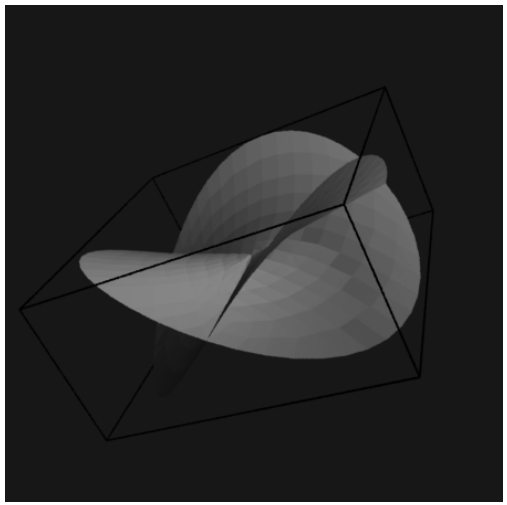
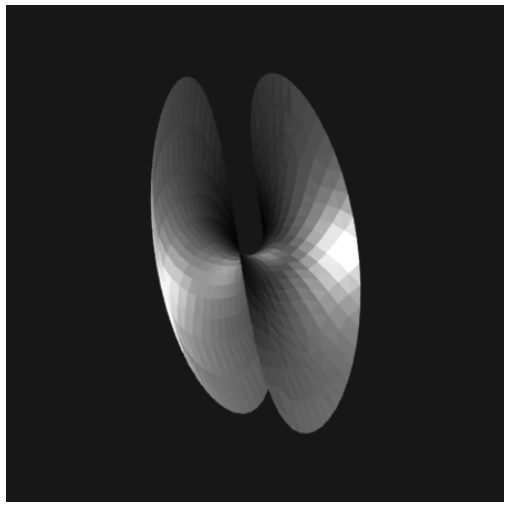
Diskrete Minimalflächen

Matthias Hofmann, Michael Rehme und Nadine Schlotz

3. Januar 2014

Die Aufgabenstellung





„Beschreiben Sie eine triangulierte Fläche in eine gegeben gekrümmte Raumkurve ein und versuchen Sie, eine Approximation an eine Minimalfläche zu berechnen.“

Vorgehensweise

- Einbeschreibung einer Fläche $\Longrightarrow$ Triangulierung.
- Einzelschrittverfahren zur Minimierung des Gesamtoberflächeninhalts
- Darstellung mittels `Geomview`

Hierfür benötigen wir noch eine geeignete Struktur.

klassen.h

```
1   #ifndef KLASSEN_H_
2   #define KLASSEN_H_
3
4   #include <iostream>
5   #include <fstream>
6   #include <list>
7   #include<vector>
8   #include<cstdlib>
9   #include<math.h>
10
11  using namespace std;
12
13  const unsigned int dim=3;
14
15  class Gitter; //forward declaration
```

```
1   //Vektor
2   class victor {
3       public:
4   bool isRand;
5           vector<double> v;
6           list<pair<int, int> > dreiecke;
7           victor();
8           victor(const vector<double> arg);
9
10          victor& operator+=(const victor& arg);
11          victor operator+(const victor& arg)const;
12
13          victor operator-(const victor& arg)const;
14          victor& operator-=(const victor& arg);
15
16          victor& operator*=(const double& arg);
17          victor operator*(const double& arg)const;
18
19          victor& operator/=(const double& arg);
20          victor operator/(const double& arg)const;
21
22
23          double operator*(const victor& arg)const;
24
25          void clear();
26
27          void ausgeben();
28  };
```

```
1   //Dreieck
2   class Dreieck
3   {
4       public:
5   Dreieck();
6           Dreieck(Gitter* vater);
7           Dreieck(Gitter* vater,int a, int b, int c);
8           virtual ~Dreieck();
9           int punkte[3];
10          Gitter* papa;
11  \begin{lstlisting}
12  //Dreieck
13  class Dreieck
14  {
15      public:
16  Dreieck();
17          Dreieck(Gitter* vater);
18          Dreieck(Gitter* vater,int a, int b, int c);
19          virtual ~Dreieck();
20          int punkte[3];
21          Gitter* papa;
```

```
1   //Dreieck
2   class Dreieck
3   {
4       public:
5   Dreieck();
6           Dreieck(Gitter* vater);
7           Dreieck(Gitter* vater,int a, int b, int c);
8           virtual ~Dreieck();
9           int punkte[3];
10          Gitter* papa;
11
12          // gibt die flaeche des dreiecks zurueck
13          double flaeche();
14          // gibt den gradienten an einer ecke (also 0 = 1. ecke, 1 = 2. ecke,
15          // 2 = 3. ecke) zurueck
16          victor gradient(int ecke);
17  };
```

```
1   //Punkteklasse
2   class Punkt
3   {
4       public:
5           Punkt();
6           Punkt(victor v, double t);
7           virtual ~Punkt();
8           Punkt& operator=(const Punkt& other);
9
10          victor Ort;
11          double parameter;
12  };
```

```
1   //Gitter
2   class Gitter
3   {
4       public:
5           Gitter();
6           Gitter(const vector<Punkt> arg1,const vector<Dreieck> arg2 );
7           Gitter(int mode);
8           std::vector<Punkt> gib();
9           virtual ~Gitter();
10          std::vector<Punkt> punkte;
11          std::vector<Dreieck> dreiecke;
12
13          void finde();
14          victor gradient(Dreieck* arg, int ecke);
15          double Oberflaeche();
16          void verbessere();
17          void verbessere(int arg);
18          void Verfeinere(int mode);
19          void Verfeinere(int mode, int arg);
20  };
```

```
1  //Methods
2  double norm(const victor& arg);
3  void vcout(victor arg);
4  void Gcout(const Gitter& arg);
5  victor def(double a, double b);
6  victor def(double a, double b,double c);
7  victor cross(victor a, victor b);
8  victor randkurve(double t, int mode);
9  Punkt randpunkt();
```

Zusammenfassung

- Wir haben eine Vektorenklasse `victor` und verfügen über Grundrechenarten, Skalarprodukt und Kreuzprodukt.
- Die Punkteklasse `Punkt` basiert im wesentlich aus einem Vektor und einem Paramter zur zugehörigen Parametrisierung.
- Wir unterscheiden zwischen Rand- und inneren Punkten. Randpunkte bleiben fest und sind durch den Wahrheitswert isRand zu erkennen.

- Unter Dreiecken verstehen wir ein Trippel aus Indizes der Punkteliste.
- Im Gitter sammeln wir eine Punkte- und Dreiecksliste. Insbesondere kann `Gitter::Oberflaeche()` die Oberfläche der Triangulierung berechnen.
- Zudem besitzt jeder Punkt über den Vektor `Ort` eine Liste sämtlicher anliegender Dreiecke. Die Erstellung dieser Liste erfolgt durch `Gitter::finde()`.
- Die Triangulierung wird in `Gitter::Verfeinere()` bewerkstelligt.
- Das Einzelschrittverfahren zur Minimierung wird in `Gitter::verbessere()` implementiert.

Triangulierung

Für gegebene Parametrisierung $\gamma : [0,1] \to \mathbb{R}^3$ einer geschlossenen Kurve erhält man eine Starttriangulatur, als das Dreieck gegeben durch $\gamma(\frac{1}{3}), \gamma(\frac{2}{3}), \gamma(1)$.

Verfeinerung

1. Halbiere jede Seite
2. Verschiebe ihn gegebenfalls auf den Rand ($p_{\text{neu}} = \frac{p_1+p_2}{2}$), falls es sich um eine Randkante handelt. (Trick: Betrachte Anzahl umliegender Dreiecke)
3. Erzeuge neue Dreiecke aus den eben definierten Punkten.

Gitter::Verfeinere()

```
1   //Sucht zu allen Punkten die zugehoerigen Dreiecke
2   void Gitter::finde() {
3       // leere Listen
4       for (unsigned int j = 0; j < punkte.size(); j++) {
5           punkte[j].Ort.dreiecke.clear();
6       }
7       for (unsigned int i = 0; i < dreiecke.size(); i++) {
8           punkte[dreiecke[i].punkte[0]].Ort.dreiecke.push_back(make_pair(i, 0));
9           punkte[dreiecke[i].punkte[1]].Ort.dreiecke.push_back(make_pair(i, 1));
10          punkte[dreiecke[i].punkte[2]].Ort.dreiecke.push_back(make_pair(i, 2));
11      }
12  }
```

```
1  //Verfeinerungsalgorithmus in Kooperation mit den anderen Grupppen, leicht modifiziert
2  void Gitter::Verfeinere(int mode) {
3   // enthaelt zu (a,b) den Punkt c.Ort = (a.Ort+b.Ort)/2, wobei a,b,c
4   //mithilfe von punktindizes definiert werden
5  std::map<std::pair<int, int>, int> erstelltepunkte;
6  std::vector<Dreieck> neuedreiecke = std::vector<Dreieck>();
7  int pcnt = 0;
8  int dcnt = 0;
9  for (unsigned int i = 0; i < dreiecke.size(); ++i) {
10 int np[3];
11 for (int j = 0; j < 3; ++j) {
12     int a = dreiecke[i].punkte[j];
13     int b = dreiecke[i].punkte[(j + 1) % 3];
14
15     if (erstelltepunkte.find(std::pair<int, int>(a, b)) == erstelltepunkte.end()
16             && erstelltepunkte.find(std::pair<int, int>(b, a))
17                     == erstelltepunkte.end()) {
18         Punkt c = Punkt();
19         c.Ort = (punkte[a].Ort + punkte[b].Ort) * 0.5;
20         if (punkte[a].parameter < punkte[b].parameter)
21             c.parameter = (punkte[a].parameter + punkte[b].parameter) * 0.5;
22         else
23             c.parameter = (1.0 + punkte[a].parameter + punkte[b].parameter)*0.5;
```

```
1            np[j] = punkte.size();
2            c.Ort.isRand = 0;
3            punkte.push_back(c);
4            pcnt++;
5            erstelltepunkte.insert(
6            std::pair<std::pair<int, int>, int>(std::pair<int, int>(a, b),np[j]));
7            } else {
8            if (erstelltepunkte.find(std::pair<int, int>(a, b))
9                == erstelltepunkte.end()) {
10               np[j] = erstelltepunkte[std::pair<int, int>(b, a)];
11               } else {
12                    np[j] = erstelltepunkte[std::pair<int, int>(a, b)];
13           }
14       }
15   }
```

```
1   // neue dreiecke erstellen
2   for (int k = 0; k < 3; ++k) {
3       Dreieck d = Dreieck(this, dreiecke[i].punkte[k], np[k], np[(k + 2) % 3]);
4       neuedreiecke.push_back(d);
5       dcnt++;
6   }
7   Dreieck e = Dreieck(this, np[0], np[1], np[2]);
8   neuedreiecke.push_back(e);
9   dcnt++;
10  }
11  dreiecke = neuedreiecke;
12  std::cout << "Erstellte " << pcnt << " neue Punkte und " << dcnt
13      << " neue Dreiecke" << std::endl;
14
15  this->finde();
16   //Wenn ein Punkt in weniger als 6 Dreiecken vertreten ist, ist es ein Randpunkt
17  for (unsigned int i = 0; i < punkte.size(); i++) {
18  if (punkte[i].Ort.dreiecke.size() < 6) {
19      punkte[i].Ort = randpunkt(punkte[i].parameter, mode).Ort;
20      punkte[i].Ort.isRand=1;
21  }
22  }
23  }
```

Gradientenverfahren

- Zu einer gegebenen Triangulierung, soll an dieser Stelle das Gitter optimiert werden. $\Longrightarrow$ Minimierung des Flächeninhalts aller angrenzenden Dreiecke eines Punktes.
- Für diese Optimierungsaufgabe lässt sich ein Abstiegsverfahren/Näherungsverfahren ausnutzen.

Flächeninhalt eines Dreiecks

Der Flächeninhalt eines Dreiecks a, b, c berechnet sich zu:

$$A = \frac{1}{2}|(b-a) \times (c-a)|$$

Der Gradient zeigt in Richtung Höhe des Dreiecks. (ohne Beweis)
Also:

$$\nabla_a A \propto ((c-a) \times (c-b)) \times (c-b)$$

Die Gesamtgradientenrichtung ergibt sich dann durch Aufsummation. Für das Abstiegsverfahren negieren wir.

Gitter::verbessere()

```
1   //Berechnet den Gradienten an einem gegebenen Eck.
2   victor Dreieck::gradient(int ecke)
3   {
4       victor a = papa->punkte[punkte[(ecke+1)%3]].Ort-papa->punkte[punkte[ecke]].Ort;
5       victor b = papa->punkte[punkte[(ecke+1)%3]].Ort-papa->punkte[punkte[(ecke+2)%3]].Ort;
6
7       victor c = cross(cross(a,b),b);
8       double nc = norm(c);
9       if(nc <= 0.0001)return def(0,0,0);
10      return c*(norm(b)/norm(c));
11  }
12
13  //Berechnet die Oberflaeche der Gitterstruktur
14  double Gitter::Oberflaeche() {
15  double neuegesamtflaeche=0;
16  for(int i = 0; i < dreiecke.size(); ++i)
17          {
18              neuegesamtflaeche += dreiecke[i].flaeche();
19          }
20  return neuegesamtflaeche;
21  }
```

```
1   // Verbessert das Gitter
2   void Gitter::verbessere() {
3   victor grad, grad_tmp, tmp;
4   double flaeche_ref;
5   double flaeche;
6   double faktor; //Schrittweite
7   double armijo=0.01; //Armijo-koeffizient
8   double neueflaeche=this->Oberflaeche();
9   double alteflaeche;
10  do {
11  alteflaeche=neueflaeche;
12  for (unsigned int i = 0; i < punkte.size(); i++) {
13  //Gradientenverfahren
14  flaeche_ref = 0;
15  grad.clear();
16
17  faktor = 2; //Schrittweite
18  //Berechne Gradienten
19  if (punkte[i].Ort.isRand == 0) {
20  for (list<pair<int, int> >::iterator it =
21  punkte[i].Ort.dreiecke.begin();
22  it != punkte[i].Ort.dreiecke.end(); ++it) {
23  grad += dreiecke[it->first].gradient(it->second);
24  flaeche_ref += dreiecke[it->first].flaeche();
25  }
```

```
1   //repeat-Schleife
2   do {
3   //double ngrad=grad*grad;
4   flaeche=0;
5
6   tmp=punkte[i].Ort;
7   faktor*=0.5;
8   punkte[i].Ort-=grad*faktor; //update Punkt
9   for (list<pair<int,int> >::iterator it = punkte[i].Ort.dreiecke.begin();
10  it != punkte[i].Ort.dreiecke.end(); ++it) {
11
12  flaeche+=dreiecke[it->first].flaeche();
13  }
14  punkte[i].Ort=tmp;
15  } while (flaeche>flaeche_ref-armijo*faktor*(grad*grad));
16  grad_tmp.clear();
17  for (list<pair<int,int> >::iterator it = punkte[i].Ort.dreiecke.begin();
18  it != punkte[i].Ort.dreiecke.end(); it++) {
19  grad_tmp+=dreiecke[it->first].gradient(it->second);
20  }
21  //Wir akzeptieren unsere Verbesserung nur wenn sie die Armijo-Bedingung erfuellt
22  punkte[i].Ort-=grad*faktor;
23  grad=grad_tmp;
24  flaeche_ref=flaeche;
25  }
26  }
27  neueflaeche=this->Oberflaeche();
28  //cout << alteflaeche-neueflaeche <<endl;
29  } while(fabs((alteflaeche-neueflaeche)/neueflaeche)>1e-7); //Berechne die Verbesserung
30  //Falls die relative Verbesserung einer gewissen Toleranz unterliegt brechen wir ab.
31  }
```

main.cpp

```
1   //Ausgabe fuer Geomview.
2   void Gcout(const Gitter& arg) {
3   ofstream file;
4   file.open("Test");
5   file << "OFF" << endl;
6   file << arg.punkte.size() << " " << arg.dreiecke.size() << " " << 4 << endl;
7   std::vector<Punkt> arg2 = arg.punkte;
8   for (unsigned int i = 0; i < arg2.size(); i++) {
9   file << arg2[i].Ort.v[0] << " " << arg2[i].Ort.v[1] << " "
10  << arg2[i].Ort.v[2] << endl;
11  }
12  std::vector<Dreieck> arg3 = arg.dreiecke;
13  for (unsigned int i = 0; i < arg3.size(); i++) {
14  file << 3 << " " << arg3[i].punkte[0] << " " << arg3[i].punkte[1] << " "
15  << arg3[i].punkte[2] << endl;
16  }
17  file.close();
18  }
```

```
1   int main(){
2   // Variablen
3   int mode, n;
4   double alteflaeche=0, neueflaeche=0;
5
6   // Nachkommastellen
7   cout.setf(ios::fixed, ios::floatfield);
8   cout.precision(8);
9
10  // Programmstruktur
11  cout << "Waehlen Sie eine Kurve aus:" <<endl;
12  cout << "(1) Tennisballkurve" << endl;
13  cout << "(2) sich selbst schneidende Kurve" << endl;
14  cout << "(3) Kreiskurve" << endl;
15  cin >> mode;
16  cout << "Anzahl der Verfeinerungschritte:" <<endl;
17  cin >> n;
18  cout << "Initialisiere" << endl;
19  Gitter g=Gitter(mode);
20  }
```

```
1   g.finde();
2   cout << "Oberflaeche: (Anfangszustand) " << g.Oberflaeche()<<endl;
3   for(int i=0; i<n; i++){
4   g.Verfeinere(mode);
5   alteflaeche=g.Oberflaeche();
6   cout <<"Oberflaeche nach Verfeinerung: " << alteflaeche <<endl;
7   g.verbessere(5);
8   neueflaeche=g.Oberflaeche();
9   cout <<"Oberflaeche nach Minimierungsschritt: " << neueflaeche <<endl;
10  cout <<"relative Verbesserung nach Minimierungsschritt: " <<  fabs((alteflaeche-neueflaeche)/a
11  }
12  cout <<"Ausgabe... Sie koennen nun mit Geomview die Datei 'Test' aufrufen" << endl;
13  Gcout(g);
14  return 0;
```