September 25th - 28th, 2017
INTERFACES
architecture . engineering . science IASS 2017 hamburg

Proceedings of the IASS Annual Symposium 2017
"Interfaces: architecture.engineering.science"
25 - 28th September 2017, Hamburg, Germany
Annette Bögle, Manfred Grohmann (eds.)

Brick patterns on shells using geodesic coordinates

Emil ADIELS*, Mats ANDER [a], Chris J K WILLIAMS[b]

* Department of Architecture, Chalmers University of Technology
Gothenburg, Sweden
emiladiels@gmail.com

[a] Department of Applied Mechanics, Chalmers University of Technology
[b] Department of Architecture, Chalmers University of Technology

Abstract

We present two separate strategies for generating brick patterns on free form shells and vaults using geodesic coordinates. The brickwork is specified by a surface on which there is a geodesic coordinate system satisfying the condition for a constant distance between bed joints. The first strategy integrates the generation of the geodesic coordinates in a form finding procedure derived from the geometrical and mechanical properties of a shell. The geometric and structural equations are solved using dynamic relaxation. The second strategy can be applied on an arbitrary surface separating the form finding and brick pattern generation enabling adaption to different constraints in the design process.

Key words: differential geometry, brick patterns, shells, brickwork, form finding, geodesic coordinates

1. Introduction

Any discussion of brickwork must include the influence of the format and shape of the brick, and this particularly applies to brick shells and vaults. The field of geometry and brickwork is historically not as well known or documented as the related topic of stereotomy, that is the 3 dimensional cutting of stones, where the literature goes back to the 16th century in the work by Philibert de l'Orme [3]. The problem of tessellating a free form surface with a single element type is generally very difficult, if not taking the production method into account. Importantly, the geometry of brickwork is never exact. Traditional bricks have an overall shape of a cuboid, but can be both curved and twisted, and can when needed be easily modified with a bricklayer's hammer or a scutch hammer, and possibly a chisel or bolster. The flexible joints, the mortar, make the structure able to tolerate the deviance of the bricks and adapt to the global geometry. This means that the craftsmen traditionally had influence on the final solution but also that there exists a resilience in the mathematical formulation.

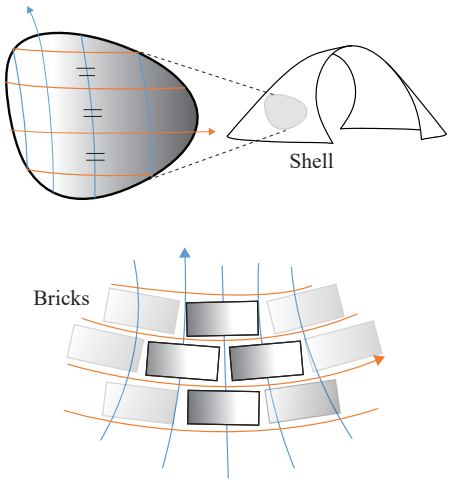

Figure 1: Geodesic coordinates on a surface guarantees and equal spacing of the coordinate curves making it suitable as basis for brick patterns on free form shells.

Old handbook drawings of patterns on vaults are often limited to the bed joints as in figure 2. The most important geometrical constraint is therefore that the distance between the bed joints is constant. This property can be found in a geodesic coordinate system comprising geodesics on a surface and their orthogonal trajectories (see section 3) which guarantees a constant length between coordinate curves on the surface. This would enable the placing of bricks in between the coordinate curves of such system, illustrated in figure 1.

We shall present two strategies to generate the geodesic coordinate pattern on a shell:

1. Combined form finding and pattern generation using dynamic relaxation as described in section 4.
2. Generation of patterns on an arbitrary shell from a designer specified curve and thereby separating form and pattern as described in section 5.

2. The geometrical and structural properties of shells

The form finding of shell structures involves the specification of the geometry of a shell in static equilibrium subject to geometric constraints. The best known example is the equal mesh or Chebyshev net used by Frei Otto for projects such as the Munich Olympic Stadium and the Mannheim Multihalle [5]. In an alternative technique Block and Ochsendorf [2] define the form found state of stress in a shell using Thrust Network Analysis.

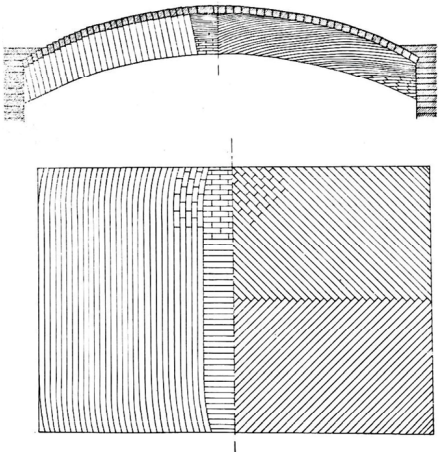

Figure 2: A drawing of a saucer dome showing various on how to place the brick pattern. Typically these brick patterns are limited to the lines of the bed joints and not every single brick, from Paulsson [6].

The geometry of the coordinate curves on a surface is specified by the components of the metric tensor a_{11}, $a_{12} = a_{21}$ and a_{22}, using the notation in Green and Zerna [4], also known as coefficients of the first fundamental form, E, F and G, using the notation in Struik [7]. The state of membrane stress in a shell is specified by the components of the membrane stress tensor, n^{11}, $n^{12} = n^{21}$ and n^{22}, again using the notation in Green and Zerna [4].

Thus we have 6 quantities to determine, but only 3 equations of equilibrium. We therefore need 3 more equations and in the case of the equal mesh net they are

$$a_{11} = a_{22} = \text{constant} \tag{1}$$

$$n^{12} = 0. \tag{2}$$

Equation (2) ensures that the state of stress corresponds to forces along the lines of the mesh.

However we cannot use an equal mesh net for brickwork because we need a constant spacing between bed joints as shown in figure 1. If

$$\frac{a_{11}a_{22} - (a_{12})^2}{a_{11}} = \text{constant} \tag{3}$$

then there is a constant distance between the curves $\theta^2 = \text{constant}$. Green and Zerna [4] use θ^1 and θ^2 for the surface coordinates instead of the u and v which are often used in books which do not use the tensor notation, such as Struik [7].

We can arbitrarily specify that the coordinate curves are orthogonal so that

$$a_{22} = \text{constant} \tag{4}$$

$$a_{12} = 0. \tag{5}$$

Thus we now have 2 equations and we require one more. If we stipulate that the principal stresses are perpendicular to the bed and head joints we have

$$n^{12} = 0. \tag{6}$$

There is no absolute requirement that the principal stresses are perpendicular to the bed and head joints in brickwork, but we have to specify some condition on stress in order to have a system of equations that we can solve. The obvious advantage of the principal stresses being perpendicular to the bed and head joints is that there is no possibility of sliding along the bed joints.

The only difference between equations (1) and (2) and equations (4), (5) and (6) is that in the former we have $a_{22} = \text{constant}$, whereas in the latter we have $a_{12} = 0$. However, we shall see that this change does make the numerical solution more difficult.

3. Geodesic coordinates

In this section we shall demonstrate the well known properties of a geodesic coordinate system using the notation from Green and Zerna [4], although they do not cover this topic. From equations (4) and (5) we have,

$$0 = \frac{\partial a_{22}}{\partial \theta^1} = \frac{\partial}{\partial \theta^1}(\mathbf{a}_2 \cdot \mathbf{a}_2) = 2\mathbf{a}_2 \cdot \frac{\partial \mathbf{a}_2}{\partial \theta^1} \tag{7}$$

$$0 = \frac{\partial a_{12}}{\partial \theta^2} = \frac{\partial}{\partial \theta^2}(\mathbf{a}_1 \cdot \mathbf{a}_2) = \frac{\partial \mathbf{a}_1}{\partial \theta^2} \cdot \mathbf{a}_2 + \mathbf{a}_1 \cdot \frac{\partial \mathbf{a}_2}{\partial \theta^2} \tag{8}$$

and therefore since

$$\frac{\partial \mathbf{a}_1}{\partial \theta^2} = \frac{\partial^2 \mathbf{r}}{\partial \theta^1 \partial \theta^2} = \frac{\partial \mathbf{a}_2}{\partial \theta^1} \tag{9}$$

we have

$$\mathbf{a}_1 \cdot \frac{\partial \mathbf{a}_2}{\partial \theta^2} = 0. \tag{10}$$

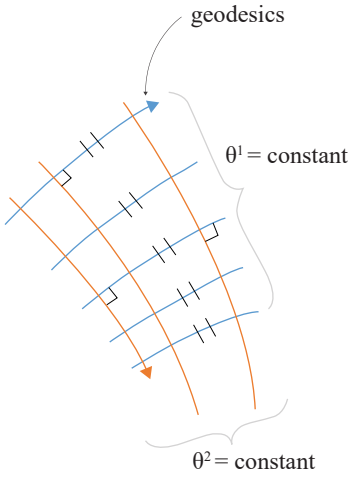

Figure 3: Geodesic coordinates, redrawn from Striuk [7]. The orthogonal trajectories intersect the geodesics at a right angle and equal length.

This last equation means that the geodesic curvature of the curves $\theta^1 = \text{constant}$ is zero, or in other words the curves $\theta^1 = \text{constant}$ are *geodesics* on the surface. The shortest distance between two points on a surface is a geodesic and it is a nice application of the calculus variations to show that this is consistent with the definition of zero geodesic curvature, although it is intuitive in the sense that a string stretched across a smooth surface will slide sideways giving the minimum length and zero geodesic curvature.

Coordinates which satisfy (4) and (5), and therefore also (10) are known as geodesic coordinates. Gauss's Theorema Egregium is particularly elegant in geodesic coordinates. Introducing the coefficients of the second fundamental form, b_{11}, $b_{12} = b_{21}$ and b_{22},

$$\frac{\partial \mathbf{a}_2}{\partial \theta^2} = b_{22}\mathbf{a}_3$$

$$\frac{\partial \mathbf{a}_2}{\partial \theta^1} = \frac{1}{2a_{11}}\frac{\partial a_{11}}{\partial \theta^2}\mathbf{a}_1 + b_{12}\mathbf{a}_3$$

so that

$$\begin{aligned}\frac{\partial^2 \mathbf{a}_2}{\partial \theta^1 \partial \theta^2} \cdot \mathbf{a}_1 &= -b_{22}b_{11} \\ &= \frac{\partial}{\partial \theta^2}\left(\frac{1}{2a_{11}}\frac{\partial a_{11}}{\partial \theta^2}\right)a_{11} + \frac{1}{2a_{11}}\frac{\partial a_{11}}{\partial \theta^2}\frac{1}{2}\frac{\partial a_{11}}{\partial \theta^2} - (b_{12})^2 \\ &= \frac{\partial}{\partial \theta^2}\left(\frac{1}{2\sqrt{a_{11}}}\frac{\partial a_{11}}{\partial \theta^2}\right)\sqrt{a_{11}} - (b_{12})^2 = \frac{\partial^2 \sqrt{a_{11}}}{(\partial \theta^2)^2}\sqrt{a_{11}} - (b_{12})^2.\end{aligned}$$

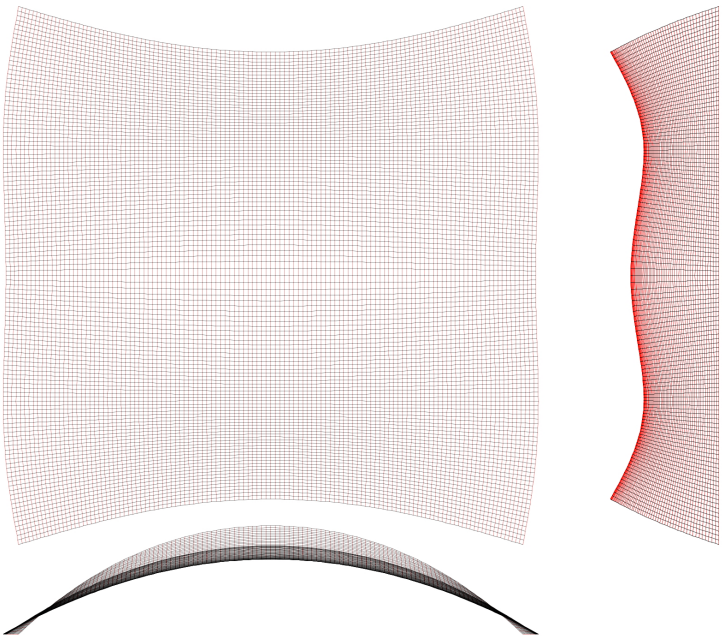

Figure 4: Plan and elevations of a brick shell supported on two edges at ground level and free along the other two. Geodesic lines are shown in black

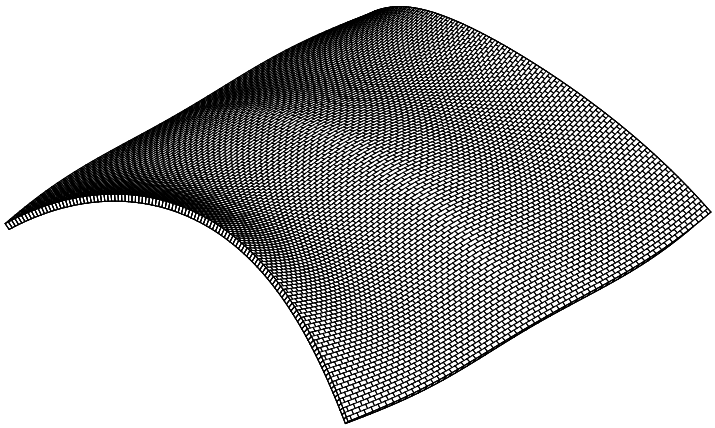

Figure 5: View showing brick pattern

Therefore

$$K = \frac{b_{11}b_{22} - (b_{12})^2}{a_{11}a_{22} - (a_{12})^2} = -\frac{1}{a_{22}\sqrt{a_{11}}} \frac{\partial^2 \sqrt{a_{11}}}{(\partial \theta^2)^2} \tag{11}$$

or

$$K = -\frac{1}{w} \frac{\partial^2 w}{\partial s^2} \tag{12}$$

where w is the spacing between geodesics and s is the distance measured along a geodesic. K is the

Gaussian curvature, that is the product of the two principal curvatures. On a *developable* surface, such as a plane or a cylinder, $K = 0$ so that Cartesian coordinates and polar coordinates are special cases of geodesic coordinates.

4. Numerical implementation of equilibrium combined with geodesic coordinates

Figures 4 and 5 show the results of the numerical implementation. The algorithm ensures that

1. The black lines in figure 4 are geodesics,
2. the spacing of the red lines along the black lines is constant and
3. the shell is in equilibrium under a uniform vertical load per unit plan area with the principal stresses parallel to the coordinate curves.

The structure is modelled by a system of pin ended members that are all in compression under downwards vertical load. The equations are solved using dynamic relaxation and rather than writing out many lines of equations or pseudocode, we consider it better to include the entire code so that readers can run the code for themselves if they so wish. The reason for this is that the code contains a number of features which we found were necessary for stability, but are difficult to explain in their entirety. For example the variable 'loadIncrementFactor' is there to ensure that the load does not vary too quickly as the spacing between geodesics changes. We also found that the relaxation factors for the different equations have to be 'tuned' so that the algorithm converges. Because the structure is in compression it is unstable with negative force densities or tension coefficients. This means that in the dynamic relaxation the nodes are moved in the *opposite* direction to the out of balance force, which is equivalent to having a negative mass.

The program is written in the Processing language `www.processing.org` which is based on Java, but the code would be essentially the same in C++ or C# for inclusion in some other graphic environment such as OpenGL or Grasshopper.

5. Geodesic coordinates on arbitrary free form shells

There are infinite number of ways of constructing a geodesic coordinate system on a smooth surface [7]. This suggests that one can separate the form finding and pattern generation to allow freedom to design a brick pattern on an arbitrary surface. By generating the geodesics orthogonal to an arbitrary surface curve, i.e. becoming the first orthogonal trajectory, it should be easy to generate the rest of orthogonal trajectories by measuring equal lengths along the geodesics. Our method for generating geodesics are based on a series of circles, having equal radius R, with its centre on the surface contained with the normal plane of its centre. Each row of overlapping circles are connected in such a way that they intersect the surface creating a locus of equally spaced points describing a geodesic, see figure 7. Since all points are contained within the normal planes the curvature only contains a normal component, i.e. the geodesic curvature is zero. The initial orthogonal trajectory is arbitrary and therefore this strategy allows to specify the direction of the brick pattern but also to make several local patterns combined into a global pattern. For a pattern to be successful the geodesics must not cross, which we can only guess in advance, and therefore it might even be necessary to divide a surface into several patches each with its own pattern. Our method was implemented and applied to different surfaces and is shown in figure 6. The script was developed in C# using the Rhinocommon SDK [1] to solve surface and curve intersections.

6. Conclusions

We have developed two different strategies for generating brick patterns which satisfies the requirement for a constant spacing between the bed joints, each having their own quality in the design. The first is

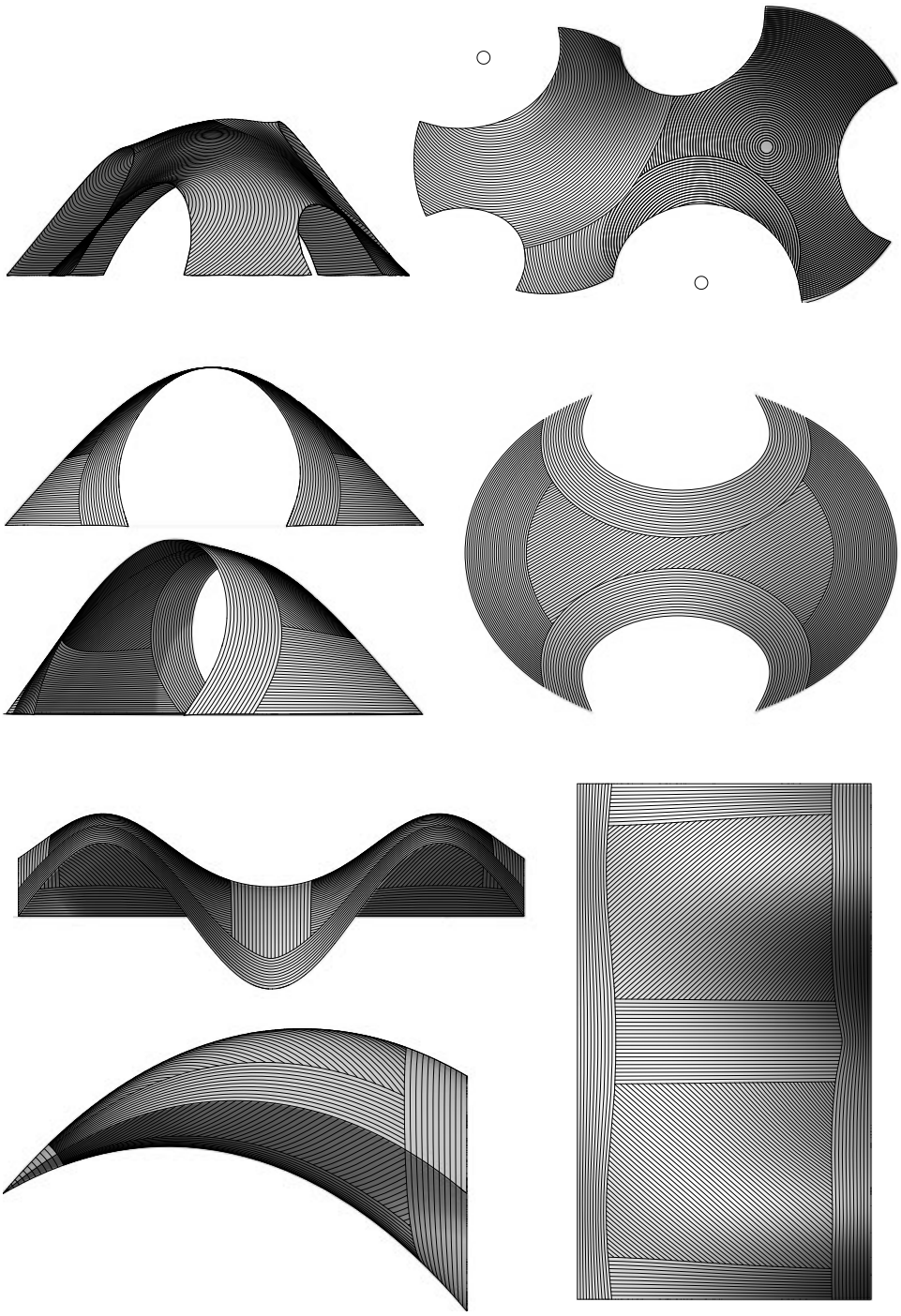

Figure 6: Plan and elevations of three different brick shells generated by our strategy described in section 5. The global pattern can consist of one or more variations of geodesic coordinates on the surface. Since the form and pattern generation is separated it is easy to adapt to edge conditions or just making unique patterns. Depending on how the initial orthogonal trajectory is constructed the pattern can take any direction. Closed initial curves results in a variation called polar geodesic coordinates, the top figure consists of three polar geodesic patterns.

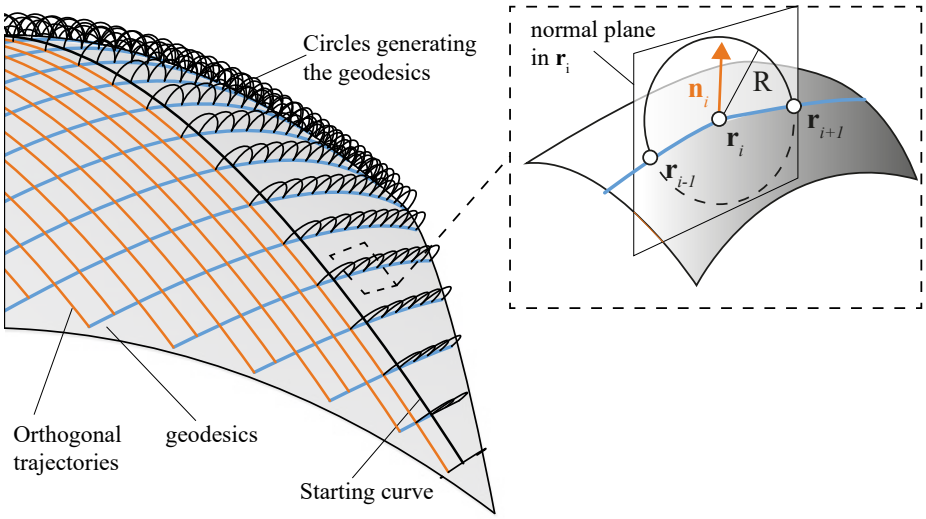

Figure 7: The generation of the geodesic coordinates is based on a initial curve from which geodesics are generated at an right angle. A series of circles with its centre on the surface, contained in the normal plane in its centre point, intersects the surface generating a locus of points describing the geodesics. Since the points on the geodesics are contained within a normal planes the geodesic curvature is zero.

derived from desired geometrical and structural properties making form, structural action and pattern integrated. This avoids sliding along the head joints but it also tells a unique design story. The second strategy disconnect the form and pattern making it easy to apply unique patterns based on visual appeal, structural or production benefits. It can be a tool for enabling conversations and cooperation between different professions in a design process.

We have not discussed the spacing of the head joints and in figure 5 the length of the bricks is shown to vary as the spacing between the geodesics varies. However it is more likely that constant length bricks would be used so that the head joints would not coincide with the geodesics.

References

[1] R. M. . Associates. Rhinocommon sdk. `http://developer.rhino3d.com/api/RhinoCommon/`, 2017. [Online; accessed 1-April-2017].

[2] P. Block and J. Ochsendorf. Thrust network analysis: A new methodology for three-dimensional equilibrium. *Journal of the International Association for Shell and Spatial Structures*, 48:167–173, 2007.

[3] P. de L'Orme. *Le premier tome de l' Architecture de Philibert de de l'Orme*. chez Federic Morel, 1567. URL `http://architectura.cesr.univ-tours.fr/traite/Images/Les1653Index.asp`.

[4] A. E. Green and W. Zerna. *Theoretical elasticity*. Oxford University Press and Dover, 2nd edition, 1968 and 1992.

[5] E. Happold and W. Liddell. Timber lattice roof for the Mannheim Bundesgartenschau. *The Structural Engineer*, 53:99–135, 1975.

[6] G. Paulsson. *Hantverkets Bok: 4, Mureri*. Lindfors bokförlag, 1936.

[7] D. J. Struik. *Lectures on classical differential geometry*. Addison - Wesley and Dover, 2nd edition, 1961 and 1988.

Appendix - Computer code listing

```
double[][][] coord, velocity, force, tangent, inPlaneNormal;
double[][] geodesicForceDen, geodesicForceDenRate;
double[][] orthogonalForceDen, orthogonalForceDenRate;
double[][] stiffness, load;
int m, n, saveCycle;
double a, bedHeight, lengthSquared, loadFactor;
boolean start;
void setup()
{
  m =  60;
  n = 120;
  coord          = new double[m + 1][n + 1][3];
  velocity       = new double[m + 1][n + 1][3];
  force          = new double[m + 1][n + 1][3];
  tangent        = new double[m + 1][n + 1][3];
  inPlaneNormal  = new double[m + 1][n + 1][3];
  stiffness      = new double[m + 1][n + 1];
  load           = new double[m + 1][n + 1];
  geodesicForceDen        = new double[m + 1][n];
  geodesicForceDenRate    = new double[m + 1][n];
  orthogonalForceDen      = new double[m][n + 1];
  orthogonalForceDenRate  = new double[m][n + 1];
  fullScreen();
  a = 5000.0;
  bedHeight = 0.25 * a / (double) n;
  lengthSquared = bedHeight * bedHeight;
  loadFactor = - 5.0 / (double) n;
  for (int i = 0; i <= m; i ++)
  {
    for (int j = 0; j <= n; j ++)
    {
      double factor1 = (double)(i - m) / (double) n;
      double factor2 = 0.3 * (double)(2.0 * j - n) / (double) (2 * n);
      coord[i][j][0] = 0.3 * factor1 * a;
      coord[i][j][1] = (factor2 - 0.5 * (0.5 * factor1 * factor1
        + 0.03 * Math.cos(1.5 * PI * factor1 * (double) n / (double) m)) * factor2) * a;
      coord[i][j][2] = 0.2 * a * (double)(j * (n - j))
        * (1.0 + 1.0 * factor1 * factor1) / (double) (n * n);
      for (int xyz = 0; xyz <= 2; xyz ++)velocity[i][j][xyz] = 0.0;
    }
  }
  for (int i = 0; i <= m; i ++)
  {
    for (int j = 0; j <= n - 1; j ++)
    {
      geodesicForceDen[i][j] = - 3.0;
      geodesicForceDenRate[i][j] = 0.0;
    }
  }
  for (int i = 0; i <= m - 1; i ++)
  {
    for (int j = 0; j <= n; j ++)
    {
      orthogonalForceDen[i][j] = 0.0;
      orthogonalForceDenRate[i][j] = 0.0;
    }
  }
  start = true;
  saveCycle = 0;
}
void draw()
{
  for (int cycle = 0; cycle <= 1000; cycle++)
  {
    for (int i = 0; i <= m; i ++)
    {
      for (int j = 0; j <= n; j ++)
      {
        for (int xyz = 0; xyz <= 2; xyz ++)force[i][j][xyz] = 0.0;
        stiffness[i][j] = 0.0;
      }
    }
    for (int i = 0; i <= m; i ++)
    {
      for (int j = 0; j <= n; j ++)
      {
        int previous_i = i - 1;
        int previous_j = j - 1;
        if (previous_i < 0)previous_i = 0;
        if (previous_j < 0)previous_j = 0;
        int next_i = i + 1;
```

```
      int next_j = j + 1;
      if (next_i > m)next_i = m;
      if (next_j > n)next_j = n;
      double scalarProduct = 0.0;
      double magnitudeSq = 0.0;
      for (int xyz = 0; xyz <= 2; xyz ++)
      {
        tangent[i][j][xyz] = coord[i][next_j][xyz] - coord[i][previous_j][xyz];
        magnitudeSq += tangent[i][j][xyz] * tangent[i][j][xyz];
      }
      double magnitude = Math.sqrt(magnitudeSq);
      for (int xyz = 0; xyz <= 2; xyz ++)tangent[i][j][xyz] /= magnitude;
      for (int xyz = 0; xyz <= 2; xyz ++)
      {
        inPlaneNormal[i][j][xyz] = coord[next_i][j][xyz] - coord[previous_i][j][xyz];
        scalarProduct += inPlaneNormal[i][j][xyz] * tangent[i][j][xyz];
      }
      for (int xyz = 0; xyz <= 2; xyz ++)
        inPlaneNormal[i][j][xyz] -= tangent[i][j][xyz] * scalarProduct;
      magnitudeSq = 0.0;
      for (int xyz = 0; xyz <= 2; xyz ++)
        magnitudeSq += inPlaneNormal[i][j][xyz] * inPlaneNormal[i][j][xyz];
      magnitude = Math.sqrt(magnitudeSq);
      for (int xyz = 0; xyz <= 2; xyz ++)inPlaneNormal[i][j][xyz] /= magnitude;
      double correctLoad = loadFactor * magnitude;
      double loadIncrementFactor = 1.0e-6;//This is to stop the load fluctuating too quickly
      if (start == true)load[i][j] = correctLoad;
      else
        load[i][j] = (1.0 - loadIncrementFactor) * load[i][j] + loadIncrementFactor * correctLoad;
      if (i != 0 && i != m)force[i][j][2] += load[i][j];
      else force[i][j][2] += 2.0 * load[i][j];
    }
  }
  if (start)start = false;
  for (int i = 0; i <= m - 1; i ++)
  {
    for (int j = 1; j <= n - 1; j ++)
    {
      double scalarProduct = 0.0;
      for (int xyz = 0; xyz <= 2; xyz ++)scalarProduct +=
        (coord[i][j + 1][xyz] - 2.0 * coord[i][j][xyz] + coord[i][j - 1][xyz])
        * inPlaneNormal[i][j][xyz];
      double carryOver = 0.9;
      double change = - 0.0005 * scalarProduct;
      orthogonalForceDenRate[i][j] =  carryOver * orthogonalForceDenRate[i][j] + change;
      orthogonalForceDen[i][j] += orthogonalForceDenRate[i][j];
      if (orthogonalForceDen[i][j] > 0.0)orthogonalForceDen[i][j] = 0.0;
      if (i != 0)
      {
        orthogonalForceDenRate[i - 1][j] = carryOver * orthogonalForceDenRate[i - 1][j] - change;
        orthogonalForceDen[i - 1][j] += orthogonalForceDenRate[i - 1][j];
        if (orthogonalForceDen[i - 1][j] > 0.0)orthogonalForceDen[i - 1][j] = 0.0;
      }
    }
  }
  for (int i = 0; i <= m - 1; i ++)
  {
    for (int j = 1; j <= n - 1; j ++)
    {
      for (int xyz = 0; xyz <= 2; xyz ++)
      {
        double delta = coord[i + 1][j][xyz] - coord[i][j][xyz];
        double component = orthogonalForceDen[i][j] * delta;
        force[i][j][xyz] += component;
        if (i != m - 1)
          force[i + 1][j][xyz] -= component;
        else
        {
          if (xyz != 0)force[i + 1][j][xyz] -= 2.0 * component;
        }
      }
      stiffness[i][j] += orthogonalForceDen[i][j];
      if (i != m - 1)stiffness[i + 1][j] += orthogonalForceDen[i][j];
      else stiffness[i + 1][j] += 2.0 * orthogonalForceDen[i][j];
    }
  }
  for (int i = 0; i <= m; i ++)
  {
    for (int j = 0; j <= n - 1; j ++)
    {
      double thisLengthSquared = 0.0;
      for (int xyz = 0; xyz <= 2; xyz ++)
      {
        double delta = coord[i][j + 1][xyz] - coord[i][j][xyz];
        thisLengthSquared += delta * delta;
        geodesicForceDenRate[i][j] = 0.5 * geodesicForceDenRate[i][j]
          - 0.001 * (thisLengthSquared - lengthSquared) / lengthSquared;
        geodesicForceDen[i][j] += geodesicForceDenRate[i][j];
        double component = geodesicForceDen[i][j] * delta;
        force[i][j][xyz] += component;
```

```
          force[i][j + 1][xyz] -= component;
        }
        stiffness[i][j] += geodesicForceDen[i][j];
        stiffness[i][j + 1] += geodesicForceDen[i][j];
      }
    }
    for (int i = 0; i <= m; i ++)
    {
      for (int j = 1; j <= n - 1; j ++)
      {
        for (int xyz = 0; xyz <= 2; xyz ++)
        {
          if (i != m || xyz != 0)
          {
            velocity[i][j][xyz] = 0.95 * velocity[i][j][xyz] + 0.2 * force[i][j][xyz] / stiffness[i][j];
            coord[i][j][xyz] += velocity[i][j][xyz];
          }
        }
      }
    }
  }
  background(255, 255, 255);
  scale(2.7 * (float)height / (float)a);
  float leftRight = 0.2 * (float)a;
  float elevDrop = 0.36 * (float)a;
  float planDrop = 0.16 * (float)a;
  float elevRight = 0.45 * (float)a;
  strokeWeight(1);
  translate(leftRight, planDrop);
  stroke(255.0, 0.0, 0.0, 150.0);
  for (int j = 0; j <= n; j ++)
  {
    for (int i = 0; i <= m - 1; i ++)
    {
      line((float)coord[i][j][1], (float)coord[i][j][0], (float)coord[i + 1][j][1], (float)coord[i + 1][j][0]);
      line((float)coord[i][j][1], -(float)coord[i][j][0], (float)coord[i + 1][j][1], -(float)coord[i + 1][j][0]);
    }
  }
  stroke(0.0, 0.0, 0.0, 150.0);
  for (int i = 0; i <= m; i ++)
  {
    for (int j = 0; j <= n - 1; j ++)
    {
      line((float)coord[i][j][1], (float)coord[i][j][0], (float)coord[i][j + 1][1], (float)coord[i][j + 1][0]);
      if (i != m)
        line((float)coord[i][j][1], -(float)coord[i][j][0], (float)coord[i][j + 1][1], -(float)coord[i][j + 1][0]);
    }
  }
  translate(0.0, elevDrop - planDrop);
  stroke(255.0, 0.0, 0.0, 150.0);
  for (int j = 0; j <= n; j ++)
  {
    for (int i = 0; i <= m - 1; i ++)
      line((float)coord[i][j][1], -(float)coord[i][j][2], (float)coord[i + 1][j][1], -(float)coord[i + 1][j][2]);
  }
  stroke(0.0, 0.0, 0.0, 150.0);
  for (int i = 0; i <= m; i ++)
  {
    for (int j = 0; j <= n - 1; j ++)
      line((float)coord[i][j][1], -(float)coord[i][j][2], (float)coord[i][j + 1][1], -(float)coord[i][j + 1][2]);
  }
  translate(elevRight - leftRight, planDrop - elevDrop);
  stroke(255.0, 0.0, 0.0, 150.0);
  for (int j = 0; j <= n; j ++)
  {
    for (int i = 0; i <= m - 1; i ++)
    {
      line(-(float)coord[i][j][2], (float)coord[i][j][0], -(float)coord[i + 1][j][2], (float)coord[i + 1][j][0]);
      line(-(float)coord[i][j][2], -(float)coord[i][j][0], -(float)coord[i + 1][j][2], -(float)coord[i + 1][j][0]);
    }
  }
  stroke(0.0, 0.0, 0.0, 150.0);
  for (int i = 0; i <= m; i ++)
  {
    for (int j = 0; j <= n - 1; j ++)
    {
      line(-(float)coord[i][j][2], (float)coord[i][j][0], -(float)coord[i][j + 1][2], (float)coord[i][j + 1][0]);
      if (i != m)
        line(-(float)coord[i][j][2], -(float)coord[i][j][0], -(float)coord[i][j + 1][2], -(float)coord[i][j + 1][0]);
    }
  }
}
```