Outline

Überblick

Einleitung

Wir erklaeren wie ein einfaches eindimensionales Diffusions-Problem mit finiten Differenz geloest werden kann. Dies geschieht in einem Ein-Datei-Programm mit einfachen Strukturen. Dies wird der Startpunkt sein. Spaeter werden wir komplexere Probleme und komplexere Diskretisierungen betrachten und hierfuer komplexere Datenstrukturen und fortgeschrittene Programmierparadigmen verwenden.

Die Ziele dieser Uebung sind

- (kurze) Einfuehrung der finite Differenzen Methode
- “Warm werden” mit C++
- Implementierung eines einfachen finite Differenzen Loesers

Neue Funktionalitaet

Um Finite Differenzen zu implementieren benoetigen wir die folgenden einfachen Funktionen:

- Vektoren um die Funktionswerte auf dem Gitter abzuspeichern
- Eine Funktion, die man an den Gitterpunkten x_i auswerten kann

Ausserdem wollen wir ein wenig Zusatzfunktionalitaet:

- Ausgabe der Loesung in eine Datei, die das Visualisieren der Ergebnisse ermoeglicht
- Kontrolle ueber Diskretisierungsparameter ueber die Eingabe

Mehr nehmen wir uns zunaechst nicht vor. Was man mehr machen kann und schon bald betrachtet wird, erfahren Sie im Ausblick.

Überblick

Das Poisson Problem

Wir betrachten das Poisson-Problem in einer Raumdimension auf dem Gebiet $\Omega = (0, 1)$. Hier soll die Differentialgleichung

$$-\partial_{xx} u(x) = f(x), \quad x \in \Omega$$

geloest werden. Als Randwerte betrachten wir zunaechst

$$u(0) = u(1) = 0$$

und nehmen an, dass eine eindeutige Loesung existiert mit $u \in C^2(\Omega) \cap C^0(\bar{\Omega})$.

Das Gitter

Die Idee der Finite Differenzen Methode ist die Darstellung der Loesung als Gitterfunktion, d.h. man beschreibt die Funktion nur durch die Funktionswerte an ausgewaehlten Punkten. Zunaechst waehlen wir ein aequidistantes Gitter. Ueber Beziehungen von Nachbarwerten zueinander werden dann die Ableitungen (approximiert). Wir betrachten das Gitter

$$\mathcal{T} = \{x_i = \frac{i}{N}, i = 0, ..N\}$$

mit den Gitterpunkten $x_i \in \mathcal{T}$ und der Indexmenge $\mathcal{I} = \{0, .., N\}$.

Approximation von Ableitungen (zentral)

Eine einfache Ableitung wird approximiert mit der (zentralen) finiten Differenz

$$\partial_x u \approx \frac{u(x_{i+1}) - u(x_{i-1})}{2h}.$$

Hier ist h der Abstand von zwei Gitterpunkten. Motivieren laesst sich diese Wahl als Sekante durch die Funktionswerte an den Punkten $x - h$ und $x + h$ welche als Approximation der Ableitung am Punkt x benutzt wird.

Approximation von Ableitungen (links-/rechtsseitig)

Alternativ kann man auch mit links- oder rechtseitigen Differenzen, z.B.

$$\partial_x u \approx \frac{u(x_{i+1}) - u(x_i)}{h}$$

arbeiten.

Analyse der Genauigkeit von Finiten Differenzen

Zur Analyse der Genauigkeit der Verfahren wird ueblicherweise eine Taylor-Entwicklung herangezogen (Uebungsaufgabe!). Hierbei zeigt sich, dass die zentrale Differenz einen Approximationsfehler der Ordnung $O(h^2)$ hat wobei links- und rechtsseitige Differenzen einen Fehler der Ordnung $O(h)$ aufweisen. Mit der Einbeziehung von mehr Nachbarpunkten koennen auch finite Differenzen hoeherer Ordnung erzielt werden. Wir werden dies aber nicht tiefergehend betrachten.

Approximation zweiter Ableitungen (zentral)

Fuer die zweite Ableitung erhaelt man die uebliche finite Differenzen Approximation:

$$\partial_{xx} u \approx \frac{u(x_{i+1}) - 2u(x_i) + u(x_{i-1})}{h^2}$$

Finite Differenzen fuer das Poisson-Problem

Bei der Finite Differenzen Methode wird jetzt fuer jeden Gitterpunkt eine Gleichung formuliert. In unserem Fall erhalten wir

$$\frac{1}{h^2}(-u_{i+1} + 2u_i - u_{i-1}) = f_i, \quad i \in \mathcal{I} \setminus \{0, N\}, u_0 = 0, u_N = 0$$

mit $u_j = u(x_j)$ und $f_j = f(x_j), j \in \mathcal{I}$.

Die Gitterfunktion

Wir loesen dies zunaechst mit einem sehr einfachen (und nicht sehr effizienten!) Loesungsverfahren. Dafuer interpretieren wir die Werte an den Gitterpunkte als Eintraege eines Vektors $u \in \mathbb{R}^n$, $n = N + 1$.

Das Richardson-Verfahren

Wir benutzen das Richardson-Verfahren (hier bezeichnet der oben stehende Index die Iteration)

$$u_i^k = u_i^{k-1} + \omega \underbrace{\left(\frac{1}{h^2}(-u_{i+1}^{k-1} + 2u_i^{k-1} - u_{i-1}^{k-1}) - f_i \right)}_{\textit{Residuum}}$$

mit dem Daempfungsparameter $\omega > 0$. Wir waehlen $\omega = h^2/2$. Wenn das Verfahren konvergiert, erhalten wir die diskrete Loesung zu unserem Problem.

Überblick

Ein Ein-Datei-Programm

Zur Umsetzung des Finite-Differenzen-Loesers betrachten wir ein einfaches Programm in C++, welches wir Schritt fuer Schritt einfuehren. Im Wesentlichen erzeugt die Dokumentation in diesem Abschnitt die Datei `poisson1D.cc`.

Inhaltsbeschreibung / Lizenz

```
1  // This file is part of the Praktikum zur Vorl. Num. PDGL, Wintersemes
2  // http://dune-project.uni-muenster.de/repos/projects/praktikum-numpde
3  // Copyright holders: Christoph Lehrenfeld, Felix Schindler
4  // License: BSD 2-Clause License (http://opensource.org/licenses/BSD-2
```

Header-Dateien (Binde Definitionen von anderen Bibliotheken ein)

Einbinden von nuetzlichen Klassen, zur Ausgabe von Informationen in Dateien oder den Ausgabestrom und dem Arbeiten mit “Vektoren” und mathematischen Funktionen.

```
5  #include <iostream>
6  #include <fstream>
7  #include <vector>
8  #include <cmath>
```

Benutzung von namespaces

Wir wollen gewisse externe Funktionaltitaet verwenden koennen. Diese sind typischweise in Namensraeume (“namespaces”) abgelegt um Eindeutigkeit von Bezeichnungen zu erleichtern. Ein typischer Namensraum mit viel Funktionalitaet ist “std”. Wir werden diesen einbinden mit der folgenden Anweisung:

```
9  using namespace std;
```

Beachte: Wenn man groessere Projekte bearbeitet und mehrere Bibliotheken einbindet, sollte man vorsichtig sein beim Aufloesen der Namensraeume.

Die Funktion *f*

Wir implementieren zunaechst die rechte Seite *f* als Funktion. Die Argumente sind so gewaehlt, dass die Funktion eine Abbildung von "double" (die Menge der Maschinenzahlen *Z* sind eine Teilmenge von $\mathbb{Q}$) nach "double" ist.

```
10  double myf(double x)
11  {
12    return sqrt(x)+sqrt(1-x)+1-2.0*x;
13  }
```

`main()` : Argumente

In dieser Funktion wird das komplette Problem geloest. Dies ist die Funktion die aufgerufen wird, wenn das Programm startet. Spaeter werden wir die Problemloesung in Komponenten aufteilen und diese auslagern.
Diese Hauptfunktion main() besitzt zwei Parameter mit den Namen argc (Argumentenzaehler) und argv (Argumentenvektor) welche Kommandezeilenargumente an das Programm uebergeben. Wird das Programm z.B. mit

```
./poisson1D 40
```

aufgerufen so ist argc$= 2$ und argv$= \{./poisson1D, 40\}$. Dieses Interface kann (spaeter) genutzt werden um Diskretisierungsparameter zu uebergeben.

main() : Diskretisierungsparameter

```
14  int main(int argc, char* argv[])
15  {
```

Wir legen den Diskretisierungsparameter *N* fest:

```
16  const int N = 50;     //number of intervals
17  const int n = N+1;    //number of points (nodes)
```

Hieraus errechnet sich die Gitterweite *h*

```
18  const double h = 1.0/N;
```

main() : Vektoren

Nun legen wir verschiedene Vektoren der Laenge n bzw. $n-2$ an und benutzen dafuer den Vektor aus der C++-Standardbibliothek

```
19  vector<int> innerpoints(n-2);   // indices of inner points [1,...,N-1
20  vector<int> points(n);          // indices of all points [0,...,N]
21  vector<double> x(n);            // coordinates of points
22  vector<double> f(n);            // function values at points
23  vector<double> u(n);            // solution values at points
```

Hierbei geben die spitzen Klammern ‹·› den Datentyp der Vektoreintraege an. Jeder Eintrag ist also vom Typ “int”, also ganzzahlig oder vom Typ “double”, also skalarwertig.

`main()` : Initialisierung der Vektoren (1)

Initialisierung der Koordinaten:

```
24  for (int i=0; i<n; ++i)
25    points[i] = i;
26
27  for (int i=0; i<n-2; ++i)
28    innerpoints[i] = i+1;
```

Wir haben hierbei zwischen `innerpoints` und `points` unterschieden:

- An den Stellen `innerpoints` muss die Loesung bestimmt werden mit Hilfe der Finite Differenzen Beziehungen.
- Zusammen mit den Randwerten ergibt sich die Loesung an allen Punkten `points` welche zur Visualisierung der Loesung von Noeten ist.

`main()` : Initialisierung der Vektoren (2)

Die Vektoren werden initialisiert mit den zugehoerigen (Start-)werten.

```
29  for (int i=0; i<n; ++i)
30  {
31    x[i] = (double)i / N;        // grid coordinate
32    f[i] = myf(x[i]);            // function val. of f at x_i
33    u[i] = 0.0;                  // initial value for u
34  }
```

main() : Richardson-Iteration 1

Jetzt definieren wir die Iterationsschleife

```
35  vector<double> u_old(n);          // last iteration vector
36  double res_max = 0.0;             // max_norm of residuum of the equa
37  const double damping = 0.5*h*h;   // damping parameter
38  for (int k=0; k<100*N*N; ++k)
39  {
40    res_max = 0.0;
41    u_old = u;       // store values of old iteration step
42    for (int i:innerpoints)
43    {
44      const double cur_res = 1.0/(h*h) * (- u_old[i+1] + 2.0 * u_old[i]
45      u[i] = u_old[i] - damping * cur_res;
46      if (res_max < abs(cur_res))
47        res_max = abs(cur_res);
48    }
```

main() : Richardson-Iteration 2

Iterationszaehlerausgabe:

```
49    // progress output:
50    cout << "\riteration: \t" << k << ": \t" << res_max;
51    // if a certain residuum size is reached, stop:
52    if (res_max < 1e-6)
53      break;
54
55  }
56
57  cout << endl;
```

main() : Loesungsausgabe

Wir wollen die Resultate unserer Rechnung ausgeben. Dafuer benutzen wir den Ausgabestream `ofstream`. Die Ausgabe erfolgt in einem csv-artigen Format. Jede Zeile entspricht einem Punkt. In einer Zeile sind Punkt und Funktionswert mit einem Komma getrennt.

```
58    //creation of output stream
59    ofstream outstr("poisson1D.out");
60    //output of result vector:
61    for (int i:points)
62      outstr << x[i] << "," << u[i] << "\n";
63
64  }
```

Überblick

Das Makefile

Um den Quellcode in eine ausfuehrbare Datei zu uebersetzen benutzen wir ein “Makefile” dieses ruft Compiler und Linker mit den passenden Optionen auf.

```
1  CXX_FLAGS = -O0 -Wall -g --std=c++0x
2
3  PROGS=poisson1D
4
5  all: $(PROGS)
6
7  clean: \
8  rm -f *.o data* $(PROGS)
9
10 default:
11 poisson1D: poisson1D.cc; \
12 $(CXX) -o $@ $< $(CXX_FLAGS)
```

Überblick

Uebersetzen

Um das Programm zu uebersetzen fuehrt einfach den Befehl aus:

```
make
```

Ausfuehren

Um nun auch das Programm auszufuehren, fuehrt er das erzeugte Programm aus:

```
./poisson1D
```

Loesung visualisieren

Zur Visualisierung der Ergebnisse benutzen wir gnuplot, also rufen wir gnuplot auf:

```
gnuplot
```

In der Konsole von gnuplot plotten wir dann die Daten, die wir zuvor in poisson1D.out gespeichert hatten:

```
set datafile separator ","
plot "poisson1D.out" w lp
```

Alternative Ausgabe und Visualisierung mit Paraview

Alternativ kann man auch nach jedem Iterationsschritt das Ergebnis ausgeben. Zum Beispiel indem man folgenden Code in die Iterationsschleife integriert:

```
//creation of output stream
ofstream outstr("out/poisson1D.csv." + to_string(k));
//output of result vector:
for (int i:points)
  outstr << x[i] << "," << u[i] << "\n";
```

Diese Datenreihen koennen z.B. gut mit `Paraview` visualisiert werden.

Überblick

Aus dieser Dokumentation erzeugte Dateien

Die Schritte, die oben erklaert wurden, resultieren in den folgenden Dateien:

- lesson01_src.tar.gz
- poisson1D.pdf
- poisson1D.beamer.pdf
- poisson1D.html

Weitere Dateien

Zur (Re-)Formatierung von `poisson1D.cc` wurde das Programm `uncrustify` mit der Konfigurationsdatei `uncrust.cfg` verwendet.

```
uncrustify -c uncrust.cfg --replace poisson1D.cc
```

Überblick

Mit dem Problem arbeiten (1):

Wie verhaelt sich die Anzahl der noetigen Iterationen in Abhaengigkeit der Gitterweite?

Mit dem Problem arbeiten (2):

Aendern Sie Randbedingungen und die rechte Seite sodass das Problem

$$-u_{xx} = \pi^2 \sin(\pi x), \quad u(0) = 0, u(1) = 0$$

geloest wird. Visualisieren Sie die Loesung und den Fehler (berechnen Sie dafuer zunaechst die analytische Loesung). Wie haengt der Fehler von der Gitterweite ab?

Mit dem Problem arbeiten (3):

Wir betrachten nun das instationaere Waermeleitproblem

$$\partial_t u - \partial_{xx} u = f, \quad x \in \Omega, \quad u(0,t) = u(1,t) = 0, u(x,0) = u_0(x)$$

Zur Diskretisierung der Zeitableitung benutzen wir die Finite-Differenzen-Approximation

$$\partial_t u(x_i) = 1/(\Delta t)(u_i^n - u_i^{n-1})$$

Vergleichen Sie die hiermit erhaltene Diskretisierung mit der fuer das stationaere Problem mit dem Iterationsverfahren. Was faellt Ihnen aus? Wie ist Δt zu waehlen?

Programmierung(1):

Fuegen Sie den Konvektionsterm $a \cdot u_x$ hinzu und loesen Sie das Problem

$$(\partial_t u) - \partial_{xx} u + a \partial_x u = 1, \quad u(0) = u(1) = 0$$

mit den Parametern $a \in \{1, 10, 100\}$. Benutzen Sie dafuer die Diskretisierung mit zentralen Differenzen. Aendern Sie die Gitterweite durch die Anzahl der Elemente. Was beobachten Sie?

Programmierung(2):

Lagern Sie die Randbedingungen, rechte Seite und Anfangswerte in andere Funktionen aus und lassen die Gitterweite bzw. `N` als Parameter ueber die Konsole uebergeben.

Programmierung(3):

Nicht-aequidistante Gitter: Aendert das Programm so, dass die Gitterweite auf dem linken Gebiet $(0.5, 1)$ kleiner ist als auf auf dem rechten Gebiet $(0, 0.5)$. Achtung: Wie muss sich das Finite Differenzen Verfahren anpassen?

Überblick

Diskretisierung / Loeser:

- Loese Gleichungssystem systematische (direkte Loeser / iterative Loeser)
- Finite Volumen / Finite Elemente
- Mehrdimensionale Probleme / Diskretisierungen
- Fehlerschaetzer / Adaptivitaet
- Nichtlineare Probleme

Programmierung:

- Objektorientiertes Programmieren: Funktionalitaet Kapseln
  - Gitter unabhaengig von Diskretisierung (gemeinsamer Nenner)
  - Gitterfunktion
  - Iterationsverfahren / Lineare Löser / Zeitschrittverfahren
  - Operatoren (FD/FV/FEM)
  - Input/Output