

École des Ponts ParisTech

2014-2015

Pierre Cuvilliers

Cours

Titre

Table des matières

[1. Conception des coques et toiles tendues 5](#_Toc419219939)

[1.1. Équivalence tension-compression, intérêt de la méthode 5](#_Toc419219940)

[1.2. Surfaces minimales, présentation, propriétés 5](#_Toc419219941)

[1.3. Forces extérieures 5](#_Toc419219942)

[1.4. Surfaces connues 5](#_Toc419219943)

[2. Recherche de surfaces à courbure constante 6](#_Toc419219944)

[2.1. Notations 6](#_Toc419219945)

[2.2. Hypothèses de base, justifications 6](#_Toc419219946)

[2.2.i. État de contraintes 6](#_Toc419219947)

[2.2.ii. Discrétisation, maillage 6](#_Toc419219948)

[2.3. Forces sur un élément de surface 6](#_Toc419219949)

[2.4. Équilibre d’un nœud dans le repère local 6](#_Toc419219950)

[2.5. Équilibre d’un nœud dans le repère global 6](#_Toc419219951)

[3. Résolution numérique, algorithme 7](#_Toc419219952)

[3.1. Point fixe 7](#_Toc419219953)

[3.2. Gradient 7](#_Toc419219954)

[3.3. Convergence 7](#_Toc419219955)

[3.3.i. Justification 7](#_Toc419219956)

[3.3.ii. Critère d’arrêt 7](#_Toc419219957)

[3.4. Pseudo-inverse 7](#_Toc419219958)

[3.5. Remise à jour qs 7](#_Toc419219959)

[3.6. Implémentation 7](#_Toc419219960)

[3.6.i. Boucles 7](#_Toc419219961)

[3.6.ii. Erreur numérique 8](#_Toc419219962)

[4. Forces extérieures 9](#_Toc419219963)

[4.1. Câbles 9](#_Toc419219964)

[4.2. Pression : idem 9](#_Toc419219965)

[4.3. Gravité : idem 9](#_Toc419219966)

[5. Grasshopper **Erreur ! Signet non défini.**](#_Toc419219967)

[1.1. Images, utilisation générale 10](#_Toc419219968)

[5.1. Entrées 13](#_Toc419219969)

[5.2. Sorties **Erreur ! Signet non défini.**](#_Toc419219970)

[5.3. Visualisations **Erreur ! Signet non défini.**](#_Toc419219971)

[5.4. Exemples 19](#_Toc419219972)

[5.5. Verifications 27](#_Toc419219973)

# Conception des coques et toiles tendues

## Équivalence tension-compression, intérêt de la méthode

## Surfaces minimales, présentation, propriétés

* État de contraintes uniforme
* Courbure moyenne
* Problème de bord type « bulle de savon »

## Forces extérieures

* Pression + Gravité
* Câbles

## Surfaces connues

* PH
* Caténoïde
* Schwartz
* Plat avec câbles
* Coussin gonflé
* Isler, Kapoor, ILEK, Mantra, Tanz-Brunnen

# Recherche de surfaces à courbure constante

## Notations

## Hypothèses de base, justifications

### État de contraintes

* Tension pure + isotrope
* Densité de contrainte

### Discrétisation, maillage

* FEM masquée
* Tension uniforme vs. Taille différente des mailles
* Différentes formes possibles via qs(i)

## Forces sur un élément de surface

## Équilibre d’un nœud dans le repère local

## Équilibre d’un nœud dans le repère global

# Résolution numérique, algorithme

## Point fixe

## Gradient

* Inversion de matrice
* Méthode de Newton ?
* Instable ? Vitesse d’approche

## Convergence

### Justification

* Lemme de Banach
* Dérivée partielle vs dérivée totale (article)

### Critère d’arrêt

* Résidu forces vs déplacement
* Adimensionnement

## Pseudo-inverse

## Remise à jour qs

* Équilibre des tensions

## Implémentation

### Boucles

* Nœuds
* Itérations positions
* Itérations qs

### Erreur numérique

# Forces extérieures

## Câbles

* Intérêt : poutres de rive, fixation toiles
* Force, équilibre repère local, équilibre repère global
* Algo point fixe

## Pression : idem

## Gravité : idem

# User manual

## Features

Kroto is a membrane form-finding tool for Rhino 5 / Grasshopper. It is based on the surface stress density method, open-source[[1]](#footnote-1), and written in Python. It is specifically aimed at finding the form of a membrane in equilibrium under a set of predefined loads, such as:

* isotropic membrane stresses, uniform (minimal surfaces) or not;
* uniform pressure;
* uniform vertical load;
* edges constraints (cables, fixed points).

However, it is not a structural analysis software and so will not output physically meaningful force and stress values nor will it find the behavior of the form-found membrane under a new set of loads (“live loads”) disrupting its equilibrium. Only the position of the points on the membrane in its equilibrium state have a meaningful value; while forces and stresses are directly proportional to the chosen stress-density coefficient, this coefficient does not have a direct physical interpretation.

Kroto does not take into account second-order effects such as:

* bending stiffness;
* material non-linearities;
* non-isotropic membrane stress states, i.e. no shear stress.

Kroto only accepts valid triangular Rhino meshes as input.

Kroto was written by Pierre Cuvilliers with Lionel du Peloux and Cyril Douthe at the Laboratoire Navier, École des Ponts ParisTech. It is part of the THINkSHELL project[[2]](#footnote-2).

## Installation

### Dependencies

To use Kroto, you will need:

* Rhino 5: <http://www.rhino3d.com/download>. At the time of writing, Rhino 5 for Mac does not support Grasshopper, so Windows only (or run it from Rhino, at your own risk[[3]](#footnote-3))! Rhino 5 SR9 recommended.
* Grashopper (again, see note 3 on page 11): <http://www.grasshopper3d.com/page/download-1>. GH 0.9.0075 recommended.
* GH Python: <http://www.food4rhino.com/project/ghpython?ufh>. GH Python 0.6.0.3 recommended.
* Kroto:

To ease the workflow, it is recommended you also install the following Grasshopper plugins:

* TT Toolbox: <http://www.food4rhino.com/project/tttoolbox?ufh>. Used for the legend display.
* Weaverbird: <http://www.giuliopiacentino.com/weaverbird/>. Helps creating meshes in Grasshopper.

### Installation

To actually install Kroto, you need to extract the “Libraries” and “UserObjects” folder in Kroto\_1.X.Y.zip to the “settings” folder of Grasshopper.

You can find the Grasshopper settings folder from Grasshopper: File > Special Folders > Settings folder; and from Rhino by running the command GrasshopperFolders and choosing Settings. Windows will ask you if you want to merge the folders (you should answer yes) and, if you had a previous version installed, if you want to overwrite the files (in that case, you should answer yes too).

After extracting the files, do not forget to reload Grasshopper into Rhino, if it was still loaded when extracting. To do that, run GrasshopperUnloadPlugin from Rhino (save current work first), and launch Grasshopper again. You could also quit and reopen Rhino, but you might also fix all your computer problems if you [try turning it off and on again](https://www.google.com/search?q=have+you+tried+turning+it+off+and+on+again).

## First run

### Files

Kroto consists of five Grasshopper user objects (see Figure 5.1) and three supporting library files. The Grasshopper user objects are:

* Solver: solves the problem defined earlier, using the provided options. Calls the meshminimize module.
* Problem: defines a Kroto problem to solve, based on a triangular mesh, external forces, and edge conditions.
* Edges: defines the edge conditions for a mesh, using cables and fixed points.
* Options: Provides the default options, and allows to change them.
* Mesh closest points: small helper components that finds the vertices of a mesh that are closest to a set of points.

If the installation went well, you should find the user objects in the THINkSHELL tab

And the Python module files are:

* Meshminimize.py: the solver itself.
* Meshminimizehelper.py: defines some helper functions, mainly for initialization.
* Vectorworks.py: defines vector and matrix calculus functions, in 3D and arbitrary dimension.
* In the sources, you will also find main.py, which can be used to run Kroto directly from Rhino, although this is undocumented.

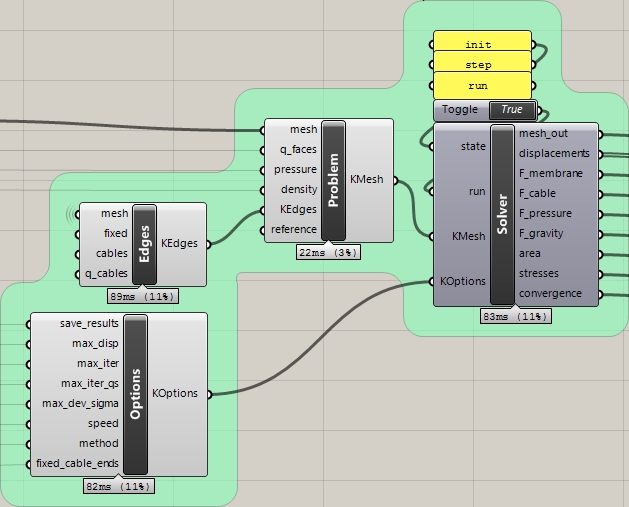


Figure 5.1: The four main Kroto components.

One of the simplest working definitions you can use to run Kroto is an inflated flat square mesh. Using only default options and no referenced Rhino geometry, it is defined by Figure 5.2. Below is a step-by-step explanation:

* First, we create a flat meshed square with the mesh plane component from Grasshopper.
* Then we triangularized this mesh as Kroto only accepts triangular meshes.
* This mesh, and the pressure value of 1.0 are passed to the Problem user object; density defaults to 0 and the edge conditions (KEdges) default to fixed (all naked vertices are pinned). The surface stress density coefficients default to 1.0.
* We then pass this problem to the Solver box, and initialize it with the string init (without quotation marks) in the state input. Defaults options are assumed (KOptions), see 0 for more explanations.
* Then we pass the string run to the state input, and the boolean True to the run input, running the form-finding algorithm for one pass (that is, 10 positions iterations ran for 2 stress density coefficients iterations, or until the maximum displacement in one iteration is less than 0.01 units). At this point, you should see something similar to the image at the top-left of Figure 5.2.
* Since the default options stop the algorithm quite early, it might be a good idea to run it again from the result mesh by drawing one of the input lines to Solver again.

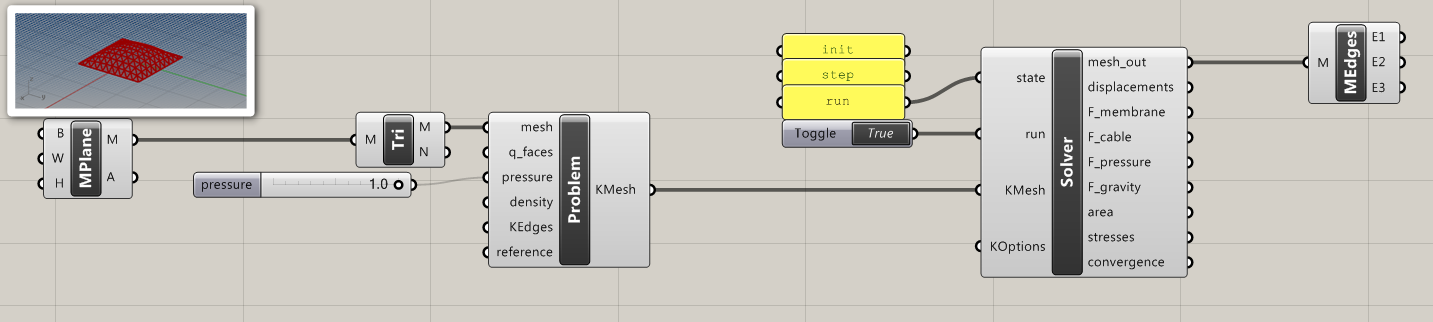


Figure 5.2: Grasshopper definition for a simple inflated square.

As a rule of thumb, a mesh with 100-200 vertices should be enough to describe usual membrane shapes. Running it for 100-400 positions iterations (max\_iter) and 1-10 (less if you are looking at true minimal surfaces, more if you are playing with external forces) stress density iterations (max\_iter\_qs) should be enough to get to a meaningful result. If you want to run all the iterations, whatever their result, set max\_disp and max\_dev\_sigma as 0, however a 0.01-0.05 max\_disp is usually enough to get a precision under one Rhino unit. See 5.7 for more details.

## Inputs

### Kroto Solver component

Runs the Kroto solver. For a list of the outputs, see 5.5.

#### state

Takes one of the string values init, step or run.

init initializes the solver, putting the mesh in its initial form and sending the options chosen to the meshminimize module.

step runs the solver for one (position) iteration, if the run input is True. If the solver has exhausted the position iteration counter but not the stress density iteration counter, it updates the stress densities and reinitializes the position iterations counter.

run runs the solver, if the run input is True, for as many iterations as defined by the options, or until it reaches the convergence criterion.

#### run

Boolean, performs the action in state when True.

#### KMesh

The problem definition to run. Generated by the Kroto Problem component.

#### KOptions

The options to pass to the solver. Generated by the Kroto Options component.

### Kroto Problem component

#### mesh

A Rhino mesh to be relaxed. Only fully triangular meshes are supported. Run your mesh through the triangulate Grasshopper component or use the \_TriangulateMesh command in Rhino if you need.

If your mesh is made from the junction of multiple meshes (using the \_Join command in Rhino), make sure you welded the seams afterwards (with the \_Weld and \_WeldEdges commands). Otherwise, the seams will actually be made of pairs of naked, duplicate vertices. If computed, this would actually be equivalent to two disjoint meshes.

#### q\_faces

A single, or a list of, float values to assign the stress densities of the mesh faces. If a single number is provided, all stress densities are set to this value. If a list is provided, all faces are assigned the corresponding value in the order in which they appear in the mesh. If no value is provided, defaults to 1.

#### pressure

A single float values to assign the pressure of the mesh faces. If no value is provided, defaults to 0.

#### density

A single float values to assign the density (vertical load) of the mesh faces. If no value is provided, defaults to 0.

#### KEdges

The edge conditions to apply to the mesh. Generated by the Kroto Edges component. If no value is provided, defaults to all naked vertices fixed (pinned).

#### reference

A reference mesh against which to compute the displacements. If no value is provided, defaults to the initial mesh in the mesh input.

### Kroto Edges component

Defines the edges of a problem for Kroto: cables and fixed-points. Returns KEdges, a set of edge conditions on a mesh.

#### mesh

The Rhino mesh to where the edge conditions are defined.

#### fixed

List of Booleans, one for each vertex in the mesh. If True, the corresponding vertex will be fixed (pinned). Use the Mesh Closest Point component to generate such a list. If no value is provided, all the naked vertices default to fixed (pinned).

#### cables

The Rhino curves that define the cables on the mesh. Only naked vertices can be on a cable, and a point can be on multiple cables. All the vertices that are on the curve (up to the current Rhino document tolerance) will be joined in the order they are on this curve. A closed curve will generate a closed cable (i.e. all vertices on the cable will have a previous and next point on the cable, even the first and last ones), but the start point will still be fixed if fixed\_cable\_ends is set to True.

#### q\_cables

A single, or a list of, float values defining the force density coefficients of the cables. If a single value is provided, all cables get this value. If a list is provided, cables are assigned the corresponding value in order. If no value is provided, defaults to 1.

#### fixed\_cable\_ends

Boolean, if True all vertices lying on the end of a cable are considered fixed (pinned). A polyline vertex is not considered as a cable end. Defaults to True.

### Kroto Options component

Defines the options for the Kroto solver. Returns KOptions, a set of options for the solver.

#### save\_results

Boolean, saves the following values at each iteration if True: membrane, pressure, gravity and cable forces for each vertex, total area of the mesh, stress value in each face. Defaults to False.

#### max\_disp

Float, stops the current fixed stress densities iteration when, between two successive position iterations, no vertex has moved of a distance greater than this value (in the unit of the current Rhino document). If set to 0, the solver always runs the maximum number of position iterations. Defaults to 0.01.

#### max\_iter

Integer, the maximum number of position iterations. Defaults to 10.

#### max\_iter\_qs

Integer, the maximum number of stress densities iterations. Defaults to 2.

#### max\_dev\_sigma

Float, stops the run when, at the end of a fixed stress densities iteration, the maximum distance to the mean stress value in all faces is less than this value. If set to 0, the solver always runs the maximum number of stress densities values iterations. Defaults to 1.

#### speed

Float, sets a reduction (if smaller than 1) or amplification (if larger than 1) coefficient to apply to the displacements of the current iteration. For example, if set to 0.5, then each vertex will only be moved of half the displacement that equilibrates the forces on this vertex. For the methods “Seidel” and “fixed-point”, you should probably leave this value at 1, as this algorithms are generally stable. The “gradient” method however, is usually only stable with speed under 0.75.

#### method

String, either seidel, fixed-point or gradient. Chooses the algorithm to use in the position iterations. fixed-point is the standard fixed-point iteration[[4]](#footnote-4): the new vertex position is the value of the function characterizing the force equilibrium at the old vertex position. seidel[[5]](#footnote-5) is similar to the fixed-point iteration but using the updated values for the positions already computed in the current iteration to compute the characteristic equilibrium function. gradient explicitly solves the equilibrium system of the current vertex, which is similar in some aspects to a Newton-Raphson[[6]](#footnote-6) iteration. In theory, gradient should be the fastest, but it is unstable in the general case and has to be slowed down by the speed input, effectively making it slower than the other two algorithms.

You should probably use seidel as it has a slight speed advantage over fixed-point and is equally stable. However it breaks most of the symmetries in your problem, which can be a good (if your problem tends to oscillate between two close symmetrical solutions) or a bad thing (if you believe an elegant, symmetrical answer is needed to your symmetrical problem).

### Mesh Closest Point component

Finds the vertices closest to each point in a collection, on a mesh. Returns fixed, a list of Booleans, one for each vertex, True if the vertex is the closest to one of the points in the list.

#### points

A list of Rhino points to find on a mesh.

#### mesh

A Rhino mesh.

## Outputs

### Kroto Solver component

#### mesh\_out

A Rhino mesh (not baked), relaxed by the solver.

#### displacements

The displacements of each vertex since the start of the run. If the solver was run multiple times, this will not be the displacement of the vertices from the initial mesh input.

#### f\_membrane

A Grasshopper tree of float values, giving the membrane forces on each vertex, at each iteration. The structure of the tree is the following: {0; A; B}[i], where A is the stress densities iteration, B the position iteration, and i the vertex number. Only written if save\_results is set to True.

#### f\_cable

A Grasshopper tree of float values, giving the cable forces on each vertex, at each iteration. The structure of the tree is the following: {0; A; B}[i], where A is the stress densities iteration, B the position iteration, and i the vertex number. Only written if save\_results is set to True.

#### f\_pressure

A Grasshopper tree of float values, giving the pressure forces on each vertex, at each iteration. The structure of the tree is the following: {0; A; B}[i], where A is the stress densities iteration, B the position iteration, and i the vertex number. Only written if save\_results is set to True.

#### f\_gravity

A Grasshopper tree of float values, giving the gravity forces on each vertex, at each iteration. The structure of the tree is the following: {0; A; B}[i], where A is the stress densities iteration, B the position iteration, and i the vertex number. Only written if save\_results is set to True.

#### stresses

A list of float values, giving stress on each face, at the end of the run.

#### area

A Grasshopper tree of float values, giving the total area of the mesh, at each iteration. The structure of the tree is the following: {0; A}[i], where A is the stress densities iteration, and i the position iteration. Only written if save\_results is set to True.

#### convergence

A Grasshopper tree of float values, giving the current value of the convergence criterion (i.e. the maximum displacement of a vertex from the previous iteration), at each iteration. The structure of the tree is the following: {0; A}[i], where A is the stress densities iteration, and i the position iteration.

## Visualizations

Four visualization clusters are provided in the example files:

* Mesh-Surface compare: returns the maximum distance between the vertices of the mesh and their closest point counterparts on the surface.
* Colorize mesh: colorizes the faces of a mesh based on the list of values provided. It uses the Legend component from TTToolbox (see 5.2.i, Dependencies), but will still work if this add-on is not in your Grasshopper installation.
* Preview displacements: draws the displacement vectors on the vertices of the mesh.
* Log-Log: changes a linear y-plot (i.e. a list of y-values regularly spaced on the x axis) to a log-log plot to be plotted with Grasshopper’s Quick Plot component.

## Choosing the options

### q\_faces

Unless you want to manually specify different values for each faces, it is probably best to leave this at 1. Most of the other values are expressed in units of the stress density coefficient, so changing this is mostly equivalent to dividing the rest of the values by the same amount.

### pressure and density

Before setting this to large values (anything larger than 1 will yield large displacements, as the forces in play will quickly outsize the membrane-only forces when q\_faces is 1), so be sure to check that your problem runs as expected before, using fewer iterations and smaller values for these coefficients.

When these coefficients are used, the convergence is usually slower so you need to adjust max\_iter and max\_iter\_qs accordingly.

### q\_cables

The (linear) force density coefficient is equivalent in units to a (surface) stress density coefficient multiplied by a squared length. This means you will have to change it when the scale of the model is changed, the Rhino units are changed, or when q\_faces is changed: most importantly, when you scale your model by a factor a, multiply q\_cables by a factor a².

Interesting values will be found when the expected forces in cables cancel out the forces applied by the membrane. A basic formula describing this equilibrium is:

where is the tension in the cable, the membrane stress in the part of the membrane pulling on the cable and the radius of curvature in the cable. Substituting and , with the length of the cable and the area of the part of the membrane pulling on the cable, we get:

This formula makes sense in the units of the current Rhino document. When the value is too small, the mesh will collapse on itself. When the value is too large, the cable will tend to be a straight line from the start to the end point.

### max\_disp

In the algorithm used, the displacement between two successive iteration is directly proportional to the residual value of the force equilibrium. Then, stopping when the maximum vertex displacement between two iterations is lower than a set value ensures that we effectively control the error at the end of the run. However, the error at the end of the run cannot be directly linked to the vertex displacement and you should expect it to be at least 10 times the value of max\_disp.

A good idea is to set max\_disp to something related to the average dimension of a face in your mesh, for example 1/1000 of this dimension. This is usually enough to have an error in the range of 1 %.

### max\_iter

The number of faces in your mesh directly influences the number of iterations needed to converge to a solution. As during one iteration, only the position of the neighboring vertices is considered in the equilibrium of a vertex, max\_iter should be at least several times the number of edges on the shortest path linking the two vertices that are the furthest apart in your mesh. For example, a value of 100 in a mesh of 10x10 faces seems reasonable and will usually lead you to an error in the order of 1 %, or a maximum vertex displacement of 1/1000 of the representative edge length.

Pressurized and weighted membranes tend to move more and take longer to reach a converged state, so you should double or triple it in this case.

If you chose max\_disp wisely, you should be able to put a very large value in max\_iter, as the solver will be stopped when the convergence criterion is reached. However, you should test your model before doing this or you risk having to wait a very long time before it returns!

On my machine, a Core i3 at 3.4 GHz (Kroto is not multithreaded, so the number of cores is mostly irrelevant), Kroto takes about 40 µs per vertex and per iteration to run. This means that a model with 500 vertices run for 200 position iterations and 5 stress density iteration will take approximately 20 seconds to run.

### max\_iter\_qs

Usually 4 or 5 stress density iteration will be enough in pure tensile membranes, sometimes even less without cables. For pressurized and weighted membranes, 10 is usually enough. This is pretty quick since we only need to make the stresses uniform in an almost converged mesh, which creates a new state outside of equilibrium that should not make the vertices move much when computed, and in turn does not change the stresses much.

### max\_dev\_sigma

This value is not critical, but if you feel like you need to use it remember that it is expressed in units of q\_faces times the area of a representative face.

It can be a good idea to leave it to 0, letting the solver always run all possible stress density iterations, as those last iterations will usually be very close to the converged state and take only a few position iteration.

### speed

As explained in 5.4.iv.f, choose 1 for the algorithm “Seidel” and “Fixed-point”, and 0.7 for the algorithm “Gradient”.

## Examples

### Catenoid

The catenoid of revolution is a simple minimal surface resting on two circles sharing a same axis, but seprated vertically along this axis: see Figure 5.3.

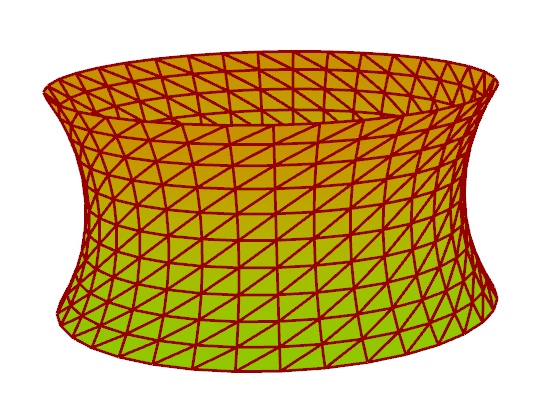


Figure 5.3 A meshed catenoid

The equation describing a catenoid resting on two circles of diameter and separated by a distance is, in the cylindrical coordinates of axis :

To solve the equation defining we can let , and solve graphically or numerically in the positive domain. Looking at Figure 5.4, we see that there will be no solution when . This corresponds to the case where the circles are so distant that the surface comprised of the union of the two disks defined by the circles has a smaller area than any potential catenoid, and thus is minimal. When , there are two solutions, only the smallest one being for a minimal surface.

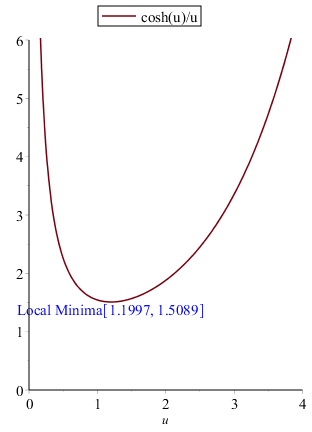


Figure 5.4 Graph of , positive values.

In Kroto, we can simulate a catenoid minimal surface by relaxing a cylindrical mesh. The comparison with the analytical solution gives an error of 0.1 % when the algorithm is run with max\_iter = 100, max\_iter\_qs = 2, and max\_disp = 0.01 %. See the example file “Catenoid” for more details.

### Schwartz D surface

The Schwartz D surface is an infinite, triply periodic minimal surface, resembling an inflated tubular diamond bond structure with a cubic lattice (see Figure 5.5). A subset of this surface is very close to what is commonly described as an hypar with straight edges (even though the hypar is not a minimal surface): see Figure 5.6.

A description of the Schwartz D surface is obtained through a Weierstrass representation, with and :

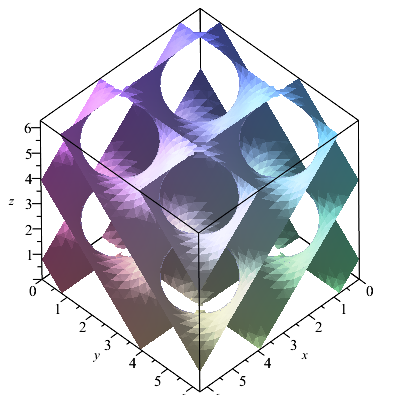


Figure 5.5 Schwartz D surface, 4 periods

The “hypar” subset of this surface can be parametrized as (Dalpé, 1998):

where .

In Kroto, we can simulate Scharz’s D surface by relaxing a planar rectangular mesh whose sides have been attached to the appropriate tetrahedral skew quadrilateral. Close to the results of 5.8.i, the comparison with the analytical solution gives an error of 0.4 % when the algorithm is run with max\_iter = 100, max\_iter\_qs = 2, and max\_disp = 0.01 %. See the example file “Schwarz” for more details.

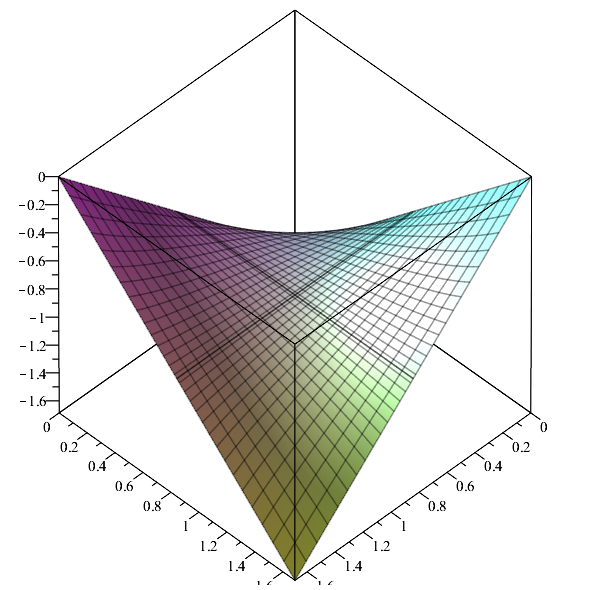


Figure 5.6 Schwartz D surface, "hypar" subset

### Scherk first surface

The Scherk fisrt surface is a doubly periodic, infinite minimal surface. It is the surface which continuously transforms a set of parallel vertical planes bounded to the positive z-domain, to the same set of planes bounded to the negative z-domain and turned by 90° around the z-axis (see Figure 5.7).

An implicit representation of this surface is:

When limited to one period, the first Scherk surface can be expressed:

In Kroto, following the same strategy as in 5.8.ii, we can simulate this surface by relaxing a planar rectangular mesh whose sides have been upward (for the sides along the x-axis) or downward (sides along the y-axis) by a “large” distance (we took 10 times the side length of the mesh). Close to the results of 5.8.i, the comparison with the analytical solution gives an error of 0.4 % when the algorithm is run with max\_iter = 100, max\_iter\_qs = 2, and max\_disp = 0.01 %. See the example file “Scherk” for more details.

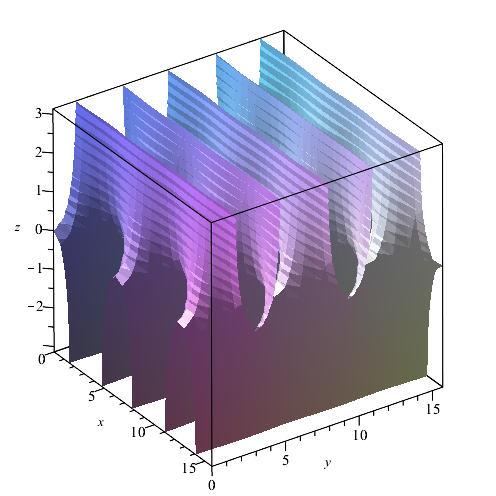


Figure 5.7 Scherk first surface, 2.5 periods

### Inflated square or “pillow”

As an example of a mesh using the cables and pressure options, we simulated an pressure-inflated square held by cables in the example file “Inflated square”. Although there is no analytical solution known to the authors, the results seem quite possible: see Figure 5.8.

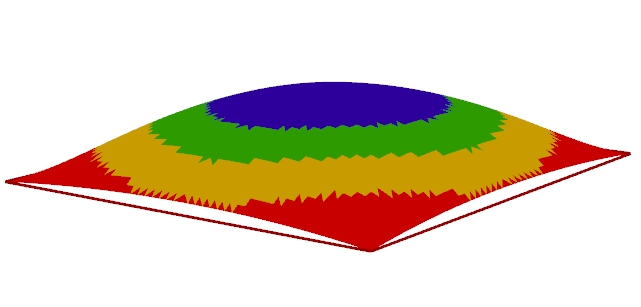


Figure 5.8 Pillow-like inflated square

### ILEK

This famous tensile structure built in Stuttgart provides a good modeling examples for multiple cables. Although it was built out of a cable net, we can still approximate it by a continuous tensile surface.The result seem coherent with the real structure: see Figure 5.9 and Figure 5.10. See the example file “ILEK” for more details.

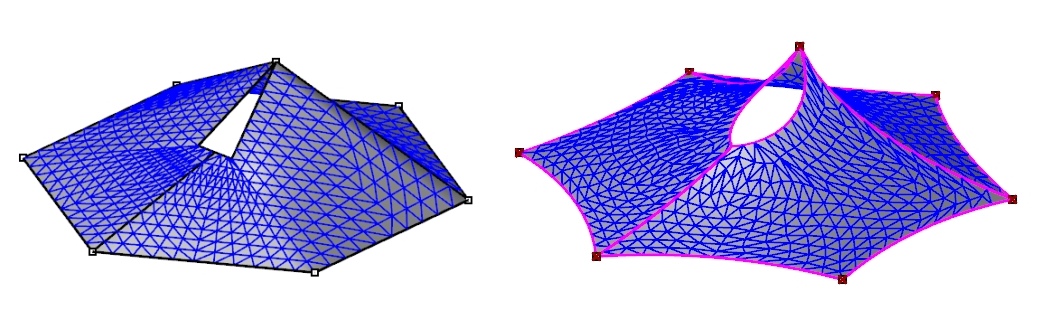


Figure 5.9 Kroto simulation of the ILEK tent structure: input mesh (left) and result (right)



Figure 5.10 The ILEK tent structure

### Chinese hat

### Arcora

## Verifications

* Schwartz à la main (4 points)
* Exemples à solution analytique (Caténoïde, Schwartz, Scherk)
* Convergence en densité de maillage

1. Sources available on Github: https://github.com/THINkSHELL/Kroto [↑](#footnote-ref-1)
2. http://thinkshell.fr/ [↑](#footnote-ref-2)
3. You can run Kroto directly from Rhino, although it is not well documented and supported. See /src/main.py for pointers. [↑](#footnote-ref-3)
4. http://en.wikipedia.org/wiki/Fixed-point\_iteration [↑](#footnote-ref-4)
5. http://en.wikipedia.org/wiki/Gauss%E2%80%93Seidel\_method [↑](#footnote-ref-5)
6. http://en.wikipedia.org/wiki/Newton%27s\_method [↑](#footnote-ref-6)