

Vega FEM Library (v1.1) User's Manual*

Jernej Barbič

Funshing Sin

Daniel Schroeder

September 10, 2012

1 Introduction

Vega is a computationally efficient and stable C/C++ library (about 50,000 lines of code) to timestep nonlinear three-dimensional deformable models. It is designed to model large deformations, including geometric and material nonlinearities, and can also efficiently simulate linear elasticity. Vega is a *middleware* physics library; it is not an end-product, or plug-and-play software. Its strength lies in its many C/C++ libraries which depend minimally on each other, and are in most cases independently reusable. Vega is open-source and free, and can be downloaded from <http://www.jernejbarbic.com/vega>. It is released under the 3-clause BSD license, which means that it can be used both in academic research and in commercial applications; see `LICENSE.txt`. This documentation aims to explain how to compile and use Vega. It also provides detail about the code structure to simplify component re-use and the addition of new features.

Vega implements several widely used methods for simulation of large deformations of 3D solid deformable objects, including co-rotational linear Finite Element Method (FEM) elasticity [MG04] (and also supports the exact tangent stiffness matrix [Bar12], similar to [CPSS10]), Saint-Venant Kirchhoff FEM deformable models (see [Bar07]), invertible FEM models [ITF04, TSIF05], and mass-spring systems. All models include support for multi-core computing. In addition to linear materials, Vega provides neo-Hookean and Mooney-Rivlin nonlinear material models. Arbitrary nonlinear materials can be added to Vega. For isotropic hyperelastic materials, this is as easy as defining an energy function and its first and second derivatives. Implemented integrators include implicit backward Euler [BW98], explicit central differences [Wri02], implicit Newmark [Wri02] and symplectic Euler. Other integrators can be added easily.

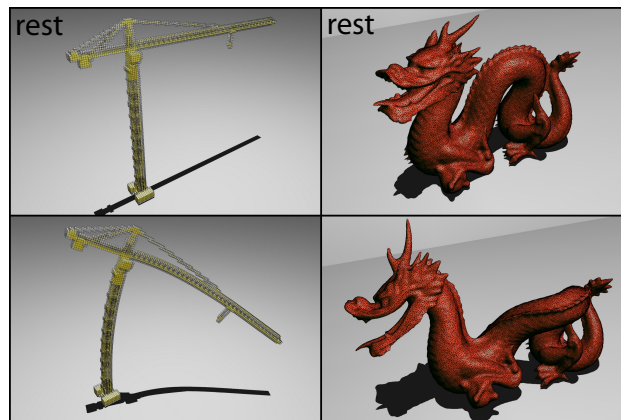


Figure 1: **Large FEM deformations:** The crane cubic (voxel) mesh is shown blended on top of the embedded triangle mesh, whereas the dragon uses the external surface of the tetrahedral mesh for rendering. Original triangle models are courtesy of Waldemar Barkowski, <http://www.artworx-media.de>, and Stanford Computer Graphics Laboratory, <http://graphics.stanford.edu/data/3Dscanrep/>, respectively.

*Authors of Vega are Jernej Barbič, Funshing Sin and Daniel Schroeder. This user's manual was written by Jernej Barbič and Daniel Schroeder. Acknowledgments: authors of the implemented papers, Doug L. James, Jovan Popović, United States National Science Foundation (CAREER-0430528, CAREER-53-4509-6600), James H. Zumberge Research and Innovation Fund at the University of Southern California, Link Foundation, Singapore-MIT GAMBIT Game Lab.

Vega is a library for 3D solid simulation¹. The input to Vega is a 3D volumetric mesh with material properties, as well as (time-varying) external forces to act on the 3D mesh at each timestep. Vega computes the displacements of mesh vertices at each timestep, as well as the internal elastic forces and their gradients (tangent stiffness matrix). Two mesh elements are supported: linear tetrahedra and axis-aligned cubes. Vega includes libraries for sparse matrix storage and arithmetics, volumetric mesh storage, code performance timing, and mesh rendering using OpenGL. For sparse linear system solving, Vega can use PARDISO [PAR], SPOOLES [SPO], or conjugate gradients. The included (Jacobi-preconditioned) conjugate gradient solver was written by Jernej Barbič by following the well-known reference [She94]. Vega also includes an elaborate example driver (executable) simulator to interactively test the implemented materials, timestepping methods, and other settings. Several example meshes and configuration files are included to demonstrate the various materials. We provide detailed experiments on the performance of Vega in [SSB12].

Finally, we find it useful to also state what Vega currently does not support. Vega currently supports 3D solid simulation on volumetric meshes; it does not support cloth and strands. It uses linearized tets and axis-aligned cubes for mesh elements; higher-order elements are not supported. Vega solves linear systems of equations using preconditioned conjugated gradients or direct sparse solvers. It does not incorporate solver acceleration strategies such as multigrid. It does not implement inverse kinematics, or space-time optimization. It does not simulate rigid bodies, articulated rigid bodies or fluids. It does not support mesh cutting or other run-time mesh modifications; although mesh material properties could be altered at run-time without much difficulty. While it does not incorporate collision detection or contact resolution, it is possible to use any external collision detection library, compute forces externally, and set them as external forces in Vega. A more elaborate contact solver could be written by using Vega’s routines to compute internal forces and stiffness matrices in arbitrary mesh configurations.

1.1 Compiling Vega

Vega is designed to be cross-platform. We have successfully compiled it on Linux, Windows and Mac OS X. We provide makefiles (**make** command) for Linux and Mac OS X.

The first step is to select either **Makefile-headers/Makefile-header.linux**, or **Makefile-headers/Makefile-header.osx**, depending on your OS. By default, linux is selected. The selection is made by creating a soft link **Makefile-headers/Makefile-header** that points to the selected Makefile. To switch to MAC OS X, use:

```
cd Makefile-header;ln -sf Makefile-header.osx Makefile-header.
```

Alternatively, you can switch using the provided scripts **Makefile-headers/selectLinux** or **Makefile-headers/selectMACOSX**.

To compile the entire set of libraries and the interactive simulator in one step, move to the **utilities/interactiveDeformableSimulator** folder and run **\$ make**. To compile an individual library **libraryName**, along with all the libraries it depends on, move to **libraries/libraryName** and run **\$ make**. The script **./build** combines **Makefile-header** selection and the compilation into one step.

To clean all the compiled files (for the interactive simulator and for all the libraries), move to **utilities/interactiveDeformableSimulator** and run **\$ make deepclean**. To only clean the simulator folder, run **\$ make clean**. Similarly, running **\$ make deepclean** in folder **libraries/libraryName** will clean the folder for **libraryName** as well as those of all the other **libraries** libraries it depends on, whereas running **\$ make clean** will only clean the **libraries/libraryName** folder.

Dependencies: Vega has no required external dependencies. It includes all the necessary components for deformable object simulation. It uses its own CG solver as the default solver for implicit integration. This solver can be substituted for external linear system sparse solvers such as the PARDISO or SPOOLES. Other solvers can be easily added as well. The provided demonstration driver uses OpenGL for visualization: it depends on the **GL**, **GLU**, **GLUT** OpenGL libraries. It also depends on the **GLUI** User Interface Library, which is used for the driver’s user interface (buttons, edit boxes, etc.). This external library (LGPL license) is included with the Vega distribution (in **libraries/glui**), unmodified as it was downloaded from <http://glui.sourceforge.net/>. The provided Makefiles will automatically compile **GLUI** to a shared library

¹extensions to shell (cloth) and strand models are possible

(to comply with the LGPL license terms), and link it to the driver. On some systems, for the build to work correctly, you may need to figure out how to satisfy the dependencies on OpenGL and/or GLUI. Default settings for the locations of all libraries are provided in the Makefile-header files for Linux and Mac OS X. If needed, custom locations can be set by modifying the `libraries/include/OpenGL-headers.h` header. For example, to set custom locations and names of the OpenGL libraries, modify the `LIBRARIES`, `STANDARD_LIBS`, `GLUI_DIR`, `GLUI_INCLUDE`, and `GLUI_LIB` variables in `Makefile-header`.

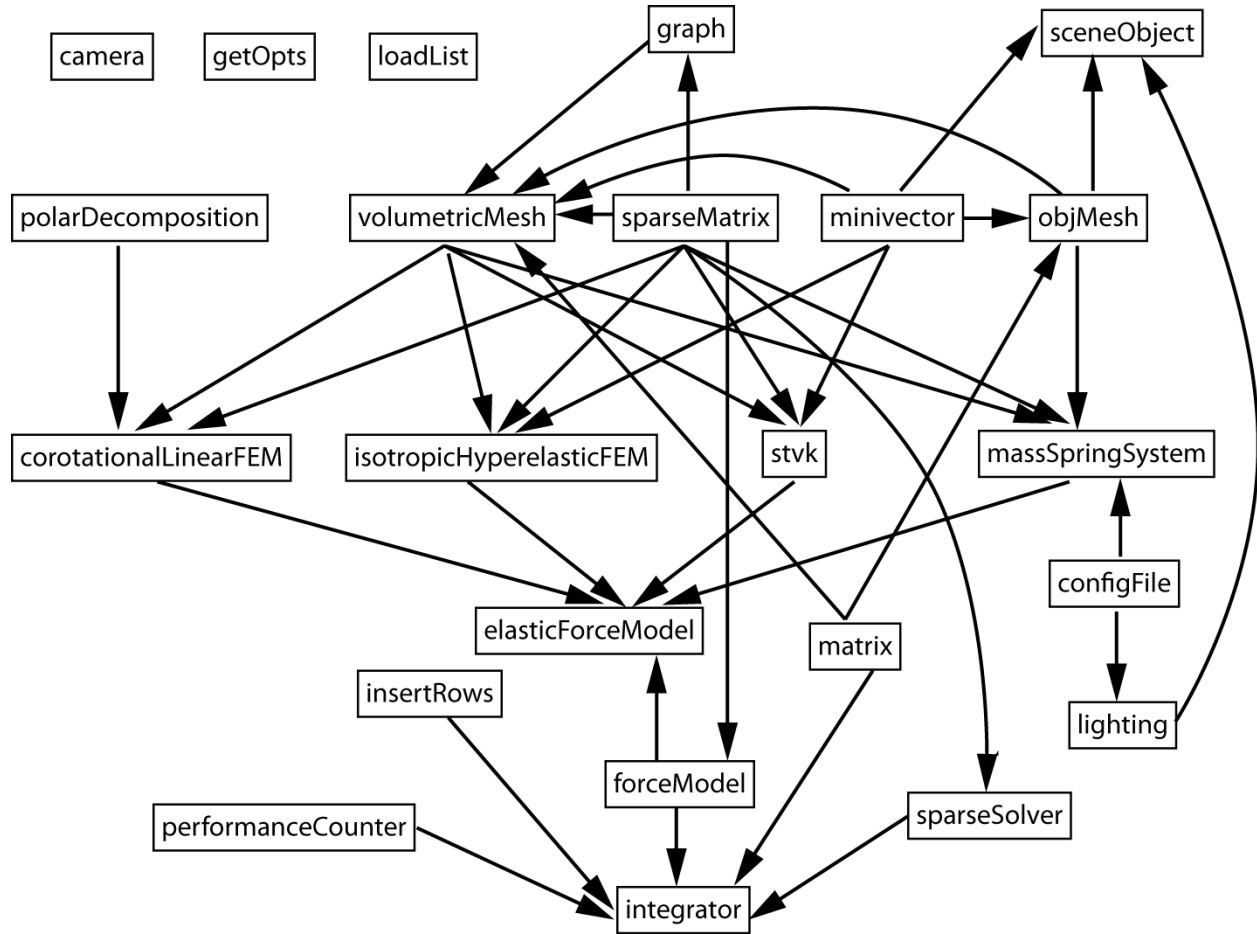


Figure 2: The dependencies of libraries in Vega.

Using PARDISO: Vega contains code to easily use the PARDISO solver from the Intel MKL library. To use PARDISO, you must first install it. To specify the names and locations of PARDISO libraries, modify the `PARDISO_DIR`, `PARDISO_INCLUDE`, and `PARDISO_LIB` variables in the `Makefile-header`. If needed, adjust the included MKL header files by modifying the `libraries/include/lapack-headers.h` header file. Next, notify Vega that PARDISO is available by defining (uncommenting) the macro `PARDISO_SOLVER_IS_AVAILABLE` in `libraries/sparseSolver/sparseSolverAvailability.h`. Finally, switch the `integrator` library to PARDISO by enabling (defining) the macro `PARDISO` in `libraries/integrator/integratorSolverSelection.h`. Then, recompile Vega and the driver.

Using SPOOLES: Vega can also use SPOOLES, a public domain sparse solver, freely available at <http://www.netlib.org/linalg/spooles/>. The library is used by the `SPOOLESolver` and `SPOOLESolverMT` classes in the `sparseSolver` library. To use SPOOLES, follow the same steps as with PARDISO. To set the location of the SPOOLES library and header files, modify the `SPOOLES_DIR`, `SPOOLES_INCLUDE`, and

SPOOLES_LIB variables in Makefile-header.

Note that the PARDISO and SPOOLES solvers are not necessary to compile our example driver. By default, Vega uses its preconditioned conjugate solver (PCG), available in `libraries/sparseSolver/CGSolver.h`. Finally, note that the make files do not monitor the `openGL-headers.h` or `lapack-headers.h` files, so modifications to these may require cleaning and remaking the project, as explained above.

Microsoft Visual Studio: The `pthread` library is required for the multi-threading functionality in Vega. This library ships with Linux and Mac OS X operating systems. For Windows, you can download a Windows port of `pthread` from <http://sources.redhat.com/pthreads-win32/>. In order to build the driver, one needs a GLUT implementation, for example, <http://user.xmission.com/~nate/glut.html>. The `stdlib.h` which ships with the recent versions of Visual Studio may have a conflict with some GLUT implementations. To fix this issue, right click on the project name in the Solution Explorer tab and select Properties→C/C++→Preprocessor→Preprocessor definitions. Then append `GLUT_BUILDING_LIB` to the existing definitions, separated by semicolons.

1.2 Driver Executable and Examples

Vega provides a complete simulation driver (executable), as well as several complete examples. Examples include all the necessary meshes and configuration files. To run the simulator on these configurations, run any of the `run_[modelname]` shell scripts in the `examples` folder. For example, try `examples/turtle/run_turtle`, `examples/asianDragon/run_asianDragon` or `examples/asianDragon/run_asianDragon_unconstrained`. You can read the `examples/guideToExamples.txt` for a brief overview of our examples. The simulator executable `interactiveDeformableSimulator` takes one command-line argument: a `.config` configuration file such as those in the `examples` folder. The `.config` file specifies such information as the mesh, the desired material and timestepping method to use, the timestep, damping, an optional secondary rendering mesh embedded in the deformable mesh, and so on. The simulator must be run in the folder containing the desired `.config` file, since this file contains relative paths to other files read by the executable. It is easiest to achieve this by using the provided `run*` shell scripts. Inside the simulator, some useful keys are 'E', 'e', and 'w', to toggle the displays of embedded rendering mesh, volumetric mesh, and volumetric mesh wireframe, respectively. Note that the main simulation window **must have focus** for the keys to work.

1.3 Short Tutorial on Using Vega

Vega is a middleware library and should be integrated with the rest of the user's C/C++ code. In this section, we give the typical steps to do so. Details on the specific methods can be found in subsequent sections.

The first step is to load a volumetric mesh from a `.veg` file (format is described in Section 1.4), which creates a `VolumetricMesh` object. It is easiest to do so using the provided `VolumetricMeshLoader` class, which automatically detects the type of elements in the mesh (tets or cubes).

```
#include "volumetricMeshLoader.h"
...

char inputFilename[96] = "myInputMeshFile.veg";
VolumetricMesh * volumetricMesh = VolumetricMeshLoader::load(inputFilename);
if (volumetricMesh == NULL)
    printf("Error: failed to load mesh.\n");
else
    printf("Success. Number of vertices: %d . Number of elements: %d .\n",
        volumetricMesh->getNumVertices(), volumetricMesh->getNumElements());
```

At this point, the mesh and its material properties have been parsed into memory, and we have a valid `VolumetricMesh` object. Next, we will initialize a specific 3D deformable model, allowing us to compute

internal forces and stiffness matrices for arbitrary deformed object configurations. Let us use the corotational linear FEM model. Because that model only supports tet meshes, we need to first check if the mesh is indeed a TetMesh.

```
#include "corotationalLinearFEM.h"
...

TetMesh * tetMesh;
if (volumetricMesh->getElementType() == VolumetricMesh::TET)
    tetMesh = (TetMesh*) volumetricMesh; // such down-casting is safe in Vega
else
{
    printf("Error: not a tet mesh.\n");
    exit(1);
}

CorotationalLinearFEM * deformableModel = new CorotationalLinearFEM(tetMesh);
```

Note that the down-casting can be avoided if the mesh is known to be a tet mesh: simply initialize TetMesh directly using the constructor in the TetMesh class. We now have a valid deformable model, and it is possible to query internal forces and tangent stiffness matrices in any mesh configuration. Typically, we want to timestep the model in time, however, so let's proceed with building an integrator for the model. We need to first create a ForceModel object, to connect our deformable model to an integrator. We also need the mass matrix, and specify which model vertices (if any) are to be fixed. Then, we can initialize the integrator. Let us use the implicit backward Euler integrator. Note that the sparse linear system solver to use for implicit integration is selected by editing the file "integratorSolverSelection.h" inside the Integrator library. The default selection is to use Vega's conjugate gradient solver.

```
#include "corotationalLinearFEMForceModel.h"
#include "generateMassMatrix.h"
#include "implicitBackwardEulerSparse.h"
...

// create the class to connect the deformable model to the integrator
ForceModel * forceModel = new CorotationalLinearFEMForceModel(deformableModel);

int r = 3 * tetMesh->getNumVertices(); // total number of DOFs

double timestep = 0.0333; // the timestep

SparseMatrix * massMatrix;
// create consistent (non-lumped) mass matrix
GenerateMassMatrix::computeMassMatrix(tetMesh, &massMatrix, true);

// This option only affects PARDISO and SPOOLES solvers, where it is best
// to keep it at 0, which implies a symmetric, non-PD solve.
// With CG, this option is ignored.
int positiveDefiniteSolver = 0;

// constraining vertices 4, 10, 14 (constrained DOFs are specified 0-indexed):
int numConstrainedDOFs = 9;
int constrainedDOFs[9] = { 12, 13, 14, 30, 31, 32, 42, 43, 44 };

// (tangential) Rayleigh damping
double dampingMassCoef = 0.0; // "underwater"-like damping (here turned off)
```

```
double dampingStiffnessCoef = 0.01; // (primarily) high-frequency damping
```

```
// initialize the integrator
```

```
ImplicitBackwardEulerSparse * implicitBackwardEulerSparse = new
    ImplicitBackwardEulerSparse(r, timestep, massMatrix, forceModel,
        positiveDefiniteSolver, numConstrainedDOFs, constrainedDOFs,
        dampingMassCoef, dampingStiffnessCoef);
```

At this point, we can start timestepping our model! By default, initial conditions are zero deformation and zero velocity. Arbitrary initial conditions could be set via `IntegratorBase::SetState`. Let us apply some forces to the model, perform a couple of timesteps, and read the results.

```
// allocate buffer for external forces
```

```
double * f = (double*) malloc (sizeof(double) * r);
```

```
int numTimesteps = 10;
```

```
for(int i=0; i<numTimesteps; i++)
```

```
{
```

```
    // important: must always clear forces, as they remain in effect unless changed
```

```
    implicitBackwardEulerSparse->SetExternalForcesToZero();
```

```
    if (i == 0) // set some force at the first timestep
```

```
    {
```

```
        for(int j=0; j<r; j++)
```

```
            f[j] = 0; // clear to 0
```

```
        f[37] = -500; // apply force of -500 N to vertex 12, in y-direction, 3*12+1 = 37
```

```
        implicitBackwardEulerSparse->SetExternalForces(f);
```

```
    }
```

```
    implicitBackwardEulerSparse->DoTimestep();
```

```
}
```

```
// allocate buffer to read the resulting displacements
```

```
double * u = (double*) malloc (sizeof(double) * r);
```

```
implicitBackwardEulerSparse->GetqState(u);
```

And that's it – these were all the necessary steps to use Vega! The array `u` now contains the mesh vertex displacements after 10 simulation timesteps. We can now continue simulating, render the object, or perform collision detection (using some external software). We could then set the resulting contact forces as the external forces so that they will affect the deformations in the subsequent steps.

1.4 Vega file format

In this section, we document the Vega file format, `.veg`. The `.veg` format extends the open source volumetric mesh file format developed by Jonathan Shewchuk and employed by Stellar [KS09, Kli08] and TetGen [Han11] mesh generation packages; meshes produced in this format are easily loadable into Vega. Note that the Stellar webpage [KS09] contains a converter script that can convert other popular mesh formats into the Stellar/TetGen format.

For simulation, it is necessary to specify mesh material properties, in addition to geometry. The Vega file format achieves that, by introducing a set of additional keywords for material specification. Vega supports *heterogeneous material properties*, i.e., different parts of the mesh can have different material properties. In the extreme case, every mesh element can have a separate set of material properties. We provide several example `.veg` meshes in the `models` folder. For heterogeneous material properties, see the turtle example in `models/turtle`.

Basics: The Vega file format is ASCII. Lines starting with `*` denote a command. Lines starting with `#` are comments. Empty lines are ignored. Files can be nested using the `*INCLUDE` command. The effect of `*INCLUDE` is to include, at that point in the `.veg` file, the entire contents of the included file. Include files can include other files; they can nest arbitrarily.

Vertices are specified using the ***VERTICES** keyword. The second line gives the number of vertices, followed by integer “3” (three-dimensional simulation), optionally followed by more parameters, which are ignored. The subsequent lines give one vertex per line, in the format:

```
<vertex index> <x> <y> <z>
```

where the vertex index starts at 1 and increments by 1 for every vertex. Example:

```
*VERTICES
4 3 0 0
1 0.5 0.5 0.5
2 1.0 -0.5 0.5
3 -1.0 0.0 1.0
4 0.25 -0.25 0.5
```

Elements are specified using the ***ELEMENTS** keyword. The second line gives the element type, either “TET” or “CUBIC”. The third line gives the number of mesh elements, followed by the number of vertex elements, followed by some optional integers that are ignored. Subsequent lines give one element per line, in the format:

```
<element index> <vertex 1> ... <vertex n>
```

where the element index starts at 1 and increments by 1 for every element, and n is the number of element vertices. For tet, $n = 4$, for cubes, $n = 8$. Example:

```
*ELEMENTS
TETS
1 4 0
1 2 3 4 1
```

specifies a mesh with a single tet, with vertices 2, 3, 4, 1 (in that order). The ordering of vertices within the element matters: for tets, it affects the tet orientation and alters results, for cubic meshes, incorrect order usually gives bad results. For cubic meshes, the correct order of vertices is (0, 0, 0), (1, 0, 0), (1, 1, 0), (0, 1, 0), (0, 0, 1), (1, 0, 1), (1, 1, 1), (0, 1, 1).

Materials are specified using the ***MATERIAL** keyword. The first line gives the material name, the second line specifies the material properties. Two material specifications are supported: “ENU”, for any material that is parameterized by Young’s modulus (E) and Poisson’s ratio (ν), and “MOONEYRIVLIN” for Mooney-Rivlin materials. Most materials in Vega use the “ENU” specification: co-rotational linear FEM, StVK, invertible StVK, invertible neo-Hookean, etc. Example:

```
*MATERIAL mat1
ENU, 1000, 1E8, 0.40
```

specifies a material with mass density $1000\text{kg}/\text{m}^3$, Young’s modulus of $10^8\text{N}/\text{m}^2$, and Poisson’s ratio of 0.4. Example Mooney-Rivlin specification:

```
*MATERIAL myMaterialName
MOONEYRIVLIN, 0.5, 0.6, 1.0
```

specifies a Mooney-Rivlin material with $\mu_{01} = 0.5$, $\mu_{10} = 0.6$ and $v_1 = 1.0$. The implemented Mooney-Rivlin material is described in Section 3.5.5 of [Bow09].

Sets store a set of integer indices. They are used to store indices of elements that share the same material (=region). First line specifies the set name, followed by a comma-separated list of set elements. Elements should be sorted and 1-indexed. Example:

```
*SET set1
11, 17, 21, 37, 113, 220, 310, 555,
556, 557, 570, 601, 991, 1013, 1210, 1225
```

Regions: Elements that share material properties are organized into *regions*. A region is specified using a `*REGION` keyword. For example,

```
*REGION
set1, material1
```

creates a region consisting of the elements specified in the Set `set1`, and assigns material `material1` to it. In order to set the entire mesh to a material, you can use the built-in set `allElements`:

```
*REGION
allElements, material1
```

If the union of all specified regions does not contain all mesh elements, the remaining elements are assigned a material as follows. The assigned material is the last material mentioned in the `.veg` if the file specified at least one material. If no material was specified, the default material is assigned to the entire mesh. The default material is of type “ENU”, with default parameters $E = 10^6 N/m^2$, $\nu = 0.45$, $\rho = 1000 kg/m^3$, where E is Young’s modulus, ν is Poisson’s ratio and ρ is mass density. If specification of regions was omitted from a `.veg` file, the entire mesh is assigned the default material.

Easy re-use of Stellar/TetGen meshes: Suppose that the mesh vertices and elements are stored in `myMesh.node` and `myMesh.ele` files, in the standard Stellar/TetGen format. The following “template” `.veg` file is the shortest way to import those meshes into Vega:

```
*VERTICES
*INCLUDE myMesh.node
*ELEMENTS
TETS
*INCLUDE myMesh.ele
```

Of course, material properties and regions could be appended as described in the previous paragraphs.

1.5 Acknowledging

If you use Vega, we will appreciate if you acknowledge it. Please use the following citation:

Jernej Barbič, Fun Shing Sin, Daniel Schroeder:
Vega FEM Library. 2012. <http://www.jernejbarbic.com/vega>

Here is the BibTeX file:

```
@misc{Vega,
  author = "Jernej Barbič and Fun Shing Sin and Daniel Schroeder",
  title = "{Vega FEM Library}",
  year = "2012",
  note = "http://www.jernejbarbic.com/vega",
}
```

2 Libraries

Below is a listing of the libraries in the `libraries` folder. The purpose of each library as a whole is described, and more specific information is given on *selected* constituent classes and member functions to highlight important functionality. For subclasses, virtual functions are only re-listed if the subclass implements a previously-abstract function or substantially alters its functionality. Note that a few functions allocate memory which must be deleted by the caller: you can recognize such functions by the `**` (pointer to pointer) calling convention. As standard in C/C++, the memory is allocated with `malloc/free` for the built-in datatypes, and `new/delete` for all the other datatypes.

2.1 camera

class SphericalCamera Provides utilities for setting the OpenGL camera based upon a spherical-coordinate system centered at a specified focus point.

```
void Look()
```

Applies the current camera transformation to the active OpenGL matrix. Note that this does not run `glMatrixMode` or `glLoadIdentity`.

```
void MoveRight(double amount)
```

```
void MoveUp(double amount)
```

```
void MoveIn(double amount)
```

```
void ZoomIn(double amount)
```

Move the camera by a user-specified offset. Supply negative values to move/zoom in the opposite direction.

```
void SavePosition(char * filename)
```

```
void LoadPosition(char * filename)
```

Save or load the camera position to a file. `LoadPosition` prints a warning if the specified file does not exist.

2.2 configFile

class ConfigFile Parses values from a text configuration file. The syntax of the configuration file is user-defined. Options can be read as `int`, `bool`, `float`, `double`, and `char *` (C-string) types. See any of the `.config` files in the `examples` folder for an example of the config file syntax.

```
int addOption(char * optionName, T * destLocation)
```

Defined for `T` as `int`, `bool`, `float`, `double`, `char`. Adds a mandatory option with name `optionName`. When the config file is later parsed, any value found for this option is written to the variable pointed to by `destLocation`. If no value for this option is found, the parse is considered to have failed. Returns a non-zero value if the option has already been defined.

```
int addOptionOptional(char * optionName, T * destLocation, T defaultvalue)
```

```
int addOptionOptional(char * optionName, char * destLocation, char * defaultvalue)
```

Adds an optional option with name `optionName`. When the config file is later parsed, the variable pointed to by `destLocation` is set to the value found in the file, or to `defaultvalue` if no value is set. Returns a non-zero value if the option has already been defined.

```
int parseOptions(char * filename)
```

Parses the options in file `filename`, writing the option values it reads to the appropriate variables. A non-zero value is returned if any mandatory options are not specified.

2.3 corotationalLinearFEM

class CorotationalLinearFEM Implements the corotational linear finite element model described in [MG04]. The class can also compute the exact tangent stiffness matrix; the implementation is described in [Bar12].

```
CorotationalLinearFEM(TetMesh * tetMesh)
```

Initializes the model based upon an input tetrahedral mesh.

```
void GetStiffnessMatrixTopology(SparseMatrix ** stiffnessMatrixTopology)
```

Writes to `*stiffnessMatrixTopology` a newly allocated (using `new`) zero matrix containing the pattern of non-zero entries of the stiffness matrix.

```
void ComputeForceAndStiffnessMatrix(double * vertexDisplacements,
    double * internalForces, SparseMatrix * stiffnessMatrix, int warp=1)
```

Computes the internal forces and stiffness matrix given a vector `vertexDisplacements` of the displacements for the vertices of the tetrahedral mesh. If either `internalForces` or `stiffnessMatrix` is `NULL`, the function does not calculate or return the corresponding information. The `warp` parameter controls whether the simulation “warps” stiffnesses and therefore supports large deformations. When warping is enabled (`warp=1` or `warp=2`), the implementation supports large deformations. The default is `warp=1`, in which case the simulation uses the approximate tangent stiffness matrix as described in [MG04]. For `warp=2`, the class computes the exact tangent stiffness matrix; our implementation is described in [Bar12]. Such an exact matrix has better simulation properties; however, it requires approximately 1.5x the computation time of the approximate matrix. If warping is disabled (`warp=0`), one obtains the standard linear FEM simulation [Sha90]. Such simulation runs faster than the warped simulations because it timesteps the linear equation $M\ddot{u} + D\dot{u} + Ku = f$. It is only accurate under small displacements.

2.4 elasticForceModel

Provides implementation of the `ForceModel` base class for the deformable models supported by Vega. This makes it possible to use these deformable models with the integrators in Vega (`integrator` library).

class CorotationalLinearFEMForceModel : public ForceModel Exposes the internal force- and stiffness matrix-calculating functionality of the CorotationalLinearFEM material using the common interface of `ForceModel`.

```
CorotationalLinearFEMForceModel(CorotationalLinearFEM *
    corotationalLinearFEM, int warp=1)
```

Sets the `CorotationalLinearFEM` object for force and stiffness calculations. The parameter `warp` has the same meaning as in the `CorotationalLinearFEM` class.

```
void SetWarp(int warp)
```

Sets the warp parameter. This makes it possible to change the warp parameter at runtime.

class MassSpringSystemForceModel : public ForceModel Exposes the internal force- and stiffness matrix-calculating functionality of the mass-spring material using the common interface of `ForceModel`.

```
MassSpringSystemForceModel(MassSpringSystem * massSpringSystem)
Sets the MassSpringSystem object used for force and stiffness calculations.
```

class StVKForceModel : public ForceModel Exposes the internal force- and stiffness matrix-calculating functionality of the StVK material using the common interface of `ForceModel`.

```
StVKForceModel(StVKInternalForces * stVKInternalForces,
    StVKStiffnessMatrix * stVKStiffnessMatrix = NULL)
```

Sets the StVK objects used for internal forces and stiffness calculations. If no `StVKStiffnessMatrix` is provided, one is constructed based upon `stVKInternalForces`.

class IsotropicHyperelasticFEMForceModel : public ForceModel Exposes the internal force- and stiffness matrix-calculating functionality of the invertible FEM materials using the common interface of `ForceModel`.

```
IsotropicHyperelasticFEMForceModel(
    IsotropicHyperelasticFEM * isotropicHyperelasticFEM)
```

Sets the invertible-elements object used for internal forces and stiffness matrix calculations.

2.5 forceModel

This library provides the abstract base class for a force model used in the `integrator` library, i.e., a “black-box” function $u \mapsto f_{\text{int}}(u)$ and its gradient in

$$M\ddot{u} + (\alpha M + \beta K(u) + D)\dot{u} + f_{\text{int}}(u) = f_{\text{ext}}.$$

Any dynamical system described by such a differential equation can then be timestepped by the `integrator` library, by providing an implementation of f_{int} and its gradient, in a class derived from `ForceModel`.

For deformable simulations, class `ForceModel` connects integrators to internal forces and tangent stiffness matrix calculator classes. This allows the different deformable models in Vega to expose their internal forces and stiffness matrices to the integrator in a uniform way. Derived classes that implement f_{int} and its gradient for the deformable models in Vega can be found in the library `elasticForceModel`. The reason for why library `forceModel` is separated from `elasticForceModel` is so that the `integrator` library can be compiled and used independently from any deformable materials in Vega. Similarly, one can compile and use `forceModel` and `elasticForceModel` independently of `integrator`, making it possible to timestep the Vega deformable models with externally provided integrators.

class ForceModel Abstract base class for a force model $f_{\text{int}}(u)$ whose gradient (typically the tangent stiffness matrix) is a *sparse* matrix. All deformable models in Vega fall into this category. Note: dense gradient matrices occur in applications involving model reduction.

```
int Getr()
```

Returns `r`, the number of object degrees of freedom (typically three times the number of vertices or particles). Note that the `r` member variable must be set by a subclass.

```
virtual void GetInternalForce(double * u, double * internalForces) = 0
```

The internal forces arising from vertex displacements `u` are written to the array `internalForces`. Must be implemented by derived classes.

```
virtual void GetTangentStiffnessMatrixTopology(SparseMatrix **
    tangentStiffnessMatrix) = 0
```

Allocates (new) a `SparseMatrix` with the correct pattern of non-zero entries to hold a stiffness matrix for this material, and write the matrix pointer to `*tangentStiffnessMatrix`. Must be implemented by derived classes.

```
virtual void GetTangentStiffnessMatrix(double * u,
    SparseMatrix * tangentStiffnessMatrix) = 0
```

The tangent stiffness matrix arising from vertex displacements `u` is written to the previously allocated `tangentStiffnessMatrix`. Must be implemented by derived classes.

```
virtual void GetForceAndMatrix(double * u, double *
    internalForces, SparseMatrix * tangentStiffnessMatrix)
```

Writes out the internal forces and stiffness matrix arising from displacements `u`, using the implementations of the functions above. May be overloaded by derived classes.

2.6 getopt

```
int getopt(int argc, char **argv, opt_t opttable[])
```

Parses the `argc` and `argv` from a main function and extracts any specified option values. Modified from public domain code by Paul Edwards.

2.7 graph

class Graph Stores an undirected graph. The vertices of the graph are represented by the integers from 0 to the number of vertices minus one, and the graph structure is constant after being set in the constructor.

```
Graph(int numVertices, int numEdges, int * edges)
```

Initializes the graph, giving it `numVertices` vertices and `numEdges` edges. The input array of integers `edges` encodes the edges as subsequent pairs of vertex indices.

```
int IsNeighbor(int vtx1, int vtx2)
```

Returns 0 if vertices `vtx1` and `vtx2` are not neighbors, and returns 1 plus the index of `vtx2` in the list of `vtx1`'s neighbors otherwise.

2.8 insertRows

Provides utilities for the insertion and removal of elements from dense 1D arrays. This is useful, for example, when constraining (fixing) vertices in a deformable simulation.

```
void InsertRows(int mFull, double * xConstrained, double * x,  
int numFixedRows, int * fixedRows, int oneIndexed=0)
```

Given an array `xConstrained`, writes out a copy of this array to `x`, with zeroes placed among the elements at each index indicated in `fixedRows`, until the total desired size `mFull` of the output array is reached. `x` is assumed to be already allocated.

```
void RemoveRows(int mFull, double * xConstrained, double * x,  
int numFixedRows, int * fixedRows, int oneIndexed=0)
```

Given an array `x`, removes the elements at the indices of `x` given in `fixedRows` and writes the reduced-size array to `xConstrained`. `mFull` is the number of elements in `x`. `xConstrained` is assumed to be already allocated.

```
void FullDOFsToConstrainedDOFs(int mFull, int numDOFs,  
int * DOFsConstrained, int * DOFs, int numFixedRows,  
int * fixedRows, int oneIndexed=0)
```

Translates indices of elements in an unreduced array to the indices which would contain those same elements in the reduced array produced by calling `RemoveRows` with `fixedRows`. Given an array `DOFs` of indices to a hypothetical array and the array `fixedRows` of indices to be removed from the hypothetical array, writes to `DOFsConstrained[i]` the index in the hypothetical reduced array that would give the element at index `DOFs[i]` in the unreduced array.

2.9 integrator

Provides several implicit and explicit integrators. All integrators solve equations of the form

$$M\ddot{u} + (\alpha M + \beta K(u) + D)\dot{u} + f_{\text{int}}(u) = f_{\text{ext}}.$$

class IntegratorBase Serves as a base class to all the integrators. The class stores displacements, velocities, and accelerations for the simulation internally, and provides access to modify these buffers, but offers no actual timestepping capabilities; this is deferred to derived classes. The class also holds Rayleigh damping coefficients that specify how much the mass and stiffness matrix contribute to the damping matrix.

```
IntegratorBase(int r, double timestep,  
double dampingMassCoef=0.0, double dampingStiffnessCoef=0.0)
```

Sets the number of degrees of freedom `r`, timestep `timestep`, and coefficients for how much to add the mass matrix and stiffness matrix to the damping matrix. Allocates (`malloc`) a number of internal buffers of `r` doubles for holding the current displacement, velocity, internal forces, and so on.

`virtual ~IntegratorBase()`

Frees (free) the internal buffers.

`virtual void ResetToRest()`

Sets the internal position, velocity, and acceleration buffers to zero.

`virtual int SetState(double * q, double * qvel=NULL) = 0`

Resets internal position buffer to the values in `q`, and does likewise for internal velocity buffer if `qvel` is not NULL. Returns 0 if successful, 1 otherwise.

`void SetqState(const double * q, const double * qvel=NULL, const double * qaccel=NULL)`

`void GetqState(double * q, double * qvel=NULL, double * qaccel=NULL)`

Copies external position, velocity, and/or acceleration buffers to the internal buffers, or vice versa. Each buffer is copied only if the pointer to the external buffer is not NULL.

`void SetExternalForces(double * externalForces)`

`void AddExternalForces(double * externalForces)`

`void GetExternalForces(double * externalForces)`

Sets or adds the values from `externalForces` to the external forces buffer, or writes the external forces buffer to `externalForces`.

`virtual void SetTimestep(double timestep)`

`double GetTimestep()`

Sets or returns the timestep value.

`virtual int DoTimestep() = 0`

Implementations perform a timestep of the simulation, given the current values in the internal/external forces, position, velocity, and acceleration buffers. The resulting position, velocity, and acceleration are saved to these buffers. Returns 0 if and only if the timestep is completed without error.

class IntegratorBaseSparse : public IntegratorBase A base class for integrators for dynamical systems characterized by (large) sparse matrices. All integrators in Vega are of this type. These integrators use a `ForceModel` class to obtain the internal forces and stiffness matrices. Stiffness matrices are sparse and stored using the `SparseMatrix` class. `IntegratorBaseSparse` stores two sparse matrices: the mass matrix and the damping matrix to be applied in addition to mass- and stiffness-based damping specified in `IntegratorBase`.

`IntegratorBaseSparse(int r, double timestep, SparseMatrix * massMatrix,`

`ForceModel * forceModel, int numConstrainedDOFs=0, int * constrainedDOFs=NULL,`

`double dampingMassCoef=0.0, double dampingStiffnessCoef=0.0)`

Initializes the number of degrees of freedom `r`, the timestep, the mass matrix, which degrees of freedom are constrained, and the `ForceModel`. The internal damping `SparseMatrix` is set to zero.

`virtual void SetForceModel(ForceModel * forceModel)`

Sets the `ForceModel` object to `forceModel`. This makes it possible to change the force model at runtime.

`virtual void SetDampingMatrix(SparseMatrix * dampingMatrix)`

Sets the damping matrix D to `dampingMatrix`. This matrix will be added to the matrix produced according to the mass and stiffness damping coefficients to get the total damping matrix for the simulation.

`virtual double GetForceAssemblyTime()`

`virtual double GetSystemSolveTime()`

Return the time taken in `DoTimestep` to generate the force/stiffness values and to solve the linear system while timestepping, respectively. It is up to subclasses to calculate these values in `DoTimestep`.

```
virtual double GetKineticEnergy()
virtual double GetTotalMass()
```

Calculates and returns the kinetic energy or total mass of the simulation, based upon the mass matrix and the velocity buffer of the class.

class ImplicitNewmarkSparse : public IntegratorBaseSparse Implements implicit Newmark integration, and expands `IntegratorBaseSparse` with features common to Newmark-style integrators.

```
ImplicitNewmarkSparse(int r, double timestep,
    SparseMatrix * massMatrix, ForceModel * forceModel,
    int positiveDefiniteSolver=0, int numConstrainedDOFs=0,
    int * constrainedDOFs=NULL, double dampingMassCoef=0.0,
    double dampingStiffnessCoef=0.0, int maxIterations = 1,
    double epsilon = 1E-6, double NewmarkBeta=0.25,
    double NewmarkGamma=0.5, int numSolverThreads=0)
```

Initializes `maxIterations`, `epsilon`, `NewmarkBeta`, `NewmarkGamma`, and `numSolverThreads` parameters, and forwards the other parameters to the `IntegratorBaseSparse` constructor. For PARDISO and SPOOLES solvers, we found best performance with 2-3 threads; more threads usually decreased performance.

```
virtual void SetTimestep(double timestep)
Sets the timestep.
```

```
virtual int SetState(double * q, double * qvel=NULL)
Sets the position (and optionally the velocity) buffer, and calculates the acceleration buffer accordingly using implicit Newmark, assuming no external force. Returns 0 if successful, 1 otherwise.
```

```
virtual int DoTimestep()
Runs a timestep of implicit Newmark, updating the internal position, velocity, and acceleration buffers accordingly. Returns 0 if and only if the timestep is completed successfully.
```

```
void SetNewmarkBeta(double NewmarkBeta)
void SetNewmarkGamma(double NewmarkGamma)
Set the value of the beta and gamma parameters, respectively, for Newmark integrators. No checking is performed to see if these values are in the appropriate range.
```

```
void UseStaticSolver(bool useStaticSolver)
Sets whether to use a static solver. The class defaults to dynamic.
```

class ImplicitBackwardEulerSparse : public ImplicitNewmarkSparse Replaces the `DoTimestep` function of `ImplicitNewmarkSparse` to implement implicit Euler integration.

```
ImplicitBackwardEulerSparse(int r, double timestep,
    SparseMatrix * massMatrix,
    ForceModel * forceModel,
    int positiveDefiniteSolver=0, int numConstrainedDOFs=0,
    int * constrainedDOFs=NULL, double dampingMassCoef=0.0,
    double dampingStiffnessCoef=0.0, int maxIterations = 1,
    double epsilon = 1E-6, int numSolverThreads=0)
```

Sets the integrator parameters with the `ImplicitNewmarkSparse` constructor.

```
virtual int SetState(double * q, double * qvel=NULL)
```

Sets the position (and optionally the velocity) buffer based on the parameters, and calculates the appropriate acceleration buffer values using implicit Euler, assuming no external forces. Returns 0 if successful, 1 otherwise.

```
virtual int DoTimestep()
```

Runs a timestep of implicit Euler, and updates the internal position, velocity, and acceleration buffers accordingly. Returns 0 if successful, 1 otherwise.

class CentralDifferencesSparse : public IntegratorBaseSparse Implements the explicit central differences integrator.

```
CentralDifferencesSparse(int numDOFs, double timestep,
    SparseMatrix * massMatrix,
    ForceModel * forceModel,
    int numConstrainedDOFs=0, int * constrainedDOFs=NULL,
    double dampingMassCoef=0.0, double dampingStiffnessCoef=0.0,
    int tangentialDampingMode=1, int numSolverThreads=0)
```

Initializes the integrator settings via the `IntegratorBaseSparse` constructor, and selects the number of threads to use for the sparse linear solver. Tangential damping mode controls how often the Rayleigh damping matrix is recomputed. This damping matrix depends on the tangent stiffness matrix, which changes in time. When 0, the damping matrix is never updated. The system matrix is then constant, so one can factor it only once. However, this is not recommended for large deformations as it leads to damping artifacts. When `tangentialDampingMode > 0`, the damping matrix is updated every “`tangentialDampingMode`”th timestep. Default is 1, i.e., update at every timestep.

```
virtual int SetState(double * q, double * qvel=NULL)
```

Sets the position (and optionally velocity) buffers. Always returns 0.

```
virtual int DoTimestep()
```

Runs a timestep of explicit central differences, and updates the internal position, velocity and acceleration buffers accordingly. Returns 0 if successful, 1 otherwise.

class EulerSparse : public IntegratorBaseSparse Implements explicit Euler integration, with a flag to enable symplectic Euler integration. Because this class never forms the tangent stiffness matrix, damping is controlled via mass-based damping only.

```
EulerSparse(int r, double timestep, SparseMatrix * massMatrix,
    ForceModel * forceModel,
    int symplectic=0, int numConstrainedDOFs=0,
    int * constrainedDOFs=NULL, double dampingMassCoef=0.0)
```

Initializes the integrator settings via the `IntegratorBaseSparse` constructor, and sets whether to perform symplectic integration.

```
virtual int SetState(double * q, double * qvel=NULL)
```

Sets the position (and optionally velocity) buffers based on the parameters, and sets the acceleration buffer via explicit (or symplectic) Euler assuming no external forces. Returns 0 if successful, 1 otherwise.

```
virtual int DoTimestep()
```

Runs a timestep of explicit (or symplectic) Euler, and updates the internal position, velocity, and acceleration buffers accordingly. Returns 0 if successful, 1 otherwise.

2.10 isotropicHyperelasticFEM

Provides classes to calculate the energy, internal forces and stiffness matrices of a variety of invertible material methods.

class IsotropicHyperelasticFEM Provides an implementation of the invertible FEM method presented in [ITF04]. Also computes the tangent stiffness matrix as presented in [TSIF05]. Supports isotropic hyperelastic materials with energy defined in terms of the three invariants I, II, III . A few such example materials are provided, and the user can define her own custom materials simply by deriving from **IsotropicMaterial**.

```
IsotropicHyperelasticFEM(TetMesh * tetMesh,  
    IsotropicMaterial *  
    isotropicMaterial, int numThreads=1,  
    double principalStretchThreshold=-DBL_MAX,  
    bool addGravity=false, double g=9.81)
```

Reads the mesh `tetMesh`, pre-computes the area weighted vertex normals, the inverse of the Dm (as in [ITF04]), and the derivative of the deformation gradient with respect to the vertex displacements. Parameter `principalStretchThreshold` specifies a minimum value for the principal stretches of element deformation gradients; stretches below this value will be clamped to it for force and stiffness calculations. The `isotropicMaterial` must be pre-created: you can use one of our provided classes, or derive from the **IsotropicMaterial** to create your own custom isotropic hyperelastic materials.

```
double ComputeEnergy(double * vertexDisplacements)
```

Given the array `vertexDisplacements` of vertex displacements from rest position, returns the non-linear elastic strain energy of the mesh. The energy is calculated based upon the principal stretches of the SVD-diagonalized deformation gradients of the individual mesh elements.

```
void ComputeForces(double * vertexDisplacements, double * internalForces)
```

Given the array `vertexDisplacements` of vertex displacements, writes the resulting vertex forces to the pre-allocated array `internalForces`. Forces are calculated from the principal stretches of the SVD-diagonalized deformation gradients of the individual mesh elements.

```
void GetStiffnessMatrixTopology(SparseMatrix ** stiffnessMatrixTopology)
```

Allocates (new) a **SparseMatrix** with the correct pattern of non-zero entries to hold a stiffness matrix for this material, and writes the matrix pointer to `*tangentStiffnessMatrix`.

```
void GetTangentStiffnessMatrix(double * u, SparseMatrix * tangentStiffnessMatrix)
```

Writes the stiffness matrix arising from vertex displacements `u` to the pre-allocated `tangentStiffnessMatrix`. The stiffness matrix is calculated based upon the principal stretches and rotation matrices from the SVD diagonalization of the deformation gradient of each element.

class IsotropicMaterial Serves as a base class for isotropic hyperelastic materials which define energy is defined in terms of the three invariants I, II, III . This represents the material of a single element (tet or cube).

```
virtual double ComputeEnergy(int elementIndex, double * invariants) = 0
```

Returns the energy of the element given the three invariants `invariants[0]`, `invariants[1]`, and `invariants[2]`. The integer variable `elementIndex` makes it possible to make the material properties vary from element to element (heterogeneous material properties).

```
virtual void ComputeEnergyGradient(int elementIndex,  
    double * invariants, double * gradient) = 0
```

Given the three invariants `invariants[0]`, `invariants[1]`, `invariants[2]` computes the derivative of the energy with respect to the invariants and writes the result to `gradient[0]`, `gradient[1]`, `gradient[2]`.


```
virtual void ComputeEnergyHessian(int elementIndex,
    double * invariants, double * hessian) = 0
```

Given the three invariants `invariants[0]`, `invariants[1]`, `invariants[2]` computes the second derivative of the energy with respect to the invariants and writes the result to `hessian` where `hessian[0] = $\partial^2\Psi/\partial I^2$` , `hessian[1] = $\partial^2\Psi/\partial I\partial II$` , `hessian[2] = $\partial^2\Psi/\partial I\partial III$` , `hessian[3] = $\partial^2\Psi/\partial II^2$` , `hessian[4] = $\partial^2\Psi/\partial II\partial III$` , and `hessian[5] = $\partial^2\Psi/\partial III^2$`

To define your own isotropic hyperelastic material, simply derive from `IsotropicMaterial` and implement functions `ComputeEnergy`, `ComputeEnergyGradient`, `ComputeEnergyHessian`.

class StVKIsotropicMaterial : public IsotropicMaterial StVK material that supports spatially varying material parameters. Provides functions to compute the energy, gradient, and Hessian of the StVK material given the three invariants I, II, III . The implemented StVK material is described in [BW08], page 158.

```
StVKIsotropicMaterial(TetMesh * tetMesh)
```

The material parameters are read from the tet mesh, and are cached in the constructor. Throws an exception if the provided tet mesh does not consist of materials specifying E, ν .

class HomogeneousIsotropicStVKMaterial : public IsotropicMaterial StVK material with homogeneous (spatially global) material parameters. Provides functions to compute the energy, gradient, and Hessian of the StVK material given the three invariants I, II, III .

```
HomogeneousIsotropicStVKMaterial(double E, double nu)
```

Initializes the Young's modulus E and the Poisson ratio ν of the material.

class NeoHookeanIsotropicMaterial : public IsotropicMaterial Neo-Hookean material that supports spatially varying material parameters. Provides functions to compute the energy, gradient, and Hessian of the Neo-Hookean material given the three invariants I, II, III . The implemented neo-Hookean material is described in [BW08], page 162.

```
NeoHookeanIsotropicMaterial(TetMesh * tetMesh)
```

The material parameters are read from the tet mesh, and are cached in the constructor. Throws an exception if the provided tet mesh does not consist of materials specifying E, ν .

class HomogeneousNeoHookeanIsotropicMaterial : public IsotropicMaterial Neo-Hookean material with homogeneous (spatially global) material parameters. Provides functions to compute the energy, gradient, and Hessian of the neo-Hookean material given the three invariants I, II, III .

```
HomogeneousNeoHookeanIsotropicMaterial(double E, double nu)
```

Initializes the Young's modulus E and the Poisson ratio ν of the material.

class MooneyRivlinIsotropicMaterial : public IsotropicMaterial Mooney-Rivlin material that supports spatially varying material parameters.

```
MooneyRivlinIsotropicMaterial(TetMesh * tetMesh)
```

The material parameters are read from the tet mesh, and are cached in the constructor. Throws an exception if the provided tet mesh does not consist of Mooney-Rivlin materials. Provides functions to compute the energy, gradient, and Hessian of the Mooney-Rivlin material given the three invariants I, II, III . The implemented Mooney-Rivlin material is described in Section 3.5.5 of [Bow09].

class HomogeneousMooneyRivlinIsotropicMaterial : **public IsotropicMaterial** Mooney-Rivlin material with homogeneous (spatially global) material parameters. Provides functions to compute the energy, gradient, and Hessian of the Mooney-Rivlin material given the three invariants I, II, III .

`HomogeneousMooneyRivlinIsotropicMaterial(double mu01, double mu10, double v1)`
 Initializes the Mooney-Rivlin material parameters μ_{01}, μ_{10}, v_1 .

2.11 lighting

class Lighting Provides tools to read an OpenGL lighting configuration from a file (using the `configFile` library) and apply it to the current rendering context. Up to eight light sources are supported. The lighting settings cannot be changed after a `Lighting` object is constructed.

`Lighting(char * configFilename)`
 Reads the specified file and extracts lighting information. Throws an exception if `ConfigFile` reports an error (i.e., an unspecified mandatory option) in the file.

`void LightScene()`
 Enables `GL_LIGHTING` and applies the lighting settings contained in the class.

2.12 loadList

class LoadList Utilities for saving and loading lists of integers to and from text files.

`static int load(char * filename, int * numListEntries, int ** listEntries, int offset=0)`

Loads a list of integers from `filename`, writing the size of this list to the `int` pointed to by `numListEntries`, and writing the address of the newly-allocated list of read integers to the `int *` pointed to by `listEntries`. Optionally, each integer has `offset` subtracted from it when read. Returns 1 if an error occurs, and 0 otherwise.

`static int save(char * filename, int numListEntries, int * listEntries, int offset=0)`
 Saves to `filename` a list of `numListEntries` integers pointed to by `listEntries`. Optionally, each integer has `offset` added to it before being written. Returns 1 on an error, and 0 otherwise.

`static void sort(int numListEntries, int * listEntries)`
 Sorts a list of `numListEntries` entries in-place.

2.13 macros

Contains macros for calculating three-dimensional dot and cross products.

2.14 massSpringSystem

Provides a collection of classes to initialize and work with mass-spring systems.

class MassSpringSystem Implements a mass-spring system, a collection of weighted particles connected by springs.

`MassSpringSystem(int numParticles, double * masses,
 double * restPositions, int numEdges, int * edges,
 int * edgeGroups, int numMaterialGroups,
 double * groupStiffness, double * groupDamping,
 int addGravity=0)`

Constructs a mass-spring system of `numParticles` particles and `numEdges` springs. `masses` specifies the mass of each particle, and `restPositions` contains three subsequent entries for the three-dimensional coordinates of each particle. `edges` contains $2 \cdot \text{numParticles}$ particle indices, with each pair defining the endpoints of a spring. `edgeGroups` specifies for each spring the index of which of the `numMaterialGroups` material property groups it belongs to, while `groupStiffness` and `groupDamping` specify the stiffness and damping parameters for each of these groups.

```
MassSpringSystem(int numParticles, double * restPositions,
    int numQuads, int * quads, double surfaceDensity,
    double tensileStiffness, double shearStiffness,
    double bendStiffness, double damping, int addGravity=0)
```

Constructs a mass-spring system based upon an organization of the `numParticles` particles into `numQuads` quads in a quad mesh. The $4 \cdot \text{numQuads}$ -entry array `quads` specifies each quad with four subsequent particle indices. Particle rest positions are defined as above, and particle masses are defined based upon `surfaceDensity` and the quad surface areas. The stiffness and damping settings determine the spring properties.

```
MassSpringSystem(int numParticles, double * restPositions,
    MassSpringSystemElementType elementType, int numElements,
    int * elements, double density, double tensileStiffness,
    double damping, int addGravity=0)
```

Constructs a mass-spring system based upon a tetrahedral mesh or cubic mesh, depending on whether `elementType` is set to TET or CUBE. The `numParticles` particles are organized into `numElements` tetrahedra or cubes, which are specified in the array `elements` (of size 4 times or 8 times `numElements`) by four or eight subsequent particle indices per element. Particle masses are set according to `density` and the element volumes, while spring properties are set according to the stiffness and damping variables.

```
void SetGravity(bool addGravity, double g=9.81)
```

Sets whether gravity is enabled, and optionally sets the gravitational constant.

```
void ComputeForce(double * u, double * f, bool addForce=false)
```

Computes the forces acting on the mass-spring particles given their displacements `u` and writes the result to the pre-allocated array `f`. If `addForce` is true, the forces are added to any existing force values in `f`. **Important:** the force `f` has the same sign as with the other deformable models in Vega, i.e., it appears on the left side in equation $M\ddot{u} + D\dot{u} + f = f_{\text{ext}}$. If you want `f` to be interpreted as an external mass-spring force acting on the particles, you must flip the sign of `f`. The same applies to the damping forces, stiffness matrices and their Hessian corrections.

```
void ComputeDampingForce(double * uvel, double * f, bool addForce=false)
```

Computes the damping forces acting on the mass-spring particles given their current velocities `uvel`, and writes the result to the pre-allocated array `f`. If `addForce` is true, the forces are added to any existing forces in `f`.

```
void GetStiffnessMatrixTopology(SparseMatrix ** stiffnessMatrixTopology)
```

Writes to `*stiffnessMatrixTopology` a newly allocated (using `new`) zero matrix containing the pattern of non-zero entries of the stiffness matrix.

```
void ComputeStiffnessMatrix(double * u, SparseMatrix * K, bool addMatrix=false)
```

Computes the stiffness matrix of the mass-spring system given the vertex displacements `u` and writes the result to the `SparseMatrix` pointed to by `K`. If `addMatrix` is true, the stiffness matrix is added to the existing entries of `*K`.

```
class MassSpringSystemFromTetMesh
```

```
    static int GenerateMassSpringSystem(TetMesh * tetMesh,
```

```

    MassSpringSystem ** massSpringSystem, double density,
    double tensileStiffness, double damping, int addGravity=0)
    Allocates a new MassSpringSystem object based upon the tetrahedral mesh tetMesh and the specified
    material properties, and writes its pointer to *massSpringSystem. Returns 1 on error, 0 otherwise.

```

class MassSpringSystemFromTetMeshConfigFile

```

    int GenerateMassSpringSystem(char * configFilename,
    MassSpringSystem ** massSpringSystem,
    MassSpringSystemTetMeshConfiguration * massSpringSystemTetMeshConfiguration = NULL)

```

Allocates a new MassSpringSystem based upon the contents of the file `configFilename`, and writes its pointer to `*massSpringSystem`. If the last argument is not NULL, the options read from the config file are written to the pointed-to object. Returns 1 on error, 0 otherwise.

class MassSpringSystemFromObjMesh

```

    int GenerateMassSpringSystem(ObjMesh * quadMesh,
    MassSpringSystem ** massSpringSystem, double surfaceDensity,
    double tensileStiffness, double shearStiffness,
    double bendStiffness, double damping, int addGravity=0)

```

Allocates a new MassSpringSystem based upon a .obj mesh `quadMesh` and the given material properties, and writes its pointer to `*massSpringSystem`. `quadMesh` must have only quadrilateral faces. Returns 1 on error, 0 otherwise.

class MassSpringSystemObjMeshConfiguration

```

    int GenerateMassSpringSystem(char * configFilename,
    MassSpringSystem ** massSpringSystem,
    MassSpringSystemObjMeshConfiguration * massSpringSystemObjMeshConfiguration = NULL)

```

Allocates a new MassSpringSystem based upon the contents of the file `configFilename`, and writes its pointer to `*massSpringSystem`. If the last argument is not NULL, the options read from the config file are written to the pointed-to object. Returns 1 on error, 0 otherwise.

class MassSpringSystemFromCubicMesh

```

    static int GenerateMassSpringSystem(CubicMesh * CubicMesh,
    MassSpringSystem ** massSpringSystem, double density,
    double tensileStiffness, double damping, int addGravity=0)

```

Allocates a new MassSpringSystem based upon the given CubicMesh and material properties, and writes its pointer to `*massSpringSystem`. Returns 1 on error, 0 otherwise.

class MassSpringSystemCubicMeshConfiguration

```

    int GenerateMassSpringSystem(char * configFilename,
    MassSpringSystem ** massSpringSystem,
    MassSpringSystemCubicMeshConfiguration * massSpringSystemCubicMeshConfiguration = NULL)

```

Allocates a new MassSpringSystem based upon the contents of the file `configFilename`, and writes its pointer to `*massSpringSystem`. Returns 1 on error, 0 otherwise.

class RenderSprings

```

    void Render(MassSpringSystem * massSpringSystem, double * u)

```

Uses GL_LINES to draw each of the springs in the mass-spring system pointed to by `massSpringSystem` given the current particle displacements `u`. Edges are color-coded based on which material group they belong to.

2.15 matrix

Provides functions to write and read dense matrices to and from a file. The templated functions, where type `real` may be `float` or `double`, operate on `real`-typed matrices stored as column-major arrays. If an error occurs, functions ending in an underscore exit with non-zero status, and functions without an underscore return a non-zero integer.

```
int ReadMatrixFromDisk(char* filename, int * m, int * n, real ** matrix)
int ReadMatrixFromDisk_(char* filename, int * m, int * n, real ** matrix)
```

Reads a binary-format matrix from `filename` into a newly-allocated (`malloc`) array, writes the pointer to this array to `*matrix`, and writes the matrix dimensions to `*m` and `*n`. An error occurs if the file reports a different matrix size or if file IO fails.

```
int WriteMatrixToDisk(char* filename, int m, int n, real * matrix)
int WriteMatrixToDisk_(char* filename, int m, int n, real * matrix)
```

Writes an `m`-by-`n` matrix stored in the array `matrix` to `filename`. An error occurs if file IO fails.

```
int ReadMatrixFromDiskTextFile(char* filename, int * m, int * n, real ** matrix)
int WriteMatrixToDiskTextFile(char* filename, int m, int n, real * matrix)
```

Read and write matrices similarly to `ReadMatrixFromDisk` and `WriteMatrixToDisk`, but encode the `float` or `double` values in text format.

```
int ReadMatrixFromDiskListOfFiles(char * fileList, int * m, int * n, real ** matrix)
```

Concatenates the columns of the matrices from several (binary-format) matrix files into a single matrix. The file of name `fileList` is read, and each line of the file is assumed to be a matrix filename. Each of these matrices must have the same number of rows. The resulting concatenation is written to `*matrix`, and its dimensions are written to `*m` and `*n`.

2.16 minivector

class Vec2d Provides basic functionality for two-dimensional vectors with `double`-valued coordinates. Overloads addition and subtraction of vectors, scalar (pre- and post-)multiplication and division, equality testing, `[]` member access, and `<<` output with `ostream`.

```
Vec2d()
Vec2d(double x, double y)
Vec2d(double entry)
```

Does no initialization, sets the two coordinates to `x` and `y`, or sets both to `entry`.

```
friend double dot(const Vec2d & vec1, const Vec2d & vec2)
```

Returns the dot product of `vec1` and `vec2`.

```
friend Vec2d norm(const Vec2d & vec1)
```

Returns a normalized copy of `vec1`.

class Vec3d Provides three-dimensional vector functionality with `double`-valued coordinates. Overloads addition and subtraction of vectors, scalar (pre- and post-)multiplication and division, equality testing, member access, and `<<` output with `ostream`.

```
Vec3d()
Vec3d(double x, double y, double z)
Vec3d(double entry)
Vec3d(const double * vec)
Vec3d(const Vec3d & vec)
```

Does no initialization, sets the three coordinates, sets all three coordinates to `entry`, sets coordinates from an array, or copies from another `Vec3d`.

```
friend double dot(const Vec3d & vec1, const Vec3d & vec2)
Returns the dot product of vec1 and vec2.
```

```
friend Vec3d cross(const Vec3d & vec1, const Vec3d & vec2)
Returns the cross product of vec1 and vec2.
```

```
friend Vec3d norm(const Vec3d & vec1)
Returns a normalized copy of vec1.
```

class Mat3d Implements a three-dimensional matrix. Overloads operators for addition and subtraction of matrices, scalar (pre- and post-) multiplication, and `<<` output for `ofstream`. The `[]` operator can be used to access the row vectors, whose elements can be accessed with a second `[]`.

```
Mat3d()
Mat3d(double x0, double x1, double x2, ..., double x8)
Mat3d(double mat[9])
Mat3d(Vec3d rows[3])
Mat3d(Vec3d row0, Vec3d row1, Vec3d row2)
Mat3d(double diag)
```

Does no initialization, initializes all 9 entries (in row-major order) either individually or with an array, sets the three rows either from an array or individually, or initializes a diagonal matrix.

```
void set(double x0, double x1, double x2, ..., double x8)
void set(double value)
Set the values of all 9 matrix entries in row-major order, or set all entries to value.
```

```
friend Mat3d tensorProduct(const Vec3d & vec1, const Vec3d & vec2)
Returns the tensor product of the two vectors. If vec1 and vec2 are interpreted as column vectors, column i of the output is vec1 scalar-multiplied by vec2[i].
```

```
friend Mat3d inv(const Mat3d &)
Returns the inverse of the input matrix. Performs no checking to ensure the inverse exists.
```

```
friend double det(const Mat3d & mat)
Returns the determinant of mat.
```

```
friend Mat3d trans(const Mat3d & mat)
Returns the transpose of mat.
```

```
void convertToArray(double * array)
Writes the matrix entries to the array in row-major order.
```

```
friend void eigen_sym(Mat3d & a, Vec3d & eig_val, Vec3d eig_vec[3])
Calculates the eigenvalues and eigenvectors of the assumed-to-be-diagonal matrix and writes the eigenvalues to eig_val and the eigenvectors to eig_vec. This routine calls a public domain routine eigen_decomposition, available in eig3.h and eig3.cpp. These two files were downloaded from: http://barnesc.blogspot.com/2007/02/eigenvectors-of-3x3-symmetric-matrix.html. The website states that the code is public domain, and that it was obtained from the public-domain code “JAMA : A Java Matrix Package”: http://math.nist.gov/javanumerics/jama/.
```

2.17 objMesh

Provides utilities to load/save .obj meshes (including mtl files), access/modify the geometry of a mesh, and render meshes using OpenGL. The mesh storage structure is as follows: there are three global arrays, storing positions of mesh vertices, texture coordinates and normals. Nested classes `ObjMesh::Group`, `ObjMesh::Face` and `ObjMesh::Vertex` then represent the mesh geometry by giving integer indices into these three global arrays, for each vertex on every face. This means that a vertex that is shared by two (or more) faces may have different normals or texture coordinates in the two faces.

class `ObjMesh::Vertex` Constructed by the `ObjMesh` constructor. Stores the information about a particular vertex in the mesh.

`unsigned int getPositionIndex()`

Returns the index of this vertex's `Vec3d` position vector in the global array of position vectors in the `ObjMesh`. The position vector may be accessed with `ObjMesh::getPosition`.

`bool hasNormal()`

`bool hasTextureCoordinate()`

Returns whether the vertex has a normal vector or a texture coordinate vector. Must be called before requesting the normal vector or texture coordinate indices with the functions below.

`unsigned int getNormalIndex()`

`unsigned int getTextureCoordinateIndex()`

Returns the index of this vertex's `Vec3d` normal vector or texture coordinate vector in the global array of normal vectors or texture coordinate vectors in the `ObjMesh`. These vectors may be accessed with the `ObjMesh::getNormal` and `ObjMesh::getTextureCoordinate` functions, respectively. Aborts if the vertex has no normal or texture coordinate.

`void setPositionIndex(unsigned int positionIndex)`

`void setNormalIndex(unsigned int normalIndex)`

`void setTextureCoordinateIndex(unsigned int textureCoordinateIndex)`

Sets the position index, normal index, or texture coordinate index for this vertex.

class `ObjMesh::Material` Constructed by the `ObjMesh` constructor. Stores material properties settings from an .obj file.

`Vec3d getKa()`

`Vec3d getKd()`

`Vec3d getKs()`

Returns a vector containing the RGB components of the ambient, diffuse, or specular lighting coefficients, respectively.

`double getShininess()`

`double getAlpha()`

Returns the shininess value for specular reflections or the alpha (transparency) value.

`void setAlpha(double alpha)`

Sets the alpha (transparency) value of the material.

class `ObjMesh::Face` Constructed by the `ObjMesh` constructor. Stores a vector of `ObjMesh::Vertex` objects that constitute a face of the model.

`void addVertex(const Vertex& v)`

Adds vertex `v` to the end of the list of vertices constituting the face.

`size_t getNumVertices()`

Returns the number of vertices in the face.

`Vertex getVertex(unsigned int vertex)`

`Vertex * getVertexHandle(unsigned int vertex)`

Returns a copy of or a pointer to the face vertex at index `vertex`.

`bool hasFaceNormal()`

Returns whether the face has a precomputed face normal.

`void setFaceNormal(Vec3d & normal)`

Sets the face normal of the face.

`Vec3d getFaceNormal()`

Returns the precomputed face normal, asserting that it has previously been set.

class `ObjMesh::Group` Constructed by the `ObjMesh` constructor. Stores a **vector** of faces constituting a group in the `.obj` file.

`void addFace(const Face& face)`

Adds a face to the **vector** of faces.

`size_t getNumFaces()`

Returns the number of faces in the group.

`Face getFace(unsigned int face)`

`Face getFaceHandle(unsigned int face)`

Returns a copy of or a pointer to the face at index `face`.

`std::string getName()`

Returns the name of the group.

`unsigned int getMaterialIndex()`

`void setMaterialIndex(unsigned int index)`

Gets or sets the material index for this group. The index refers to the global array of materials in the `ObjMesh`.

`void removeFace(unsigned int i)`

Removes the face at index `i`.

class `ObjMesh` Reads and stores the model information from an `.obj` file.

`ObjMesh(const std::string& filename, int verbose=1)`

Initializes the object based upon the contents of the `.obj` file `filename`. Throws an `ObjMeshException` if any failure occurs during the process.

`ObjMesh()`

Initializes an empty model.

`std::string filename()`

Returns the name of the `.obj` file used to initialize the object. Returns an empty string if the object was not constructed from a file.


```

void printInfo()
Prints detailed information about the model.

bool isTriangularMesh()
bool isQuadrilateralMesh()
Returns whether the mesh is triangular or quadrilateral.

void triangulate()
Subdivides faces so that the mesh consists only of triangular faces.

void getBoundingBox(double expansionRatio, Vec3d * bmin, Vec3d * bmax)
Return a bounding box of the mesh. The tightest fitting box is scaled by expansionRatio. For
expansionRatio=1, one obtains the tight-fitting bounding box.

Vec3d getPosition(int vertexIndex)
Vec3d getTextureCoordinate(int textureCoordinateIndex)
Vec3d getNormal(int normalIndex)
Returns the vertex position, texture coordinate, or normal vector held at index vertexIndex,
textureCoordinateIndex, normalIndex, in the global array of vertex position, texture coordinate, or normal
vectors.

Vec3d getPosition(Vertex & vertex)
Vec3d getTextureCoordinate(Vertex & vertex)
Vec3d getNormal(Vertex & vertex)
Returns the vertex position, texture coordinate, or normal vector for the given vertex.

void setPosition(int vertexIndex, Vec3d & position)
void setTextureCoordinate(int textureCoordinateIndex, Vec3d & textureCoordinate)
void setNormal(int normalIndex, Vec3d & normal)
void setPosition(Vertex & vertex, Vec3d & position)
void setTextureCoordinate(Vertex & vertex, Vec3d & textureCoordinate)
void setNormal(Vertex & vertex, Vec3d & normal)
Sets the vertex position, texture coordinate or normal in the array of global vertex positions, texture
coordinates or normals. Returns the vertex position, texture coordinate, or normal vector for the given
vertex.

Material material(unsigned int index)
Material * materialHandle(unsigned int index)
Returns a copy of or a pointer to the material at index index in the global array of materials.

size_t getNum...()
void add...(...)
For each of Vertices, Faces, Normals, TextureCoordinates, Groups, or Materials, returns the number
of them in the object, or adds a new one of them to the object.

void buildFaceNormals()
Calculates the geometric normals of each face (assuming counter-clockwise winding) and caches them in
the ObjMesh::Face objects. This must be done before using any of the vertex-normal generation functions
below.

void buildVertexFaceNeighbors()
void clearVertexFaceNeighbors()

```

(Re-)Builds or clears an internally-stored structure for easy lookup of faces neighboring a given vertex. This must be done before using any of the vertex-normal generation functions below.

```
void buildVertexNormals(double angle)
```

The functions `buildFaceNormals` and `buildVertexFaceNeighbors` must be called first. Sets the vertex normals of the mesh to create a smooth-looking surface except for at angles exceeding `angle`.

```
void buildVertexNormalsFancy(double angle)
```

The functions `buildFaceNormals` and `buildVertexFaceNeighbors` must be called first. Uses a slower but more precise method to set the vertex normals of the mesh to create a smooth-looking surface except for at angles exceeding `angle`.

```
void save(const std::string & filename, int outputMaterials = 0)
```

Saves the mesh to file `filename`, in `obj` format, optionally writing the mesh materials to `filename.mtl`.

class `ObjMeshRender` Renders an `ObjMesh` object via OpenGL.

```
ObjMeshRender(ObjMesh * mesh)
```

Initializes the object based upon the mesh `mesh`.

```
void render(int geometryMode, int renderMode)
```

```
unsigned int createDisplayList(int geometryMode, int renderMode)
```

Renders the object, or returns the integer identifier of an OpenGL display list to render the object. `geometryMode` is a bit field of options, controlling whether to render faces, edges, vertices, and/or normals. `renderMode` is a bit field of options to control rendering settings such as whether to use flat or smooth shading, whether to render textures, and/or whether to use colors or to set `glMaterial` material properties. See `ObjMeshRender.h` for more detailed info.

```
void renderSpecifiedVertices(int * specifiedVertices, int numSpecifiedVertices)
```

Renders the `numSpecifiedVertices` vertices whose indices are specified in the array `specifiedVertices`.

```
void renderGroupEdges(char * groupName)
```

Renders all the edges in the group of name `groupName` in the mesh.

```
void loadTextures(int textureMode)
```

Loads any textures specified in the material file of the mesh used to construct the `ObjMesh` mesh. `textureMode` specifies the color blend mode and mip-mapping settings for the textures; see `ObjMeshRender::Texture::loadTexture`.

```
static unsigned char * loadPPM(std::string filename, int * width, int * height)
```

Loads a binary-format RGB `.ppm` file (type P6), writes the dimensions to `*width` and `*height`, and returns a pointer to a newly-allocated (new) array of size `3 · width · height` of the components of the pixels of the image. The order of pixels in the output is bottom row to top row. Returns `NULL` upon failure.

class `ObjMeshRender::Texture` Stores the information for an OpenGL texture.

```
Texture()
```

Initializes the object to store no texture.

```
~Texture()
```

If the object stores a texture, calls `glDeleteTextures` on it.

```
void loadTexture(std::string fullPath, int textureMode)
```

Loads the texture stored in the binary-format `.ppm` file at `fullPath` (path from location of executable to texture file) and writes it to an OpenGL texture. `textureMode`, a bit field of options `OBJMESHRENDER_GL_REPLACE` or `OBJMESHRENDER_GL_MODULATE` for blend mode and `OBJMESHRENDER_GL_NOMIPMAP` or `OBJMESHRENDER_GL_USEMIPMAP` for whether to use mip-mapping, indicates the settings for this texture.

`bool hasTexture()`

Returns whether this object has loaded a texture.

`unsigned int texture()`

Returns OpenGL's identifier for the texture this object holds. Asserts that a texture has been loaded.

`int textureMode()`

Returns the texture mode as passed to this object in `loadTexture`.

2.18 performanceCounter

class PerformanceCounter Used to measure the execution time of a block of code. Construct the timer before a block of code to begin timing, stop it explicitly after the block executes, and then read the elapsed time at roughly microsecond accuracy.

`PerformanceCounter()`

Starts the counter at the current system time. If needed, the start time can be explicitly reset later (see below).

`void StartCounter()`

Restarts the counter at the current system time.

`void StopCounter()`

Stops the counter at the current system time.

`double GetElapsedTime()`

Returns the elapsed time between when the timer was last started and when it was last stopped. Note that one cannot use a single timer to measure the total running time of two blocks of code by starting and stopping it twice.

2.19 polarDecomposition

Provides tools to calculate the polar decomposition and the gradient of the polar decomposition of a 3×3 matrix. The code in the `PolarDecomposition` class is an adapted version of the code published by Graphics Gems IV, Ken Shoemake, 1993, "Polar Decomposition of 3×3 matrix in 4×4 , $M = QS$.", available at: <http://tog.acm.org/GraphicsGems/>. The website states that "Using the code is permitted in any program, product, or library, non-commercial or commercial." For details, see the header to `polarDecomposition.h`.

class PolarDecomposition

`static double Compute(const double * M,
double * Q, double * S, double tol = 1.0e-6)`

Calculates the polar decomposition of input matrix M into matrices Q and S (within an accuracy threshold `tol`) such that $M = QS$, Q is a 3×3 orthogonal matrix, and S is a 3×3 symmetric matrix. All matrices are encoded as arrays in row-major format, and Q and S are assumed to be already allocated.

class PolarDecompositionGradient

`static void Compute(const double * M, const double * Q,
const double * S, const double * MDot, double * omega,`

```
double * QDot, double * SDot, const double * MDotDot=NULL,
double * omegaDot=NULL, double * QDotDot = NULL)
```

Takes as input matrices M , Q , S (such that $M = QS$ is the polar decomposition), 3×3 derivative of M $MDot$, and optionally the 3×3 second derivative $MDotDot$. Calculates 3×3 derivatives $QDot$ and $SDot$, and 3-vector of rotational velocity ω , along with second derivatives $QDotDot$ and ωDot if $MDotDot$ is provided and the output pointers are not `NULL`. Again, all matrices are stored as row-major arrays, and all non-optional output matrix arrays must be pre-allocated.

2.20 sceneObject

Allows for easy rendering of `.obj` meshes, either in their original state or under user-specified deformations. File-loading and mesh-rendering functionality is provided by the `obj` library. All indices taken or returned by the classes below are zero-indexed.

class SceneObject Stores the model from an `.obj` file and provides tools to render the entire model or its constituent features.

```
SceneObject(char * filename)
```

Constructs the object from the `.obj` file at `filename`. Calls `exit` upon failure.

```
virtual void Render()
```

Renders the entire model with an `ObjMeshRender` object.

```
virtual void RenderVertices()
```

```
virtual void RenderEdges()
```

```
virtual void RenderNormals()
```

Render the specified components of the model using `ObjMeshRender`.

```
virtual void RenderVertex(int vertex)
```

Renders the vertex at index `vertex` in the mesh.

```
void HighlightVertex(int i)
```

Renders vertex `i` with a large green dot.

```
void BuildDisplayList()
```

Builds a display list for the model to speed up rendering. If a list has been built, it is automatically used by the `Render` function above.

```
void PurgeDisplayList()
```

Deletes the display list, if there currently is one.

```
ObjMesh * GetMesh()
```

Returns a pointer to the underlying `ObjMesh` object from the `obj` library, for more direct access to the mesh.

```
void ComputeMeshRadius(Vec3d & centroid, double * radius)
```

Writes to `*radius` the smallest radius of a sphere centered at `centroid` that contains the entire mesh.

```
void ComputeMeshGeometricParameters(Vec3d * centroid, double * radius)
```

Writes the coordinates of the center of the mesh (the average of its vertices) to `*centroid`, and the smallest radius of a sphere around this center which contains the entire mesh to `*radius`.

```
void ExportMeshGeometry(int * numVertices, double ** vertices,
int * numTriangles, int ** triangles)
```

Writes to `*numVertices` and `*numTriangles` the number of vertices and triangles in the mesh, and writes to `*vertices` and `*triangles` pointers to newly-allocated (`malloc`) arrays of the three coordinates of each vertex and the three vertex indices of each triangle, respectively.

```
virtual int GetClosestVertex(Vec3d & queryPos, double * distance=NULL,
    double * auxVertexBuffer = NULL)
```

Returns the index of the mesh vertex closest to the specified query location. The `auxVertexBuffer` option is only need for derived classes; you may pass `NULL` when using `GetClosestVertex` with `SceneObject`. To find the nearest vertex in a mesh that has been deformed, you need to use the `SceneObjectDeformable` subclass.

```
int SetUpTextures(LightingModulationType lightingModulation, MipmapType mipmap)
```

Initializes any textures specified by the `.obj` file, and enables texture rendering. Calling this function is mandatory for texture mapping to work. Must be called after OpenGL has been initialized. The `lightingModulation` parameter selects either `MODULATE` or `REPLACE` for the texture blend mode, and the `mipmap` parameter selects either `USEMIPMAP` or `NOMIPMAP` for whether to use mip-mapping. Returns 0 iff successful.

```
bool hasTextures()
```

After `SetUpTextures` has been called, returns whether the model has textures.

```
void EnableTextures()
```

```
void DisableTextures()
```

Enables or disables textures for rendering. Does not check whether textures have been set up.

```
void BuildFaceNormals()
```

Calculates the normals of the mesh faces.

```
void BuildNeighboringStructure()
```

Must be called before any of the three vertex normal methods below. Creates a data structure to optimize finding neighboring faces of mesh vertices.

```
void BuildVertexNormals(double thresholdAngle=85.0)
```

Must be called after `BuildNeighboringStructure`. Calculates smooth normals for each vertex by averaging the face normals of each incident triangle. The `thresholdAngle` parameter preserves sharp angles in the mesh; for a given vertex, if a triangle incident to it has angle greater than `thresholdAngle` between its normal and the normal of the first triangle incident to the vertex, then this triangle does not contribute to or use the smooth normal at the vertex.

```
void BuildNormals(double thresholdAngle=85.0)
```

Must be called after `BuildNeighboringStructure`. Builds face and vertex normals for the mesh.

```
void BuildNormalsFancy(double thresholdAngle=85.0)
```

Must be called after `BuildNeighboringStructure`. Sets mesh normals, handling sharp edges in a more precise way.

```
void ShowPointLabels(int k, int l)
```

Renders vertex indices alongside the vertices of indices `k` through `l`. These values are zero-indexed, but they are written as one-indexed.

```
void ShowPointLabels()
```

Renders all vertex indices.

class SceneObjectWithRestPosition : public SceneObject Adds to **SceneObject** a record of the rest position of the model, i.e. the initial position of each vertex.

SceneObjectWithRestPosition(char * filename)

Loads the .obj mesh at filename, and saves a record of the rest position of the mesh.

void GetVertexRestPositions(float * buffer)

void GetVertexRestPositions(double * buffer)

Writes the rest position coordinates of each vertex in the mesh to the pre-allocated array **buffer**, in either float or double format.

double * GetVertexRestPositions()

Provides direct access to the internal array of rest positions.

class SceneObjectDeformable : public SceneObjectWithRestPosition Adds the ability to deform the model to a given displacement from the rest position, so it may be rendered in the deformed position, and to reset the model to an undeformed state.

SceneObjectDeformable(char * filenameOBJ)

Initializes the model from the .obj file at filenameOBJ.

void GetSingleVertexRestPosition(int vertex, double * x, double * y, double * z)

void SetSingleVertexRestPosition(int vertex, double x, double y, double z)

Gets or sets the rest position of the vertex at index **vertex** in the mesh.

void GetSingleVertexPositionFromBuffer(int vertex,

double * x, double * y, double * z)

Returns the current model position of the vertex at index **vertex**.

void ResetDeformationToRest()

Resets all vertices to their rest positions.

void SetVertexDeformations(double * u)

void SetVertexDeformations(float * u)

For array **u** which specifies displacements for the coordinates of each vertex in the model, sets the model to be displaced by **u** from its rest position.

void AddVertexDeformations(double * u)

Adds displacements **u** of the coordinates of each model vertex to the current positions of the vertices.

virtual void SetLighting(Lighting * lighting)

Sets the scene lighting using the specified **Lighting** object.

2.21 sparseMatrix

Provides tools for efficiently storing and running calculations with matrices that have few non-zero elements. These matrices do not need to be square. Rows and columns are zero-indexed.

class SparseMatrixOutline Used to declare which positions will be non-zero in a **SparseMatrix**. This class cannot perform matrix arithmetic, but non-zero entries can be freely added. Then, the class is used to construct a **SparseMatrix** object, which performs arithmetic but whose zero positions cannot be changed. Contained in the same file as **SparseMatrix**.

SparseMatrixOutline(int n)

`SparseMatrixOutline(int n, double diagonal)`
`SparseMatrixOutline(int n, double * diagonal)`
 Sets the number of rows in the matrix, and optionally a starting value for all the diagonal entries or an array of starting values for the diagonals.

`SparseMatrixOutline(char * filename, int expand=1)`
 Constructs the outline from a file. For `expand` greater than 1, expands each element to a diagonal block of size `expand` by `expand`. This is useful, for example, with mass matrices.

`void AddEntry(int i, int j, double value=0.0)`
`void AddBlock3x3Entry(int i, int j, double * matrix3x3)`
`void AddBlockMatrix(int i, int j, const SparseMatrix * block, double scalarFactor=1.0)`
 Adds a value (or a matrix of values) starting at row `i`, column `j`. “Add” in this context means addition, not insertion; if the position in the matrix is already non-zero, this new value is added to the existing value. `scalarFactor` is multiplied with the block matrix values before they are added.

`double GetEntry(int i, int j)`
 Gets the matrix value at row `i`, column `j`, or zero if the position has not been set.

class SparseMatrix A matrix storage class optimized for the case of the matrix having few non-zero elements, i.e., a sparse matrix. The class uses row-based sparse storage. Each matrix row is stored as a list of non-zero column positions and the corresponding entries. The class can perform matrix arithmetic, as well as many other operations such as assigning submatrices or removing matrix rows and columns. It is constructed by first specifying the locations of non-zero entries using the `SparseMatrixOutline` class. Once this is finalized, one can initialize `SparseMatrix`, which writes the row elements into a linear array for fast subsequent access. The sparsity pattern cannot be changed by `SparseMatrix`.

`SparseMatrix(SparseMatrixOutline * sparseMatrixOutline)`
`SparseMatrix(const SparseMatrix & source)`
`SparseMatrix(char * filename)`
 Initializes the matrix values from a `SparseMatrixOutline`, another `SparseMatrix`, or a file.

`int Save(char * filename, int oneIndexed=0)`
 Saves the matrix to disk. Returns 1 on error, 0 otherwise.

`void SetEntry(int row, int j, double value)`
`void AddEntry(int row, int j, double value)`
`void GetEntry(int row, int j)`
 Sets, adds to, or gets the value in the `j`th *non-zero* element of row `row` of the matrix.

`int GetRowLength(int row)`
 Returns the number of non-zero elements in the row.

`int GetColumnIndex(int row, int j)`
 Returns the column index of the `j`th *non-zero* element in row `row`.

`SparseMatrix operator+(const SparseMatrix & mat2)`
`SparseMatrix operator-(const SparseMatrix & mat2)`
 Returns a `SparseMatrix` equal to the sum of this matrix and `SparseMatrix mat2`. These two matrices must have the same pattern of non-zero entries.

`void MultiplyMatrix(int numDenseRows, int numDenseColumns, const double * denseMatrix, double * result)`

Multiplies this `SparseMatrix` matrix with a dense matrix `denseMatrix` of dimensions `numDenseRows` \times `numDenseColumns`. The dense matrices are stored as column-major arrays of doubles, and the output matrix `result` is assumed to be previously allocated.

2.22 sparseSolver

Offers a common interface for classes which solve linear systems, and implements it for Conjugate Gradients, Pardiso and SPOOLES.

class LinearSolver Serves as an abstract base class for solvers of linear systems.

```
virtual int SolveLinearSystem(double * x, const double * rhs) = 0
```

Implementations solve the linear system $Ax = rhs$, where the matrix A has been specified earlier via some implementation-specific method.

class CGSolver : public LinearSolver Solves the linear system $Ax = b$ using Conjugate Gradients. The implementation closely follows the reference [She94]. The matrix A must be symmetric positive-definite. Here, x is represented by a dense array of doubles, and A is represented either by a `SparseMatrix` or a user-provided function that calculates Ax for an input x .

```
CGSolver(SparseMatrix * A)
```

```
CGSolver(int n, blackBoxProductType callBackFunction, void * data)
```

Initializes the solver from a `SparseMatrix` or a user-provided function as described above.

`blackBoxProductType` is a function that returns `void` and accepts `const void *` (pointer to matrix A), `const double *` (pointer to vector x), and `double *` (pointer to product Ax).

```
int SolveLinearSystemWithoutPreconditioner(double * x, const double * b,
double eps=1e-6, int maxIter=1000, int verbose=0)
```

Solves the equation $Ax = b$ for an input b and the current value of the matrix A . Output is given on the pre-allocated vector x , which should be initialized to an initial guess for the solution. The function terminates when the number of iterations exceeds the input `maxIter` or the error falls below the input `eps`; the return value is the number of iterations taken, multiplied by -1 if the system did not converge.

```
int SolveLinearSystemWithJacobiPreconditioner(double * x, const double * b,
double eps, int maxIter, int verbose=0)
```

Solves the equation $Ax = b$ using Jacobi preconditioning. Note that this will fail if the diagonal of A contains any zero entries.

```
virtual int SolveLinearSystem(double * x, const double * b)
```

Implements the virtual function from `LinearSolver` by calling `SolveLinearSystemWithJacobiPreconditioner` with the default parameters, `eps=1e-6`, `maxIter=1000`, `verbose=0`.

class PardisoSolver : public LinearSolver Solves the linear system $Ax = b$ using the PARDISO solver. Matrix A must be symmetric and invertible.

```
PardisoSolver(const SparseMatrix * A, int numThreads,
int positiveDefinite=0, int directIterative=0,
int verbose=0)
```

Initializes the solver based upon symmetric matrix A . The matrix used for solving the linear system may be changed later, but it must have the same topology as A . `numThreads` threads are used for the calculation, and the last three parameters declare whether the matrix is positive definite, whether to use the direct iterative solving method, and whether all the member functions are verbose.

`int ComputeCholeskyDecomposition(const SparseMatrix * A)`
 Performs complete Cholesky factorization on `A`, setting the current value of `A` as the matrix to be used for the next system solve. Returns the error status of the `pardiso` function.

`virtual int SolveLinearSystem(double * x, const double * rhs)`
 Solves the linear system $Ax = rhs$ for the value of the matrix `A` that was last given to the constructor or to the `ComputeCholeskyDecomposition` function *at the time* it was passed to the function. The function fails with return value 101 if it is called after the `directIterative` option was enabled in the constructor; otherwise, returns the error status of the `pardiso` function.

`int SolveLinearSystemDirectIterative(const SparseMatrix * A, double * x, const double * rhs)`
 Solves the linear system $Ax = rhs$ with the direct iterative method, using the input matrix `A`. Returns the error status of the `pardiso` function.

class SPOOLESSolver : public LinearSolver Solves the linear system $Ax = b$ using the public domain SPOOLES solver. Matrix `A` must be symmetric and invertible.

`SPOOLESSolver(const SparseMatrix * A, int verbose=0)`
 Initializes the solver with the symmetric matrix `A`, so that the value of `A` *at the time of construction* will be used for all calls of `SolveLinearSystem`. Setting `verbose` to 1 prints feedback during the constructor, and to 2 additionally prints feedback during the solve. Throws an exception upon failure.

`virtual int SolveLinearSystem(double * x, const double * rhs)`
 Solves the system $Ax = rhs$ using the matrix `A` from the constructor. Returns 0 iff successful.

class SPOOLESSolverMT : public LinearSolver Solves the linear system $Ax = b$ using the public domain multithreaded SPOOLES solver. Matrix `A` must be symmetric and invertible.

`SPOOLESSolverMT(const SparseMatrix * A, int numThreads, int verbose=0)`
 Initializes the solver with the matrix `A`, so that the value of `A` *at the time of construction* will be used for all calls of `SolveLinearSystem`, and sets the number of threads `numThreads` to use during the solve. Throws an exception upon failure. Setting `verbose` to 1 prints feedback during the constructor, and to 2 additionally prints feedback during the solve. Further diagnostic information is written to a file called `SPOOLES.message`.

`virtual int SolveLinearSystem(double * x, const double * rhs)`
 Solves the linear system $Ax = rhs$ for the matrix `A` from the constructor, using `numThreads` threads as specified in the constructor. Returns the exit code of the solver (1 if successful).

2.23 stvk

Implements the Saint-Venant Kirchhoff (StVK) deformable model, providing classes to evaluate energy, internal forces, stiffness matrix, and Hessian tensor for a given volumetric mesh and vertex displacements. Optionally, the energy, internal forces, and stiffness matrix calculations can be multi-threaded (using the `pthreads` library).

class StVKElementABCD Serves as a base class for classes that generate the St. Venant-Kirchhoff ABCD coefficients (see [Bar07]) for tetrahedral or cubic mesh elements.

`virtual void AllocateElementIterator(void ** elementIterator)`
 Allocates and returns a pointer to a data structure used to hold information about a single mesh element for use in the ABCD calculations. The base class allocates a dummy structure, but implementations allocate structures of varying types, so void pointers are necessary.

```
virtual void ReleaseElementIterator(void * elementIterator)
De-allocates the data structure produced by AllocateElementIterator.
```

```
virtual void PrepareElement(int e1, void * elementIterator)
Writes to the pre-allocated iterator elementIterator the appropriate pre-calculated information for the element at index e1 in the volumetric mesh.
```

```
virtual Mat3d A(void * elementIterator, int i, int j) = 0
virtual double B(void * elementIterator, int i, int j) = 0
virtual Vec3d C(void * elementIterator, int i, int j, int k) = 0
virtual double D(void * elementIterator, int i, int j, int k, int l) = 0
```

Calculate the A, B, C, or D values for the element whose iterator `elementIterator` has been allocated and prepared by the functions above, and for the chosen vertices `i`, `j`, `k`, and `l` of the element. Each integer ranges from 0 to (number of vertices per element)−1.

class StVKCubeABCD : public StVKElementABCD Implements the base ABCD coefficients class for cube-shaped elements.

```
StVKCubeABCD(double cubeSize)
```

Initializes the class given the undeformed edge length `cubeSize` of the cubes in the cubic mesh. No other information about the mesh is needed to be able to compute the ABCD values, which are identical for each element in the mesh, though the ABCD functions still accept the `elementIterator` argument for consistency with the base class.

class StVKTetABCD : public StVKElementABCD Implements the base ABCD coefficients class for tetrahedral elements.

```
StVKTetABCD(TetMesh * tetMesh)
```

Reads the mesh `tetMesh` to extract element-specific information used during the ABCD calculations. This information is distributed to the ABCD calculation functions via the `elementIterator` object produced by the class.

class StVKTetHighMemoryABCD : public StVKElementABCD Implements the base ABCD coefficients class for tetrahedral elements, optimizing ABCD access speed at the expense of greater memory usage, by explicitly precalculating all ABCD values at the time of construction.

```
StVKTetHighMemoryABCD(TetMesh * tetMesh)
```

Reads the mesh `tetMesh` to extract element-specific information, and uses this information to calculate all the ABCD coefficients for each element. As a result, the ABCD calculation function implementations of this subclass consist of a single array lookup.

class StVKElementABCDDoader Provides a single function to create an appropriate `StVKElementABCD` object for a given volumetric mesh.

```
static StVKElementABCD * load(VolumetricMesh * volumetricMesh,
unsigned int loadingFlag=0)
```

Given the mesh `volumetricMesh`, allocates (`new`) and returns the pointer to an instance of a child class of `StVKElementABCD` which can produce the appropriate ABCD values for the mesh. Setting `loadingFlag` to 1 will instruct the loader to create `StVKTetHighMemoryABCD` instances for tet meshes. Returns NULL or calls `exit` with non-zero status upon failure.

class StVKInternalForces Calculates internal forces and energy for the StVK material.

**StVKInternalForces(VolumetricMesh * volumetricMesh,
StVKElementABCD * precomputedABCDIntegrals, bool addGravity=false, double g=9.81)**
Initializes the class to use mesh **volumetricMesh** and the precalculated information **precomputedABCDIntegrals** for that mesh (usually generated by **StVKElementABCDDoader** given the mesh).

virtual double ComputeEnergy(double * vertexDisplacements)
Given the array **vertexDisplacements** of vertex displacements from rest position, returns the non-linear elastic strain energy of the mesh.

virtual void ComputeForces(double * vertexDisplacements, double * internalForces)
Given the array **vertexDisplacements** of vertex displacements, writes the resulting vertex forces to the pre-allocated array **internalForces**

void SetGravity(bool addGravity) Sets whether to apply the gravitational force **g** that was specified in the constructor.

VolumetricMesh * GetVolumetricMesh()
StVKElementABCD * GetPrecomputedIntegrals()
Returns the volumetric mesh or precomputed integrals used by this class.

class StVKInternalForcesMT : public StVKInternalForces Extends the main internal forces class to multithread the internal force and energy calculations using **pthreads**.

**StVKInternalForcesMT(VolumetricMesh * volumetricMesh,
StVKElementABCD * precomputedABCDIntegrals, bool addGravity,
double g, int numThreads)**
Constructs the class as per **StVKInternalForces**, and sets the number of threads to use for the multi-threaded calculations.

virtual double ComputeEnergy(double * vertexDisplacements)
Calculates strain energy using **numThreads** threads, as set in the constructor.

virtual void ComputeForces(double * vertexDisplacements, double * internalForces)
Writes internal forces to the pre-allocated array **internalForces**, using **numThreads** threads to calculate the forces, as set in the constructor.

class StVKStiffnessMatrix Calculates stiffness matrices and the topology of stiffness matrices for the StVK material.

StVKStiffnessMatrix(StVKInternalForces * stVKInternalForces)
Initializes the stiffness matrix computation, by providing a previously initialized **StVKInternalForces** class.

void GetStiffnessMatrixTopology(SparseMatrix ** stiffnessMatrixTopology)
Writes to ***stiffnessMatrixTopology** a newly allocated (using **new**) zero matrix containing the pattern of non-zero entries of the stiffness matrix.

**virtual void ComputeStiffnessMatrix(double * vertexDisplacements,
SparseMatrix * sparseMatrix)**
Writes to **sparseMatrix** the stiffness matrix of the material under the deformation specified by the array **vertexDisplacements**. Matrix **sparseMatrix** must have the pattern of non-zero entries produced by **GetStiffnessMatrixTopology**.

```
VolumetricMesh * GetVolumetricMesh()
StVKElementABCD * GetPrecomputedIntegrals()
Return the volumetric mesh or precomputed integrals being used by this class.
```

class StVKStiffnessMatrixMT : public StVKStiffnessMatrix Extends the main stiffness matrix class to multithread the stiffness matrix calculation process using `pthreads`.

```
StVKStiffnessMatrixMT(StVKInternalForces * stVKInternalForces,
    SparseMatrix ** sparseMatrix, int numThreads)
Constructs the class as per StVKStiffnessMatrix, and sets the number numThreads of threads to use for stiffness matrix calculation.
```

```
virtual void ComputeStiffnessMatrix(double * vertexDisplacements,
    SparseMatrix * sparseMatrix)
Computes the stiffness matrix using numThreads threads, as set in the constructor.
```

2.24 volumetricMesh

Provides tools to load and store tetrahedral or cubic volumetric meshes, set and access their material properties, and perform various calculations with them.

class VolumetricMesh Serves as an abstract base class for volumetric meshes, holding the vertices and elements of the mesh, identifying regions of the mesh with common material properties and listing those properties, and providing functionality common to meshes with different element geometries. Derived into `TetMesh` and `CubicMesh` to store tet meshes and meshes consisting of cubes (voxels). `VolumetricMesh` is abstract; you cannot initialize it directly, but must be initialized via one of its two subclasses. Note: The loading procedure is made easier by the `VolumetricMeshLoader` class.

```
static elementType getElementType(char * filename)

virtual elementType getElementType() = 0
Returns the element type of the mesh in file filename or of the current mesh. The result is “TET” for TetMesh objects and “CUBIC” for CubicMesh objects. If filename cannot be read, the type is set to INVALID.

int getNumElementVertices()
Returns the number of vertices per mesh element.

virtual int save(char * filename) = 0
Saves the mesh, including materials, sets and regions to a filename. Format is .veg, which may be later loaded by VolumetricMeshLoader.

int getNumVertices()
int getNumElements()
int getNumMaterials()
int getNumSets()
int getNumRegions()
Returns various mesh objects.

Vec3d * getVertex(int i)
Material * getMaterial(int i)
Set * getSet(int i)
Region * getRegion(int i)
```

Returns a pointer to the specified type of object, given its index in the list of that type of object in the `VolumetricMesh`.

```
void setMaterial(int i, Material & material)
```

Sets the material at index `i` in the array of materials to `material`, and copies these material properties to each mesh element whose containing region lists index `i` as the material which defines its material properties.

```
virtual double getElementVolume(int el) = 0
```

Implementations return the volume of the element at index `el`.

```
virtual bool containsVertex(int element, Vec3d pos) = 0
```

Returns whether the element at index `i` contains the vector `pos`.

```
int getClosestElement(Vec3d pos)
```

```
int getClosestVertex(Vec3d pos)
```

```
int getContainingElement(Vec3d pos)
```

Returns the index of the nearest element, nearest vertex, or containing element of `pos` in the undeformed mesh. For the third function, -1 is returned if no containing element is found.

```
void exportMeshGeometry(int * numVertices, double ** vertices,
    int * numElements, int * numElementVertices, int ** elements)
```

Exports the geometric information of the mesh to memory arrays. The number of vertices, number of elements, and number of vertices per element are written to `*numVertices`, `*numElements`, and `*numElementVertices`. Array `*vertices` is set to point to a newly-allocated (`malloc`) array of the coordinates of all the vectors, and `*elements` is set to point to a newly-allocated (`malloc`) array of the indices of the vertices constituting each mesh element.

```
int generateInterpolationWeights(int numTargetLocations,
    double * targetLocations, int ** vertices, double ** weights,
    double zeroThreshold = -1.0, std::vector<int> * closestElementList = NULL,
    int verbose=0)
```

Generates interpolation information for embedding a (typically higher-resolution) rendering mesh within a deformable volumetric mesh. Given `numTargetLocations` points (rendering mesh vertices), with coordinates specified in the array `targetLocations`, the function determines which mesh element is closest to or contains each point. It then expresses each point's position as a weighted average of the coordinates of that element's vertices, such that the weights sum to one (standard tet barycentric coordinates). Note that weights may be negative if the point is outside the volumetric mesh. Output arrays `*vertices` and `*weights` are allocated (`malloc`) inside the function. For each target location, the `*vertices` array gives the integer indices of `VolumetricMesh` vertices of the mesh element containing the target. The array `*weights` gives the barycentric weights. The length of both arrays is `numElementVertices * numTargetLocations`. If `zeroThreshold > 0`, then the points that are further than `zeroThreshold` away from any volumetric mesh vertex, are assigned weights of 0. If `closestElementList` is not NULL, the closest elements will be returned in the vector `closestElementList`. The function returns the number of target points which do not lie inside any element.

```
static int loadInterpolationWeights(char * filename,
    int numTargetLocations, int numElementVertices,
    int ** vertices, double ** weights)
```

```
static int saveInterpolationWeights(char * filename,
    int numTargetLocations, int numElementVertices,
    int * vertices, double * weights)
```

Reads or writes the arrays `vertices` and `weights` of interpolation information from or to `filename`. Returns 0 iff successful.

```
static void interpolate(double * u, double * uTarget, int numTargetLocations,
    int numElementVertices, int * vertices, double * weights)
```

Interpolates volumetric mesh vertex deformations `u` to an embedded (triangle, volumetric, etc.) mesh, using barycentric linear interpolation. The computed embedded mesh vertex deformations are stored into the pre-allocated array `uTarget`. The number of the embedded mesh vertices must be provided in `numTargetLocations`. The number of vertices in an element of the volumetric mesh must be provided in `numElementVertices`. Input parameters `vertices` and `weights` must have been generated by a previous call to `generateInterpolationWeights`. Note that `u` can be set to either the deformed positions or the displacements of the volumetric mesh vertices to obtain in `uTarget` either the deformed positions or the displacements of the target points.

```
VolumetricMesh(int numVertices, double * vertices,
    int numElements, int numElementVertices, int * elements,
    double E=1E6, double nu=0.45, double density=1000)
```

This non-public constructor is called by derived classes' constructors. It constructs a mesh with `numVertices` vertices whose coordinates are given in the size $3 \cdot \text{numVertices}$ array `vertices` and with `numElements` elements (each with `numElementVertices` vertices), whose constituent vertices are specified by the `numElements`·`numElementVertices` size array of vertex indices `elements`.

class VolumetricMesh::Material A nested subclass of `VolumetricMesh`. Stores material properties. Constructed by the `VolumetricMesh` constructor. This class is abstract and stores common functionality for storing materials. Derived into classes `ENuMaterial` and `MooneyRivlinMaterial`.

class VolumetricMesh::ENuMaterial The `ENuMaterial` class stores material properties for any material that is parameterized by Young's modulus (E) and Poisson's ratio (ν). Most materials in Vega falls into this category, e.g., co-rotational linear FEM, StVK, invertible StVK, invertible neo-Hookean, etc. Constructed by the `VolumetricMesh` constructor.

```
double getE()
double getNu()
double getDensity()
double getLambda()
double getMu()
```

Return the E , ν , or density property of the material, as well as the Lamé coefficients λ and μ (computed from E, ν).

```
char * getName()
```

Returns the name string of the material. Note that the return value is a pointer to memory within the `VolumetricMesh::Material` object.

```
void setE(double E)
void setNu(double nu)
void setDensity(double density)
void setName(char * name)
```

Set the relevant material property or the name of the material.

class VolumetricMesh::MooneyRivlinMaterial Stores a Mooney-Rivlin material, parameterized by $\mu_{01}, \mu_{10}, \nu_1$. Sister class to `ENuMaterial`. Constructed by the `VolumetricMesh` constructor.

class VolumetricMesh::Set A nested subclass of `VolumetricMesh`. Stores a **set** of integer indices. Constructed by the `VolumetricMesh` constructor. Typically, it is used to store indices of elements that use the same material.

```
void addElement(int element)
```

Adds `element` to the set.

```
void getElements(std::set<int> & elements)
```

Stores a copy of the set elements to the set `elements`.

class VolumetricMesh::Region A region is a subpart of the mesh consisting of common material properties. A nested subclass of `VolumetricMesh`. Constructed by the `VolumetricMesh` constructor.

```
int getMaterialIndex()
```

Returns the index of the `VolumetricMesh` material which corresponds to the mesh elements in this region.

```
int getSetIndex()
```

Returns the index of the `VolumetricMesh` set which corresponds to the mesh elements in this region.

class CubicMesh : public VolumetricMesh Stores a cubic mesh in which each mesh element is a cube. All cubes have the same size, and must originate from an axis-aligned grid of uniform size. Such meshes can be easily created from input triangular geometry using the `LargeModalDeformationFactory` utility.

```
CubicMesh(int numVertices, double * vertices, int numElements,
           int * elements, double E=1E6, double nu=0.45, double density=1000)
```

Constructs a cubic mesh of `numVertices` vertices with coordinates in the array `vertices`, organized into `numElements` cubes by the array `elements` which contains eight vertex indices for each cube. Assumes that the input actually specifies cube-shaped elements that come from an axis-aligned grid. Upon failure, throws an exception.

`CubicMesh(char * filename)` Loads the mesh from the file `filename` (.veg format). Throws an exception on failure.

```
static CubicMesh * CreateFromUniformGrid(int resolution,
                                          int numVoxels, int * voxels, double E=1E6, double nu=0.45, double density=1000)
```

Constructs a mesh that is a subset of a uniform voxel grid of dimensions `resolution`³. The `numVoxels` mesh elements (voxels) are specified by their integer coordinates in the integer array `voxels` (length `3*numVoxels`). The function returns a pointer to a newly-allocated (new) `CubicMesh` with this geometry.

```
double cubeSize()
```

Returns the grid size (=length of an edge) of the cubes in the mesh.

class TetMesh : public VolumetricMesh Stores a tetrahedral mesh. The tetrahedral mesh elements may be arbitrarily shaped. To create tetrahedral meshes, you can use a mesh generation package such as TetGen [Han11] or Stellar [KS09, Kli08].

`TetMesh(char * filename)` Loads the mesh from the file `filename` (.veg format). Throws an exception on failure.

class VolumetricMeshLoader Provides a single function to load cubic or tetrahedral meshes. It is not necessary to know the mesh element type in advance.

`static VolumetricMesh * load(char * filename)` Loads the mesh in file `filename` and returns the pointer to a newly-allocated (new) `CubicMesh` instance or `TetMesh`. Returns `NULL` or throws an exception upon failure, depending on which subfunction experiences the failure.

References

- [Bar07] Jernej Barbič. *Real-time Reduced Large-Deformation Models and Distributed Contact for Computer Graphics and Haptics*. PhD thesis, Carnegie Mellon University, August 2007.
- [Bar12] Jernej Barbič. Exact Corotational Linear FEM Stiffness Matrix. Technical report, University of Southern California, 2012.
- [Bow09] Allan Bower. *Applied Mechanics of Solids*. CRC Press, 2009.
- [BW98] David Baraff and Andrew P. Witkin. Large Steps in Cloth Simulation. In *Proc. of ACM SIGGRAPH 98*, pages 43–54, July 1998.
- [BW08] Javier Bonet and Richard D. Wood. *Nonlinear Continuum Mechanics for Finite Element Analysis, 2nd Ed.* Cambridge University Press, 2008.
- [CPSS10] Isaac Chao, Ulrich Pinkall, Patrick Sanan, and Peter Schröder. A Simple Geometric Model for Elastic Deformations. *ACM Transactions on Graphics*, 29(3):38:1–38:6, 2010.
- [Han11] Hang Si. TetGen: A Quality Tetrahedral Mesh Generator and a 3D Delaunay Triangulator, 2011. <http://tetgen.org/>.
- [ITF04] G. Irving, J. Teran, and R. Fedkiw. Invertible Finite Elements for Robust Simulation of Large Deformation. In *Symp. on Computer Animation (SCA)*, pages 131–140, 2004.
- [Kli08] Matthew Klingner. *Improving Tetrahedral Meshes*. PhD thesis, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 2008.
- [KS09] Matthew Klingner and Jonathan Richard Shewchuk. Stellar: A tetrahedral mesh improvement program, 2009. <http://www.cs.berkeley.edu/~jrs/stellar/>.
- [MG04] M. Müller and M. Gross. Interactive Virtual Materials. In *Proc. of Graphics Interface 2004*, pages 239–246, 2004.
- [PAR] PARDISO: Parallel Direct Sparse Solver Interface. Pardiso project, www.pardiso-project.org and Intel MKL, software.intel.com/en-us/articles/intel-mkl.
- [Sha90] Ahmed A. Shabana. *Theory of Vibration, Volume II: Discrete and Continuous Systems*. Springer-Verlag, 1990.
- [She94] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain, 1994.
- [SPO] SPOOLES: SParse Object Oriented Linear Equations Solver. Boeing Phantom Works. www.netlib.org/linalg/spooles/spooles.2.2.html.
- [SSB12] Fun Shing Sin, Daniel Schroeder, and Jernej Barbič. Vega: Nonlinear FEM Deformable Object Simulator. *Computer Graphics Forum*, 2012. accepted, to appear.
- [TSIF05] Joseph Teran, Eftychios Sifakis, Geoffrey Irving, and Ronald Fedkiw. Robust Quasistatic Finite Elements and Flesh Simulation. In *Symp. on Computer Animation (SCA)*, pages 181–190, 2005.
- [Wri02] Peter Wriggers. *Computational Contact Mechanics*. John Wiley & Sons, Ltd., 2002.