GeMA – MeshLib

28/11/2019 – Version 1.1

Tecgraf
PUC-RIO

MeshLib purpose

Provide auxiliary functions for:

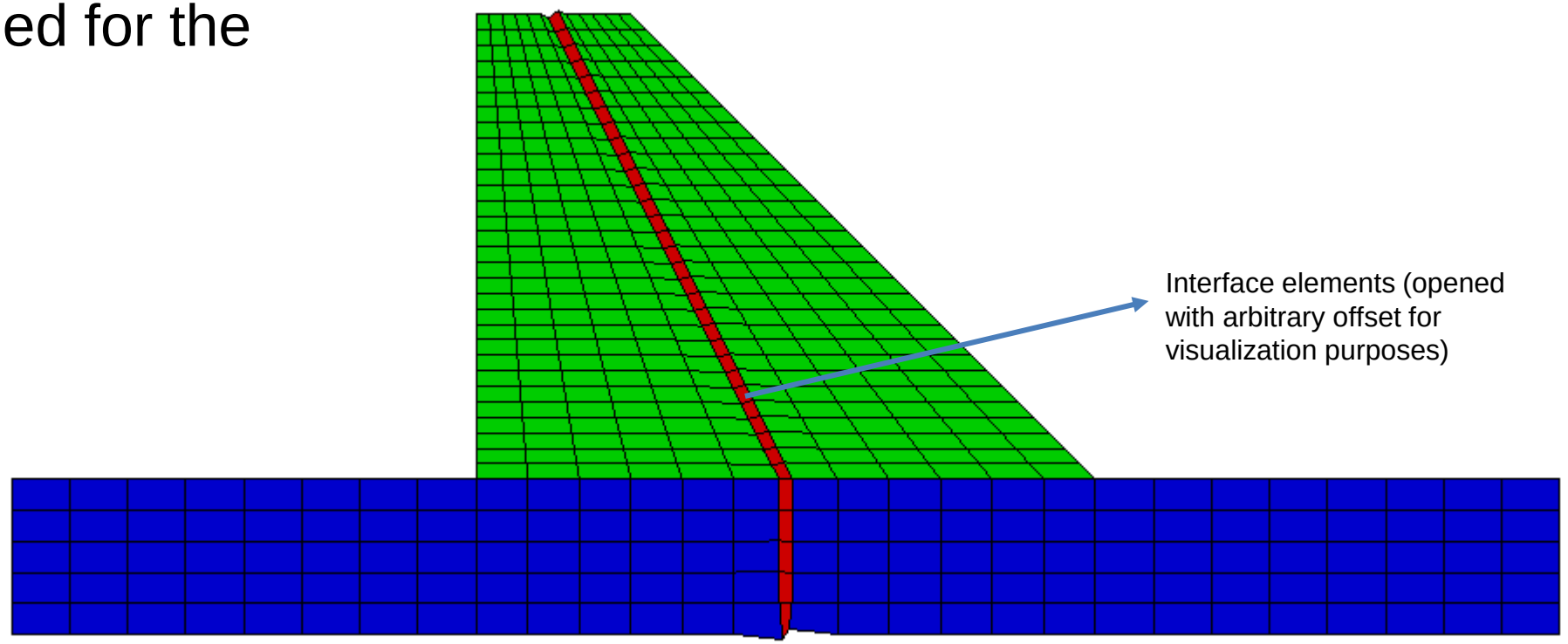
- Creating 2D and 3D, ‘grid like’, meshes.
  - Supports several element types: quad4, quad8, quad9, tri3, tri6, hex8, hex20 and hex27
  - 3D meshes are extruded from a 2D section
  - Grid can be built from several rectangular blocks
  - Function returns the set of mesh nodes, cells and borders
  - Can set materials and initial values (for node and cell attributes)
  - Supports adding interface elements (2D only)
  - Supports parametric meshes
- Adding interface elements to a 2D mesh
  - Supports any mesh, not only those built with help from meshLib
- Saving meshes to a model file
  - Model file can then be read by using the standard Lua dofile() function

I - CREATING A NEW 2D MESH

Creating a 2D mesh

meshLib.**build2DGrid**(elemType, xPoints, yPoints, blockList, matList, options)

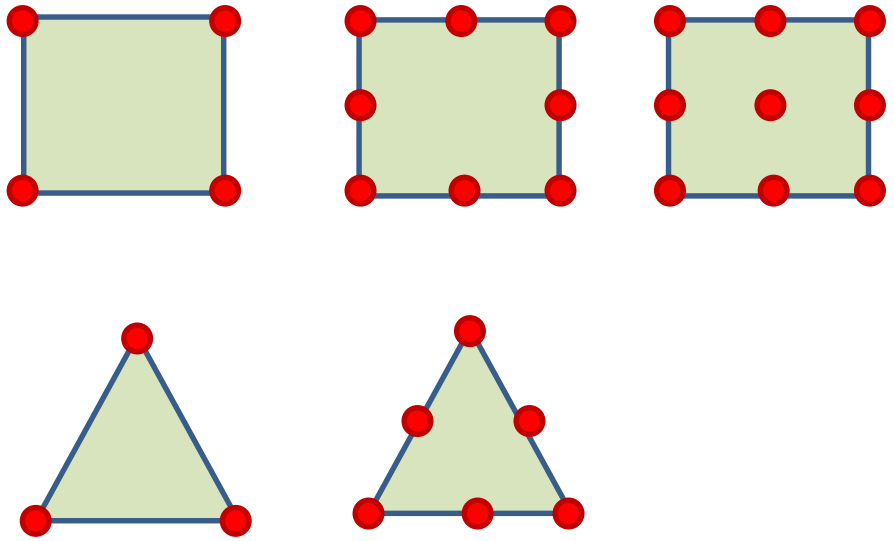
- Should be called from the model file
- Returns the set of nodes, cells and borders needed for the Mesh definition



Element types

meshLib.build2DGrid(**elemType**, xPoints, yPoints, blockList, matList, options)

- ‘quad4’
- ‘quad8’
- ‘quad9’
- ‘tri3’
- ‘tri6’



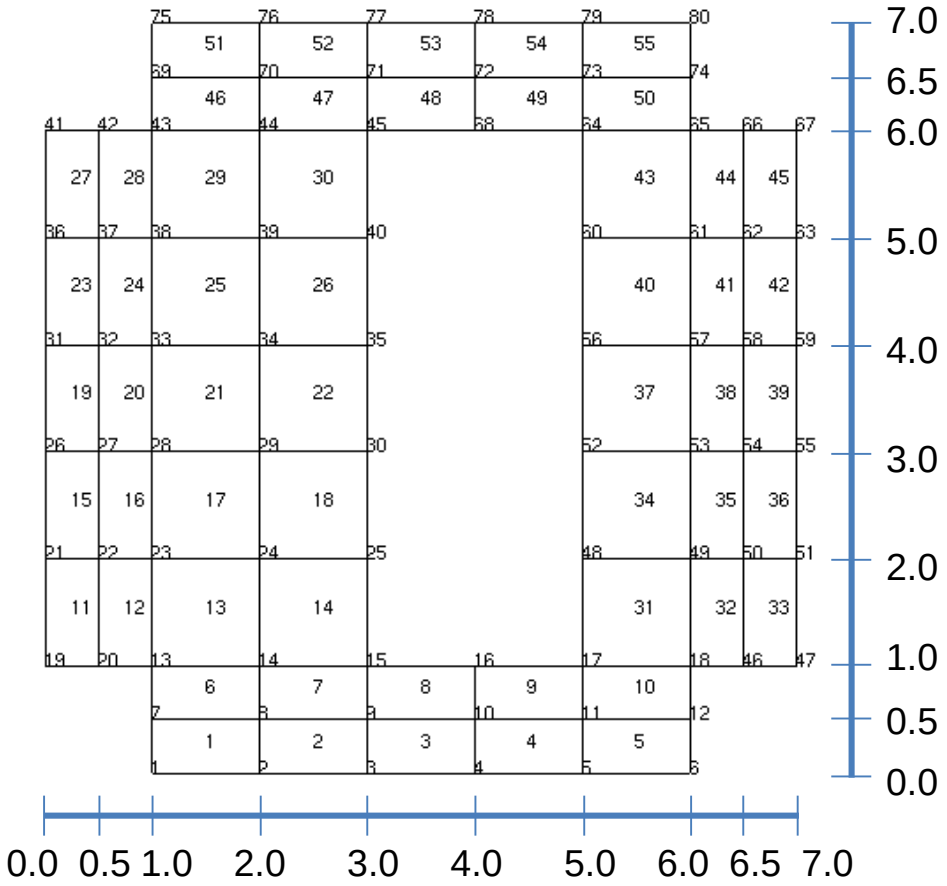
Coordinates

meshLib.build2DGrid(elemType, **xPoints**, **yPoints**, blockList, matList, options)

{0.0, 0.5, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 6.5, 7.0}

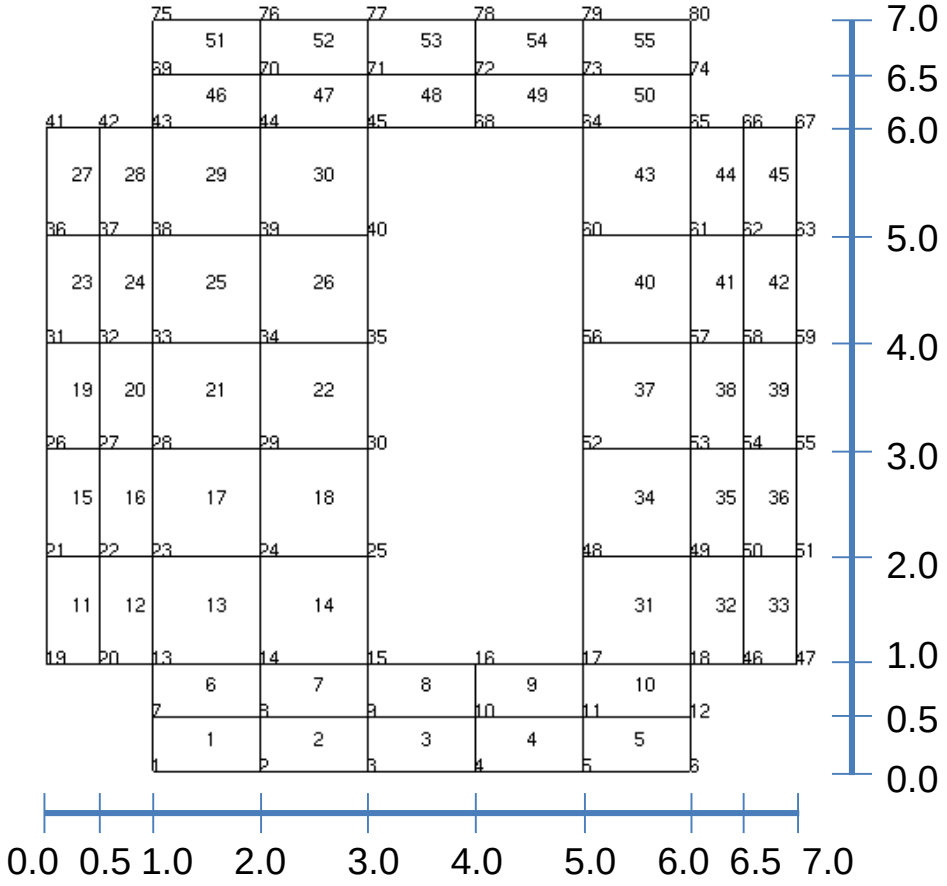
- Tables with coordinates for grid lines / columns
- Independent axis
- Any spacing
- Can be created through utilitarian functions:
  - meshLib.regularSpacing()
  - meshLIb.densitySpacing()
  - meshLib.merge()

(See examples 1, 2 and 3)



Blocks

meshLib.build2DGrid(elemType, xPoints, yPoints, **blockList**, matList, options)

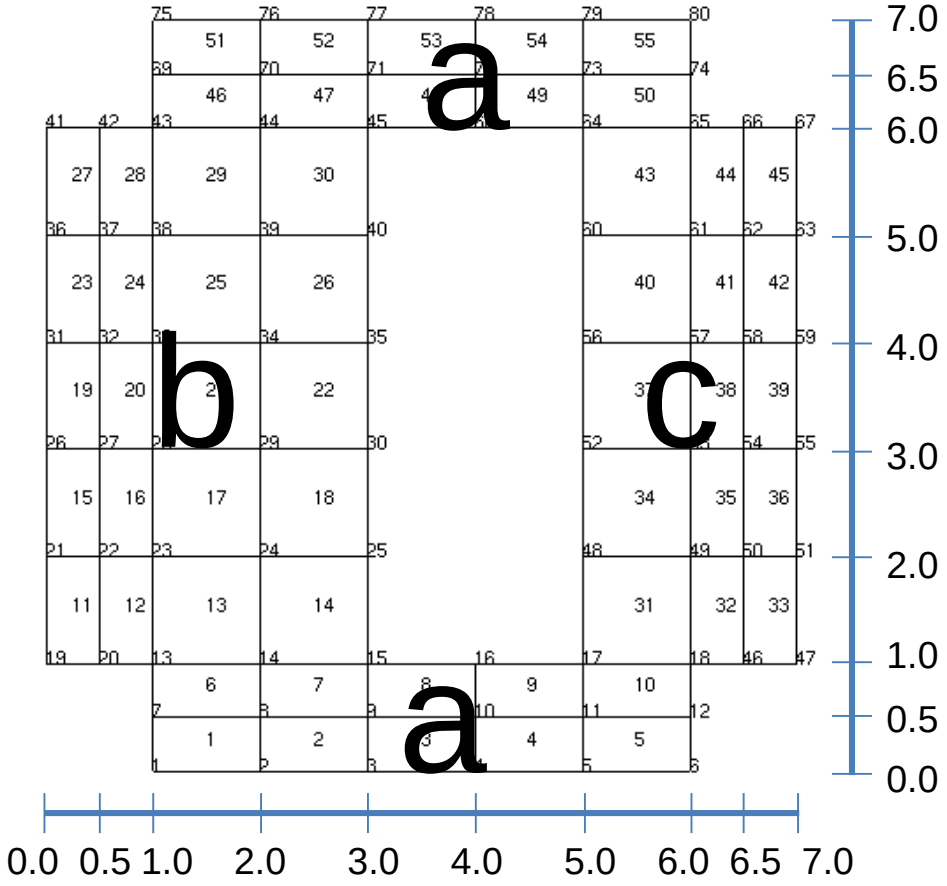



```
                Y          X
{{0.0, 1.0, {1.0, 6.0, cellGroup = 'a'}},
 {1.0, 6.0, {0.0, 3.0, cellGroup = 'b'},
                       {5.0, 7.0, cellGroup = 'c'}},
 {6.0, 7.0, {1.0, 6.0, cellGroup = 'a'}}
}
```

- For each interval in Y, specifies the X interval for each block.
- Allows the definition of cell groups and materials.
- Optional. If absent, mesh will be a single rectangular block.

Blocks

meshLib.build2DGrid(elemType, xPoints, yPoints, **blockList**, matList, options)

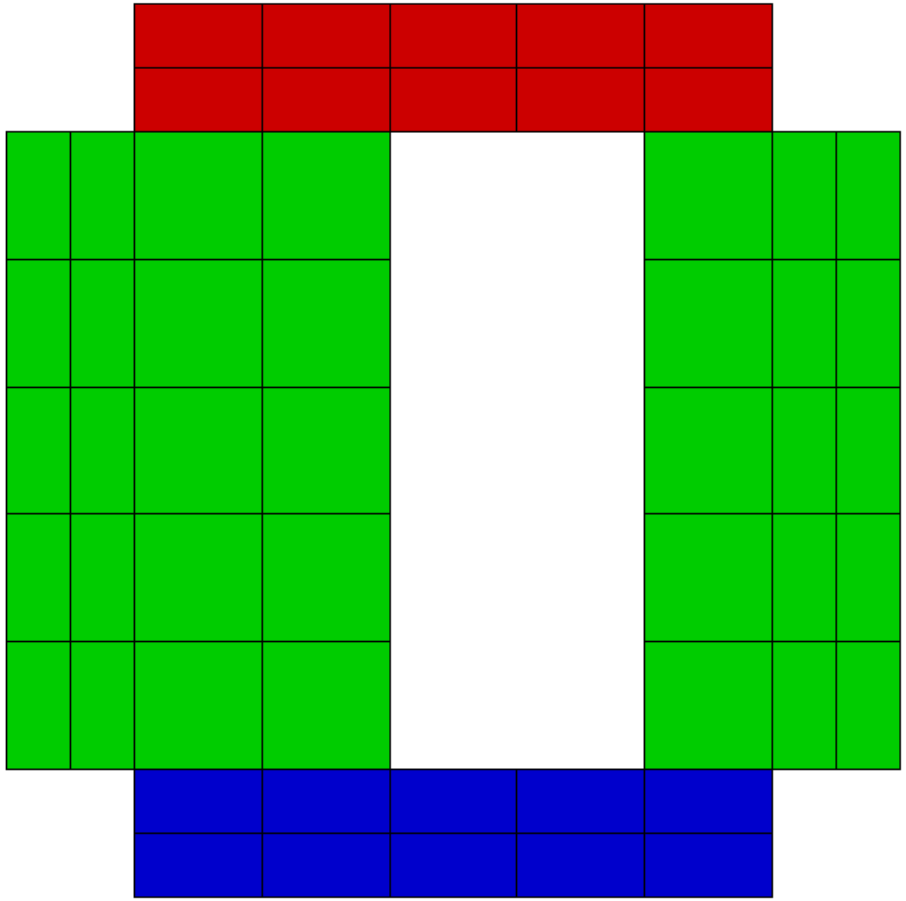



```
        Y          X
{{0.0, 1.0, {1.0, 6.0, cellGroup = 'a'}},
 {1.0, 6.0, {0.0, 3.0, cellGroup = 'b'},
            {5.0, 7.0, cellGroup = 'c'}},
 {6.0, 7.0, {1.0, 6.0, cellGroup = 'a'}}
}
```

- For each interval in Y, specifies the X interval for each block.
- Allows the definition of cell groups and materials.
- Optional. If absent, mesh will be a single rectangular block.

Materials

meshLib.build2DGrid(elemType, xPoints, yPoints, blockList, **matList**, options)



{material = 2}

- Table with property set names + defaults
- Each block can define its own material

```
{{0.0, 1.0, {1.0, 6.0, cellGroup = 'a', material = 1}},
 {1.0, 6.0, {0.0, 3.0, cellGroup = 'b'},
                {5.0, 7.0, cellGroup = 'c'}},
 {6.0, 7.0, {1.0, 6.0, cellGroup = 'a', material = 3}}
}
```

Options

meshLib.build2DGrid(elemType, xPoints, yPoints, blockList, matList, **options**)

Optional table with additional control options:

- blockf
- nodef
- cellf

User functions for value initialization (examples 7 and 15)
"nodef" can also be used for creating parametric meshes (examples 19 and 20)

- borders

Border generation (examples 5, 6, 9 and 12)

- interfaceList
- interfaceOptions

Adding interface elements (examples 10 to 15)

- useGridCoords

Coordinate kind for blockList, borders, interfaceList, midxLine and midyColumn (example 12)

- topxPoints
- midxPoints
- midxLine
- rightyPoints
- midyPoints
- midyColumn

Additional sets of coordinates for creating slanted grid rows and columns (examples 8, 9 and 12)

Examples

- The complete code for the next examples can be found on:

  meshLibExample.lua

- **Pre-requisites**: Some of the examples assume that the user is at least familiar with GeMA terminology and the standard way of manually creating meshes.
- Some examples do not show the complete code, focusing on the relevant parts / changes from previous examples. See the full code on meshLibExample.lua
- When running the example, uncomment on the orchestration the meshes that you want printed and/or saved to a file. All files are saved to the “out” sub-directory.

Example 1

A simple rectangular ‘tri3’ mesh, with X coordinates varying from 0.0 to 20.0 and Y coordinates from 0.0 to 50.0. It contains 10 element columns and 15 element rows.

```
dofile('$SCRIPTS/meshLib.lua')
```

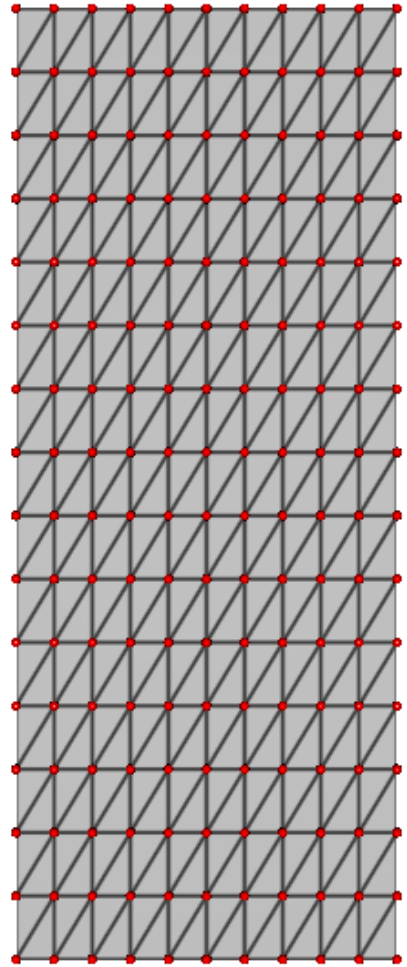
Loads the MeshLib library. Must be called before using any of its functions

```
local xPoints = meshLib.regularSpacing(0.0, 20.0, 10)
local yPoints = meshLib.regularSpacing(0.0, 50.0, 15)

local nodes, cells, borders = meshLib.build2DGrid('tri3', xPoints, yPoints)

Mesh{
  id       = 'mesh1',
  typeName = 'GemaMesh.elem',

  coordinateDim = 2,
  stateVars     = {'T'},

  nodeData = nodes,
  cellData = cells,
  boundaryEdgeData = borders
}
```

The Mesh creation code above is the same for most examples and so will be omitted unless any relevant changes are needed.

meshLib.**regularSpacing**(first, last, n)

first	The xmin / ymin coordinate
last	The xmax / ymax coordinate
n	The number of desired divisions (n elements, n+1 points)



Example 1

A simple rectangular ‘tri3’ mesh, with X coordinates varying from 0.0 to 20.0 and Y coordinates from 0.0 to 50.0. It contains 10 element columns and 15 element rows.

```
local xPoints = meshLib.regularSpacing(0.0, 20.0, 10)
local yPoints = meshLib.regularSpacing(0.0, 50.0, 15)

local nodes, cells, borders = meshLib.build2DGrid('tri3', xPoints, yPoints)

Mesh{
  id       = 'mesh1',
  typeName = 'GemaMesh.elem',

  coordinateDim = 2,
  stateVars     = {'T'},

  nodeData = nodes,
  cellData = cells,
  boundaryEdgeData = borders
}
```

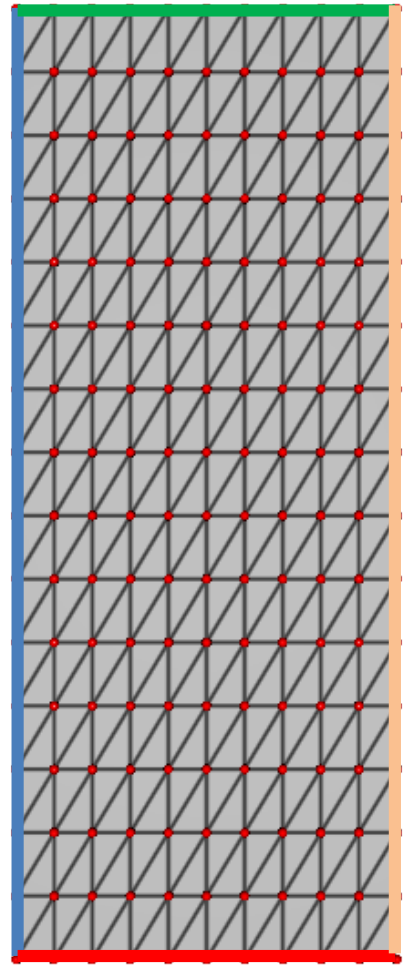
The Mesh creation code above is the same for most examples and so will be omitted unless any relevant changes are needed.

meshLib.**regularSpacing**(first, last, n)

first	The xmin / ymin coordinate
last	The xmax / ymax coordinate
n	The number of desired divisions (n elements, n+1 points)

By default, `borders` are named: gridLeft, gridRight, gridTop and gridBottom

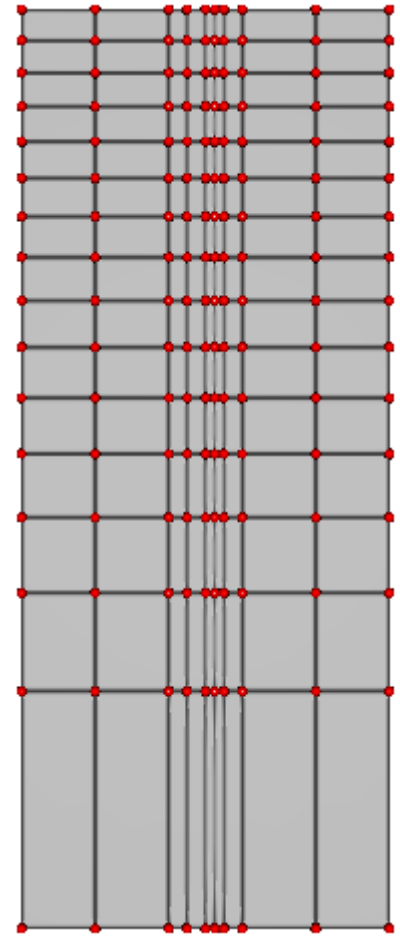
(we will see how to change that on further examples)



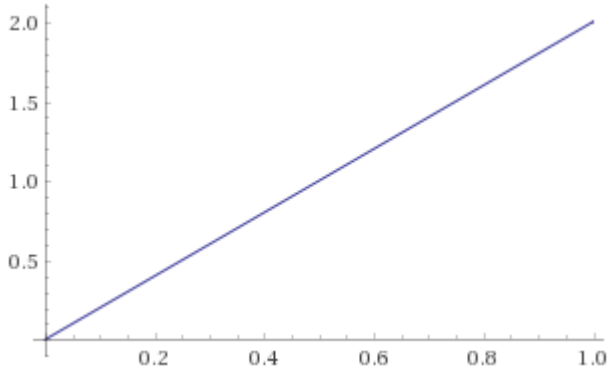
Example 2

Example 1 with varying grid densities and a quad4 mesh. On X, grid positions are defined manually and on Y using a density function.

```
local xPoints = {0.0, 4.0, 8.0, 9.0, 10.0, 10.5, 11.0, 12.0, 16.0, 20.0}
local yPoints = meshLib.densitySpacing(0.0, 50.0, 15, function(t) return 2*t end)

local nodes, cells, borders = meshLib.build2DGrid('quad4', xPoints, yPoints)
```



The density function is evaluated between 0 and 1, and the relative value of the function determines the grid density on the region. It must be a positive function, with higher values meaning denser spacing. The used algorithm calculates the area beneath the curve and spreads points in a way that the area between each point is equal



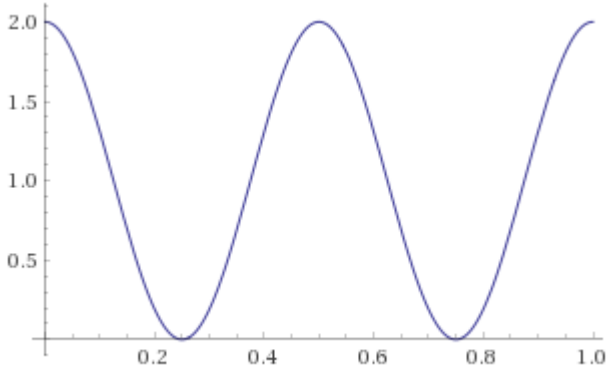
meshLib.**densitySpacing**(first, last, n, f)

first	The xmin / ymin coordinate
last	The xmax / ymax coordinate
n	The number of desired divisions (n elements, n+1 points)
f	The density function

Example 3

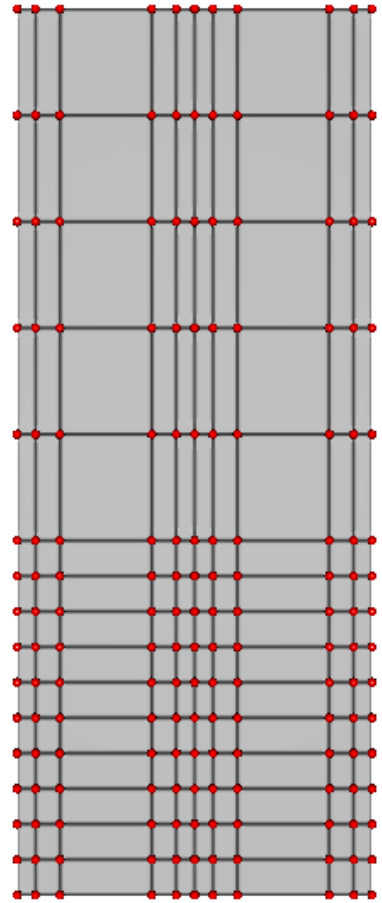
Like Example 2, but now X points are given by a periodic density function and Y points by merging two regular regions.

```
local xPoints = meshLib.densitySpacing(0.0, 20.0, 10,
                                       function(t) return 1 + math.cos(4*math.pi*t) end)

local yPoints = meshLib.merge(meshLib.regularSpacing( 0.0, 20.0, 10),
                              meshLib.regularSpacing(20.0, 50.0,  5))

local nodes, cells, borders = meshLib.build2DGrid('quad4', xPoints, yPoints)
```



meshLib.**merge**(...)

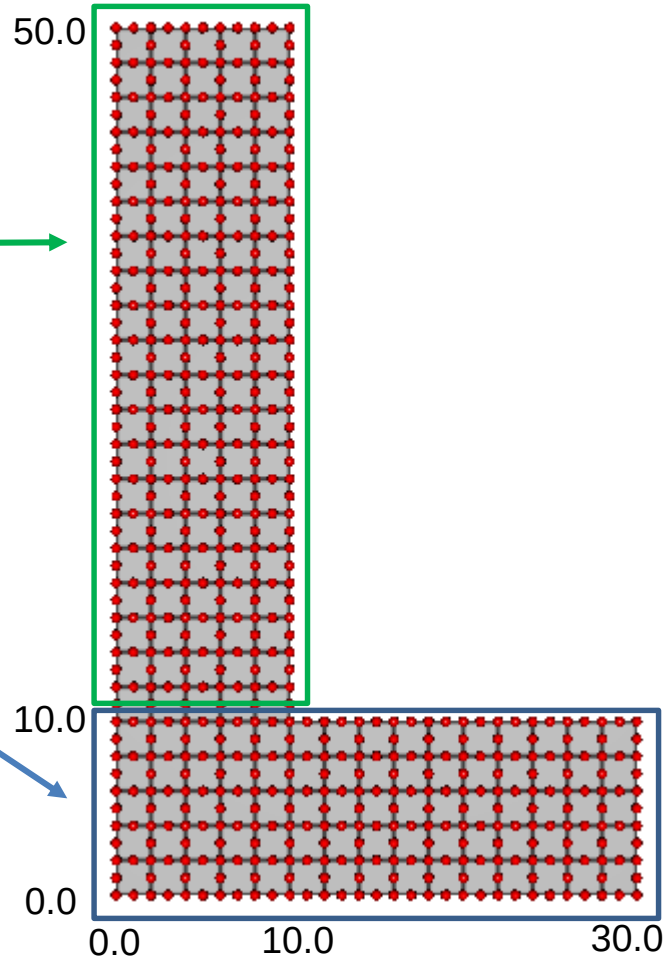
... A set of two or more tables with increasing coordinates, each table starting at a coordinate greater than or equal to the last coordinate from the previous one.



Example 4

An 'L' shaped mesh with Quad8 elements. The horizontal part of the 'L' has X coordinates varying from 0 to 30 with 15 subdivisions. The vertical part has Y coordinates from 0 to 50 with 25 subdivisions. Both arms have 'width' equal to 10.0. It also specifies a 'material' value for all cells equal to 1.

```
local xPoints = meshLib.regularSpacing(0.0, 30.0, 15)
local yPoints = meshLib.regularSpacing(0.0, 50.0, 25)
local blocks = {{ 0.0, 10.0, {0.0, 30.0}},
                {10.0, 50.0, {0.0, 10.0}},
}

local nodes, cells, borders = meshLib.build2DGrid('quad8', xPoints, yPoints,
                                                  blocks, {material = 1})

Mesh{
  id       = 'mesh4',
  typeName = 'GemaMesh.elem',

  coordinateDim  = 2,
  stateVars      = {'T'},
  cellProperties = {'material'},

  nodeData = nodes,
  cellData = cells,
  boundaryEdgeData = borders
}
```



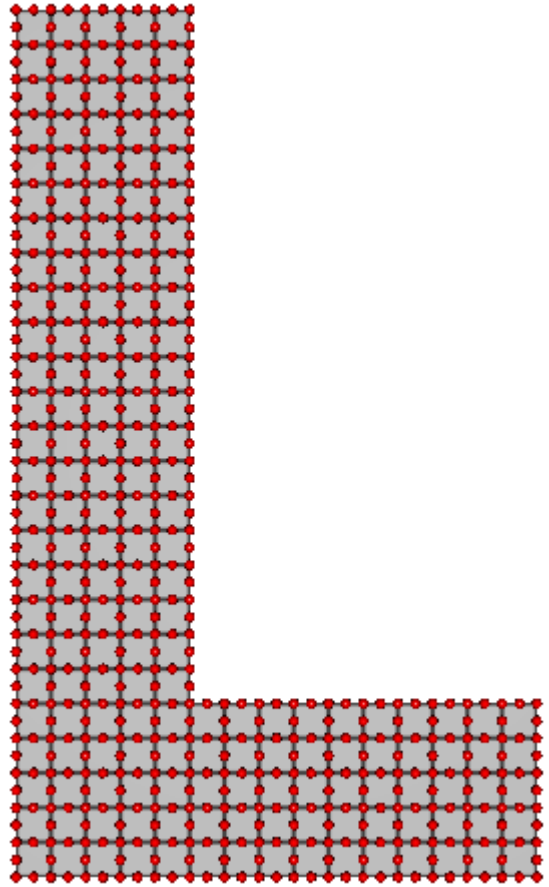
Example 4

An 'L' shaped mesh with Quad8 elements. The horizontal part of the 'L' has X coordinates varying from 0 to 30 with 15 subdivisions. The vertical part has Y coordinates from 0 to 50 with 25 subdivisions. Both arms have 'width' equal to 10.0. It also specifies a 'material' value for all cells equal to 1.

```
local xPoints = meshLib.regularSpacing(0.0, 30.0, 15)
local yPoints = meshLib.regularSpacing(0.0, 50.0, 25)
local blocks = {{ 0.0, 10.0, {0.0, 30.0}},
                {10.0, 50.0, {0.0, 10.0}},
}

local nodes, cells, borders = meshLib.build2DGrid('quad8', xPoints, yPoints,
                                                  blocks, {material = 1})

Mesh{
  id       = 'mesh4',
  typeName = 'GemaMesh.elem',

  coordinateDim  = 2,
  stateVars      = {'T'},
  cellProperties = {'material'},

  nodeData = nodes,
  cellData = cells,
  boundaryEdgeData = borders
}
```

Default 'material' property set value for all elements

Property set name



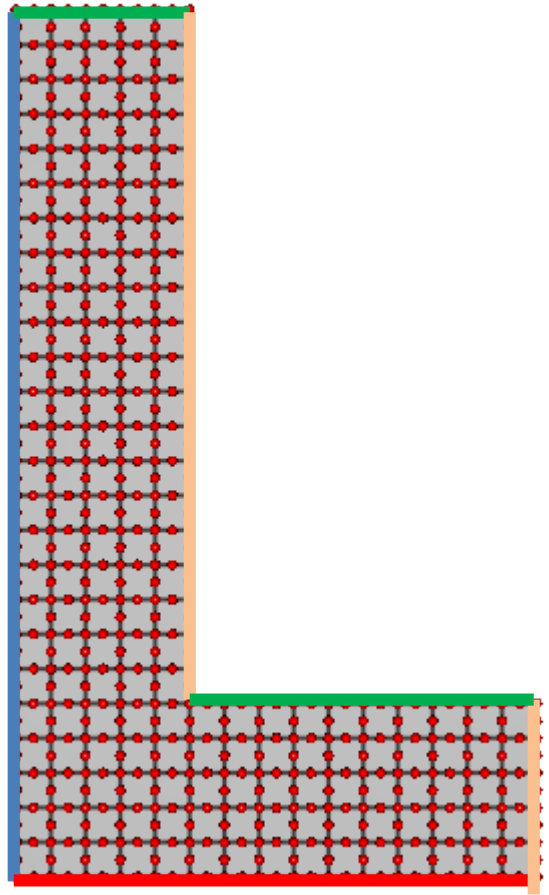
Example 4

An 'L' shaped mesh with Quad8 elements. The horizontal part of the 'L' has X coordinates varying from 0 to 30 with 15 subdivisions. The vertical part has Y coordinates from 0 to 50 with 25 subdivisions. Both arms have 'width' equal to 10.0. It also specifies a 'material' value for all cells equal to 1.

```
local xPoints = meshLib.regularSpacing(0.0, 30.0, 15)
local yPoints = meshLib.regularSpacing(0.0, 50.0, 25)
local blocks = {{ 0.0, 10.0, {0.0, 30.0}},
                {10.0, 50.0, {0.0, 10.0}},
}

local nodes, cells, borders = meshLib.build2DGrid('quad8', xPoints, yPoints,
                                                  blocks, {material = 1})

Mesh{
  id       = 'mesh4',
  typeName = 'GemaMesh.elem',

  coordinateDim  = 2,
  stateVars      = {'T'},
  cellProperties = {'material'},

  nodeData = nodes,
  cellData = cells,
  boundaryEdgeData = borders
}
```

`borders`: gridLeft, gridRight, gridTop, gridBottom

(Top and right might not be what is needed. See the next example)



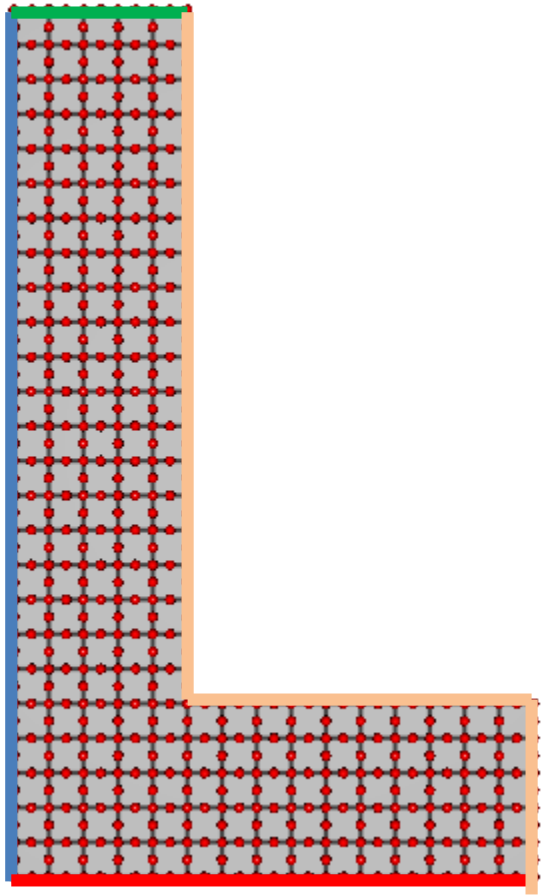
Example 5

Like Example 4, adding a custom border definition.

```
local xPoints = meshLib.regularSpacing(0.0, 30.0, 15)
local yPoints = meshLib.regularSpacing(0.0, 50.0, 25)

local blocks = {{ 0.0, 10.0, {0.0, 30.0}},
                {10.0, 50.0, {0.0, 10.0}},
}

local borderDef = {
        {'leftBorder',   {1, 'left'}, {2, 'left'}},
        {'rightBorder',  {2, 'right'}, {1, 'top', 10.0, 30.0}, {1, 'right'}},
        {'topBorder',    {2, 'top'}},
        {'bottomBorder', {1, 'bottom'}},
}

local nodes, cells, borders = meshLib.build2DGrid('quad8', xPoints, yPoints,
                                                  blocks,{material = 1},
                                                  {borders = borderDef})
```

`borders`: leftBorder, rightBorder, topBorder, bottomBorder

Options table with user border definition



Example 5

Like Example 4, adding a custom border definition.

```
local xPoints = meshLib.regularSpacing(0.0, 30.0, 15)
local yPoints = meshLib.regularSpacing(0.0, 50.0, 25)

local blocks = {{ 0.0, 10.0, {0.0, 30.0}},
                {10.0, 50.0, {0.0, 10.0}},
}

local borderDef = {
        {'leftBorder',   {1, 'left'}, {2, 'left'}},
        {'rightBorder',  {2, 'right'}, {1, 'top', 10.0, 30.0}, {1, 'right'}},
        {'topBorder',    {2, 'top'}},
        {'bottomBorder', {1, 'bottom'}},
}

local nodes, cells, borders = meshLib.build2DGrid('quad8', xPoints, yPoints,
                                                  blocks,{material = 1},
                                                  {borders = borderDef})
```

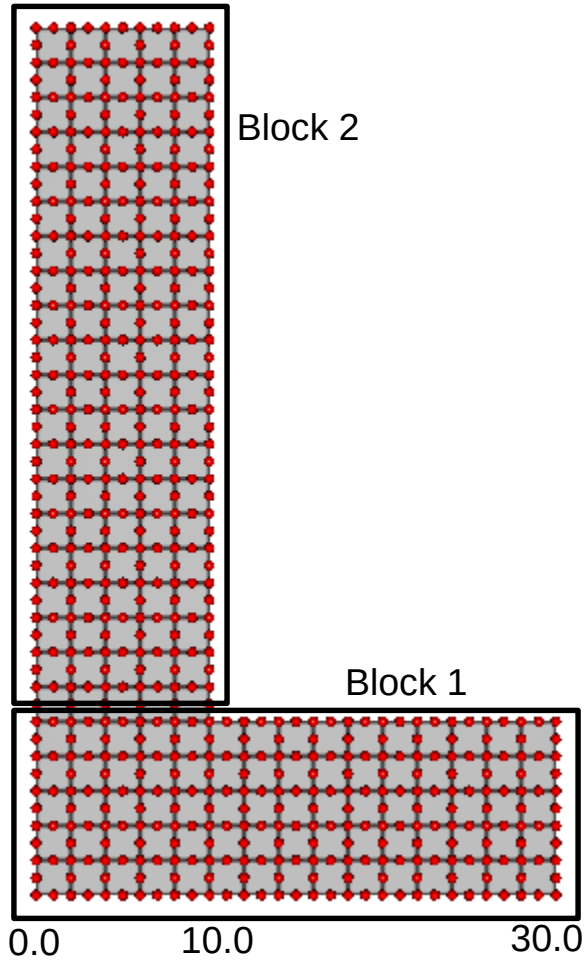
Block 1

Block 2

Block number

Block side

Block side interval



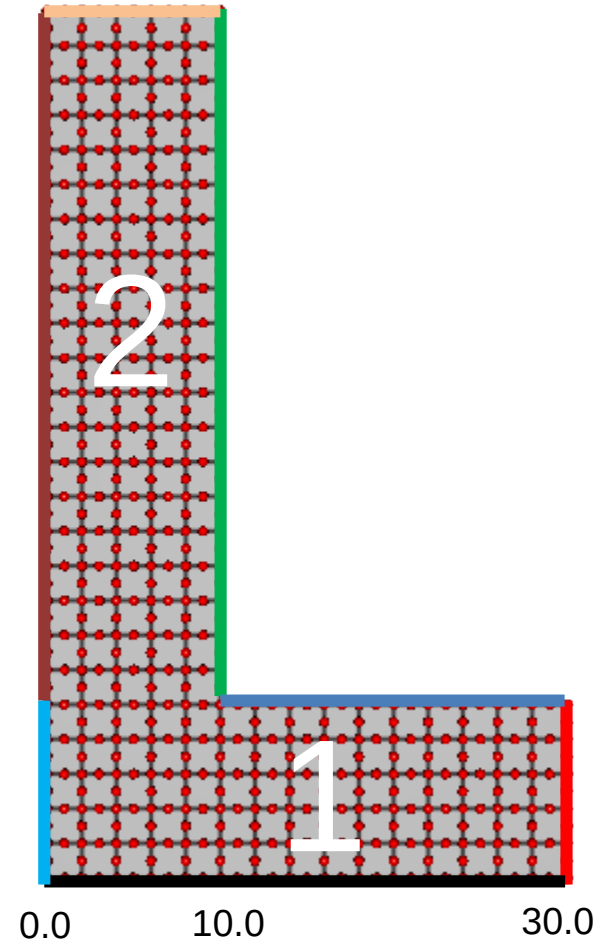
Example 5

Like Example 4, adding a custom border definition.

```
local xPoints = meshLib.regularSpacing(0.0, 30.0, 15)
local yPoints = meshLib.regularSpacing(0.0, 50.0, 25)

local blocks = {{ 0.0, 10.0, {0.0, 30.0}},
                {10.0, 50.0, {0.0, 10.0}},
}

local borderDef = {
        {'leftBorder',   {1, 'left'}, {2, 'left'}},

        {'rightBorder',  {2, 'right'}, {1, 'top', 10.0, 30.0}, {1, 'right'}},

        {'topBorder',    {2, 'top'}},

        {'bottomBorder', {1, 'bottom'}},
}


local nodes, cells, borders = meshLib.build2DGrid('quad8', xPoints, yPoints,
                                                   blocks,{material = 1},
                                                   {borders = borderDef})
```

Block 1

Block 2

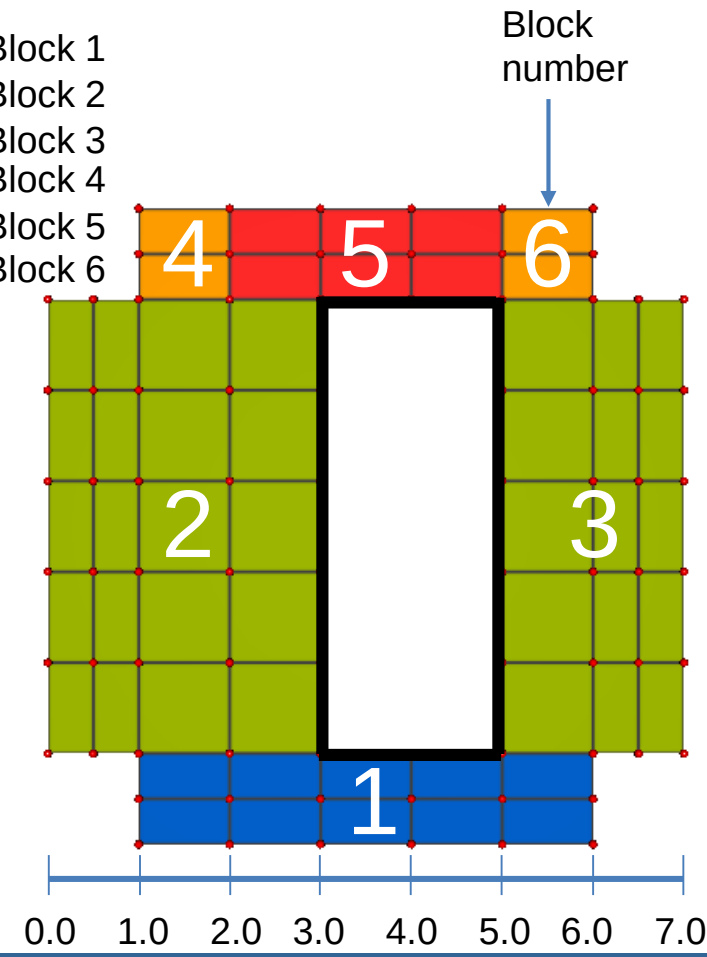


Example 6

A plate with a central rectangular hole and cut corners. Blocks are used to specify varying materials and to group elements into cell groups. A user defined border for the central hole is specified.

```
local points = {0.0, 0.5, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 6.5, 7.0}

local blocks  = {{0.0, 1.0, {1.0, 6.0, cellGroup = 'a', material = 1}},
                 {1.0, 6.0, {0.0, 3.0, cellGroup = 'b'},
                            {5.0, 7.0, cellGroup = 'c'}},
                 {6.0, 7.0, {1.0, 2.0, cellGroup = 'a', material = 3},
                            {2.0, 5.0, cellGroup = 'a', material = 4},
                            {5.0, 6.0, cellGroup = 'a', material = 3}}
                }

local borderDef = {{'hole', {2, 'right'}, {1, 'top', 3.0, 5.0},
                            {3, 'left'}, {5, 'bottom', 3.0, 5.0}},
                   }

local nodes, cells, borders = meshLib.build2DGrid('quad4', points, points,
                                                  blocks, {material = 2},
                                                  {borders = borderDef})
```

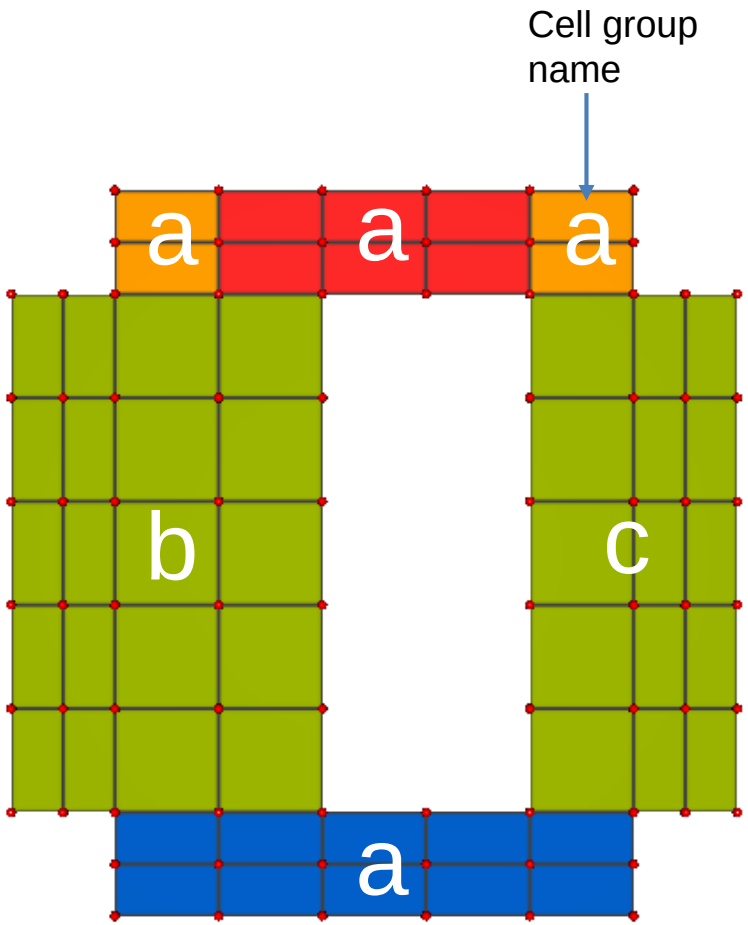
`borders`: hole



Example 6

A plate with a central rectangular hole and cut corners. Blocks are used to specify varying materials and to group elements into cell groups. A user defined border for the central hole is specified.

```
local points = {0.0, 0.5, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 6.5, 7.0}

local blocks  = {{0.0, 1.0, {1.0, 6.0, cellGroup = 'a', material = 1}},
                 {1.0, 6.0, {0.0, 3.0, cellGroup = 'b'},
                            {5.0, 7.0, cellGroup = 'c'}},
                 {6.0, 7.0, {1.0, 2.0, cellGroup = 'a', material = 3},
                            {2.0, 5.0, cellGroup = 'a', material = 4},
                            {5.0, 6.0, cellGroup = 'a', material = 3}}
                }

local borderDef = {{'hole', {2, 'right'}, {1, 'top', 3.0, 5.0},
                            {3, 'left'}, {5, 'bottom', 3.0, 5.0}},
                   }

local nodes, cells, borders = meshLib.build2DGrid('quad4', points, points,
                                                  blocks, {material = 2},
                                                  {borders = borderDef})
```



Material :

Example 7

A square plate with a user defined function for specifying initial values for the state variable and node attributes + a user function for initializing cell materials and attributes.

```
local points  = meshLib.regularSpacing(0.0, 10.0, 10)
local options = {nodef = initNodalValues, cellf = initCellValues}

local nodes, cells, borders = meshLib.build2DGrid('quad4', points, points, nil, nil, options)

Mesh{
  id       = 'mesh7',
  typeName = 'GemaMesh.elem',

  coordinateDim  = 2,
  stateVars      = {'T'},
  nodeAttributes = {{id = 'na1', description = 'A test node attribute'},
                    {id = 'na2', dim = 2, description = 'Another (vectorial) test node attribute'},
                   },

  cellProperties = {'material', 'another'},
  cellAttributes = {{id = 'ca1', description = 'A test cell attribute'},
                   },

  nodeData = nodes,
  cellData = cells,
  boundaryEdgeData = borders
}
```

Materials will be assigned by the cell function

Two aditional node attributes

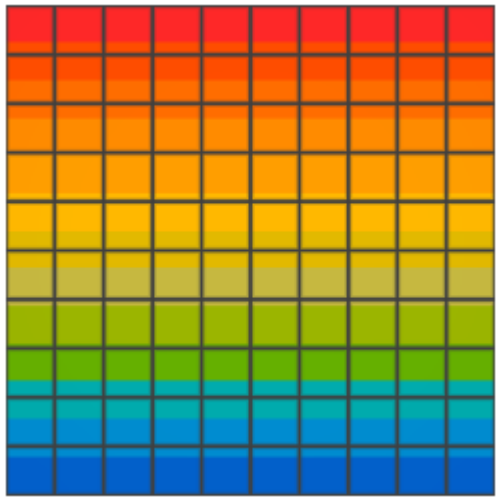
Mesh bound to two property sets

Aditional cell attribute

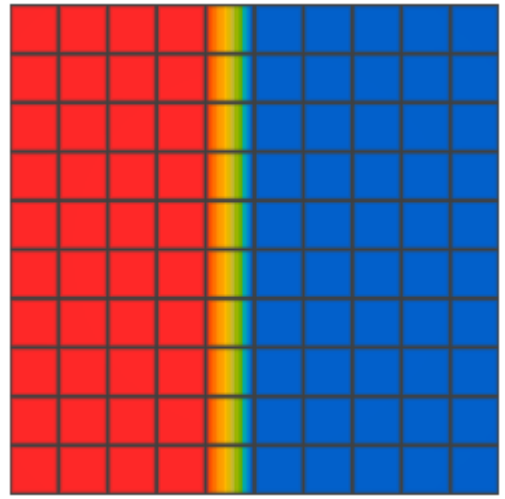
Example 7

```
local function initNodalValues(nodeDef)
  -- On this function, nodeDef is a table initialized with node coordinates
  -- This function should append values for node attributes and state variables
  -- to nodeDef.  The order is first node attributes (in the order given on the
  -- mesh), then state vars.
  local x, y = nodeDef[1], nodeDef[2]

  -- Init values for na1, na2 and T.  Those are dummy values without any special
  -- meaning other than showing how to do it :)
  if x < 5 then -- Init na1
    nodeDef[3] = 1
  else
    nodeDef[3] = 0
  end
  nodeDef[4] = {x, 0}  -- Init na2 with a vector
  nodeDef[5] = y * 100 -- Init T with a gradient depending on the y coordinate
end
```

T



na1



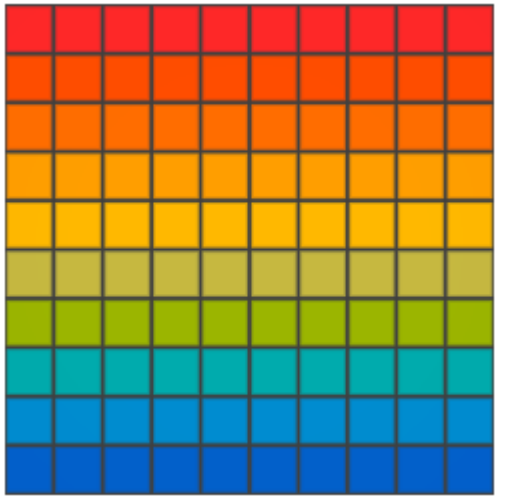
This same function can also be used for transforming coordinates.
See the later examples for parametric meshes

Example 7

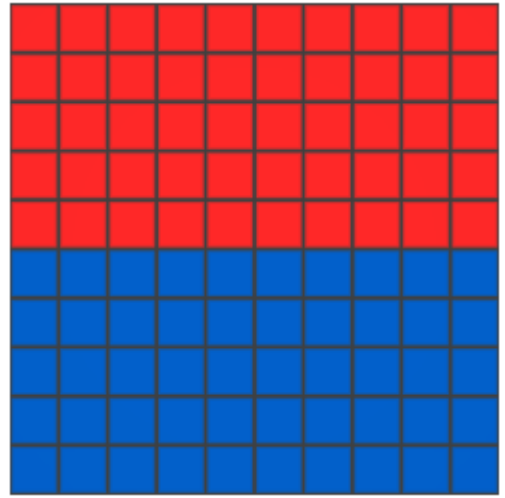
```
local function initCellValues(cellDef, nodeList)
  -- On this function, cellDef is a table initialized with the cell node list,
  -- and nodeList a table that can be indexed by node number to get its coordinates.
  -- This function should append values for cell attributes to cellDef and also set
  -- property set values as (key, value) pairs.
  local nnodes = #cellDef -- The number of nodes in this cell

  -- As demonstration, lets get the minimum and maximum y values for nodes in this cell
  local ymin, ymax = math.huge, -math.huge
  for i = 1, nnodes do
    local node  = cellDef[i]
    local nodeY = nodeList[node][2]
    if nodeY < ymin then ymin = nodeY end
    if nodeY > ymax then ymax = nodeY end
  end

  -- Init ca1
  local yc = (ymin + ymax) / 2.0
  cellDef[nnodes+1] = yc

  -- Init Property sets (all property sets MUST be initialized)
  if yc <= 2.5 then
    cellDef.material = 1
    cellDef.another  = 1
  elseif yc <= 5.0 then
    cellDef.material = 2
    cellDef.another  = 1
  elseif yc <= 7.5 then
    cellDef.material = 3
    cellDef.another  = 2
  else
    cellDef.material = 4
    cellDef.another  = 2
  end
end
```

material

ca1



another



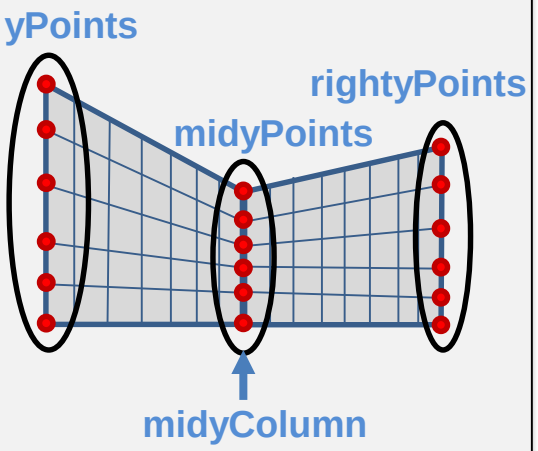
Example 8

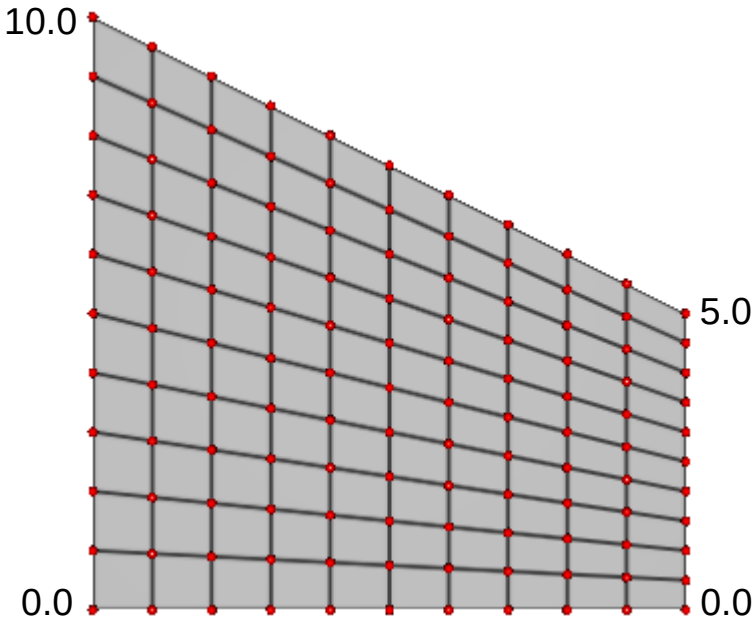
A grid with slanted rows, created by specifying two sets of y coordinates, one for the left border and another for the right border.

```
local xPoints      = meshLib.regularSpacing(0.0, 10.0, 10)
local leftyPoints  = meshLib.regularSpacing(0.0, 10.0, 10)
local rightyPoints = meshLib.regularSpacing(0.0,  5.0, 10)

local options = {rightyPoints = rightyPoints}

local nodes, cells, borders = meshLib.build2DGrid('quad4', xPoints, leftyPoints, nil, nil, options)
```

Up to 3 coordinate sets (with the same size) can be used, in each direction, to create the mesh. The first one is given by the function parameters (**xPoints** and **yPoints**), and will always be tied to the bottom / left side of the grid.

A second set, with values for the top and right borders, can be given by the **topxPoints** and **rightyPoints** fields of the **options** table, as seen above. A third set, with values for a mid row / column, can be given by the **midxPoints** and **midyPoints** fields, with the row / column position given by **midxLine** and **midyColumn**. See the next example.



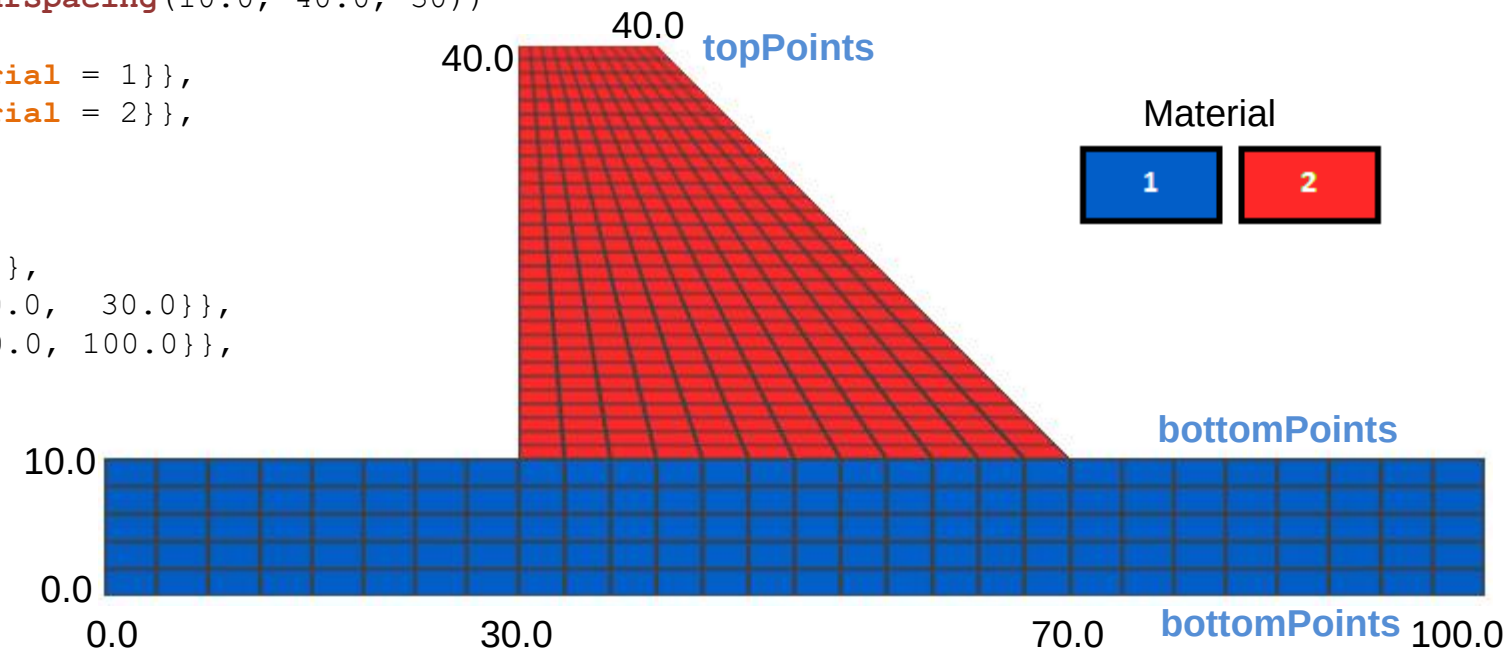


Example 9

Simple ‘dam’ like mesh, defined with the help of 3 x coordinate sets and with a user defined border.

```
local bottomPoints = meshLib.merge(meshLib.regularSpacing(0.0, 30.0, 8),
                                   meshLib.regularSpacing(30.0, 70.0, 12),
                                   meshLib.regularSpacing(70.0, 100.0, 8))

local topPoints    = meshLib.merge(meshLib.regularSpacing(0.0, 30.0, 8),
                                   meshLib.regularSpacing(30.0, 40.0, 12),
                                   meshLib.regularSpacing(40.0, 100.0, 8))

local yPoints      = meshLib.merge(meshLib.regularSpacing( 0.0, 10.0, 5),
                                   meshLib.regularSpacing(10.0, 40.0, 30))

local blocks = { { 0.0, 10.0, { 0.0, 100.0, material = 1}},
                 {10.0, 40.0, {30.0,  70.0, material = 2}},
}

local borderDef = { {'damTop',      {2, 'top'}},
                    {'base',        {1, 'bottom'}},
                    {'leftGround',  {1, 'top',  0.0,  30.0}},
                    {'rightGround', {1, 'top', 70.0, 100.0}},
}

local options = { borders    = borderDef,
                  topxPoints = topPoints,
                  midxPoints = bottomPoints,
                  midxLine   = 10.0,
}
```

The [30, 40] interval is the dam top. The other two intervals exist to complete the set since all x sets must have the same number of points



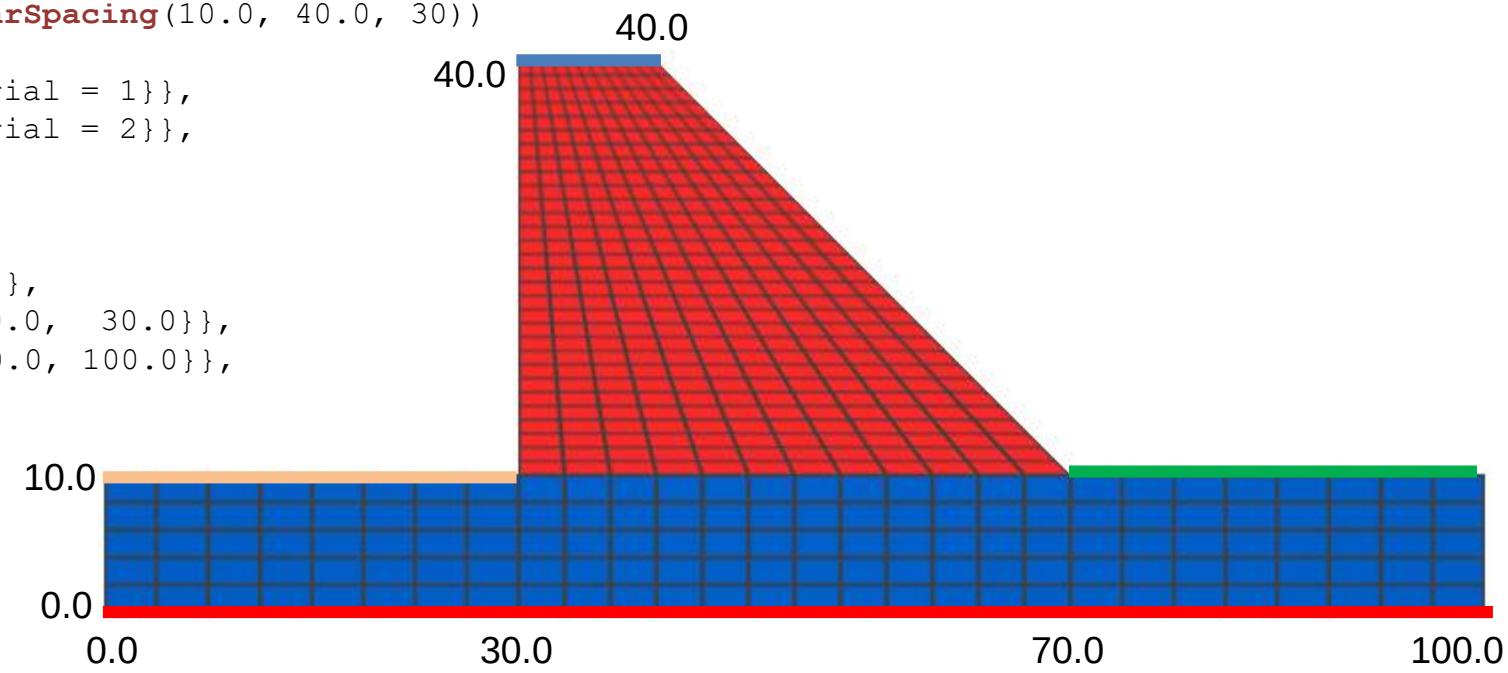

```
local nodes, cells, borders = meshLib.build2DGrid('quad4', bottomPoints, yPoints, blocks, {material = 1}, options)
```

Example 9

Simple ‘dam’ like mesh, defined with the help of 3 x coordinate sets and with a user defined border.

```
local bottomPoints = meshLib.merge(meshLib.regularSpacing(0.0, 30.0, 8),
                                   meshLib.regularSpacing(30.0, 70.0, 12),
                                   meshLib.regularSpacing(70.0, 100.0, 8))

local topPoints    = meshLib.merge(meshLib.regularSpacing(0.0, 30.0, 8),
                                   meshLib.regularSpacing(30.0, 40.0, 12),
                                   meshLib.regularSpacing(40.0, 100.0, 8))

local yPoints      = meshLib.merge(meshLib.regularSpacing( 0.0, 10.0, 5),
                                   meshLib.regularSpacing(10.0, 40.0, 30))

local blocks = { { 0.0, 10.0, { 0.0, 100.0, material = 1}},
                 {10.0, 40.0, {30.0,  70.0, material = 2}},
}

local borderDef = { {'damTop',      {2, 'top'}},
                    {'base',        {1, 'bottom'}},
                    {'leftGround',  {1, 'top',  0.0,  30.0}},
                    {'rightGround', {1, 'top', 70.0, 100.0}},
}

local options = { borders    = borderDef,
                  topxPoints = topPoints,
                  midxPoints = bottomPoints,
                  midxLine   = 10.0,
}
```

borders: damTop, base, leftGround, rightGround

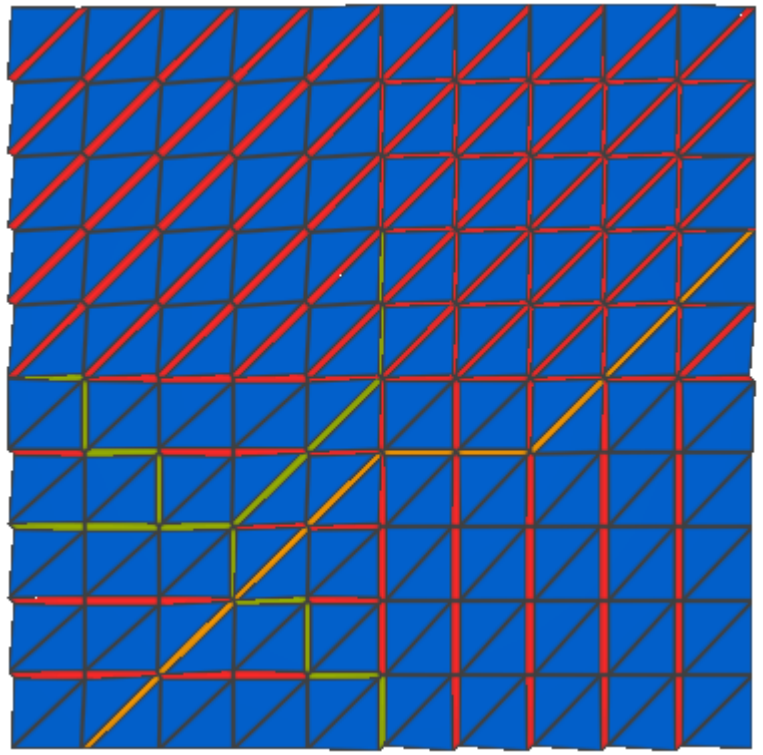



```
local nodes, cells, borders = meshLib.build2DGrid('quad4', bottomPoints, yPoints, blocks, {material = 1}, options)
```

II – ADDING INTERFACE ELEMENTS

Adding interface elements using build2DGrid()

- Available for tri3 and quad4 meshes
  - No quadratic interface elements available in GeMA (yet…)
- User defined paths on the mesh
  - Given by sequences of node coordinates (x, y) through **options.interfaceList**
  - Each coordinate pair must correspond to a grid edge
  - Overlapping and border edges are ignored
  - Coordinates can be given as (line, column) pairs if options.useGridCoords is true
  - Each path can be tied to a material
- Inside grid blocks
  - Each block can be fully filled with horizontal, vertical and / or diagonal interface elements
  - The **interface** tag, added to the block definition, specifies which edges will be replaced by an interface element. Its value should be a string composed by the desired combination of letters 'h', 'v' and 'd'.
  - The **interfaceMatList** block tag allows for tying interface elements to a specific material
- Above options can be combined



Adding interface elements using build2DGrid()

- Interface adding options are given by **options.interfaceOptions**
  - hasMidNode — If true, the interface element will have 6 nodes instead of 4

  - matList — Default material list for interface elements
  - cellf — Like options.cellf for interface elements

If nil, this values will be auto filled with the material list passed in the call to build2DGrid() and with options.cellf

  - useQuad4 — Debug option for replacing the created interface elements by quad4 elements (for pre-processors that do not understand interface elements)
  - offset — Debug option for ‘opening’ the interface element a little bit so that it can be viewed on a pré-processor before the analysis. A value in the mesh scale. Something like 5% of a standard element length is a good measure

Example 10

A tri3 square plate with three added 'fractures' represented by interface elements. Each fracture path is given by a list of node coordinates. The square plate is tied to material 1 while fractures 1 and 2 are tied to the default fracture material 2 and fracture 3 to material 3.

```
local points = meshLib.regularSpacing(0.0, 10.0, 10)

local paths = {
  {{0, 3}, {1, 3}, {2, 3}, {3, 3}, {4, 4}, {5, 5}, {5, 6}, {5, 7}},
  {{5, 0}, {5, 1}, {4, 1}, {4, 2}, {3, 2}, {3, 3}, {2, 3}, {2, 4}, {1, 4}, {1, 5}, {0, 5}},
  {{10, 7}, {9, 6}, {8, 5}, {7, 4}, {6, 4}, {5, 4}, {4, 3}, {3, 2}, {2, 1}, {1, 0}, material = 3},
}

local options = {
  interfaceList    = paths,
  interfaceOptions = {matList = {material = 2}, offset = 0.05},
}

local nodes, cells, borders = meshLib.build2DGrid('tri3', points, points,
                                                  nil, {material = 1},
                                                  options)
```

Path 1

Path 2

Path 3 material

Path 3

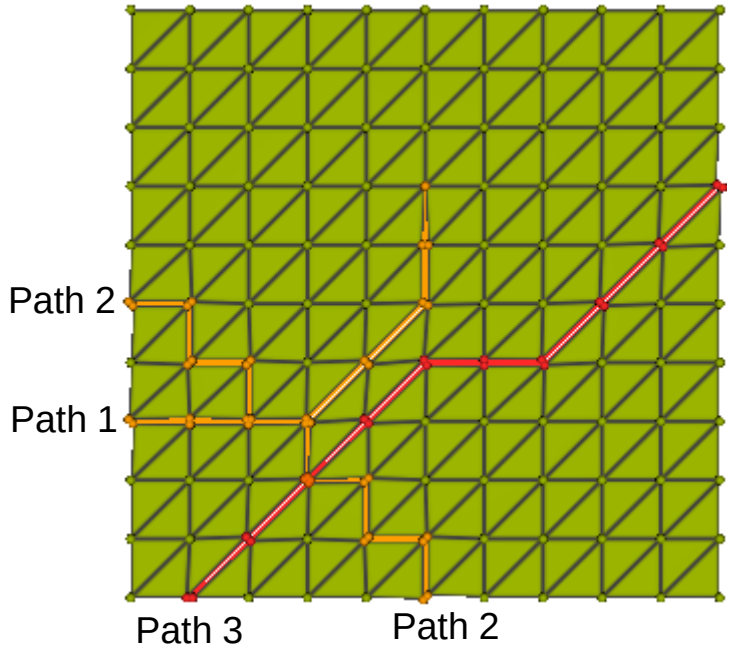
Node (x, y) coordinate

Default interface material

Debug offset. Should be removed before analysis

Default cell material

Material : 1 2 3



Example 11

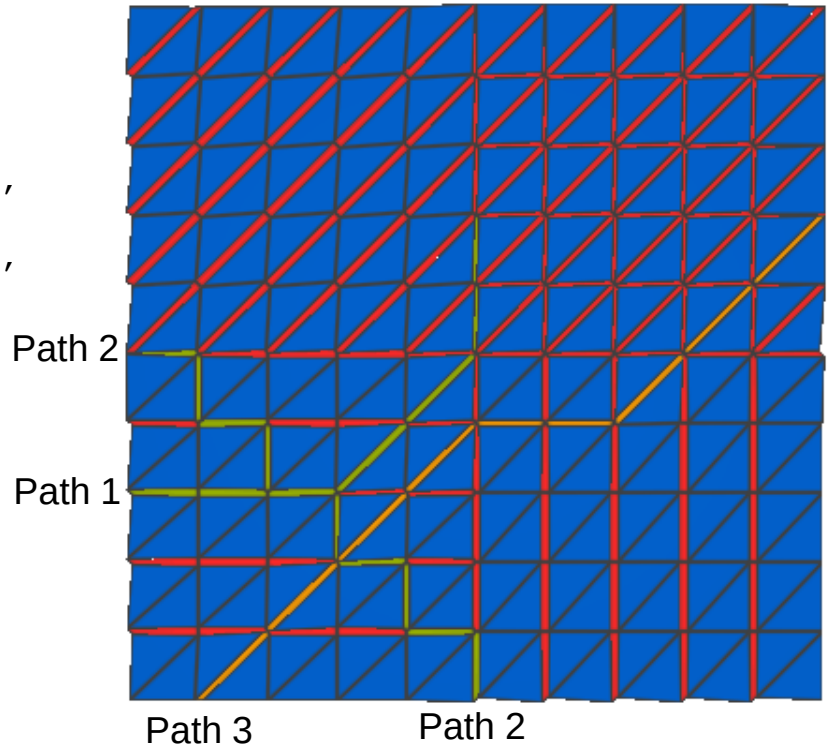
Like example 10, but breaking the plate into four distinct blocks and adding to each one a set of interface elements aligned to horizontal, vertical, diagonal and all directions.

```
local points = meshLib.regularSpacing(0.0, 10.0, 10)

local blocks = {
  {0.0,  5.0, {0.0,  5.0, interface = 'h',   interfaceMatList = {material = 4}},
              {5.0, 10.0, interface = 'v',   interfaceMatList = {material = 4}}},
  {5.0, 10.0, {0.0,  5.0, interface = 'd',   interfaceMatList = {material = 4}},
              {5.0, 10.0, interface = 'hvd', interfaceMatList = {material = 4}}},
}
```

Block interface direction control

Material for block interface elements

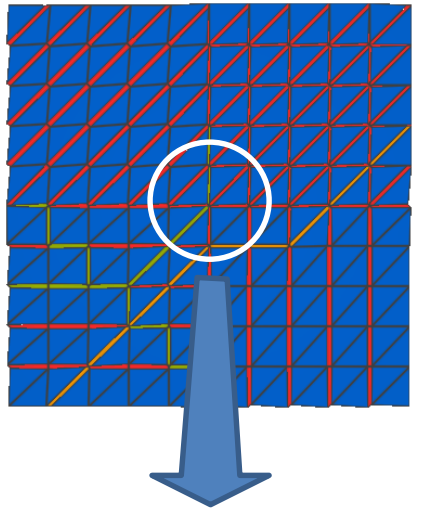
```
local paths = {
  {{0, 3}, {1, 3}, {2, 3}, {3, 3}, {4, 4}, {5, 5}, {5, 6}, {5, 7}},
  {{5, 0}, {5, 1}, {4, 1}, {4, 2}, {3, 2}, {3, 3}, {2, 3}, {2, 4},
   {1, 4}, {1, 5}, {0, 5}},
  {{10, 7}, {9, 6}, {8, 5}, {7, 4}, {6, 4}, {5, 4}, {4, 3}, {3, 2},
   {2, 1}, {1, 0}, material = 3},
}

local options = {
  interfaceList    = paths,
  interfaceOptions = {matList = {material = 2}, hasMidNode = true, offset = 0.05},
}

local nodes, cells, borders = meshLib.build2DGrid('tri3', points, points, blocks,
                                                  {material = 1}, options)
```



Material :

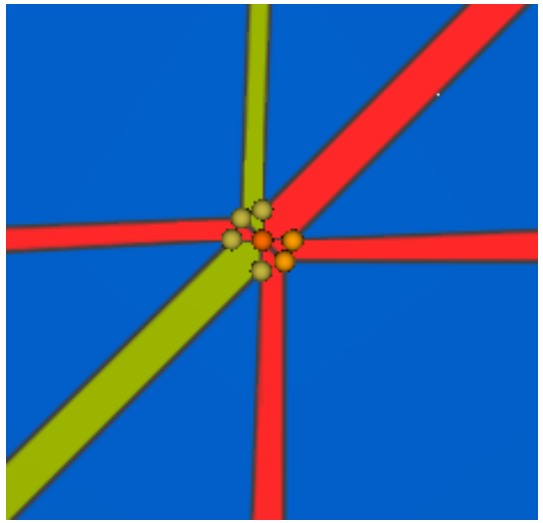
Example 11

Like example 10, but breaking the plate into four distinct blocks and adding to each one a set of interface elements aligned to horizontal, vertical, diagonal and all directions.



```
local points = meshLib.regularSpacing(0.0, 10.0, 10)

local blocks = {
  {0.0,  5.0, {0.0,  5.0, interface = 'h',   interfaceMatList = {material = 4}},
              {5.0, 10.0, interface = 'v',   interfaceMatList = {material = 4}}},
  {5.0, 10.0, {0.0,  5.0, interface = 'd',   interfaceMatList = {material = 4}},
              {5.0, 10.0, interface = 'hvd', interfaceMatList = {material = 4}}},
}


local paths = {
  {{0, 3}, {1, 3}, {2, 3}, {3, 3}, {4, 4}, {5, 5}, {5, 6}, {5, 7}},
  {{5, 0}, {5, 1}, {4, 1}, {4, 2}, {3, 2}, {3, 3}, {2, 3}, {2, 4},
   {1, 4}, {1, 5}, {0, 5}},
  {{10, 7}, {9, 6}, {8, 5}, {7, 4}, {6, 4}, {5, 4}, {4, 3}, {3, 2},
   {2, 1}, {1, 0}, material = 3},
}

local options = {
  interfaceList    = paths,
  interfaceOptions = {matList = {material = 2}, hasMidNode = true, offset = 0.05},
}

local nodes, cells, borders = meshLib.build2DGrid('tri3', points, points, blocks,
                                                  {material = 1}, options)
```

Debug offset. Should be removed before analysis



Example 12

Like example 9 (dam), adding a fracture path along column 15. In order to specify the fracture path easily, coordinates are givens as (line, column) pairs instead of (x, y) values. The base block has material equal to 1, while the top dam has material = 2 and the fracture has material = 3.

```
local bottomPoints = meshLib.merge(meshLib.regularSpacing(0.0, 30.0, 8),
                                   meshLib.regularSpacing(30.0, 70.0, 12),
                                   meshLib.regularSpacing(70.0, 100.0, 8))
local topPoints    = meshLib.merge(meshLib.regularSpacing(0.0, 30.0, 8),
                                   meshLib.regularSpacing(30.0, 40.0, 12),
                                   meshLib.regularSpacing(40.0, 100.0, 8))
local yPoints      = meshLib.merge(meshLib.regularSpacing( 0.0, 10.0, 5),
                                   meshLib.regularSpacing(10.0, 40.0, 30))

local blocks = { { 1,  6, {1, 29, material = 1}},
                 { 6, 36, {9, 21, material = 2}} }

local borderDef = { {'damTop',      {2, 'top'}},
                    {'base',        {1, 'bottom'}},
                    {'leftGround',  {1, 'top',  1,  9}},
                    {'rightGround', {1, 'top', 21, 29}} }
```

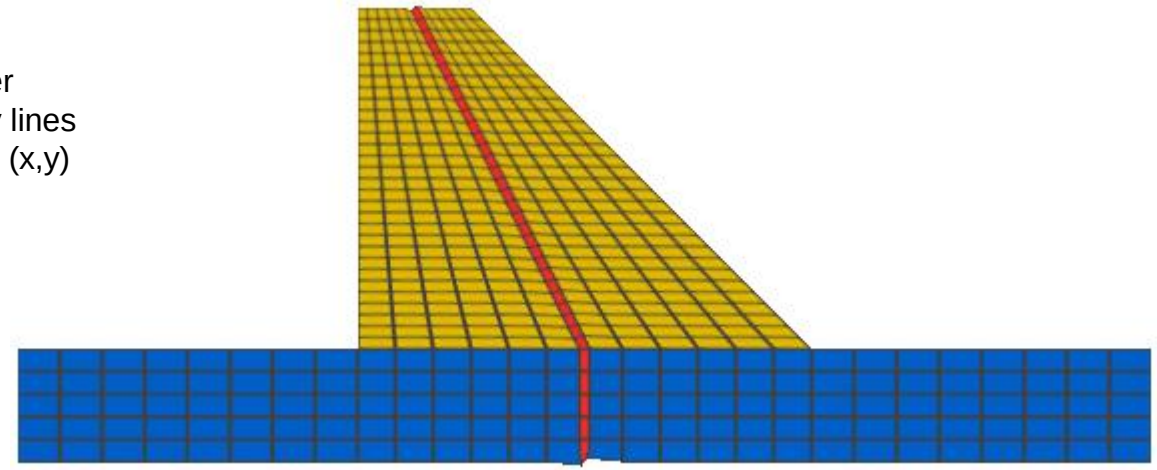
Block and border coordinates given by lines / columns instead of (x,y)

```
-- Creates a fracture path along column 15. Uses a loop instead of manually
-- enumerating the nodes in the path where interface elements should be added
local fracturePath = {material = 3}
for line = 1, 36 do
  fracturePath[line] = {line, 15}
end
```

Enable line / column coordinates

```
local options = { useGridCoords    = true,          borders       = borderDef,
                  topxPoints       = topPoints,     midxPoints    = bottomPoints,
                  midxLine         = 6,             interfaceList = {fracturePath},
                  interfaceOptions = {useQuad4 = true, offset = 0.4} }

local nodes, cells, borders = meshLib.build2DGrid('quad4', bottomPoints, yPoints, blocks, {material = 1}, options)
```

Material : 1 2 3



With useQuad4 = true, interface elements are generated as quad4 elements. Like offset, this is a debug option and should be removed before analysis

Adding interface elements at the orchestration

- Mesh must have topological query support (the standard GemaMesh will do)
- User defined paths
  - Created by calling meshLib.**addInterfaceElementsTo2DMesh**(mesh, pathList, options)
  - The ‘pathList’ is a list of paths, each consisting of a list with sequences of node indices
  - Each node pair must correspond to a mesh edge
  - Overlapping and border edges are ignored
  - Each path can be tied to a different material
  - Adding options are the same as the ones available when using build2DGrid()
- Filling the pathList with edges inside mesh rectangular areas can be done automatically by calling:
  - meshLib.**add2DMeshEdgesToPathList**(mesh, pathList, region, matList)
  - The ‘region’ parameter is optional and when missing, the entire mesh will be filled
  - The ‘matList’ parameter is also optional and defines the material values for the region

Example 13

Adds interface elements at the orchestration phase to a mesh created on the model (manually or by build2DGrid()). At the orchestration, every mesh edge will be replaced by an interface element.

Model

```
local points = meshLib.regularSpacing(0.0, 10.0, 10)
local nodes, cells, borders = meshLib.build2DGrid('tri3', points, points, nil, {material = 1})

Mesh{ id           = 'mesh13',
      typeName     = 'GemaMesh.elem',
      coordinateDim    = 2,
      stateVars        = {'T'},
      cellProperties   = {'material'},
      nodeData         = nodes,
      cellData         = cells,
      boundaryEdgeData = borders
}
```

Cell material

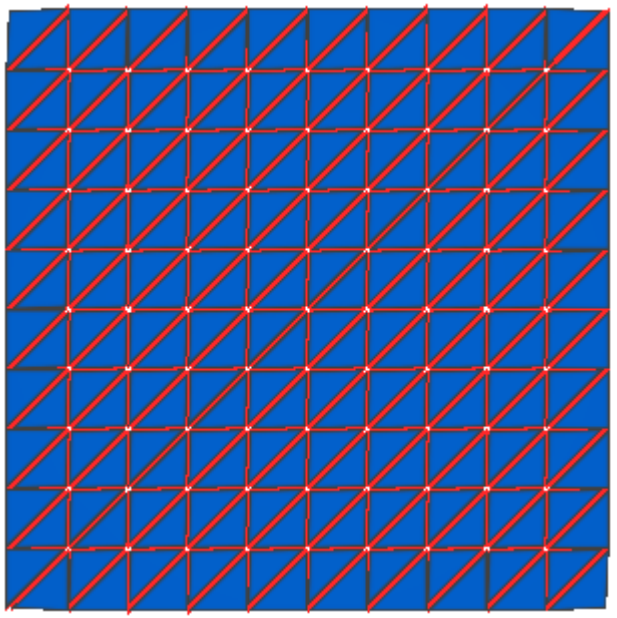
Orchestration

```
function ProcessScript()
  local paths = {}
  meshLib.add2DMeshEdgesToPathList('mesh13', paths)

  local options = {offset = 0.05,
                   matList = {material = 2}}

  meshLib.addInterfaceElementsTo2DMesh('mesh13', paths, options)
  ...
end
```

Empty path list, filled by the function

Debug offset. Should be removed before analysis

Interface material



Example 14

Similar to example 13. At the orchestration, two user defined fractures are created with material 2. Also, every mesh edge inside two square regions are transformed into an interface element and tied to material 3 for the first region and to material 4 for the second.

Orchestration

```
function ProcessScript()
 local paths = {
    {6, 17, 16, 27, 26, 37, 36, 47, 46, 57, 56},
    {2, 14, 26, 38, 50, 51, 52, 64, 76, 88},
  }

  meshLib.add2DMeshEdgesToPathList('mesh14', paths,
                                   {6, 10, 0, 5},
                                   {material = 3})

  meshLib.add2DMeshEdgesToPathList('mesh14', paths,
                                   {0, 4, 6, 10},
                                   {material = 4})

  local options = {offset = 0.05, matList = {material = 2}}

  meshLib.addInterfaceElementsTo2DMesh('mesh14', paths, options)
end
```

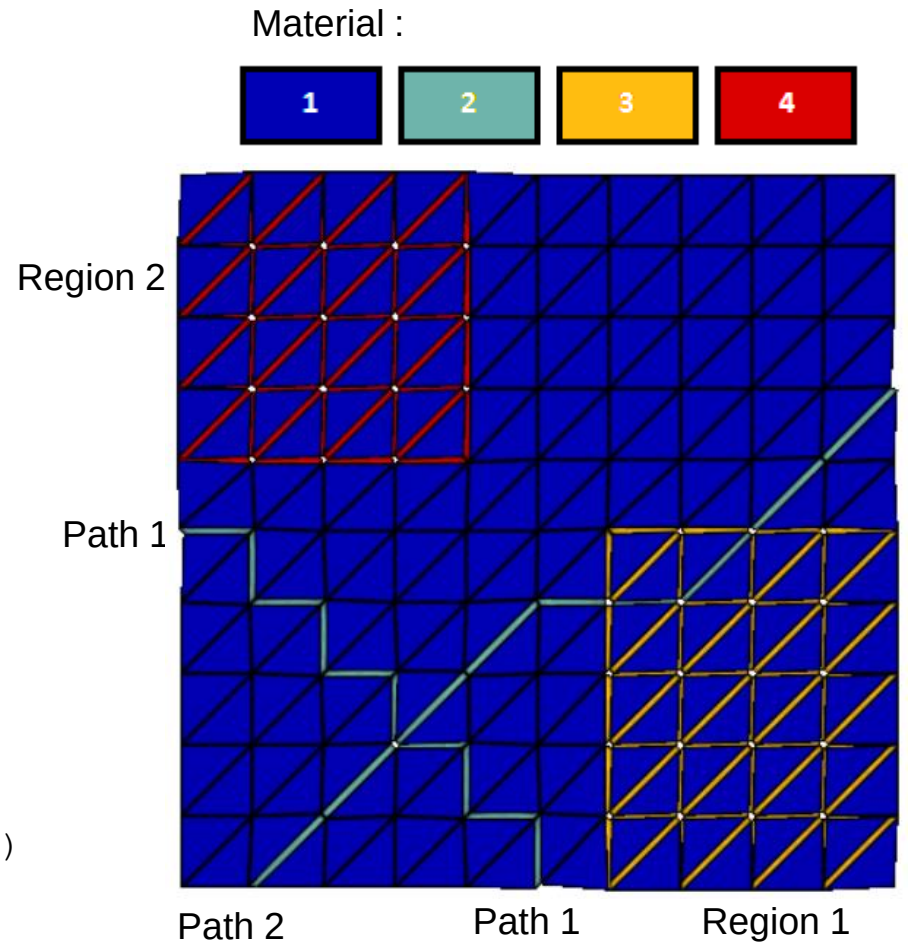
Path 1

Path 2

Node indices

Region 1 (xmin, xmax, ymin, ymax)

Region 2



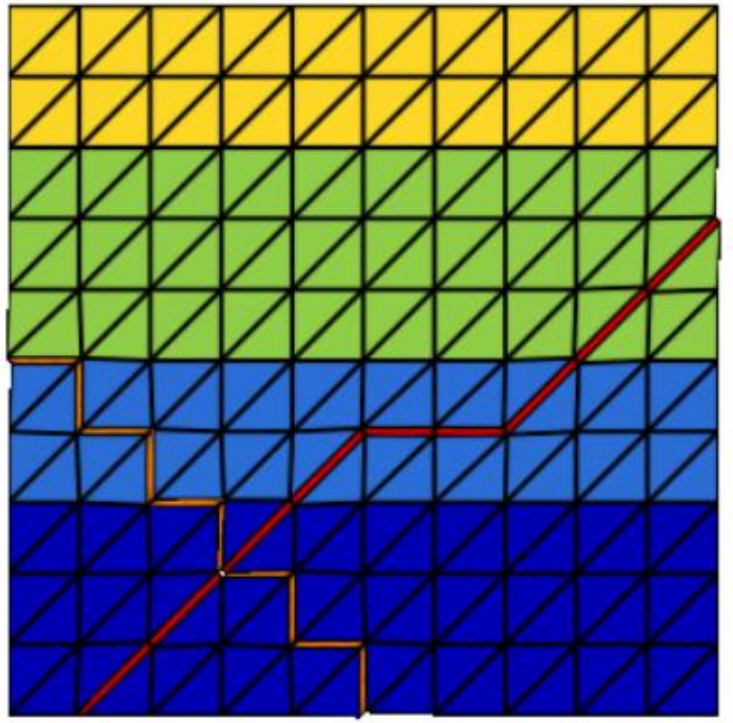
Example 15

This example revisits example 7 (the one with initialization functions) adding two fracture paths on the orchestration. The new interface elements are initialized by using another user function.

Orchestration

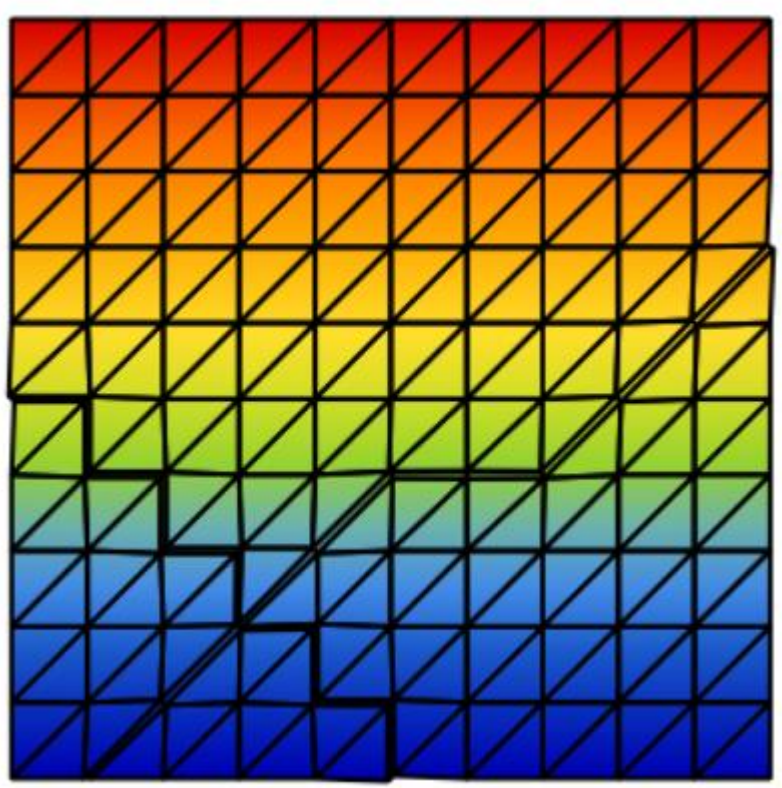
```
local function initInterfaceValues(cellDef, nodeAc, pathIndex)
  -- On this function, cellDef is a table initialized with the cell node list, nodeAc is
  -- a mesh coordinate accessor and pathIndex is the value of the 'pathIndex' key in the
  -- path entry that originated this interface element or its path index in pathList if
  -- the pathIndex key was not given. The function should add cell attribute and property
  -- set values to the cellDef table. Values should be added to the table as name/value
  -- pairs for both property sets and attributes.
  cellDef.ca1 = pathIndex * 10
  cellDef.material = pathIndex
  cellDef.another  = 3
end

function ProcessScript()
 local paths = {
    {6, 17, 16, 27, 26, 37, 36, 47, 46, 57, 56, pathIndex = 5},
    {2, 14, 26, 38, 50, 51, 52, 64, 76, 88,     pathIndex = 6},
  }

  local options = {offset = 0.05, cellf = initInterfaceValues}
  meshLib.addInterfaceElementsTo2DMesh('mesh15', paths, options)
end
```

Material :

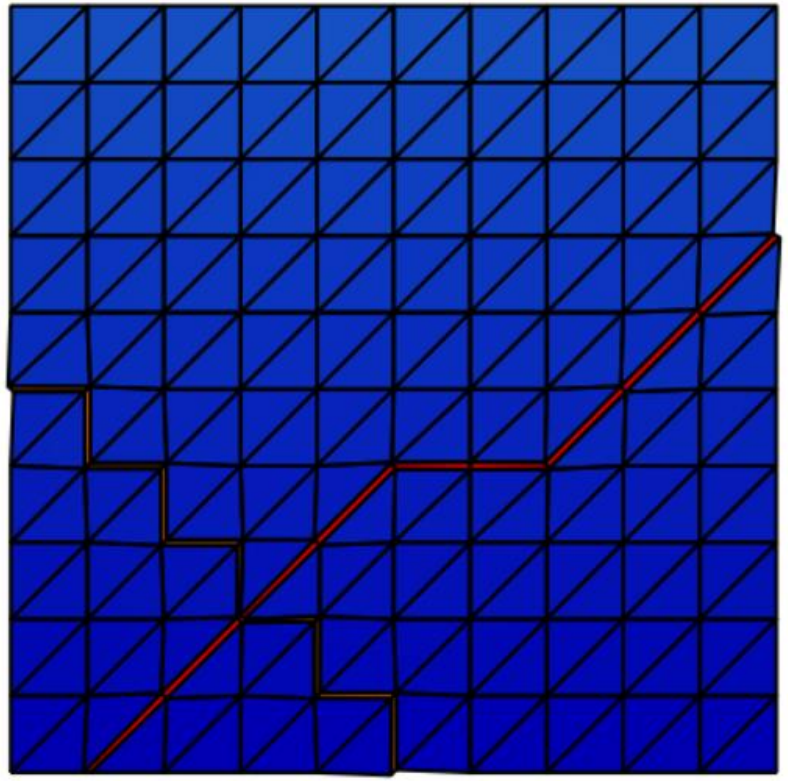


Example 15

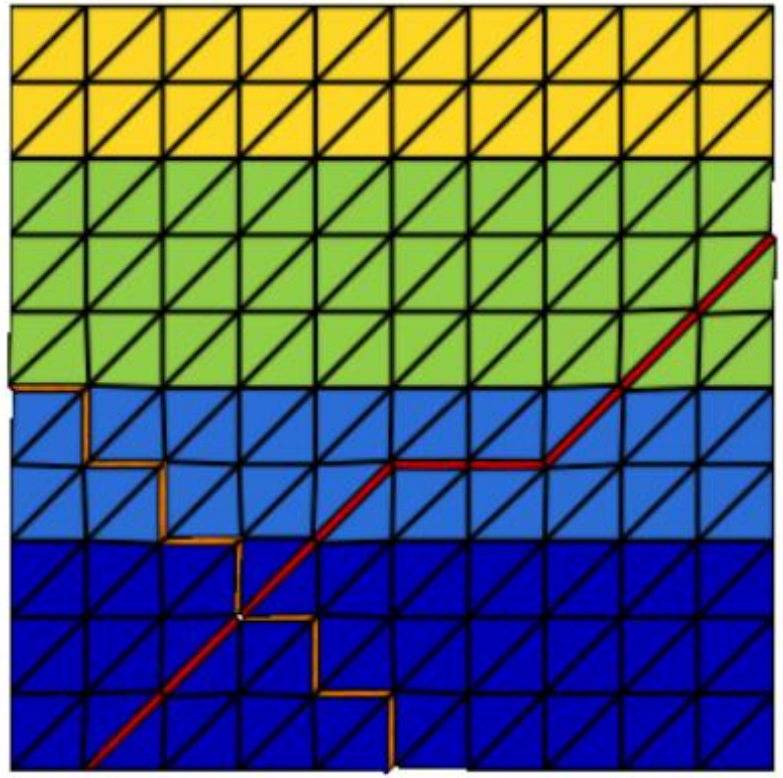
This example revisits example 7 (the one with initialization functions) adding two fracture paths on the orchestration. The new interface elements are initialized by using another user function.



T: Nodal values are copied when nodes are splitted to insert an interface element



ca1: Cell values are filled by the supplied user function



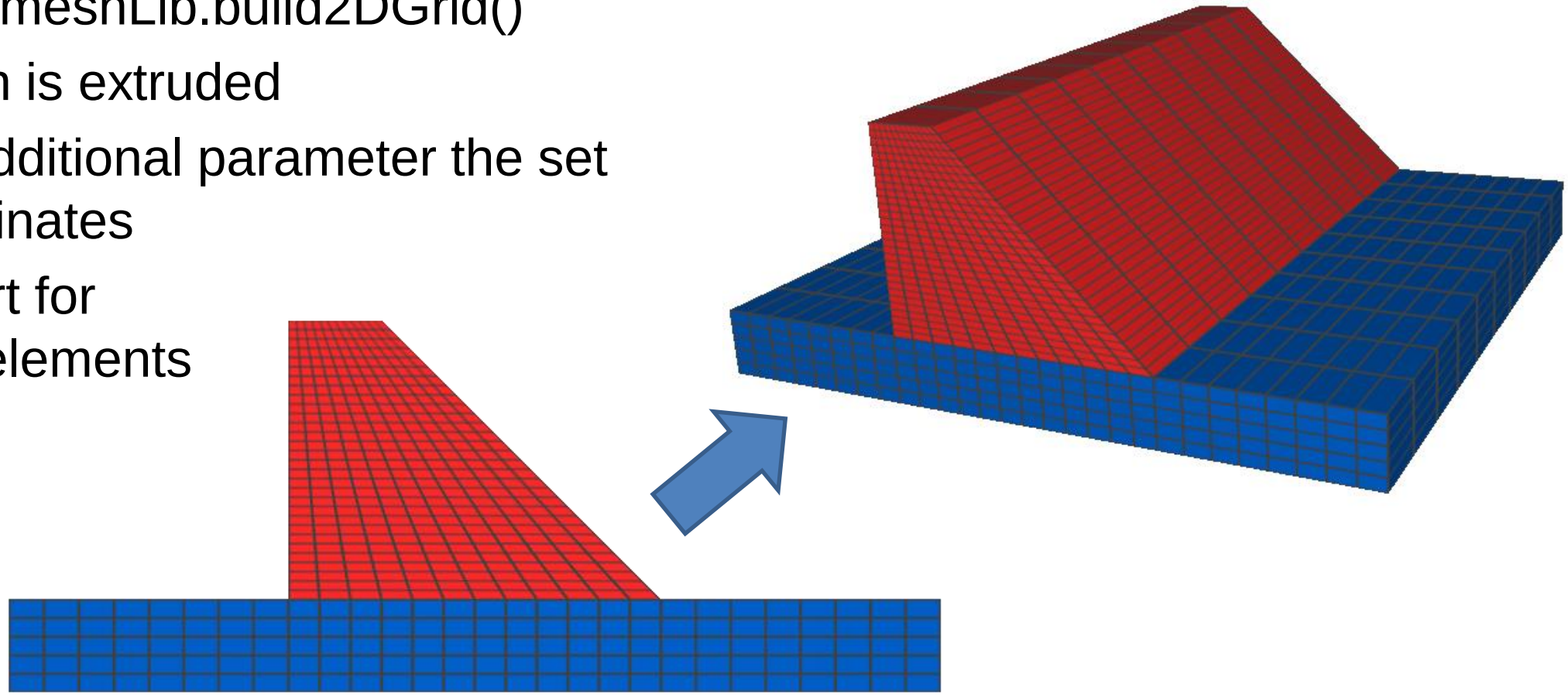
k: Material values are also filled by the supplied user function

III - CREATING A NEW 3D MESH

Creating a 3D mesh

meshLib.**build3DGrid**(elemType, xPoints, yPoints, zPoints, blockList, matList, options)

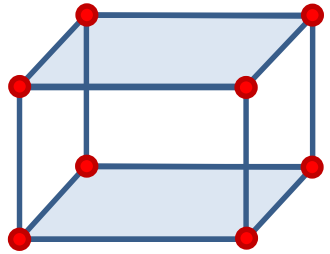
- Analog to meshLib.build2DGrid()
- 2D section is extruded
- Gets as additional parameter the set of z coordinates
- No support for interface elements

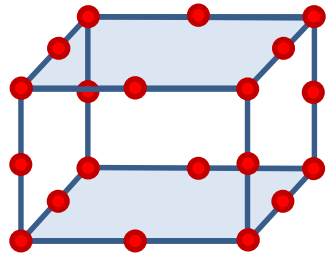


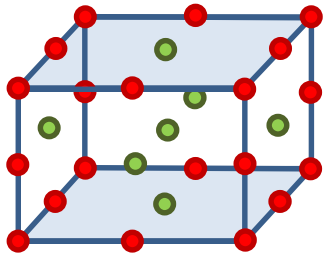
Element types

meshLib.build3DGrid(**elemType**, xPoints, yPoints, zPoints, blockList, matList, options)

- ‘hex8’
- ‘hex20’
- ‘hex27’







Coordinates

meshLib.build3DGrid(elemType, xPoints, yPoints, **zPoints**, blockList, matList, options)

- Extra table with coordinates for grid depth
- Exactly the same semantics as xPoints and yPoints

Blocks, material and options

meshLib.build3DGrid(elemType, xPoints, yPoints, zPoints, **blockList**, **matList**, **options**)

- Exactly the same semantics as in meshLib.build2DGrid()
- Fields related to interface elements are not supported

Example 16

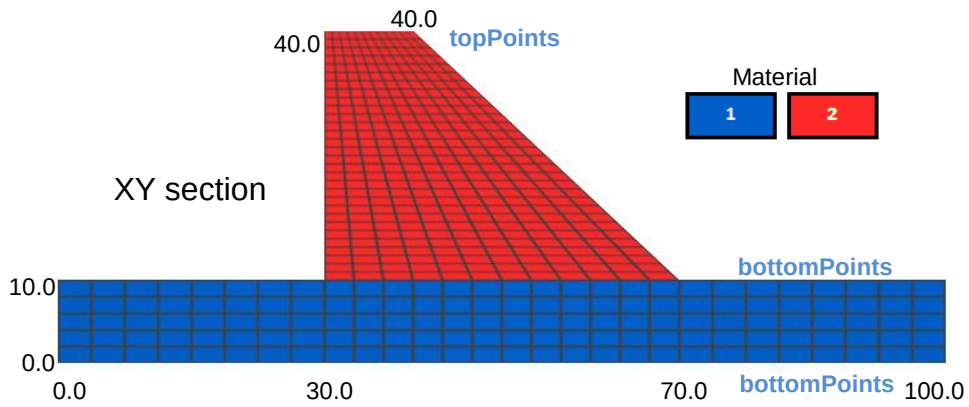
3D version of example 9, without defining custom borders

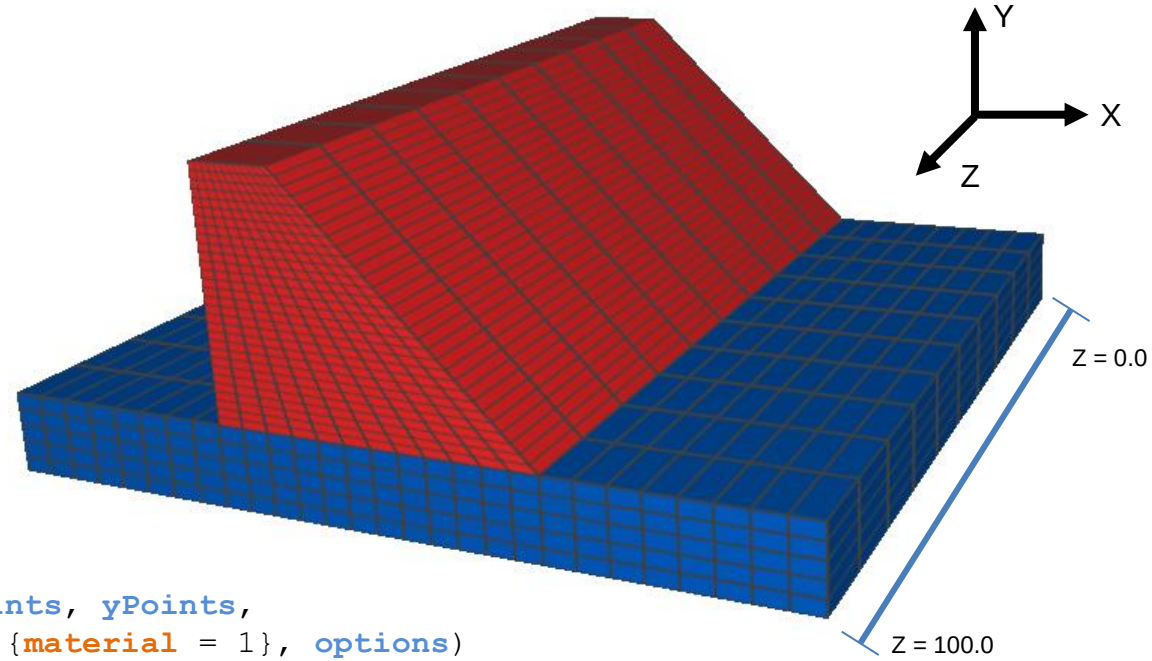
```
local bottomPoints = meshLib.merge(meshLib.regularSpacing(0.0, 30.0, 8),
                                   meshLib.regularSpacing(30.0, 70.0, 12),
                                   meshLib.regularSpacing(70.0, 100.0, 8))

local topPoints    = meshLib.merge(meshLib.regularSpacing(0.0, 30.0, 8),
                                   meshLib.regularSpacing(30.0, 40.0, 12),
                                   meshLib.regularSpacing(40.0, 100.0, 8))

local yPoints      = meshLib.merge(meshLib.regularSpacing( 0.0, 10.0, 5),
                                   meshLib.regularSpacing(10.0, 40.0, 30))

local zPoints      = meshLib.regularSpacing(0.0, 100.0, 10)
```

Points for the “depth” dimension

```
local blocks = { { 0.0, 10.0, { 0.0, 100.0, material = 1}},
                 {10.0, 40.0, {30.0,  70.0, material = 2}},
}

local options = { topxPoints = topPoints,
                  midxPoints = bottomPoints,
                  midxLine   = 10.0,
}
```

```
local nodes, cells, borders = meshLib.build3DGrid('hex8', bottomPoints, yPoints,
                                                   zPoints, blocks, {material = 1}, options)
```





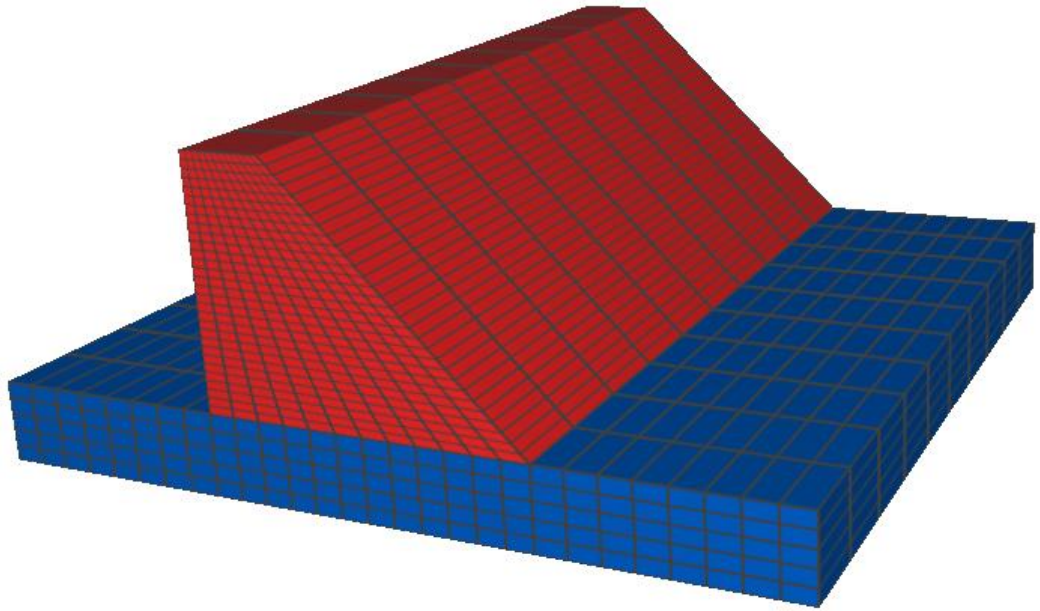
Example 16

3D version of example 9, without defining custom borders

```
local nodes, cells, borders = meshLib.build3DGrid('hex8', bottomPoints, yPoints, zPoints,
                                                   blocks, {material = 1}, options)

Mesh{
  id       = 'mesh16',
  typeName = 'GemaMesh.elem',

  coordinateDim  = 3,
  stateVars      = {'T'},
  cellProperties = {'material'},

  nodeData         = nodes,
  cellData         = cells,
  boundaryFaceData = borders
}
```

Don’t forget that the mesh is now 3D...

... and that borders are now faces and not edges



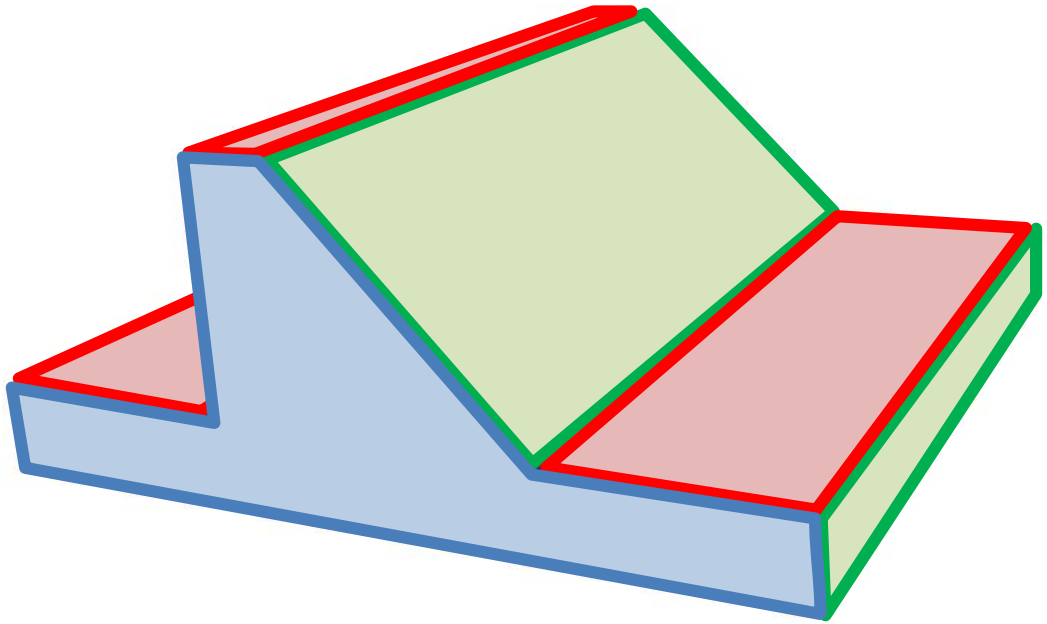
Example 16

3D version of example 9, without defining custom borders

```
local nodes, cells, borders = meshLib.build3DGrid('hex8', bottomPoints, yPoints, zPoints,
                                                   blocks, {material = 1}, options)

Mesh{
  id       = 'mesh16',
  typeName = 'GemaMesh.elem',

  coordinateDim  = 3,
  stateVars      = {'T'},
  cellProperties = {'material'},

  nodeData         = nodes,
  cellData         = cells,
  boundaryFaceData = borders
}
```

By default, `borders` are named:

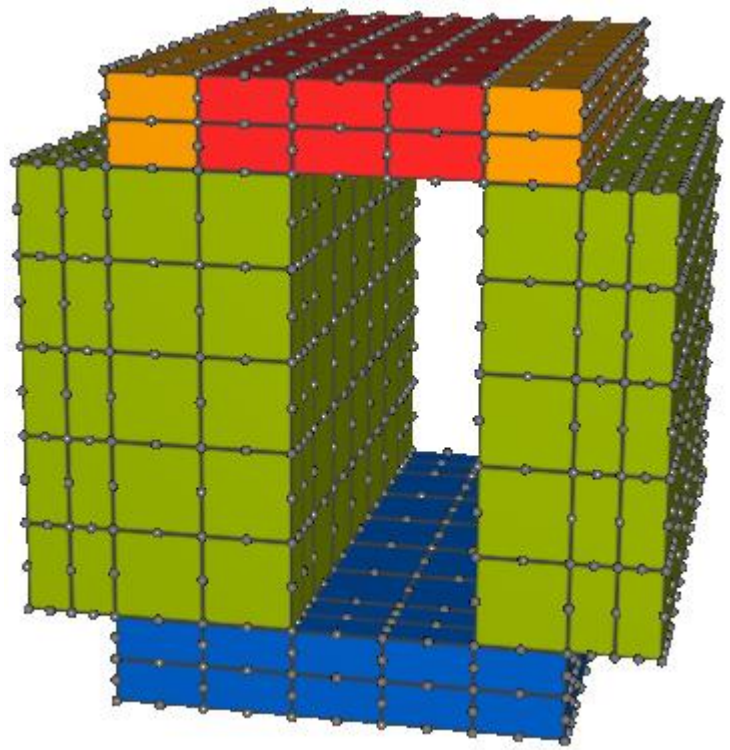
- gridFront,
- gridBack.
- gridLeft,
- gridRight,
- gridTop,
- gridBottom.



Example 17

3D version of example 6, including custom borders and using a quadratic hexahedron

```
-- On this example, grid coordinates are the same for X, Y and Z
-- so we need only one table
local points = {0.0, 0.5, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 6.5, 7.0}

local blocks  = {{0.0, 1.0, {1.0, 6.0, cellGroup = 'a', material = 1}},
                 {1.0, 6.0, {0.0, 3.0, cellGroup = 'b'},
                            {5.0, 7.0, cellGroup = 'c'}},
                 {6.0, 7.0, {1.0, 2.0, cellGroup = 'a', material = 3},
                            {2.0, 5.0, cellGroup = 'a', material = 4},
                            {5.0, 6.0, cellGroup = 'a', material = 3}}
                }


local borderDef = {{'hole', {2, 'right'}, {1, 'top', 3.0, 5.0},
                            {3, 'left'}, {5, 'bottom', 3.0, 5.0}},
                   }


local nodes, cells, borders = meshLib.build3DGrid('hex20',
                                                   points, points, points,
                                                   blocks, {material = 2},
                                                   {borders = borderDef})
```



Example 17

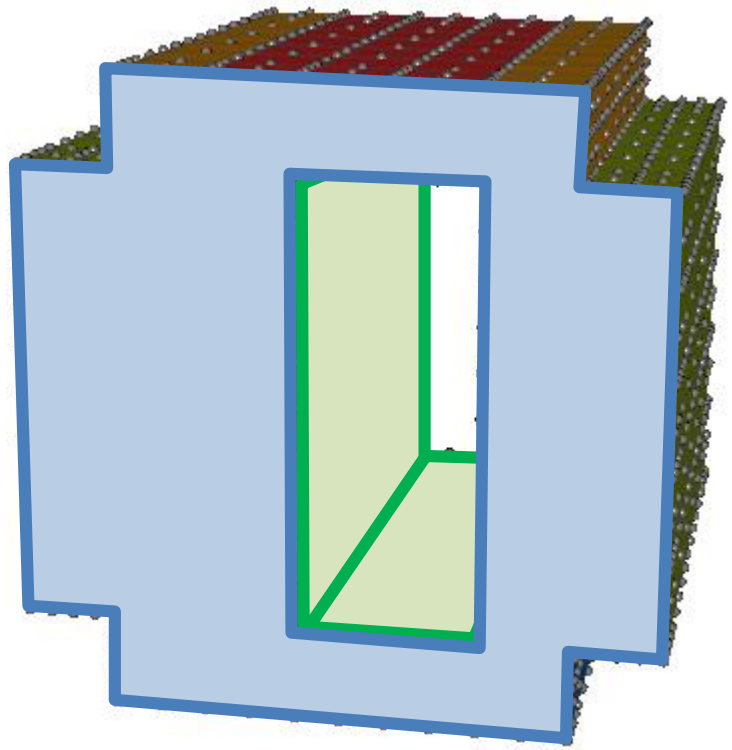
3D version of example 6, including custom borders and using a quadratic hexahedron

```
local borderDef = {{'hole', {2, 'right'}, {1, 'top', 3.0, 5.0},
                            {3, 'left'}, {5, 'bottom', 3.0, 5.0}},
                  }
```

```
local nodes, cells, borders = meshLib.build3DGrid('hex20',
                                                   points, points, points,
                                                   blocks, {material = 2},
                                                   {borders = borderDef})
```

`borders`: hole, gridFront and gridBack

“gridFront” and “gridBack” are always defined, storing the faces on the extruded xy sections



IV – SAVING A MESH MODEL

Saving a mesh model at the Orchestration

- When interface elements are added at the orchestration, the mesh can be saved in a format readable by the model file, saving time when loading the model again. The same technique can also be applied to any other lengthy initialization process.

meshLib.**saveMeshToModelFile**(mesh, fileName, nodeAttributes, cellAttributes, options)

mesh	The mesh to be saved. Can be either the mesh name or a mesh object
filename	The name of the file to be created. Can contain macros such as $SIMULATIONDIR or $SIMULATIONNAME.
nodeAttributes	An optional table naming the set of node attributes and state vars to be saved. The list order MUST place node attributes before state variables and MUST be equal to the order that will be used when loading the file. If equal to nil, all of the mesh attributes and state variables will be saved in an unspecified order.

Saving a mesh model at the Orchestration

meshLib.**saveMeshToModelFile**(mesh, fileName, nodeAttributes, cellAttributes, options)

cellAttributes	Like nodeAttributes, but defining the set of saved cell attributes (all cell properties are always saved).
options	Options controlling saving behavior: - saveDef: If false (default), values equal to the data default value will be saved as nil or skipped if possible. If set to true, all node and cell values will be saved to the file. - evalFunctions: If false (default), function values will be saved as the function name. If set to true, the function will be evaluated and its result saved. - useAcFormat: If false (default), numeric values will be saved using a full precision. If set to true, the data configured format will be used.

Loading a saved mesh model

- Loading the saved mesh can be done by using the standard Lua **dofile**() function.

- Like meshLib.build2DGrid() the call to **dofile**() will return three tables with node, cell and border definitions that can be used when building the mesh.

- Its important that the mesh structure (state vars, node attributes, cell properties and cell attributes) to be compatible with the saved mesh structure.

Example 18

Loads the resulting mesh saved from Example 15 into the new mesh 18.

Model

```
local nodes, cells, borders = dofile('$SIMULATIONDIR/out/mesh18_mesh.lua')

Mesh{
  id         = 'mesh18',
  typeName = 'GemaMesh.elem',
  coordinateDim  = 2,
  stateVars      = {'T'},
  nodeAttributes = {{id = 'na1', description = 'A test node attribute'},
                    {id = 'na2', dim = 2, description = 'Another (vectorial) test node attribute'},
                   },
  cellProperties = {'material', 'another'},
  cellAttributes = {{id = 'ca1', description = 'A test cell attribute'},
                   },
  nodeData = nodes,
  cellData = cells,
  boundaryEdgeData = borders
}
```

Expanded to the directory where the simulation file was loaded.

Orchestration

```
function ProcessScript()

meshLib.saveMeshToModelFile('mesh15', '$SIMULATIONDIR/out/mesh18_mesh.lua',
                            {'na1', 'na2', 'T'}, {'ca1'})


end
```

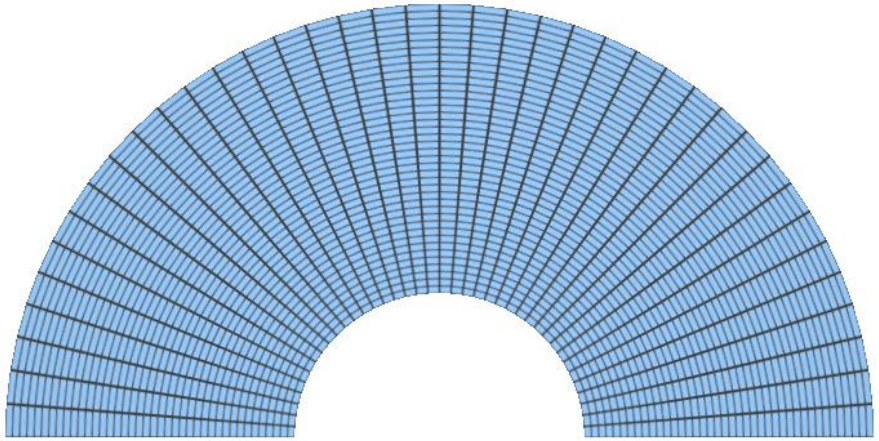
V – PARAMETRIC MESHES

Creating Meshes for Parametric Surfaces

- To create a mesh for a parametric surface, we can use xPoints, yPoints (and zPoints) to create the grid in normalized coordinates and then transform those coordinates from the parametric space to (x, y) or (x, y, z) coordinates through the “nodef” function.

- As seen on examples 7 and 15, this function is generally used to provide custom initialization values for node attributes, but it can also be used to change node coordinates.

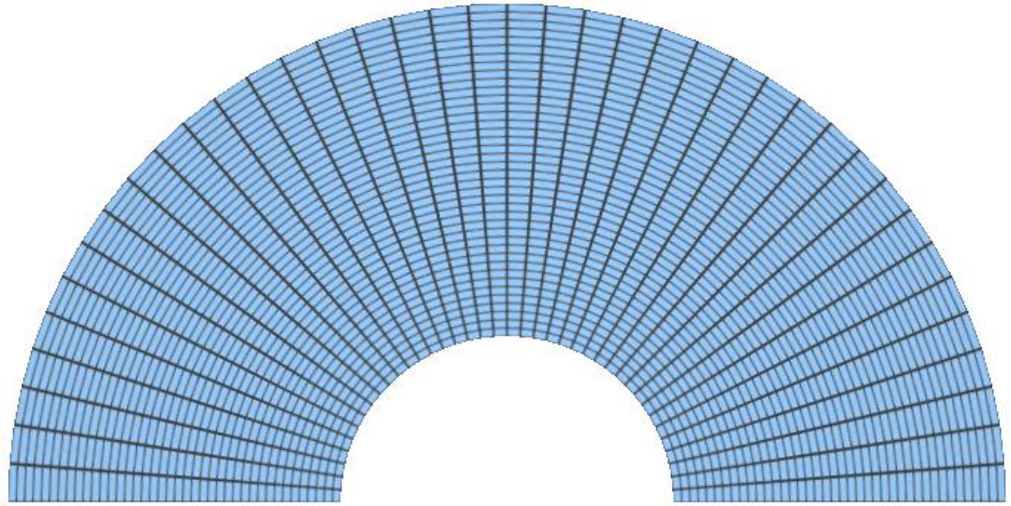
- 3D meshes based on a parametric XY section can follow the same principle



Example 19

A parametric plane transformed into a half-disk by a simple polar transformation using the node initialization function to transform node coordinates

```
local function transfromCoordinates(nodeDef)
  -- Get the original node coordinates, from 0.0 to 1.0
  local r, s = nodeDef[1], nodeDef[2]

  -- Transforms them to the desired x, y values and
  -- updates the node coordinates by writing back to
  -- nodeDef
  r = 5 + r*10      -- r from 5 to 15
  s = s * math.pi   -- s from 0 to PI

  nodeDef[1] = r * math.cos(s)
  nodeDef[2] = r * math.sin(s)
end
```



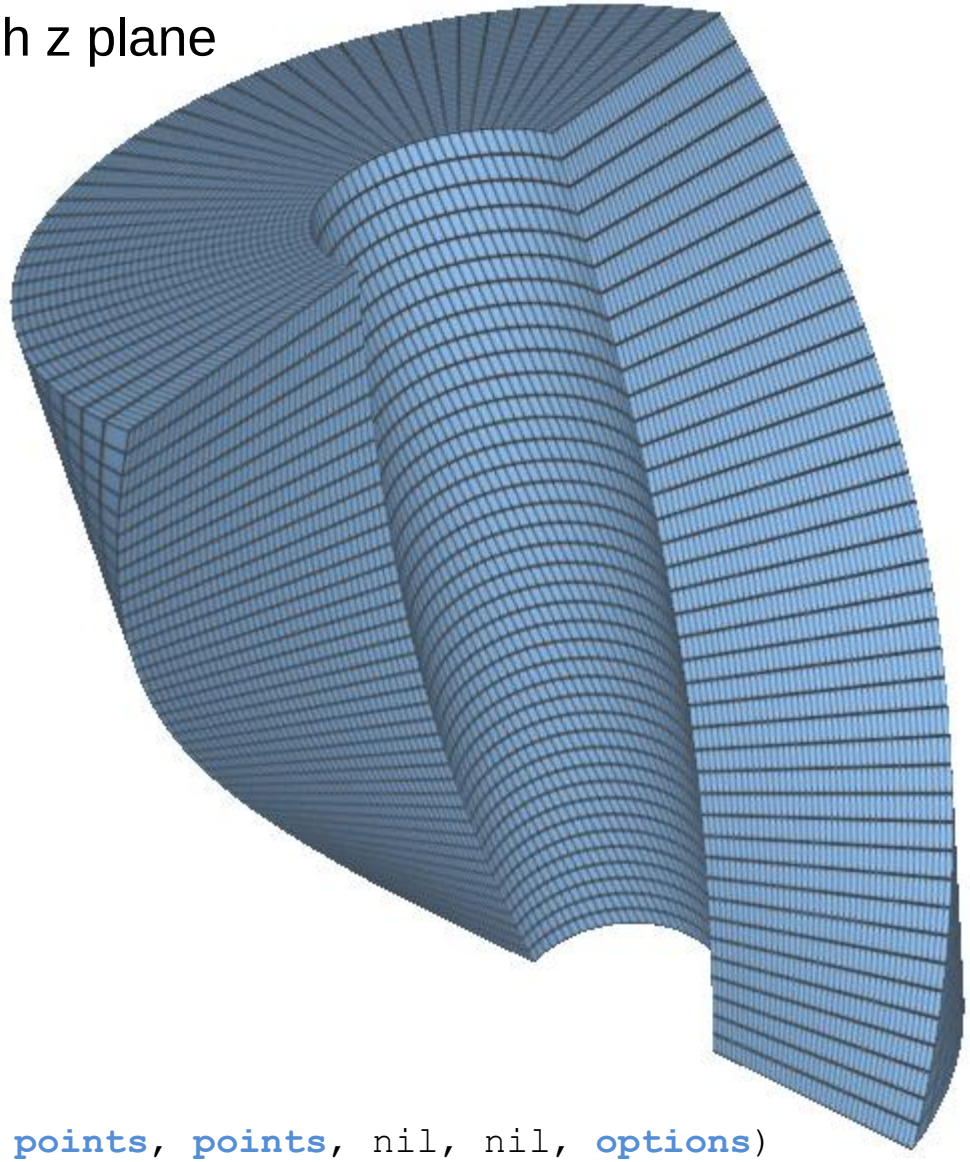
```
local points  = meshLib.regularSpacing(0.0, 1.0, 40)
local options = {nodef = transfromCoordinates}
```

Coordinates varying from 0.0 to 1.0

```
local nodes, cells, borders = meshLib.build2DGrid('quad4', points, points, nil, nil, options)
```

Example 20

3D version of example 19, also adding a small rotation for each z plane



```
local function transfromCoordinates3D(nodeDef)
  local r, s, t = nodeDef[1], nodeDef[2], nodeDef[3]

  local rot = t * math.pi/2.0 -- Rottaion angle for each XY slice

  r = 5 + r*10    -- r from 5 to 15
  s = s * math.pi -- s from 0 to PI
  t = t * 40      -- t from 0 to 40

  local x = r * math.cos(s)
  local y = r * math.sin(s)
  local z = t

  nodeDef[1] = x * math.cos(rot) - y * math.sin(rot)
  nodeDef[2] = x * math.sin(rot) + y * math.cos(rot)
  nodeDef[3] = z
end

local points  = meshLib.regularSpacing(0.0, 1.0, 40)
local options = {nodef = transfromCoordinates3D}

local nodes, cells, borders = meshLib.build3DGrid('hex8', points, points, points, nil, nil, options)
```