1, y=0.03}; F = Vector3:new{x=0.202, y=0.03}
K = Vector3:new{x=0.1, y=0.1}; L = Vector3:new{x=0.202, y=0.1}

patch3 = CoonsPatch:new{north=Line:new{p0=K, p1=L},
                        east=Line:new{p0=F, p1=L},
                        south=Line:new{p0=E, p1=F},
                        west=Line:new{p0=E, p1=K}}

grid3 = StructuredGrid:new{psurface=patch3, niv=21, njv=21}
grid3:write_to_vtk_file("the-minimal-grid.vtk")

dofile("sketch-minimal-grid.lua")
```

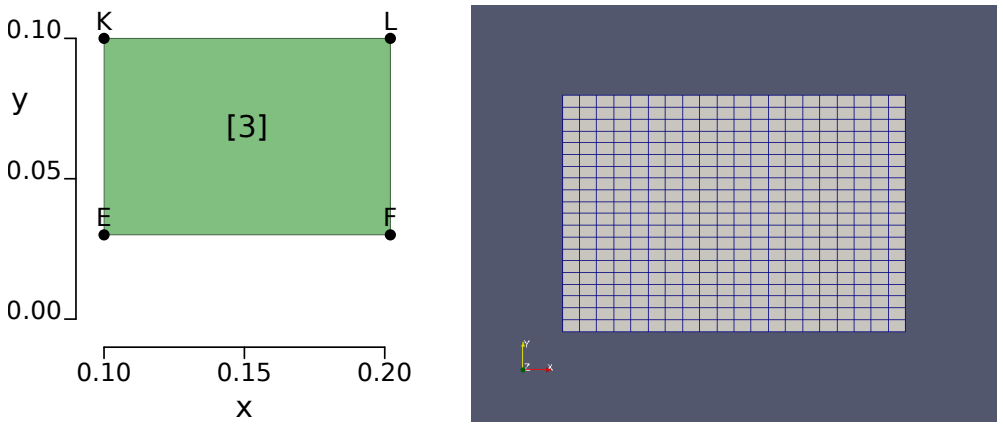


Figure 3.2: A single block for a simple subregion from the eventual model of the Ingo's beer bottle.

The script for making the SVG sketch was called on the last line of the previous script. Its content is a bit ugly, especially for drawing the axes, but it allows us to make a drawing customized to the situation.

```
-- sketch-minimal-grid.lua
-- Called by the user input script to make a sketch of the flow domain.
```

```
-- PJ, 2016-09-28
sk = Sketch:new{renderer="svg", projection="xyortho", canvas_mm={0.0,0.0,120.0,120.0}}
sk:set{viewport={0.0,-0.1,0.3,0.2}}
sk:start{file_name="the-minimal-grid.svg"}

-- for flow domain
sk:set{line_width=0.1, fill_colour="green"}
sk:render{surf=patch3}
sk:text{point=0.25*(E+F+K+L), text="[3]", font_size=12}

-- labelled points of interest
sk:dotlabel{point=E, label="E"}; sk:dotlabel{point=F, label="F"}
sk:dotlabel{point=K, label="K"}; sk:dotlabel{point=L, label="L"}

-- axes
sk:set{line_width=0.3} -- for drawing rules
sk:rule{direction="x", vmin=0.1, vmax=0.2, vtic=0.05,
       anchor_point=Vector3:new{x=0.1,y=-0.01},
       tic_mark_size=0.004, number_format="%.2f",
       text_offset=0.012, font_size=10}
sk:text{point=Vector3:new{x=0.15,y=-0.035}, text="x", font_size=12}
sk:rule{direction="y", vmin=0.0, vmax=0.1, vtic=0.05,
       anchor_point=Vector3:new{x=0.09,y=0},
       tic_mark_size=0.004, number_format="%.2f",
       text_offset=0.005, font_size=10}
sk:text{point=Vector3:new{x=0.07,y=0.075}, text="y", font_size=12}

sk:finish{}
```

3.2 StructuredGrid Class

The constructor for `StructuredGrid` class accepts any of the following items:

- `path`: a Path object along which a number of discrete points will be defined to represent a one-dimensional grid.
- `psurface`: a ParametricSurface object over which a regular mesh of points will be defined to represent the vertices of a two-dimensional grid.
- `pvolume` a ParametricVolume object through which a 3D mesh of points will be defined.

The constructor first looks for a `path` item and, if not found, tries the `psurface` item. If that is not found, it finally tries for a `pvolume` item. As soon as it has found a valid geometric item, it proceeds, ignoring the other options. The other essential items for defining the grid are the number of vertices in each index direction, which are named `niv`, `njv` and `nkv`. The index directions i, j and k for the grid correspond to the parametric coordinate directions r, s and t respectively. Remember that the number of finite-volume cells in each index direction is one less than the number of vertices in that direction. We will often think of our the size of our grids in terms of numbers of cells and so will specify the number for the vertices in the call to this constructor as $n + 1$, where n is the number of cells that we want.

With no further information, the grid generator will uniformly distribute the vertices in each parametric direction. This may or may not correspond to a uniform distribution in x,y,z-space, depending on the definition of the supplied geometric object. To get the uniformity in the distribution of the mesh vertices, you may need to use an ArcLengthParameterizedPath (see page 13), for example, when defining the edges of your surface patch.

Sometimes you really do want a non-uniform distribution of points along one or more of the index directions. The distribution of vertices along each edge of the geometric object can be specified via a cluster-function object. These are `UnivariateFunction` objects of the following flavours:

- `LinearFunction:new{t0=`t_0`, t1=`t_1`}` where $t_{new} = t_0 \times (1 - t_{old}) + t_1 \times t_{old}$. Default values for t_0 and t_1 are 0.0 and 1.0, respectively.

- `RobertsFunction:new{end0=`$false$`, end1=`$false$`, beta=`β`}` where the `end0`, `end1` integer flags indicate which end (possibly both) we wish to cluster toward. The value of `beta` > 1.0 specifies the strength of the clustering, with the clustering being stronger for smaller values of `beta`. For example, a value of 1.3 would be relatively weak clustering while a value of 1.01 is quite strong clustering. Set the `end0` and `end1` items to be true to cluster toward each end of the parameter space (i.e. $t = 0$ and $t = 1$), respectively.

- `LuaFnClustering:new{luaFnName='myClusterFn'}` where `luaFnName` is the name of a user-defined function that specifies the clustering. The user-defined function should appear before this cluster-function object is called. The user's function should accept a single float argument and return a single float value. The role of the user-defined function is to provide a mapping from a uniform distribution of parameter-space $t = 0 \rightarrow 1$ to some non-uniform distribution, $s = 0 \rightarrow 1$. An example of a user-defined function that gives a parabolic distribution is:

```
function myParabola(t)
   s = t*t
   return s
end
```

  This could be used as a cluster function by creating a `LuaFnClustering` object:

```
myClustering = LuaFnClustering:new{luaFnName='myParabola'}
```

To specify a single cluster function for a one-dimensional mesh, provide a `cf` item in the table given to the StructuredGrid constructor. To specify the sets of cluster function objects needed for 2D and 3D, grids, provide a table of named objects as the `cfList` item. In 2D, the names of the edges are `north`, `east`, `south` and `west`. In 3D, the 12 edges have the names `edge01`, `edge12`, `edge32`, `edge03`, `edge45`,

`edge56`, `edge76`, `edge47`, `edge04`, `edge15`, `26` and `edge37`. Note that the 3D names use the vertex labels (see Figure 2.10) to indicate which way the r, s or t parameter varies. Look at your debugging cube from Appendix A to get a good idea of the arrangement. (You did cut out and stick the edges of your cube together, didn't you?) You don't need to specify a cluster function for every edge. If you don't specify a function for a particular edge, it will get a default linear function that results in a uniformly-distributed set of points.

So, putting all that together for a two-dimensional grid, we have the constructor signature:

`StructuredGrid:new{psurface=`$myPatch$`, niv=`n_i`, njv=`n_j`,`
`  cfList={north=`cf_N`, south=`cf_S`, west=`cf_W`, east=`cf_E`} }`

Omitting the longer cluster-function list, the constructor signature for a parametric volume, will look as simple as:

`StructuredGrid:new{pvolume=`$myVolume$`, niv=`n_i`, njv=`n_j`, nkv=`n_k`}`

3.2.1 StructuredGrid Methods

Once you have a StructuredGrid object, $sgrid$, there are a number of methods that can be called.

- $sgrid$`:get_niv()` returns the number of vertices in the i-index direction. The empty parentheses indicate that no argument needs to be supplied.
- $sgrid$`:get_njv()` returns the number of vertices in the j-index direction.
- $sgrid$`:get_nkv()` returns the number of vertices in the k-index direction. For a 2D grid, this will be 1.
- $sgrid$`:get_vtx(`i, j, k`)` returns the Vector3 object giving the location of the vertex in x,y,z-space. If j and/or k are not supplied, values of zero are assumed. Index values for each direction start and 0 and range up to (but less than) the number of vertices in that direction. This is at odds with the usual Lua convention for starting at 1 but it aligns with the D-code implementation of the StructuredGrid class.
- $sgrid$`:subgrid(`$i_0, n_i, j_0, n_j, k_0, n_k$`)` returns a StructuredGrid object that has been constructed as a subsection of the original grid. In each direction the i_0 specifies the first vertex of the subgrid and n_i counts out the further number of vertices to end of the subsection. n_i is effectively the number of cells in i-direction of the subgrid. If values for k_0 and n_k are not supplied, values of 0 and 1, respectively, are assumed. Same for j_0 and n_j.
- $sgrid$`:write_to_vtk_file(`$fileName$`)` writes the grid, in classic VTK format, to the specified file. This format is a simple ASCII text format that may be viewed with any text editor or displayed with a visualization program like Paraview.
- $sgrid$`:write_to_gzip_file(`$fileName$`)` writes the grid to the native format used by Eilmer4. It is a essentially a gzipped text file with a format defined by

the code implemented in the D-code file `src/geom/sgrid.d`. It is very similar to the classic VTK format.

- *sgrid*`:joinGrid(`*otherGrid*, *joinLocation*`)` will append *otherGrid* to the *joinLocation* boundary of *sgrid*. The join location may be specified as `east`, `imax`, `north`, or `jmax`.
- *sgrid*`:find_nearest_cell_centre(`x, y, z`)` returns the single index of the cell whose centre lies closest to the x,y,z point and the distance from that point to the position of the cell centre. Arguments not supplied default to zero.

3.2.2 Importing a Gridpro Grid

The unbound function `importGridproGrid(`*fileName*, *scale*`)`, will read a complete Gridpro grid file and return a list (i.e. a Lua table) of StructuredGrid objects. A complete Gridpro grid file may contain several blocks, hence the return of a Lua table. Care should be taken with Gridpro grids built from CAD geometries, which are typically dimensioned in millimetres. For such a file, the required value for *scale* would be 0.001, to convert coordinates to metres. The default value for *scale* is 1.

3.3 UnstructuredGrid Class

This is a new class for Eilmer4 and we haven't had a lot of experience using it. Presently, the options for constructing and UnstructuredGrid object are:

- construct from a StructuredGrid object,
- import the grid in SU2 format, maybe from a file written by Pointwise,
- use Heather's paver, which is a work in progress.

If you rummage through the example set that accompanies the Eilmer source code, you will find examples of building and importing unstructured grids.

3.4 Building a multiblock grid

When making a flow domain that is reasonably complicated, it's probably best to build a collection of blocks where each block is roughly a quadrilateral, but with the bounding paths fitted to the curves of the object to be modelled. Figure 3.3 shows the resulting grid, after dividing the full gas-flow region into 6 blocks.

```
1 -- the-plain-bottle.lua
2
3 -- Create the nodes that define key points for our geometry.
4 A = Vector3:new{x=-0.07, y=0.0}; B = Vector3:new{x=-0.05, y=0.0}
5 C = Vector3:new{x=0.0, y=0.0}; D = Vector3:new{x=0.005, y=0.012}
6 E = Vector3:new{x=0.1, y=0.03}; F = Vector3:new{x=0.202, y=0.03}
7 G = Vector3:new{x=0.207, y=0.0}; H = Vector3:new{x=0.3, y=0.0}
8 I = Vector3:new{x=-0.07, y=0.1}; J = Vector3:new{x=-0.05, y=0.1}
9 K = Vector3:new{x=0.1, y=0.1}; L = Vector3:new{x=0.202, y=0.1}
10 M = Vector3:new{x=0.3, y=0.1}; N = Vector3:new{x=0.3, y=0.03}
```

```
11
12 -- Some interior Bezier control points
13 CD_b1 = Vector3:new{x=0.0, y=0.006}
14 DJ_b1 = Vector3:new{x=-0.008, y=0.075}
15 GF_b1 = Vector3:new{x=0.207, y=0.027}
16 DE_b1 = Vector3:new{x=0.0064, y=0.012}
17 DE_b2 = Vector3:new{x=0.0658, y=0.0164}
18 DE_b3 = Vector3:new{x=0.0727, y=0.0173}
19
20 -- Now, we join our nodes to create lines that will be used to form our blocks.
21 -- lower boundary along x-axis
22 AB = Line:new{p0=A, p1=B}; BC = Line:new{p0=B, p1=C}
23 GH = Line:new{p0=G, p1=H}
24 CD = Bezier:new{points={C, CD_b1, D}} -- top of bottle
25 DE = Bezier:new{points={D, DE_b1, DE_b2, DE_b3, E}} -- neck of bottle
26 EF = Line:new{p0=E, p1=F} -- side of bottle
27 -- bottom of bottle
28 GF = ArcLengthParameterizedPath:new{
29    underlying_path=Bezier:new{points={G, GF_b1, F}}}
30 -- Upper boundary of domain
31 IJ = Line:new{p0=I, p1=J}; JK = Line:new{p0=J, p1=K}
32 KL = Line:new{p0=K, p1=L}; LM = Line:new{p0=L, p1=M}
33 -- Lines to divide the gas flow domain into blocks.
34 AI = Line:new{p0=A, p1=I}; BJ = Line:new{p0=B, p1=J}
35 DJ = Bezier:new{points={D, DJ_b1, J}}
36 JD = ReversedPath:new{underlying_path=DJ}; EK = Line:new{p0=E, p1=K}
37 FL = Line:new{p0=F, p1=L};
38 NM = Line:new{p0=N, p1=M}; HN = Line:new{p0=H, p1=N}
39 FN = Line:new{p0=F, p1=N}
40
41 -- Define the blocks, boundary conditions and set the discretisation.
42 n0 = 10; n1 = 4; n2 = 20; n3 = 20; n4 = 20; n5 = 12; n6 = 8
43
44 patch = {}
45 patch[0] = CoonsPatch:new{north=IJ, east=BJ, south=AB, west=AI}
46 patch[1] = CoonsPatch:new{north=JD, east=CD, south=BC, west=BJ}
47 patch[2] = CoonsPatch:new{north=JK, east=EK, south=DE, west=DJ}
48 patch[3] = CoonsPatch:new{north=KL, east=FL, south=EF, west=EK}
49 patch[4] = CoonsPatch:new{north=LM, east=NM, south=FN, west=FL}
50 patch[5] = CoonsPatch:new{north=FN, east=HN, south=GH, west=GF}
51
52 grid = {}
53 grid[0] = StructuredGrid:new{psurface=patch[0], niv=n1+1, njv=n0+1}
54 grid[1] = StructuredGrid:new{psurface=patch[1], niv=n2+1, njv=n0+1}
55 grid[2] = StructuredGrid:new{psurface=patch[2], niv=n3+1, njv=n2+1}
56 grid[3] = StructuredGrid:new{psurface=patch[3], niv=n4+1, njv=n2+1}
57 grid[4] = StructuredGrid:new{psurface=patch[4], niv=n5+1, njv=n2+1}
58 grid[5] = StructuredGrid:new{psurface=patch[5], niv=n5+1, njv=n6+1}
59
60 for ib = 0, 5 do
61    fileName = string.format("the-plain-bottle-blk-%d.vtk", ib)
62    grid[ib]:write_to_vtk_file(fileName)
63 end
64 dofile("sketch-plain-bottle.lua")
```

Each of the blocks is generated independently of the others. It is your responsibil-

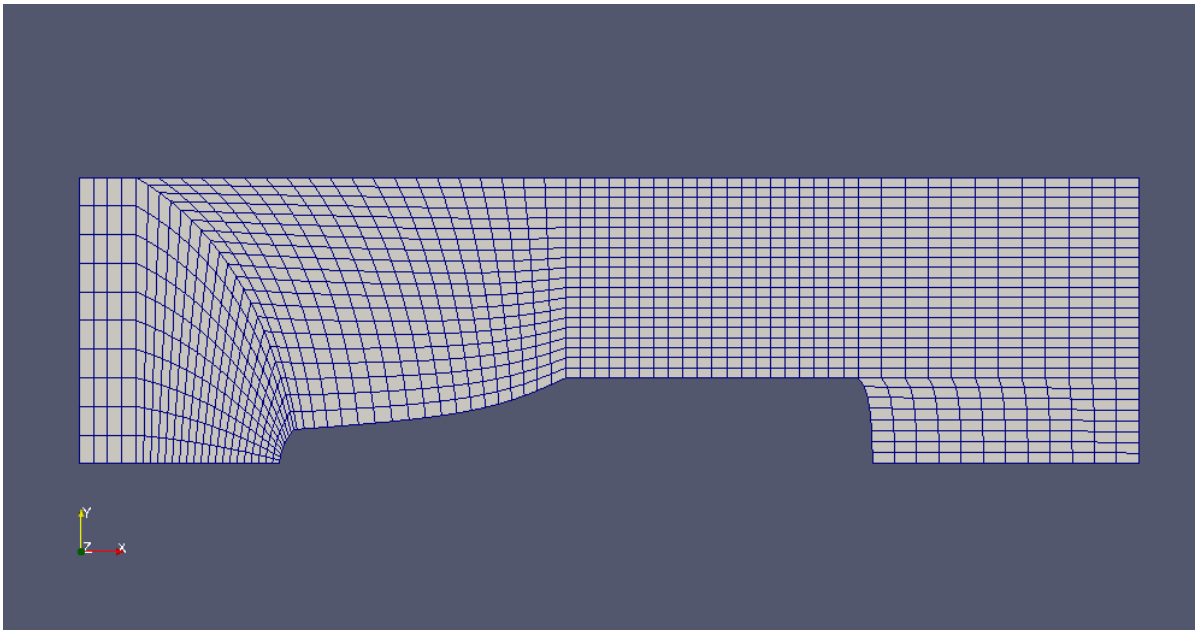


Figure 3.3: A multiple-block model of the region around Ingo's beer bottle.

ity to ensure that the defining edges that are common to pairs of blocks are consistent and that the cell-discretization along each of these edges is consistent with the corresponding discretization of any adjacent edge of another block. The first constraint is easy to meet by defining each edge once only and reusing that path in the definition of different blocks. Sometimes, the orientation of a pair of blocks and the particular directions of the paths within each block means that one defining edge needs to be in the opposite sense to the original. In this case the `ReversedPath` class may be useful. For an example of this, note that the script actually constructs such a path for `JD` using the path `DJ` on line 36. Tha path `JD` is oriented so that it can serve as a `north` boundary for patch 1 whereas the path `DJ` is oriented such that it can be used as a `west` boundary for patch 2.

3.4.1 Improving the grid with clustering

We can now tweak the grid and improve the distribution and shape of the cells by adjusting the clustering of the points along each of the block edges. See Figure 3.4 for the result of the following script. The particular values used for the strength of the clustering are ad-hoc and some trial and error has been used to get these particular values.

Again, the distribution of points along each edge of each block is computed independently, so it is the responsibility of the user to ensure that the cells along the corresponding edges of adjoining blocks are aligned. This will require the use of matching clustering functions on these edges.

```
1 -- the-clustered-bottle.lua
2
3 -- Create the nodes that define key points for our geometry.
```

```
A = Vector3:new{x=-0.07, y=0.0}; B = Vector3:new{x=-0.05, y=0.0}
C = Vector3:new{x=0.0, y=0.0}; D = Vector3:new{x=0.005, y=0.012}
E = Vector3:new{x=0.1, y=0.03}; F = Vector3:new{x=0.202, y=0.03}
G = Vector3:new{x=0.207, y=0.0}; H = Vector3:new{x=0.3, y=0.0}
I = Vector3:new{x=-0.07, y=0.1}; J = Vector3:new{x=-0.05, y=0.1}
K = Vector3:new{x=0.1, y=0.1}; L = Vector3:new{x=0.202, y=0.1}
M = Vector3:new{x=0.3, y=0.1}; N = Vector3:new{x=0.3, y=0.03}

-- Some interior Bezier control points
CD_b1 = Vector3:new{x=0.0, y=0.006}
DJ_b1 = Vector3:new{x=-0.008, y=0.075}
GF_b1 = Vector3:new{x=0.207, y=0.027}
DE_b1 = Vector3:new{x=0.0064, y=0.012}
DE_b2 = Vector3:new{x=0.0658, y=0.0164}
DE_b3 = Vector3:new{x=0.0727, y=0.0173}

-- Now, we join our nodes to create lines that will be used to form our blocks.
-- lower boundary along x-axis
AB = Line:new{p0=A, p1=B}; BC = Line:new{p0=B, p1=C}
GH = Line:new{p0=G, p1=H}
CD = Bezier:new{points={C, CD_b1, D}} -- top of bottle
DE = Bezier:new{points={D, DE_b1, DE_b2, DE_b3, E}} -- neck of bottle
EF = Line:new{p0=E, p1=F} -- side of bottle
-- bottom of bottle
GF = ArcLengthParameterizedPath:new{
   underlying_path=Bezier:new{points={G, GF_b1, F}}}
-- Upper boundary of domain
IJ = Line:new{p0=I, p1=J}; JK = Line:new{p0=J, p1=K}
KL = Line:new{p0=K, p1=L}; LM = Line:new{p0=L, p1=M}
-- Lines to divide the gas flow domain into blocks.
AI = Line:new{p0=A, p1=I}; BJ = Line:new{p0=B, p1=J}
DJ = Bezier:new{points={D, DJ_b1, J}}
JD = ReversedPath:new{underlying_path=DJ}; EK = Line:new{p0=E, p1=K}
FL = Line:new{p0=F, p1=L};
NM = Line:new{p0=N, p1=M}; HN = Line:new{p0=H, p1=N}
FN = Line:new{p0=F, p1=N}

-- Define the blocks, boundary conditions and set the discretisation.
n0 = 10; n1 = 4; n2 = 20; n3 = 20; n4 = 20; n5 = 12; n6 = 8

patch = {}
patch[0] = CoonsPatch:new{north=IJ, east=BJ, south=AB, west=AI}
patch[1] = CoonsPatch:new{north=JD, east=CD, south=BC, west=BJ}
patch[2] = CoonsPatch:new{north=JK, east=EK, south=DE, west=DJ}
patch[3] = CoonsPatch:new{north=KL, east=FL, south=EF, west=EK}
patch[4] = CoonsPatch:new{north=LM, east=NM, south=FN, west=FL}
patch[5] = CoonsPatch:new{north=FN, east=HN, south=GH, west=GF}

rcfL = RobertsFunction:new{end0=true, end1=false, beta=1.2}
rcfR = RobertsFunction:new{end0=false, end1=true, beta=1.2}

grid = {}
grid[0] = StructuredGrid:new{psurface=patch[0], niv=n1+1, njv=n0+1}
cfList = {north=RobertsFunction:new{end0=false, end1=true, beta=1.1}, south=rcfR}
grid[1] = StructuredGrid:new{psurface=patch[1], niv=n2+1, njv=n0+1,
                             cfList=cfList}
cfList = {north=rcfR, west=RobertsFunction:new{end0=true, end1=false, beta=1.1}}
```

```
grid[2] = StructuredGrid:new{psurface=patch[2], niv=n3+1, njv=n2+1,
                             cfList=cfList}
grid[3] = StructuredGrid:new{psurface=patch[3], niv=n4+1, njv=n2+1}
grid[4] = StructuredGrid:new{psurface=patch[4], niv=n5+1, njv=n2+1,
                             cfList={north=rcfL, south=rcfL}}
grid[5] = StructuredGrid:new{psurface=patch[5], niv=n5+1, njv=n6+1,
                             cfList={north=rcfL, south=rcfL}}

for ib = 0, 5 do
   fileName = string.format("the-clustered-bottle-blk-%d.vtk", ib)
   grid[ib]:write_to_vtk_file(fileName)
end
```

Further improvement of the grid can be made by introducing a layer of blocks around the bottle surface, so that the cells near the surface can be made always nearly orthogonal and much more finely clustered toward the surface. The extra blocks add to the complexity of the input script but provide some decoupling with respect to cell number along block edges and allow the fine clustering of cells toward the bottle surface without greatly increasing the cell refinement in other parts of the gas-flow region. Such a grid would be suited to simulations of viscous flows.

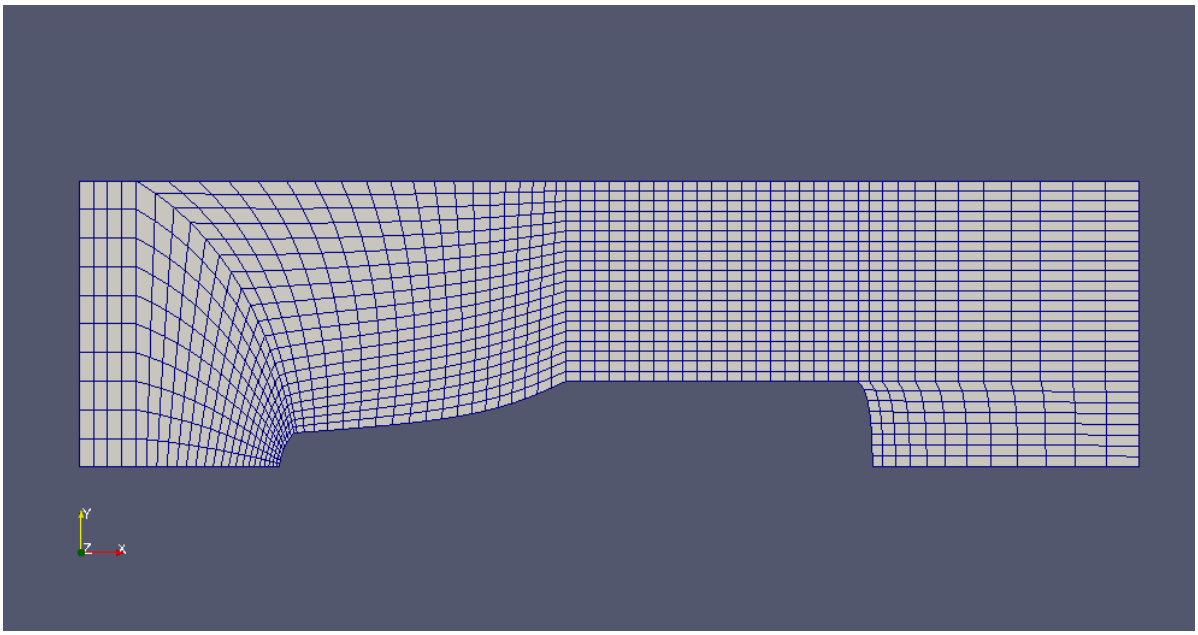


Figure 3.4: An improved multiple-block grid around Ingo's beer bottle.

References

[1] Peter Jacobs and Rowan Gollan. Implementation of a compressible-flow simulation code in the D programming language. In *Advances of Computational Mechanics in Australia*, volume 846 of *Applied Mechanics and Materials*, pages 54–60. Trans Tech Publications, 9 2016.

[2] Peter A. Jacobs and Rowan J. Gollan. The Eilmer 4.0 flow simulation program: Guide to the transient flow solver, including some examples to get you started. School of Mechanical and Mining Engineering Technical Report 2017/26, The University of Queensland, Brisbane, Australia, February 2018.

[3] Roberto Ierusalimschy. *Programming in Lua*. Lua.org, 2006.

[4] M. J. Zucrow and J. D. Hoffman. *Gas Dynamics Volume 1*. John Wiley & Sons, New York, 1976.

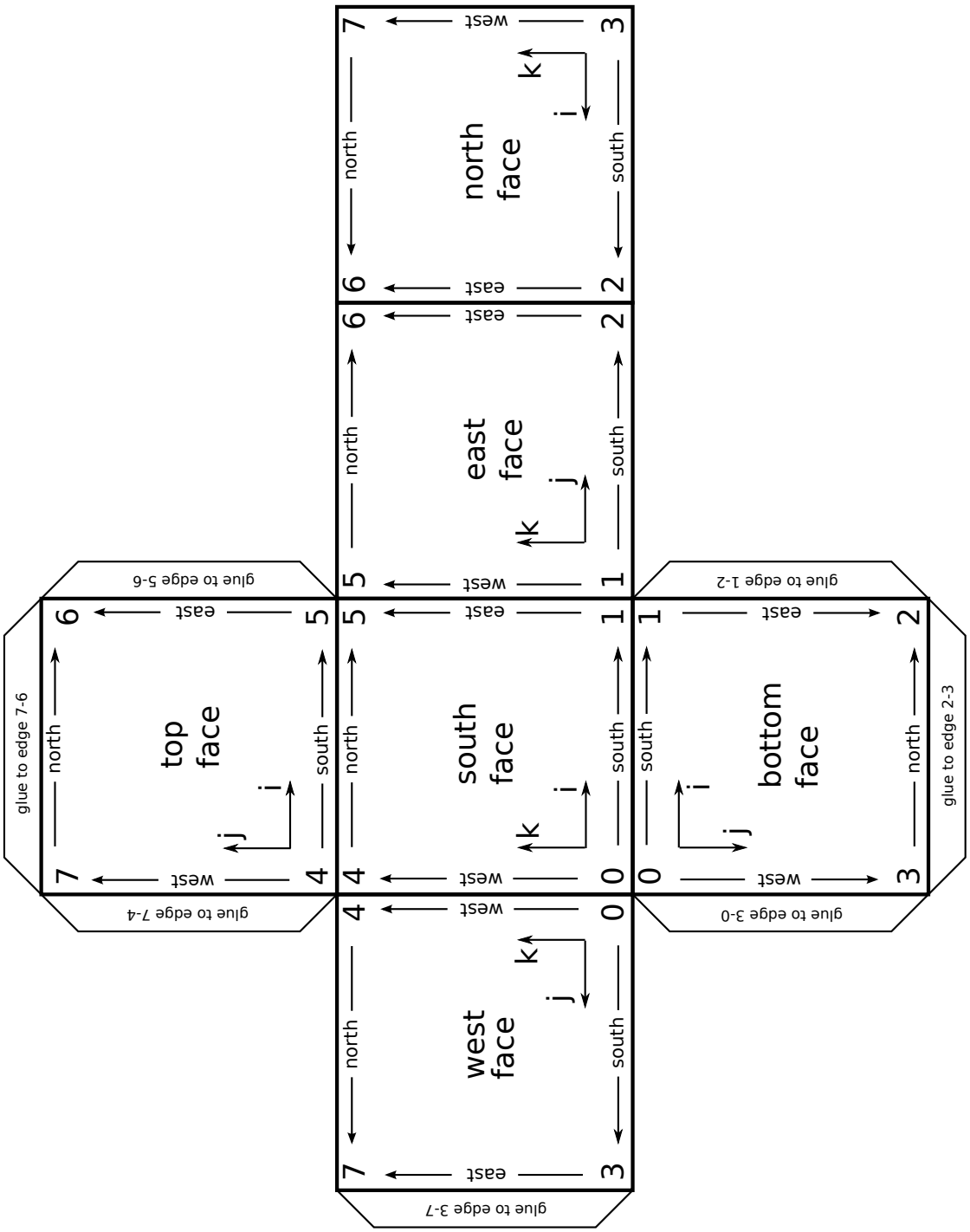
[5] P. M. Knupp. A robust elliptic grid generator. *Journal of Computational Physics*, 100(2):409–418, 1992.

Index

A

Make your own debugging cube

Cut out the development on the reverse of this page, fold along all of the edges and stick the your own cube together. A pair of cubes is very handy for sorting out the specification of connections between structured-grid blocks.


north face
west
east
north
south
i
k
east face
west
east
north
south
j
k
south face
west
east
north
south
k
i
bottom face
west
east
north
south
i
j
top face
west
east
north
south
j
i
west face
west
east
north
south
k
j
glue to edge 3-7
glue to edge 5-6
glue to edge 1-2
glue to edge 7-4
glue to edge 3-0
glue to edge 7-6
glue to edge 2-3