Veamy

An extensible object-oriented C++ library
for the virtual element method

Veamy Primer

Version 2.0

Rev. 0
January, 2018

Copyright and License

CEMCEN - Center for Modern Computational Engineering
Department of Computer Science
Department of Mechanical Engineering
Facultad de Ciencias Físicas y Matemáticas
Universidad de Chile
Av. Beauchef 851, Santiago 8370456, Chile

TABLE OF CONTENTS

1 New and updated feature summary

From Veamy v1.1.1 to Veamy 2.0:

- Add documentation to the source code.
- Implement VEM for the two-dimensional Poisson problem.
- Implement Feamy, a FEM module that uses three-node triangular finite elements for the solution of the two dimensional linear elastostatic problem.
- Add methods to compute the L^2-norm and H^1-seminorm of the error.
- Improve the in-built polygonal mesh generator.
- Change to Eigen's sparse solver for the solution of the system of linear equations.
- Add additional test files.
- New simplified methods to impose essential and Neumann boundary conditions.
- Fix several bugs.

From Veamy 1.0 to Veamy v1.1.1:

- Add documentation.
- Add method to include custom precision for printing output data.
- Add plane stress material formulation.
- Update installation instructions.
- Include more tests and mesh examples.
- Fix several bugs

2 Features of Veamy

Veamy is an open source C++ library that implements the virtual element method. The current release of this library allows the solution of the two-dimensional linear elastostatic problem and the two-dimensional Poisson problem. The two-dimensional linear elastostatic problem can also be solved using the standard three-node finite element triangle. For this, a module called Feamy is available within Veamy.

Features:

- Includes its own mesher based on the computation of the constrained Voronoi diagram. The meshes can be created in arbitrary two-dimensional domains, with or without holes, with procedurally generated points.
- Meshes can also be read from OFF-style text files.
- Allows easy input of boundary conditions by constraining domain segments and nodes.
- The results of the computation can be either written into a file or used directly.
- PolyMesher meshes and boundary conditions can be read straightforwardly in Veamy to solve problems using the VEM.

3 Source code

All the information related to Veamy and its source code is available on the web:

http://camlab.cl/research/software/veamy/

Download the code before proceeding with the rest of this primer.

4 Up and running with Veamy

Veamy has been tested on Unix-like machines only. First of all, make sure that CMake is available in your machine. If it is not, install it before proceeding with the rest of this primer. To install CMake on Ubuntu machines, on a terminal type and execute:

```
sudo apt-get install cmake
```

Unpack the code to a folder of your choice. Fig. 1 shows the content of Veamy that was unpacked to "/home/Software/"

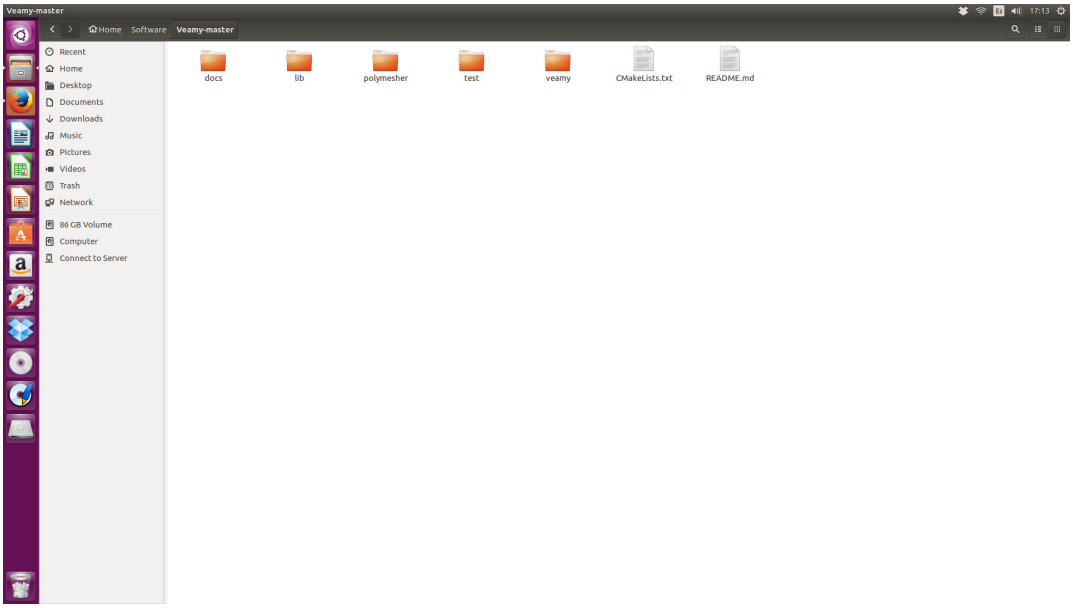


Fig. 1: Veamy source code.

Go inside "test" folder of Veamy's root directory (see Fig. 2). This test folder is where the main C++ setup file implementing a problem of interest must be placed. In this example, a "cantilever beam subjected to a parabolic end load" will be solved in Veamy. This problem is part of the numerical examples provided in:

A. Ortiz-Bernardin, C. Alvarez, N. Hitschfeld-Kahler, A. Russo, R. Silva, E. Olate-Sanzana. Veamy: an extensible object-oriented C++ library for the virtual element method. arXiv:1708.03438 [cs.MS]

You may consult the details of the geometry and boundary conditions therein as in this primer we only refer to the final main C++ setup file to run the example.

The implementation of the cantilever beam subjected to a parabolic end load is provided in the main C++ setup file named "ParabolicMain.cpp" (see Fig. 2).

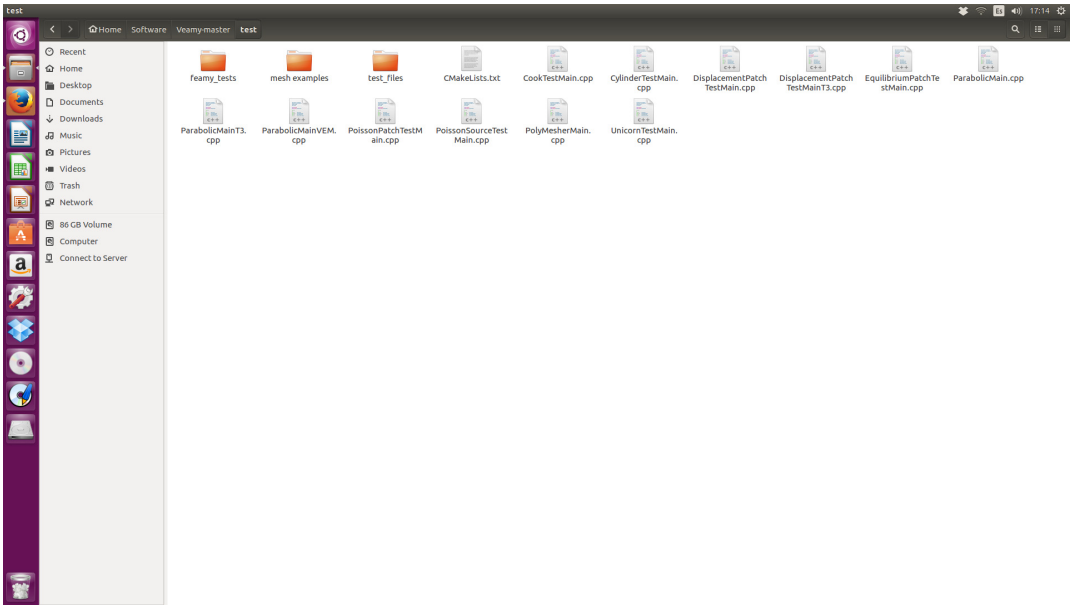


Fig. 2: Veamy's test folder. The main C++ setup file implementing a problem of interest must be placed in this folder. Several main setup C++ files are shown. In this part of the primer, the C++ file "ParabolicMain.cpp" will be used.

Open "ParabolicMain.cpp" file. If you are interested, browse the code in this file to realize how a problem implementation is setup in Veamy. To run this problem is important to update the folder where the output files will be stored. In order to specify the output folder, check the instructions that are provided as comments in "ParabolicMain.cpp" (see Fig. 3). Modify accordingly, save and close the setup file.

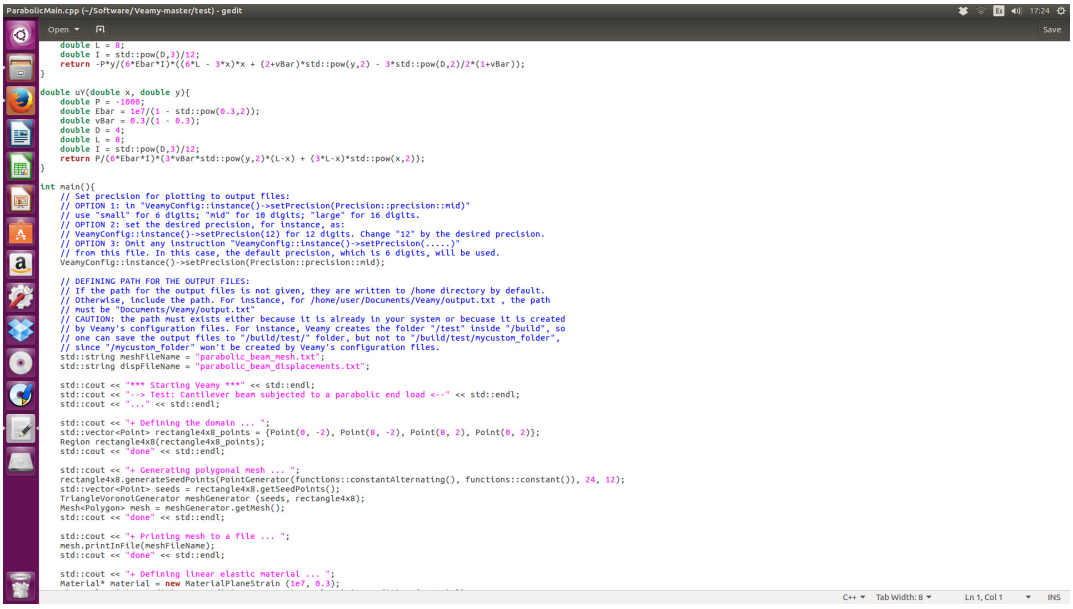


Fig. 3: Main C++ setup file for the cantilever beam subjected to a parabolic end load.

Now, the test folder contains a file named "CMakeLists.txt". This file is important because it controls which main C++ setup file will be processed in Veamy. The file inside "test" folder is shown in Fig. 4.

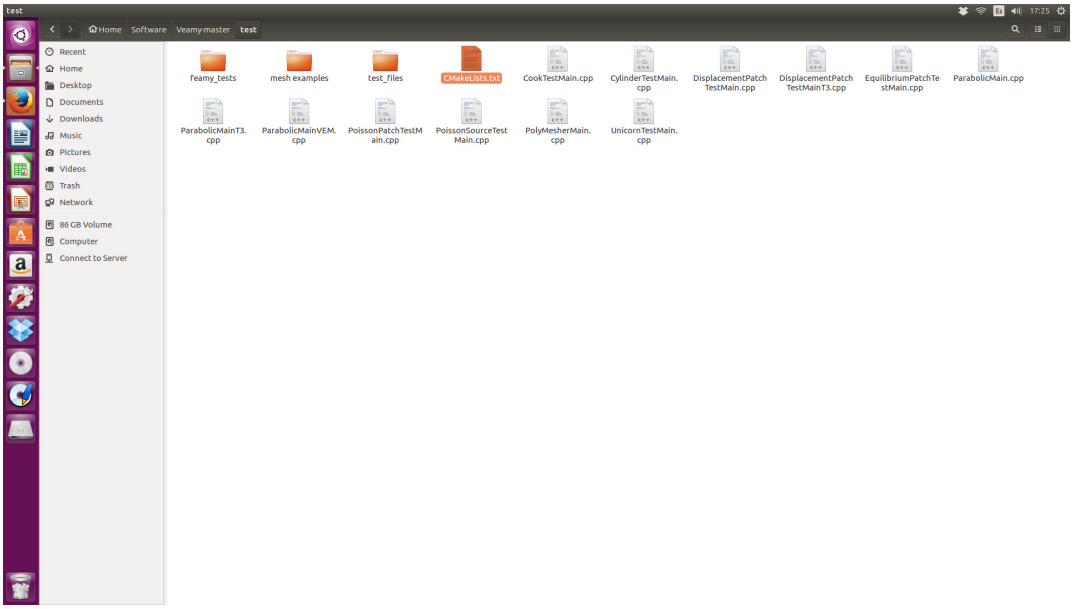


Fig. 4: CMakeLists.txt is located in test folder and controls which main C++ setup file is processed in Veamy.

Open "CMakeLists.txt" and on the highlighted zone, write the name of the main C++ setup problem file, in this case, "ParabolicMain.cpp," as shown in Fig. 5. Save and close the file.

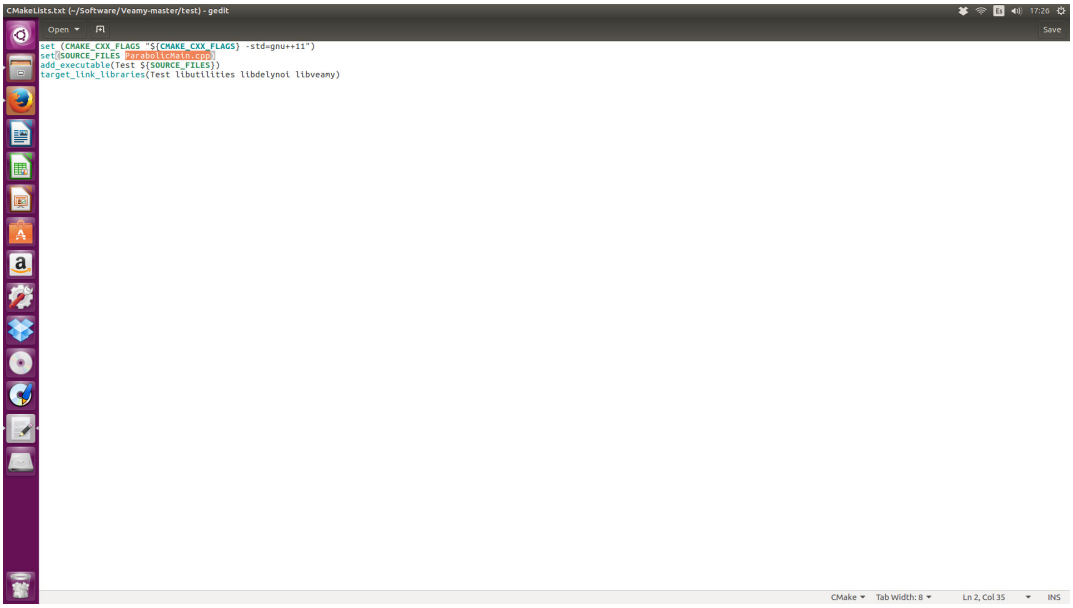


Fig. 5: Open "CMakeLists.txt" and on the highlighted zone, write the name of the main C++ setup problem file.

Go back to the Veamy's root folder and there create a folder "build" (Fig. 6).

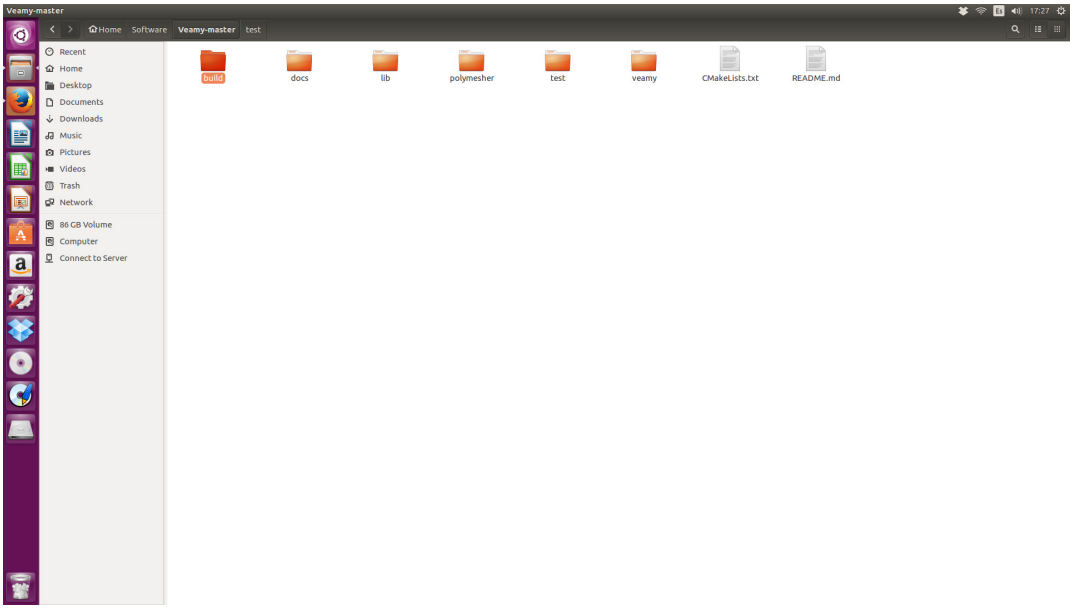


Fig. 6: In Veamy's root folder create the folder "build".

Go inside the "build" folder and on a terminal, type and execute:

```
cmake ..
```

to create the makefiles. Then, to compile the program, on a terminal type and execute:

```
make
```

Several files are created. Also, another folder called "test" is created inside "build". The executable of the test problem is stored in this "test" folder and is called "Test". Go inside "build/test/" folder (Fig. 7) and, on a terminal, type and execute:

```
./Test
```

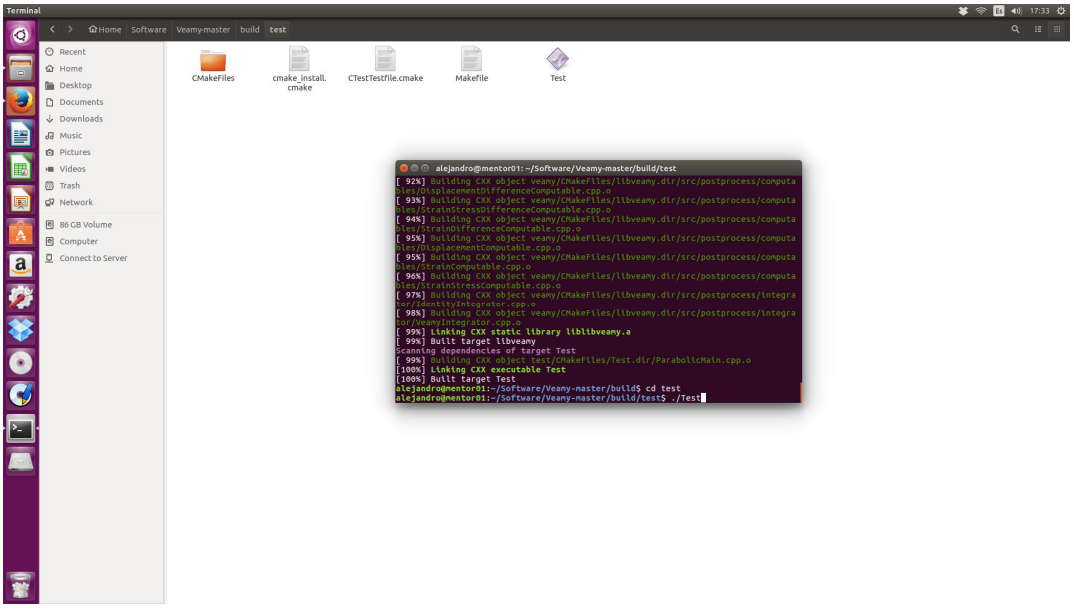


Fig. 7: Go inside "build/test/" folder and on a terminal type and execute ./Test

While running, Veamy prints out some messages on the screen indicating the progress of the simulation, as shown in Fig. 8.

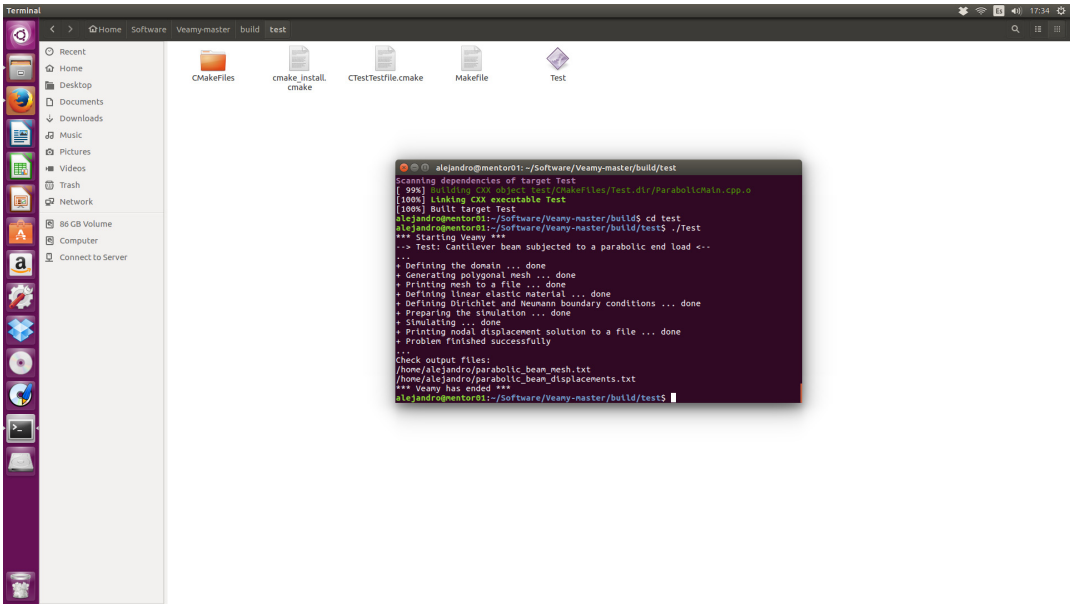


Fig. 8: Veamy prints out some messages while running the simulation.

The last lines of the printed out messages indicate the location of the output folders. The output files contain the mesh and the nodal displacement solution. The mesh can be visualized using the MATLAB function "plotPolyMesh.m" that is inside the folder "Veamy_root_directory/lib/visualization/" or if you want to visualize both the mesh and the nodal solution, use the MATLAB function "plotPolyMeshDisplacements.m" for the elasticity problem or "plotPolyMeshScalarField.m" for the Poisson problem that are also available in the "visualization" folder (see Fig. 9). The plots for the beam subjected to a parabolic end load are shown in Fig. 10.

Fig. 9: Use "plotPolyMesh.m" to visualize the mesh, or "plotPolyMeshDisplacements.m" or "plotPolyMeshScalarField.m" to visualize both the mesh and the nodal solution. These files are located inside the folder "Veamy_root_directory/lib/visualization/".

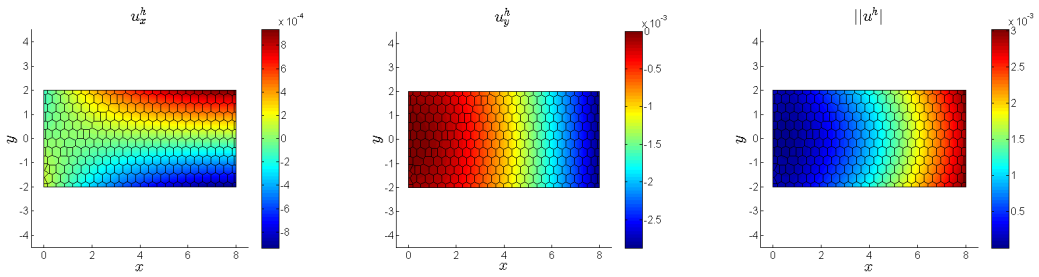


Fig. 10: Mesh and nodal displacements for the beam problem are plotted using the "plotPolyMeshDisplacements.m" MATLAB function.

5 Using a PolyMesher mesh and boundary conditions in Veamy

Now, we show how to use a mesh and boundary conditions obtained from PolyMesher. This primer assumes that the user knows how to use PolyMesher. This problem is part of the numerical examples provided in:

A. Ortiz-Bernardin, C. Alvarez, N. Hitschfeld-Kahler, A. Russo, R. Silva, E. Olate-Sanzana. Veamy: an extensible object-oriented C++ library for the virtual element method. arXiv:1708.03438 [cs.MS]

You may consult the details of the geometry and boundary conditions therein as in this primer we only refer to the final main C++ setup file to run the example.

The procedure is straightforward. In PolyMesher add a call to the MATLAB function "PolyMesher2Veamy.m". This function is located in "Veamy_root_directory/polymesher/", as shown in Fig. 11. The call to this function is done on the last line of the "PolyMesher.m" function, as shown in Fig. 12. After defining a model and boundary conditions, and performing the meshing copy the file "polymesher2veamy.txt" to a folder of your choice to be used in Veamy. In the source code of Veamy, the example file containing the PolyMesher mesh and boundary conditions is located inside the folder "Veamy_root_directory/test/test_files/".

The implementation of the PolyMesher to Veamy example is provided in the main C++ setup file named "PolyMesherMain.cpp" (see Fig. 13). This setup file as usual is inside "Veamy_root_directory/test/" folder. Go to this folder and open "PolyMesherMain.cpp" (see Fig. 13). Explore this file to see details about its implementation. The function that reads the PolyMesher mesh and boundary conditions is "initProblemFromFile". You will have to provide the path to the folder where the PolyMesher mesh and boundary conditions are located. Update the output folders (check the instructions that are provided as comments). Modify the paths accordingly, save and close the setup file.

Fig. 11: The MATLAB function "PolyMesher2Veamy.m" is located in folder "Veamy_root_directory/polymesher/".

PolyMesher.m

```
%      elements written in Matlab", Struct Multidisc Optim, 2012,         %
%      DOI 10.1007/s00158-011-0706-z                                      %
%                                                                         %
% Ref2: A Pereira, C Talischi, GH Paulino, IFM Menezes, MS Carvalho,      %
%      "Implementation of fluid flow topology optimization in PolyTop",   %
%      Struct Multidisc Optim, 2013, DOI XX.XXXX/XXXXXX-XXX-XXX-X         %
%-------------------------------------------------------------------------%
function [Node,Element,Supp,Load,P] = PolyMesher(Domain,NElem,MaxIter,P)
if ~exist('P','var'), P=PolyMshr_RndPtSet(NElem,Domain); end
NElem = size(P,1);
Tol=5e-6; It=0; Err=1; c=1.5;
BdBox = Domain('BdBox'); PFix = Domain('PFix');
Area = (BdBox(2)-BdBox(1))*(BdBox(4)-BdBox(3));
Pc = P; figure;
while(It<=MaxIter && Err>Tol)
  Alpha = c*sqrt(Area/NElem);
  P = Pc; %Lloyd's update
  R_P = PolyMshr_Rflct(P,NElem,Domain,Alpha);   %Generate the reflections
  [P,R_P] = PolyMshr_FixedPoints(P,R_P,PFix); % Fixed Points
  [Node,Element] = voronoin([P;R_P]);           %Construct Voronoi diagram
  [Pc,A] = PolyMshr_CntrdPly(Element,Node,NElem);
  Area = sum(abs(A));
  Err = sqrt(sum((A.^2).*sum((Pc-P).*(Pc-P),2)))*NElem/Area^1.5;
  fprintf('It: %3d   Error: %1.3e\n',It,Err); It=It+1;
  if NElem<=2000, PolyMshr_PlotMsh(Node,Element,NElem); end;
end
[Node,Element] = PolyMshr_ExtrNds(NElem,Node,Element);  %Extract node list
[Node,Element] = PolyMshr_CllpsEdgs(Node,Element,0.1);  %Remove small edges
[Node,Element] = PolyMshr_RsqsNds(Node,Element);        %Reoder Nodes
BC=Domain('BC',{Node,Element}); Supp=BC{1}; Load=BC{2}; %Recover BC arrays
PolyMshr_PlotMsh(Node,Element,NElem,Supp,Load);         %Plot mesh and BCs
PolyMesher2Veamy(Node,Element,NElem,Supp,Load);         %Plot mesh to a Veamy mesh format
%-------------------------------------------------- GENERATE RANDOM POINTSET
```

Fig. 12: Call to "PolyMesher2Veamy.m" in "PolyMesher.m" is done on its last line.

Fig. 13: Main C++ setup file for the PolyMesher mesh and boundary condition example.

From now on, the procedure to run the PolyMesher problem in Veamy is identical to the one performed for the beam problem.

Go inside the "Veamy_root_directory/build/" folder and on a terminal, type and execute to update the makefiles:

```
cmake ..
```

Then, to compile the program, on a terminal type and execute:

```
make
```

If this procedure has been done several times before, many of the libraries are likely to be already compiled, so the compilation procedure is quite short in comparison with the first time compilation. The executable of the test problem is stored in the "build/test/" folder and is called "Test". Go inside "build/test/" folder and, on a terminal, type and execute:

```
./Test
```

The output screen for the PolyMesher problem is shown in Fig. 14. The last lines of the printed out messages indicate the location of the output folders. The output files contain the mesh and the nodal displacement solution. The mesh can be visualized using the MATLAB function "plotPolyMesh.m" that is inside folder "Veamy_root_directory/lib/visualization/" or if you want to visualize both the mesh and the nodal displacement solution, use the MATLAB function "plotPolyMeshDisplacements.m" that is also available in the "visualization" folder. The mesh and the nodal displacements for the PolyMesher example are shown in Fig. 15.

Fig. 14: Output screen for the PolyMesher example. Use "plotPolyMesh.m" to visualize the mesh or "plotPolyMeshDisplacements.m" to visualize both the mesh and the nodal displacement solution. Both MATLAB files are located inside folder "Veamy_root_directory/lib/visualization/".

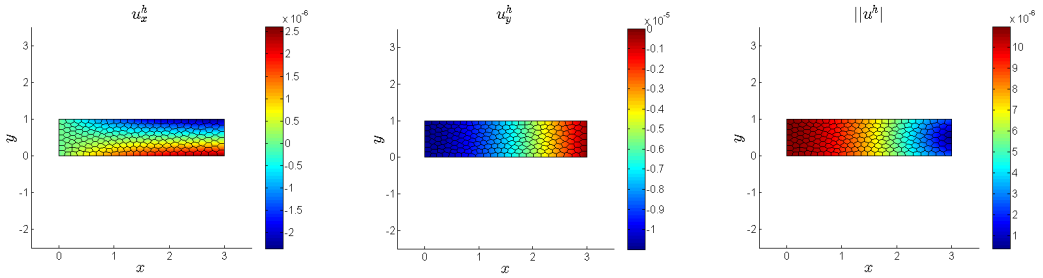


Fig. 15: Nodal displacements for the PolyMesher example are plotted using the "plotPolyMeshDisplacements.m" MATLAB function.

6 Using a generic mesh file

Reading a generic mesh file is very similar to the process of using a PolyMesher mesh. The only difference is that boundary conditions are not provided with the mesh file. That is, the mesh is read from a file, but the boundary conditions must be provided in Veamy similarly as done in the cantilever beam problem of Section 4. An example of this is provided in the main C++ setup file "Veamy_root_directory/test/EquilibriumPatchTestMain.cpp". In this main C++ setup file, the external test mesh, which is the file "Veamy_root_directory/test/test_files/equilibriumTest_mesh.txt", is read by the function "createFromFile":

```
std::string externalMeshFileName =
        "Software/Veamy-master/test/test_files/equilibriumTest_mesh.txt";
Mesh<Polygon> mesh;
mesh.createFromFile(externalMeshFileName);
```

As you can confirm by exploring the external mesh file “equilibriumTest_mesh.txt”, it contains the nodal coordinates of the mesh and the element connectivity.

7 Additional examples

These additional examples require the user to have read the previous sections of this primer.

7.1 Perforated Cook´s membrane

The implementation of the perforated Cook’s membrane is provided in the main C++ set-up file named “CookTestMain.cpp”. This setup file as usual is inside “Veamy_root_directory/test/” folder. Go to this folder and open “CookTestMain.cpp” (see Fig. 16). Explore this file to understand its implementation. Be sure you update the path to the output files. The important lines of code are highlighted. They provide the information for the four points that define the geometry and three circular holes on it.

This problem is part of the numerical examples provided in:

A. Ortiz-Bernardin, C. Alvarez, N. Hitschfeld-Kahler, A. Russo, R. Silva, E. Olate-Sanzana. Veamy: an extensible object-oriented C++ library for the virtual element method. arXiv:1708.03438 [cs.MS]

You may consult the details of the geometry and boundary conditions therein as in this primer we only refer to the final main C++ setup file to run the example.

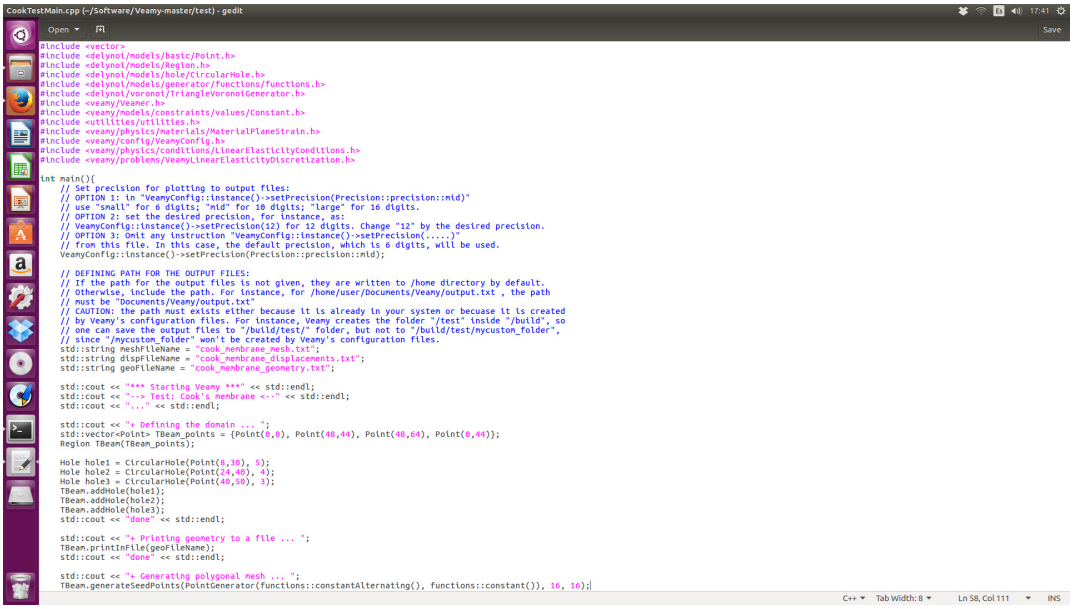


Fig. 16: Main C++ setup file for the perforated Cook’s membrane example.

In order to run the test, follow the same steps described in the previous examples. Once you have compiled the problem, go inside “build/test/” folder and, on a terminal, type and execute:

```
./Test
```

The output files are visualized, as in the previous examples, using the MATLAB function “plotPolyMeshDisplacements.m”. The plots are shown in Fig. 17.

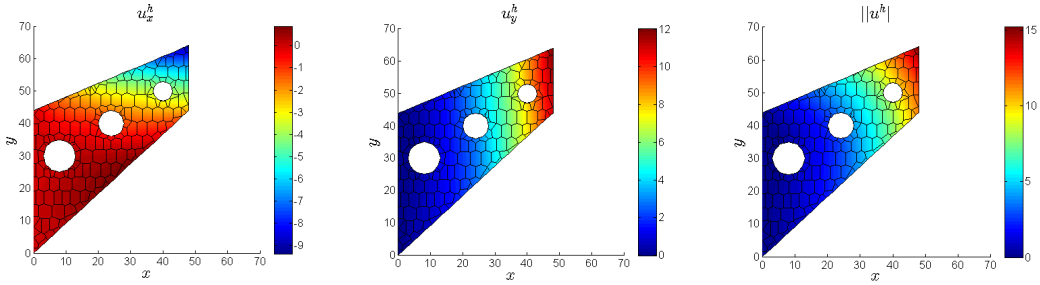


Fig. 17: Nodal displacements for the perforated Cook's membrane problem are plotted using the "plotPolyMeshDisplacements.m" MATLAB function.

7.2 A toy example

In this example, a Unicorn loaded on its back and fixed at its feet is solved using Veamy. This problem is part of the numerical examples provided in:

A. Ortiz-Bernardin, C. Alvarez, N. Hitschfeld-Kahler, A. Russo, R. Silva, E. Olate-Sanzana. Veamy: an extensible object-oriented C++ library for the virtual element method. arXiv:1708.03438 [cs.MS]

You may consult the details of the geometry and boundary conditions therein as in this primer we only refer to the final main C++ setup file to run the example.

The implementation of the Unicorn problem is provided in the main C++ setup file named "UnicornTestMain.cpp". This setup file as usual is inside "Veamy_root_directory/test/" folder. Go to this folder and open "UnicornTestMain.cpp" (see Fig. 18). Be sure you update the path to the output files. The important lines of code are highlighted. They provide the information for the points that define the boundary of the Unicorn.

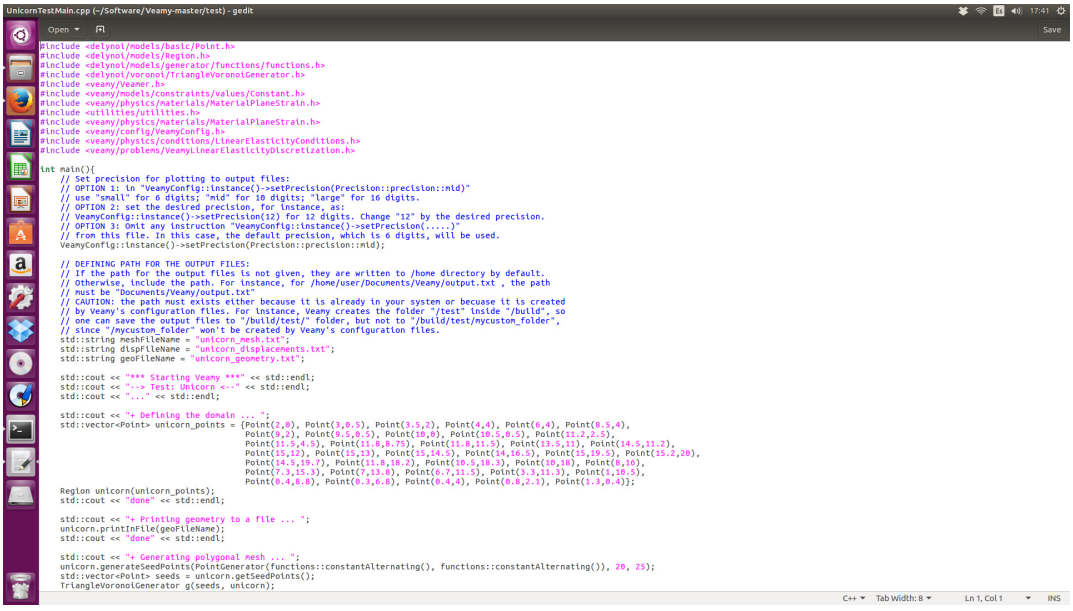


Fig. 18: Main C++ setup file for the Unicorn example.

In order to run the test, follow the same steps described in the previous examples. Once you have compiled the problem, go inside “build/test/” folder and, on a terminal, type and execute:

```
./Test
```

The output files are visualized, as in the previous examples, using the MATLAB function “plotPolyMeshDisplacements.m”. The plots are shown in Fig. 19.

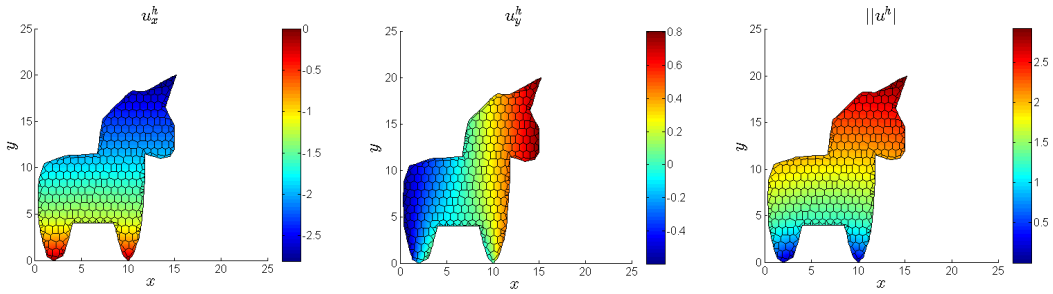


Fig. 19: Nodal displacements for the Unicorn problem are plotted using the “plotPolyMeshDisplacements.m” MATLAB function.

8 Geometry definition and mesh generation

Geometry definition and polygonal mesh generation in Veamy are handled using Delynoi, an object oriented C++ library for the generation of polygonal meshes that is based on the constrained Voronoi diagram. Delynoi depends on two external open source libraries, whose code is included in the repository:

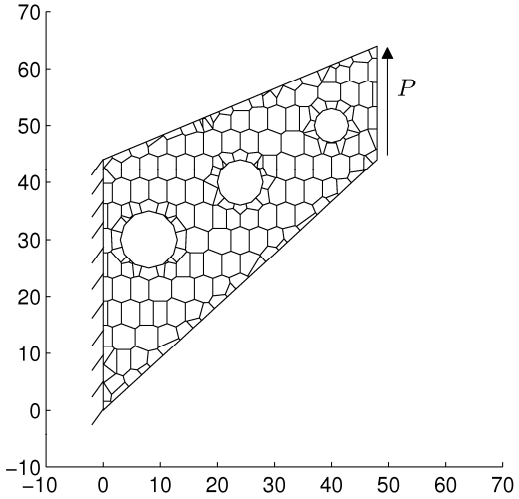
- Triangle - A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator.
- Clipper - an open source freeware library for clipping and offsetting lines and polygons.

All the information related to Delynoi and its source code is available on the web:

http://camlab.cl/research/software/delynoi/

Nevertheless, few examples are presented in what follows.

Example: Perforated Cook's membrane



Define the corner points of the Cook's membrane

```
std::vector<Point> TBeam_points = {Point(0,0), Point(48,44), Point(48,64), Point(0,44)};
```

Define the region formed by the points

```
Region TBeam(TBeam_points);
```

Define holes

```
Hole hole1 = CircularHole(Point(8,30), 5); Hole hole2 = CircularHole(Point(24,40), 4);
Hole hole3 = CircularHole(Point(40,50), 3);
```

Add holes to the region

```
TBeam.addHole(hole1); TBeam.addHole(hole2); TBeam.addHole(hole3);
```

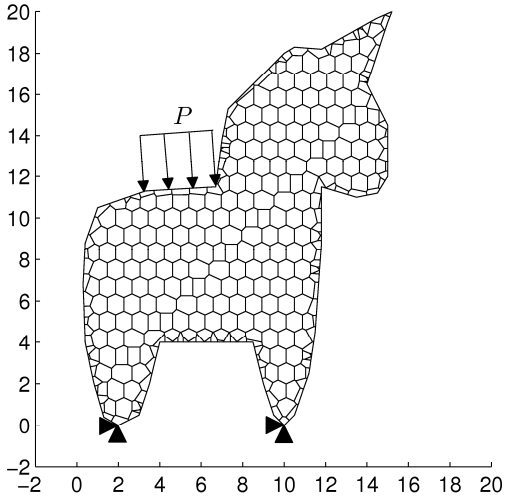
Generate seeds points in the region

```
TBeam.generateSeedPoints(PointGenerator(functions::constantAlternating(),
            functions::constant()), 16, 16);
std::vector<Point> seeds = TBeam.getSeedPoints();
```

Compute the polygonal mesh using the constrained Voronoi diagram

```
TriangleVoronoiGenerator g(seeds, TBeam);
Mesh<Polygon> mesh = g.getMesh();
```

Example: Unicorn



Define the points of the Unicorn boundary

```
std::vector<Point> unicorn_points = {Point(2,0), Point(3,0.5), Point(3.5,2), Point(4,4),
Point(6,4), Point(8.5,4), Point(9,2), Point(9.5,0.5), Point(10,0), Point(10.5,0.5),
Point(11.2,2.5), Point(11.5,4.5), Point(11.8,8.75), Point(11.8,11.5), Point(13.5,11),
Point(14.5,11.2), Point(15,12), Point(15,13), Point(15,14.5), Point(14,16.5), Point(15,19.5),
Point(15.2,20), Point(14.5,19.7), Point(11.8,18.2), Point(10.5,18.3), Point(10,18),
Point(8,16), Point(7.3,15.3), Point(7,13.8), Point(6.7,11.5), Point(3.3,11.3), Point(1,10.5),
Point(0.4,8.8), Point(0.3,6.8), Point(0.4,4), Point(0.8,2.1), Point(1.3,0.4)};
```

Define the region formed by the points

```
Region unicorn(unicorn_points);
```

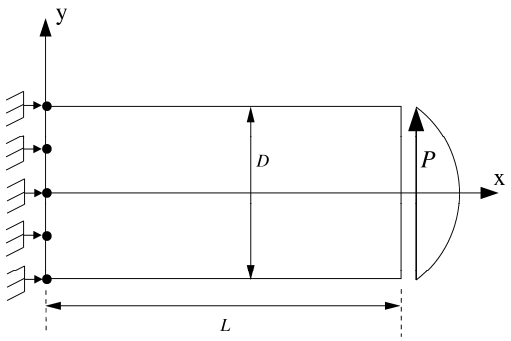
Generate seeds points in the region

```
unicorn.generateSeedPoints(PointGenerator(functions::constantAlternating(),
            functions::constantAlternating()), 20, 25);
std::vector<Point> seeds = unicorn.getSeedPoints();
```

Compute the polygonal mesh using the constrained Voronoi diagram

```
TriangleVoronoiGenerator g(seeds, unicorn);
Mesh<Polygon> mesh = g.getMesh();
```

Example: Cantilever beam subjected to a parabolic end load



Define the corner points of the beam

```
std::vector<Point> rectangle4x8_points={Point(0, -2), Point(8, -2), Point(8, 2), Point(0, 2)};
```

Define the region formed by the points

```
Region rectangle4x8(rectangle4x8_points);
```

Generate seeds points in the region

```
rectangle4x8.generateSeedPoints(PointGenerator(functions::constantAlternating(),
                functions::constant()), 24, 12);
std::vector<Point> seeds = rectangle4x8.getSeedPoints();
```

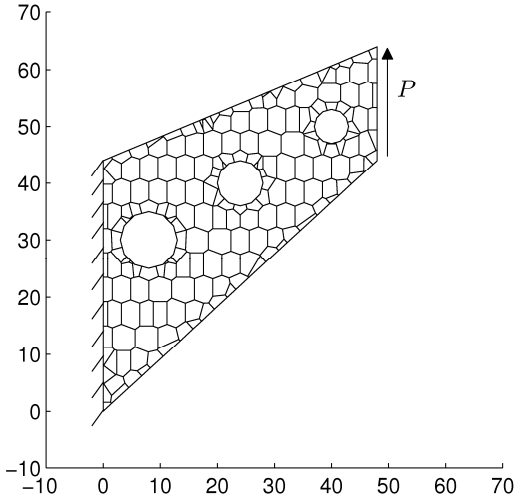
Compute the polygonal mesh using the constrained Voronoi diagram

```
TriangleVoronoiGenerator g(seeds, rectangle4x8);
Mesh<Polygon> mesh = g.getMesh();
```

9 Problem conditions: material definition, body/source terms, essential and natural boundary conditions

The material, body/source terms and boundary conditions are declared as part of an object of a class pertaining to the type of problem (linear elasticity or Poisson). Available materials are isotropic linear elastic (plane strain and plane stress). Boundary conditions are assigned by constraining domain segments and nodes. Some examples follow.

Example: Perforated Cook's membrane



Elastic Material

```
Material* material = new MaterialPlaneStrain(240, 0.3); // Also available: MaterialPlaneStress
LinearElasticityConditions* conditions = new LinearElasticityConditions(material);
```

Essential boundary conditions on the left edge:

```
PointSegment leftSide(Point(0,0), Point(0,44));
SegmentConstraint left(leftSide, mesh.getPoints(), new Constant(0));
```

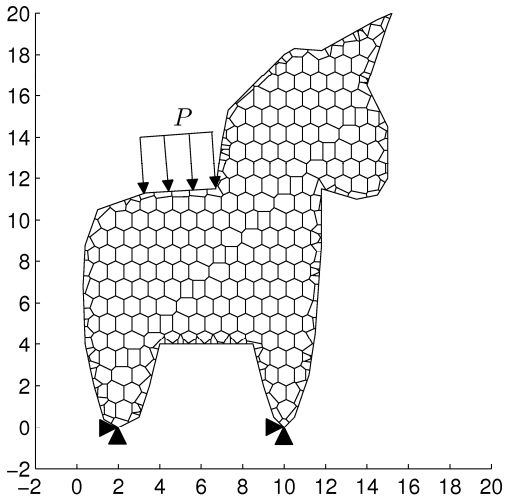
Natural boundary condition on the right edge:

```
PointSegment rightSide(Point(48,44), Point(48,64));
SegmentConstraint right(rightSide, mesh.getPoints(), new Constant(6.25));
```

Add boundary conditions to the model:

```
conditions->addEssentialConstraint(left, mesh.getPoints(),
elasticty_constraints::Direction::Total);
conditions->addNaturalConstraint(right, mesh.getPoints(),
elasticty_constraints::Direction::Vertical);
```

Example: Unicorn



Elastic Material

```
Material* material = new MaterialPlaneStrain(1e4, 0.25); // Also available: MaterialPlaneStress
LinearElasticityConditions* conditions = new LinearElasticityConditions(material);
```

Essential boundary conditions at Unicorn's feet:

```
Point leftFoot(2,0);
PointConstraint left(leftFoot, new Constant(0));
Point rightFoot(10,0);
PointConstraint right(rightFoot, new Constant(0));
```

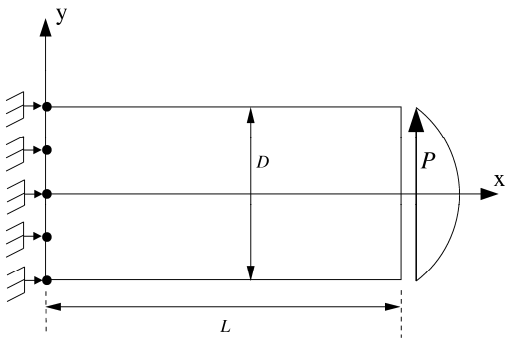
Natural boundary condition on Unicorn's back:

```
PointSegment backSegment(Point(6.7,11.5), Point(3.3,11.3));
SegmentConstraint back (backSegment, mesh.getPoints(), new Constant(-200));
```

Add boundary conditions to the model:

```
conditions->addEssentialConstraint(left, elasticity_constraints::Direction::Total);
conditions->addEssentialConstraint(right, elasticity_constraints::Direction::Total);
conditions->addNaturalConstraint(back, mesh.getPoints(),
elasticty_constraints::Direction::Total);
```

Example: Cantilever beam subjected to a parabolic end load



User defined functions:

```
double tangencial(double x, double y){
    double P = -1000; double D =  4;
    double I = std::pow(D,3)/12; double value = std::pow(D,2)/4-std::pow(y,2);
    return P/(2*I)*value;
}
double uX(double x, double y){
    double P = -1000; double Ebar = 1e7/(1 - std::pow(0.3,2));
    double vBar = 0.3/(1 - 0.3); double D = 4;
    double L = 8; double I = std::pow(D,3)/12;
    return -P*y/(6*Ebar*I)*((6*L - 3*x)*x + (2+vBar)*std::pow(y,2) -
            3*std::pow(D,2)/2*(1+vBar));
}
double uY(double x, double y){
    double P = -1000; double Ebar = 1e7/(1 - std::pow(0.3,2));
    double vBar = 0.3/(1 - 0.3); double D = 4;
    double L = 8; double I = std::pow(D,3)/12;
    return P/(6*Ebar*I)*(3*vBar*std::pow(y,2)*(L-x) + (3*L-x)*std::pow(x,2));
}
```

Elastic Material

```
Material* material = new MaterialPlaneStrain(1e7, 0.3); // Also available: MaterialPlaneStress
LinearElasticityConditions* conditions = new LinearElasticityConditions(material);
```

Essential boundary conditions on the left edge:

```
Function* uXConstraint = new Function(uX);
Function* uYConstraint = new Function(uY);
PointSegment leftSide(Point(0,-2), Point(0,2));
SegmentConstraint const1 (leftSide, mesh.getPoints(), uXConstraint);
SegmentConstraint const2 (leftSide, mesh.getPoints(), uYConstraint);
```

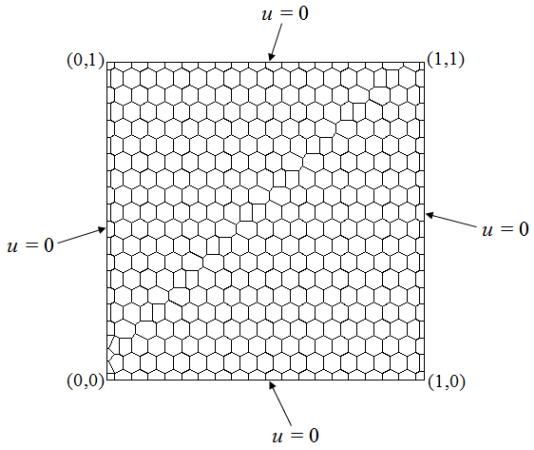
Natural boundary condition on the right edge:

```
Function* tangencialLoad = new Function(tangencial);
PointSegment rightSide(Point(8,-2), Point(8,2));
SegmentConstraint const3 (rightSide, mesh.getPoints(), tangencialLoad);
```

Add boundary conditions to the model:

```
conditions->addEssentialConstraint(const1, mesh.getPoints(),
elasticty_constraints::Direction::Horizontal);
conditions->addEssentialConstraint(const2, mesh.getPoints(),
elasticty_constraints::Direction::Vertical);
conditions->addNaturalConstraint(const3, mesh.getPoints(),
elasticty_constraints::Direction::Vertical);
```

Example: Poisson problem with a non constant source term (f=32y(1-y)+32x(1-x))



User defined functions:

```
double sourceTerm(double x, double y){
    return (32*y*(1-y) + 32*x*(1-x));
}
std::vector<double> exactScalarField(double x, double y){
    return {16*x*y*(1-x)*(1-y)};
}
std::vector<double> exactGradScalarField(double x, double y){
    return {16*y*(1-y)*(1-2*x),16*x*(1-x)*(1-2*y)};
}
```

Body/Source term

```
BodyForce* f = new BodyForce(sourceTerm);
PoissonConditions* conditions = new PoissonConditions(f);
```

Essential boundary conditions on the left edge:

```
PointSegment leftSide(Point(0,0), Point(0,1));
SegmentConstraint left (leftSide, mesh.getPoints(), new Constant(0)); // u=0;
```

Essential boundary conditions on the bottom edge:

```
PointSegment downSide(Point(0,0), Point(1,0));
SegmentConstraint down (downSide, mesh.getPoints(), new Constant(0)); // u=0;
```

Essential boundary conditions on the right edge:

```
PointSegment rightSide(Point(1,0), Point(1, 1));
SegmentConstraint right (rightSide, mesh.getPoints(), new Constant(0)); // u=0;
```

Essential boundary conditions on the top edge:

```
PointSegment topSide(Point(0, 1), Point(1, 1));
SegmentConstraint top (topSide, mesh.getPoints(), new Constant(0)); // u=0;
```

Add boundary conditions to the model:

```
conditions->addEssentialConstraint(left, mesh.getPoints());
conditions->addEssentialConstraint(down, mesh.getPoints());
conditions->addEssentialConstraint(right, mesh.getPoints());
conditions->addEssentialConstraint(top, mesh.getPoints());
```

10 Setting precision for printing on output files

In order to set the decimal precision for the floating-point values that are written to output files, one of the following instructions can be added to the lines of code in the main C++ setup file:

For predefined 6 decimals use:

```
VeamyConfig::instance()->setPrecision(Precision::precision::small);
```

For predefined 10 decimals use:

```
VeamyConfig::instance()->setPrecision(Precision::precision::mid);
```

For predefined 16 decimals use:

```
VeamyConfig::instance()->setPrecision(Precision::precision::large);
```

There is also a way to directly set the number of decimals. For instance, to set 12 decimals use:

```
VeamyConfig::instance()->setPrecision(12)
```

- If these instructions are omitted, the default number of decimals used to write the output files is 6.
- The example files that are located in the “test” folder of Veamy’s root directory use the foregoing instructions for setting the precision. See these example files for more details.

11 Veamy’s website

Check Veamy’s website for newer versions:

http://camlab.cl/research/software/veamy/

--- THE END ---