Abgabe des zweiten Programmierblatts von Sven Ullmann (3467077) und Thomas Stegmeyer (3401787)

Contents

Aufgabe 1

Die Funktionsweise von 1a) wird über die restlichen Aufgaben demonstriert. Die Funktion, um das Gitter zu bauen, sieht wie folgt aus

```
type 'buildGitter.m'
```

```
function Omega = buildGitter(h)
    knots = 3/h+1;
    Omega = zeros(knots,knots,4);
    counter1 = 1;
    inner = 1;
    for i=1:knots
        for j=1:knots
            Omega(i,j,1) = (i-1)*h;
            Omega(i,j,2) = (j-1)*h;
            if (i-1)*h > 2 && (j-1)*h < 1
                Omega(i,j,1) = NaN;
                Omega(i,j,2) = NaN;
            elseif (i-1)*h < 1 && (j-1)*h > 2
                Omega(i,j,1) = NaN;
                Omega(i,j,2) = NaN;
            else
                Omega(i,j,3) = counter1;
                counter1 = counter1 + 1;
            end
        end
    end

    for i=1:knots
        for j=1:knots
            counter2 = 0;
            if Omega(i,j,3)==0
            else
                if j+1<=knots && Omega(i,j+1,3)~=0
                    counter2 = counter2 + 1;
                end
                if j-1>0 && Omega(i,j-1,3)~=0
                    counter2 = counter2 + 1;
                end
                if i+1<=knots && Omega(i+1,j,3)~=0
                   counter2 = counter2 + 1;
                end
                if i-1>0 && Omega(i-1,j,3)~=0
                    counter2 = counter2 + 1;
                end
                if i-1>0 && j-1>0 && Omega(i-1,j-1,3)~=0
                    counter2 = counter2 + 1;
```

```
                end
                if i+1<=knots && j+1<=knots && Omega(i+1,j+1,3)~=0
                    counter2 = counter2 + 1;
                end
                if i-1>0 && j+1<=knots && Omega(i-1,j+1,3)~=0
                    counter2 = counter2 + 1;
                end
                if i+1<=knots && j-1>0 && Omega(i+1,j-1,3)~=0
                    counter2 = counter2 + 1;
                end
            end
            if counter2 == 8
                Omega(i,j,4) = inner;
                inner = inner + 1;
            end
        end
    end
end
```

Wir haben uns in der Implementierung dafür entschieden, die ausgeschnitten Ecken als NaN in der Gittermatrix Omega zu speichern. Damit ist eine Verallgemeinerung für Gebiete aus mehreren zusammengesetzten Würfel / Rechteck relativ einfach zu realisieren. Desweiteren haben wir bereits in der Initialiserung des Gitters eine Nummerierung aller Knoten, sowie eine Nummerierung aller Inneren Knoten vorgenommen. Vorallem letzteres ist hier leider nicht sehr effizient implementiert, aber es erfüllt seinen Zweck. Die Nummerierungen werden vorallem in der Aufgabe 2 folgenden assemble Routine sehr wichtig.

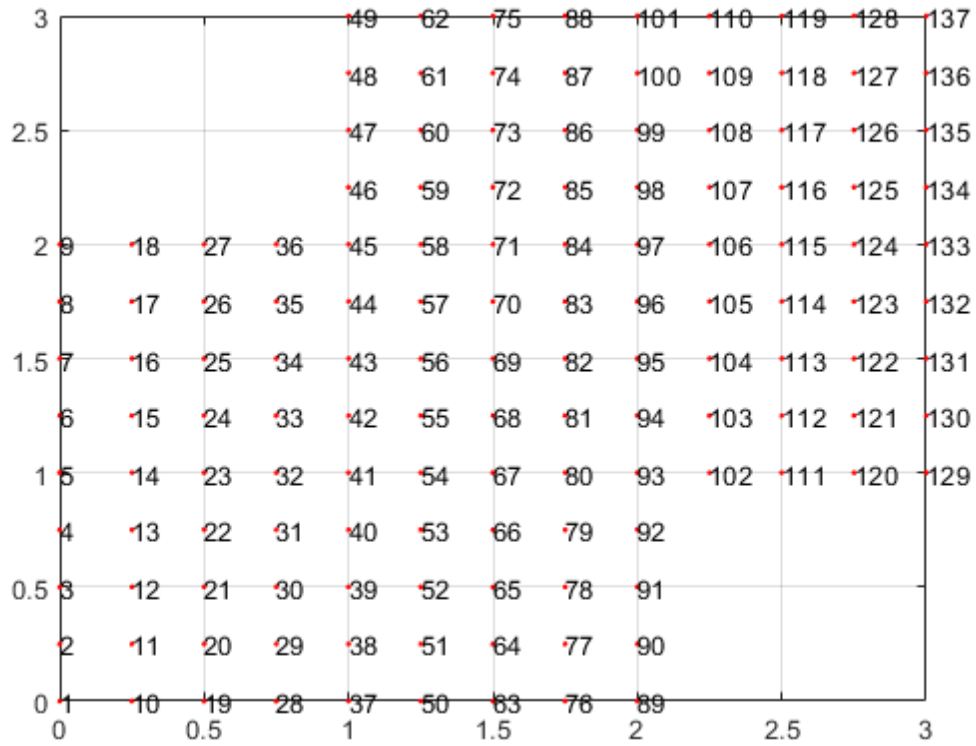
Um das Gitter zu visualisieren haben wir folgenden Routine verwendet:

```
type 'visualize_grid.m'
```

```
function visualize_grid(h)
Omega = buildGitter(h);
figure;
plot(Omega(:,:,1),Omega(:,:,2),'r.');
grid on
hold on
len = length(Omega(1,:,:));
k = 1;
for i=1:len
    for j=1:len
        if Omega(i,j,3)~=0
            text(Omega(i,j,1),Omega(i,j,2),num2str(k),'Color','k')
            drawnow
            pause(1/len^2)
            k = k + 1;
        end
    end
end
hold off
end
```

Damit sieht unser Gitter mit Gitterweite h = 1/4 wie folgt aus

```
visualize_grid(1/4)
```



Aufgabe 2

a) Um die Matrix A und den Vektor b für das anschließend zu lösenden LGS aufzubauen, haben wir uns folgenden Routine überlegt. Wir iterieren erneut über die Gesamte Gittermatrix, betrachten jetzt aber nur inneren Knoten. Dank der Vorarbeit in Aufgabe 1 ist das jetzt ein Kinderspiel. Von jedem inneren Knoten aus betrachten die für den jeweiligen Differenzenstern relevanten Knoten. Liegen diese ebenfalls im inneren, so wird ein Eintrag in Matrix A erzeugt. Sonst wird nichts (bzw. die 0) in die Matrix A eintragen. In den Vektor b tragen wir einfach die inneren den Funktionswert von f ausgewertet am jeweiligen innerne Knoten ein. Hier ist eine verallgemeinerung auf beliebige Randwerte auch möglich, allerdings ist das schon noch mit etwas Arbeit verbunden.

```
type 'assemble2.m'
```

```
function [A,b,u] = assemble2(h,Omega)
knots = 3/h+1;
inner = max(Omega(:,:,4),[],'all');
A = sparse(inner,inner);
b = zeros(inner,1);
u = zeros(inner,2);
counter0 = 1;
counter1 = 1;
for i=1:knots
    for j=1:knots
        if Omega(i,j,4)==0
        else
            A(counter1,counter1) = 4;
            if j+1<=knots && Omega(i,j+1,4)~=0
                diff = Omega(i,j,4) - Omega(i,j+1,4);
                A(counter1,counter1-diff) = -1;
            end
            if j-1>0 && Omega(i,j-1,4)~=0
                diff = Omega(i,j,4) - Omega(i,j-1,4);
                A(counter1,counter1-diff) = -1;
            end
            if i+1<=knots && Omega(i+1,j,4)~=0
```

```
                diff = Omega(i,j,4) - Omega(i+1,j,4);
                A(counter1,counter1-diff) = -1;
            end
            if i-1>0 && Omega(i-1,j,4)~=0
                diff = Omega(i,j,4) - Omega(i-1,j,4);
                A(counter1,counter1-diff) = -1;
            end
        u(counter1,2) = counter0;
        b(counter1,1) = f(Omega(i,j,1),Omega(i,j,2));
        counter1 = counter1 + 1;
        end
        if Omega(i,j,3) ~= 0
            counter0 = counter0 + 1;
        end
    end
end
A = (1/(h^2))*A;
end
```

b) Um das entstanden LGS Au=b zu lösen, verwenden wir den einfachsten, aber laut Google sehr effizieten Weg des '\'. Da wir für die danach anstehende Visualisierung der Lösung auch wieder die Randknoten benötigen, fügen wir diese im Anschluss an richtiger Stelle wieder zu hinzu, sodass der resultierende Vektor u die Lösung auf allen Knoten widerspiegelt.

```
type 'solve.m'
```

```
function u = solve(A,b,u_h, Omega)
    n = max(Omega(:,:,3),[],'all');
    inner = max(Omega(:,:,4),[],'all');
    u = zeros(n,1);
    u_h(:,1) = A\b;
    for i=1:inner
        if u_h(i,2)~=0
            u(u_h(i,2),1) = u_h(i,1);
        end
    end
end
```

c) Nun müssen wir nur noch die Lösung visualisieren: Wir speichern dafür den Vektor u wieder als Matrix, und zwar so das die jeweiligen Vektoreinträge mit der Nummerierung des Gitters übereinstimmt. Es ergibt sich folgendes:

```
type 'visualize.m'
```

```
function visualize(u, Omega, h)
len = length(Omega(1,:,:));
U = zeros(len,len);
k = 1;
for i=1:len
    for j=1:len
        if Omega(i,j,3)==0
        else
            U(i,j) = u(k);
            k = k + 1;
        end
    end
end
```

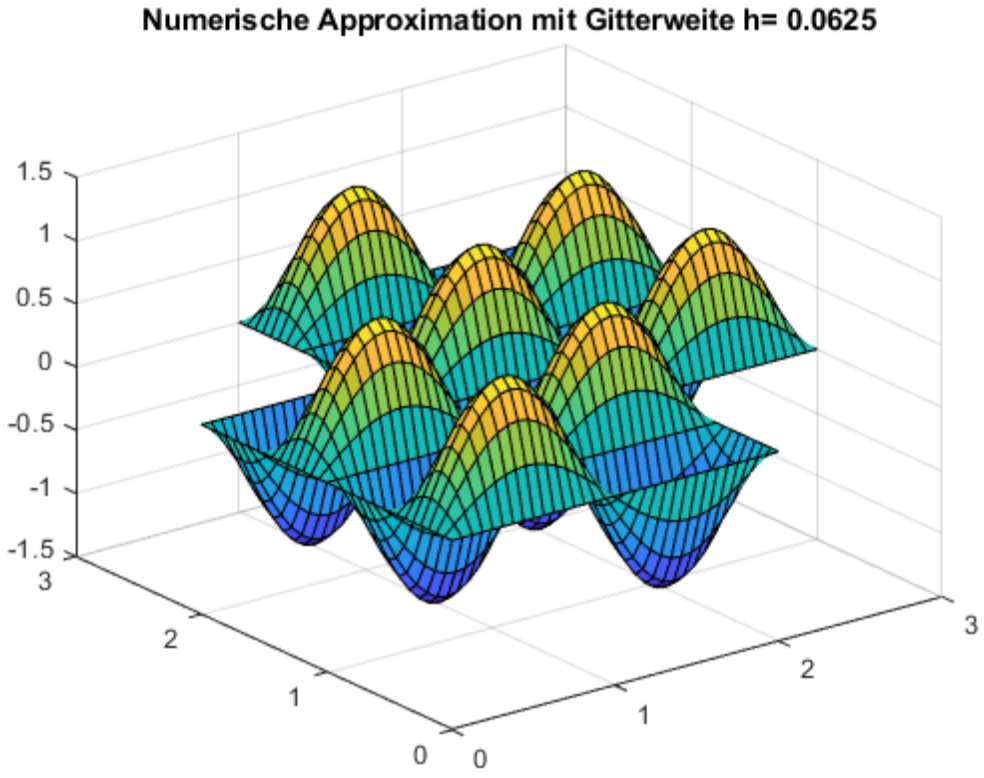
```
figure;
surf(Omega(:,:,1),Omega(:,:,2),U)
title("Numerische Approximation mit Gitterweite h= " + h)
end
```

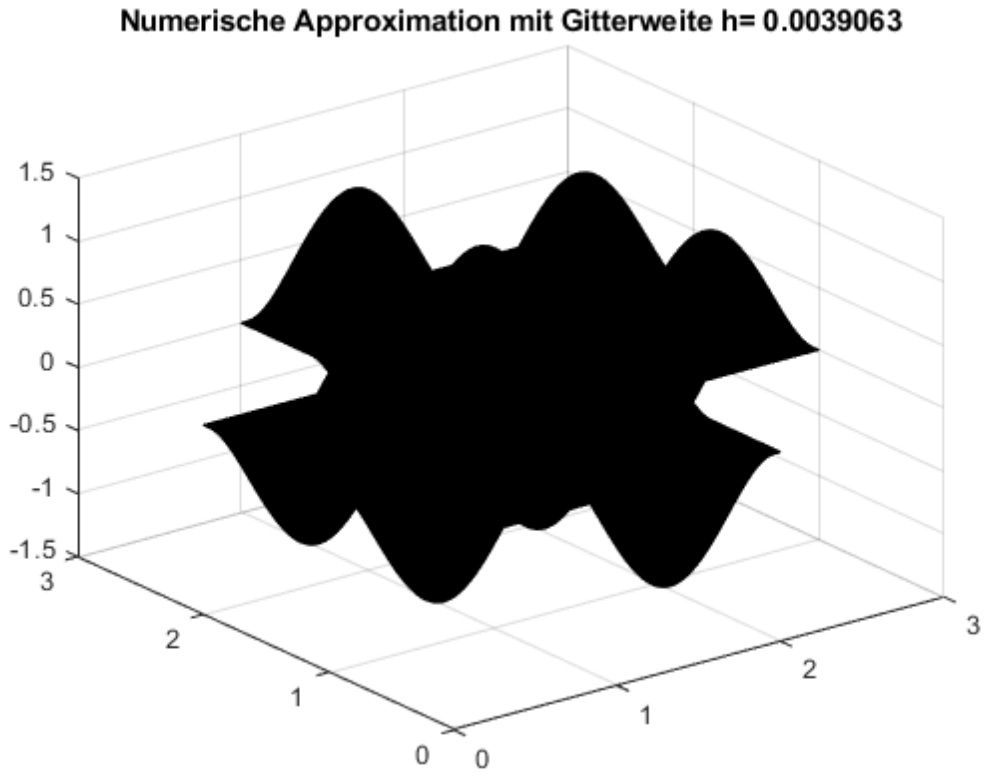
Die Methode ausgeführt wie in der Aufgabenstellung gefordert ergibt:

```
h_c = [1/(2^2), 1/(2^4),1/(2^8)];

for i = h_c
    Omega = buildGitter(i);
    [A,b,u_h] = assemble2(i,Omega);
    u = solve(A,b,u_h, Omega);
    visualize(u, Omega, i);
end
```

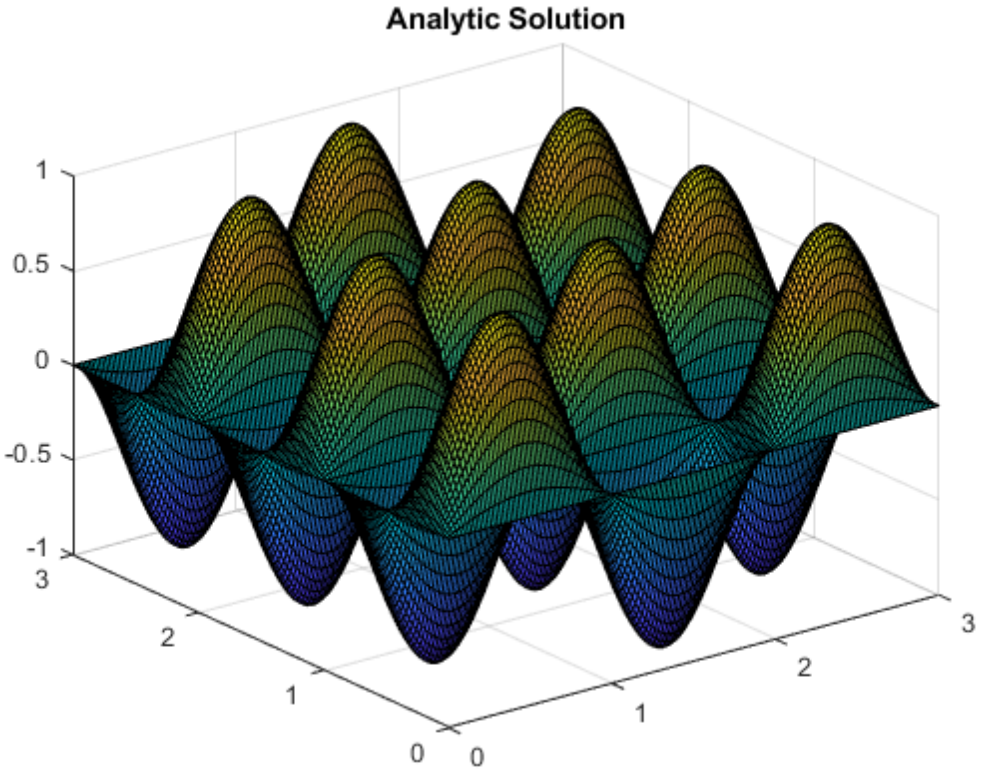

Numerische Approximation mit Gitterweite h= 0.25
1.5
1
0.5
0
-0.5
-1
-1.5
3
2
1
0
0
1
2
3


Numerische Approximation mit Gitterweite h= 0.0625
1.5
1
0.5
0
-0.5
-1
-1.5
3
2
1
0
0
1
2
3



Um sich plausibel zu machen, das diese Ergebnisse in der Augennorm richtig aussehen hier einmal die Analytische Lösung geplottet auf einem recht fein aufgelösten Gitter über das gesamte (0,3)x(0,3) Gebiet.

```
analytic_sol_func = @(x,y) sin(pi*x).*sin(2*pi*y);
[X,Y] = meshgrid(0:0.02:3,0:0.02:3);
figure;
surf(X,Y,analytic_sol_func(X,Y))
title('Analytic Solution')
```



Das sieht doch schonmal gut aus!

Aufgabe 3

a) Jetzt ging es um die Fehler im Vergleich zur Analytischen Lösung, welche uns als

```
analytic_sol_func = @(x,y) sin(pi*x)*sin(2*pi*y);
```

angegeben gegeben wurde. Um den Fehler in der L_inf Norm zu messen haben wir folgende Routine entwickelt:

```
type 'infty_error.m'
```

```
function error = infty_error(u, Omega)
len = length(Omega(1,:,:));
n = max(Omega(:,:,3),[],'all');
analytic_sol_func = @(x,y) sin(pi*x)*sin(2*pi*y);
analytic_sol = zeros(n,1);
k = 1;
for i=1:len
    for j=1:len
        if Omega(i,j,3)~=0
            analytic_sol(k) = analytic_sol_func(Omega(i,j,1),Omega(i,j,2));
            k = k+1;
        end
    end
end
error = max(u - analytic_sol);
end
```

b) Um die Konvergenzordnung 2 auch numerisch Nachzuweisen, haben wir die diese Fehler für einige Gitterweite in einem loglog Plot dargestellt. Dafür verwenden wir die Methode

```
type 'convergence.m'
```

```
function T = convergence()
 H= [1/4,1/8,1/16,1/32,1/64,1/128];
 error= zeros(length(H),1);
 for i=1:length(H)
     Omega = buildGitter(H(i));
     [A, b, u_h] = assemble2(H(i), Omega);
     u = solve(A,b,u_h,Omega);
     error(i,1) = infty_error(u,Omega);
 end
 T = table(H',error(:,1),'VariableNames',{'Gitterweite','Fehler'});
 figure;
 loglog(H,error(:,1))
 grid on
 title('$ L_{inf}$ Fehler ueber Gitterweite', 'interpreter', 'latex')
 xlabel("Gitterweite h")
 ylabel('$ L_{inf}$ Fehler','interpreter', 'latex')
end
```

Angewendet ergibt sich damit folgendes Bildchen:

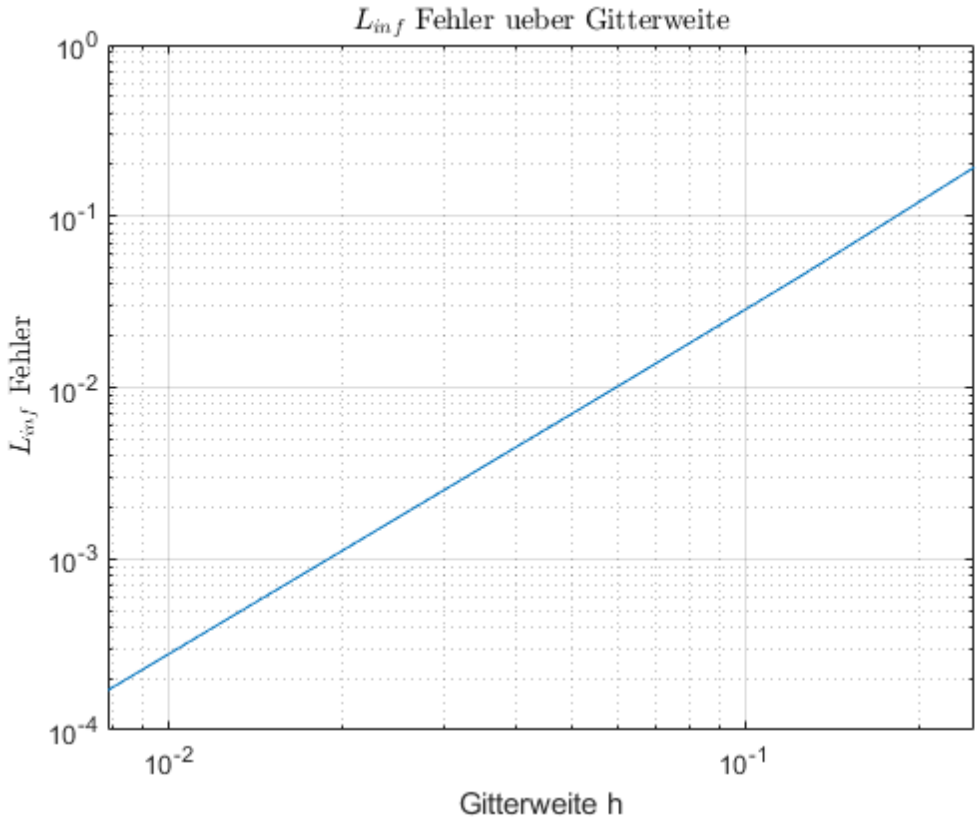
```
T = convergence()
```

```
T =

  6×2 table

    Gitterweite      Fehler
    ___________    _________

          0.25        0.19277
         0.125       0.044762
        0.0625       0.010989
       0.03125       0.002735
      0.015625     0.00068297
     0.0078125     0.00017069
```



Quadratischer Fehlerabfall ist hier jetzt auch sehr schön zu erkennen. Interessanterweise ergibt sich für Gitterweite h=0.5 eine Lösung, welche auf allen ausgewerteten Gitterpunkten fast bis auf Maschinengenauigkeit übereinstimmt. Dies hat uns sehr überrascht:

```
Omega_05 = buildGitter(0.5);
[A_05,b_05,u_05h] = assemble2(0.5,Omega_05);
u_05 = solve(A_05,b_05,u_05h,Omega_05);
error_05 = infty_error(u_05,Omega_05)
```

```
error_05 =

   1.0688e-15
```

Published with MATLAB® R2019b