TexGen Scripting Guide

By
Louise P. Brown
Frank Gommer
Andrew C. Long
Mikhail Y. Matveev
Xuesen Zeng
Shibo Yan

Contents

Preface

TexGen is software developed at the University of Nottingham for geometric modelling of textiles and textile composites. It provides a tool for generating realistic geometric

representations of textiles as a pre-processor for simulations for prediction of a range of material properties such as mechanical properties, permeability and thermal conductivity.
It is open source software; executables are available for download from Sourceforge: http://texgen.sourceforge.net and source code is available on GitHub https://github.com/louisepb/TexGen. There is a graphical user interface for easy generation of standard textile weaves but there is greater flexibility, and more complex textiles can be created by making use of the Python scripting interface.
This document describes the use of the TexGen Python application programming interface (API). The API documentation[1] provides definitions of all of the functions available for use in Python scripts for generating textile models and this document gives guidance on their use for creation of textile models. It applies to TexGen v3.8.0 and higher.
The example scripts used in the document are all available on GitHub https://github.com/louisepb/TexGenScriptingGuide

1. TexGen Structure

TexGen is split into several modules as shown in Figure 1.

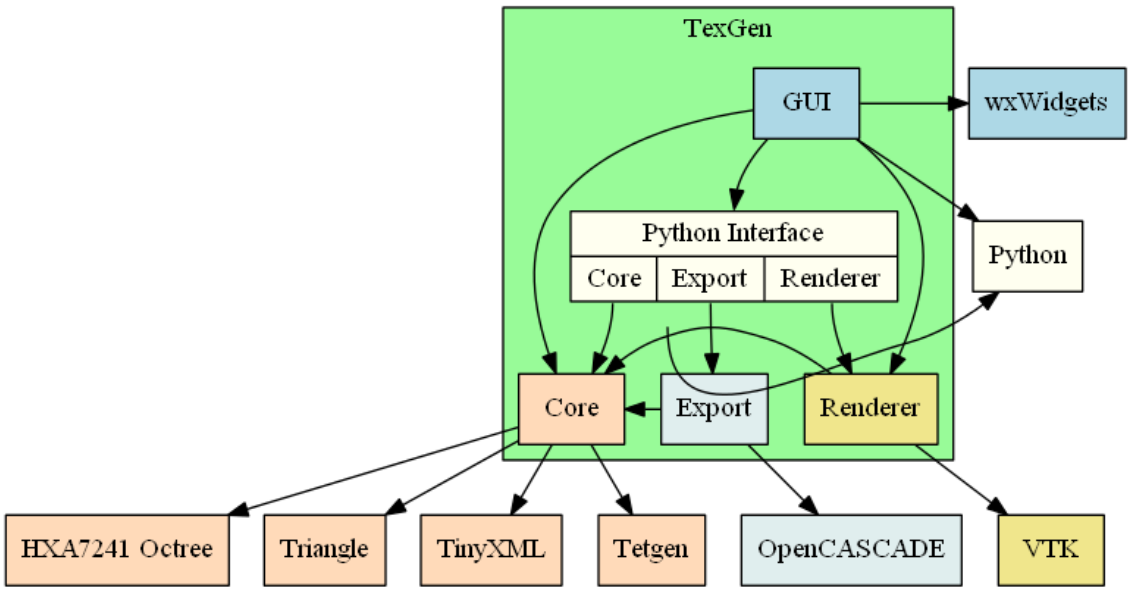


Figure 1: TexGen modules

This guide mainly documents the Core, Renderer and Export modules. TexGen is written in C++ and the Python interface is automatically generated by the Simplified Wrapper and Interface Generator (SWIG)[2] allowing Python to call functions within the Core, Renderer and Export modules.
The classes found within the core module and how they relate to each other are shown in Figure 1. An understanding of this structure and how it relates to the modelling theory[1, 2] is essential in order to be able to make best use of the full functionality of the TexGen API for the creation of complex textile models.

[1] http://texgen.sourceforge.net/api
[2] http://www.swig.org/

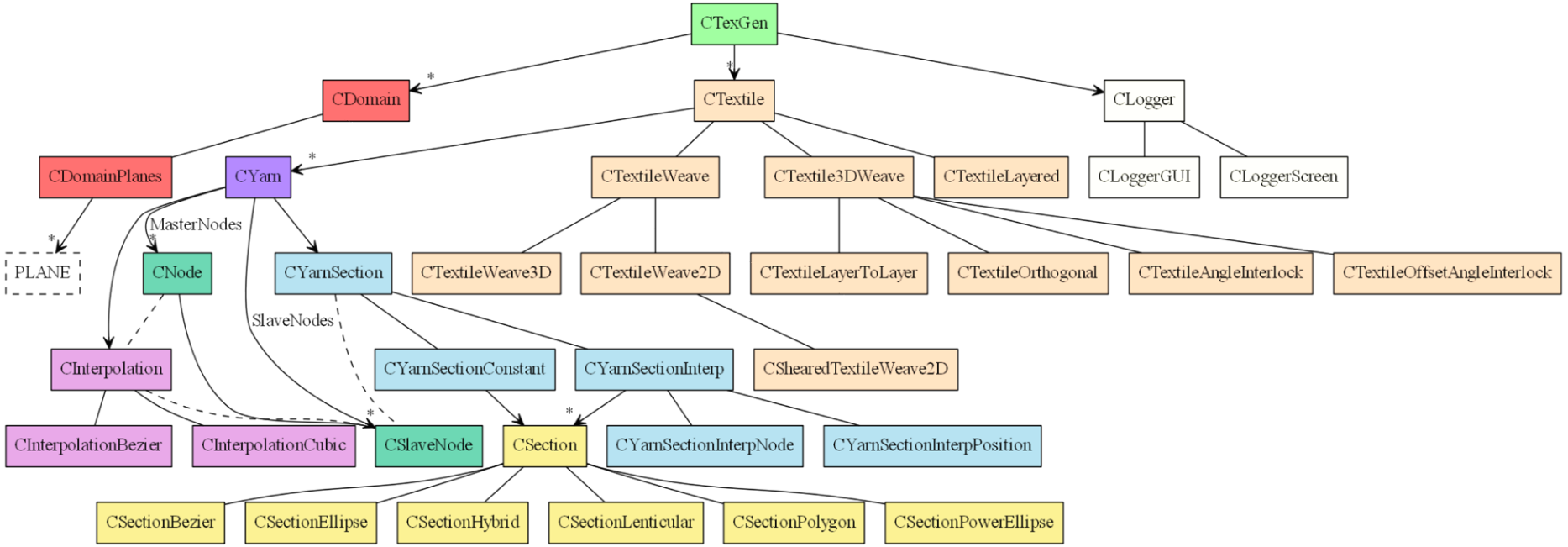


Figure 2: Core module structure

1.1 Python Programming

This guide is not intended to be a tutorial on using the Python language. A selection of online Python tutorials can be found here:
http://texgen.sourceforge.net/index.php/Scripting_Guide

1.1.1 Scripts using Python3

From TexGen version 3.13.0 the software is built with Python3. This version uses the runpy module to execute scripts from the TexGen GUI and it is therefore necessary to import the TexGen core library at the start of the script:

```
from TexGen.Core import *
```

1.1.2 Python STL Templates

The API reference shows function parameters as they would be for the C++ code in which TexGen is written. In order to access the Python versions of functions where C++ Standard Template Library (STL) functions have been used a set of templates have been defined as listed below.

```
template(StringVector) vector<string>;
template(PlaneVector) vector<TexGen::PLANE>;
template(XYZPair) pair<TexGen::XYZ, TexGen::XYZ>;
template(YarnList) list<TexGen::CYarn>;
template(YarnVector) vector<TexGen::CYarn>;
template(YarnVectorPtr) vector<TexGen::CYarn*>;
template(TextileVector) vector<TexGen::CTextile*>;
template(TextileMap) map<string, TexGen::CTextile*>;
template(NodeVector) vector<TexGen::CNode>;
template(SlaveNodeVector) vector<TexGen::CSlaveNode>;
template(XYZVector) vector<TexGen::XYZ>;
template(XYVector) vector<TexGen::XY>;
template(DoubleVector) vector<double>;
template(IntPair) pair<int, int>;
template(IntPairVector) vector<pair<int, int> >;
template(PointInfoVector) vector<TexGen::POINT_INFO>;
template(DoubleXYZPair) pair<double, TexGen::XYZ>;
template(DoubleXYZPairVector) vector< pair<double, TexGen::XYZ> >;
template(BoolPair) pair<bool, bool>;
template(IntList) list<int>;
template(IntVector) vector<int>;
template(BoolVector) vector<bool>;
template(ElementDataList) list<TexGen::MESHER_ELEMENT_DATA>;
template(MeshDataVector) vector<TexGen::CMeshDataBase*>;
template(LinearTransformationVector) vector<CLinearTransformation>;
template(FloatVector) vector<float>;
template(MeshVector) vector<TexGen::CMesh>;
```

For example, a string vector would be declared as StringVector() in a Python script and used in place of a vector<string> shown in the C++ version of the function declaration.

1.1.3 For-Loops

Care should be taken when using *for* loops to access and change elements in a list of objects. For example in the case of a list of yarns, *yarns* = [*ListElements*], the following code will create a copy, *yarn*, of each list element, manipulate the copy and not change the original members of the *yarns* list.

```
for yarn in yarns:
        do something with yarn
```

If the list elements need to be manipulated then they should be accessed directly using an index into the list.

```
for i in range(len(yarns)):
        do something with yarns[i]
```

Any changes made to *yarns*[i], will be made to the actual element in the list.

2. Creating a Textile Model

2.1. Create Textile

The structure in Figure 2 gives an indication of the order in which a textile model is built up using a script. The yarns which make up the textile are all created within a CTextile object. This class is the most generic and is the base class for a number of more specialised classes (see Chapter 4). These more specialised classes automatically create yarn patterns and repeats which can easily be manipulated. The use of the CTextile class requires complete specification of the yarns directly by the programmer and gives maximum control of the textile specification.

An object using the CTextile class, or one of the classes which inherit from it, is therefore the first thing to be created in a script.

```
Textile = CTextile( )
```

2.2. Create Yarns

A textile structure is made up of one or more yarns created using the CYarn class, which are then added to the CTextile object.

A single yarn is created using the CYarn class:

```
Yarn = CYarn( )
```

To create multiple yarns a list can be created using [ ]:

```
Yarns = [ CYarn( ), CYarn( ), … ]
```

This creates one or more yarn objects which contain data which specify a set of master nodes (Section 2.3. Create Master Nodes), an interpolation function which specifies the path which joins the master nodes (Section 3), the cross-sections along the length of the yarn (Section 5) and yarn properties. If the nodes specified form one section of a repeating yarn pattern then a repeat vector may also be specified.

2.2.1 Set Resolution

The yarn resolution defines how fine a mesh is generated to represent the yarn. There are two versions of this function

```
Yarn.SetResolution( NumSectionPoints )
Yarn.SetResolution( NumSlaveNodes, NumSectionPoints )
```

Where just one parameter is specified this defines the number of points around the yarn cross-section as shown in Figure 3a and b. From this the number of slave nodes along the yarn is calculated automatically such that the distance between the nodes is equal to the average distance between section points.
Alternately the number of slave nodes along the yarn can be specified directly using the second function with two arguments (Figure 3c).

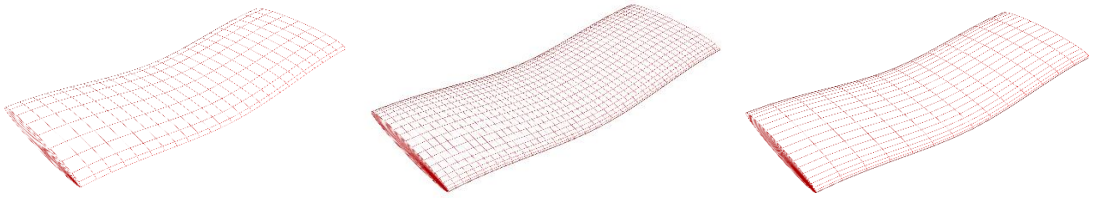


Figure 3: a) SetResolution(20) b) SetResolution(40) c) SetResolution(15,40)

2.3. Create Master Nodes

Nodes are defined as points in three dimensional space. In TexGen three-dimensional points are specified using the XYZ structure for which the Cartesian *x*, *y* and *z* coordinates (x_{pos}, y_{pos} z_{pos}) are specified:

```
Point = XYZ( xpos, ypos, zpos )
```

Individual components of the point can be accessed using:

```
posx = Point.x
posy = Point.y
posz = Point.z
```

Nodes are specified using the CNode class. This class includes Cartesian coordinates, x_i, y_i, z_i , the local yarn orientation in the form of tangents to the global coordinate system, an 'up' vector which is not parallel to the tangent vector and is used to find the normal plane which orientates the cross-section and an angle used where the cross-section is not normal to the yarn tangent (see Section 0). Using the default orientations (node tangent in the *x*-*y* plane), a node, N_i, can be specified by:

```
Ni = CNode( XYZ(xi, yi, zi) )
```

Yarn master nodes are created by adding nodes to the yarns using

```
Yarn.AddNode( Ni )
```

or

```
Yarns[i].AddNode( Ni )
```

to add to a specific yarn where a list of yarns has been specified.
The nodes are stored sequentially; the order in which the nodes are added to a yarn will specify the yarn path.

2.4. Add Yarns to Textile

The yarn(s) must then be added to the CTextile object which should have been created earlier in the script.

```
textile.AddYarn( Yarns )
```

The default yarn shape is a circular cross-section with diameter 1. This cross-sectional shape can be changed as desired (Section 55).

2.5. Add Textile to Database

If the script is to be run from within the TexGen GUI then the textiles are stored in a list (in the CTexGen object). The textile is added to the list using

```
AddTextile( "Name", textile )
```

If the textile name, *Name*, is omitted then a default name will then be assigned. If a textile with the name selected already exists, an error message will be generated.

Example: Create a single yarn by specifying nodal positions and using the default shape, node tangent assignments and yarn path interpolation.

```
# SingleYarn.py

#Create an object of the CTextile class
textile = CTextile( )

#Create a single yarn object:
yarn = CYarn( )

#Add 3 nodes to the yarn, specifying the coordinate positions only:
yarn.AddNode( CNode( XYZ(0, 0.0, 0)) )
yarn.AddNode( CNode( XYZ(1, 3.5, 0)) )
yarn.AddNode( CNode( XYZ(0, 7.0, 0)) )

# Add the yarn to the textile
textile.AddYarn( yarn )

# Add the textile with the name "single yarn" to the TexGen database
# If the script is run from within the TexGen GUI the yarn is displayed
AddTextile("single yarn", textile)
```

The resulting yarn:

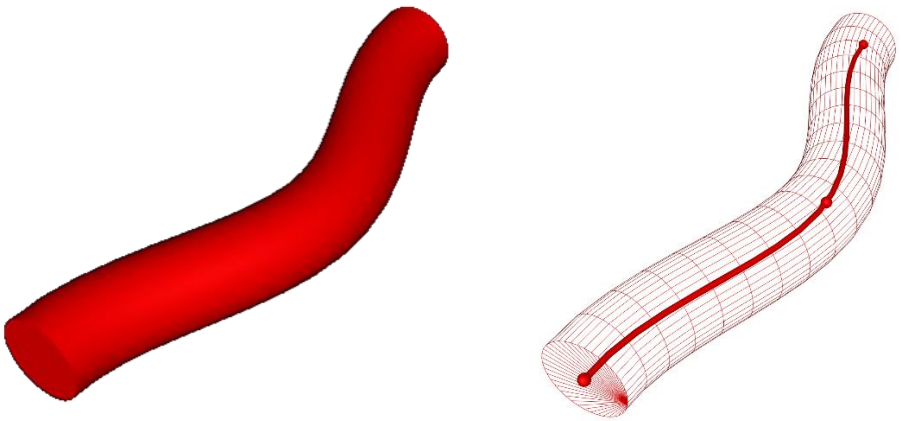


Figure 4: A single yarn created by specifying 3 nodes only (left). Showing nodes and interpolated yarn path (right)

2.6. Specifying Yarns with Varying Orientations

The default 'Up' vector is defined as (0,0,1)[2]. This is used to calculate the orientation of the yarn normal by calculating the cross-product of the yarn tangent and the Up vector. If the orientation of the yarn tangent is vertical, or close to vertical, then the use of the default Up vector will give incorrect results, manifested by a twist in the yarn when rendered. In order to avoid this an alternative Up vector should be specified using the extra parameters in the CNode initialisation:

```
Ni = CNode( XYZ(xi, yi, zi), XYZ(xt, yt, zt), XYZ(xu, yu, zu) )
```

where XYZ(x_t, y_t, z_t) and XYZ(x_u, y_u, z_u) are the tangent, V_{tan}, and up, V_{up}, vectors of the node respectively. Typically, V_{tan} is a unit vector tangent to the yarn path at the defined node, and V_{up} is a unit vector representing the up direction which should be approximately perpendicular to the tangent vector. V_{tan} will be redefined automatically when the interpolation function calculates the yarn path.

Example: Create an L-shaped yarn to demonstrate the use of node tangent and up vectors.

```
# UpVectors.py
#Create a textile and  yarn
textile = CTextile()
yarn = CYarn()

#Add nodes to the yarn
yarn.AddNode( CNode( XYZ(0.25, 5, 0.1), XYZ(0,-1,0),  XYZ(0, 0, 1) ) )
yarn.AddNode( CNode( XYZ(0.50, 3,  0),   XYZ(0,-1,0),  XYZ(0, 0, 1) ) )
yarn.AddNode( CNode( XYZ(0.00, 1,  0),   XYZ(0,-1,0),  XYZ(0, 0, 1) ) )
yarn.AddNode( CNode( XYZ(0.00, 0, -5),   XYZ(0,1,-1),  XYZ(0,-1,0) ) )

# Add the yarn to the textile
textile.AddYarn( yarn )
```

```
# Add the textile to the TexGen database
AddTextile( "demo", textile )
```

The result will then look like Figure 5(left) or (right) by removing the definition of tangent and up vectors.

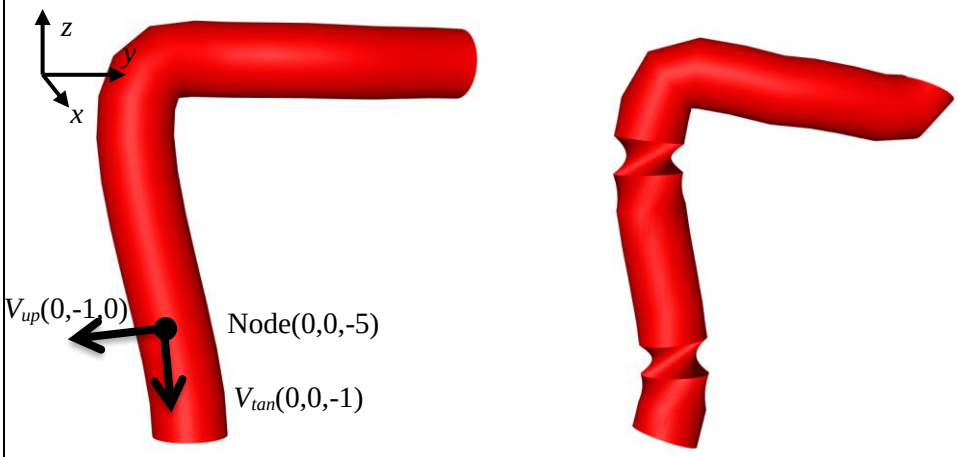


Figure 5: L-shaped yarn with (left) and without (right) tangent and up-vectors defined

The Up and tangent vectors can also be set subsequent to the creation of the node using

```
Ni.SetUp(XYZ( xu, yu, zu) )
Ni.SetTangent( XYZ(xt, yt, zt) )
```

3. Specifying Yarn Path Interpolation

After the nodes for a yarn have been specified an interpolation function is then set which calculates the yarn path between the nodes. A set of slave nodes are calculated which specify the final yarn path, the density of which can be set using the CYarn.SetResolution function.
The theory behind the different interpolation functions is described in Sherburn's thesis [2]. If no interpolation function is specified for a yarn the default CInterpolationBezier will be used.
All of the interpolation functions use the same CInterpolation base class which contains the calculated slave nodes. The yarn tangents for the master nodes are calculated when the interpolation is executed.
After the interpolation object has been created it must be assigned to the yarn. Both of these are typically performed in the same command.

3.1 CInterpolationBezier

The CInterpolationBezier function takes three optional Boolean parameters: *bPeriodic*, *bForceInPlaneTangent* and *bForceMasterNodeTangent*. *bPeriodic* is set to True by default ensuring that the both the position and slope are continuous at either end of the yarn section. *bForceInPlaneTangent* forces the tangent at the node to be oriented in the *xy* plane and is set to False by default. *bForceMasterNodeTangent* forces all master node tangents to be oriented in the *xy* plane and is set to False by default.

```
Interpolation = CInterpolationBezier()
Yarn.AssignInterpolation( Interpolation )
```

The example above will use the default parameters. Note that if one or more parameters are specified any subsequent parameters will take the default values. To set the second or third parameters the previous parameters must then also be set manually. For example, to set the second parameter whilst using the defaults for the first and third the command would be as follows:

```
Interpolation = CInterpolationBezier( True, True)
```

3.2 CInterpolationCubic

The CInterpolationCubic function takes the same parameters, with the same defaults, as the Bezier interpolation in Section 3.3.1

```
Interpolation = CInterpolationCubic()
```

3.3 CInterpolationLinear

Again the CInterpolationLinear function is used in the same way:

```
Interpolation = CInterpolationLinear()
```

4. Periodicity and Repeats

4.1 Repeats

The norm when creating TexGen models would be to add nodes to represent the smallest repeating section of a yarn. In order to create a larger section of a textile repeat vectors are specified for each yarn. The repeat vector, with Cartesian coordinates x_r, y_r and z_r is specified using:

```
yarn.AddRepeat( XYZ(x_r,y_r, z_r) )
```

4.2 Create Domains

After a yarn and repeat vectors have been specified a potentially infinite textile is created. In order to specify the specific region of the textile which is to be examined a domain is specified which defines the area of the model which is to be used, for example to be exported to third party software. Typically the domain will represent a unit cell but could be a larger number of repeats depending on the type of simulation which is to be run using the model.
The domain is built up from a set of planes[2] which represent a convex shape.

4.3 Default Domains

If one of the classes based on either CTextileWeave or CTextile3DWeave has been used then the simplest way to define the domain is to use the default domain function. In this case the domain will be assigned which is appropriate for the type of weave used:

```
textile.AssignDefaultDomain()
```

This function will define a domain which describes 6 planes which trims the textile to a unit cell. It should be noted that by default the domain is 10% larger than the distance between the maximum and minimum points on the yarns in z-direction.

4.4 Specifying Domain Planes

To manually specify a set of domain planes the CDomainPlanes() class is used. This can be initialised in two ways. Firstly with the lower and upper limits, *lo* and *up*, of a bounding box, in which case six planes will automatically be created to create the box domain:

```
Domain= CDomainPlanes( XYZ(x_lo, y_lo, z_lo), XYZ(x_up, y_up, z_up) )
```

Here the subscripts *lo* and *up* specify the lower and upper box limits in Cartesian coordinates, respectively.
Secondly the planes can be specified manually. First an empty domain object is created:

```
Domain= CDomainPlanes( )
```

The planes are then created using the PLANE() function and added to the CDomainPlanes object. The plane, *Plane*, is defined as a normal vector with Cartesian

components x_n, y_n and z_n. The vector length, d, is the distance of the plane from the origin:

```
Plane= PLANE(XYZ(xn, yn, zn), d)
Domain.AddPlane(Plane )
```

After the domain has been specified, it is added to the textile.

```
textile.AssignDomain(Domain)
```

Example: For the yarn created as shown in the example of Section 2.5, add the line below before the AddTextile() call. This will specify bounding box of the domain.

```
textile.AssignDomain(CDomainPlanes(XYZ(-0.5, 0, -0.5), XYZ(1.5, 7, 0.5)))
```

The yarn in the domain will then look like:

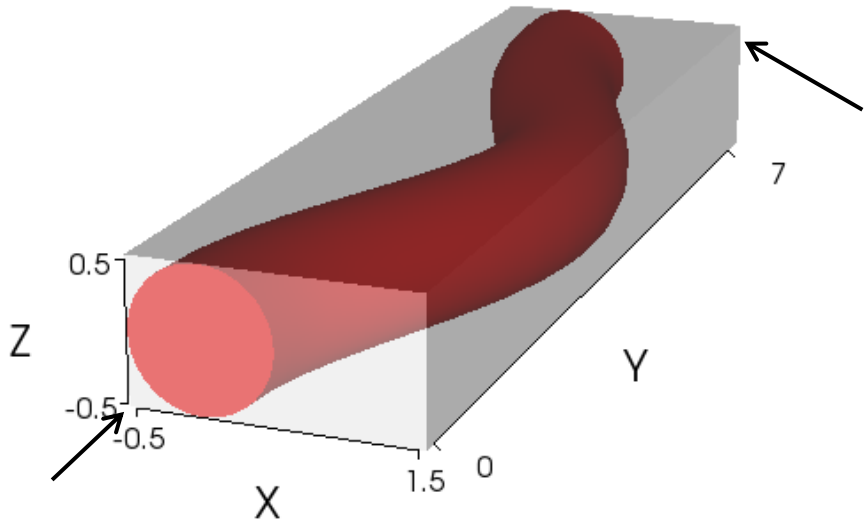


Figure 6: Example domain (shaded in grey) containing the yarn path example of Section 2.5. The points used to specify the bounding box are marked with arrows.

The same result can be achieved by specifying the 6 planes of the bounding box separately. Again, add this code before the AddTextile() call in the example of Section 2.5:

```
domain = CDomainPlanes()
domain.AddPlane( PLANE(XYZ(1, 0, 0), -0.5) )
domain.AddPlane( PLANE(XYZ(-1, 0, 0), -1.5) )
domain.AddPlane( PLANE(XYZ(0, 1, 0), 0) )
domain.AddPlane( PLANE(XYZ(0, -1, 0), -7) )
domain.AddPlane( PLANE(XYZ(0, 0, 1), -0.5) )
domain.AddPlane( PLANE(XYZ(0, 0, -1), -0.5) )
textile.AssignDomain(domain)
```

Example: After creating the yarn (Section 2.5), the yarn repeat is specified for the three principal axes before the AddYarn() command. In the *x*-direction, the yarn will be repeated in parallel at a distance of 3, in the *y*-direction, the yarn is thought to be continuous and the hence the yarn repeat is set to 7. For the *z*-axis, the yarns are repeated at a distance of 1 and staggered in *x*-direction by 0.5.
The resulting model is shown in Figure 7.

```
# Add repeat vectors
yarn.AddRepeat( XYZ(3, 0, 0) )
yarn.AddRepeat( XYZ(0, 7, 0) )
yarn.AddRepeat( XYZ(0.5, 0, 1) )
```

A 5x larger domain is specified by adding

```
Plower = XYZ(-0.5, 0, -0.5)
Pupper = XYZ(5*1.5, 5*7, 5*0.5)
textile.AssignDomain( CDomainPlanes(Plower, Pupper) )
```

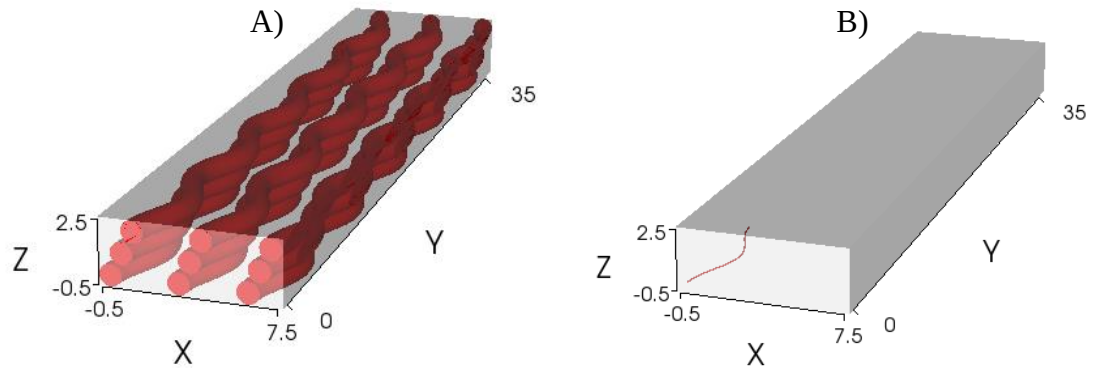


Figure 7: The yarn of the Example in Section**2.5.** Add Textile to Database with A) the added repeat and a 5 times larger domain than defined in the example in Section 3.5.2 and B) the specified yarn path.

4.5 Prismatic Domains

For some, more complex, textile configurations a box-shaped domain may not be appropriate. Typically, for a composite material the domain area will represent the area to be filled with the matrix material. If the yarns form a net-shaped textile, for example a T-piece or a curved section of a braided textile, then it is useful to be able to create the domain to conform to the area occupied by the yarns.
In this case, a domain is created using the CDomainPrism class. This creates a prismatic domain with a constant cross-section. In order to do this, the CYarn class is utilised to create a yarn with constant cross-section, specified using a CSectionPolygon to create the shape required. The length and orientation of the domain are determined by a master node at either end of the 'yarn'. The surface of the domain is then generated by creating the yarn surface mesh, forcing one plane to be created between each pair of nodes around the polygon section outline.
Domains generated in this way can have a concave shape if needed.

First a vector of points must be set up which define the outline of the domain in x, y coordinates:

```
# Create vector for domain polygon points
```

```
points = XYVector()
```

Then add the points to the vector using:

```
points.push_back(XY(x,y))
```

Finally create the domain using the points vector and two XYZ points defining the length and orientation of the prism to be created and assign the domain to the textile:

```
domain = CDomainPrism(points, XYZ(x1, y1, z1), XYZ(x2, y2, z2))
textile.AssignDomain(domain)
```

Example: Creating a prism domain with a T-shaped cross-section

```
#Create a textile
textile = CTextile()

# Create vector for domain polygon points
points = XYVector()

# Add points to polygon and create prism domain using
# the vector of points and start and end point of prism
#T domain
points.push_back(XY(1,0))
points.push_back(XY(0.4,0))
points.push_back(XY(0.3,0.05))
points.push_back(XY(0.15,0.2))
points.push_back(XY(0.1,0.3))
points.push_back(XY(0.1,1))
points.push_back(XY(-0.1,1))
points.push_back(XY(-0.1,0.3))
points.push_back(XY(-0.15,0.2))
points.push_back(XY(-0.3,0.05))
points.push_back(XY(-0.4,0))
points.push_back(XY(-1,0))
points.push_back(XY(-1,-0.1))
points.push_back(XY(1,-0.1))
domain = CDomainPrism(points,XYZ(0,0,0), XYZ(0,2,0))

# Add the domain to the textile
textile.AssignDomain(domain)

# Create a yarn passing through the domain
yarn = CYarn()
yarn.AddNode( CNode(XYZ(0,-0.2,0)))
yarn.AddNode( CNode(XYZ(0, 2.2,0)))

section = CSectionEllipse(0.1,0.1)
yarn.AssignSection( CYarnSectionConstant(section))
yarn.SetResolution(10)
textile.AddYarn(yarn)
AddTextile('test', textile)
```

The resulting prism domain is shown in Figure 8.

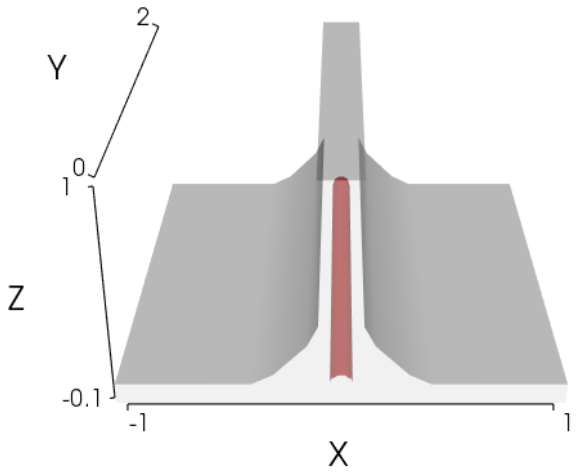


Figure 8: T-shaped prism domain example.

Repeat vectors must be considered carefully with this type of domain. For the T-shape illustrated it may be appropriate to have a repeat vector in the y direction only.

The accompanying example script, PrismDomain.py, gives examples of use of the CDomainPrism class with other cross-section shapes.

5. Yarn Cross-Sections

A variety of yarn cross-sectional shapes can be specified (Section 3.1) which can then be applied either to the entire yarn or at specified points along the yarn (Section 3.2).

5.1. Cross-Sectional Shapes

The cross-sectional shape of the created yarns can be specified as a variety of different shapes. These shapes are assigned after the nodes have been added to the yarn (Section 2.3. Create Master Nodes). A number of different shapes are available in TexGen, all with the same CSection base class.
All cross-sections are defined as two dimensional shapes which are then applied to the yarn at the specified points, the TexGen software automatically calculating the 3D points to generate the yarn surface.

5.1.1. CSectionEllipse

To define an elliptical yarn cross-section with width, *w*, and height, *h*, use:

```
Shape = CSectionEllipse( w, h )
```

5.1.2. CSectionLenticular

To define a lenticular yarn cross-section with width, *w*, and height, *h*, use:

```
Shape = CSectionLenticular( w, h, Distortion )
```

The *Distortion* parameter defines the vertical offset of the widest part of the shape.

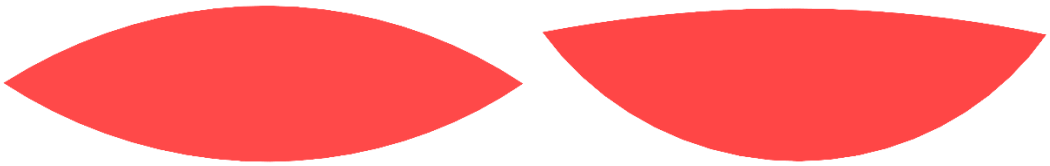


Figure 9: Lenticular section a) *w*=1, *h*=0.3, b) *w*=1, *h*=0.3, *distortion*=0.1

5.1.3. CSectionPowerEllipse

Power elliptical yarn cross-sections are defined using a width, *w* and height, *h*. The power parameter governs the shape, making the section resemble a rectangle with rounded edges when power<1 or a shape similar to a lenticular cross-section when power>1. The final parameter, *XOffset*, is optional and is the offset from the centre which specifies the maximum height of the shape.

```
Shape = CSectionPowerEllipse (w, h, Power, XOffset)
```

Figure 10: Power ellipse, w = 1, h = 0.5 a) power = 0.5, b) power = 0.5, xOffset = 0.25, c) power = 1.3

5.1.4. CSectionRectangle

To define a rectangular cross-section with width, *w*, and height, *h*, use:

```
Shape = CSectionRectangle( w, h)
```

Note that it is preferable to use this, rather than a power ellipse with a power of 0, as the volume mesh is much more regular.

5.1.5. CSectionPolygon

The polygon section can be used for irregularly shaped cross-sections where the outline is specified by a vector of points. The STL template XYVector() is created, points are added to it and then this is used to create the shape:

```
# PolygonSection.py
# Set up a vector of points to specify the 2D yarn cross-section
Points = XYVector()

Points.push_back(XY(1.47887, -0.080294))
Points.push_back(XY( 1.1267, 0.0421169))
Points.push_back(XY( 0.763247, 0.123317))
Points.push_back(XY( 0.393884, 0.173093))
Points.push_back(XY( 0.0221259, 0.190496))
Points.push_back(XY( -0.349862, 0.166824))
Points.push_back(XY(-0.716936, 0.103844))
Points.push_back(XY( -1.26423, -0.0106595))
Points.push_back(XY( -0.94015, -0.143083))
Points.push_back(XY(-0.567315, -0.141597))
Points.push_back(XY( 0.178121, -0.132177))
Points.push_back(XY( 0.550909, -0.13775))
Points.push_back(XY( 0.923697, -0.14277))
Points.push_back(XY( 1.10991, -0.133933))

Shape = CSectionPolygon(Points)
```

Figure 11: Polygon section

5.1.7. CSectionHybrid

The hybrid section shape is constructed from a combination of several previously defined shapes. This can be done in several ways.

1. Two shapes can be defined and then used as the top and bottom half of the hybrid shape.

```
Top = CSectionEllipse( 1.0, 0.4 )
Bottom = CSectionPowerEllipse( 1.0, 0.4, 0.4, 0.25 )
sections.AddSection( CSectionHybrid( Top, Bottom ) )
```

2.Specify four shapes which can then be used for each of four quadrants.

```
TopLeft = CSectionRectangle(1.0, 0.4)
TopRight = CSectionEllipse( 1.0, 0.4 )
BottomLeft = CSectionPowerEllipse(1.0, 0.4, 1.5)
BottomRight = CSectionPowerEllipse( 1.0, 0.4, 0.4, 0.25 )
sections.AddSection( CSectionHybrid(TopRight, TopLeft, BottomLeft,…
                                                    BottomRight) )
```

3. Create an empty hybrid shape and then add shapes, specifying the position of the divisions between them.

```
Hybrid = CSectionHybrid()
Hybrid.AddDivision(0.25)
Hybrid.AddDivision(0.75)
Hybrid.AssignSection(0, CSectionLenticular(1.0,0.4))
Hybrid.AssignSection(1, CSectionEllipse(1.0,0.4))
sections.AddSection(Hybrid)
```

Figure 12: Hybrid Section a) Two sections b) Four sections c) Two divisions at 0.25 and 0.75

5.1.8. CSectionRotated

The CSectionRotated shape is used to rotate another, already defined, shape. The shape to be rotated and an angle in radians must be specified:

```
Shape = CSectionEllipse( 1.0, 0.5)
RotatedShape = CSectionRotated( Shape, pi/8 )
```

5.1.9. CSectionScaled

The CSectionScaled shape is used to create a scaled version of another shape. The scale parameter is an XY structure which specifies the x and y scale factors respectively.

```
ScaledShape = CSectionScaled( Shape, XY(2,1) )
```

5.2. Applying Cross-Sections To a Yarn

Cross sections are applied to a yarn using one of the classes derived from the CYarnSection base class. The CYarnSectionConstant class will specify a constant cross-section yarn and the CYarnSectionInterpNode and CYarnSectionInterpPosition classes enable the cross-section to be varied along the length of the yarn.

5.2.1 Constant Cross-Section

To specify a cross-section which is constant along the length of a yarn the CYarnSectionConstant is used.

```
section = CYarnSectionConstant(MyShape)
```

5.2.2 Cross-Sections Specified at Master Nodes

To assign a cross-section at each node first the CYarnSectionInterpNode object must be created and then the AddSection method called for each node. Note that the same number of sections as nodes must be specified.

```
section=CYarnSectionInterpNode()
section.AddSection( MyShape_i )
```

5.2.3 Cross-Sections Specified at Given Positions Along the Yarn

Varying yarn shapes can be assigned at different (arbitrary) positions along the yarn. This can be achieved using the CYarnSectionInterpPosition() function. A yarn is considered to have a normalised length of 1. Hence the position, d_n, at which the yarn cross-section will be specified is in the range $0 \leq d_n \leq 1$. The cross-sectional shape will be linearly interpolated between the different cross-sections. If the cross-sections are not specified for the start and end of the yarn then the interpolation function will ensure that these are the same to ensure periodicity of the yarn.

```
section = CYarnSectionInterpPosition()
section.AddSection( d_n, MyShape )
```

5.2.4 Assigning the Cross-Sections to the Yarn

After the sections have been defined they must then be assigned to the yarn:

```
yarn.AssignSection(section)
```

After this has been done, the yarn can be added to the textile as described in Section 2.4. Add Yarns to Textile

Example: Create a textile and add a yarn which is defined by three master nodes similar to the example shown in Section 2.3. Create Master NodesVarying cross-sectional shapes will be defined at the three master nodes. First, an ellipse will be created, second the same ellipse will be assigned, rotated by 45° and at the end a power ellipse will be assigned with a power <<1 which will lead to an almost rectangular cross-section.

```
# CrossSections.py

#Create textile
textile = CTextile( )

# Create yarn
yarn = CYarn( )

# Add nodes to yarn
yarn.AddNode( CNode(XYZ(0, 0.0, 0)) )
yarn.AddNode( CNode(XYZ(1, 3.5, 0)) )
yarn.AddNode( CNode(XYZ(0, 7.0, 0)) )

#Define three different cross-sectional shapes at the master nodes
section=CYarnSectionInterpNode()
section.AddSection(CSectionEllipse(1, 0.1))
```

```
section.AddSection(CSectionRotated(CSectionEllipse(1, 0.1),
math.radians(45)) )
section.AddSection(CSectionPowerEllipse (2, 0.3, 0.1, 0.0))
#Assign the cross-sections to the yarn and add to the textile
yarn.AssignSection(section)
textile.AddYarn(yarn)

#Add the textile to the database
AddTextile("varying shapes", textile)
```

The result will look like:

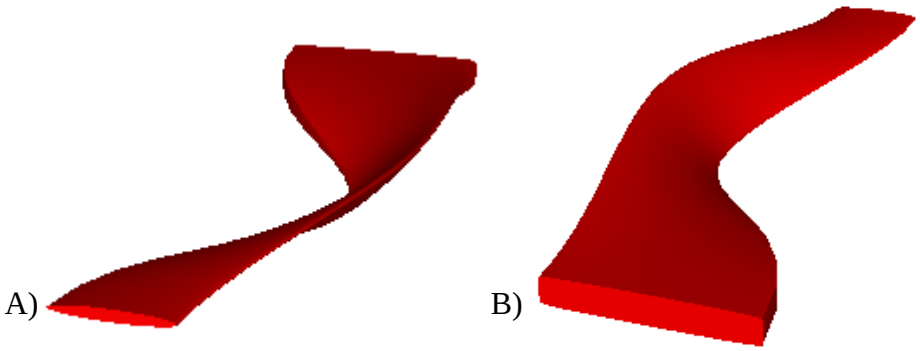


Figure 13: Yarn with varying cross-sections. A) Front view and B) view from the back.

5.2.5 Define Cross-Sections Which are not Normal to the Yarn Path

By default the yarn cross-sectional shapes are defined normal to the specified yarn paths. It is possible; however, to align a cross-section at an angle, α, to the normal of a yarn path. This is achieved by specifying the angle as a rotation of the yarn cross-section to the local coordinate system at the node. This can be set by:

```
node.SetAngle(alpha)
```

with α specified in radians.

This example creates a straight yarn which consists of 3 nodes and a constant elliptical yarn cross-section. The elliptical yarn cross-section at the centre node is then rotated with respect to the normal by 45°.

```
# AngledCrossSection.py
##############################################
#Create a straight yarn with constant elliptical cross-section
textile = CTextile( )
yarn = CYarn( )
yarn.AddNode( CNode(XYZ(0, 0.0, 0)) )
yarn.AddNode( CNode(XYZ(0, 3.5, 0)) )
yarn.AddNode( CNode(XYZ(0, 7.0, 0)) )
ellipse=CSectionEllipse(1, 0.3)
section = CYarnSectionConstant(ellipse)
yarn.AssignSection(section)

##############################################
#rotate the cross-sections at mid node
index=1
```

```
node = yarn.GetNode(index)
node.SetAngle(PI/4)
yarn.ReplaceNode(index, node)

##########################################################
#visualise the textile
textile.AddYarn(yarn)
AddTextile("tex", textile)
```

The yarn cross-section will be as shown below.

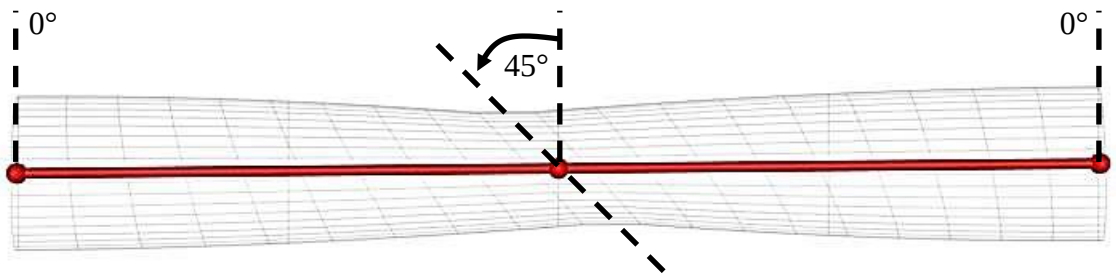


Figure 14: Yarn top view shown with transparent surface and defined constant cross-sectional shape. The cross-section at the centre is rotated by 45° with respect to the normal to the yarn path.

6 Properties

Properties can be set for either an individual yarn or an entire textile. CProperties is the base class from which both the yarn and textile properties are derived. It allows the following properties to be set:

- Yarn linear density
- Fibre density
- Fibre area
- Fibre diameter
- Number of fibres per yarn
- Young's modulus, *x*
- Young's modulus, *y*
- Young's modulus, *z*
- Shear modulus, *xy*
- Shear modulus, *xz*
- Shear modulus, *yz*
- Coefficient of thermal expansion, *x*
- Coefficient of thermal expansion, *y*
- Coefficient of thermal expansion, *z*
- Poisson's ratio, *x*
- Poisson's ratio, *y*
- Poisson's ratio, *z*

When using any of the functions associated with the CProperties class the units can be specified. A complete list of the available units is given here. If no units are specified then the default units are used.
The following sections illustrate setting properties for textiles and yarns. For a complete list of available functions see the API documentation for CPropertiesTextile and CPropertiesYarn

6.1. Textile Properties

To set a property to be associated with the CTextile class use, eg:

```
Textile.SetYoungsModulusX( val, "GPa" )
```

To set the same property for all yarns in a textile use:

```
Textile.SetAllYarnsYoungsModulusX( val )
```

This function iterates through each yarn in the textile and sets the property for each yarn. In this case the default units (MPa) would be used.

Properties which can only be set for an entire textile are:

- Matrix Young's modulus
- Matrix Poisson's ratio
- Matrix coefficient of thermal expansion
- Areal density

6.2. Yarn Properties

To set the property for a specific yarn use, eg:

```
Yarn.SetFibreArea( val, "mm^2" )
```

6.3 Use of Properties by Export Functions

The properties set are used when exporting models, for example as an Abaqus input file. The material properties are exported and associated with the yarns or element sets as appropriate. The fibre volume fractions, V_f., associated with each element are also exported and are calculated using the property data and the local cross-sectional area. In this case it is not necessary to have specified all of the possible properties, the following combinations being sufficient.

The nominal fibre diameter, d_f, and the number of fibres per yarn, n_f:

```
textile.SetFibreDiameter( df, unit )
textile.SetFibresPerYarn( nf )
```

Note that the fibres are assumed to have a constant circular cross-sectional areas.

Alternatively, the fibre area, *A*, in a yarn cross-section in *unit* (default *unit* = 'm^2') within a yarn can be set:

```
textile.SetFibreArea( A, unit )
```

If the cross-sectional area of a yarn, A_{yarn}, is known (as defined by the specified cross-section) then the fibre volume fraction, $V_{f\_}$ is calculated using:

```
Vf_ = A / Ayarn
```

Setting the linear density of the yarn, *roh_linear*, and the fibre density, *roh*, could also be used to calculate the V_f by TexGen:

```
textile.SetFibreDensity( roh, unit )
textile.SetYarnLinearDensity( roh_linear, unit )
```

The default units for *roh* and *roh_linear* are *unit* = 'kg/m^3' and *unit* = 'kg/m', respectively.

Setting the areal density, A_f, of a fabric can also be used to calculate the V_f.

```
textile.SetArealDensity( Af, 'kg/m^2' )
```

Default *unit* = 'kg/m^2'

7. Automatically Generated Textiles

Predefined weave patterns can be created using textile classes which use the CTextile class as a base class. They include extra information about the weave pattern together with functionality to build the yarns in the textile using the pattern information. A diagram of the textile classes is shown in Figure 15. Because the textiles use the CTextile base class it is still possible to access the yarn and node data to make refinements after the textile has been generated.

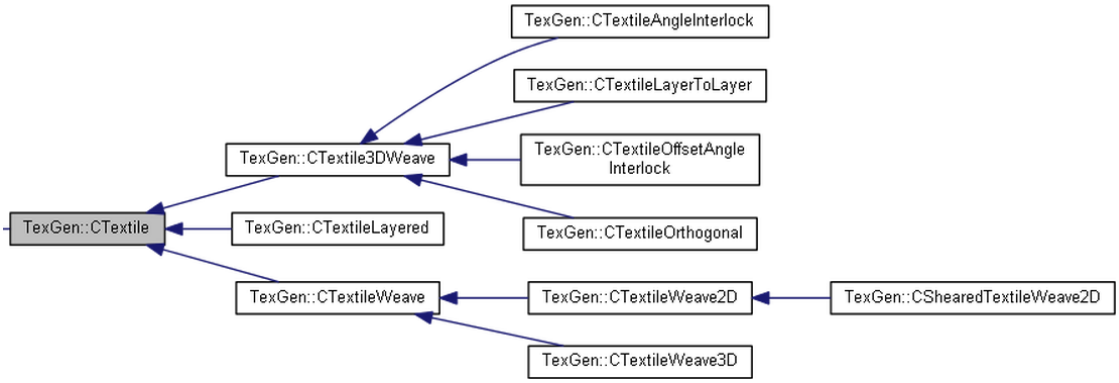


Figure 15: Schematic of the CTextile class structure in TexGen.

7.1 2D Weave - CTextileWeave2D

A 2D weave can be created using CTextileWeave2D.

```
# Create a 2D weave textile
weave = CTextileWeave2D( nweft, nwarp, s, t, ref )
```

where the number of weft and warp yarns in the unit cell, *nweft* and *nwarp*, the spacing between the yarns, *s*, the thickness of the fabric, *t*, and the Boolean operator (True/False) to refine the model, *ref*, are specified.

This class includes a two dimensional array, size (*nwarp* x *nweft)* x 2, which specifies the weave pattern by defining whether the warp and weft yarn are up or down at each warp/weft crossover. After calling the constructor for the class the warp yarns are initially at the top as shown in Figure 16a. To create a weave pattern, the yarns need to be interlaced. This is achieved by swapping the master node positions by calling:

```
weave.SwapPosition( weft_i, warp_i )
```

See also: http://texgen.sourceforge.net/index.php/Scripting_Create_Models

Example: Create a 2/2 plain weave using the CTextileWeave2D class.

```
# 2DTextile.py

# Specify weave parameters
nwarp=2    #Number of weft yarns in the unit cell
nweft=2    #Number of warp yarns in the unit cell
```

```
s=1         #Spacing between the yarns
t=0.1       #Thickness of the fabric (sum of two yarn heights)
ref=True  #Refine model (True/False)

# Create 2D textile
weave = CTextileWeave2D( nweft, nwarp, s, t, ref )

# Set the weave pattern
weave.SwapPosition(0, 0)
weave.SwapPosition(1, 1)

#Add to the textile database
AddTextile(weave)
```

The resulting yarn paths before and after selected master nodes were swapped are shown in Figure 16.

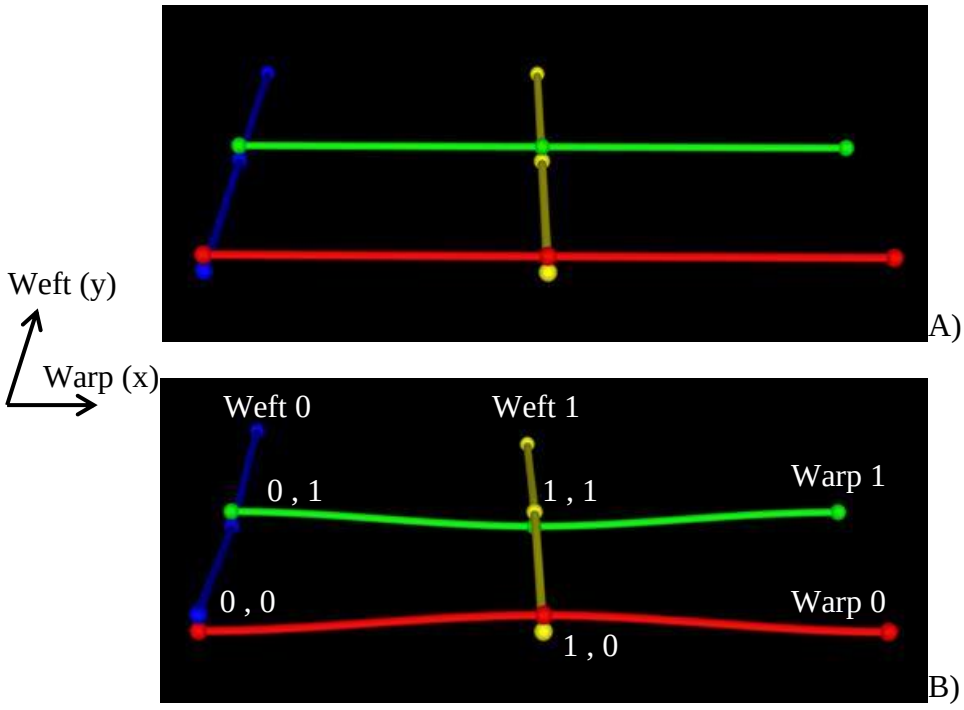


Figure 16: Yarn path and master nodes of a CTextileWeave2D with 2 warp and 2 weft yarns. A) Directly after creating the yarn paths and B) after swapping specified master nodes indexes given (warp, weft).

For this weave, the yarn widths, *w*, and yarn heights, *h*, can be set for all yarns with a single command:

```
weave.SetYarnWidths( w )
weave.SetYarnHeights( h )
```

A gap can be specified which is enforced when the refine option is specified to define the gap between the yarn surfaces.

```
weave.SetGapSize( g )
```

Yarn width and heights can also be set for individual yarns, using the warp or weft index. For the warp yarns, $warp_i$, use:

```
weave.SetXYarnWidths( warpi, w )
weave.SetXYarnHeights( warpi, h )
```

and for the weft yarns, $weft_i$:

```
weave.SetYYarnWidths( welti, w )
weave.SetYYarnHeights( welti, h )
```

Individual yarn spacing, *s*, between a yarn and the yarn with the next higher index, *i*+1, can be set by for the warp and weft yarns by:

```
weave.SetXYarnSpacings( warpi, s )
weave.SetYYarnSpacings( welti, s )
```

A default model domain (Section 4.3 Default Domains) can be assigned which creates domain boundaries in the spaces parallel to the yarns (Figure 18). This ensures that yarns are not split longitudinally in the unit cell:

```
weave.AssignDefaultDomain()
```

Using the function as above uses the default parameters, the first specifying whether a sheared domain is to be created (default is False), the second whether an extra 10% height should be added over the actual height of the yarns (default is True). To specify the default domain without extra height use

```
Weave.AssignDefaultDomain( False, True )
```

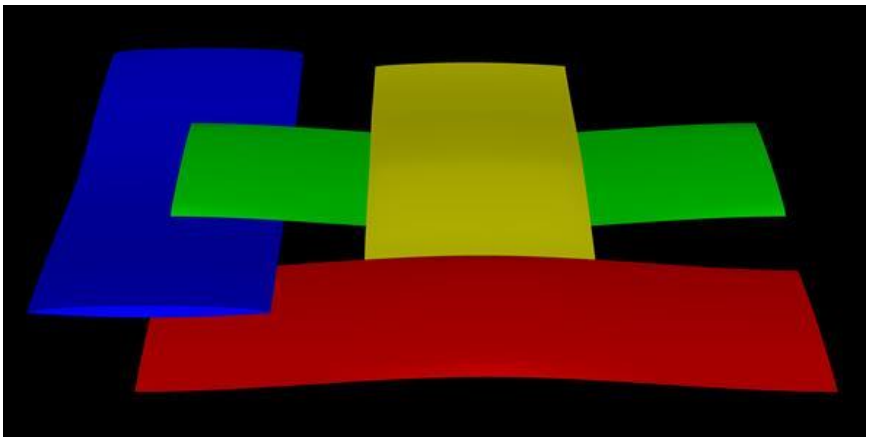


Figure 17: Example plain weave with nweft=2, nwarp=2, s=1, g=0.8, t=0.1and swap positions as shown in Figure 16.

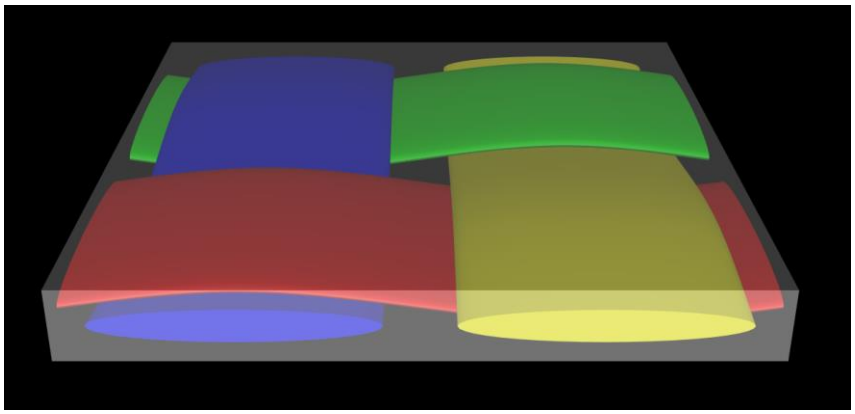


Figure 18: Plain weave with default domain

7.2 Sheared Textiles - CShearedTextileWeave2D

Sheared textiles are created in much the same way as the 2D textiles described in Section 7.1 2D Weave - CTextileWeave2D. They are created using the CShearedTextileWeave2D class which includes an extra parameter for specifying the shear angle. Again a refine option is available, the theory of which is described by Brown et al. [3]

An example script for creating a sheared textile is given below and the resulting textile shown in Figure 19.

```
# ShearedTextile.py
# Create a 4x4 2d woven textile with yarn spacing of 5 and thickness 2
# The fourth parameter is the shear angle in radians
# The sixth parameter indicates whether to refine the textile to avoid
intersections
Textile = CShearedTextileWeave2D(4, 4, 5, 2, 5*pi/180.0, True);

# Set the weave pattern
Textile.SwapPosition(3, 0);
Textile.SwapPosition(2, 1);
Textile.SwapPosition(1, 2);
Textile.SwapPosition(0, 3);

# Adjust the yarn width and height
Textile.SetYarnWidths(4);
Textile.SetYarnHeights(0.8);

# Setup a default domain
# Use True parameter to generate a sheared domain (otherwise the
# domain will be aligned to the axes, cutting the yarns)
Textile.AssignDefaultDomain( True )

# Add the textile
AddTextile(Textile)
```

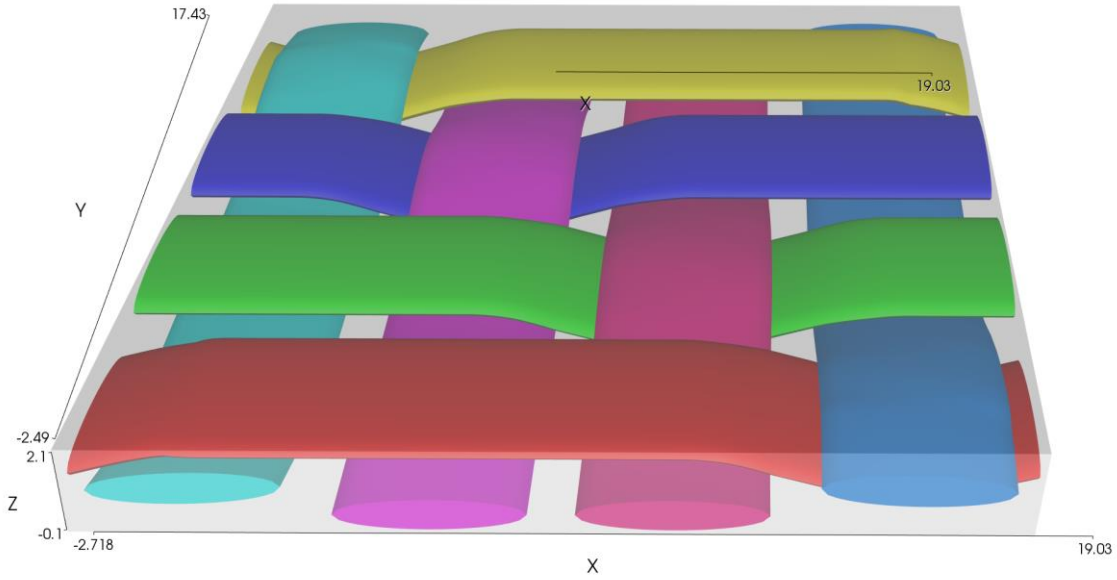


Figure 19: Sheared textile generated using the CShearedTextile2D class with refinement

7.3 Layered Textiles - CTextileLayered

To create a stack of textiles, the CTextileLayered class (Figure 15) can be used. This class is used to layer any combination of textiles which have previously been created. To create a layered textile use:

```
LayeredTextile = CTextileLayered()
```

After creating textiles as described in previous sections the textile layers are added one at a time to the CTextileLayered class:

```
LayeredTextile.AddLayer( LayerName, XYZ(xoff, yoff, zoff) )
```

The *LayerName* in this case is the name of any textile in the TexGen database. Several layers of the same textile can be used. An XYZ vector, x_{off}, y_{off}, z_{off} , defines the offset of the layer with respect to the first layer in the stack, thus enabling a shift between individual layers if required. When all layers have been added to the textile it should then be added to the TexGen database using the AddTextile command.

Example: A layered textile which consists of two layers of a plain weave is created:

```
# LayeredTextile.py
# Create a plain weave textile
t = 0.1 #layer thickness
weave = CTextileWeave2D( 2, 2, 0.8, t, True )
weave.SwapPosition(0, 0)
weave.SwapPosition(1, 1)
weave.SetGapSize(0.01) # set a gap between yarns

#Create the layered textile
LTextile = CTextileLayered()

# Add a plain weave layer with no offsets
```

```
LTextile.AddLayer( weave, XYZ(0, 0, 0) )

# Add a second plain weave layer, offset in the z-direction
# by the textile thickness and by 0.5 and 0.5 in the x and y directions
LTextile.AddLayer( weave, XYZ(0.5, 0.5, t) )

# Get the default domain of the plain weave and its min and max
coordinates
Domain = weave.GetDefaultDomain()
Min = XYZ()
Max = XYZ()
Domain.GetBoxLimits( Min, Max )

# Get the domain upper surface
Plane = PLANE()
index = Domain.GetPlane( XYZ(0,0,-1), Plane )
# Offset the top surface of the domain by the depth of the plain weave
domain
Plane.d -= Max.z - Min.z
Domain.SetPlane( index, Plane )

# Assign the extended domain to the layered textile
LTextile.AssignDomain( Domain )

#Add the textile to the database
AddTextile( "LayeredTex", LTextile )
```

The resulting layered textile is shown in Figure 20.

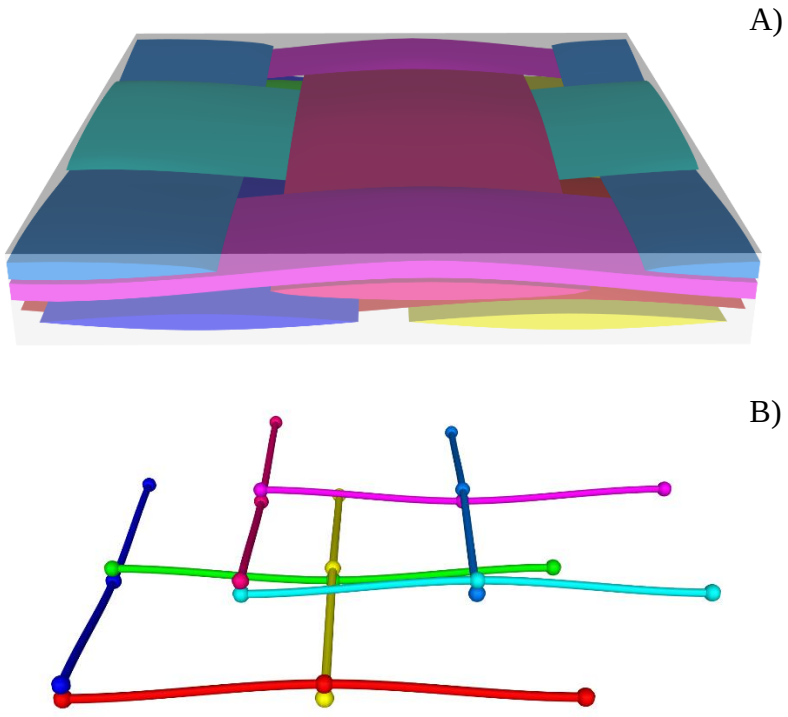


Figure 20: Two layer plain weave example. A) the surfaces of the yarns and B) the yarn paths and master nodes are shown.

7.4 3D Textiles

There are three types of automatically generated 3D weaves which all use the CTextile3DWeave base class. This contains the information about the number of warp and weft yarns, number of warp and weft layers and the number of binder yarns as well as a three dimensional grid which defines the weave structure. The grid contains integers which define whether the yarn at that point is a warp, weft or there is no yarn at that point. This is illustrated in Figure 21 where the initial setup for a layer-to-layer textile has been carried out and the grid indices are shown.

When constructing a textile using these classes the textile is built up from layers of straight warp and weft yarns with an extra layer at the top and bottom to allow for binders to pass over and under the warp/weft stacks. Figure 21 illustrates the empty layer with a z index of 0 which creates space for binder yarns to be placed at the bottom of the stack, ie underneath the bottom weft yarn.

In the warp direction a ratio of warp to binder yarns specifies the locations of the warp stacks and the through thickness binder yarns. Figure 21 shows a warp:binder ratio of 1:2.

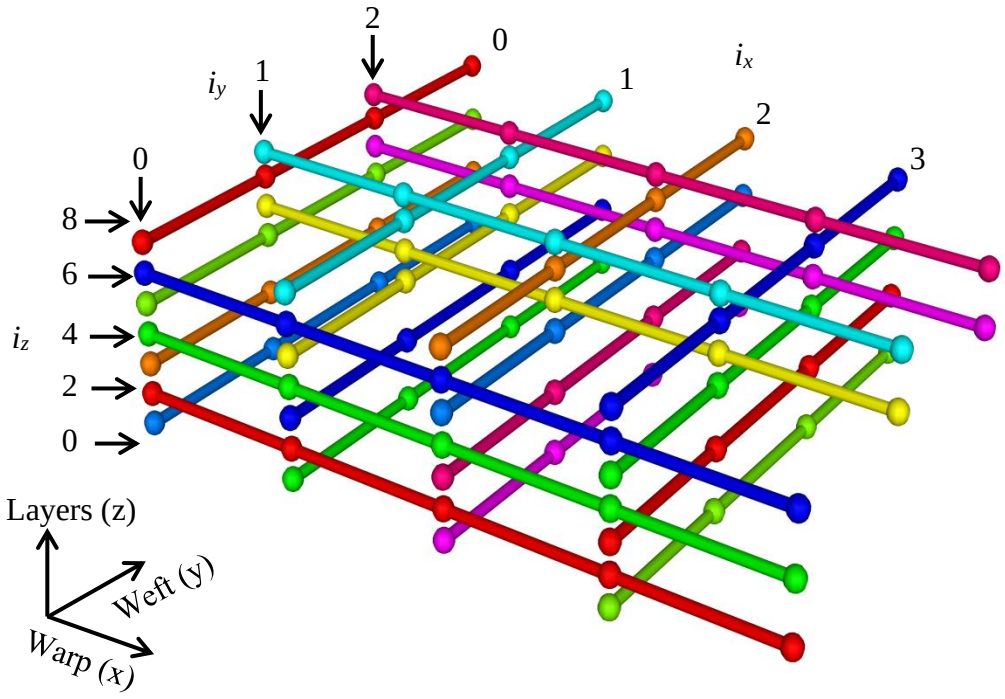


Figure 21: Example yarn paths and nodes of a 3D weave before weave positions are swapped. The basic parameters for CTextileLayerToLayer( ) are n_x = 3, n_y = 4, n_{bl} = 2, n_{weft}=4. The grid indices are displayed.

7.4.1 Orthogonal 3D Textile

In an orthogonal textile it is assumed that the binder yarns pass through the stack vertically. The grid defining the weave structure contains an entry at each crossing of the warp and weft stacks and there are therefore only two possible positions for the binder yarns in the grid, either at the top or bottom.

To create an orthogonal 3D textile use the CTextileOrthogonal() class:

```
weave = CTextileOrthogonal(nx, ny, sx, sy, hx, hy, ref)
```

where:

n_x – number of x (warp & binder) yarns
n_y – number of y (weft) yarns
s_x – x (warp / binder) yarn spacing
s_y – y (weft) yarn spacing
h_x – x (warp) yarn heights
h_y – y (weft) yarn heights
ref – a Boolean variable showing whether geometry refinement is required or not

The ratio of warp to binder yarns must be specified and this pattern will be repeated to give the total number of yarns in the warp direction. Setting up the ratio 1:1 will give equal number of warp and binder yarns repeating one after the other. If the warp ratio is set to zero then all yarns in the *x* direction will be binders.

```
weave.SetWarpRatio( 1 )
weave.SetBinderRatio( 1 )
```

Layers are added to the grid weave pattern as required and the yarns are subsequently created and their node positions allocated using this information. There are two ways of building up the layers. The simpler way is to specify a number of warp and weft layers and then use the SetupLayers function to automatically generate them.

```
NumWarpLayers = 2
NumWeftLayers = 3
Textile.SetupLayers( NumWarpLayers, NumWeftLayers )
```

Alternatively the stack can be built manually giving more control over the positioning of the layers. (The SetupLayers function will generate alternating warp and weft layers). The stack is built up from the bottom so, in order to leave space in the grid for the binder yarn when it passes underneath the stack an empty layer should be added first.

```
Weave.AddNoYarnLayer()
weave.AddYLayer()
weave.AddWarpLayer()
```

The *x* and *y* dimensions of the grid are governed by n_x and n_y, the *z* dimension being the total number of layers added.

Alternatively, if needed *numLayers* weft layers can be added simultaneously with no space left for warp layers:

```
weave.AddYLayers(numLayers)
```

The binder yarns are added by calling the function

```
weave.AddBinderLayer()
```

which adds a layer of binder yarns parallel to warp yarns. In the case of an orthogonal weave there will be one binder layer at the top of the stack.

The dimensions of warp, weft and binder yarns are assigned as well as the spacing between them:

```
weave.SetWarpYarnWidths( warp_width )
weave.SetBinderYarnWidths( binder_width )
weave.SetBinderYarnHeights( binder_height )
weave.SetBinderYarnSpacings( binder_spacing )
weave.SetYYarnWidths( weft_width )
```

Cross-sections of the all yarns are power-ellipses by default and their shape should be defined by a power-parameter α (with value of 0 giving a rectangle, value of 1 giving an ellipse and value higher giving a lenticular cross-section):

```
weave.SetWarpYarnPower( αwarp )
weave.SetWeftYarnPower( αweft )
weave.SetBinderYarnPower( αbinder )
```

In order to create the binder yarn path the binder position is specified to be at either the top or bottom of the stack at each intersection of the binder yarn with the weft yarns. The binder yarns are initialised in the top stack and therefore the positions where they are at the bottom of the stack should be specified. For each position in the *x,y* plane *(i,j)* of the grid the binder location can be swapped between the top and bottom position using:

```
weave.SwapBinderPosition( i, j )
```

When the grid of information about positions of warp, weft and binder yarns has been established TexGen then uses that information to automatically generate a textile. In the first instance one node is generated for each grid location, specifying nodal positions at the point where yarns cross, shown by the red yarn in the foreground of Figure 22. This results in large intersections between the binder yarn and top and bottom weft yarns. Additional nodes are added to the binder yarns such that they follow the contour of the weft yarns around which they pass. This is illustrated by the green yarn at the rear of Figure 22.

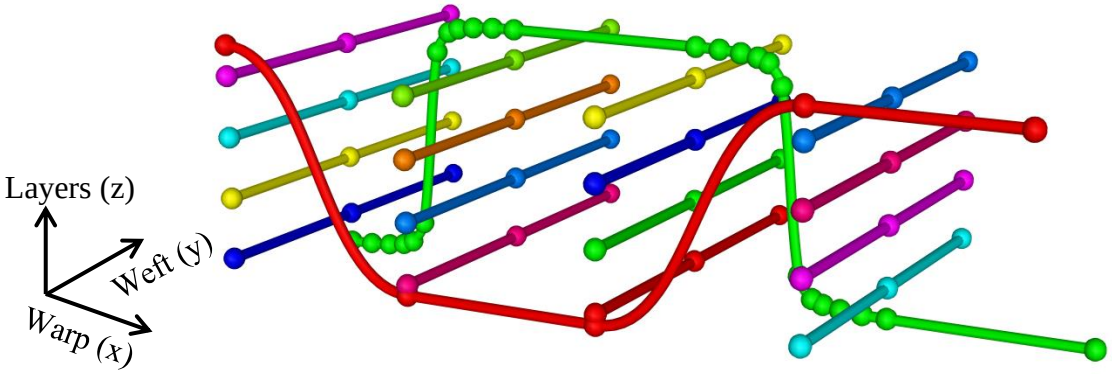


Figure 22: Difference between warp and binder (*x*) yarns with a similar path – note the different number of nodes at the yarn intersections: green yarn defined as binder yarn, red: yarn defined as warp yarn. The additional nodes of the binder yarn allow better refinement of the binder path.

When a textile is required to have a certain thickness, $t_{textile}$, it is possible to use the refinement option in TexGen which automatically adjusts yarn dimensions (width and

height) and shape in order to achieve targeted thickness. The algorithm aims to keep intra-yarn fibre volume fraction unchanged where possible but, where necessary in order to achieve the target thickness, the dimensions of the yarn cross-sections may be changed such that the volume fraction is increased. Checks are made to ensure that realistic volume fractions are maintained. If this is not possible an error message will be generated when the script is executed.

Textile.SetThickness($t_{textile}$)

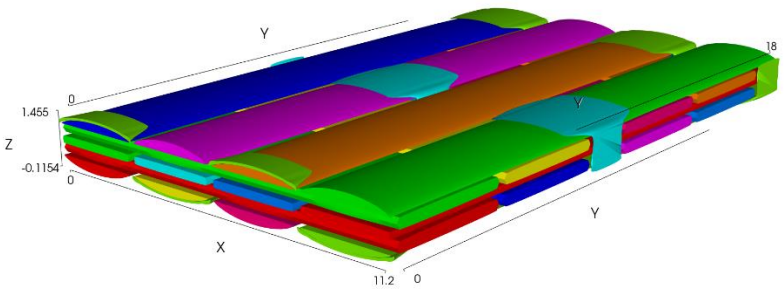
Note that SetThickness() requires the refinement option *ref* to be set to True and for yarn properties to be set.

Example script to create a 3D orthogonal weave with 4 warp and 2 binder yarns in *x*-direction, 4 weft yarns and 3 weft layers

```
# Orthogonal.py
# Create an orthogonal textile with 6 warp and 4 weft yarns, 1.0 warp and weft spacing,
# warp height 0.35 and weft height 0.25. True/False selects refinement
Textile = CTextileOrthogonal( 6, 4, 1.0, 1.0, 0.35, 0.25, True)

# Set the ratio of warp/binder yarns
Textile.SetWarpRatio( 2 )
Textile.SetBinderRatio( 1 )

# Add yarn layers.
# Set up numbers of warp and weft layers
NumWarpLayers = 2
NumWeftLayers = 3
Textile.SetupLayers( NumWarpLayers, NumWeftLayers )

# Alternatively the layers can be added manually if a different configuration
# to the standard alternating warp and weft layers is required. There must
#  always be one NoYarn layer and one Binder layer
# Textile.AddNoYarnLayer()
# Textile.AddYLayers()
# Textile.AddWarpLayer()
# Textile.AddYLayers()
# Textile.AddWarpLayer()
# Textile.AddYLayers()
# Textile.AddBinderLayer()

# Adjust the yarn widths, heights and spacings
Textile.SetWarpYarnWidths( 3.6 )
Textile.SetWarpYarnHeights( 0.35 )
Textile.SetWarpYarnSpacings( 3.8 )

Textile.SetBinderYarnWidths( 1.375 )
Textile.SetBinderYarnHeights( 0.16 )
Textile.SetBinderYarnSpacings( 1.4 )
```

```
# Weft yarns
Textile.SetYYarnWidths(2.58)
Textile.SetYYarnSpacings(2.8)

# Set the power of the power ellipses used
Textile.SetWarpYarnPower(0.6)
Textile.SetWeftYarnPower(0.6)
Textile.SetBinderYarnPower(0.8)

# Binder yarns may only be at top/bottom position in orthogonal
# In this case the binder layer was created as the top layer so switching will place them at the bottom
Textile.SwapBinderPosition( 0,2)
Textile.SwapBinderPosition( 2,2)
Textile.SwapBinderPosition( 1,5)
Textile.SwapBinderPosition( 3,5)

# Create a default domain to fit the textile
Textile.AssignDefaultDomain()

# Set fibre properties
Textile.SetFibreDiameter( WARP, 0.007, "mm" )
Textile.SetFibreDiameter( WEFT, 0.007, "mm" )
Textile.SetFibreDiameter( BINDER, 0.007, "mm" )
Textile.SetFibresPerYarn( WARP, 5000 )
Textile.SetFibresPerYarn( WEFT, 8000 )
Textile.SetFibresPerYarn( BINDER, 3500 )
# Set the target thickness for the refined textile
Textile.SetThickness(1.4)
# Set the maximum volume fraction allowed in the refine process
Textile.SetMaxVolFraction(0.78)
# Reset the domain height to suit the refined textile
Textile.SetDomainZValues()

# Add the textile
AddTextile(Textile)
```

The resulting textile and yarn path is shown in Figure 23Figure 23.



A)

B)

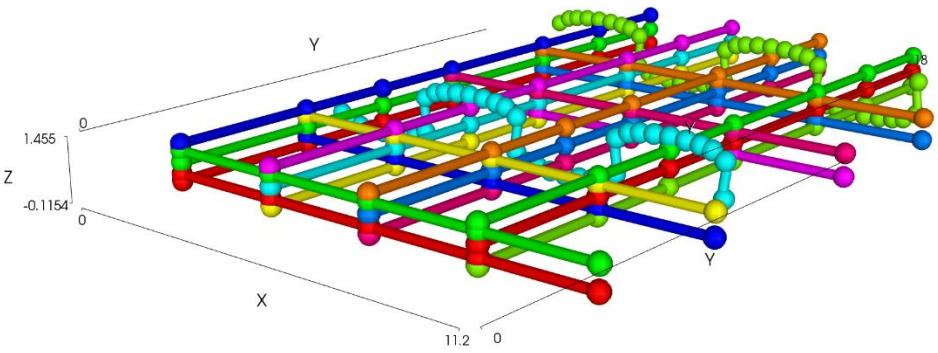


Figure 23: Example 3D weave. A) The geometrical model and B) the yarn paths and nodes.

7.4.2. Create 3D Layer To Layer Textiles

Layer to layer textiles consist of vertically stacked 90° weft (*y*) yarns and 0° warp (*x*) yarns, held together by a single or a stack of binder (*x*) yarns. For this, the CTextileLayerToLayer() class is used to create a textile.

```
weave = CTextileLayerToLayer(nx, ny, sx, sy, hx, hy, nbl)
```

For this textile, the following parameters need to be specified:
n_x – number of x (warp & binder) yarns
n_y – number of y (weft) yarns
s_x – x (warp / binder) yarn spacing
s_y – y (weft) yarn spacing
h_x – x (warp / binder) yarn heights
h_y – y (weft) yarn heights
n_{bl} – number of binder layers

The ratio of warp to binder yarns should be specified as described in Section 7.4 3D Textiles.

```
weave.SetWarpRatio( r_warp )
weave.SetBinderRatio( r_bind )
```

The number of layers of warp and weft yarns are then specified and the layers set up as described for the orthogonal yarn in Section 5.4.1. In this case the SetupLayers function also specifies the number of binder layers:

```
weave.SetupLayers( nwarp, nweft, nbl )
```

Here n_{warp} and n_{weft} are the number of warp and weft layers, respectively. Whereas n_{weft} can be chosen as desired, n_{warp} and n_{bl} are limited to the following ranges:
$n_{warp} = n_{weft} - 1$
and
$1 \leq n_{bl} \leq n_{weft}$

Adding a textile set-up with the commands described above will result in a textile geometry such as the one shown in Figure 24: Example 3D textile geometry created with: $n_x = 2$, $n_y = 4$, $s_x = 1.$, $s_y = 1$, $h_x = 0.1$, $h_y = 0.2$,Figure 24. This initial lay-up can

then be further modified, e.g. the interlacing path of the binders be set or the yarn geometries be adjusted.

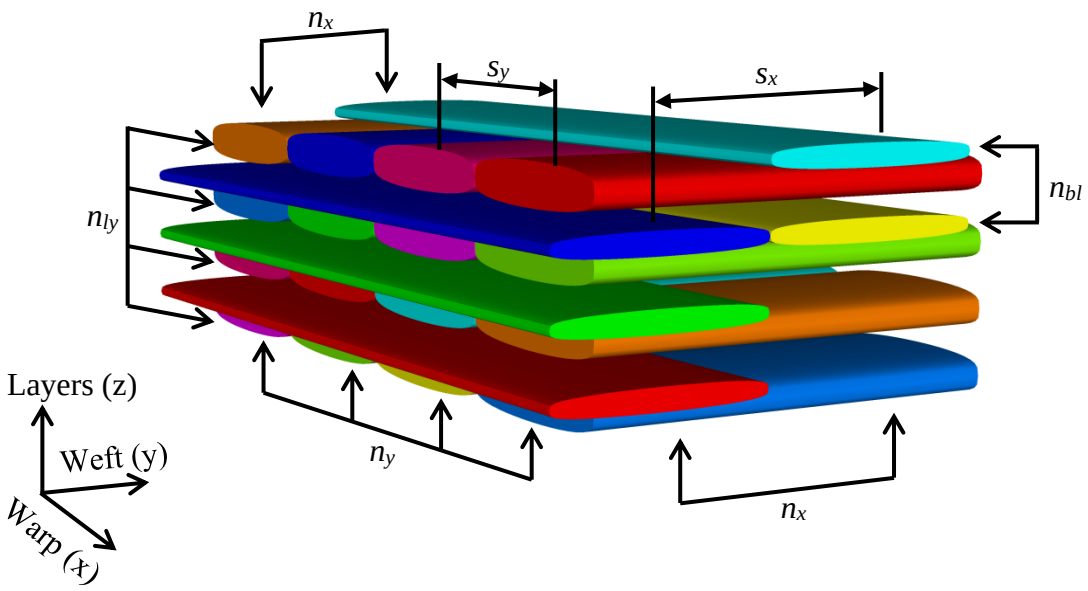


Figure 24: Example 3D textile geometry created with: n_x = 2, n_y = 4, s_x = 1., s_y = 1, h_x = 0.1, h_y = 0.2, n_{bl} = 2 and n_{weft}=4

The yarn dimensions and shape can be set in the same way as for the orthogonal textile:

```
weave.SetYYarnWidths( wy )
weave.SetBinderYarnWidths(  wb )
weave.SetBinderYarnHeights( hb )
weave.SetWarpYarnPower( pwarp )
weave.SetWeftYarnPower( pweft )
weave.SetBinderYarnPower( pbind )
```

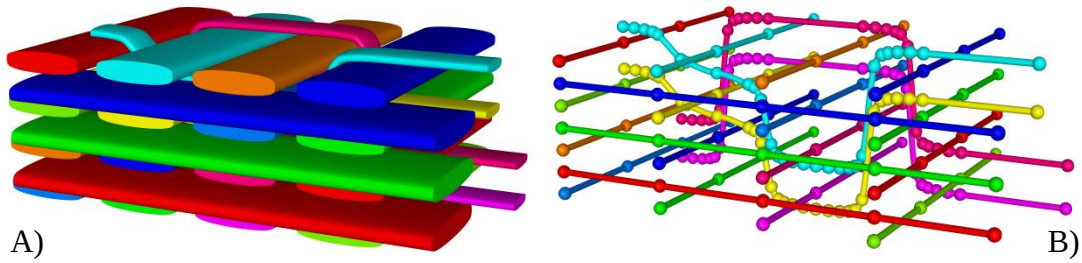
To create interlacing binder yarns the vertical position of the top binder at a given x,y position can be specified. The offset from the top of the textile is specified in terms of the possible binder locations not the actual grid coordinate, an offset of 0 being at the top and 1 the next position down, underneath the top weft layer. The maximum offset is governed by the number of weft layers and number of binder layers, for example in Figure 24: Example 3D textile geometry created with: n_x = 2, n_y = 4, s_x = 1., s_y = 1, h_x = 0.1, h_y = 0.2, the maximum possible offset is 3 which would result in the lower binder yarn being at grid position 0.

```
weave.SetBinderPosition(ix, iy, offsetz)
```

Example script to create a 3D weave with 1 warp and, 2 binder yarns in *x*-direction. The binder layers consist out of 2 vertically stacked yarns. The 3D weave consist out of 4 weft layers. The binder yarn path is adjusted to create an interlock and the yarn widths and heights are set.

```
# LayerToLayerTextile.py
#create textile
NumBinderLayers = 2
NumXYarns = 3
NumYYarns = 4
XSpacing = 1.0
YSpacing = 1.0
XHeight = 0.2
YHeight = 0.2

weave = CTextileLayerToLayer(NumXYarns, NumYYarns,XSpacing,
            YSpacing, XHeight, YHeight, NumBinderLayers)

#set number of binder / warp yarns
NumBinderYarns = 2
NumWarpYarns = NumXYarns - NumBinderYarns
weave.SetWarpRatio( NumWarpYarns )
weave.SetBinderRatio( NumBinderYarns )

#setup layers: 3 warp, 4 weft
weave.SetupLayers( 3, 4, NumBinderLayers)

#set yarn dimensions: widths / heights
weave.SetYYarnWidths( 0.8 )
weave.SetYYarnWidths( 0.8 )
weave.SetBinderYarnWidths( 0.4 )
weave.SetBinderYarnHeights( 0.1 )

#define offsets for the two binder yarns
P = [[0, 1, 3, 0],[3, 0, 0, 3]]

#assign the z-positions to the binder yarns
for y in range(nwarp,NumXYarns): #loop through number of binder yarns
        offset = 0
        for x in range(0,NumYYarns): #loop through the node positions
                weave.SetBinderPosition(x, y, P[y-NumWarpYarns][offset])
                offset = offset+1

# Add textile to database
AddTextile('test3D', weave)
```

The resulting textile and yarn path are shown in Figure 25: Example 3D weave. A) The geometrical model and B) the yarn paths and nodes.Figure 25.



A) B)

Figure 25: Example 3D weave. A) The geometrical model and B) the yarn paths and nodes.

7.4.4 Create Decoupled Layer to Layer Textiles

In Section 7.4.3 layer to layer textiles were described where the binder yarns in a given stack are 'tied' together and follow the same pattern through the weft stacks. In the CTextileDecoupledLToL class the binder yarns at a given warp position can follow different paths through the weft stacks with the constraints that they cannot cross each other and that there much always be at least one weft yarn between the binder yarns.

```
Textile = CTextileDecoupledLTo(nx, ny, sx, sy, hx, hy, nbl)
```

For this textile, the following parameters need to be specified:
n_x – number of x (warp & binder) yarns
n_y – number of y (weft) yarns
s_x – x (warp / binder) yarn spacing
s_y – y (weft) yarn spacing
h_x – x (warp / binder) yarn heights
h_y – y (weft) yarn heights
n_{bl} – number of binder layers

The ratio of warp to binder yarns and warp and weft layers are set up in the same way as for the layer to layer textile in section 7.4.3.

The initial lay-up can then be further modified, setting the paths of the binder yarns through the textile. To create interlacing binder yarns the vertical position of the binder yarns at a given x,y position can be specified using an array of offsets, one for each binder in the stack. The offset from the top of the textile is specified in terms of the possible binder locations not the actual grid coordinate, an offset of 0 being at the top and 1 the next position down, underneath the top weft layer. The maximum offset is governed by the number of weft layers and number of binder layers as for the layer to layer textile

```
weave.SetBinderPosition(ix, iy, offset_arrayz)
```

Example script to create a 3D weave with 2 warps and, 2 binder yarns in *x*-direction. There are two binder yarns in each binder stack. The 3D weave consists of 3 weft layers. The binder yarn path is specified by a 3 dimensional array of binder positions and the yarn widths and heights are set.

```
# DecoupledLToLTextile.py

#Set up 3D Weave textile
```

```
numBinderLayers = 2
numXYarns = 4
numWefts = 6

warpSpacing = 1.42
weftSpacing = 1.66
warpHeight = 0.3
weftHeight = 0.3

Textile = CTextileDecoupledLToL( numXYarns, numWefts, warpSpacing,
weftSpacing, warpHeight, weftHeight, numBinderLayers, True)

#set number of binder / warp yarns
NumBinderYarns = 2

BinderRatio = 1
WarpRatio = 1
Textile.SetWarpRatio( WarpRatio )
Textile.SetBinderRatio( BinderRatio )

# Set up layers: 2 warp, 3 weft
Textile.SetupLayers( 2, 3, numBinderLayers)

# Define offsets for the two binder yarns
binderYarns = [[[0, 1, 1, 2, 1, 0],
             [3, 2, 3, 3, 2, 3]],
            [[1, 0, 1, 0, 2, 1],
            [2, 3, 2, 2, 3, 2]]]

#Check if length of binderYarns positions equal to numWefts
for z in range(NumBinderYarns):
    for y in range( numBinderLayers ):
        if len(binderYarns[z][y]) != numWefts:
            raise Exception("Too many binder yarn positions specified, must be
equal to number of wefts.")

binderWidth = 1.2
binderHeight = 0.3
warpWidth = 1.2
weftWidth = 1.2

Textile.SetYYarnWidths( weftWidth )
Textile.SetXYarnWidths( warpWidth )
Textile.SetBinderYarnWidths( binderWidth )
Textile.SetBinderYarnHeights( binderHeight )
Textile.SetBinderYarnPower( 0.2 )
Textile.SetWarpYarnPower(1.0)
Textile.SetWeftYarnPower(1.0)

#Decompose binder yarn offsets into stacks

repeat = BinderRatio + WarpRatio
```

```
# Loop for the number of binder yarn stacks
for z in range(NumBinderYarns):
    # Loop through the weft stacks
    for x in range( numWefts ):
        zOffsets = IntVector()
        # Loop through binder layers
        for y in range( numBinderLayers):
            list.append( int(binderYarns[z][y][x]))
        # Calculate the binder y position (ie warp yarn index)
        ind = z/BinderRatio
        BinderIndex = WarpRatio + (ind * repeat) + z%BinderRatio
        Textile.SetBinderPosition(x, int(BinderIndex), zOffsets)

Textile.SetWeftRepeat( True )
Textile.AssignDefaultDomain( )

Textile.SetFibresPerYarn(WARP, 12000)
Textile.SetFibresPerYarn(WEFT, 12000)
Textile.SetFibresPerYarn(BINDER, 12000)
Textile.SetFibreDiameter(WARP, 0.0026, "mm")
Textile.SetFibreDiameter(WEFT, 0.0026, "mm")
Textile.SetFibreDiameter(BINDER, 0.0026, "mm")

Textile.BuildTextile()
AddTextile( Textile )
```

The resulting textile and yarn path are shown in Figure 25: Example 3D weave. A) The geometrical model and B) the yarn paths and nodes.Figure 25.

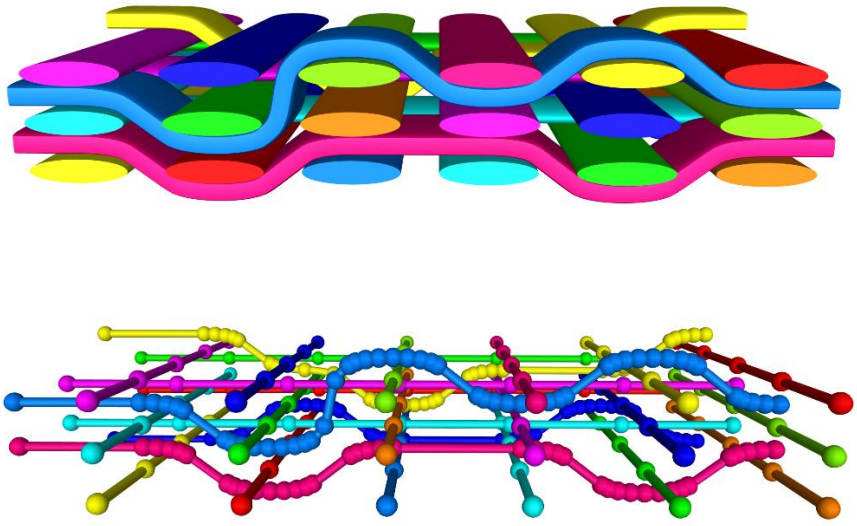


Figure 26: Example decoupled layer to layer weave. A) The geometrical model and B) the yarn paths and nodes.

7.4.5 Create Angle Interlock Textiles

8 Rotate Textiles

To rotate the textile, use the following command:

```
weave.Rotate( WXYZ(XYZ(xr,yr,zr), θ), XYZ(x0,y0,z0) )
```

Here, x_0, y_0, and z_0 give the coordinates of the centre of rotation.The coordinates x_r, y_r and z_r specify the axis of rotation. For example, setting x_r=y_r=0 and z_r=1 specifies a rotation around the z - axis. The rotation angle, θ, is specified in radians. To convert the angle in degrees to radians, use:
angle=math.radians(θ)For textiles specified using the weave wizard the nodes will have been created with the SetInPlaneTangents flag set. For rotations about axes other than the z-axis this needs to be set to False in order for the yarns to retain the expected paths. This is done using

```
weave.SetInPlaneTangents( False )
```

In order to rotate the domain as well as the textile this must be done in a separate operation:

```
Domain = weave.GetDomain()
Domain.Rotate( WXYZ(XYZ(0,0,1), math.radians(45)))
```

9 Saving TexGen Models

TexGen models can be saved as .tg3 files which are text files in XML format.

```
SaveToXML(fname+".tg3", textile, out_type)
```

The parameters are:
fname - a string of the file name and optionally preceded by the path.
textile - string of the textile name, this can be obtained using GetTextile( )
out_type - keyword defining the output style. Possible keywords are

- OUTPUT_MINIMAL – for automatically generated textiles saves only minimal information from which yarns can be built
- OUTPUT_STANDARD – saves both textile and yarn data (nodes and cross-sections)
- OUTPUT_FULL – saves textile, yarn and mesh data (includes cross-section coordinates for each slave node)

A saved .tg3 file can be loaded from file using:

```
ReadFromXML( fname +".tg3")
```

10 Mesh Generation and Export

There are several methods of mesh generation built into TexGen. These all use the CMesh class which stores a set of nodes and elements.

The element types available are:
TRI, 3 nodes
QUAD, 4 nodes
TET, 4 nodes
PYRAMID, 5 nodes
WEDGE, 6 nodes
HEX, 8 nodes
LINE, 2 nodes
POLYLINE, 2 nodes
QUADRATIC_TET, 10 nodes
POLYGON, number of nodes is variable, start node must be the same as the end node

10.1. Export surface mesh

Surface meshes of textiles can be exported in the following formats:

- VTK unstructured grid[3] (.vtu)
- ASCII STL[4] (.stl)
- Binary STL (.stl)
- 3D points (.pts)

The yarn resolution defines the surface mesh resolution and may be set using the Yarn.SetResolution() function (Section 2.2.1 Set Resolution).

The surface mesh is generated by creating a CMesh object. The textile AddSurfaceToMesh function is then used to create the surface mesh for all yarns in the textile and store the nodes and elements in the mesh object. The elements used in the surface mesh are 4 noded QUAD elements.

```
textile = GetTextile()
mesh = CMesh()

TrimDomain = True
textile.AddSurfaceToMesh( mesh, TrimDomain )

# The outline of the domain can also be added to the mesh
mesh.InsertMesh(textile.GetDomain().GetMesh())
```

The Boolean variable, *TrimDomain*, is set to specify whether or not the yarns are trimmed to the domain dimensions. If not then the mesh is generated for the entire yarns.

[3] www.vtk.org/
[4] https://en.wikipedia.org/wiki/STL_(file_format)

The surface mesh can then be exported as one of the available file types. The second parameter in SaveToSTL defines whether the file is saved as binary (True) or ASCII (false).

```
mesh.SaveToVTK( FileName )                  # Adds .vtu extension
mesh.SaveToSTL( FileName.stl, isbinary )    # Adds .stl extension
mesh.SaveToSCIRun( FileName )               # Adds .pst extension
```

10.2 Dry Fibre Export

A conformal mesh of just the yarns can be created using 6 noded wedge and 8 noded hex elements. The theory behind the creation of this mesh is described by Sherburn[2]. In this case the mesh generation is done automatically by the CreateAbaqusInputFile function. The mesh size is defined by the yarn resolution.

The mesh to be generated by the dry fibre export can be previewed in the GUI by rendering the textile volume. If required the number of mesh layers can be changed from the default by specifying in the section mesh:

```
# DryTextileExport.py
# Generates conformal mesh of just the yarns and
# creates ABAQUS input file
weave = GetTextile().GetWeave()

# Set the number of layers in the section mesh
# First parameter specifies number of layers (must be even number)
# Second parameter defaults to True to give triangular elements at corners
# producing wedge elements. False to force all 4 noded elements,
# producing all hex elements
sectionMesh = CSectionMeshRectangular (4,True)
weave.AssignSectionMesh(sectionMesh)
```

Note that because of the limitations in the Python interface for accessing inherited classes from base classes it is necessary to call the GetWeave function in order to be able to access the weave functions not contained in the base class (ie in Ctextile). For a textile created using one of the 3D weave classes use

```
weave = GetTextile().Get3Dweave()
```

Deformation in x,y and z directions are specified and the CsimulationAbaqus object is created which will generate the ABAQUS input file:

```
# Set up any x,y,z transformations
tension = ClinearTransformation()
tension.AddScale( 1.1, 1, 1)

# Setup class which generates ABAQUS files with conformal wedge/hex
elements
deformer = CsimulationAbaqus()
deformer.SetIncludePlates( True )  # Whether to include compression plates
deformer.AddDeformationStep(tension)
```

Finally the input file is created. Correction for small intersections (smaller than the depth of one mesh element) can be carried out by setting the *bAdjustIntersections* parameter to True. If this is carried out the changes will only be present in the exported mesh. To change the TexGen model set the *bRegenerateMesh* parameter to True. The element type can be specified as C3D8 or C3D8R by setting the *bElementType* parameter to True or False respectively.

```
fName = 'ExportDry'
bRegenerateMesh = False;  # True if want to regenerate mesh after
intersection correction
bElementType = False   # True for C3D8, False for C3D8R
bAdjustIntersections = False
Tolerance = 1e-6;
deformer.CreateAbaqusInputFile(textile, fName, bRegenerateMesh ,
bElementType, bAdjustIntersections, Tolerance )
```

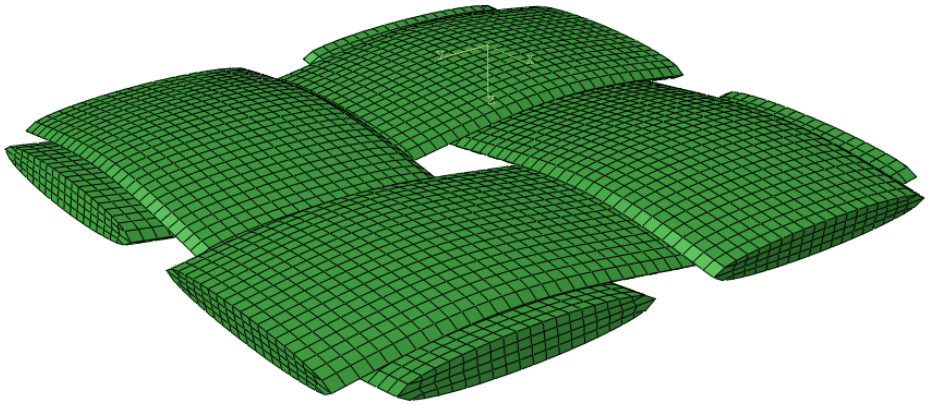


Figure 27: Dry fibre export as displayed in ABAQUS showing the four mesh layers

10.3 Voxel Mesh Export

A voxel mesh can be generated for a textile by splitting the domain into equal sized brick (hex) elements (voxels). The voxels are assigned either to one of the yarns or the matrix based on the centre point of the voxel. The voxel size should be selected such that it will capture all elements of the geometry, for example a 3D textile with fine binder yarns will need a voxel size small enough to ensure continuity of voxels in the binder yarn.
A voxel mesh, *VoxMesh*, can be created in TexGen by using the voxel mesh class

```
VoxMesh = CRectangularVoxelMesh( boundaries )
```

The *boundaries* parameter specifies which boundary condition class is to be used. In the case of a simply repeating unit cell this is the CPeriodicBoundaries class. The alternative CShearedPeriodicBoundaries and CStaggeredPeriodicBoundaries will be described in Sections 10.3.2 Sheared Voxel Mesh. and 10.3.1 Exploiting "Staggered" Boundary Conditions. The parameter can either be passed as a string 'CPeriodicBoundaries' or, in this case, omitted as this is the default.

The voxel mesh is then generated and the mesh and boundary conditions exported as an ABAQUS .inp file using the SaveVoxelMesh function:

```
VoxMesh.SaveVoxelMesh( Textile, fname, vx, vy, vz, out_matrix,
                                    out_yarns, boundary_conditions, etype)
```

The input variables are:
Textile = a textile derived from the CTextile class
fname = a string of the output file name
v_x, v_y, v_z = integers defining the number of voxels in x, y and z-directions
out_matrix = True if outputting matrix elements, False if not
out_yarns = True if outputting yarn elements, False if not
boundary_conditions = One of: MATERIAL_CONTINUUM
SINGLE_LAYER_RVE
STAGGERED_BC
SHEARED_BC
ROTATED_BC
NO_BOUNDARY_CONDITIONS
etype = Integer specifying the voxel (brick) element type to be used
0 = C3D8R elements[5]
1 = C3D8 elements[6]

This function will save three files with the same name, *fname*. The
.inp file which contains node and element information as well as node sets and boundary conditions.
.ori file which contains the orientation data of the elements.
.eld file which contains fibre volume fraction information

Example: Create a voxel mesh of the single yarn in the example of Section 2.3. Create Master Nodes
The voxel mesh will consist out of 20x20x20 elements and only the yarn will be written as output in the .inp file for Abaqus, i.e. in this case matrix elements are not output.

```
# VoxelExport.py
# Create a single repeating yarn with a domain
# and export the voxel mesh as an ABAQUS .inp file
# with periodic boundary conditions

# Create a yarn with repeats and add to a textile
textile = CTextile( )

yarn = CYarn( )
yarn.AddNode( CNode(XYZ(0, 0.0, 0)) )
yarn.AddNode( CNode(XYZ(1, 3.5, 0)) )
yarn.AddNode( CNode(XYZ(0, 7.0, 0)) )
```

[5] Eight-node brick element with reduced integration:
http://web.mit.edu/calculix_v2.7/CalculiX/ccx_2.7/doc/ccx/node27.html
[6] Eight-node brick element
http://web.mit.edu/calculix_v2.7/CalculiX/ccx_2.7/doc/ccx/node26.html

```
#  Add repeats
yarn.AddRepeat( XYZ(3, 0, 0) )
yarn.AddRepeat( XYZ(0, 7, 0) )
yarn.AddRepeat( XYZ(0.5, 0, 1) )

textile.AddYarn(yarn)

# Assign a domain using a min/max box
textile.AssignDomain(CDomainPlanes(XYZ(-0.5, 0, -0.5), XYZ(1.5, 7, 0.5)))

# AddTextile is not required if don't need to visualise yarns
AddTextile( "SingleYarn", textile)

#create a voxel mesh with the default boundaries 'CPeriodicBoundaries'
VoxMesh = CRectangularVoxelMesh()

vx=20
vy=20
vz=20

VoxMesh.SaveVoxelMesh(textile, "test_out.inp", vx, vy, vz, False, True,
MATERIAL_CONTINUUM, 0 )
```

The outputs for different v_x, v_y and v_z combinations are shown in Figure 28: The yarn from the example in Chapter 2 meshed with different voxel sizes and imported in to Abaqus CAE. A) Voxel mesh v_x=20, v_y=20 and v_z =20. B) A voxel mesh with v_x=40, v_y=40 and v_z=20..

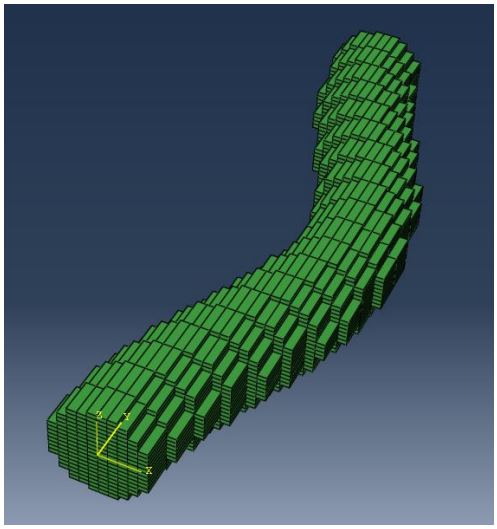
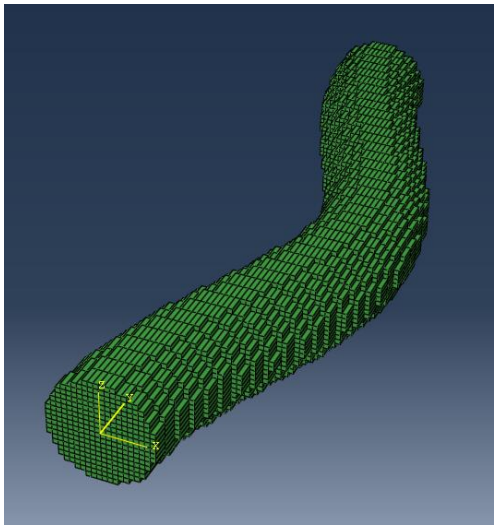
A)   B)

Figure 28: The yarn from the example in Chapter 2 meshed with different voxel sizes and imported in to Abaqus CAE. A) Voxel mesh v_x=20, v_y=20 and v_z =20. B) A voxel mesh with v_x=40, v_y=40 and v_z=20.

10.3.1 Exploiting "Staggered" Boundary Conditions[7]

The repetitive pattern in a textile may exhibit more than one case of translational symmetry. The simplest case is when the periodicity vector is equal to the size of the unit cell. Another case is the so-called staggered pattern when in one direction a (reduced) unit cell has to be translated by the full length and in the second direction it can be translated by less than the full length. An example of such a pattern is shown in Figure 29: Reduction of the unit cell of an orthogonal textile. This approach helps to reduce the size of the model which can, in turn, reduce computational cost.

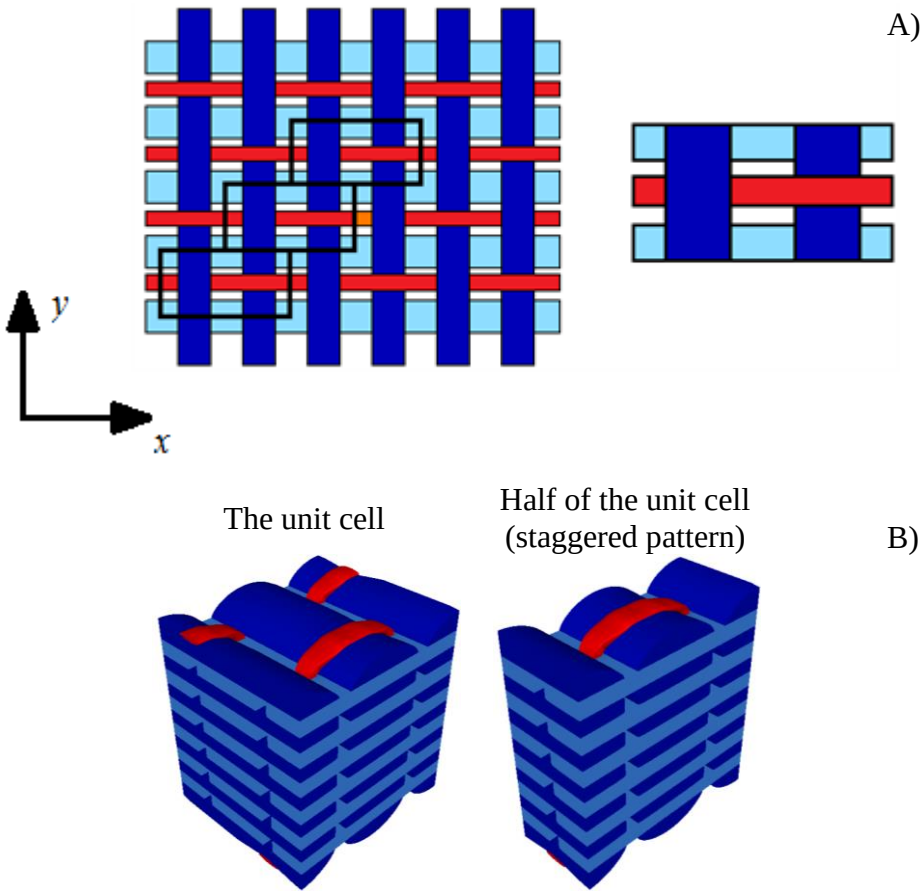


Figure 29: Reduction of the unit cell of an orthogonal textile

In order to use this feature in TexGen a reduced unit cell is created where the model has an offset in the *x*-direction as shown in Figure 29. The voxel mesh, *Vox*, needs to be created with "Staggered" boundary conditions selected and then an *XOffset* is specified. The value of *XOffset* is set to the proportion of the unit cell length in *x*-direction by which the unit cell is offset in order to achieve the correct pattern ($0 < XOffset < 1$).

```
Vox = CStaggeredVoxelMesh( 'CStaggeredPeriodicBoundaries' )
Vox.SetOffset( XOffset
```

[7] M.Y. Matveev. Effects of variabilities on mechanical properties of textile composites. PhD thesis, The University of Nottingham 2014. (Appendix F)

For the model in Figure 29B above, the value of *XOffset* is equal to 0.5 (i.e. half of the unit cell (*y*) length). Using

```
Vox.SaveVoxelMesh( Textile, fname, vx, vy, vz, out_matrix,
                                out_yarns, tbound, etype)
```

TexGen will save a voxel mesh and apply boundary conditions which link corresponding points as shown in Figure 30: Corresponding points of the 1/2 unit cell..

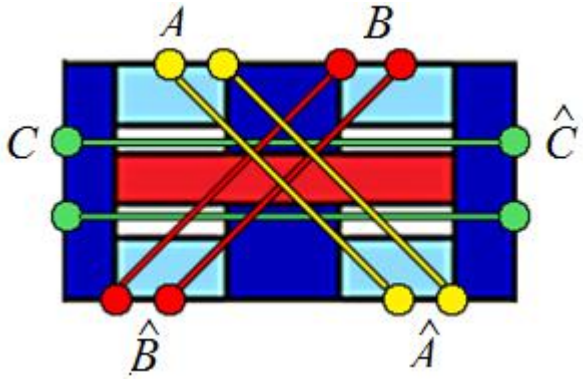


Figure 30: Corresponding points of the 1/2 unit cell.

Note that the mesh resolution, v_x, in the case of staggered voxel meshes needs be divisible without remainder by *XOffset*. This will be checked during export and if this condition is not met an error message will be displayed:
"Number of x voxels must satisfy N mod (1/offset) = 0 condition
Unable to create ABAQUS input file: Error calculating voxel sizes"

It should be noted that the staggered boundary conditions are only for displacement in *x*, *y* and *xy* direction. Any contribution in through thickness (*z*-direction) is ignored.

10.3.2 Sheared Voxel Mesh.

Where a sheared textile has been created with a sheared domain it is possible to export a sheared voxel mesh. In this case the voxels are parallepiped shapes so that they align with the sides of the domain. The boundary conditions then take into account the extra x offset required to match corresponding nodes on opposite sides of the domain due to the shear. The SaveVoxelMesh function is then called as described in Section 10.3 Voxel Mesh Export, this time using the SHEARED_BC parameter.

```
Vox = CShearedVoxelMesh( 'CShearedPeriodicBoundaries' )
Vox.SaveVoxelMesh( textile, fName, vx, vy, vz, True, False, SHEARED_BC, 0
```

An example of a sheared voxel mesh is shown in Figure 29.

.

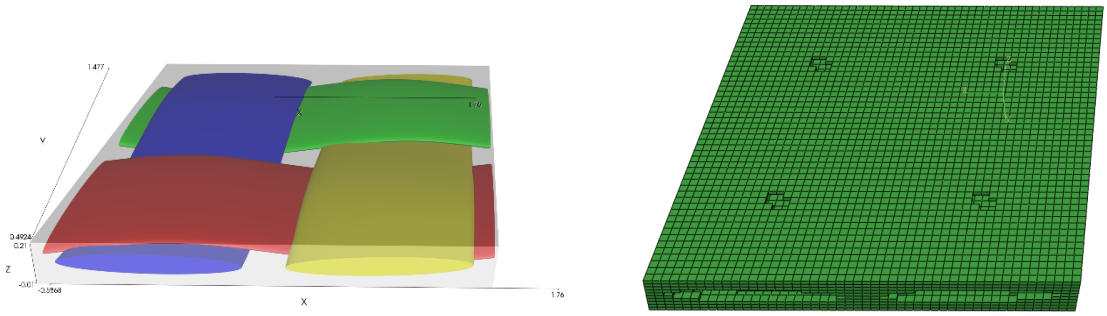


Figure 31: a) Sheared plain weave with sheared domain b) Voxel mesh export of matrix only showing voxels aligned to the sheared domain

10.3.3 Rotated Voxel Mesh

Where a textile has been created with a rotated domain a voxel mesh can be exported with voxels aligned to the rotated axes of the domain. The boundary conditions take into account the offsets required to match corresponding nodes on opposite sides of the domain. The Save VoxelMesh function is called using the ROTATED_BC parameter

```
Vox = CRotatedVoxelMesh( 'CRotateddPeriodicBoundaries' )
VoxSaveVoxelMesh( textile, fName, vx, vy, vz, True, False, ROTATED_BC, 0)
```

10.4 Voxel Mesh with Octree Refinement and Smoothing

Octree mesh refinement allows the creation of meshes with voxels of variable size. This is achieved by selective refinement of voxels which are close to yarns surfaces. These voxels are split into 8 smaller voxels which can be also be split in smaller voxels until a desired element size is achieved. This allows creation of an octree-refined voxel mesh which can give more precise results than a regular voxel mesh with the same number of elements.

In addition to the octree refinement, a smoothing algorithm can be applied to the generated octree-refined mesh. The smoothing algorithm moves the nodes on the yarns surface in order to generate a smoother surface which would not create stress concentrations. An example is shown in Figure 32.

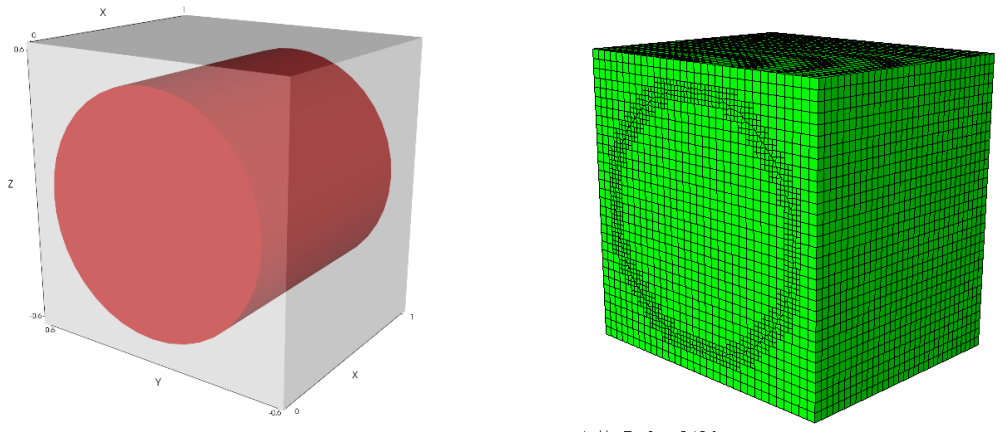


Figure 32 Voxel mesh with octree refinement and Laplacian smoothing

After creation of a textile the voxel mesh with octree refinement and smoothing is generated using the COctreeVoxelMesh class

```
Textile = GetTextile()
oc = COctreeVoxelMesh()
# Generate an octree mesh
Oc.SaveVoxelMesh(Textile, "OctreeMesh", 10, 10, 10, 3, 5, False, 50, 0.3,
0.3, …
        False, False)
# Delete the object to free memory
del oc
```

The function SaveVoxelMesh in the COctreeVoxelMesh class takes 10 arguments:

```
SaveVoxelMesh( Textile, Filename, XVoxNum, YVoxNum, ZVoxNum, Min
Refinement, Max Refinement, Smoothing, Smoothing Iterations, Smoothing
Parameter 1, Smoothing Parameter 2, Create Surfaces, Create Cohesive
Contact)
```

The arguments are:

- Textile – textile geometry to mesh
- Filename – filename for the resulting mesh (TexGen will create inp, ori and eld files)
- XVoxNum, YVoxNum, ZVoxNum – initial number of voxels in the x, y and z directions respectively. Try to choose these such that the initial voxel dimensions are all similar so as to start the refinement with voxels which are close to cubic.
- Min Refinement – minimum level of refinement. All elements will be refined at least to this level. Each level of refinement reduces the size of element by the factor of 2. This means that the maximum element size in a mesh with minimum level of refinement M will be L/2M, where L is the size of textile
- Max Refinement – maximum level of refinement. Only elements on the interface between two materials or close to it will be refined to this level of refinement.
- Smoothing – enabling the smoothing algorithm. The smoothing algorithm will move the yarn surface nodes according to the Laplacian smoothing algorithm
    - False - no smoothing
    - True - smoothing enabled
- Smoothing iterations – number of smoothing iterations
- Smoothing Parameter 1 – smoothing coefficient 1
- Smoothing Parameter 2 – smoothing coefficient 2
- Create surfaces - generate sets of nodes for surfaces between yarns and matrix
    - False - no surface sets will be generated
    - True - surfaces will be generated
- Create cohesive contact – define cohesive contact between yarns and matrix
    - False - no contact generated
    - True - nodes on the surfaces between yarns and matrix will be duplicated to "decouple" the yarns and matrix, cohesive contacts will be created

10.5 Tetrahedral Volume Mesh Export

A tetrahedral mesh (called volume mesh in TexGen) can be generated for a textile by using the volume mesh export functions. The elements are assigned either to one of the yarns or the matrix based on the centre point of the element.
The volume mesh is generated using the CMesher class. First the class is created; the *boundary_conditions* input variable is specified as for the voxel mesh export (Section 10.3 Voxel Mesh Export) but, in this case, the STAGGERED_BC option is unavailable. The default value is for no boundary conditions to be specified, in which case the variable can be omitted.

```
mesher = CMesher( boundary_conditions )
```

By default the mesh will be created to be non-periodic. If a periodic mesh is required it is set using

```
mesher.SetPeriodic(True)
```

A seed size can be set for the mesh which, effectively, sets the length of the elements along the edges of the domain. The top surface of the domain is then triangulated based on these dimensions.

```
mesher.SetSeed(0.1)
```

The mesh is then generated using the CreateMesh function and saved to either an ABAQUS input file or in VTK format:

```
mesher.CreateMesh(weave)
mesher.SaveVolumeMeshToABAQUS(fname, textile_name)
mesher.SaveVolumeMeshToVTK(fname)
```

The input variables are:
fname = a string of the output file name
textile_name = name assigned to the textile

The save to Abaqus function will save three files as described for the voxel export (Section 10.3 Voxel Mesh Export)

Example: Create a volume mesh of a plain weave textile and output as an Abaqus input file.

```
# VolumeMeshExport.py

# Specify weave parameters
nwarp=2    #Number of weft yarns in the unit cell
nweft=2    #Number of warp yarns in the unit cell
s=1        #Spacing between the yarns
t=0.1      #Thickness of the fabric (sum of two yarn heights)
ref=True   #Refine model (True/False)

# Create 2D textile
weave = CTextileWeave2D( nweft, nwarp, s, t, ref )
# Set the weave pattern
weave.SwapPosition(0, 0)
weave.SwapPosition(1, 1)

# Assign a default domain
weave.AssignDefaultDomain()

#Add to the textile database
AddTextile(weave)
# Create CMesher object and set periodicity and seed size
mesher = CMesher(MATERIAL_CONTINUUM)
mesher.SetPeriodic(True)
mesher.SetSeed(0.1)
# Generate the mesh
mesher.CreateMesh(weave)
# Save the mesh to Abaqus
mesher.SaveVolumeMeshToABAQUS("VolumeMesh.inp", weave.GetName())
```

The textile and volume mesh created are shown in Figure 33.

A) B)

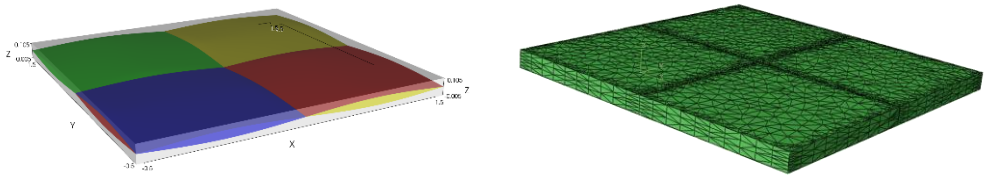


Figure 33 a) Textile for export, b) Volume mesh imported into Abaqus CAE

11 Intersection Correction

Where there are small intersections in a model these can be corrected using the CAdjustMeshInterference class. These functions were created to modify textile models where very small intersections are present which stop FE simulations from running. The algorithm uses the interference detection to identify nodes on the surface of the yarns where intersections are present and then the volume mesh is used to adjust these points. If the intersection depth is greater than the surface element in the volume mesh then the algorithm will stop and give a warning message that the intersections are too large for correction. In this case adjustment to the model is more appropriate.

The textile must have a domain specified in order to use this function.

First a CAdjustMeshInterference object is created and then the function called which corrects the intersections for the whole textile. The second parameter, *tolerance*, in the AdjustTextileMesh call may be omitted in which case the default value of 10^{-8} will be used. This gives the allowable distance between the adjusted points.

```
textile = GetTextile('textile name')
tolerance = 1e-7
Adjust = CAdjustMeshInterference()
Adjust.AdjustTextileMesh( Textile, tolerance )
```

Function Index

T

V

X

References

1. A C Long and L.P. Brown, *Modelling the geometry of textile reinforcements for composites: TexGen,* in *Composite reinforcements for optimum performance* P. Boisse, Editor. 2011, Woodhead Publishing Ltd.
2. M Sherburn, *Geometric and Mechanical Modelling of Textiles*. 2007, The University of Nottingham.
3. L P Brown, X.Z., A C Long, I A Jones. *Recent Developments in the Realistic Geometric Modelling of Textile Structures using TexGen*. in *1st International Conference on Digital Technologies for the Textile Industries*. 2013. Manchester, UK.