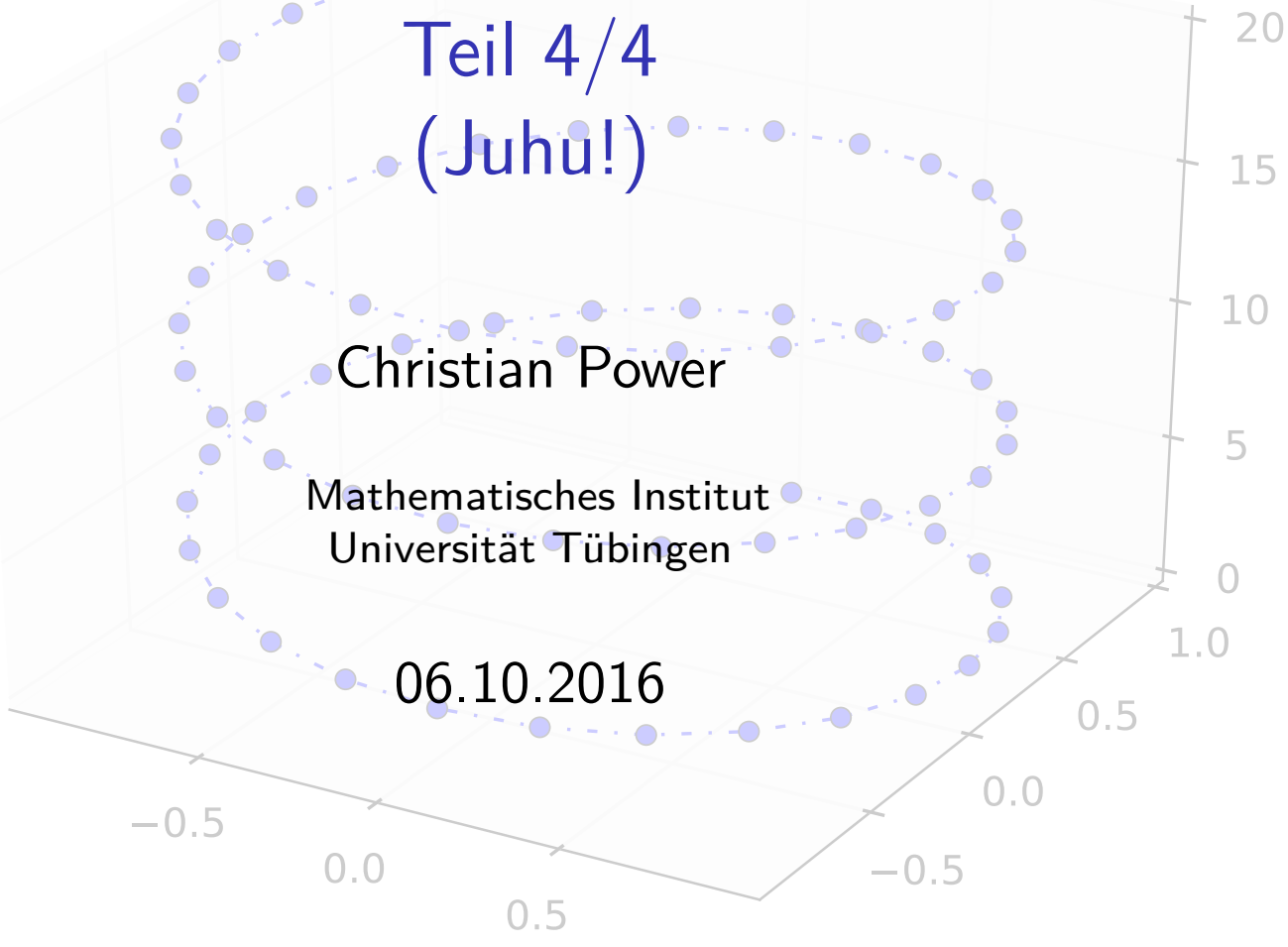
Programmiervorkurs für die Numerik
Teil 4/4
(Juhu!)

Christian Power

Mathematisches Institut
Universität Tübingen

06.10.2016



Wiederholung

In diesem Kurs haben wir gelernt

- mit Kontrollstrukturen zu programmieren (`if` und `for`),
- Funktionen zu benutzen um unsere Programme zu organisieren,
- mit Matrizen und Vektoren um zu gehen,
- errechnete Daten mit Plots zu visualisieren.

Gliederung

Lösungen für die Aufgaben des 3. Übungsblattes

Aufgabe 1:

```
1 function [V,O] = kugel(r)
2 % Berechne Volumen V und Oberflaeche r
3 % fuer Radius r
4 assert(r>0)
5 V = 4/3*pi*r^3;
6 O = 4*pi*r^2;
7 assert(V>0 & O>0)
8 end
```

Lösungen für die Aufgaben des 3. Übungsblattes

Aufgabe 2 (Hilfsfunktion):

```
1 function [y] = geo_R(z,n)
2 % Geometrische Reihe bis n mit z
3 y = 0;
4 for i = 0:n
5     y = y + z^i;
6 end
7 end
```

Lösungen für die Aufgaben des 3. Übungsblattes

Aufgabe 2 (Skript):

```
1 % Blatt 3 Auf. 2
2 clear all
3 z = .25;
4 for n = [5 10 50 100]
5     geo_R(z,n)
6 end
```

Lösungen für die Aufgaben des 3. Übungsblattes

Aufgabe 3:

```
1  function [C] = matmult(A,B)
2  % Matrizen Multiplikation
3  n = size(A,1);
4  m = size(A,2);
5  k = size(B,2);
6  assert(m == size(B,1))
7
8  C = zeros(n,k);
9  for i=1:n
```

Lösungen für die Aufgaben des 3. Übungsblattes

Aufgabe 3 (Fortsetzung):

```
1  C = zeros(n,k);
2  for i=1:n
3      for j=1:k
4          S = 0;
5          for l=1:m
6              S = S + A(i,l)*B(l,j);
7          end
8          C(i,j) = S;
9      end
10 end
11 end
```

Lösungen für die Aufgaben des 3. Übungsblattes

Aufgabe 4 (Hilfsfunktion):

```
 1  function [y] = plotThis(x)
 2  xlen = length(x);
 3  y = zeros(xlen);
 4  for i=1:xlen
 5       if x(i) < 0
 6            y(i) = -x(i)*x(i);
 7       elseif x(i) >= 0 & x(i) <= 1
 8            y(i) = x(i);
 9       else % x(i) > 1
10            y(i) = x(i) * x(i) + 1;
11       end
12  end
13  end
```

Lösungen für die Aufgaben des 3. Übungsblattes

Aufgabe 4 (Skript):

```
1 % Blatt 3 Aufg. 4
2 x = -2:.1:2;
3 plot(x,plotThis(x))
4 title('Blatt 3 Aufg. 4')
5 xlabel('-2\leq x \leq 2')
6 ylabel('plotThis()')
7 grid on
```

Lösungen für die Aufgaben des 3. Übungsblattes

Aufgabe 5 (Hilfsfunktion):

```
1 function [y] = fschar(b,x)
2 % fschar(x) = bx^2
3 % Precondition: b ist ein Skalar
4 y = b * x .* x;
5 end
```

Lösungen für die Aufgaben des 3. Übungsblattes

Aufgabe 5 (Skript):

```
% Blatt 4 Aufg. 5
x = -2:.1:2;
b = [.5 1 1.5 2];
y = zeros(length(b), length(x));
for i = 1:length(b)
    y(i,:) = fschar(b(i),x);
end
plot(x,y)
title('Blatt 4 Aufg. 5')
xlabel('-2\leq x \leq 2')
ylabel('bx^2')
legend('b=.5', 'b=1', 'b=1.5', 'b=2')
```

Gliederung

Performance

- **Performance** ist das Zeitverhalten eines Programms.
- Performance ist wichtig, wenn der Anwender oder der Programmiertutor es ausdrücklich verlangt.
- In allen anderen Fällen sollte Performance erst am Schluss optimiert werden; wenn überhaupt.
- Donald Knuth, den Erfinder von TEX, stammt das Zitat „Premature optimization is the root of all evil“. Gemeint ist damit, dass wenn man die Performance optimiert, bevor das Programm überhaupt fertig ist, sich selbst ein Bein stellt.
- Performance wird in MATLAB mit `tic`, `toc` und in Julia mit `tic()`, `toc()` gemessen.

Zeit messen

in Julia

```
1  """
2  b3a4fun(x) :R -> R mit drei
3  Definitionsbereiche.
4  Für x < 0 gilt f(x)=-x^2,
5  für 0 ≤ x ≤ 1 gilt f(x) = x
6  und für 1 < x gilt f(x) = x^2 + 1
7  """
8  function b3a4fun(x::Real)
```

Zeit messen

in Julia

```
 1  function b3a4fun(x::Real)
 2      rv
 3      if x < 0
 4          rv = -x^2
 5      elseif x <= 1 # x >= 0 gilt autom.
 6          rv = x
 7      else # x > 1
 8          rv = x^2 + 1
 9      end
10  end # function
```

Zeit messen

in Julia

```
 1  function b3a4fun(x::Array{Float64,1})
 2      rv = zeros(length(x))
 3      ind_set = x .< 0
 4      rv[ind_set] = -x[ind_set].^2
 5      ind_set = 0 .<= x .<= 1
 6      rv[ind_set] = x[ind_set]
 7      ind_set = x .> 1
 8      rv[ind_set] = x[ind_set].^2 + 1
 9      return rv
10  end # function
```

Zeit messen

in Julia

```
1  x = [a for a=-2:.0001:2] # 40001 elem
2  tic()
3  b3a4fun(x[:])
4  toc()
5  tic()
6  for elem in x
7      b3a4fun(elem)
8  end
9  toc()
```

Gliederung

Bugs

- Im Schnitt besteht Programmieren darin, dass man (im Besten Fall) 90% der Zeit nach seinen eigenen Fehlern sucht.
- Programmierfehler werden als **Bugs** bezeichnet.
- **Debuggen** ist der Vorgang absichtlich **Bugs** zu finden und zu beheben.
- Alle Bugs können i.A. nicht vollständig eliminiert werden. Der *letzte Bug* ist unter Programmierern als ironischer Scherz zu verstehen.
- Es gibt aber Techniken mit denen man Bugs finden kann oder, noch besser, vorbeugen kann.

Arten von Bugs

- Im Allgemeinen unterscheidet man zwischen drei verschiedenen Fehlern:
  1. Fehler vom Compiler.
  2. Fehler zur Laufzeit.
  3. Logische Fehler.
- Mit Compiler-Fehler schlagen sich typischerweise Studenten am Anfang viel herum. Diese Fehler sind aber die dankbarsten, denn der Computer sagt euch, dass etwas falsch ist. Als Beispiel seien Syntax-Fehler genannt wie z.B. `A = * A`.
- Wenn das Programm vom Computer akzeptiert wird, kann es immer noch zur Laufzeit abbrechen. Als Beispiel sei ein Abbruch durch `assert()` genannt.
- Logische Fehler sind mit Abstand am schwierigsten aus zu machen. In diesen Fall ist das Programm aus Sicht des Computers fehlerfrei, aber das Ergebnis ist trotzdem falsch. Als Beispiel sei die naive Lösung der Aufgabe 4 von Blatt 3 genannt.

Fehlerquellen

für Numerik relevant

- *Schwammige Vorgaben* von Programmteilen, z.B. eine Funktion erledigt zu viele Aufgaben.
- *Unvollständiges Programm* während des Programmieren, z.B. wir haben uns noch nicht Gedanken über negative oder komplexe Zahlen gemacht.
- *Unerwartete Argumente* für Funktionen, z.B. `matmult` mit Vektoren.
- *Unerwarteter Zustand* von Daten, z.B. eine andere Funktion überschreibt aus versehen die Matrix `A`.
- *Logische Fehler*.

Gliederung

Debuggen

- Debuggen funktioniert ungefähr so:
  1. Sorge dazu, dass das Programm vom Compiler akzeptiert wird.
  2. Sorge dazu, dass es keine Fehler zur Laufzeit gibt.
  3. Sorge dazu, dass das Programm das tut, was es tun soll.
- Dieser Algorithmus wird so lange wiederholt, bis man mit dem Ergebnis zufrieden ist.

Debuggen

- *Vorsicht:* So funktioniert Debuggen nicht!

  `while` Das Programm funktioniert nicht
    Schaue zufällig in meinen Code und ändere es,
    so dass es besser aussieht.
  `end`

- Offensichtlich ist das ein schlechter Algorithmus. Er wird aber von vielen Studenten praktiziert, wenn sie sich verloren fühlen oder ohne Anhaltspunkt nach Fehlern in ihren Programm suchen.
- Die entscheidende Frage beim Debuggen lautet:

  Woran könnte ich festmachen, dass mein Programm das Richtige ausrechnet?

  Solang man diese Frage nicht beantworten kann, findet man sich im oben genannten schlechten Algorithmus wieder.

Praktische Typs für das Debuggen

- Mit kryptische Fehler Meldungen vom Compiler kann man so umgehen: Kommentiere ca. die Hälfte des Codes aus. Wenn der Fehler noch da ist, wiederhole den Prozess. Wenn der Fehler weg ist kommentiere die Hälfte des aus kommentierten Codes wieder ein und wiederhole den Prozess.
- Logische Fehler können in Laufzeit-Fehler verschoben werden indem man `assert()` benutzt.
- Darüber hinaus können in Julia Typen wie z.B. `Number`, `Real`, `Bool`, `ComplexReal` oder `ArrayReal, 2` benutzt werden Anforderungen an Argumente zu formulieren. Die Funktion `function myfun(x::Real)` akzeptiert z.B. kein Matrizen. Dafür muss eine zusätzliche Funktion `function myfun(x::Array{Real, 2})` bereit gestellt werden.

Praktische Typs für das Debuggen

- Kommentiere euren Code sinnvoll, z.B. muss jede Funktion ein Kommentar besitzen, was seine Funktion im Programm beschreibt.
- Teile deinen Code (sinnvoll) mit mehr Funktionen auf („Teile und herrsche“). Man kann nur Fehler beheben die man auch sehen kann, was nicht der Fall ist, wenn die Funktionen zu lang sind.
- Benutze Tab-Einrückungen um deinen Code (logisch) zu strukturieren.

Geheimtipp: Testen

- **Testen** ist die systematische Suche nach Fehlern im Code.
- Der Unterschied zu Debuggen liegt darin, dass man beim Testen absichtlich versucht seinen Code zum Absturz zu bringen.
- Es gibt viele verschiedene Arten von Tests wie z.B. Blackbox-Tests, Whitebox Tests, Regressions-Tests oder Mocking, die hier nicht behandelt werden können.

Ein einfacher Test

Eine einfache aber bereits sinnvolle Art von Testen funktioniert wie folgt:

- Man legt für das jeweilige Numerik Projekt eine Datei namens `progA1_test` an.
- Bevor man sich die Details einer Funktion für das Projekt überlegt, denkt man sich zunächst einfach nur den Namen und einen genauen Kommentar, was die Funktion überhaupt machen soll.
- Anschließend schreibt man mehrere `assert()`, die die obige (noch nicht fertig geschriebene) Funktion erfüllen soll in `progA1_test`. Es ist gut, wenn man systematisch mit `for` und/ oder $\mathrm{rand}()$ verschiedene Werte testet (falls möglich).

Ein einfacher Test

- Nachdem man sich die Tests überlegt hat, schreibt man die Funktion, die man später benutzen möchte.
- `progA1_test` muss danach ohne Fehler laufen.
- *Wichtig:* Niemals einen Test löschen. Das hat den folgenden Grund: Wenn man die Funktion im später ändern möchte, dann muss sie alle bereits bestanden Tests wieder bestehen.

Gliederung

Übungsblätter

In Numerik praktizieren wir das Kreuz-System.

- Am Anfang der Übungsgruppe wird gekreuzt, welche Aufgaben bearbeitet wurden.
- Der Tutor entscheidet nach eigenem Ermessen, wer die Aufgabe vorrechnen soll.
- Wenn es beim Vorrechnen schwere Verständigungsprobleme zwischen Tutor und Student gibt, wird das ganze Blatt aberkannt.
- Wiederholt sich das mehrmals, werden alle Kreuze aberkannt.
- 50% der Kreuze sind notwendig um zur Klausur zugelassen werden.

Programmieraufgaben

- Es gibt jedes zweite Blatt mindestens ein Programmieraufgabe.
- Es dürfen Aufgaben zu Dritt abgegeben werden. Das muss aber auf jeder Datei oben klar kenntlich gemacht werden.
- Die Abgabe muss als zip-Datei nach dem Muster

  `ProgA1_name1_name2_name3.zip`

  abgegeben werden. Es ist nicht notwendig alle Dateien in ein extra Ordner zu packen.
- Es wird ausdrücklich gefordert, dass alle Plots in der zip-Datei dabei sein müssen.

Noch fragen?