

# **Veamy**

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

## **Veamy Primer**

Version 2.1

Rev. 0  
September, 2018

## Copyright and License

Veamy, Copyright © 2017-2018  
by Catalina Álvarez, Nancy Hitschfeld-Kahler, Alejandro Ortiz-Bernardin  
<http://camlab.cl/research/software/veamy/>

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

---



Your use or distribution of Veamy or any derivative code implies that you agree to this License.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <<http://www.gnu.org/licenses/>>.

## **TABLE OF CONTENTS**

1	New and updated features summary .....	3
2	Features of Veamy .....	3
3	Source code .....	4
4	Up and running with Veamy .....	4
5	Using a PolyMesher mesh and boundary conditions in Veamy .....	9
6	Using a generic mesh file .....	12
7	Additional examples .....	13
7.1	Perforated Cook's membrane .....	13
7.2	A toy example .....	14
8	Geometry definition and mesh generation .....	16
9	Problem conditions: material definition, body/source terms, essential and natural boundary conditions .....	19
10	Setting precision for printing to output files .....	23
11	Veamy's website .....	23
	APPENDIX A: General structure of the main setup file .....	24
A.1	Setup file .....	24
A.2	Post processing .....	26

## 1 New and updated features summary

From Veamy 2.0 to Veamy 2.1:

- Add several test files for testing Feamy, the FEM module of Veamy.
- Fix some bugs.
- Update Veamy Primer: more details are added to sections devoted to using external mesh files (PolyMesher mesh and generic mesh files); Appendix A is added to explain the general structure of the main C++ setup file.

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 built-in 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/”



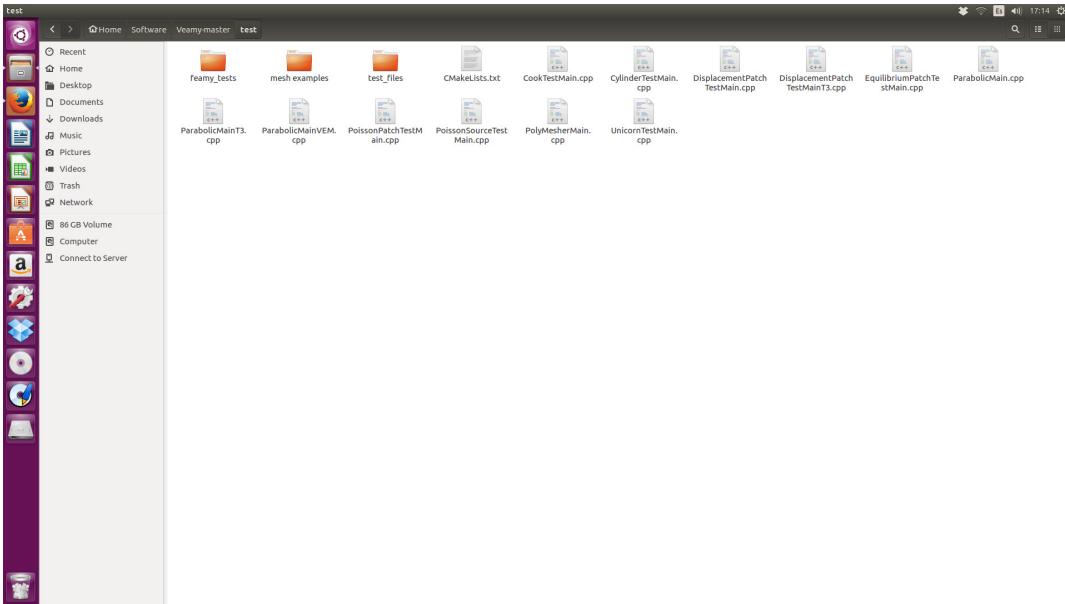
Fig. 1: Veamy source code.

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 \[cs.MS\]](https://arxiv.org/abs/1708.03438)

The geometry and boundary conditions for this problem along with a detailed explanation of the setup file is provided in Appendix A of this tutorial manual. Readers that are interested in learning more about the general structure of the setup file are referred to Appendix A. In the remainder of this section, 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).



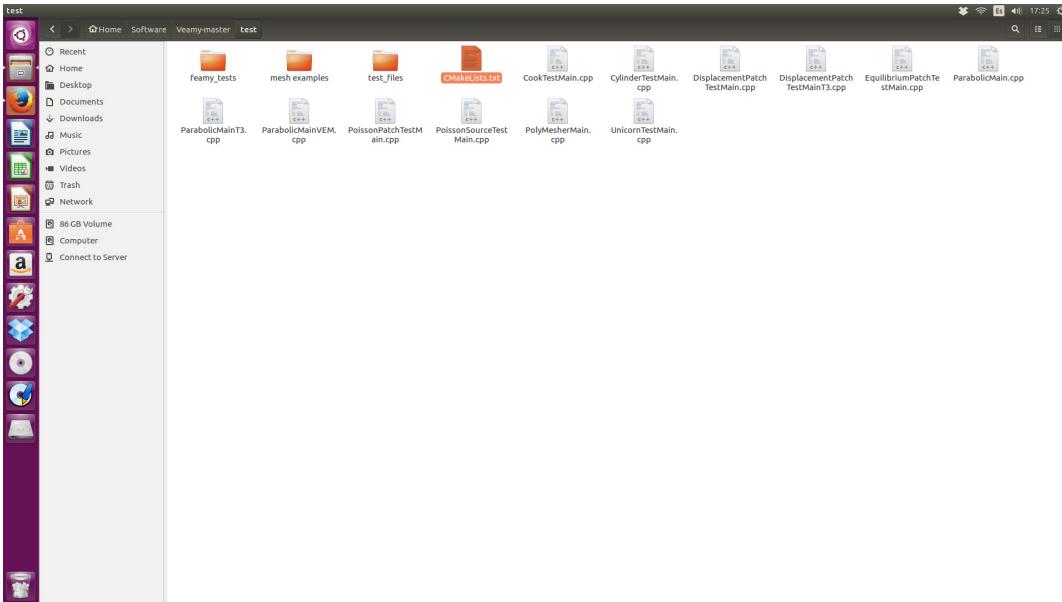
**Fig. 2:** Veamy’s test folder. The main C++ setup file implementing a problem of interest must be placed in this folder. Several main setup C++ files are shown. In this part of the primer, the C++ file “ParabolicMain.cpp” will be used.

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.

```
ParabolicMain.cpp (~Software/Veamy-master/test) - gedit
Open ▾
double L = 4;
double vBar = 1.0/(1 - std::pow(0.3, 2));
return -P*(vBar*I)*(x*1 - 3*x)*x + (2+vBar)*std::pow(y, 2) - 3*std::pow(D, 2)/(1+vBar));
}
double uv(double x, double y){
double P = -1000;
double vBar = 1.0/(1 - std::pow(0.3, 2));
double D = 4;
double I = 1;
double L = std::pow(D, 3)/12;
return P/(vBar*I)*(3*vBar*std::pow(y, 2)*(L-x) + (3*L-x)*std::pow(x, 2));
}
int main(){
//precision for plotting to output files:
//OPTION 1: in "VeamyConfig::instance()>setPrecision(Precision::precision::mid"
//use "small" for 6 digits; "mid" for 10 digits; "large" for 16 digits.
//OPTION 2: get the desired precision, for instance, as:
//VeamyConfig::instance()>getPrecision("large");
//OPTION 3: omit an instruction "VeamyConfig::instance()>setPrecision(...)".
//From this file, in this case, the default precision, which is 6 digits, will be used.
//VeamyConfig::instance()>getPrecision("mid");
//DEFINING PATH FOR THE OUTPUT FILES:
//if the path for the output files is not given, they are written to /home directory by default.
//otherwise, include the path. For instance, for /home/user/documents/veamy/output.txt , the path
//must be "documents/veamy/output.txt".
//output path can be static because it is already in your system or because it is created
//by Veamy's configuration files. For instance, Veamy creates the folder "test" inside "build", so
//one can save the output files to "/build/test" folder, but not to "/build/test/mycustom_folder",
//because the latter does not exist in the system. Veamy's configuration files.
std::string meshFileName = "parabolic_beam_mesh.txt";
std::string dispFileName = "parabolic_beam_displacements.txt";
std::cout << "**** Starting Veamy ****" << std::endl;
std::cout << "---- Test: cantilever beam subjected to a parabolic end load ----" << std::endl;
std::cout << "-----" << std::endl;
std::cout << "----- Defining the domain ... " ;
std::vector<Point> rectangle4x8_points = {Point(0, -2), Point(0, -2), Point(0, 2), Point(0, 2)};
Region rectangle4x8(rectangle4x8_points);
std::cout << "done" << std::endl;
std::cout << "----- Generating polygonal mesh ... ";
rectangle4x8.generatePoints(PolylineGenerator(functions::constantAlternating(), functions::constant()), 24, 12);
std::cout << "----- Generating Voronoi mesh ... ";
TriaLagrangeVoronoiGenerator meshGenerator(seeds, rectangle4x8);
Mesh<Polygons> mesh = meshGenerator.getMesh();
std::cout << "----- Mesh generated ... ";
mesh.printToFile(meshFileName);
std::cout << "done" << std::endl;
std::cout << "----- Defining linear elastic material ... ";
Material material = new MaterialPlaneStrain(1e7, 0.3);
```

**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.



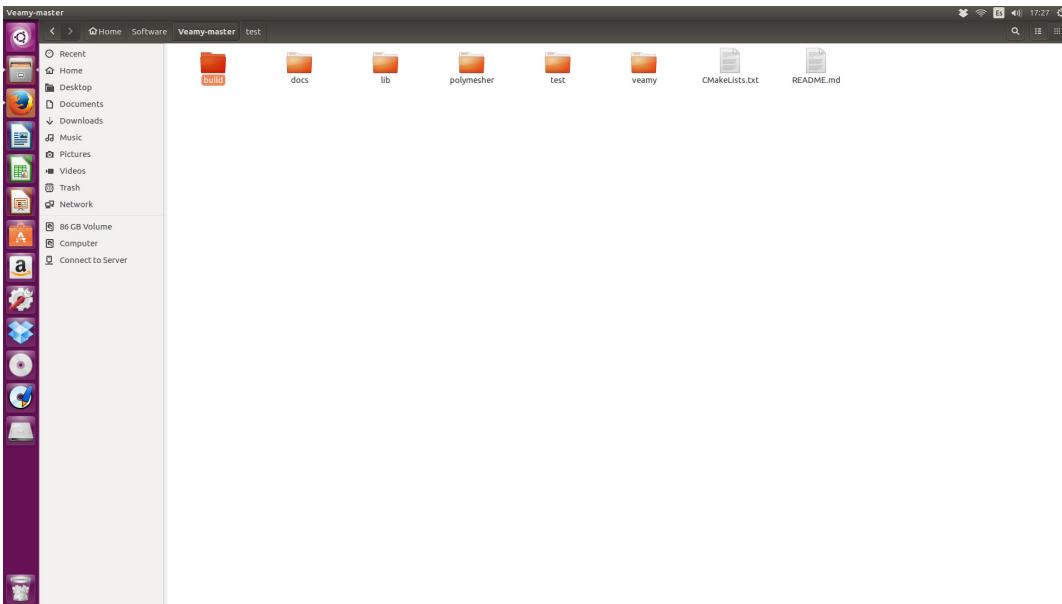
**Fig. 4:** CMakeLists.txt is located in test folder and controls which main C++ setup file is processed in Veamy.

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.

```
CMakeLists.txt (~Software/Veamy-master/test)- gedit
Open ▾  Save
set(CMAKE_CXX_FLAGS "-fPIC -std=gnu++11")
set(SOURCE_FILES ParabolicMain.cpp)
add_executable(test ${SOURCE_FILES})
target_link_libraries(test libutilities libdynol libveamy)
```

**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:

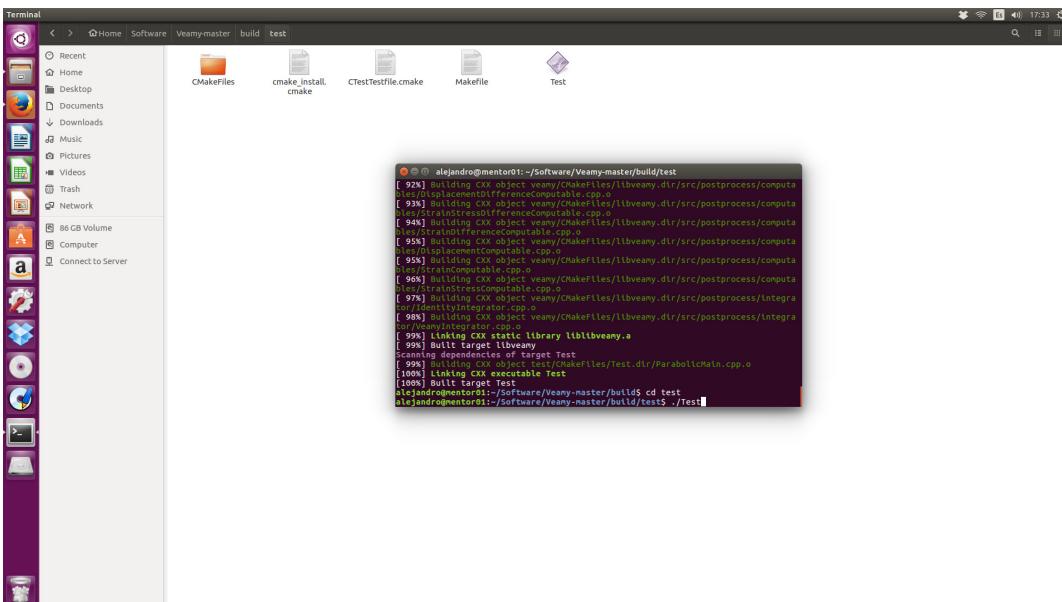
```
cmake ..
```

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.

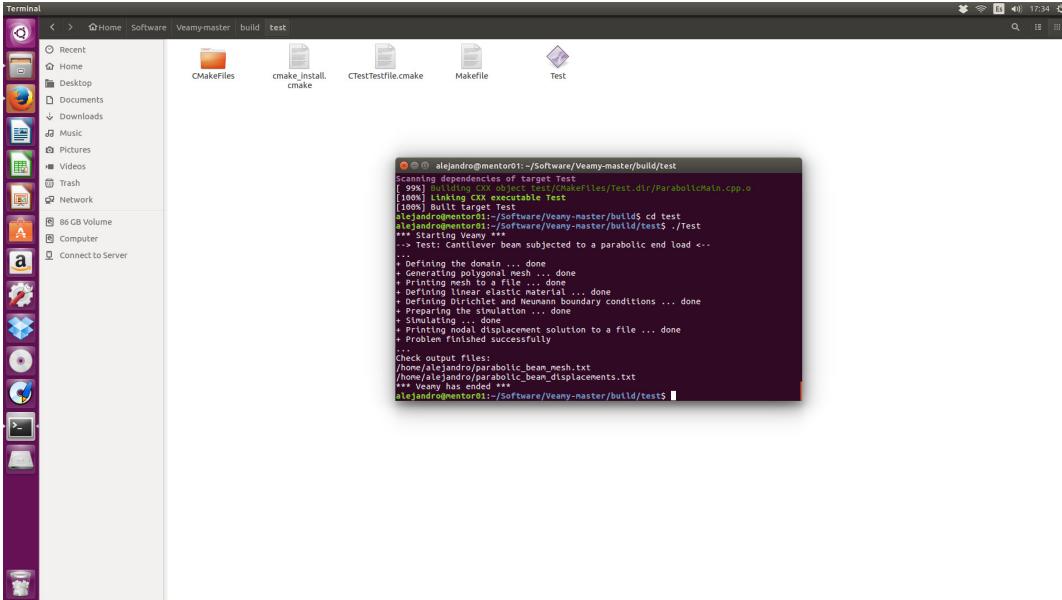
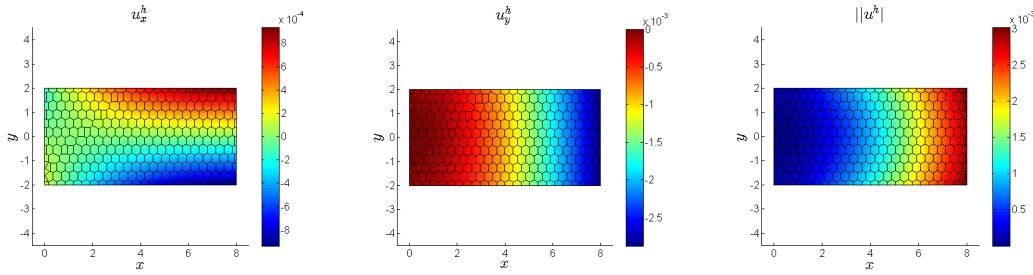


Fig. 8: Veamy prints out some messages while running the simulation.

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`” to translate a mesh created in PolyMesher to a format that is readable by Veamy. 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 procedure in PolyMesher, copy the created 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 translated PolyMesher mesh and boundary conditions is located inside the folder “`Veamy_root_directory/test/test_files/`”.

NOTE: The boundary conditions in PolyMesher are indicated by numbers: 0 (free) and 1 (fixed). See the translated example file “`polymesher2veamy.txt`” that is located inside the folder “`Veamy_root_directory/test/test_files/`”.

The implementation of the main C++ setup file that uses the translated PolyMesher mesh and boundary condition is provided as the file named “`PolyMesherMain.cpp`” (see Fig. 13). This setup file as usual is inside the folder “`Veamy_root_directory/test/`”. 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.

NOTE: It is also possible to use a PolyMesher mesh without boundary conditions in Veamy. This provides a means to apply more general boundary conditions. We simply create the mesh with the boundary conditions (we will delete the boundary conditions later, so it is not important how they are defined) as stated above. Then, we open the created file “polymesher2veamy.txt” that contains the translated mesh and delete the boundary conditions from it. We save the file and proceed as instructed in Section 6 of this manual.



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: %d Err: %.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_ClpsEdgs(Node,Element,0.1); %Remove small edges
[Node,Element] = PolyMshR_RsqNds(Node,Element); %Reorder 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.

```

PolyMesherMain.cpp (-/Software/Veamy-master/test) - gedit
Open ▾  File  Save
#include <veamy/physics/materials/MaterialPlaneStrain.h>
#include <veamy/physics/materials/MaterialPlaneStress.h>
#include <veamy/physics/conditions/LinearElasticityConditions.h>
#include <veamy/problems/VeamyLinearElasticityDiscretization.h>

int main()
{
    // Set precision for plotting to output files:
    // OPTION 1: In "VeamyConfig::Instance()> setPrecision(Precision::precision::mld)"
    // use "small" for 6 digits; "md" for 10 digits; "large" for 16 digits.
    // OPTION 2: set the desired precision, for instance, as follows:
    // VeamyConfig::Instance().setPrecision(12); Change "12" by the desired precision.
    // OPTION 3: Omit any instruction "VeamyConfig::Instance()> setPrecision(....)"
    // from this file. In this case, the default precision, which is 6 digits, will be used.
    VeamyConfig::Instance().setPrecision(Precision::precision::mld);

    // DEFINING PATH FOR THE OUTPUT FILES:
    // If you want to save the output files, they are written to "/home" directory by default.
    // Otherwise, include the path. For instance, for "/home/user/Documents/Veamy/output.txt", the path
    // must be "Documents/Veamy/output.txt"
    // Note: if you use a relative path, because it is already in your system or because it is created
    // by Veamy's configuration files. For instance, Veamy creates the folder "/test" inside "/build", so
    // one can save the output files to "/build/test/" folder, but not to "/build/test/mycustom_folder",
    // because the folder "/mycustom_folder" does not exist in Veamy's configuration files.
    std::string meshFileName = "polyMesher_test.mesh";
    std::string dispFileName = "polyMesher_test_displacements.txt";

    // File that contains the PolyMesher mesh and boundary conditions. like Matlab Function
    // PolyMesher2Veamy.m to generate this file.
    std::string polyMesherMeshFileName = "Software/Veamy-master/test/test_files/polyMesher2veamy.txt";

    std::cout << "*** Starting Veamy ***" << std::endl;
    std::cout << " --- Test: Using a PolyMesher mesh and boundary conditions ---" << std::endl;
    std::cout << " ---" << std::endl;
    std::cout << "+ Defining linear elastic material ... ";
    Material1D* material = new MaterialPlaneStrain(1.0, 0.3);
    LinearElasticityConditions* conditions = new LinearElasticityConditions(material);
    std::cout << " ---" << std::endl;

    std::cout << "+ Preparing the simulation from a PolyMesher mesh and boundary conditions ... ";
    VeamyLinearElasticityDiscretization* problem = new VeamyLinearElasticityDiscretization(conditions);
    Veamy v(problem);
    Mesh<Polygons> mesh = v.IntPtrProbleFromfile(polyMesherMeshFileName);
    std::cout << "done" << std::endl;

    std::cout << "+ Printing mesh to a file ... ";
    mesh.printToFile(meshFileName);
    std::cout << "done" << std::endl;
    std::cout << "+ Simulating ... ";
    Eigen::Vector3d x = v.simulate(mesh);
    std::cout << "done" << std::endl;
    std::cout << "+ Printing nodal displacement solution to a file ... ";
    v.writeDisplacements(dispFileName, x);
}

```

**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 ..
```

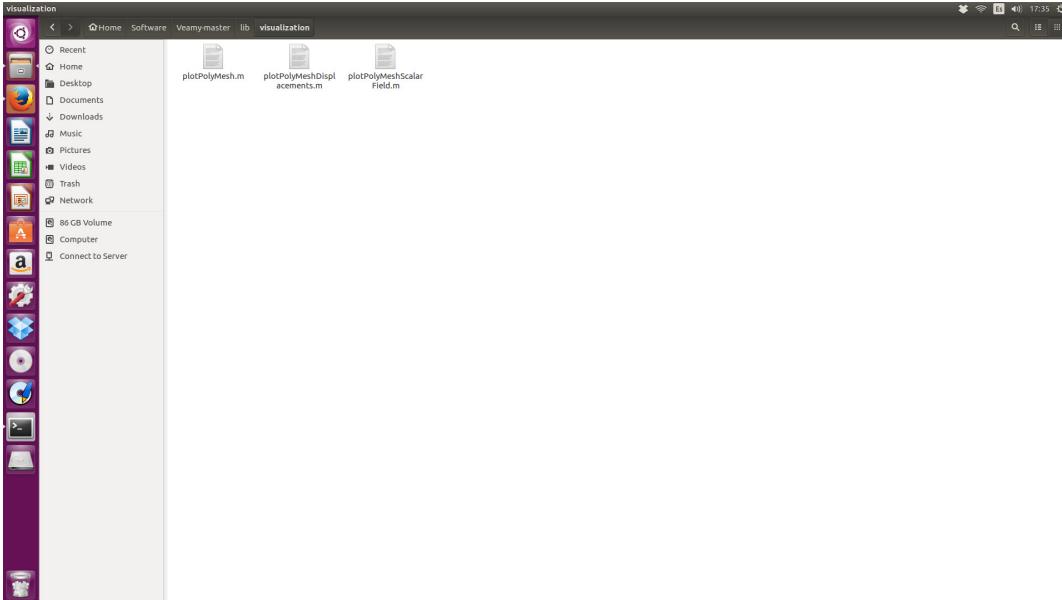
Then, to compile the program, on a terminal type and execute:

```
make
```

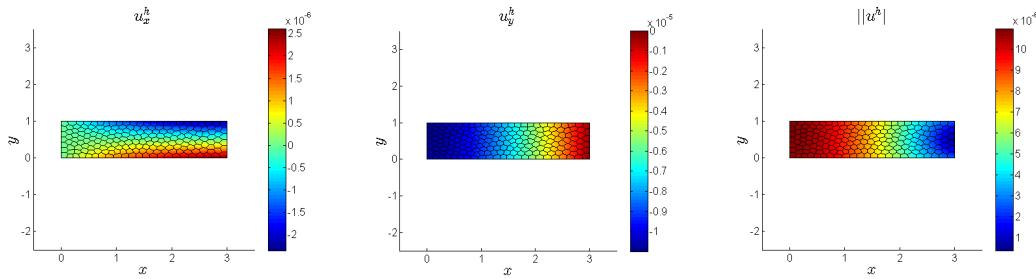
If this procedure has been done several times before, many of the libraries are likely to be already compiled, so the compilation procedure is quite short in comparison with the first time compilation. The executable of the test problem is stored in the “build/test/” folder and is called “Test”. Go inside “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 reading a PolyMesher mesh. The only difference is that boundary conditions are not provided in the mesh file. That is, the mesh file contains only the mesh information. To read this mesh file, we use the function “`createFromFile`”. The boundary conditions must be defined in Veamy similarly as done, for instance, in the cantilever beam problem that is implemented in the main C++ setup file “`ParabolicMain.cpp`” (see Appendix A, where this setup file is explained in detail). An example of the use of a generic mesh file is provided in the main C++ setup file “`EquilibriumPatchTestMain.cpp`”, where the generic mesh, “`Veamy_root_directory/test/test_files/equilibriumTest_mesh.txt`”, is read by the function “`createFromFile`”:

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

As you can confirm by exploring the generic mesh file “equilibriumTest\_mesh.txt”, it contains the nodal coordinates of the mesh and the element connectivity in the following format:

```
First line: number_of_nodes_in_the_mesh
Following lines: x-nodal-coordinates y-nodal-coordinates
One line: number_of_elements_in_the_mesh
Following lines: number_of_nodes_per_element(N) node1 node2 ... nodeN
```

## 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.

```
CookTestMain.cpp (~Software/Veamy-master/test)- gedit
Open ▾ Save
#include <vector>
#include <deDyno1/models/basic/Point.h>
#include <deDyno1/models/Region.h>
#include <deDyno1/models/hole/CircularHole.h>
#include <deDyno1/models/generator/functions/functions.h>
#include <deDyno1/models/generator/triangleOrLorenzGenerator.h>
#include <veamy/veamy.h>
#include <veamy/models/conditions/values/constant.h>
#include <veamy/problems/problems.h>
#include <veamy/physics/materials/MaterialPlaneStrain.h>
#include <veamy/config/VeamyConfig.h>
#include <veamy/problems/problems.h>
#include <veamy/problems/VeamyLinearElasticityDiscretization.h>
int main(){
    // Set precision for plotting to output files
    // OPTION 1: Use <VeamyConfig::Instance()>->setPrecision(Precision::precision::mid)
    //           for 6 digits, "large" for 10 digits.
    // OPTION 2: set the desired precision, for instance, as:
    // VeamyConfig::Instance()->setPrecision(12) for 12 digits, change "12" by the desired precision.
    // VeamyConfig::Instance()->setPrecision(...,-...,-...)
    // from this file. In this case, the default precision, which is 6 digits, will be used.
    VeamyConfig::Instance()->setPrecision(Precision::precision::mid);

    // DEFINING PATH FOR THE OUTPUT FILES:
    // If the path for the output files is not given, they are written to /home directory by default.
    // To change the path, include <veamy/veamy.h>, for instance, to </home/user/documents/VeamyOutput.txt>, the path
    // must be <documents/VeamyOutput.txt>
    // CAUTION: the path must exists either because it is already in your system or because it is created
    // by Veamy. If the path does not exist, it creates the folder "test" inside "build", so
    // one can save the output files to "/build/test/" folder, but not to "/build/test/mycustom_folder",
    // since "mycustom_folder" won't be created by Veamy's configuration files.

    std::string dispFileName = "cook_membrane_displacements.txt";
    std::string geoFileName = "cook_membrane_geometry.txt";

    std::cout << "**** Starting Veamy ****" << std::endl;
    std::cout << "**** Test: Cook's membrane ****" << std::endl;
    std::cout << "****" << std::endl;

    std::vector<Point> Tbeam;
    Region Tbeam(Tbeam_points);

    Hole hole1 = CircularHole(Point(8,30), 3);
    Hole hole2 = CircularHole(Point(24,40), 4);
    Hole hole3 = CircularHole(Point(40,50), 3);
    Tbeam.addhole(hole1);
    Tbeam.addhole(hole2);
    Tbeam.addhole(hole3);
    std::cout << "done" << std::endl;

    std::cout << "Printing geometry to a file ... ";
    Tbeam.printFile(geoFileName);
    std::cout << "done" << std::endl;

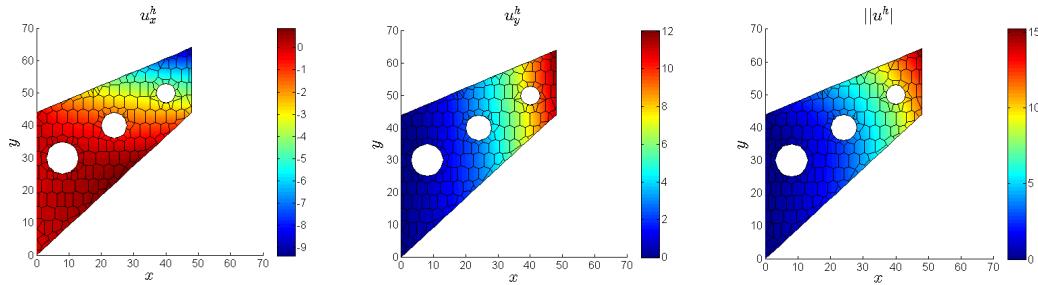
    std::cout << "Generating polygonal mesh ... ";
    Tbeam.generateSeedPoints(PointGenerator(functions::constantAlternating(), functions::constant()), 16, 16);
}
```

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.

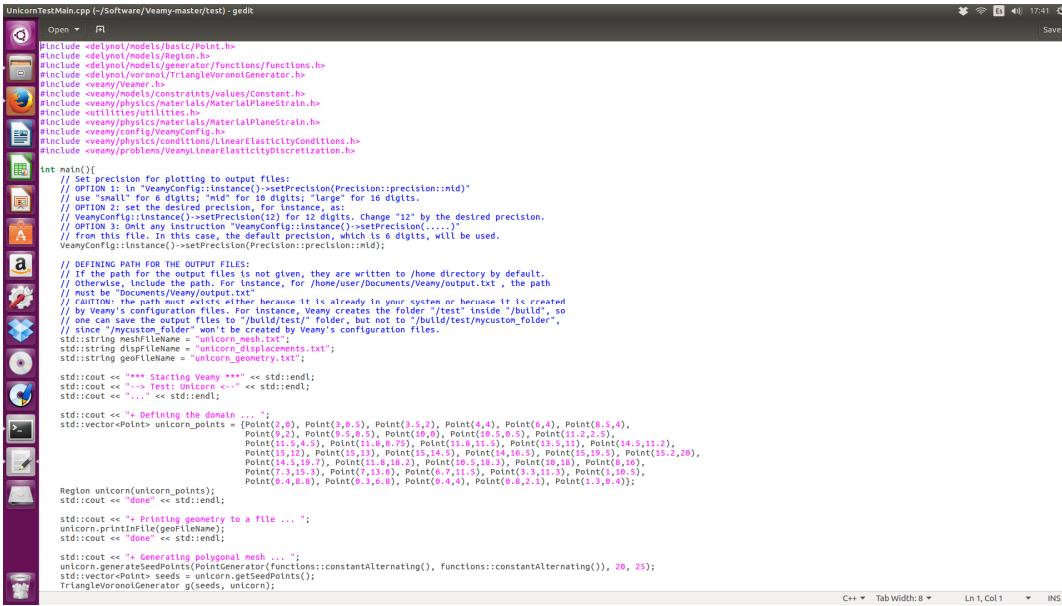
## 7.2 A toy example

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 \[cs.MS\]](https://arxiv.org/abs/1708.03438)

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.



```

#include <delynol/models/basic/Point.h>
#include <delynol/models/basic/Region.h>
#include <delynol/functions/functions.h>
#include <delynol/voronoi/TriangleVoronoiGenerator.h>
#include <veamy/veamer.h>
#include <veamy/constraints/values/constant.h>
#include <veamy/physics/materials/MaterialPlaneStrain.h>
#include <veamy/utilities/utilities.h>
#include <veamy/physics/materials/MaterialPlaneStrain.h>
#include <veamy/config/veamyConfig.h>
#include <veamy/physics/conditions/LinearElasticityConditions.h>
#include <veamy/problems/VeamyLinearElasticityDiscretization.h>

int main()
{
    // Precision for plotting to output files:
    // OPTION 1: In "VeamyConfig::Instance()>setPrecision(Precision::precision::mid"
    //           use "small" for 8 digits; "mid" for 10 digits; "large" for 16 digits.
    //           Note: "large" is the default.
    //           VeamyConfig::Instance()>setPrecision(12) for 12 digits. Change "12" by the desired precision.
    //           OPTION 3: Omit any instruction "VeamyConfig::Instance()>setPrecision(...)".
    //           VeamyConfig::Instance()>setPrecision(Precision::precision::mid), which is 8 digits, will be used.

    // DEFINING PATH FOR THE OUTPUT FILES:
    // If the path for the output files is not given, they are written to /home directory by default.
    // Otherwise, include the path. For instance, for /home/user/Documents/Veamy/output.txt , the path
    // /home/user/Documents/Veamy must exist either because it is already in your system or because it is created
    // by Veamy's configuration files. For instance, Veamy creates the folder "test" inside "/build", so
    // /home/user/Documents/Veamy/test will be created, but not to /build/test/veamycustom_folder,
    // since "veamycustom_folder" won't be created by Veamy's configuration files.
    std::string meshfileName = "unicorn_mesh.txt";
    std::string geofileName = "unicorn_geometry.txt";
    std::string pointsfile = "unicorn_points.txt";
    std::string configfile = "veamyConfig.txt";

    std::cout << " *** Starting Veamy *** " << std::endl;
    std::cout << " --- Test: Unicorn <-- " << std::endl;
    std::cout << "... " << std::endl;

    std::cout << "+ Defining the domain ... ";
    std::vector<Point> unicorn_points = {Point(2,0), Point(0,3), Point(-3,2), Point(0,0), Point(8,5), Point(8,5,4),
                                         Point(1,5,4), Point(1,5,0), Point(0,5,0), Point(10,5,0,1), Point(11,2,5,1),
                                         Point(11,5,4,5), Point(11,8,8,7), Point(11,8,11,5), Point(11,5,11), Point(14,5,11,2),
                                         Point(11,12), Point(11,13), Point(11,14,5), Point(14,10,5), Point(15,19,5), Point(15,2,20),
                                         Point(15,11,5), Point(15,12,5), Point(15,13,5), Point(15,14,5), Point(15,15,5), Point(15,16,5),
                                         Point(7,1,4,5,3), Point(7,1,3,0), Point(0,1,1,5), Point(3,1,1,5), Point(1,1,0,5),
                                         Point(0,4,0,8), Point(0,3,0,8), Point(0,4,4), Point(0,8,2,1), Point(1,3,0,4)};
    Region unicorn(unicorn_points);
    std::cout << "done" << std::endl;

    std::cout << "+ Printing geometry to a file ... ";
    unicorn.printInFile(geofileName);
    std::cout << "done" << std::endl;

    std::cout << "+ Generating polygonal mesh ... ";
    unicorn.generateSeedPoints(<PointGenerator(functions::constantAlternating()), functions::constantAlternating(), 20, 25>;
    std::vector<Point> seeds = unicorn.getSeedPoints();
    TriangleVoronoiGenerator g(seeds, unicorn);
}

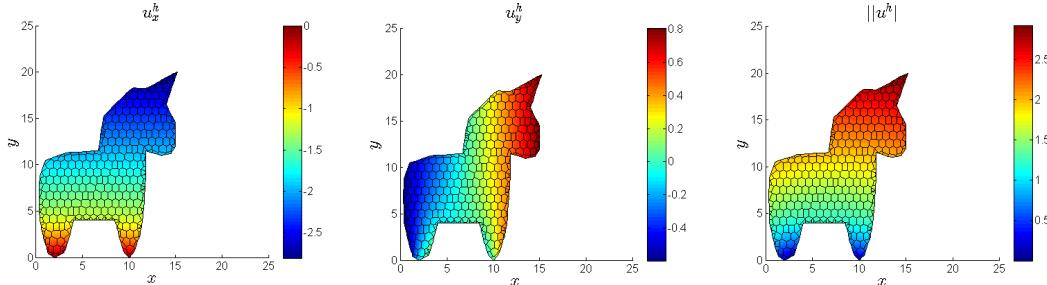
```

**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:

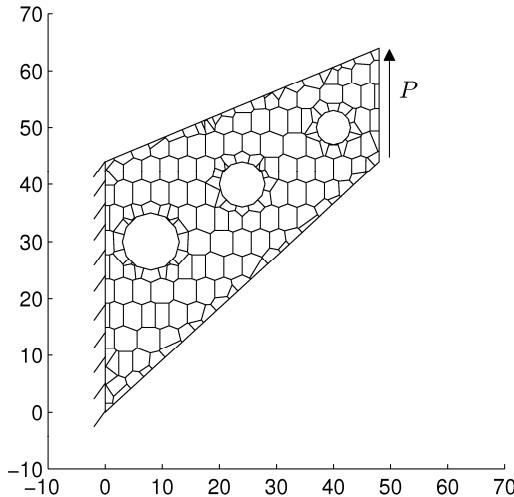
- Triangle - A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator.
- Clipper - an open source freeware library for clipping and offsetting lines and polygons.

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

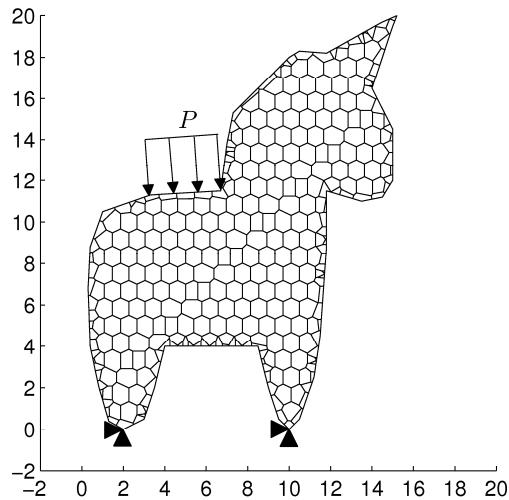
```
TBeam.addHole(hole1); TBeam.addHole(hole2); TBeam.addHole(hole3);
```

#### 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**

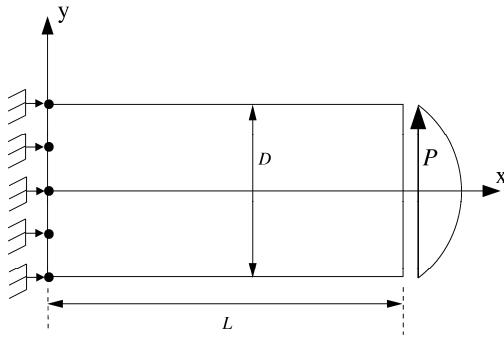
```
Region unicorn(unicorn_points);
```

**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();
```

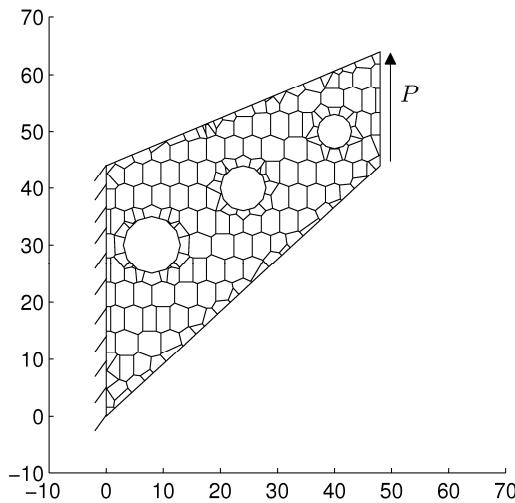
**Compute the polygonal mesh using the constrained Voronoi diagram**

```
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:**

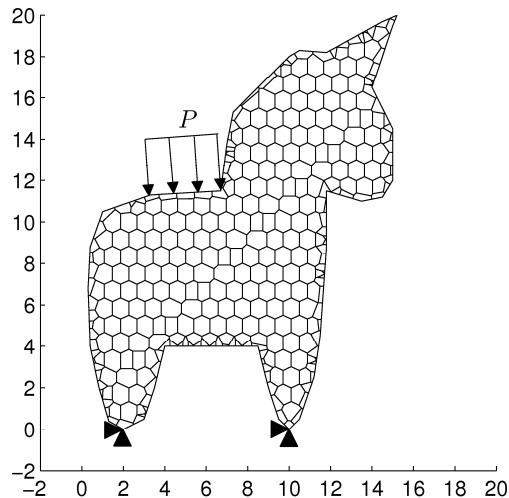
```
PointSegment leftSide(Point(0,0), Point(0,44));
SegmentConstraint left(leftSide, mesh.getPoints(), new Constant(0));
```

**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(), elasticity_constraints::Direction::Total);
conditions->addNaturalConstraint(right, mesh.getPoints(), elasticity_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:**

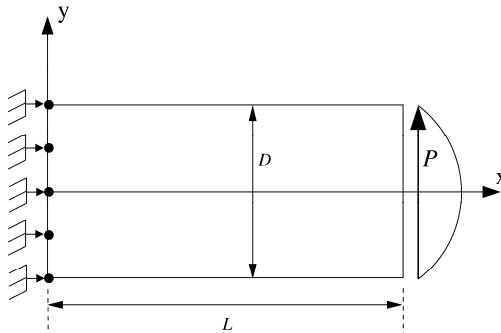
```
Point leftFoot(2,0);  
PointConstraint left(leftFoot, new Constant(0));  
Point rightFoot(10,0);  
PointConstraint right(rightFoot, new Constant(0));
```

**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(), elasticity_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);
```

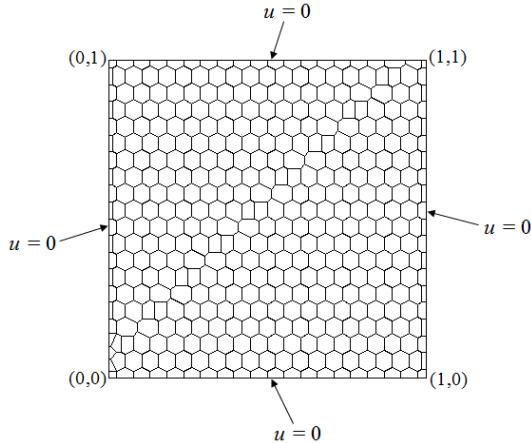
**Natural boundary condition on the right edge:**

```
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(), elasticity_constraints::Direction::Horizontal);
conditions->addEssentialConstraint(const2, mesh.getPoints(), elasticity_constraints::Direction::Vertical);
conditions->addNaturalConstraint(const3, mesh.getPoints(), elasticity_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 to 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/>

## APPENDIX A: General structure of the main setup file

In this appendix, the general structure of the setup file for solving a problem using Veamy is explained by means of a problem consisting in a cantilever beam that is subjected to a parabolic end load  $P$ . Fig. A.1 illustrates the geometry and boundary conditions. Plane strain state is assumed. The essential boundary conditions on the clamped edge are applied according to the analytical solution given by

$$u_x = -\frac{Py}{6E_Y I} \left( (6L - 3x)x + (2 + \bar{\nu})y^2 - \frac{3D^2}{2}(1 + \bar{\nu}) \right),$$

$$u_y = \frac{P}{6E_Y I} (3\bar{\nu}y^2(L - x) + (3L - x)x^2),$$

where  $E_Y = E_Y / (1 - \nu)$  with the Young's modulus set to  $E_Y = 1 \times 10$  psi, and  $\bar{\nu} = \nu / (1 - \nu)$  with the Poisson's ratio set to  $\nu = 0.3$ ;  $L = 8$  in. is the length of the beam,  $D = 4$  in. is the height of the beam, and  $I$  is the second-area moment of the beam section. The total load on the traction boundary is  $P = -1000$  lbf.

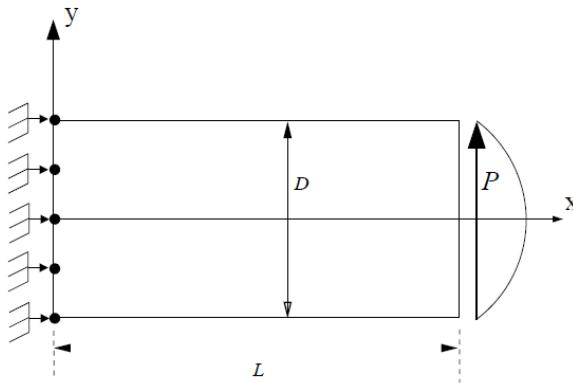


Fig. A.1: Model geometry and boundary conditions for the cantilever beam problem.

### A.1 Setup file

A main C++ file is written to setup the problem. As there are different aspects to consider, we divide the setup file in several blocks and explain each of them. Herein only the main parts of this setup file are described. The complete setup instructions for this problem are provided in the file “ParabolicMain.cpp” that is located in the folder “Veamy/test/.”

**Listing 1** shows the definition of the problem domain, the generation of base points for the Voronoi diagram, and the computation of the polygonal mesh.

```

1 std::vector<Point> rectangle_points = {Point(0, -2), Point(8, -2), Point(8, 2), Point(0, 2)};
2 Region rectangle(rectangle_points);
3 rectangle.generateSeedPoints(PointGenerator(functions::constantAlternating(), functions::constant()), 24,
12);
4 std::vector<Point> seeds = rectangle.getSeedPoints();
5 TriangleVoronoiGenerator meshGenerator (seeds, rectangle);
6 Mesh<Polygon> mesh = meshGenerator.getMesh();
```

**Listing 1:** Domain definition and mesh generation for the beam subjected to a parabolic end load.

We proceed to initialize all the structures needed to represent the conditions of the problem at hand. In first place, an object of the Material class is created and used to initialize an object of the class LinearElasticityConditions. This is shown in **Listing 2**.

```

1 Material* material = new MaterialPlaneStrain (1e7, 0.3);
2 LinearElasticityConditions* conditions = new LinearElasticityConditions(material);

```

**Listing 2:** Definition of the elastic material and initialization of the problem conditions.

We create a constraint that represents the essential boundary condition that is imposed on the left side of the beam, including the segment it affects, the value of the constraint and the direction (in the Cartesian coordinate system) in which the constraint is imposed. This implementation is shown in Listing 3.

```

1 double uX(double x, double y){
2     double P = -1000;
3     double Ebar = 1e7/(1 - std::pow(0.3,2));
4     double vBar = 0.3/(1 - 0.3);
5     double D = 4; double L = 8; double I = std::pow(D,3)/12;
6     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));
7 }
8
9 double uY(double x, double y){
10    double P = -1000;
11    double Ebar = 1e7/(1 - std::pow(0.3,2));
12    double vBar = 0.3/(1 - 0.3);
13    double D = 4; double L = 8; double I = std::pow(D,3)/12;
14    return P/(6*Ebar*I)*(3*vBar*std::pow(y,2)*(L-x) + (3*L-x)*std::pow(x,2));
15 }
16 Function* uXConstraint = new Function(uX);
17 Function* uYConstraint = new Function(uY);
18 PointSegment leftSide(Point(0,-2), Point(0,2));
19 SegmentConstraint const1 (leftSide, mesh.getPoints(), uXConstraint);
20 SegmentConstraint const2 (leftSide, mesh.getPoints(), uYConstraint);
21 conditions->addEssentialConstraint(const1, mesh.getPoints(), elasticity_constraints::Direction::Horizontal);
22 conditions->addEssentialConstraint(const2, mesh.getPoints(), elasticity_constraints::Direction::Vertical);

```

**Listing 3:** Definition of the essential boundary condition on the left side of the beam.

Listing 4 presents the implementation of the natural boundary condition (the parabolic load) that is applied on the right side of the beam. The parabolic load is constructed using a function called tangencial.

```

1 double tangencial(double x, double y){
2     double P = -1000; double D = 4;
3     double I = std::pow(D,3)/12;
4     double value = std::pow(D,2)/4-std::pow(y,2);
5     return P/(2*I)*value;
6 }
7 Function* tangencialLoad = new Function(tangencial);
8 PointSegment rightSide(Point(8,-2), Point(8,2));
9 SegmentConstraint const3 (rightSide, mesh.getPoints(), tangencialLoad);
10 conditions->addNaturalConstraint(const3, mesh.getPoints(), elasticity_constraints::Direction::Vertical);

```

**Listing 4:** Definition of the natural boundary condition on the right side of the beam.

The linear elastostatic problem is initialized with the problem conditions previously defined by creating an object of the class VeamyLinearElasticityDiscretization. And the latter along with the mesh is used to initiate a Veamer instance that represents the system. Finally, to obtain the nodal displacements solution the simulate method is invoked. These instructions are presented in Listing 5.

```

1 VeamyLinearElasticityDiscretization* problem = new VeamyLinearElasticityDiscretization(conditions);
2 Veamer v(problem);
3 v.initProblem(mesh);
4 Eigen::VectorXd displacements = v.simulate(mesh);

```

**Listing 5:** Initialization of the system that represents the beam subjected to a parabolic end load and start of the simulation.

The output of the simulate method is a column vector that contains the nodal displacements solution. To print the nodal displacements solution to an output file, the writeDisplacements method is called after the simulate method. This is shown in Listing 6. The resulting text file is named as the string stored in displacementsFileName. The text file contains the computed displacements in the following format: nodal index, x-displacement and y-displacement. An extract of the output file generated for the beam subjected to a parabolic end load is shown in Listing 7.

```
1 v.writeDisplacements(displacementsFileName, displacements);
```

**Listing 6:** Printing of nodal displacements solution to an output file.

```
1      0      9.38002e-005      -0.000100889
2      1      0.000137003      -0.000101589
3      2      9.30384e-005      -0.000115664
4 ...
```

**Listing 7:** Extract of the output file for the beam subjected to a parabolic end load.

The output file contains no information about the geometry of the problem. The geometry information is kept in the Mesh instance created at the beginning of the example. Mesh includes a method to print its geometrical data to a text file with a single line of code, as shown in Listing 8.

```
1 mesh.printInFile(meshFileName);
```

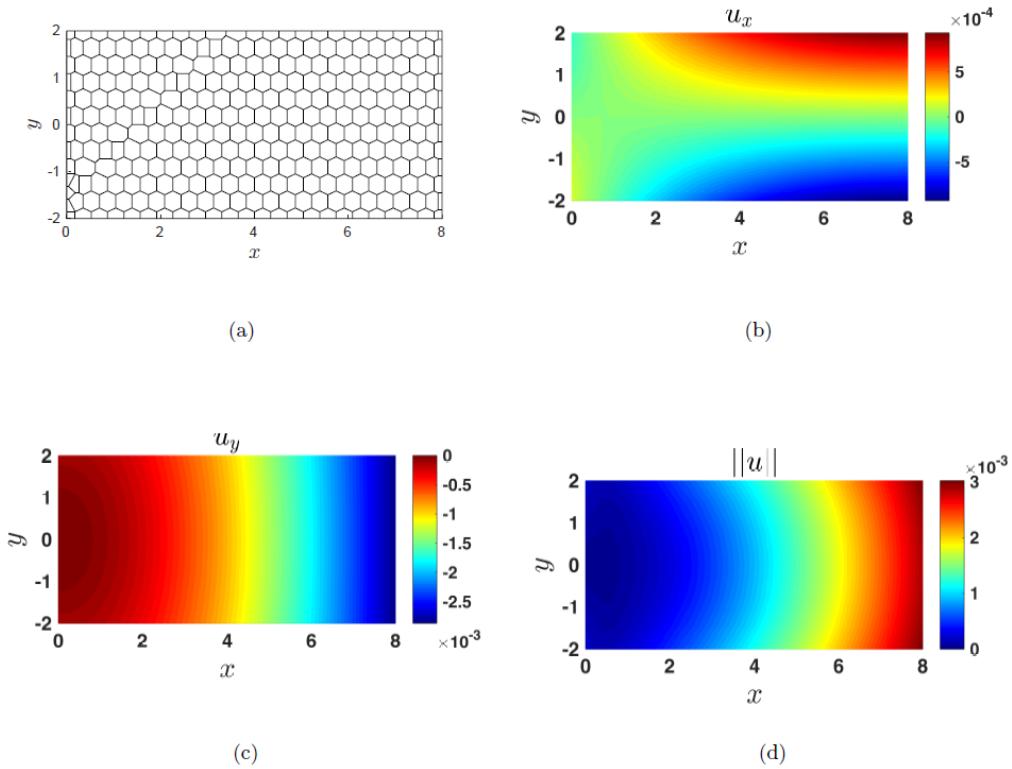
**Listing 8:** Printing of mesh data to a text file.

The text file containing the mesh information is named as the string stored in meshFileName and is arranged in the following format:

- First line: number of nodal points in the polygonal mesh.
- Following lines: x-coordinate y-coordinate for each nodal point in the mesh.
- One line: number of element edges in the polygonal mesh.
- Following lines: index-of-start-point index-of-end-point for each element edge in the polygonal mesh.
- One line: number of elements in the polygonal mesh.
- Following lines: number-of-element-nodes list-of-nodal-indexes centroid-x-coordinate centroid-y-coordinate for each polygon in the mesh.

## A.2 Post processing

Veamy does not provide a post processing interface. The user may opt for a post processing interface of their choice. Here we visualize the displacement results using a MATLAB function written for this purpose. This MATLAB function is provided in the folder “Veamy/lib/visualization/” as the file “plotPolyMeshDisplacements.m.” In addition, a file named “plotPolyMesh.m” that serves for plotting the mesh is provided in the same folder. Fig. A.2 presents the polygonal mesh used and the VEM solutions.



**Fig. A.2:** Solution for the cantilever beam subjected to a parabolic end load using Veamy. (a) Polygonal mesh, (b) VEM horizontal displacement, (c) VEM vertical displacement, (d) norm of the VEM displacement.

--- THE END ---