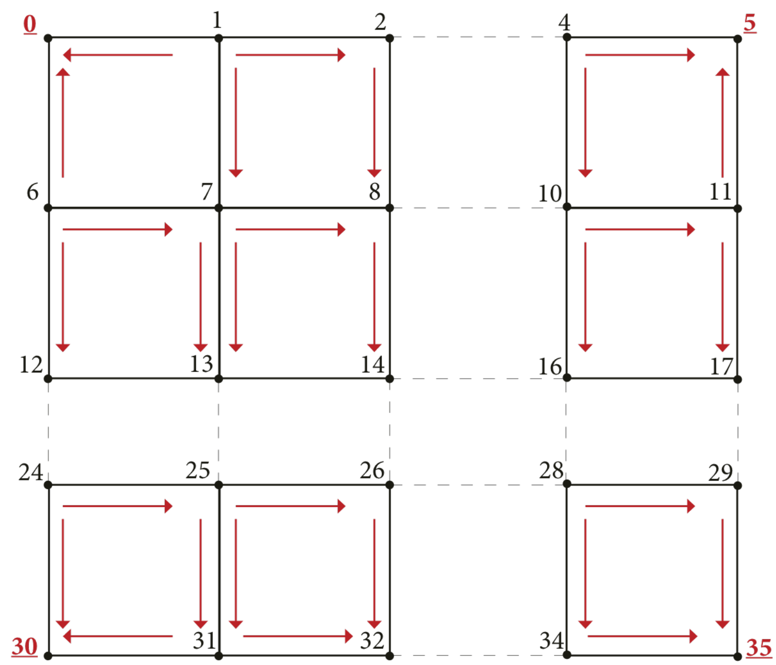
Force Density Method API:
Form Finding and Surface Structures

Alexandros Haridis

MIT Department of Architecture
Spring 2015



Force Density Method

The *Force Density Method (FDM)* is presented in detail in Klaus Linkwitz (2014), "Force Density Method," Ch.6 in S. Adriaenssens, P. Block, D. Veenendaal & C. Williams. *Shell Structures for Architecture: Form Finding and Optimization*. This application solves the example grid depicted in *p.69* which is a grid made with 36 nodes, 25 faces and thus, from its Euler characteristic, 60 edges.

This implementation consists of two primary computational abstractions. The first is called the *BranchNodeGraph* API which is a data structure that provides a framework for building the Branch-Node Matrix **C** and its sub-matrices $\mathbf{C_N}$ and $\mathbf{C_F}$ for new and fixed points, respectively. Internally, it keeps information about the topology (connectivity) of a given structural system, which is represented as a set of *indices* (nodes) and a set of *directed links* (branches/ connections) between these indices. For instance, the branch that links the indices 1 and 7 is represented internally with an object *Branch* that represents branch **17**. The directions of the links depend on whether one of the two ends is a fixed point or not (see above image). Here, the grid is considered to have four fixed corner points. The *Branch Node Matrix* **C** is constructed by linking the nodes first in the horizontal direction and then in the vertical direction. Following, is an assembled branch-node matrix **C** where rows correspond to unique edges, and columns to unique vertices. The left side corresponds to the new nodes $\mathbf{C_N}$ and the right side to the fixed nodes $\mathbf{C_F}$.

-1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0

1 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 1 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 1 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0

....

0 0 0 0 0 0 0 0 0 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0

0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 -1 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 -1

The second computational abstraction is the *ForceDensityMethod* API which is a data structure that provides standard FDM operations on the matrix **C** and its sub-matrices, and calculations of the node coordinates **x**, **y**, **z**, the vector **F** for tension forces, and a measure of structural performance given by the summation $\Sigma_i F_i L_i$. The data structure is internally represented by a *BranchNodeGraph* that stores the geometry of the structural grid (node coordinates), including information about boundary conditions. The following calculations are done within the *ForceDenistyMethod* data structure. Force densities are distributed uniformly across the grid (equal to 1). For each set of prescribed force densities I get exactly one equilibrium state solving the following linear system:

$$\boldsymbol{x_N} = \boldsymbol{D_N^{-1}}(\boldsymbol{p_x} - \boldsymbol{D_F x_F})$$

$$\boldsymbol{y_N} = \boldsymbol{D_N^{-1}}(\boldsymbol{p_y} - \boldsymbol{D_F y_F}) \qquad (1)$$

$$\boldsymbol{z_N} = \boldsymbol{D_N^{-1}}(\boldsymbol{p_z} - \boldsymbol{D_F z_F})$$

were

$$\boldsymbol{D_N} = \boldsymbol{C_N^T Q C_N} \text{ and } \boldsymbol{D_F} = \boldsymbol{C_N^T Q C_F} \qquad (2)$$

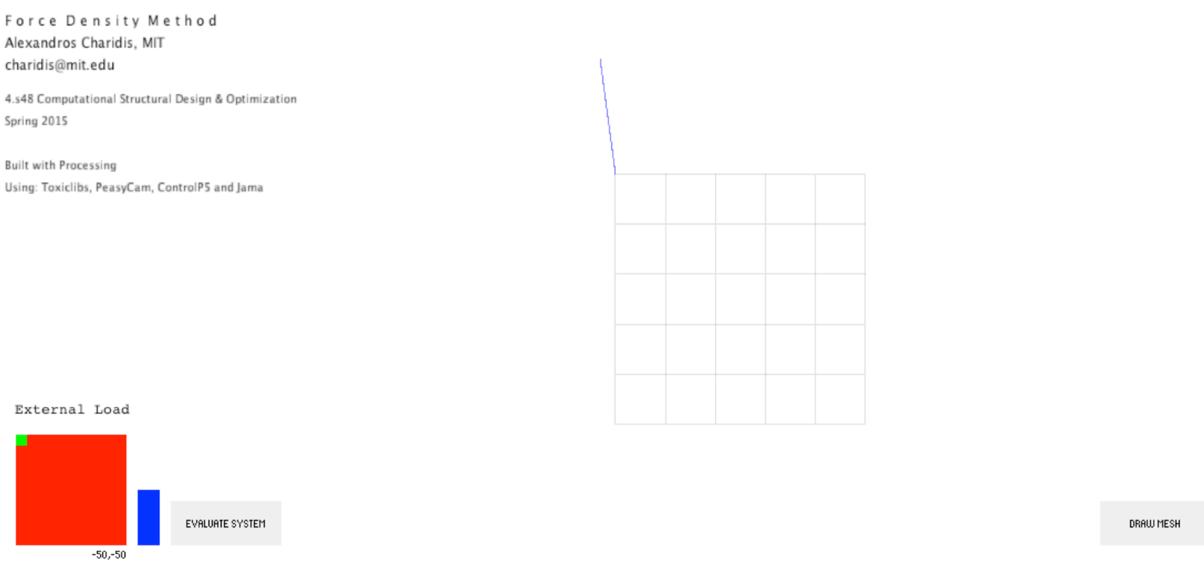
p *is the vector of load components* which I assume to be the force due to *gravity*, it is represented as the vector $[\mathbf{0}\ \mathbf{0} - \boldsymbol{g}]^{\boldsymbol{T}}$ were $\boldsymbol{g}$ is a scalar, and I control it (in all three axis) externally from the graphical user interface I built for it (see below). The system of equations (1) is solved as a linear system of the form **Ax** = **b**. The vector containing the lengths of each branch is assembled as per the following expression:

$$\boldsymbol{L} = \ (\boldsymbol{U}^2 + \boldsymbol{V}^2 + \boldsymbol{W}^2)^{1/2} \ (3)$$

Matrices **U**, **V**, and **W** are the diagonal matrices of the vectors:

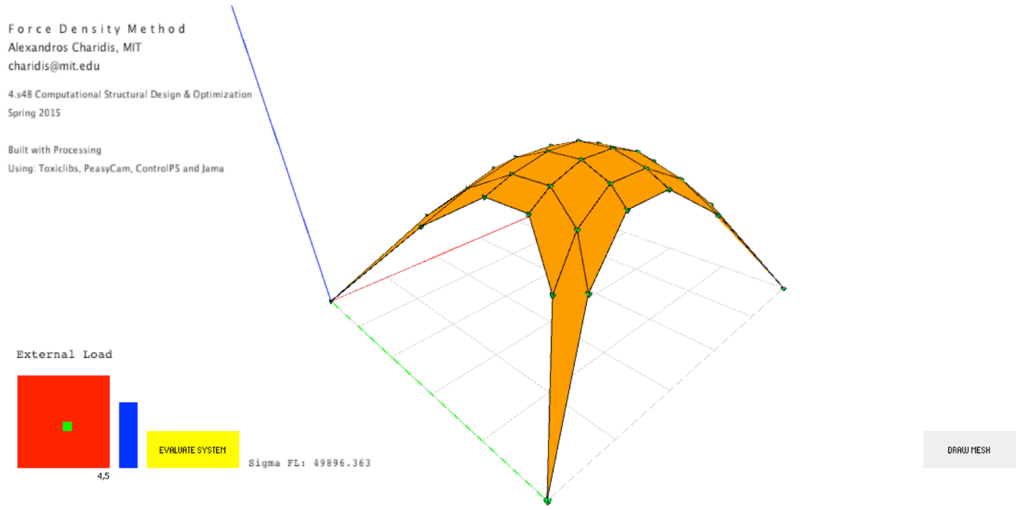
$$\begin{aligned} \boldsymbol{u} &= \ \boldsymbol{Cx}, \boldsymbol{x} = [\boldsymbol{x_0}\ \boldsymbol{x_1}\ \cdots\ \boldsymbol{x_{35}}]^{\boldsymbol{T}} \\ \boldsymbol{v} &= \ \boldsymbol{Cy}, \boldsymbol{y} = [\boldsymbol{y_0}\ \boldsymbol{y_1}\ \cdots\ \boldsymbol{y_{35}}]^{\boldsymbol{T}} \\ \boldsymbol{w} &= \ \boldsymbol{Cz}, \boldsymbol{z} = [\boldsymbol{z_0}\ \boldsymbol{z_1}\ \cdots\ \boldsymbol{z_{35}}]^{\boldsymbol{T}} \end{aligned} \qquad (4)$$

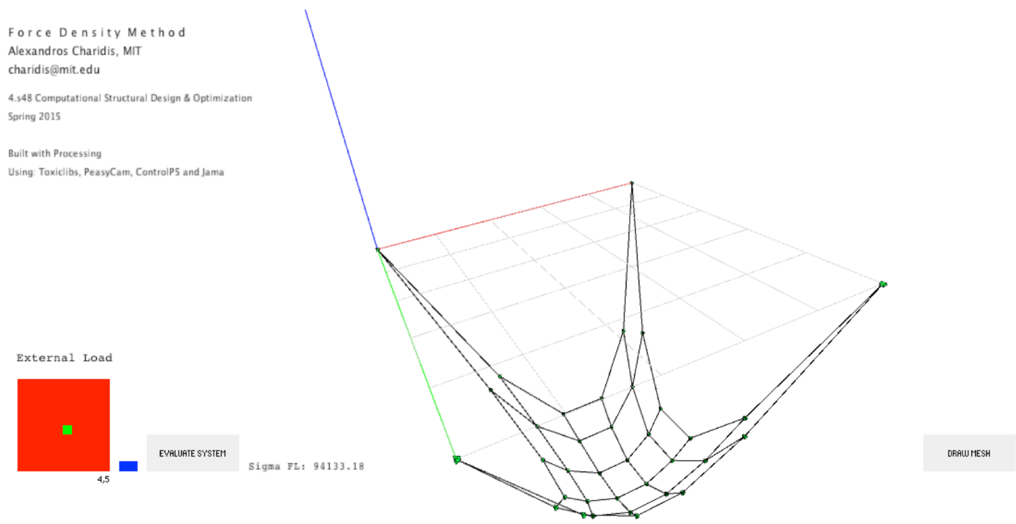
Finally, since the vector of force densities is pre assembled and the lengths are computed as shown in (3), the computation of the structural performance $\Sigma_i F_i L_i$ is reduced to a *matrix-matrix* multiplication i.e. $\boldsymbol{FL}$.

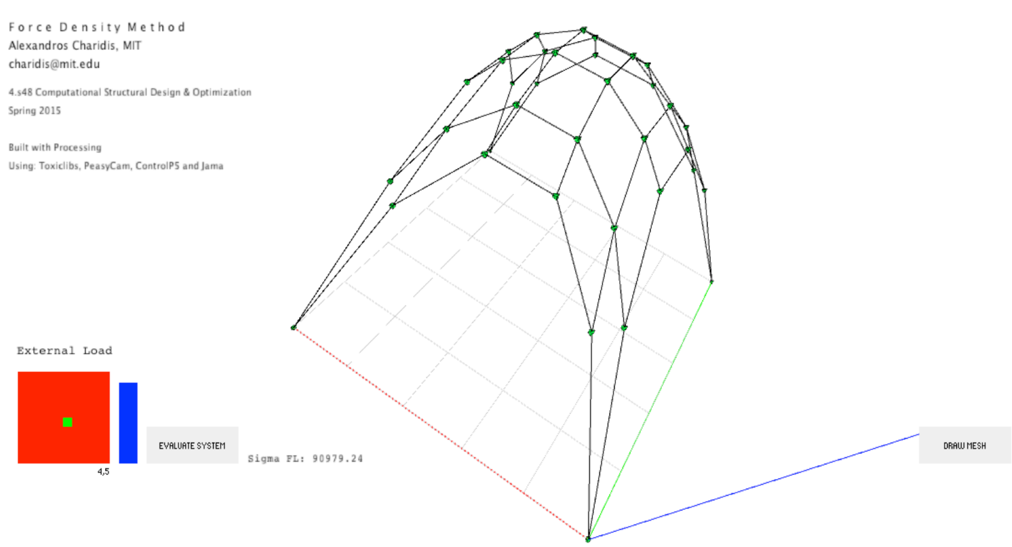


The figure above shows the graphical user interface and the default 3D scene of the application. The program offers controls for adjusting the external load $\boldsymbol{p}$. The evaluation of a structural system is triggered every time the user presses the button "Evaluate System." This is closely related to the intuition behind the FDM that is a non-dynamic method of computing pre-stressed structures in equilibrium. However, the interactivity I provide with the application offers users an instant visual feedback of the way the structure reconfigures when the users adjusts the load vector. In addition, because the system is evaluated once at a time, the program computes its structural performance and prints it on the screen.

(example screenshots)


Force Density Method
Alexandros Charidis, MIT
charidis@mit.edu
4.s48 Computational Structural Design & Optimization
Spring 2015
Built with Processing
Using: Toxiclibs, PeasyCam, ControlP5 and Jama
External Load
4,5
EVALUATE SYSTEM
Sigma FL: 49896.363
DRAW MESH


Force Density Method
Alexandros Charidis, MIT
charidis@mit.edu
4.s48 Computational Structural Design & Optimization
Spring 2015
Built with Processing
Using: Toxiclibs, PeasyCam, ControlP5 and Jama
External Load
4,5
EVALUATE SYSTEM
Sigma FL: 94133.18
DRAW MESH


Force Density Method
Alexandros Charidis, MIT
charidis@mit.edu
4.s48 Computational Structural Design & Optimization
Spring 2015
Built with Processing
Using: Toxiclibs, PeasyCam, ControlP5 and Jama
External Load
4,5
EVALUATE SYSTEM
Sigma FL: 90979.24
DRAW MESH

Dynamic Form Finding

This section explains a computer application for dynamic form finding of meshes represented as *Particle Spring* (PS) systems. Within the area of Computer Graphics, PS systems are used for physically based simulation, for example for physical cloth simulation. PS systems are sometimes called Mass Spring Systems because instead of dealing with an abstract, mass-less and geometry-less spring system, the system deals with physical masses of specific geometry and weight.

A *Mass Spring System* (MSS) is a set of *masses* – points in 3D space with (x, y, z) coordinates and mass – and a set of directed *links* – springs with assigned stiffness, rest length, damping constant, force – that connect them according to a predefined topology. The key abstraction behind a MSS is the notion of *state* of the system. A state holds at any given moment the translation vector and velocity vector of each mass that contributes to the overall state (dynamic movement) of the system.

$$state = \begin{bmatrix} \boldsymbol{x_1} \\ \boldsymbol{u_1} \\ \boldsymbol{x_2} \\ \boldsymbol{u_2} \\ \vdots \end{bmatrix}$$

More concretely, a mass is the couple $[x \ \ \dot{x}]$ and a MSS is $N * [x \ \ \dot{x}]$ where N is the number of masses in the system. Because the system is dynamic, at any given moment we have to compute the *next state* of the system given the current state. MSS requires solving a first order ordinary differential equation (ODE) that takes as input the position and velocity of the system and computes the velocity and acceleration of the system, thus indicating where the system should move.

$$next\ state = \begin{bmatrix} \boldsymbol{u_1} \\ \boldsymbol{a_1} \\ \boldsymbol{u_2} \\ \boldsymbol{a_2} \\ \vdots \end{bmatrix}$$

Acceleration suggests a second order differential equation. Therefore, we replace acceleration ***a*** with **F/m** since at any given moment we can compute all the forces and all the masses of the system. Hense, the next state of the MSS is written as:

$$next\ state = \begin{bmatrix} \boldsymbol{u_1} \\ \boldsymbol{F_1/m_1} \\ \boldsymbol{u_2} \\ \boldsymbol{F_2/m_2} \\ \vdots \end{bmatrix}$$

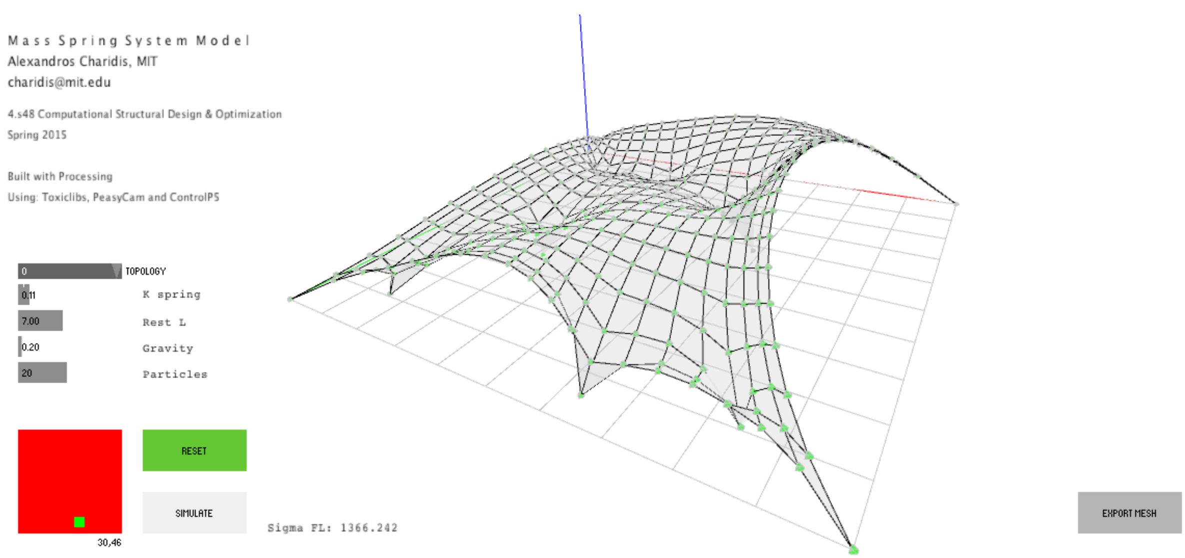
or using a different notation, $N * \left[u \ \ \frac{F}{m}\right]$. This is the time derivative of the state vector at any given moment. For the numerical approximation of the solution I have implemented a simple *Euler Forward* method. The only way that the ODE solver and the actual MSS communicate is a function called `evalF` that accepts a state vector and returns its time derivative and updates all the positions and velocities of the masses in the system.

$$state\ vector \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix} \rightarrow evalF \rightarrow state\ vector' \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix}$$

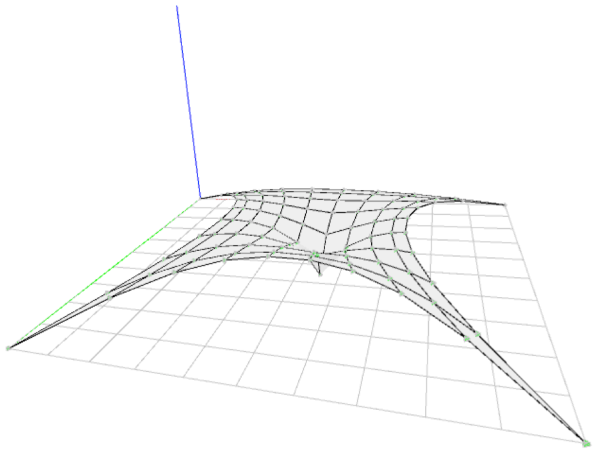
The forces that act upon the MSS can be grouped broadly in three categories: *Unary forces* that exert a constant force, such as the gravity $\boldsymbol{g}$ and viscous drag of the form $f = -k_d \boldsymbol{v}$, where k is the drag constant; *N-ary forces* that exert force to fixed sets of particles, such as spring forces based on Hooke's law; *forces from spatial interaction,* such as attraction, repulsion or the force of a hypothetical wind.

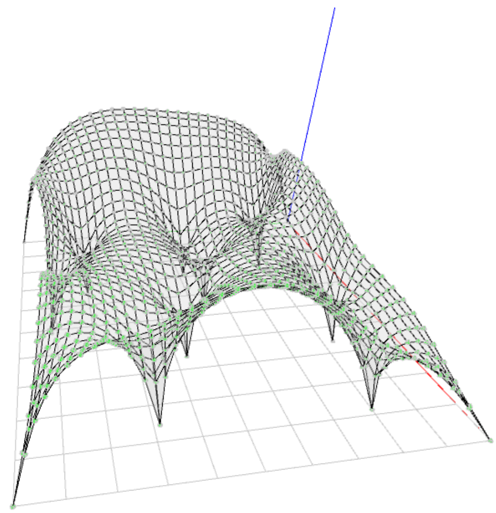
The initial mesh is a rectangular system with *m*x*n* masses and is anchored at its four corners. The density (number of masses) of the mesh can be controlled via the GUI of the application. The overall footprint of the mesh is precomputed at 10*10 units in each side. In addition, new anchor points can be added in real time as the dynamic simulation evolves.

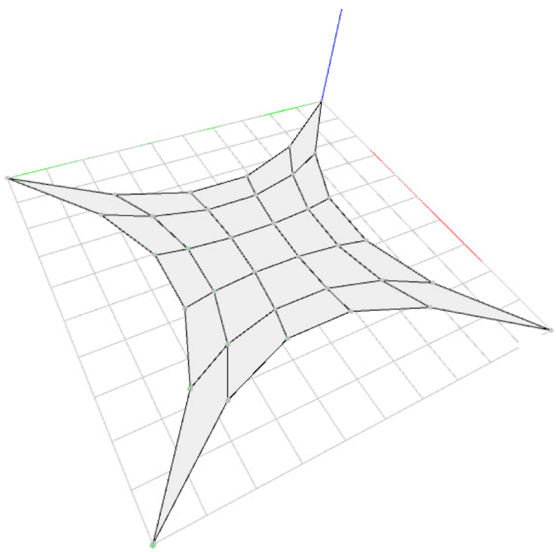
Below is the user interface and the an example 3D scene. There are controls for adjusting the spring stiffness, the rest length of the springs, the gravitational force, the number of particles, a 2D slider interface for adding new anchor points (the user clicks in the red rectangle and the point where the user clicks is mapped to the real 3D space), and a 1D slider for altering the topology of the mesh (with 4 topology variants). Finally, the application prints at each step of the simulation the overall performance of the system as per the usual measurement $\Sigma_i F_i L_i$.

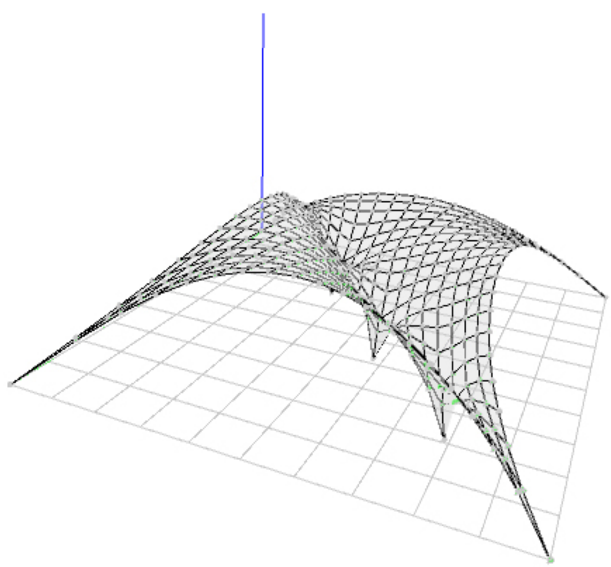


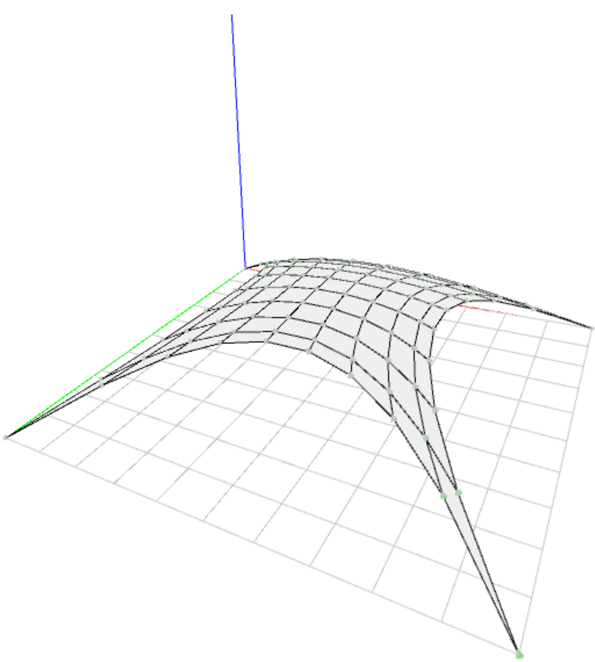
(example screenshots)





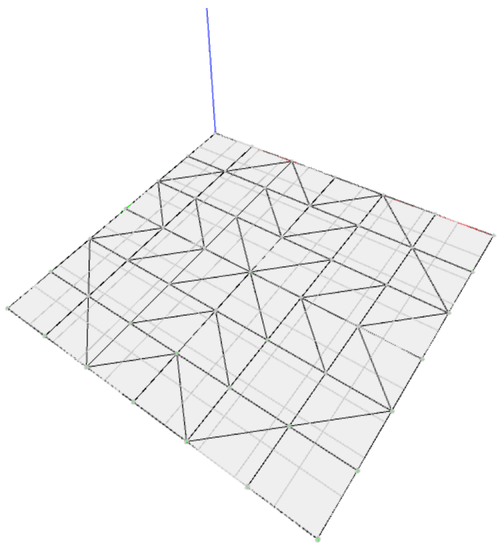


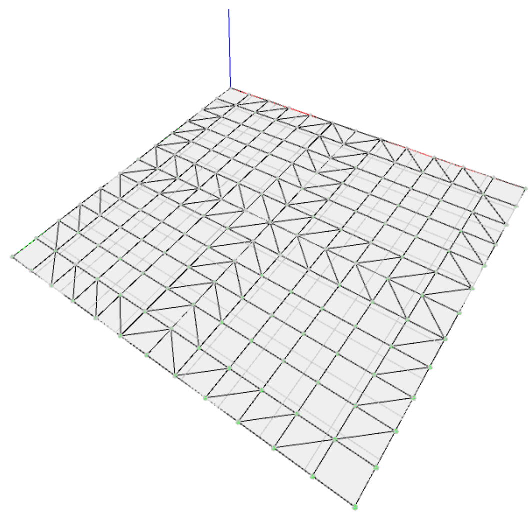




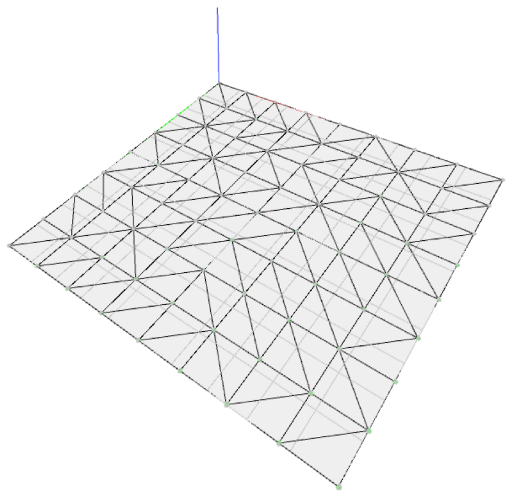
Alternative Grid Shell Topologies

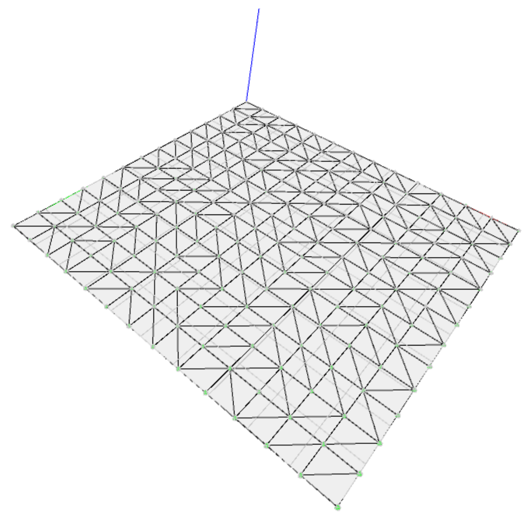
Topology 1



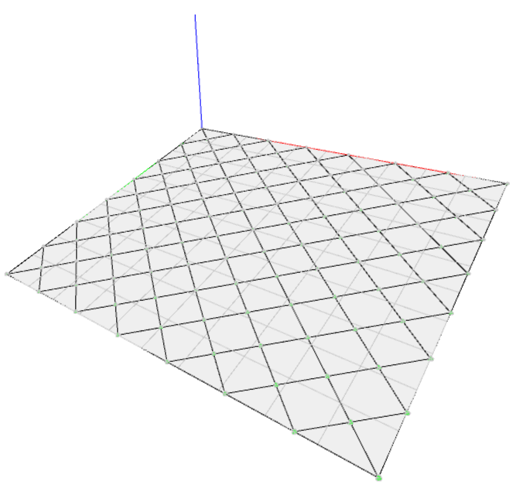


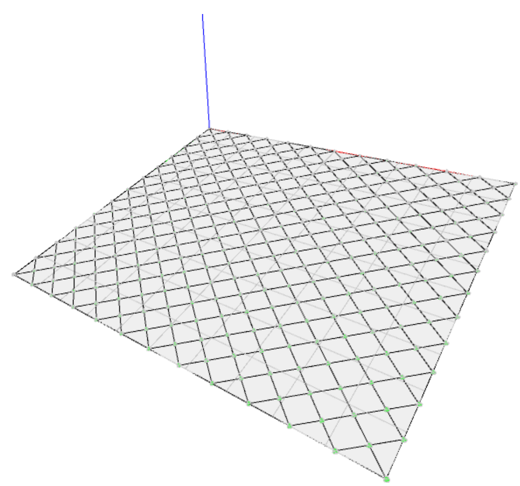
Topology 2



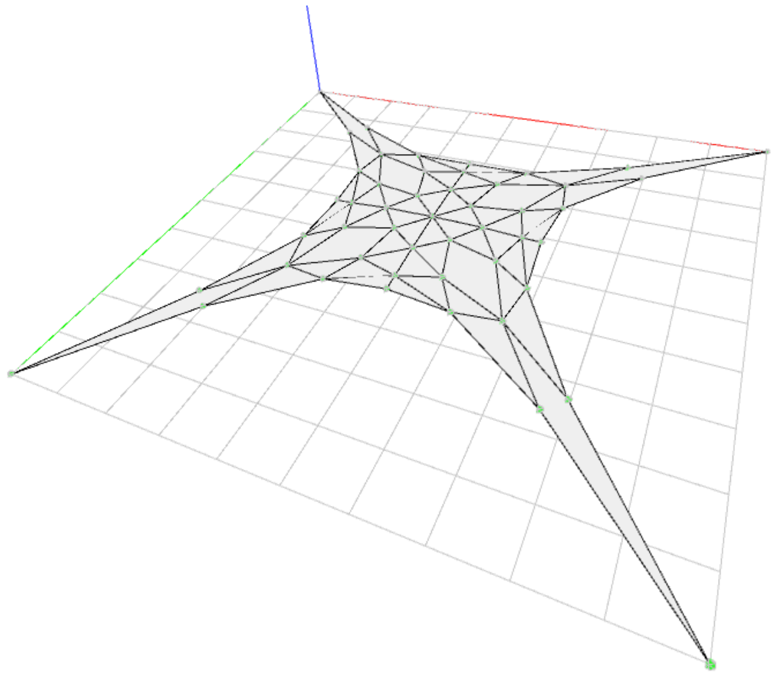


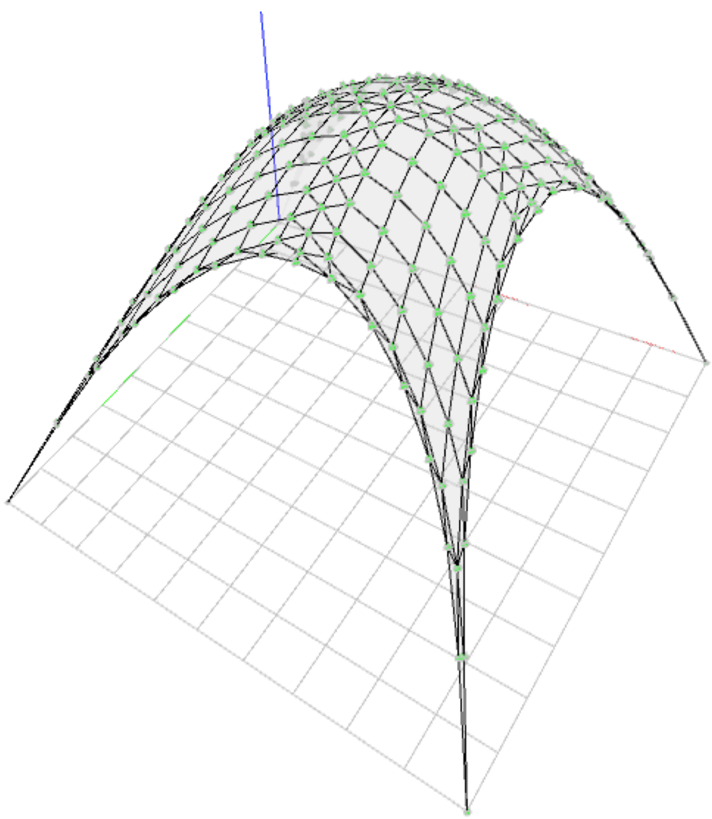
Topology 3



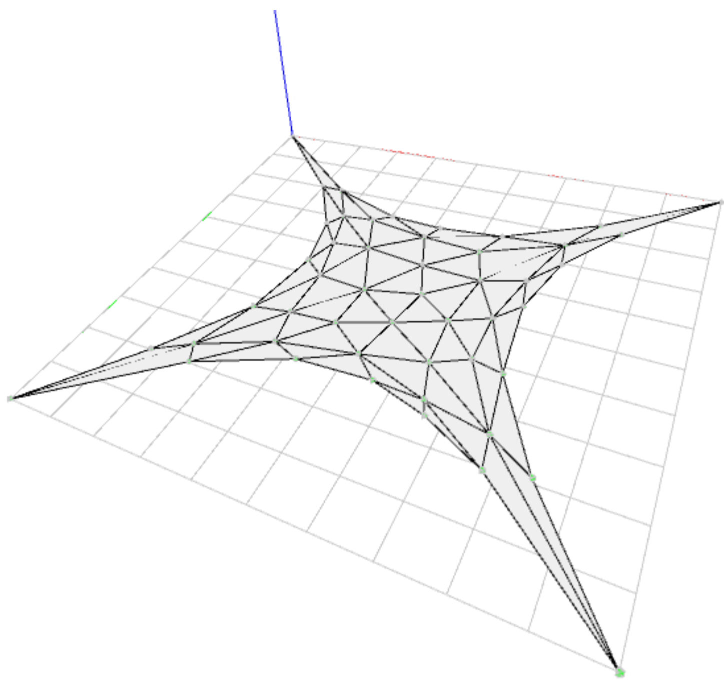


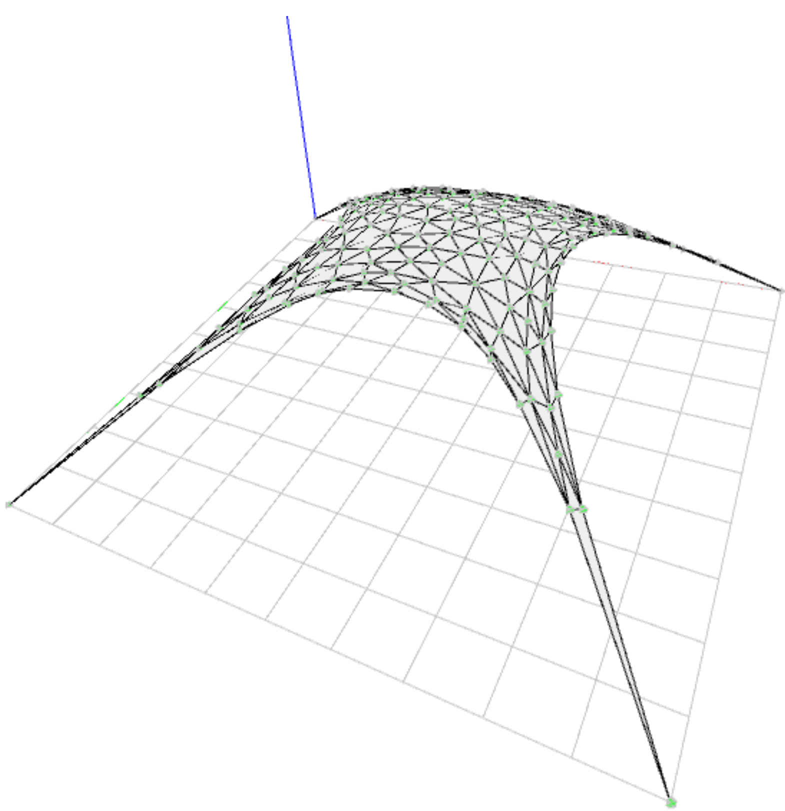
Topology 1





Topology 2





Topology 3