Generacja trójkątnej siatki obliczeniowej

June 6, 2020

1 Generacja trójkątnych siatek obliczeniowych w Julii

Jan Starczewski
nr. albumu 291105
Informatyka Stosowana

1.1 Wstęp

Celem projektu jest próba implementacji algorytmu DistMesh w języku `Julia`. Algorytm generujący siatki trójkątne na płaszczyźnie jest powszechnie znanym rozwiązaniem w dziedzinie metod numerycznych, stworzony w `MatLabie`, wielokorotnie przepisywany, doczekał się implementacji w wielu popularnych językach programowania takich jak `Python`, `R`, a nawet `Julia`. Nie zmienia to faktu, że reimplementacja tego algorytmu własnoręcznie jest świetnym sposobem na podstawowe zaznajomienie się z tematem siatek obliczeniowych. Język programowania `Julia` jest stosunkowo nowym narzędzien w dziedzinie analizy numerycznej i analizy danych.

1.2 Implementacja algorytmu w projekcie

Projekt pokazuje proces generacji początkowej siatki obliczeniowej, na której bazuje algorytm DistMesh, wykrzystując narzędzia, które powszechnie nie zostały użyte w przykładowych implementacjach alogrytmu, dostepnych w internecie np. Julia DistMesh. Projekt należy interpetować jako poradnik pozwalajacy na poznanie podstaw Julii i zrozumienie problemu generacji siatek na przykładzie. Implementowany algorytm bardzo mocno bazuje na przykładzie opisanym w tym dokumencie. Nie jest on jednak przepisywany jeden do jednego, z tego powodu, że część użytch w nim rozwiązań bazuje na funkcjonalnościach dostępnych w narzedziu Matlab, a niedostępnych bezpośrednio w `Julii`. Z tego powodu czasami zostały wprowadzone modyfikacje mające na celu uzyskanie wyników identycznych, jak te zwracane przez oryginalną implementacje.

1.3 Generacja początkowej siatki obliczeniowej

Funkcja generujaca siatkę w naszym uproszczonym przykładzie posiada taki oto nagłówek

```
function generate(fd, fh, bbox, h0)
```

Następujące parametry to * `fd` oznaczona funkcja odległości * `fh` oczekiwana długość boku, zdefiniowana jako funkcja zwracajaca stałą wartość * `bbox` Boundry Box defiiujący obszar wewnatrz którego naniesiemy punkty * `h0` odległość początkową miedzy punktami

1.4 Oznaczona funkcja odległości

Ważnym problemem podczas twórzenia algorytmu było przyjęcie konwencji w jaki sposób będziemy definiować kształt wewnatrz którego narysowana zostanie siatka obliczeniowa. Wybrane rozwiazanie problemu to zastosowanie oznaczonych funkcji odległości d(x,y), które przyjmują wartości ujemne wewnątrze kształtu. Funkcja odległości zwraca odległość do najbliższej granicy dla wybranego punktu.

W projekcie będziemy starali się pokryć siatką obliczeniową wntęrze okregu zdefiniowanego przez nastepujące równanie \$ {x$^{\{2\}\}+\{y}$ {2}}=1 \$
Tworzymy oznaczoną funkcję odległości dla tego kształtu, którą bedziemy używać w projekcie do generacji siatki. Punkt `p` oznacza dwuelementową macierz reprezentującą punkt `[x y]`

[1]:
```
function fd(p)
    return sqrt(sum(p.^2))-1
end
```

[1]: `fd (generic function with 1 method)`

1.5 Długość boku siatki dla danego punktu

Definiujemy również funkcję, która określa długość boku sitaki dla danych wartości `xi y`, w przykładzie przyjmujemy, że jest to wartość stała.

[2]:
```
function fh(x,y)
    return 0.2
end
```

[2]: `fh (generic function with 1 method)`

1.6 Meshgrid

Pierwszym elementem algorytmu jest utworzenie siatki wypełnionej punktami. W Matlabie jest to zrealizowane w następujący sposób

```
[x,y]=meshgrid(bbox(1,1):h0:bbox(2,1),bbox(1,2):h0*sqrt(3)/2:bbox(2,2));
```

W Julii nie istnieje funkcja `meshgrid` o takiej samej funkcjonalnosci, z tego powodu jesteśmy zmuszeni napisać ją sami. Funkcja przyjmuje jako parametry dwa wektory o tym samym typie danych, zdefiniowanym przez typ generyczny `T`. Funkcja tworzy macierz puntków będacych ułożeniem poczatkowym.

[3]:
```
function meshgrid(vx::AbstractVector{T}, vy::AbstractVector{T}) where {T}
    m, n = length(vy), length(vx)
    gx = reshape(repeat(vx, inner = m, outer = 1), m, n)
    gy = reshape(repeat(vy, inner = 1, outer = n), m, n)
    return gx, gy
end
```

[3]: `meshgrid (generic function with 1 method)`

Funkcja `repeat` służy do powielania elementów macierzy. Co widać najlepiej na przykładzie

[4]:
```
a = [1; 2]
```

[4]:
```
2-element Array{Int64,1}:
 1
 2
```

[5]:
```
repeat(a, inner = 2)
```

[5]:
```
4-element Array{Int64,1}:
 1
 1
 2
 2
```

Funkcja `reshape` tworzy macierz z wektora, o wymiarach zdefiniowanych przez parametry `m` i `n` w przypadku funkcji `meshgrid`.

1.7 Początkowe rozłożenie punktów

Do funkcji `meshgrid` przekazujemy punkty początkowe, wyznaczone na podstawie `BoundryBox'a` z odległością `h0` między punktami na osi OX i odległością `h1` na osi OY

[6]:
```
bbox = [-1 -1; 1 1]
```

[6]:
```
2×2 Array{Int64,2}:
 -1  -1
  1   1
```

[7]:
```
h0 = 0.2
```

[7]:
```
0.2
```

[8]:
```
h1 = h0 * sqrt(3)/2
```

[8]:
```
0.17320508075688773
```

Definiujemy wektor `v1` reprezentujący kolejno współrzędne `x` punktów początkowych.

[9]:
```
v1 = bbox[1, 1]:h0:bbox[2, 1]
```

[9]:
```
-1.0:0.2:1.0
```

Definiujemy wektor `v2` reprezentujacy kolejno współrzędne `y` punktów początkowych.

[10]:
```
v2 = bbox[1, 2]:h1:bbox[2, 2]
```

[10]:
```
-1.0:0.17320508075688773:0.9052558883257651
```

Następnie przekazujemy stworzone wektory do funkcji `meshgrid`. W powstałym ułożeniu początkowym, co drugi punkt przesuwamy połowe wartości `h0` względem osi OX w celu uzyskania "efektu kratownicy".

[11]:
```
x, y = meshgrid(v1, v2)
x[2:2:end, :] = x[2:2:end, :] .+ h0 / 2
```

[11]:
```
6×11 Array{Float64,2}:
 -0.9  -0.7  -0.5  -0.3  -0.1  0.1  0.3  0.5  0.7  0.9  1.1
 -0.9  -0.7  -0.5  -0.3  -0.1  0.1  0.3  0.5  0.7  0.9  1.1
 -0.9  -0.7  -0.5  -0.3  -0.1  0.1  0.3  0.5  0.7  0.9  1.1
 -0.9  -0.7  -0.5  -0.3  -0.1  0.1  0.3  0.5  0.7  0.9  1.1
 -0.9  -0.7  -0.5  -0.3  -0.1  0.1  0.3  0.5  0.7  0.9  1.1
 -0.9  -0.7  -0.5  -0.3  -0.1  0.1  0.3  0.5  0.7  0.9  1.1
```

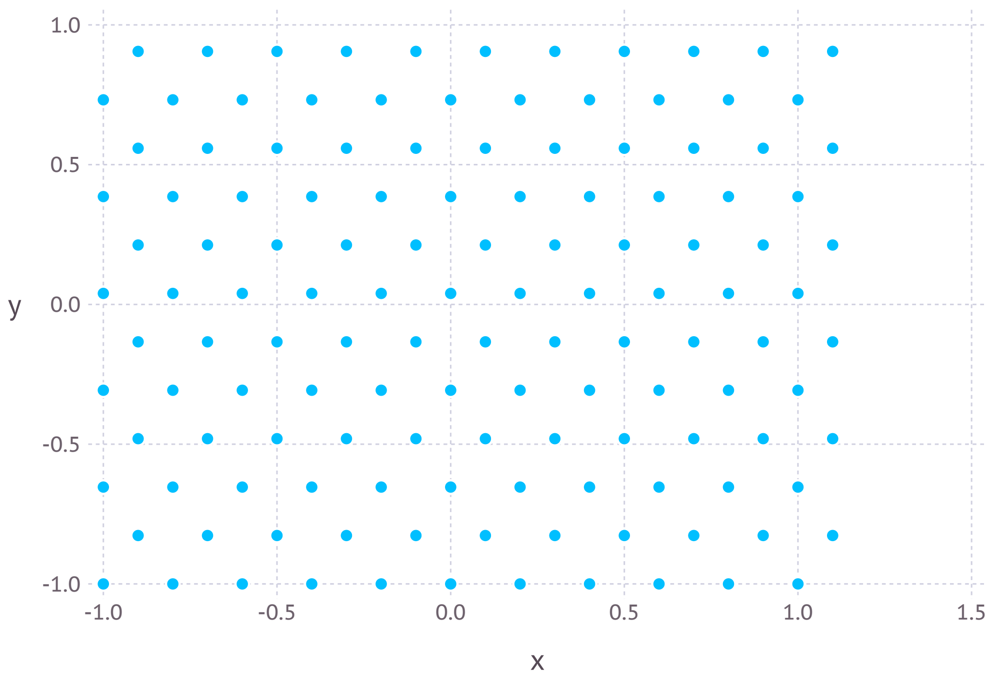
Otrzymane wyniki wzializujemy na wykresie punktowym, w celu sprawdzenia poprawności wykonanych operacji. Do narysowania wykresu wykrzystamy bibliotekę `Plots`.

[12]:
```
using Pkg;
Pkg.add("Cairo");
Pkg.add("Fontconfig");
using Cairo;
using Fontconfig;
using Gadfly;
```

```
  Updating registry at `~/.julia/registries/General`
  Updating git-repo
`https://github.com/JuliaRegistries/General.git`
[1mFetching: [========================================>]
100.0 %.0 % Resolving package versions…
  Updating `~/.julia/environments/v1.3/Project.toml`
 [no changes]
  Updating `~/.julia/environments/v1.3/Manifest.toml`
 [no changes]
 Resolving package versions…
  Updating `~/.julia/environments/v1.3/Project.toml`
 [no changes]
  Updating `~/.julia/environments/v1.3/Manifest.toml`
 [no changes]
```

[13]:
```
p = [x[:] y[:]]
points_plot = Gadfly.plot(x=p[:,1], y=p[:,2])
#Gadfly.draw(PDF("Points.PDF"), points_plot)
```

[13]:



1.7.1 Odrzucenie puntków poza rozpatrywaną przestrzenią

Następnym krokiem jest usunięcie tych punktów, które znajdują się poza rozpatrywaną przez nas przestrzenią. Wartość funkcji odległości dla tych punktów będzie większa bądź równa 0, jednak przyrównywać będziemy do stałej `geps = .001*h0` oznaczającej "tolerancję" przy wykonywanych obliczeniach geometrycznych.

[14]:
```
geps = 0.001*h0
```

[14]:
```
0.0002
```

Odrzucenie punktów Matlabie jest przeprowadzone z wykrzystaniem poniższej operacji, gdzie `varargin{:}` oznacza dodatkowe parametry, które może przyjąc funkcja `fd` i `fh` co zapewnia większą kontrolę nad pracą algorytmu w bardziej wyspecjalizowanych przypadkach.

```
p=p(feval(fd,p,varargin{:})<geps,:);
```

W rozpatrywanym przykładzie, nie bierzemy pod uwagę dodatkowych parametrów. Wartość `valid_p` zawiera wszystkie punkty spełniajace kryterium, że wartośc funkcji odległości jest mniejsza niż `geps`. Poniższa instrukcja iteruje po wszystkich rzędach macierzy `p` i zostawia tylko te rzędy (reprezentujące punkt), które spełniają kryterium określone w warunku `if`. Każdy taki rząd jest złączany wzdłóż swojej długości (wzdłóż pierwszego wymiaru).

[15]:
```
valid_p = [vcat(point_row) for point_row in eachrow(p) if fd(point_row) < geps]
```

[15]:
```
88-element Array{Array{Float64,1},1}:
 [-0.9, -0.1339745962155614]
 [-0.9, 0.21243556529821417]
 [-0.7000000000000001, -0.48038475772933675]
 [-0.8, -0.3071796769724491]
 [-0.7000000000000001, -0.1339745962155614]
 [-0.8, 0.039230484541326494]
 [-0.7000000000000001, 0.21243556529821417]
 [-0.8, 0.38564064605510184]
 [-0.7000000000000001, 0.5588457268119895]
 [-0.5, -0.8267949192431123]
 [-0.6, -0.6535898384862245]
 [-0.5, -0.48038475772933675]
 [-0.6, -0.3071796769724491]

 [0.6, -0.3071796769724491]
 [0.7, -0.1339745962155614]
 [0.6, 0.039230484541326494]
 [0.7, 0.21243556529821417]
 [0.6, 0.38564064605510184]
 [0.7, 0.5588457268119895]
 [0.6, 0.7320508075688772]
 [0.8, -0.3071796769724491]
 [0.9, -0.1339745962155614]
 [0.8, 0.039230484541326494]
 [0.9, 0.21243556529821417]
 [0.8, 0.38564064605510184]
```

Następnie zamieniamy wektor zawierajacy mniejsze dwuelementowe wektory współrzednych na macierz, wykonując poniższą instrukcję.

```
valid_p_m = transpose(reshape(vcat(valid_p...), 2, length(valid_p)))
```

Z uwagi na dośc sporą złożoność powyższej instrukcji, rozłożymy ją na części pierwsze. Zaczynamy od rozbicia wektora wektorów na pojedynczy wektor o zawierący wszystkie współrzędne.

[16]:
```
valid_p_spread = vcat(valid_p...)
```

[16]:
```
176-element Array{Float64,1}:
 -0.9
 -0.1339745962155614
 -0.9
  0.21243556529821417
 -0.7000000000000001
 -0.48038475772933675
 -0.8
 -0.3071796769724491
 -0.7000000000000001
```

```
-0.1339745962155614
-0.8
 0.039230484541326494
-0.7000000000000001
 ⋮
 0.6
 0.7320508075688772
 0.8
-0.3071796769724491
 0.9
-0.1339745962155614
 0.8
 0.039230484541326494
 0.9
 0.21243556529821417
 0.8
 0.38564064605510184
```

Uzyskaliśmy pojedynczy wektor, który zawiera wszystkie punkty spłaszczone do pierwszego wymiaru, a dokładnie, naprzemiennie wartości `x`, `y` poszczególnych punktów. Teraz zajmiemy się przekształceniem wektora w macierz o dwóch rzędach i ilości kolumn równej ilości punktów.

[17]:
```
valid_p_spread_m = reshape(valid_p_spread, 2, length(valid_p))
```

[17]:
```
2×88 Array{Float64,2}:
 -0.9       -0.9       -0.7       -0.8      …  0.8        0.9       0.8
 -0.133975   0.212436  -0.480385  -0.30718     0.0392305  0.212436  0.385641
```

Finalnie transponujemy macierz, aby zachować preferowaną dla nas strukturę o dwóch kolumnach reprezentujących wartośći `x` i `y` punktów.

[18]:
```
valid_points = transpose(valid_p_spread_m)
```

[18]:
```
88×2 LinearAlgebra.Transpose{Float64,Array{Float64,2}}:
 -0.9  -0.133975
 -0.9   0.212436
 -0.7  -0.480385
 -0.8  -0.30718
 -0.7  -0.133975
 -0.8   0.0392305
 -0.7   0.212436
 -0.8   0.385641
 -0.7   0.558846
 -0.5  -0.826795
 -0.6  -0.65359
 -0.5  -0.480385
 -0.6  -0.30718
```

```
0.6  -0.30718
0.7  -0.133975
0.6   0.0392305
0.7   0.212436
0.6   0.385641
0.7   0.558846
0.6   0.732051
0.8  -0.30718
0.9  -0.133975
0.8   0.0392305
0.9   0.212436
0.8   0.385641
```

1.7.2 Prawdopodobnieństwo zatrzymania punktu

Dla siatek obliczeniowych o różnych długościach krawędzi, w celu uzyskania szybszej zbieżności dokonuje się odrzucenia punktów, dla których wartość funkcji \$ h(x,y) \$ jest proporcjonalna do \$ 1/{h(x,y)}^2 \$. W Matlabie jest to zrealizowane przez następujące instrukcje.

```
r0=1./feval(fh,p,varargin{:}).^2;
p=[pfix; p(rand(size(p,1),1)<r0./max(r0),:)];
```

Rozbijemy powyższy blok na poszczególne kroki.

[19]:
```
r0 = 1 ./ fh(valid_points[:,1], valid_points[:,2]).^2
```

[19]:
```
24.999999999999996
```

Określamy `r0_max`

[20]:
```
r0_max = r0./max(r0)
```

[20]:
```
1.0
```

[21]:
```
valid_points_r = [vcat(point_row) for point_row in eachrow(valid_points) if␣
↪(rand(Float64, size(valid_points)))[1] < r0_max ]
```

[21]:
```
88-element Array{Array{Float64,1},1}:
 [-0.9, -0.1339745962155614]
 [-0.9, 0.21243556529821417]
 [-0.7000000000000001, -0.48038475772933675]
 [-0.8, -0.3071796769724491]
 [-0.7000000000000001, -0.1339745962155614]
 [-0.8, 0.039230484541326494]
 [-0.7000000000000001, 0.21243556529821417]
 [-0.8, 0.38564064605510184]
 [-0.7000000000000001, 0.5588457268119895]
 [-0.5, -0.8267949192431123]
 [-0.6, -0.6535898384862245]
```

```
[-0.5, -0.48038475772933675]
[-0.6, -0.3071796769724491]

[0.6, -0.3071796769724491]
[0.7, -0.1339745962155614]
[0.6, 0.039230484541326494]
[0.7, 0.21243556529821417]
[0.6, 0.38564064605510184]
[0.7, 0.5588457268119895]
[0.6, 0.7320508075688772]
[0.8, -0.3071796769724491]
[0.9, -0.1339745962155614]
[0.8, 0.039230484541326494]
[0.9, 0.21243556529821417]
[0.8, 0.38564064605510184]
```

Otrzymany wektor wektorów zamieniamy na macierz, analogicznie do wcześniej omówionego przykładu.

[22]:
```
points = transpose(reshape(vcat(valid_points_r...), 2, length(valid_points_r)))
```

[22]:
```
88×2 LinearAlgebra.Transpose{Float64,Array{Float64,2}}:
 -0.9  -0.133975
 -0.9   0.212436
 -0.7  -0.480385
 -0.8  -0.30718
 -0.7  -0.133975
 -0.8   0.0392305
 -0.7   0.212436
 -0.8   0.385641
 -0.7   0.558846
 -0.5  -0.826795
 -0.6  -0.65359
 -0.5  -0.480385
 -0.6  -0.30718

  0.6  -0.30718
  0.7  -0.133975
  0.6   0.0392305
  0.7   0.212436
  0.6   0.385641
  0.7   0.558846
  0.6   0.732051
  0.8  -0.30718
  0.9  -0.133975
  0.8   0.0392305
  0.9   0.212436
```

```
0.8   0.385641
```

Finalnie mamy tyle samo punktów poczatkowych, co przed operacją, ze względu na stałą wartość funkcji `fh`.

[23]:
```
N = size(points,1)
```

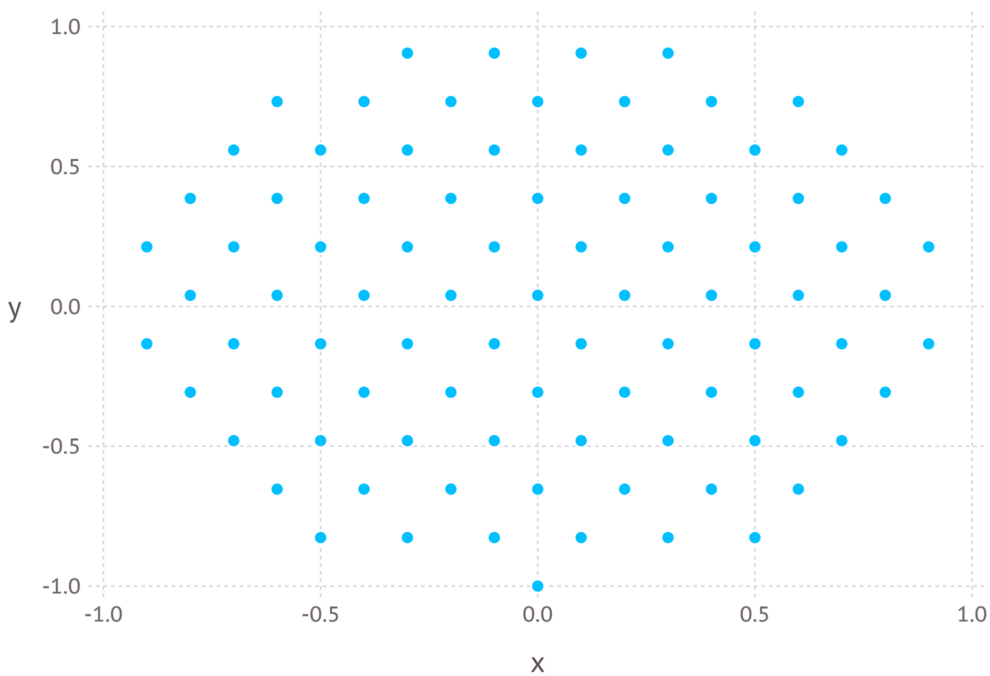
[23]:
```
88
```

1.7.3 Wizualizacja punktów początkowych

Punkty wizualizujemy na wykresie.

[24]:
```
r_points_plot = Gadfly.plot(x=points[:,1], y=points[:,2])
#Gadfly.draw(PDF("Rpoints.PDF"), r_points_plot)
```

[24]:



1.8 Pętla główna alogrytmu

Kolejnym elementem algorytmu jest wejscie w pętlę główną, której zadaniem jest coraz lepsze umiejscowienie punktów, w oparciu, o które będziemy rysować siatkę. Struktura pętli głównej prezentuje się następująco. W wartośći `pold` przechowujemy położenie punktów po każdej retriangulacji.

```
pold = Inf
while true
# Triangulacja
# Przesuwanie punktów
# Kryteria zatrzymania
end
```

Poniżej zajmiemy się implementacją ciała głównej pętli `while`, począwszy od przeprowadzenia triangulacji Delone. Do przeprowadzenia triangulacji wykorzystamy natywną bibliotekę napisaną w języku `Julia`, która nie jest opakowaniem na kod napisany w innym języku posiadający interfejs podobny tego dstępnego w Matlabie. Konsekwencją tego działania jest lekkie odejście od schematu algorytmu, opisanego w dokumencie co przekłada się na potrzębę własnej implementacji poszczególnych kroków alogrytmu z zachowaniem oczekiwanej funkcjonalnośći.

1.8.1 Triangulacja

Wykorzystywana bilbioteka do triangulacji posiada jedno zasadnicze ogarniczenie. Rozpatrywane na płaszczyźnie punkty musza należeć do przedziału od `1.0` do `2.0` co implikuje przeskalowanie rozpatrywanego przez nas przedziału na ten, który możemy poddać triangulacji. Dodatkowym "kosztem" wykorzystania tego rozwiązania jest przejscie z prymitywnych typów danych wykorzystywanych do reprezentacji punktów, na strukturę `Point2D`, na której oparta jest biblioteka. Stała `ttol = 0.1` mówi jak daleko mogą relatywnie przesunąc się punkty przed triangulacją.

[25]:
```
ttol = 0.1
pold = Inf
```

[25]: Inf

1.8.2 Wynik cząstkowy triangulacji

Definiujemy warunek, pod którym zachodzi triangulacja, przechodzimy do pisania ciała bloku `if`, który jest odpowiedzialny za triangulację i narysowanie wyniku cząstkowego.

[26]:
```
if(maximum(sqrt.(sum((points.-pold).^2, dims = 2)/h0))>ttol)
    # Triangulacja
end
```

W celu przeprowadzenia triangulacji, wciągamy bibliotekę i tworzymy obiekt reprezentujacy tesselację.

[27]:
```
using VoronoiDelaunay
tess = DelaunayTessellation();
```

Biorąc pod uwagę ogarniczenia biblioteki obliczam wartości potrzebne do poprawnego przeskalowania punktów.

[28]:
```
#Skala
scale = abs(1/(bbox[1] - bbox[2]))
```

[28]: 0.5

[29]:
```
#Przesunięcie OX
transx = sqrt((-(0-bbox[1]*scale)-1)^2)
```

[29]: 1.5

[30]:
```
#Przesunięcie OY
transy = sqrt((-(0-bbox[1,2]*scale)-1)^2)
```

[30]: 1.5

Punkty początkowe mapujemy na strutkurę `Point2D` skalując, a nastepnie przesuwając wartości `x` i `y`.

[31]:
```
points2d = map(el -> Point(el[1]*scale+transx, el[2]*scale+transy),␣
 ↪eachrow(points))
```

[31]:
```
88-element Array{Point2D,1}:
 Point2D(1.05, 1.4330127018922192)
 Point2D(1.05, 1.606217782649107)
 Point2D(1.15, 1.2598076211353315)
 Point2D(1.1, 1.3464101615137753)
 Point2D(1.15, 1.4330127018922192)
 Point2D(1.1, 1.5196152422706632)
 Point2D(1.15, 1.606217782649107)
 Point2D(1.1, 1.692820323027551)
 Point2D(1.15, 1.7794228634059948)
 Point2D(1.25, 1.0866025403784438)
 Point2D(1.2, 1.1732050807568877)
 Point2D(1.25, 1.2598076211353315)
 Point2D(1.2, 1.3464101615137753)
 ⋮
 Point2D(1.8, 1.3464101615137753)
 Point2D(1.85, 1.4330127018922192)
 Point2D(1.8, 1.5196152422706632)
 Point2D(1.85, 1.606217782649107)
 Point2D(1.8, 1.692820323027551)
 Point2D(1.85, 1.7794228634059948)
 Point2D(1.8, 1.8660254037844386)
 Point2D(1.9, 1.3464101615137753)
 Point2D(1.95, 1.4330127018922192)
 Point2D(1.9, 1.5196152422706632)
 Point2D(1.95, 1.606217782649107)
 Point2D(1.9, 1.692820323027551)
```

Dodajemy punkty do obiektu reprezentujacego tesselację.

[32]:

```
push!(tess, points2d)
```

Dokonujemy wizualizacji siatki początkowej z wykorzystaniem biblioteki `Gadfly`

[33]:

```
x, y = getplotxy(delaunayedges(tess))
mesh_plot = Gadfly.plot(x=x, y=y, Geom.path)
#Gadfly.draw(PDF("Mesh.pdf"), mesh_plot)
```

[33]: