A Simple Mesh Generator in *Mathematica*

Version 0.1

Zhe Hu
huzhe@iit.edu
Illinois Institute of Technology
September, 2004

In this notebook we experiment with Persson and Strang's mesh generation algorithm in *Mathematica*. The algorithm was implemented in MATLAB in their original paper [1], which is highly recommended to be read along with this notebook.

Introduction

This *Mathematica* notebook is an effort to transcribe the MATLAB code of a 2-D mesh generation algorithm as described explicitly in Persson and Strang's paper [1]. The goal is to make the algorithm executable in *Mathematica* so that its users can also experiment with the algorithm.

Since the algorithm was expressed very clearly from their original paper [1] including the MATLAB code, which is a perfect example of literate programming in MATLAB, it is pretty easy to translate the MATLAB code "literally" into *Mathematica.* Such translation is virtually always possible in either directions even without human interference. And such a Rosetta Stone kind of translation might be useful if one species of people coding in either MATLAB or *Mathematica* were to disappear, future generations would still be able to rediscover one programming language by reading its interpretation in the other one.

However, it is so tempting to present the literate programming capability of *Mathematica* by following its general principles; that is, (a) documentation mingles with code and both get pretty-printed; (b) shuffle code pieces for human readability. I decided to transcribe the code manually.

The original MATLAB code was documented as 8 steps (sections) in sequential order, which is easy to follow because the ideas behind the code were explained beforehand in early parts of the paper. So it is recommended that you read part 1 and 2 of the original paper. Instead of following the MATLAB code literally in 8 steps, this notebook breaks the code pieces apart and examines each of them separately.

Overview

In order to generate a triangular mesh, in the 2-D case, one needs to find locations of the meshpoints $\boldsymbol{p}(x, y)$. Then both MATALB and *Mathematica* have built-in Delaunay triangulation function of generating the triangular mesh from these meshpoints.

In Persson and Strang's algorithm [1], user specifies the relative size of the mesh triangles by the size function $h(x, y)$. In order to generate mesh triangles as well as the edges of them (meshbars) according to $h(x, y)$, an objective function using "force" analogy is proposed, in which the meshbars act like springs exerting forces that move the meshpoints around until an equilibrium (zero force) is reached. This "force" related objective function is difficult to minimize (or zero) because it is not continuous as stated in the paper, "The force vector $\boldsymbol{F}(\boldsymbol{p})$ is not a continuous function of $\boldsymbol{p}$, since the topology (the presence or absence of connecting bars) is changed by Delaunay as the points move."
So the algorithm, in general, turns into an iterative process of minimizing the objective function

```
NestWhile[moveMeshpoints, initpoints, !(goodEnough[#])&]
```

The initial locations of the meshpoints can be randomly distributed or chosen according to $h(x, y)$. Since $h(x, y)$ is a relative measure, one doesn't need to define the exact size of each mesh triangle at location (x, y).
The function `moveMeshpoints` keeps changing the locations of meshpoints so as to minimize this "force" related objective function until the result is regarded by function `goodEnough` to be so.

In practice, meshpoints always have to reside within certain geometric region. User also needs to define this bounded region by a distance function $d(x, y)$. And one difficult task is to maintain the meshpoints within the 2-D region (whether it is a disk or a rectangle with a hole in it) during the iterative process. After each move, if some meshpoints are pushed out of the 2-D region, they need to be brought back onto its boundary, which is carried out using this distance function $d(x, y)$.

Using the Distance Function $d(x, y)$

Since distance function determines the shape for the algorithm, Let's start from it. Each distance function returns negative for points within the defined region; positive or zero otherwise. Here is a distance function for a disk, bounded by the circle.

```
dcircle[{x_, y_}, {cx_, cy_, r_}] := (x - cx)^2 + (y - cy)^2 - r^2
```

Here is another example of a rectangular region, defined by the lower left point (x_1, y_1) and the upper right point (x_2, y_2). (This is the typical mathematical way of defining coordinates for a rectangular region, which is different from the one for drawing rectangles on the computer screen.)

```
drectangle[{x_, y_}, {x1_, y1_, x2_, y2_}] :=
 -Min[Min[Min[-y1 + y, y2 - y], -x1 + x], x2 - x]
```

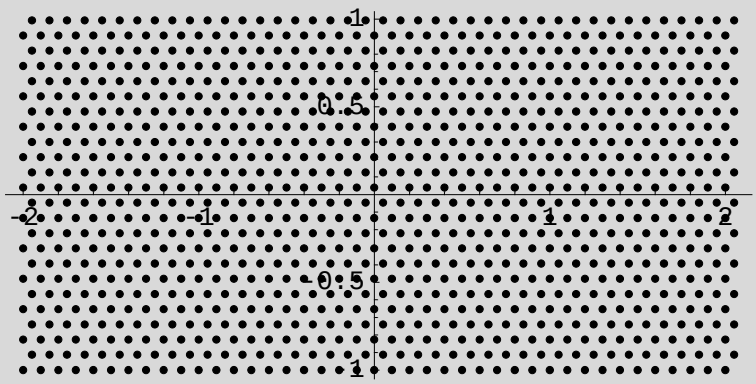
To test if these distance functions can distinguish points within or out of the defined region, we start by seeding meshpoints in a rectangular region, still defined by the lower left point (x_1, y_1) and the upper right point (x_2, y_2). And

points in every other row are shifted so that the distance between every two points is h_0 (corresponding to *step 1* in the paper).

```
seedMeshpoints[{x1_, y1_, x2_, y2_}, h0_] :=
 Table[{x + (1 - (-1)^Quotient[y-y2,h0 Sqrt[3]/2])/4 h0, y},
   {y, y1, y2, Sqrt[3]/2 h0}, {x, x1, x2, h0}] // Flatten[#, 1] &
```

Try to seed meshpoints in a rectangle between $(-2, -1)$ and $(2, 1)$ with distance 0.1 between every two meshpoints.

```
seedMeshpoints[{-2, -1, 2, 1}, 0.1] // ListPlot[#,
   AspectRatio → Automatic, PlotStyle -> PointSize[0.01]] &
```



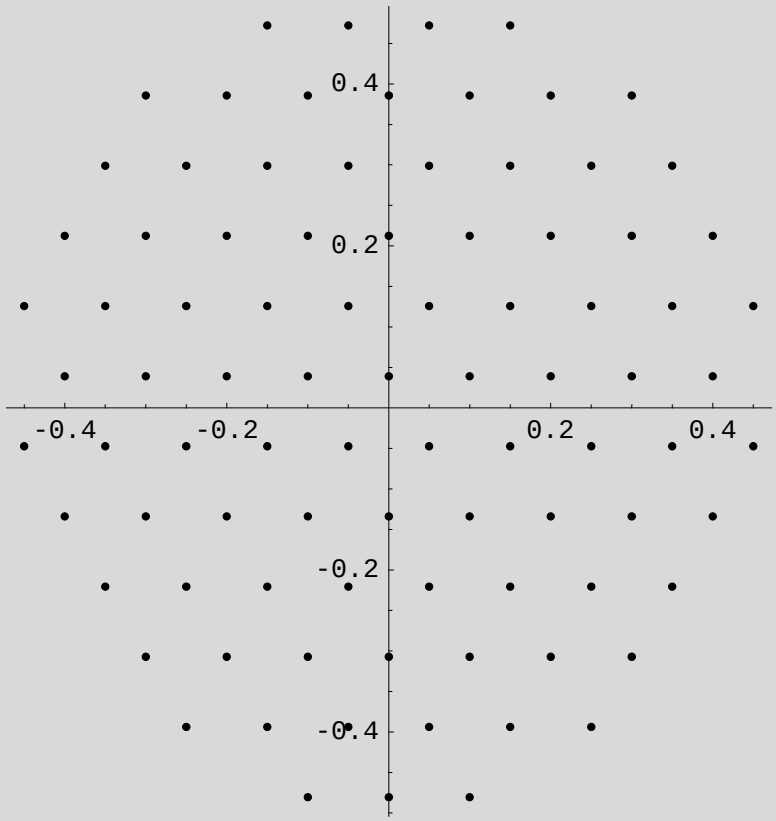

```
- Graphics -
```

Function `selectMeshpoints` use the distance function to select only meshpoints within the defined region (corresponding to part of *step 2* in the paper).

```
selectMeshpoints[d_, {x1_, y1_, x2_, y2_}, h0_] :=
  Select[seedMeshpoints[{x1, y1, x2, y2}, h0], d[#] < 0 &];
```

The distance function `dcircle` selects meshpoints within a disk.

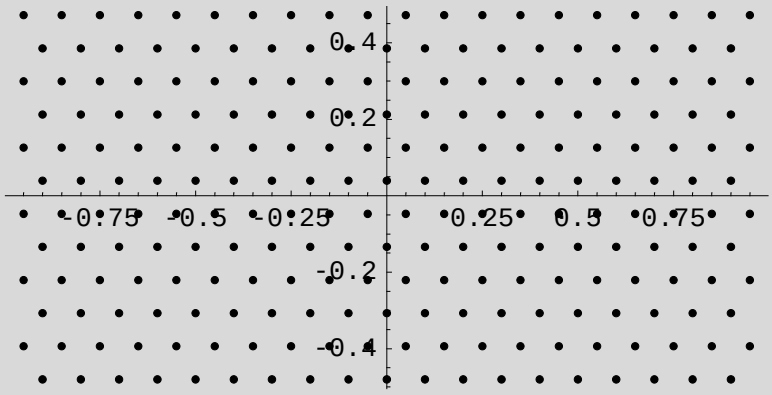
```
selectMeshpoints[dcircle[#, {0, 0, 0.5}] &,
  {-2, -1, 2, 1}, 0.1] // ListPlot[#,
   AspectRatio → Automatic, PlotStyle -> PointSize[0.01]] &
```



- Graphics -

The distance function `drectangle` selects meshpoints within a rectangle. You may notice from its definition that the four corner points may not be included. This glitch can be fixed later.

```
selectMeshpoints[drectangle[#, {-1, -0.5, 1, 0.5}] &,
  {-2, -1, 2, 1}, 0.1] // ListPlot[#,
   AspectRatio → Automatic, PlotStyle -> PointSize[0.01]] &
```



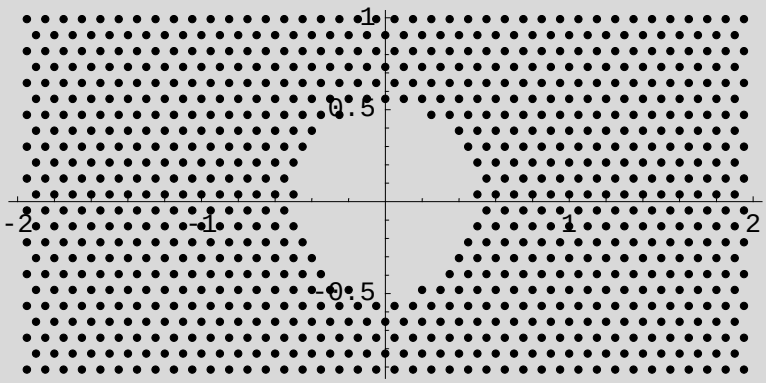
- Graphics -

How to define the distance function of a rectangle with a hole in the center? There are ways to combine distance functions of simple geometric shapes to form more complex ones. For example, `ddiff` takes one shape out of the other.

```
ddiff[d1_, d2_] := Max[d1, -d2]
```

For more ways of combining distance functions, see *Figure 4.1* in the paper. Here is a disk taken out of a rectangle

```
selectMeshpoints[ddiff[drectangle[#, {-2, -1, 2, 1}],
    dcircle[#, {0, 0, 0.5}]] &, {-2, -1, 2, 1}, 0.1] //
 ListPlot[#, AspectRatio → Automatic,
   PlotStyle -> PointSize[0.01]] &
```




```
- Graphics -
```

There are two other places where the distance function is used. One is to select interior meshbars (discussed later). The other is to move meshpoints back onto the boundary (*step 7*), if they were pushed out.

The closest boundary point to an outsider is along the gradient direction of the distance function. The function `NGrad` calculates numerically the gradient for a given distance function at a specific point.

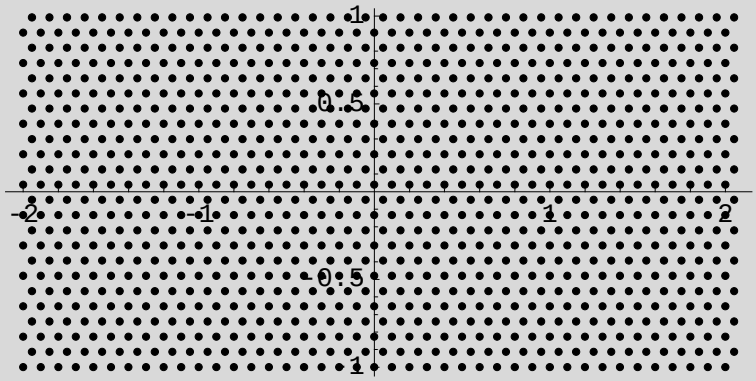
```
NGrad[d_, {x0_, y0_}] :=
 {ND[d[{x, y0}], x, x0], ND[d[{x0, y}], y, y0]}
```

Then the function `backtoBoundary` moves an outsider onto the closest boundary point. (`FindRoot` is used to find the boundary point, which is a little bit different from *step 7* in paper.)

```
backtoBoundary[d_, {x_, y_}] :=
 Module[{expr}, expr = {x, y} - s NGrad[d, {x, y}];
  expr /. (FindRoot[d[expr], {s, 0}])]
```

To test this function, let's move all these points in a rectangle onto a circular boundary.

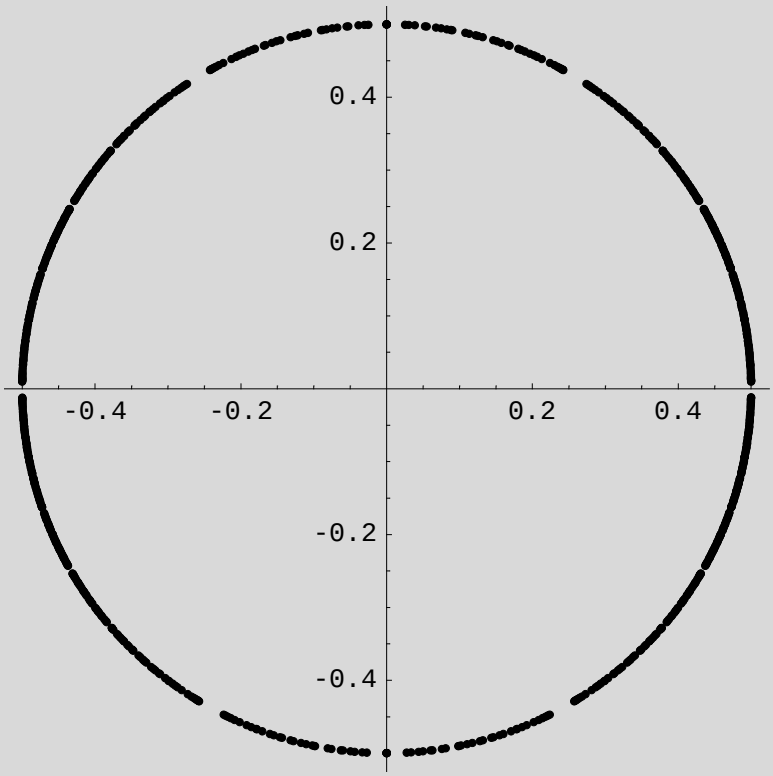
```
(pointseverywhere = seedMeshpoints[{-2, -1, 2, 1}, 0.1]) //
 ListPlot[#, AspectRatio → Automatic,
   PlotStyle -> PointSize[0.01]] &
```




```
- Graphics -
```

It's going to be very crowded.

```
(backtoBoundary[dcircle[#, {0, 0, 0.5}] &, #] & /@
   pointseverywhere) // ListPlot[#,
   AspectRatio → Automatic, PlotStyle -> PointSize[0.01]] &
```



- Graphics -

Well, it is a bit of abuse, since normally the function is only used to move points out of the region back to its boundary.

Distributing as the Size Function *h*(*x*, *y*)

The other important function in the algorithm is $h(x, y)$. It is actually the goal of the algorithm to make the size of mesh triangles distributed, in 2-D case, as $h(x, y)$. The simplest case would be $h(x, y) = 1$, such that all the mesh triangles are of the same size. A more interesting example, the following definition ask for triangle sizes related by $\sqrt{x^2 + y^2}$, which means the closer they are to the center point $(0, 0)$, the smaller the triangle meshes are. It therefore asks for more meshpoints as they are closer to the center point $(0, 0)$. So the size function $h(x, y)$ determines the locations of meshpoints indirectly.

```
h[{x_, y_}] := 0.1 + 0.1 √(x² + y²)
```

To distribute meshpoints initially, for example, in a 2-D disk, we assign a value to each would-be meshpoint (x_i, y_i) by applying function $h(x, y)$ to it. Since $h(x, y)$ is a relative measure, there is a scaling process, i.e. to normalize all the values between 0 and 1. Such scaling process is essential and will appear again later (*step 6* in the paper). (The definition of function `scale` is a bit like MATLAB's style. It makes use of the "Listable" property of the function `Divide`.)

```
scale[x_List] := x / Max[x]
```

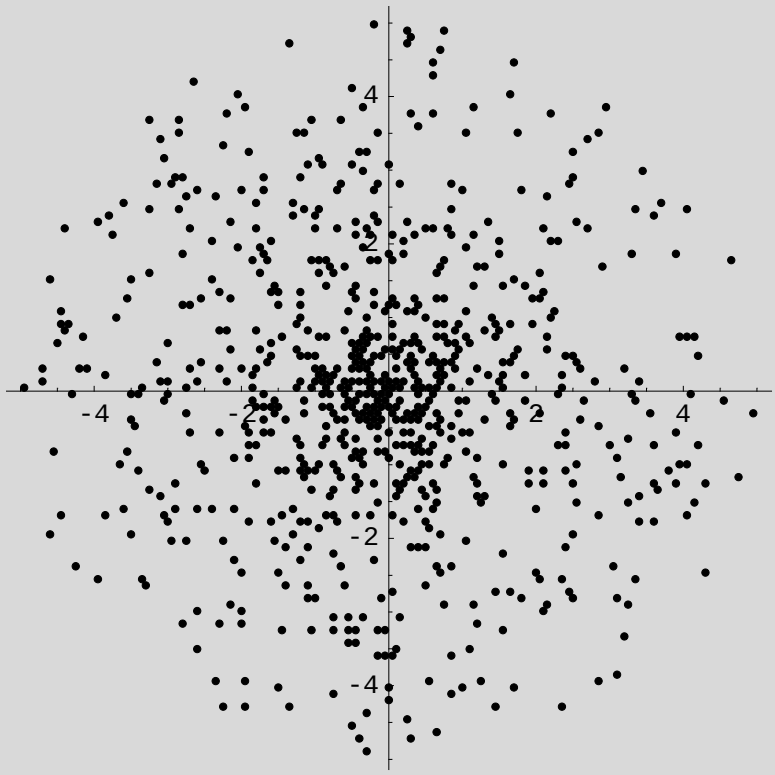
```
General::spell1 : Possible spelling error: new symbol
   name "scale" is similar to existing symbol "Scale". More…
```

To attach a normalized value for each meshpoint

```
pts = selectMeshpoints[
   dcircle[#, {0, 0, 5}] &, {-10, -10, 10, 10}, 0.1] ;
values = scale[(1 / h[#]^2 & /@ pts)] ;
(* a more compact expression can be
   values = Divide@@({#,Max[#]}&[1/h[#]^2&/@pts]) *)
```

In the algorithm, a lottery is drawn to select which meshpoints to survive. The attached values isn't actually $h(x, y)$ but $1/h(x, y)^2$ instead, as being the probability of "survival" (part of *step 2* in the paper). The meshpoint at locations (x_i, y_i) where $h(x_i, y_i)$ is smaller, has larger $1/h(x_i, y_i)^2$ value so that has higher probability to survive, therefore more meshpoints are located at where $h(x, y)$ demands smaller mesh triangles.

```
(p = Extract[pts, Position[values, _?(# > Random[] &)]]) //
 ListPlot[#, AspectRatio → Automatic,
   PlotStyle -> PointSize[0.01] ] &
```




```
- Graphics -
```

The above graphic certainly shows more points toward the center, doesn't it. From now on, the number of meshpoints in the list p is determined. No points will be drop out, but relocated while minimizing the "force".

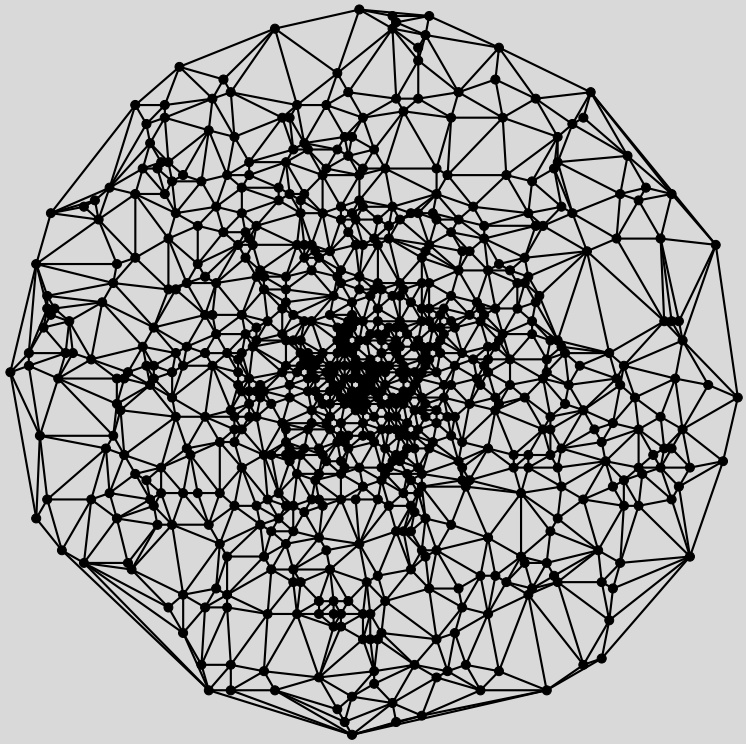
Delaunay Triangulation

Now we have a fixed number of meshpoints selected. Delaunay triangulation can begin (part of *step 3* in the paper)

```
t = DelaunayTriangulation[p];
```

The *Mathematica* function returns mesh triangles in a different format from MATLAB function's. It even has a function to plot the meshes directly. We can define such a function ourselves (*step 5* in the paper).

```
(* the Mathematica built-in function *)
PlanarGraphPlot[p, LabelPoints → False]
```



```
- Graphics -
```

But before we can plot these meshbars, we need to process the return values from `DelaunayTriangulation` to get the unique meshbars.

```
(* the input is directly
  from DelaunayTriangulation function*)
meshbar[t_] := Union[Sort /@
   (Thread[List[Sequence @@ #]] & /@ t // Flatten[#, 1] &)]
```

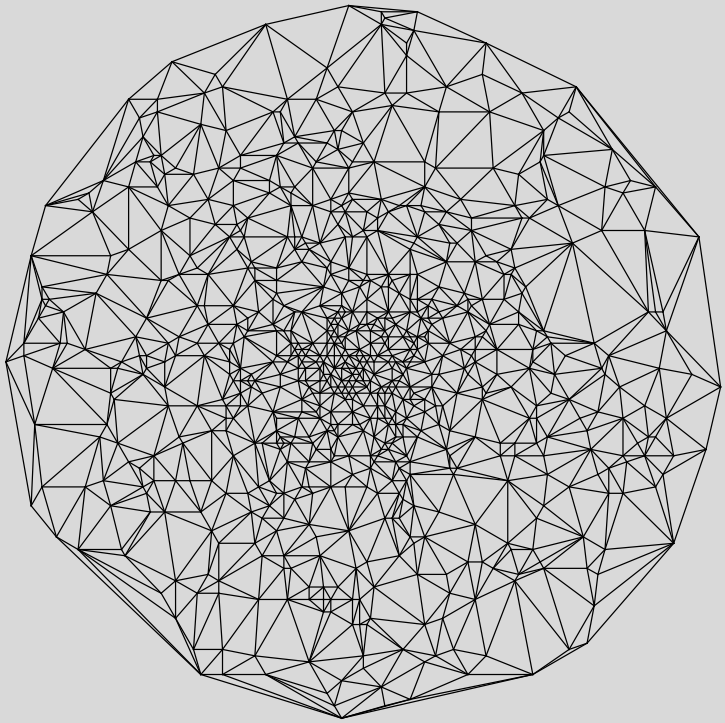
Get the unique bars (*step 4* in the paper)

```
bars = meshbar[t];
```

```
meshbarPlot[p_, bar_List] := Show[
  Graphics[Line /@ (Part[p, #] & /@ bar)], AspectRatio → Automatic]
```

Our version of "PlanarGraphPlot"

```
meshbarPlot[p, bars]
```

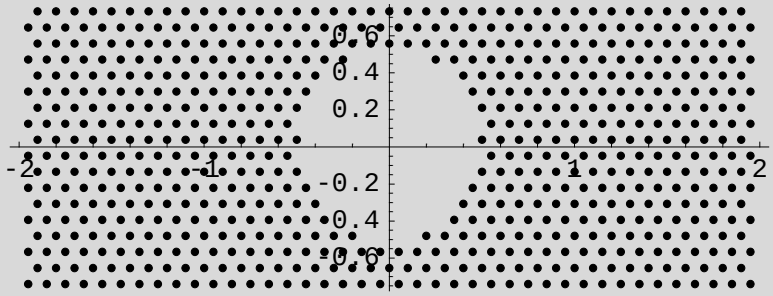


```
- Graphics -
```

As mentioned earlier, the distance function is used again to select meshbars that are interior to the defined region. For example, the following distance function defines a rectangle with a hole in the middle.

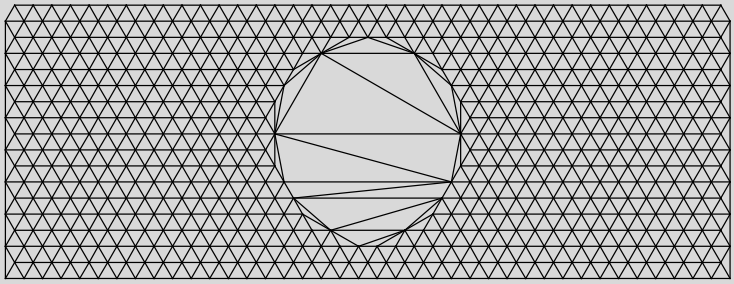
```
d[{x_, y_}] := ddiff[drectangle[{x, y}, {-2, -0.8, 2, 0.8}],
  dcircle[{x, y}, {0, 0, 0.5}]]
```

```
(p2 = selectMeshpoints[d, {-2, -1, 2, 1}, 0.1]) //
 ListPlot[#, AspectRatio → Automatic,
   PlotStyle -> PointSize[0.01]] &
```



- Graphics -

```
bars2 = meshbar[DelaunayTriangulation[p2]];
meshbarPlot[p2, bars2]
```



- Graphics -

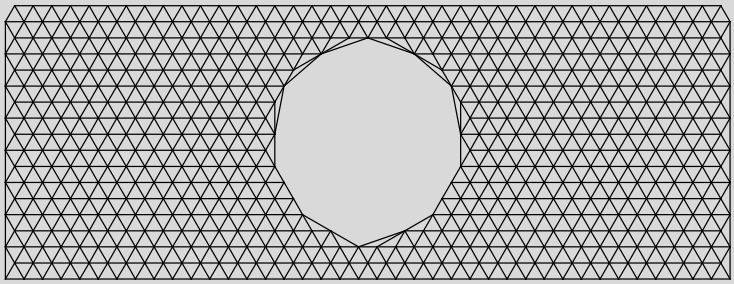
Obviously, these meshbars across the hole in the center need to be removed. Here we use a "middle point" approach that is different from the original MATLAB code. However the idea is the same; that is, using distance function to identify meshbars that lie out of the defined region.

$$\text{middlepoint}[p\_, \{a\_, b\_\}] := \frac{p[[a]] + p[[b]]}{2}$$

It selects only the meshbars interior to the region.

```
bars2 = Select[bars2, d[middlepoint[p2, #]] < 0 &];
```

```
meshbarPlot[p2, bars2]
```



```
- Graphics -
```

Now you may notice that two upper corner points of the rectangle are missing. This is caused by the definition of its distance function. To remedy for this, we need to add these points to our list of meshpoints list `p`. Now the list is completely initialized after putting in these missing meshpoints (if there is any). They are called fixed meshpoints because they won't move at all during the force equilibrium process.

Since now we have the meshbars, they will start exerting force on the connecting meshpoints to achieve an equilibrium. Let the force be with them.

Force be with them

These meshbars act like a special kind of springs that have the following force vs. length relationship

$$f(l,\ l_0) = \begin{cases} k(l - l_0) & \text{if } l < l_0, \\ 0 & \text{if } l \geq l_0 . \end{cases}$$

The function `barlength` calculates l (the actual meshbar length) in the above equation, given the two connecting meshpoints of a a meshbar.

```
barlength[p_, {a_, b_}] := Norm[p[[a]] - p[[b]]]
```

How about l_0? It looks like a desired length since $l = l_0$ will certainly turn force f into 0. Guess who involves in the calculation of l_0, size function $h(x, y)$ again. It is the second yet most important appearance of this function. It is essentially the core of the algorithm—the desired mesh triangle size yields the desired meshbar length. So the iterative process to zero the force is really a process to adjust the meshbar length to be close to l_0, which inevitably and implicitly adjusts the mesh triangles to the size and distribution as $h(x, y)$. The clever part is that still $h(x, y)$ gives a relative measure, so a scaling process with scaling factor $\left(\frac{\sum l_i^2}{\sum h(x_i, y_i)^2} \right)^{1/2}$ is used.

```
scale[a_List, {p_, bars_}] :=
  Sqrt[(Plus @@ Map[barlength[p, #]^2 &, bars])/(Plus @@ a^2)] * a
```

The parameter `Fscale=1.2` blows l_0 a little bit. Now let's return to our disk example. To find the desired length l_0 for each meshbar, we use the middle point of each meshbar as input to $h(x, y)$ and then scale the output.

```
l0 = With[{Fscale = 1.2},
   Fscale × scale[h[middlepoint[p, #]] & /@ bars, {p, bars}]];
```

```
General::spell1 : Possible spelling error: new symbol
   name "Fscale" is similar to existing symbol "scale". More…
```

Now assuming $k = 1$, the force can be calculated according to the "spring" relationship.

```
F = Max[#, 0] & /@ (l0 - (barlength[p, #] & /@ bars));
```

Now the force needs to be dissected into vectors of x-axis and y-axis components at each meshpoint so that they can be summed in these two independent directions.

```
Fvec = F (Subtract @@ p[[#]] / barlength[p, #] & /@ bars);
```

The MATLAB code (*step 6*) uses sparse matrix to sum forces (both x- and y-component) at each meshpoint. Here we try the same way by defining a sparse array in *Mathematica*. The sparse array is a square matrix of $n \times n$ dimensions, with n = number of meshpoints. Each row stores the forces for one meshpoint and the columns are forces (x- and y-component) exerted on it by neighboring meshpoints through connecting meshbars. Since one meshpoint only connects with a small fraction of the total number of meshpoints, the matrix is very sparse (i.e., many zeros in the matrix). A little trick (`Hold`) is used to store (F_x, F_y) at position (a, b) in the sparse array.

```
ForceRule[{a_, b_}, {Fx_, Fy_}] := {{a, b} → Hold[{Fx, Fy}],
  (* Newton's Third Law *)
  {b, a} → Hold[{-Fx, -Fy}]}
```

A sparse array, `forcematrix` is built from these rules

```
forcematrix = SparseArray[#, {Length[p], Length[p]}] &@
   (MapThread[ForceRule, {bars, Fvec}] // Flatten[#, 1] &);
```

Then the movements of the meshpoints are induced by the force acting on it (Aristotle-ly) according to the formula

$\boldsymbol{p}_{n+1} = \boldsymbol{p}_n + \Delta t\, \boldsymbol{F}(\boldsymbol{p}_n)$

We calculate the second term on the right hand side (with $\Delta t = 0.2$) by summing all forces on each row of the sparse array.

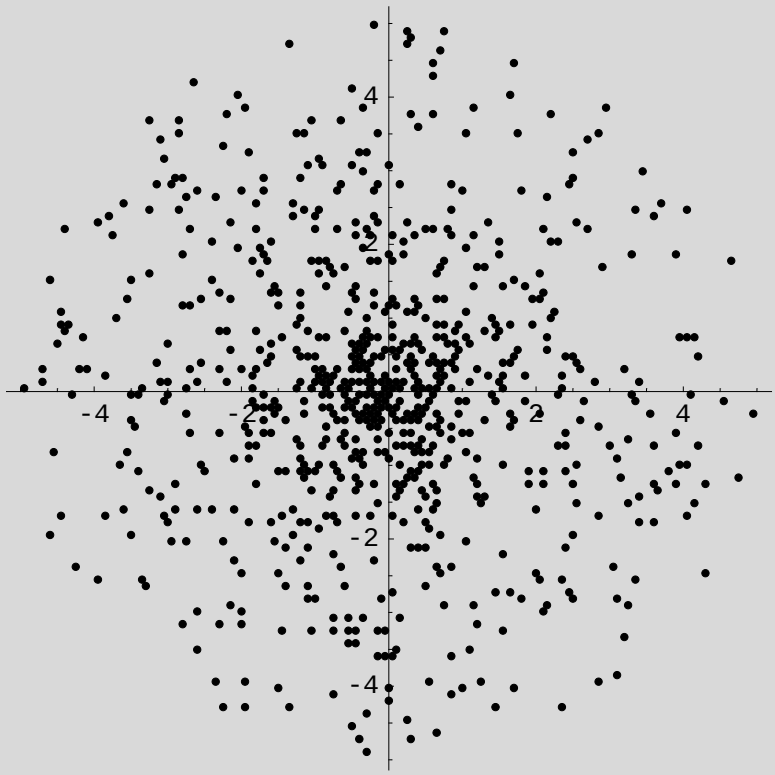
```
dp = With[{deltat = 0.2}, deltat × Table[
     Plus @@ (ReleaseHold /@ forcematrix[[i]]), {i, Length[p]}]];
```

So the movement of meshpoints

```
pp = p + dp;
```

As shown below, the meshpoints moves from the initial location

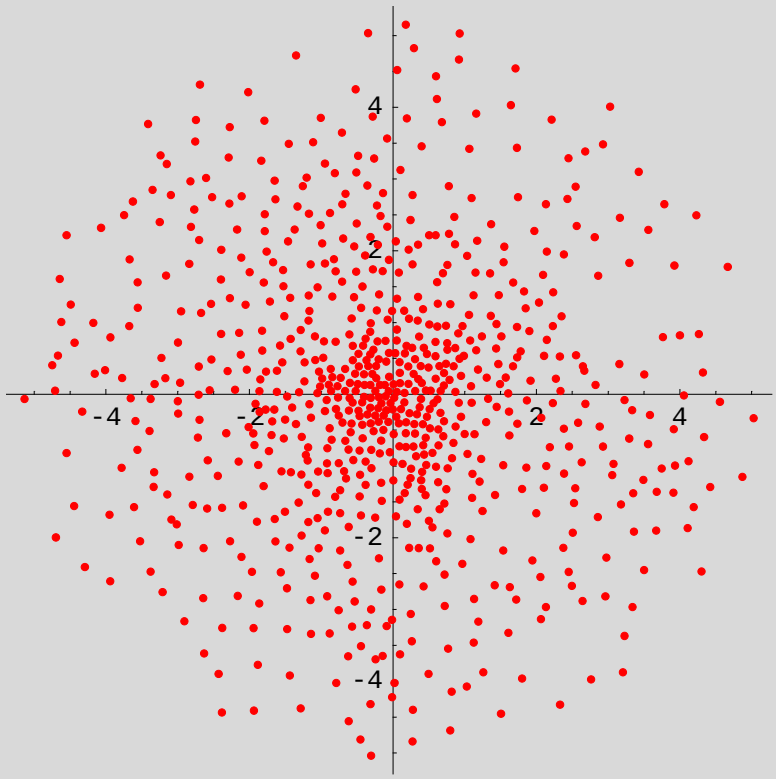
```
ListPlot[p, AspectRatio → Automatic,
 PlotStyle → PointSize[0.01]]
```



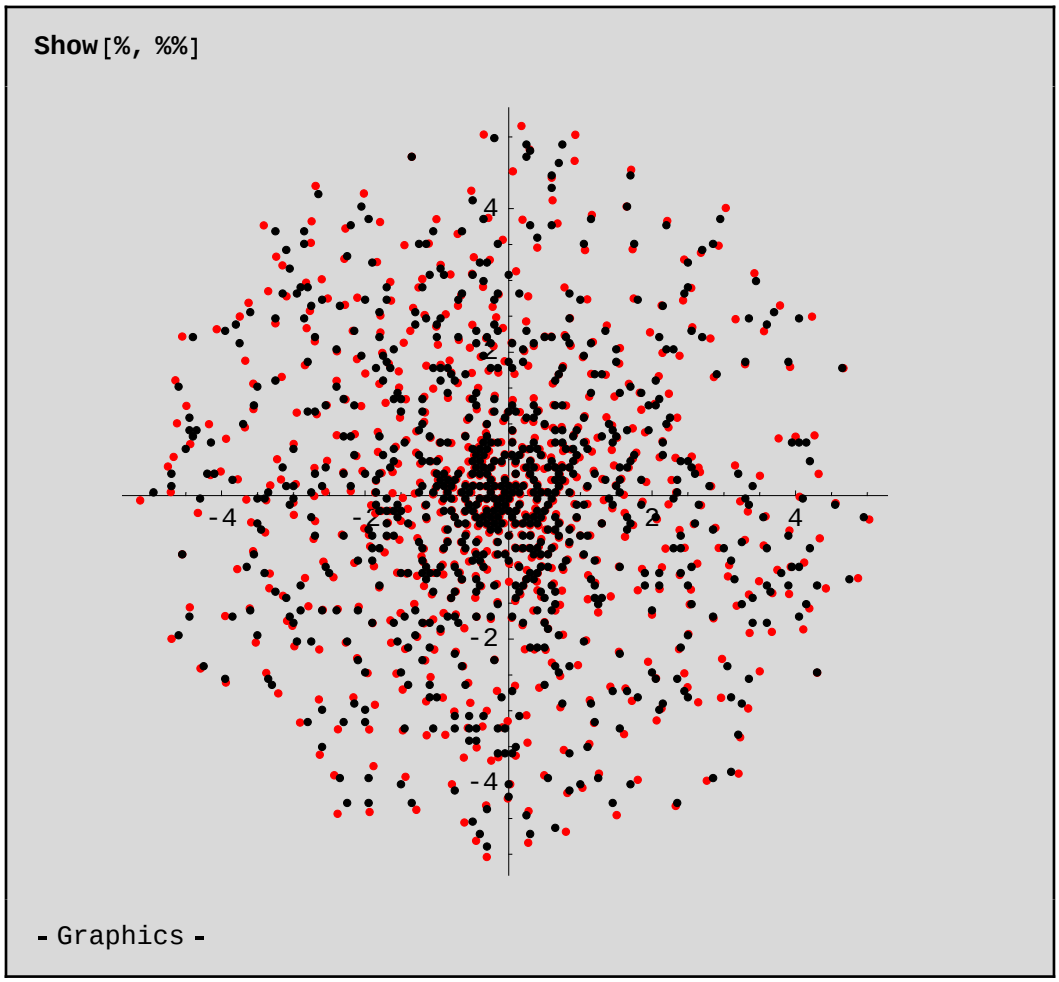

```
- Graphics -
```

after one iteration

```
ListPlot[pp, AspectRatio → Automatic,
 PlotStyle → {PointSize[0.01], RGBColor[1, 0, 0]}]
```




```
- Graphics -
```



All these efforts up till now only moves meshpoints for one step. In order to go further, we need to define a function by wrapping up those temporary variables inside.

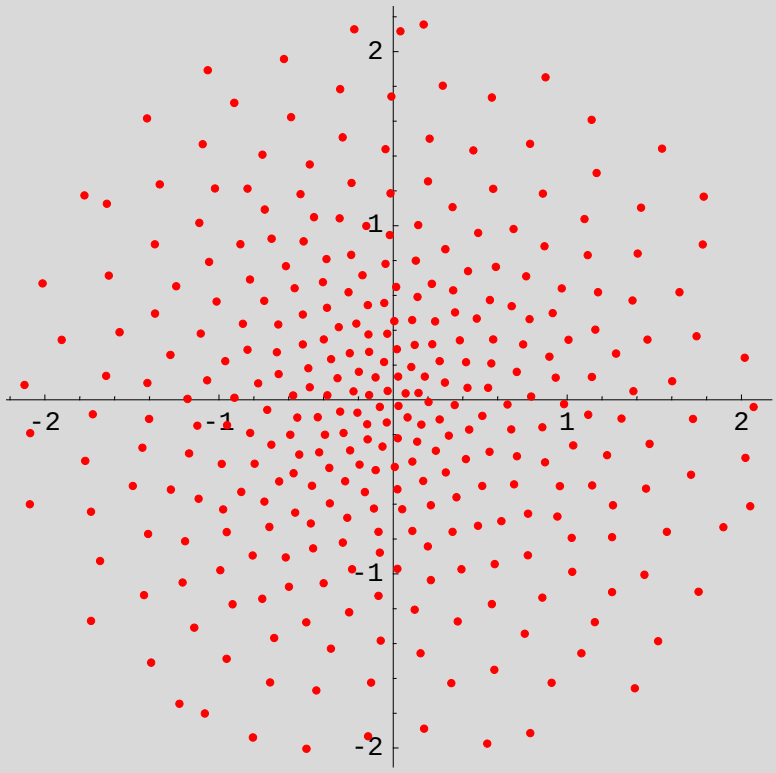
```
forcemove[p_] := Module[{t, bars, l0, F, Fvec, forcematrix, dp},
  t = DelaunayTriangulation[p];
  (* select only interia meshbars *)
  bars = Select[meshbar[t], d[middlepoint[p, #]] < 0 &];
  l0 = With[{Fscale = 1.2},
    Fscale × scale[h[middlepoint[p, #]] & /@ bars, {p, bars}]];
  F = Max[#, 0] & /@ (l0 - (barlength[p, #] & /@ bars));
  Fvec = F (Subtract @@ p[[#]] / barlength[p, #] & /@ bars);
  forcematrix = SparseArray[#, {Length[p], Length[p]}] &@
    (MapThread[ForceRule, {bars, Fvec}] // Flatten[#, 1] &);
  dp = With[{deltat = 0.2}, deltat × Table[Plus @@
       (ReleaseHold /@ forcematrix[[i]]), {i, Length[p]}]];
  Return[p + dp]
 ]
```

Now we can try more iterations.

```
d[{x_, y_}] := dcircle[{x, y}, {0, 0, 2}];
h[{x_, y_}] := 0.1 + 0.1 √(x^2 + y^2);
pts = selectMeshpoints[d, {-10, -10, 10, 10}, 0.1];
values = scale[(1 / h[#]^2 & /@ pts)];
p2 = Extract[pts, Position[values, _?(# > Random[] &)]];
pp2 = Nest[forcemove, p2, 10];
```

So after 10 iterations, the result is quite good already.

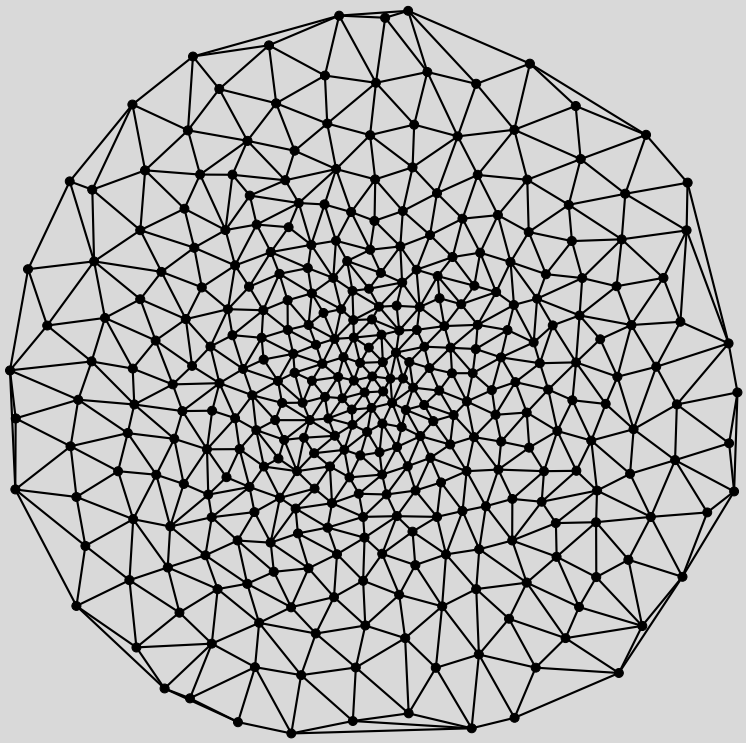
```
ListPlot[pp2, AspectRatio → Automatic,
 PlotStyle → {PointSize[0.01], RGBColor[1, 0, 0]}]
```




```
- Graphics -
```

Once we have the locations of the meshpoints, it is simply a function call to generate the meshes.

```
(* the Mathematica built-in function *)
PlanarGraphPlot[pp2, LabelPoints → False]
```



```
- Graphics -
```

Good Enough?

There are many ways to define and combine the termination criteria.

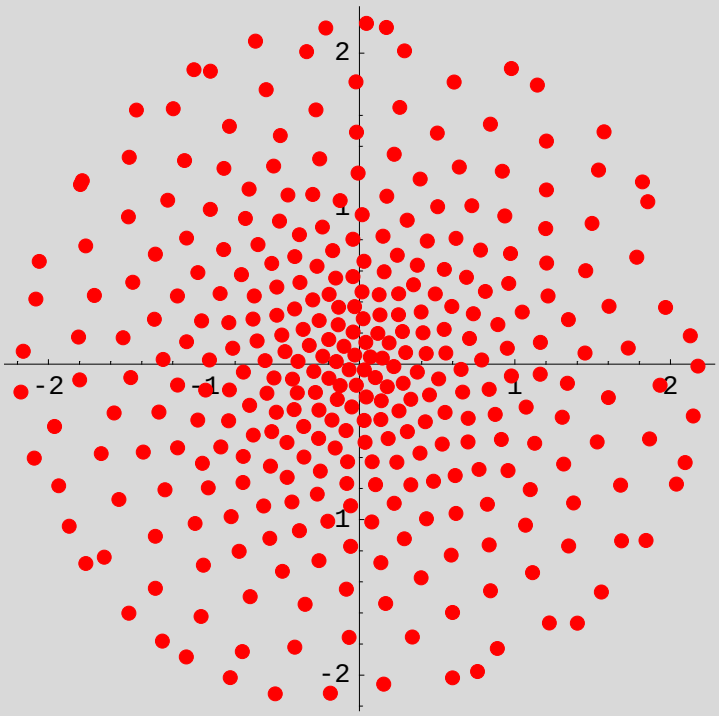
A convenient way for *Mathematica* would be, i.e. if two latest iterations output nearly equal locations, the program doesn't go any further. (Like *step 8* in the paper, we only choose interior meshpoints to compare.)

```
goodEnough[p1_, p2_, eps_] :=
 Max[Norm /@ (Extract[(p2 - p1), Position[p2, _?(d[#] < 0 &)]])] <
  eps
```

Now we can assembly the whole program as predicted in the "overview" section. (It may take a while to run depending on the speed of your computer.)

```
pp2 =
  NestWhile[forcemove, p2, ! (goodEnough[#1, #2, 0.02]) &, 2, 50];
```

```
ListPlot[pp2, AspectRatio → Automatic,
 PlotStyle → {PointSize[0.02], RGBColor[1, 0, 0]}]
```




```
- Graphics -
```

The boundary looks rough above. To complete the algorithm, we need to add the section that brings points outside onto the boundary.

Mesh Generator as a *Mathematica* Package

All the previous sections have taken the original code pieces apart so that we can experiment with each component of the algorithm. To assemble them together, we can put it into a *Mathematica* package (.m) file and fasten some screws along the way. The package effectively hides the details of the algorithm so that end users can simply load the package and use the functions available.

The *Mathematica* package is in a separate file (`meshgenerator.m`). To construct the package, I basically copy and paste the cells from these previous sections into it. This could become tedious when the functions are numerous. However there is no reason why an automatic process (maybe called `MathematicaTangle`) can't be used for larger package constructions. Each code piece in an input cell can be tagged by a section name. And by using `Author-`

`Tools` and `NotebookFind`, `NotebookWrite` functions, etc., one can extract and assemble the pieces into a new notebook. The whole notebook cell can then be marked as a initialization cell and saved as *Mathematica* package file.

Here is an example of using the package:

```
(* clear everything defined before*)
Remove["Global`*"]
```

```
(* set the current directory to where the
  package file is stored and read in the package *)
SetDirectory["c:\\"]; << meshgenerator`
```

```
?generateMesh
```

```
generateMesh[d_,h_,h0_,{x1_,y1_,x2_,y2_}] generates a mesh
  for a region defined by the distance function d(x,y) and
  size function h(x,y). The mesh sise is h0 and an initial
  rectangle region is defined by (x1,y1) and (x2,y2).
```

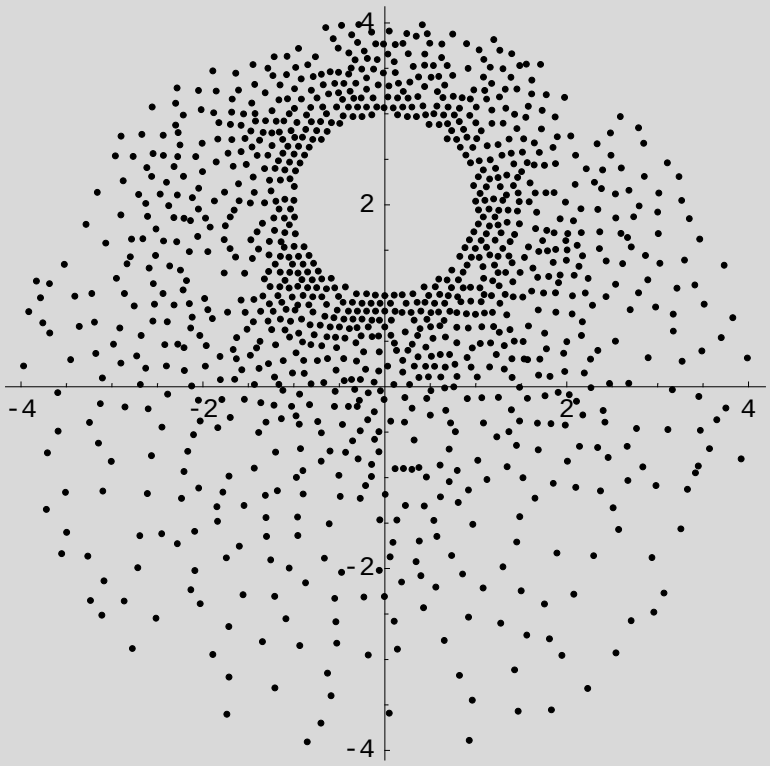
```
Options[generateMesh]
```

```
{Fscale → 1.2, DeltaT → 0.2, Eps → 0.2, MaxSteps → 50}
```

```
d[{x_, y_}] :=
  ddiff[dcircle[{x, y}, {0, 0, 4}], dcircle[{x, y}, {0, 2, 1}]];
h[{x_, y_}] := 0.1 + 0.4 Sqrt[x^2 + (y - 2)^2];
```

```
(* It may take quite a while to run *)
p2 = generateMesh[d, h, 0.1, {-10, -10, 10, 10}];
```

```
ListPlot[p2, AspectRatio → Automatic]
```




```
- Graphics -
```

Summary and Discussion

This *Mathematica* notebook sets up an experiment environment for Persson and Strang's mesh generation algorithm. It also tries to document the program in a literate programming style.

The translation from the original MATLAB code to the *Mathematica* code is far from complete. It only touches the program in Figure 3.1 of the paper, though, I hope, it provides a good starting point for testing and understanding the algorithm. Various experiments can be run from this point, such as defining different distance functions or size functions, testing different scaling process or objective function, etc., especially when the optimization process gets stuck, you can try many places to shake.

There are multiple ways to program in *Mathematica*, thus to transcribe the original MATLAB code. The original MATLAB code makes good use of MATLAB's list processing ability and therefore is very concise. Here we adopted a somewhat functional programming style in *Mathematica*. Like structure programming isn't really about getting rid of "goto"s, functional programming isn't really about getting rid of "for" loops. A functional program avoids relying on variables while builds up on definitions of functions. It then uses `Map (/@), Apply (@@)`, and `Nest`, etc., to glue these functions together. The advantage is that once these functions are declared, they won't change anymore. So testing or running the program becomes declarative rather than imperative, which means the order of which expres-

sions are evaluated doesn't matter so much. Whereas the expressions that make heavy use of variables can have trouble (side-effects) brought by changing the evaluation order or reevaluating them, which is not favorable for testing and experimenting.

I am still learning to program in *Mathematica*. So the code here can certainly be improved both towards more elegant in style and faster in speed, not to mention the hidden bugs to squash. Like so many open software and open documents on the internet, future versions of this notebook will be better through suggestions and advice from readers like you.

My email address: huzhe@iit.edu

Mathematica Packages Needed

```
<< NumericalMath`NLimit`
```

```
<< DiscreteMath`ComputationalGeometry`
```

References

[1] Per-Olof Persson and Gilbert Strang, "A Simple Mesh Generator in MATLAB," SIAM Review (August, 2004)

[2] Roman Maeder, "Programming in *Mathematica*," Addison-Wesley, 1997