Ein Blick über den Tellerrand . . . mit FreeFem++

Syntax und mehr

Steffen Weißer

Universität des Saarlandes

11. Dezember 2015

Gliederung

Die ersten Datentypen

- `bool` logischer Wert, entweder `true` oder `false`
- `int` ganzzahliger Wert
- `real` Gleitpunktzahl
- `string` Zeichenkette

Auf Variablen dieser Datentyppen können die üblichen Operationen angewendet werden. Außerdem gibt es auch einige Erweiterungen.

```
1  int i, j=5;
2  real a = j%2 + pi;
3  real b = a^2 + sin(a);
4  bool w = a>4;
5  string s1 = "Hallo";
6  string s2 = s1 + " du da!";
```

Ausgabe

Die Syntax für die Ausgabe ist wie in *C++*. Um einen Text und eine Variable auf der Standardausgabe (`cout`) auszugeben verwendet man beispielsweise:

```
1  cout << "a = " << a << endl;
```

Hierbei erzeugt `endl` einen Zeilenumbruch. Die Ausgabe in eine Datei erfolgt über einen Ausgabestriem (`ofstream`):

```
1  real a = pi/1000.;
2  {
3    ofstream file("datei.txt");
4    file.scientific;  // Format aendern
5    file << "Ob das klappt? " << a << endl;
6  } // Variable file wird geloescht und somit die
     Datei geschlossen.
```

Globale Variablen und System Kommandos

Es gibt einige globale Variablen wie z.B. `x, y, z, label, region, P, N, nu_triangle, ...` deren Namen reserviert sind. Sie dienen als Link zu den FE-Tools.

System Kommandos können mit Hilfe des Befehls `exec` ausgeführt werden. Z.B. unter Linux

```
exec("shell command");
```

oder unter Windows

```
exec("c:\\cygwin\\bin\\ls.exe");
```

Funktionen

- **Mathematische Funktionen:**
  `sin(x)`, `asin(x)`, `log(x)`, `log10(x)`, `exp(x)`, ...
  sowie `ceil(x)`, `floor(x)`, `tgamma(x)`, `erf(x)`, ...
- **Funktionen mit zwei/drei Variablen:**
  Variablen sind hierbei `x`, `y`, `z` und der Rückgabewert skalar

```
1  func f = exp(2*pi*x) * cos(2*pi*y);
```

- **Funktionen (Methoden):**
  Beginnne mit dem Schlüsselwort `func`

```
1  real a = 1.2;
2  func real phi(real t) {
3    return a*sin(t);
4  }
```


- **FE-Funktionen:** später mehr

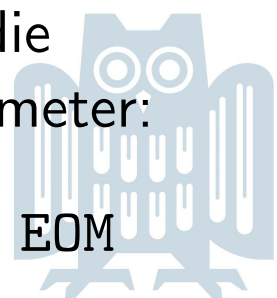
Schleifen und Präprozessor

Die Syntax der Schleifen ist ebenfalls wie in *C++*:

```
1  real sum = 0;
2  for (int i=0; i<4; i++) {
3    sum += i;
4  }
5
6  while (sum > 0) {
7    cout << "sum = " << sum << endl;
8    sum = sum-1; // Fehler bei: sum--;
9  }
```

Ähnlich zur `define`-Direktive in *C* gibt es in *FreeFem++* die Präprozessor-Direktive `macro`, ebenfalls mit und ohne Parameter:

```
macro <name>(<param. list>) <replacement> // EOM
```



Achtung: Das Makro muss mit einem Kommentar beendet werden!

Felder (Vektoren)

```
<Typ> [int] <Name>(<Laenge>);
```

```
1  real [int] tab(10), tab1(10); // 2 array of 10
       real
2  real [int] tab2; // bug array with no size
3  tab = 1.03; // set all the array to 1.03
4  tab[1]=2.15;
5  cout << tab[1] << " " << tab[9] << " size of tab
       = " << tab.n << endl;
6  tab.resize(14); // change the size of array tab
       to 12 with preserving first value
7  tab(10:13)=3.14; // set unset value
8  cout <<" resize tab: " << tab << endl;
9  real [string] tt;
10 tt["+"]=1.5;
11 int[int] ii(0:5); // set array ii to 0,1,...,5
```

Der :-Operator erzeugt ein Feld bzw. Vektor. Hierbei ist `a:b` äquivalent zu `a:1:b`. Die Länge des Vektors `a:c:b` beträgt $\lfloor |(b-a)/c| + 1 \rfloor$ und die i-te Komponente lautet $a + i(b-a)/c$, z.B.

$$
\begin{aligned}
1:5 &= 1,2,3,4,5 \\
1:2:5 &= 1,3,5 \\
1.:1.5:5 &= 1,2.5,4
\end{aligned}
$$

Achtung: **Nicht** den :-Operator verwenden! Seltsames Verhalten!

Desweiteren kann mit Vektoren gerechnet werden. Hierbei bezeichnet a‘ den transponierten Vektor zu a.

```
1  real[int] a(5),b(5),c(5);
2  a =1;
3  a(0:4:2) = 2;
4  a(3:4) = 4;
5  b = a+ a;
6  b += 2*a;
7  b /= 2;
8  c = -a+4*b;
9  c = a .* b;
10 c = a ./ b;
11 //c = b/2; // does not exist
12
13 cout << " ||a||_1 = " << a.l1 << endl;
14 cout << " ||a||_2 = " << a.l2 << endl;
15 cout << " ||a||_infty = " << a.linfty << endl;
16 cout << " sum a_i = " << a.sum << endl;
17 cout << " max a_i = " << a.max << " a[ " << a.
       imax << " ] = " << a[a.imax] << endl;
18 cout << " a'*a = " << (a'*a) << endl;
```

Mehrdimensionale Felder (Matrizen)

```
<Typ> [int,int] <Name>(<Laenge>);
```

```
1  int N=3,M=4;
2  real[int,int] A(N,M);
3  real[int] b(N),c(M);
4  b=[1,2,3];
5  c=[4,5,6,7];
6
7  A=1;              // set the all matrix
8  A(2,:) = 4;       // the full line 2
9  A(1,0:2) = 3; // set the line 1 from 0 to 2
10 A = 2.*b*c';
11 b = A*c;
```

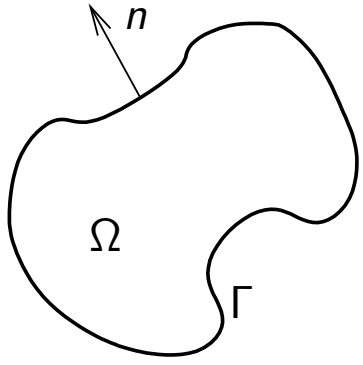
Datentyp matrix

```
real [int,int] A = [[ 0, 1, 0, 10],
                    [ 0, 0, 2, 0],
                    [ 0, 0, 0, 3],
                    [ 4, 0, 0, 0]];
matrix sparseA = A;
cout << sparseA << endl;
sparseA = 0.5 * sparseA; // und so weiter ...
```

```
# Sparse Matrix (Morse)
# first line: n m (is symmetic) nbcoef
# after for each nonzero coefficient:   i j a_ij
    where (i,j) \in  {1,...,n}x{1,...,m}
4 4 0  5
          1           2 1
          1           4 10
          2           3 2
          3           4 3
          4           1 4
```

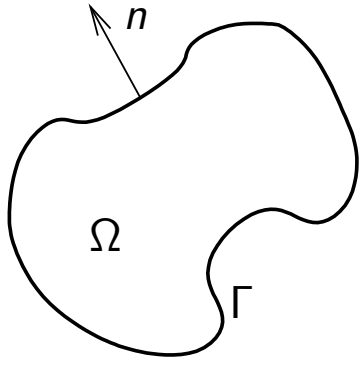
Modellproblem: die Poisson-Gleichung

$$-\Delta u = f \quad \text{in } \Omega$$
$$u = g \quad \text{auf } \Gamma = \partial\Omega$$



Modellproblem: die Poisson-Gleichung

$$-\Delta u = f \quad \text{in } \Omega$$
$$u = g \quad \text{auf } \Gamma = \partial\Omega$$



Variationsformulierung:

Setze $u = u_0 + u_g$ mit $u_0 = 0$ und $u_g = g$ auf $\Gamma = \Gamma_D$. Es folgt

$$\text{Finde } u_0 \in H_0^1(\Omega): \quad a(u_0, v) = \widetilde{F}(v) \quad \forall v \in H_0^1(\Omega)$$

$$\text{wobei} \quad \widetilde{F}(v) = \int_\Omega f\, v\, dx - a(u_g, v)$$

mit

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v\, dx.$$

Galerkin Approximation

$$\text{Finde } u_0 \in V: \quad a(u_0, v) = \widetilde{F}(v) \quad \forall v \in V$$

$$\Downarrow$$

$$\text{Finde } u_{0,h} \in V_h: \quad a(u_{0,h}, v_h) = \widetilde{F}(v_h) \quad \forall v_h \in V_h$$

$$\text{wobei} \quad V_h \subset V \quad \text{mit} \quad \dim V_h < \infty$$

Galerkin Approximation

$$\text{Finde } u_0 \in V: \quad a(u_0, v) = \widetilde{F}(v) \quad \forall v \in V$$

$$\Downarrow$$

$$\text{Finde } u_{0,h} \in V_h: \quad a(u_{0,h}, v_h) = \widetilde{F}(v_h) \quad \forall v_h \in V_h$$

$$\text{wobei} \quad V_h \subset V \quad \text{mit} \quad \dim V_h < \infty$$

Ist $\{\varphi_1, \ldots, \varphi_n\}$ eine Basis von V_h und $u_{0,h} = \sum_{j=1}^{n} \alpha_j \varphi_j$ ergibt sich das lineare Gleichungssystem

$$A\alpha = b \quad \text{mit } A_{ij} = a(\varphi_j, \varphi_i),\ b_i = \widetilde{F}(\varphi_i),\ i, j = 1, \ldots, n.$$

Galerkin Approximation

$$\text{Finde } u_0 \in V: \quad a(u_0, v) = \widetilde{F}(v) \quad \forall v \in V$$

$$\Downarrow$$

$$\text{Finde } u_{0,h} \in V_h: \quad a(u_{0,h}, v_h) = \widetilde{F}(v_h) \quad \forall v_h \in V_h$$

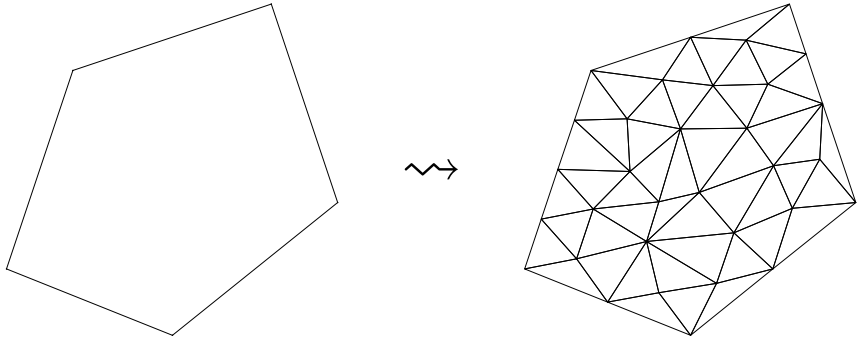
$$\text{wobei} \quad V_h \subset V \quad \text{mit} \quad \dim V_h < \infty$$

Ist $\{\varphi_1, \ldots, \varphi_n\}$ eine Basis von V_h und $u_{0,h} = \sum_{j=1}^{n} \alpha_j \varphi_j$ ergibt sich das lineare Gleichungssystem

$$A\alpha = b \quad \text{mit } A_{ij} = a(\varphi_j, \varphi_i),\ b_i = \widetilde{F}(\varphi_i),\ i, j = 1, \ldots, n.$$

Finite Elemente Methode:

$$\Omega \rightsquigarrow \mathcal{T}_h(\Omega)$$



Netzgenerierung

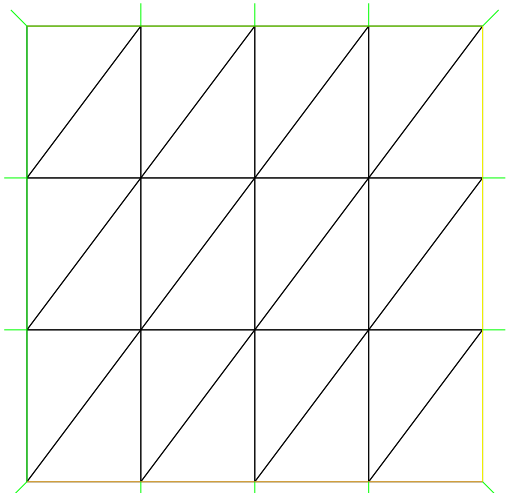
Der Datentyp eines Netzes ist `mesh`. Im folgenden werden verschiedene Methoden zur Netzgenerierung vorgestellt.

Die Diskretisierung des Einheitsquadrats $\Omega = (0,1)^2$.

`square(n,m,...)`

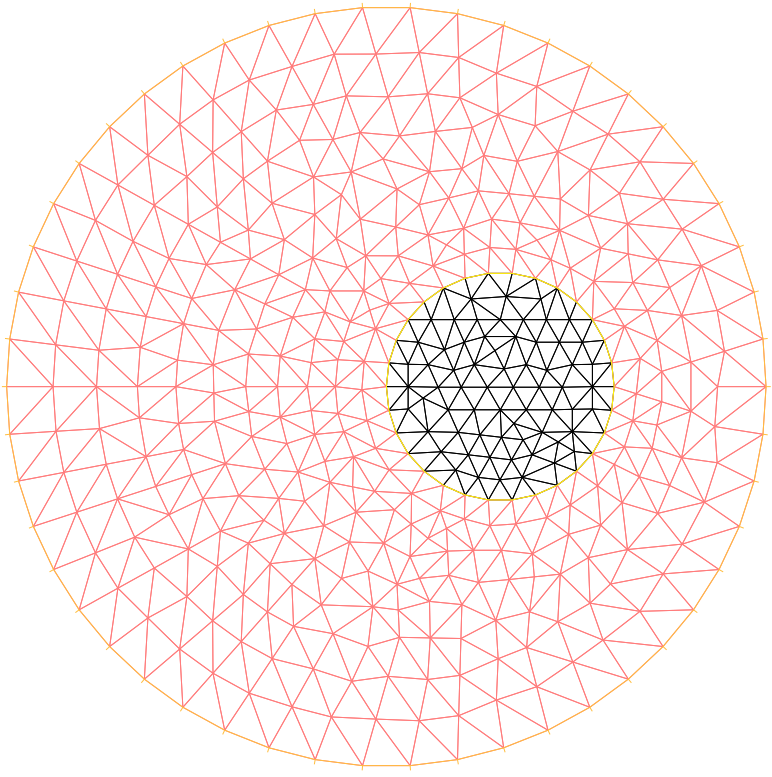
Erzeugt ein Gitter mit m Zeilen und n Spalten aus Dreiecken. Es können weitere Optionen angegeben werden, um ein beliebiges Quadrat $(a,b)^2$ zu diskretisieren und um die Orientierung der Dreiecke zu beeinflussen.

```
1  mesh Th = square(4,3);
2  plot(Th);
```

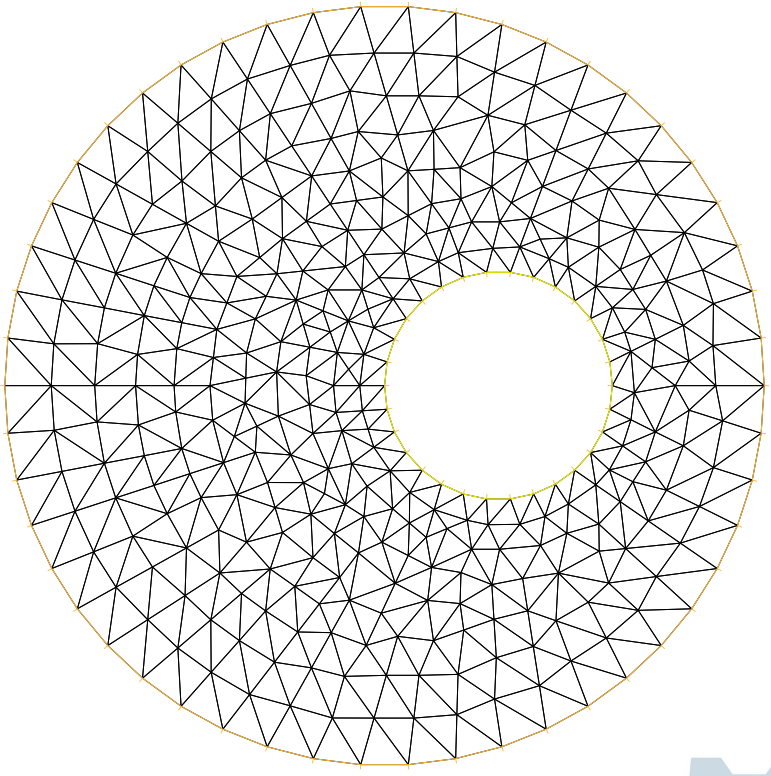


Es kann zunächst der Rand des Gebietes Ω definiert und anschließend ein Netz erzeugt werden.

```
1  border a(t=0,2*pi){ x=cos(t); y=sin(t);label=1;}
2  border b(t=0,2*pi){ x=0.3+0.3*cos(t); y=0.3*sin(
       t);label=2;}
3  plot(a(50)+b(+30)) ; // to see a plot of the
       border mesh
4  mesh Th1= buildmesh(a(50)+b(+30));
5  mesh Th2 = buildmesh(a(50)+b(-30));
6  plot(Th1,wait=1,ps="Thwithouthole.eps");
7  plot(Th2,wait=1,ps="Thwithhole.eps");
8
9  cout << "Part 1 has region number " << Th1(-0.5,
       0).region << endl;
10 cout << "Part 2 has redion number " << Th1( 0.3,
       0).region << endl;
```



Th1



Th2

- Ein Netz kann gespeichert

```
savemesh(Th,"MeinNetz.msh");
```

  und später wieder geladen werden

```
mesh Th = readmesh("MeinNetz.msh");
```

- Für komplexere Geometrien kann ein externes Tool verwendet werden. Das *C*-Programm `triangle` diskretisiert z.B. 2D Gebiete.
- Für 3D Probleme kann die Software `TetGen` verwendet werden. Diese kann direkt aus *FreeFem++* aufgerufen werden.
- Es stehen noch viele weitere Tools zur Verfügung, siehe Dokumentation. Z.B. kann ein Netz deformiert werden (`movemesh()`) oder dem Problem angepasst werden (`adaptmesh()`).

FE-Funktionen bzw. Approximationsraum V_h

Hat man ein Netz, so kann man einen Approximationsraum definieren. Der Datentyp lautet `fespace`. Anschließend können diskrete Funktionen definiert werden.

```
1  mesh Th = square(5,5);
2  fespace Vh(Th,P1);
3  Vh uh;  // Diskrete Funktion
```

Das zweite Argument bei der Definition des Approximationsraums spezifiziert diesen Raum. In *FreeFem++* stehen viele Möglichkeiten zur Auswahl, z.B.

- `P0` stückweise konstante Funktionen
- `P1` stetige stückweise lineare Funktionen
- `P2` stetige stückweise quadratische Funktionen
- `RT0` Raviart-Thomas Finite Elemente vom Grad 0
- ...

Eine diskrete Funktion $u_h \in V_h$ ist immer gegeben als Linearkombination von Basisfuntionen, d.h.

$$u_h(x, y) = \sum_{i=1}^{N} \alpha_i \varphi_i(x, y),$$

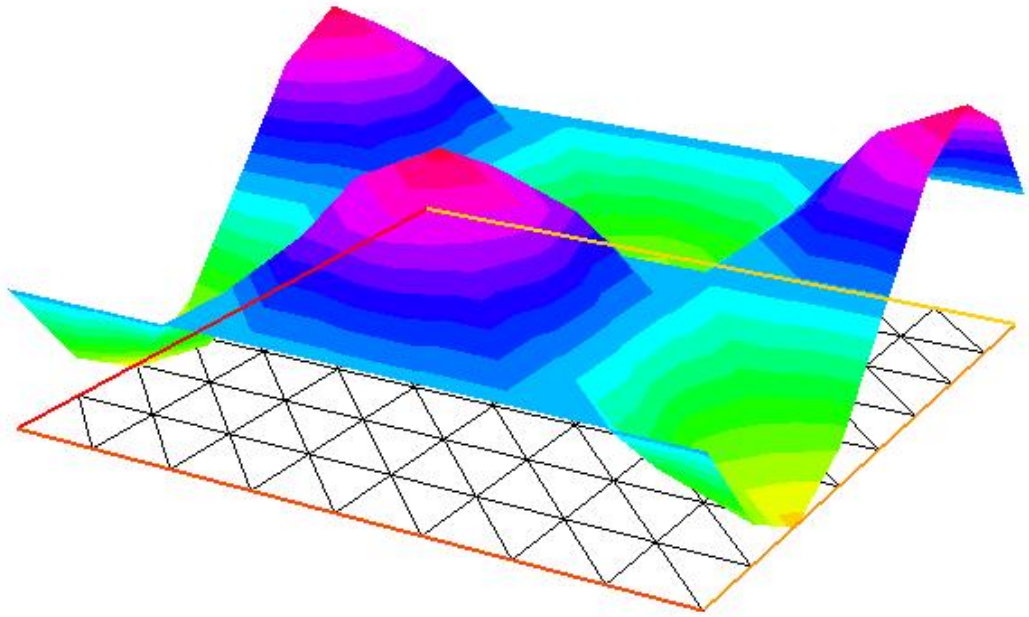
wobei $V_h = \operatorname{span}\{\varphi_1, \ldots, \varphi_N\}$. Den Vektor $(\alpha_1, \ldots, \alpha_N)^\top$ der Koeffizienten erhält man in *FreeFem++* mit `uh[]`. Möchten man also α_2 so schreibt man `uh[][1]`.

Auf diese Weise kann man den maximalen Durchmesser der Dreiecke im Netz bestimmen.

```
1  mesh Th = square(8,8);
2  fespace Ph(Th,P0);
3  Ph h = hTriangle;
4  cout << "size of mesh = " << h[].max << endl;
```

Man erhält ganz leicht aus einer stetigen Funktion eine diskrete.

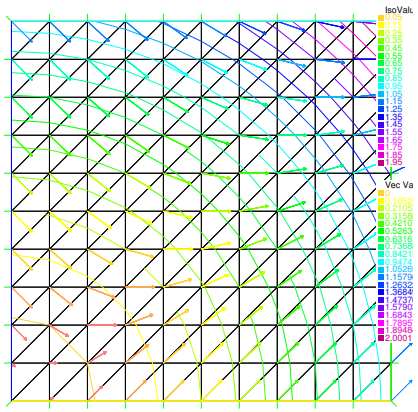
```
1  func f = sin(2*pi*x)*cos(2*pi*y);
2  mesh Th = square(8,8);
3  fespace Vh(Th,P1);
4  Vh fh;  // Diskrete Funktion
5  fh = f;
6  plot(Th,fh);
```

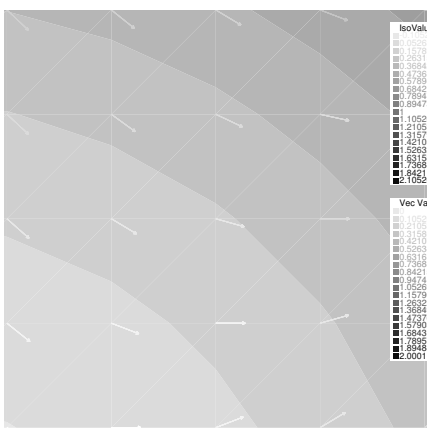


Visualisierung

Den Befehl `plot()` haben wir ja bereits kennen gelernt.

```
1  mesh Th=square(10,10);
2  fespace Vh(Th,P1);
3  Vh uh=x*x+y*y,vh=-y^2+x^2;
4  plot(Th,uh,[uh,vh],value=true,ps="plot1.eps",
       wait=true);
5  // zoom on box defined by the two corner points
       [0.1,0.2] and [0.5,0.6]
6  plot(uh,[uh,vh],bb=[[0.1,0.2],[0.5,0.6]],grey=1,
       wait=true,fill=1,value=1,ps="plot2.eps");
```





Zur Untersuchung von Approximationsfehlern eignet sich der Umweg über `gnuplot`. Zunächst schreibt man eine Tabelle in eine Datei (z.B. `daten.dat`) und visualisiert diese anschließend mit `gnuplot`. Dies ist auch direkt aus `FreeFem++` möglich, siehe Dokumentation.

Ein entsprechender Kommandozeilenbefehl für `gnuplot` lautet z.B.

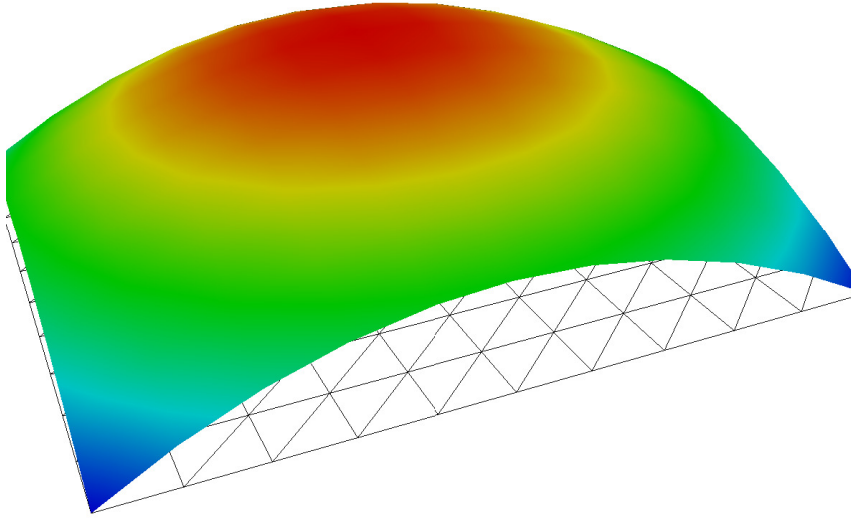
```
1 gnuplot -e "plot 'daten.dat' w lp; pause -1;"
```

Beispiel für ein FreeFem++-Programm.

```
1  int i, N=10;
2  real [int] hmax(10);
3  border a(t=0,2*pi){ x=cos(t); y=sin(t);}
4
5  for (i=1; i<=N; i++) {
6    mesh Th = buildmesh(a(10*i));
7    fespace Ph(Th,P0);
8    Ph h = hTriangle;
9    hmax[i-1] = h[].max;
10   plot(Th,wait=true);
11 }
12
13 {
14   ofstream file("daten.dat");
15   file.scientific;
16   for (i=0; i<N; i++)
17     file << i << "\t" << hmax[i] << endl;
18 }
```

Ein weiteres Tool zur Visualisierung ist die externe Software `medit`. Diese kann ebenfalls direkt aus `FreeFem++` aufgerufen werden.

```
1  load "medit"
2  mesh Th=square(10,10,[2*x-1,2*y-1]);
3  fespace Vh(Th,P1);
4  Vh uh=2-x*x-y*y;
5  medit("mm",Th,uh);
```



Lösen des Modellproblems

Für die Approximation $u_h = u_{0,h} + u_{g,h}$ mit der diskreten Fortsetzung $u_{g,h}$ der Randdaten g haben wir:

$$\text{Finde } u_{0,h} \in V_h : \quad a(u_{0,h}, v_h) = \widetilde{F}(v_h) \quad \forall v_h \in V_h$$
$$\text{wobei} \quad V_h \subset V \quad \text{mit} \quad \dim V_h < \infty$$

Lösen des Modellproblems

Für die Approximation $u_h = u_{0,h} + u_{g,h}$ mit der diskreten Fortsetzung $u_{g,h}$ der Randdaten g haben wir:

$$\text{Finde } u_{0,h} \in V_h : \quad a(u_{0,h}, v_h) = \widetilde{F}(v_h) \quad \forall v_h \in V_h$$
$$\text{wobei} \quad V_h \subset V \quad \text{mit} \quad \dim V_h < \infty$$

Dies schreiben wir für *FreeFem++* wieder in der Form:

$$\int_\Omega \nabla u_h \cdot \nabla v_h \, dx - \int_\Omega f v_h \, dx = 0 \quad \text{und} \quad u_h|_\Gamma = g$$

für $u_h, v_h \in V_h$.

Achtung: V_h diskretisiert hier $H^1(\Omega)$ anstelle von $H_0^1(\Omega)$.

$$\int_\Omega \nabla u_h \cdot \nabla v_h \, dx - \int_\Omega f v_h \, dx = 0 \quad \text{und} \quad u_h|_\Gamma = g$$

```
1  func f = 10;
2  func g = sin(x)*sin(y);
3  mesh Th = square(10,10);
4  fespace Vh(Th,P1);
5  Vh uh, vh;
6  problem ModProb(uh,vh) =
7        int2d(Th)( dx(uh)*dx(vh) + dy(uh)*dy(vh) )
8      - int2d(Th)( f*vh ) + on(1,2,3,4,uh=g);
9  ModProb;  // Problem loesen
10 plot(Th,uh,wait=true);
```

$$\int_\Omega \nabla u_h \cdot \nabla v_h \, dx - \int_\Omega f v_h \, dx = 0 \quad \text{und} \quad u_h|_\Gamma = g$$

```
1  func f = 10;
2  func g = sin(x)*sin(y);
3  mesh Th = square(10,10);
4  fespace Vh(Th,P1);
5  Vh uh, vh;
6  problem ModProb(uh,vh) =
7        int2d(Th)( dx(uh)*dx(vh) + dy(uh)*dy(vh) )
8      - int2d(Th)( f*vh ) + on(1,2,3,4,uh=g);
9  ModProb;  // Problem loesen
10 plot(Th,uh,wait=true);
```

- **int2d(Th)** Integriert über dem diskretisierten Gebiet in 2D, es gibt auch 1D und 3D Integrationen über Gebieten sowie über Oberflächen, siehe Dokumentation Kapitel 6.10.
- **on()** Spezifiziert die Dirichlet Randdaten, Ziffern sind die `label` der Randstücke (bei `square()` lauten diese 1,2,3,4).

Für die Berechnung des Gradienten kann ein `macro` verwendet werden. Im folgenden Beispiel sind auf einem Teil des Randes Neumann Randbedingungen vorgeschrieben.

```
1  macro grad(u) [dx(u), dy(u)] // EOM
2  ...
3  problem ModProb(uh,vh) =
4            int2d(Th)( grad(uh)'*grad(vh) )
5          - int2d(Th)( f*vh )
6          - int1d(Th,3,4)( gN*vh )
7          + on(1,2,uh=gD);
8  ...
```

Alternativ kann man auch das lineare Gleichungssystem aufstellen und lösen: Sei $u_h = \sum_{j=1}^{n} \alpha_j \varphi_j$ so ist

$$A\alpha = b \quad \text{mit} \quad A_{ij} = \int_\Omega \nabla\varphi_j \cdot \nabla\varphi_i \, dx, \quad b_i = \int_\Omega f\varphi_i \, dx.$$

```
1  func f = 10; func g = sin(x)*sin(y);
2  mesh Th = square(10,10);
3  fespace Vh(Th,P1);
4  Vh uh, vh;
5  real [int] b(Vh.ndof);
6  varf a(uh,vh) =
7       int2d(Th)( dx(uh)*dx(vh) + dy(uh)*dy(vh) )
8     + on(1,2,3,4,uh=g);
9  varf F(uh,vh) =
10      int2d(Th)( f*vh ) + on(1,2,3,4,uh=g);
11 matrix A = a(Vh,Vh); // Berechne Matrix
12 b      = F(0,Vh);    // Berechne rechte Seite
13 uh[] = A^-1*b;       // Loese LGS
14 plot(Th,uh,wait=true);
```

- Mit `varf` kann man eine Bilinearform (Variationsformulierung!?) angeben.
- Bei der Konstruktion der rechten Seite muss man einen Umweg über eine Bilinearform machen.
- Die Dirichlet Daten müssen sowohl bei der Bilinearform als auch bei der rechten Seite angegeben werden. Dies obwohl die rechte Seite nicht von u_h abhängt.
- Man kann bei der Konstruktion der Matrix und bei der Problemstellung den Algorithmus und Parameter angeben, mit dem das LGS gelöst werden soll. Zur Verfügung stehen z.B. `LU, CG, Cholesky, GMRES, UMFPACK, ...`

```
1  matrix A = a(Vh,Vh,solver=CG);
2  ...
3  problem ModProb(uh,vh,solver=UMFPACK) = ...
```

Fazit

- einfache Syntax
- schnelle Implementierung kleiner Probleme
- einfaches Testen von Theorien
- es ist etwas Vorsicht geboten