Ein kleiner Programmierkurs

Danny Pingitzer
email: pingitzer@stud.uni-heidelberg.de

In Zusammenarbeit mit
Peter Bastian
Universität Heidelberg
Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Im Neuenheimer Feld 205, D-69120 Heidelberg
email: peter.bastian@iwr.uni-heidelberg.de

11. Februar 2022

Beschreibung des Kurses

- Kurze Einführung in LINUX
- Einrichten eines LINUX-Subsystems für Windows 10
- Programmierung in C++ unter LINUX
- Idee des Kurses: „Lernen an Beispielen“, keine rigorose Darstellung
- Keine Vorkenntnisse erforderlich
- Übungsaufgaben zur Vertiefung
- Möglichkeit Fragen zu stellen

Fragen über CodiMD

1. `https://codimd.mathphys.stura.uni-heidelberg.de/`
2. Notiz erstellen und mir dann die url schicken.
3. Beispiel für das Format der Nachricht:
4. `https://codimd.mathphys.stura.uni-heidelberg.de/ProgrammierkursTest?view`
5. Am besten Programmcode + Fehlermeldung (falls vorhanden) + Problembeschreibung (Wo euer Verständnisproblem liegt)

Inhaltsverzeichnis

Warum Programmieren lernen?

- Auch interessant für das Lehramt:
- Hilft beim (detaillierten) Verständnis von Algorithmen
- Zur Kontrolle von Lösungen
- Visualisierung
- Für die Lehre

Beispiel: Lösung eines LGS

- Für kleine Matrizen noch von Hand machbar
- Für N = 100+ oder sogar 1000+ relativ schwierig
- Außerdem sehr Fehleranfällig
- Ein Programm schafft auch sehr große Matrizen in sehr kurzer Zeit
- Relativ fehlerfrei, außer natürlich Numerik (Rundungsfehler, etc.)

Linux-Basics

Man hat zwei Möglichkeiten Linux zu bedienen:

- graphische Benutzeroberfläche (GUI)
- Texteingabe (Shell)
- GUI einfach zu erlernen, ähnlich wie bei Windows
- Shell ist deutlich produktiver, aber erfordert Lernen der Syntax (am Anfang etwas ungewohnt)

Wichtige Befehle

cd - Wechselt das momentane Arbeitsverzeichnis.

- Syntax: cd <Dateipfad>
- Pfadangabe meist relativ zum Arbeitsverzeichnis
- Relative Pfadangabe in Linux:
- Momentanes Verzeichnis: .
- Unterverzeichnis: ./<Verzeichnis>
- Oberverzeichnis: ..

Wichtige Befehle

- ls - Zeige Inhalt des aktuellen Verzeichnisses.
- mkdir <Verzeichnis> - Erstelle neues Verzeichnis
- cp <datei1> <datei2> - Kopiere datei1 auf datei2 (datei2 kann verzeichnis sein)
- mv wie cp nur verschieben anstatt kopieren
- rm <datei> - datei löschen
- rm -rf Lösche Verzeichnis mit Inhalt

Weitere Befehle

- sudo <Befehl> Befehl als Administrator ausführen
- man <Befehl> - zeigt einem die Bedienungsanweisung für den Befehl
- Für alles Weitere: linux cheat sheet bei google eingeben
- Zusammenstellung der wichtigsten Befehle

LINUX unter Windows 10

- Nicht alle haben einen LINUX-Rechner zuhause.
- Windows 10 hat dafür einen "LINUX-Emulator"
- Installation:
  1. Windows Powershell öffnen
  2. Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-Linux eingeben
  3. Computer Neustarten falls gefordert
  4. Im Microsoft Store Ubuntu runterladen & installieren
  5. Auch zu finden unter: https://docs.microsoft.com/de-de/windows/wsl/install-win10

Aufgabe 1: Einrichten eurer Arbeitsumgebung

- Entweder Ubuntu oder Ubuntu unter Windows 10 installieren
- Git installieren
- HDNum repository runterladen
- Befehl: git clone `https://parcomp-git.iwr.uni-heidelberg.de/Teaching/hdnum.git`
- Eigenen Ordner erstellen

Was ist Programmierung?

- Ein Programm ist im praktischen Sinne eine Aneinanderreihung von Befehlen, die ein Computer (in Reihenfolge) auszuführen hat.
- Es gibt verschiedene Programmiersprachen in denen die Befehle geschrieben werden können.
- Wir verwenden C++.
- In der Regel (auch in C++) ist Groß- und Kleinschreibung zu beachten.

Workflow

C++ ist eine „kompilierte“ Sprache. Um ein Programm zur Ausführung zu bringen sind folgende Schritte notwendig:

1. Erstelle/Ändere den Programmtext mit einem **Editor**.
2. Übersetze den Programmtext mit dem **C++-Übersetzer** (auch C++-Compiler) in ein Maschinenprogramm.
3. Führe das Programm aus. Das Programm gibt sein Ergebnis auf dem Bildschirm oder in eine Datei aus.
4. Falls Ergebnis nicht korrekt, gehe nach 1!

Hallo Welt !

Öffne die Datei `hallohdnum.cc` mit einem Editor:

```
$ gedit hallohdnum.cc
```

```
1  // hallohdnum.cc
2  #include <iostream>     // notwendig zur Ausgabe
3  #include <vector>
4  #include "hdnum.hh"     // hdnum header
5
6  int main ()
7  {
8    std::cout << "Numerik␣0␣ist␣ganz␣leicht!" << std::endl;
9    std::cout << "1+1=" << 1+1 << std::endl;
10
11   return 0;
12 }
```

- `iostream` ist eine sog. „Headerdatei“
- `#include` erweitert die „Basissprache“.
- `int main ()` braucht man immer: „Hier geht's los“.
- `{ ... }` klammert Folge von Anweisungen.
- Anweisungen werden durch Semikolon abgeschlossen.

Hallo Welt laufen lassen

- Gebe folgende Befehle ein:
  - Zum Kompilieren des Programms:

    ```
    $ g++ -o hallohdnum -I../../  hallohdnum.cc
    ```

  - g++ ist unser C++-Compiler
  - -o hallohdnum spezifiziert die dabei entstehende Datei
  - -I../../ spezifiziert das Verzeichnis der Headerdateien
  - hallohdnum.cc ist die Datei, die wir kompilieren wollen
  - Zum Ausführen des Programms:

    ```
    $ ./hallohdnum
    ```

- Dies sollte dann die folgende Ausgabe liefern:

```
Numerik 0 ist ganz leicht!
1+1=2
```

(Zahl-) Variablen

- Aus der Mathematik: „$x \in M$“. Variable x nimmt einen beliebigen Wert aus der Menge M an.
- Geht in C++ mit: `M x;`
- **Variablendefinition**: `x` ist eine Variable vom **Typ** `M`.
- Mit **Initialisierung**: `M x(0);`
- Wert von Variablen der „eingebauten“ Typen ist sonst nicht definiert.

```
1  // zahlen.cc
2  #include <iostream>
3
4  int main ()
5  {
6    unsigned int i; // uninitialisierte natuerliche Zahl
7    double x(3.14); // initialisierte Fliessommazahl
8    float y(1.0);   // einfache Genauigkeit
9    short j(3);     // eine 'kleine' Zahl
10   std::cout << "(i+x)*(y+j)=" << (i+x)*(y+j) << std::endl;
11
12   return 0;
13 }
```

Datentypen

Datentypen sind sozusagen Wertebereiche der Variablen, bestimmen also welche Werte eine Variable annehmen kann.
Die wichtigsten Datentypen:

- int: Ganze Zahl im Wertebereich $[-2^{31}, 2^{31} - 1]$
- unsigned int: Natürliche Zahl im Wertebereich $[0, 2^{32} - 1]$
- float: Gleitkommazahl mit einfacher Präzision
  im Bereich [-3.4e38, 3.4e38]
- double: float mit doppelter Präzision
  im Bereich [-1.80e+308, 1.80e+308]
- char: (einzelne) ASCII-Zeichen ("Buchstaben"),
  z.b. "b","f","C","%",
- string: Zeichenketten, z.b. "Hallo","fjofj"

Zuweisung

- Den Wert von Variablen kann man ändern. Sonst wäre es langweilig :-)
- Dies geht mittels Zuweisung:

```
double x(3.14); // Variablendefinition mit Initialisierung
double y;       // uninitialisierte Variable
y = x;          // Weise y den Wert von x zu
x = 2.71;       // Weise x den Wert 2.71, y unverändert
y = (y*3)+4;    // Werte Ausdruck rechts von = aus
                // und weise das Resultat y zu!
```

Blöcke

- Block: Sequenz von Variablendefinitionen und Zuweisungen in geschweiften Klammern.

```
{
  double x(3.14);
  double y;
  y = x;
}
```

- Blöcke können rekursiv geschachtelt werden.
- Eine Variable ist nur in dem Block *sichtbar* in dem sie definiert ist sowie in allen darin enthaltenen Blöcken:

```
{
  double x(3.14);
  {
    double y;
    y = x;
  }
  y = (y*3)+4; // geht nicht, y nicht mehr sichtbar.
}
```

Whitespace

- Das Einrücken von Zeilen dient der besseren Lesbarkeit, notwendig ist es (fast) nicht.
- `#include`-Direktiven müssen *immer* einzeln auf einer Zeile stehen.
- Ist das folgende Programm lesbar?

```
1  // whitespace.cc
2  #include <iostream> // includes auf eigener Zeile!
3  #include <iomanip>
4  #include <cmath>
5
6  int main(){
7    double x(0.0);
8    std::cout<<"Gebe␣eine␣lange␣Zahl␣ein:␣";std::cin >> x;
9    std::cout<<"Wurzel(x)=␣"<<std::scientific<<std::showpoint
10             <<std::setprecision(16)<<sqrt(x)<< std::endl;
11
12   return 0;
13 }
```

Arithmetik

Mit Variablen lassen sich sogenannte Operationen durchführen, z.b. auch Arithmetik:

```
1  // operationen.cc
2  #include <iostream>
3
4  int main ()
5  {
6    int a = 6; //Variable a vom Typ int wird definiert.
7    int b = 2;
8    int c = a + b; // Wert von a + b wird c zugewiesen.
9
10   // Natürlich geht auch:
11   c = a*b;   //c wurde oben bereits definiert
12   c = a-b;
13   c = a/b;   // nur möglich, falls a/b ganzzahlig sonst wird gerundet!
14
15   // Auch möglich:
16   c = a/c;   //a wird durch c geteilt und dann c zugewiesen.
17
18   // Oder eben auch komplexere Ausdrücke:
19   c = b+a/b-a;
20   c = (a+b)/(a-b);
21   // es gilt Punkt-vor-Strich, aber Klammern gehen vor
22
```

Arrays

Eine Variable kann auch mehrere Werte speichern, wenn man es als Array definiert:

- Ein Array ist eine Liste von Variablen.
- Wird definiert durch Typ Variable[Größe];
- z.b. int a[10];
- Der Zugriff erfolgt über Variable[index];
- Der erste index ist 0, daher ist der letzte index Größe-1!
- Beispiel: a[0], a[4], a[8], a[9]
- a[10] ist außerhalb des Arrays und produziert einen Fehler!

Beispiel: Fibonacci-Zahlen

```
int main()
{
int x[10]; // Definition eines double Arrays mit 10 Werten

x[0] = 1; // Initialisierung!
x[1] = 1;
x[2] = x[1] + x[0];
x[3] = x[2] + x[1];

// usw....

}
```

Ein- und Ausgabe

Manchmal möchte man bei der Ausführung wissen welche Werte Variablen haben, oder auch Variablen durch Eingabe festlegen.

- C++ benutzt zur Ein- und Ausgabe sogenannte streams
- Findet man im Headerfile " iostream "
- std::cout « Variable « std::endl; zur Ausgabe der Variable
- std::cin » Variable; zur Eingabe in die Variable (auf Datentyp achten!)
- Das std::endl dient dem Zeilenumbruch
- Eigentlich optional aber oft empfehlenswert

Aufgabe 2

1. Öffnet operationen.cc im Verzeichnis examples/progkurs mit einem Editor. (z.b. mit Befehl vi operationen.cc)
2. Gebt nach jeder Zuweisung an c den Wert von c im Format "c = (Wert von c)" aus.
3. Kompiliert operationen.cc so wie oben hallohdnum.cc.
4. Überlegt euch ob die Werte euren Erwartungen entsprechen.

If-Anweisung

- Aus der Mathematik kennt man eine „Zuweisung“ der folgenden Art.
  Für $x \in \mathbb{R}$ setze

$$y = |x| = \begin{cases} x & \text{falls } x \geq 0 \\ -x & \textit{sonst} \end{cases}$$

- Dies realisiert man in C++ mit einer `If`-Anweisung:

```
double x(3.14), y;
if (x>=0)
{
  y = x;
}
else
{
  y = -x;
}
```

Varianten der `If`-Anweisung

- Die geschweiften Klammern kann man weglassen, wenn der Block nur eine Anweisung enthält:

```
double x(3.14), y;
if (x>=0) y = x; else y = -x;
```

- Der `else`-Teil ist optional:

```
double x=3.14;
if (x<0)
  std::cout << "x␣ist␣negativ!" << std::endl;
```

- Weitere Vergleichsoperatoren sind < <= == >= > !=
- Beachte: = für Zuweisung, aber == für den Vergleich zweier Objekte!

While-Schleife

- Bisher: Sequentielle Abfolge von Befehlen wie im Programm angegeben. Das ist langweilig :-)
- Eine Möglichkeit zur Wiederholung bietet die While-Schleife:
  `while` (*Bedingung*)
  { *Schleifenkörper* }
- Beispiel:

```
int i=0; while (i<10) { i=i+1; }
```

- Bedeutung:
  1. Teste Bedingung der While-Schleife
  2. Ist diese *wahr* dann führe Anweisungen im Schleifenkörper aus, sonst gehe zur ersten Anweisung nach dem Schleifenkörper.
  3. Gehe nach 1.
- Anweisungen im Schleifenkörper beeinflussen normalerweise den Wahrheitswert der Bedingung.
- Endlosschleife: Wert der Bedingung wird nie *falsch*.

Pendel (analytische Lösung; `while`-Schleife)

- Die Auslenkung des Pendels mit der Näherung $\sin(\phi) \approx \phi$ und $\phi(0) = \phi_0$, $\phi'(0) = 0$ lautet:

$$\phi(t) = \phi_0 \cos\left(\sqrt{\frac{g}{l}}t\right).$$

- Das folgende Programm gibt diese Lösung zu den Zeiten $t_i = i\Delta t$, $0 \leq t_i \leq T$, $i \in \mathbb{N}_0$ aus:

Pendel (analytische Lösung, while-Schleife)

```
 1 // pendelwhile.cc
 2 #include <iostream> // header für Ein-/Ausgabe
 3 #include <cmath>    // mathematische Funktionen
 4 int main ()
 5 {
 6   double l(1.34);    // Pendellänge in Meter
 7   double phi0(0.2);  // Amplitude im Bogenmaß
 8   double dt(0.05);   // Zeitschritt in Sekunden
 9   double T(30.0);    // Ende in Sekunden
10   double t(0.0);     // Anfangswert
11
12   while ( t<=T )
13   {
14     std::cout << t << "␣"
15               << phi0*cos(sqrt(9.81/l)*t)
16               << std::endl;
17     t = t + dt;
18   }
19
20   return 0;
21 }
```

Wiederholung (for-Schleife)

- Möglichkeit der Wiederholung: `for`-Schleife:
  `for` (*Anfang*; *Bedingung*; *Inkrement*)
  { *Schleifenkörper* }
- Beispiel:

```
for (int i=0; i<=5; i=i+1)
{
  std::cout << "Wert␣von␣i␣ist␣" << i << std::endl;
}
```

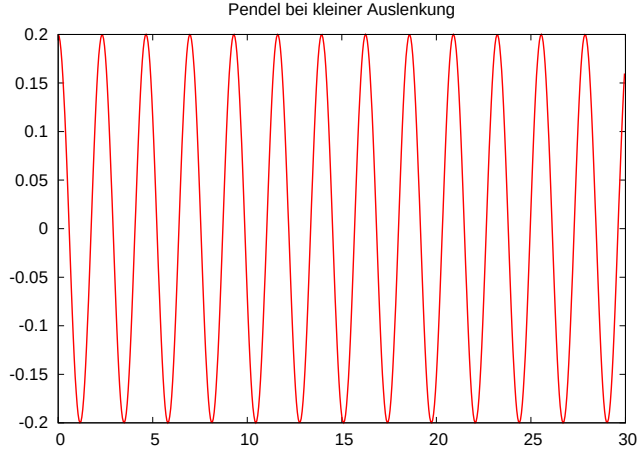
- Enthält der Block nur eine Anweisung dann kann man die geschweiften Klammern weglassen.
- Die *Schleifenvariable* ist so nur innerhalb des Schleifenkörpers sichtbar.
- Die `for`-Schleife kann auch mittels einer *while*-Schleife realisiert werden und umgekehrt.

Pendel (analytische Lösung, for-Schleife)

```
 1 // pendel.cc
 2 #include <iostream> // header für Ein-/Ausgabe
 3 #include <cmath>    // mathematische Funktionen
 4
 5 int main ()
 6 {
 7   double l(1.34);    // Pendellänge in Meter
 8   double phi0(0.2);  // Amplitude im Bogenmaß
 9   double dt(0.05);   // Zeitschritt in Sekunden
10   double T(30.0);    // Ende in Sekunden
11   for (double t=0.0; t<=T; t=t+dt)
12   {
13     std::cout << t << "␣"
14               << phi0*cos(sqrt(9.81/l)*t)
15               << std::endl;
16   }
17
18   return 0;
19 }
```

Visualisierung mit `Gnuplot`

- `Gnuplot` erlaubt einfache Visualisierung von Funktionen $f : \mathbb{R} \to \mathbb{R}$ und $g : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$.
- Für $f : \mathbb{R} \to \mathbb{R}$ genügt eine zeilenweise Ausgabe von Argument und Funktionswert.
- Umlenken der Ausgabe eines Programmes in eine Datei:
  `$ ./pendel > pendel.dat`
- Starte `gnuplot`
  `gnuplot> plot "pendel.dat"with lines`



Aufgabe 3

1. Gebt eine Wertetabelle für f(x) = x^2 im Bereich [-50 ,50] aus.
2. Orientiert euch dabei an pendel.cc für das Format der Ausgabe.
3. Plottet die Funktion mit Gnuplot.

Numerische Lösung des Pendels

- Volles Modell für das Pendel aus der Einführung:

$$\frac{d^2\phi(t)}{dt^2} = -\frac{g}{l}\sin(\phi(t)) \qquad \forall t > 0,$$

$$\phi(0) = \phi_0, \qquad \frac{d\phi}{dt}(0) = u_0.$$

- Umschreiben in System erster Ordnung:

$$\frac{d\phi(t)}{dt} = u(t), \quad \frac{d^2\phi(t)}{dt^2} = \frac{du(t)}{dt} = -\frac{g}{l}\sin(\phi(t)).$$

- Eulerverfahren für $\phi^n = \phi(n\Delta t)$, $u^n = u(n\Delta t)$:

$$\begin{aligned} \phi^{n+1} &= \phi^n + \Delta t\, u^n & \phi^0 &= \phi_0 \\ u^{n+1} &= u^n - \Delta t\,(g/l)\,\sin(\phi^n) & u^0 &= u_0 \end{aligned}$$

Pendel (expliziter Euler)

```
1  // pendelnumerisch.cc
2  #include <iostream> // header für Ein-/Ausgabe
3  #include <cmath>    // mathematische Funktionen
4
5  int main ()
6  {
7    double l(1.34);   // Pendellänge in Meter
8    double phi(3.0);  // Anfangsamplitude in Bogenmaß
9    double u(0.0);    // Anfangsgeschwindigkeit
10   double dt(1E-4);  // Zeitschritt in Sekunden
11   double T(30.0);   // Ende in Sekunden
12   double t(0.0);    // Anfangszeit
13
14   std::cout << t << " " << phi << std::endl;
15   while (t<T)
16   {
17     t = t + dt;          // inkrementiere Zeit
18     double phialt(phi);// merke phi
19     double ualt(u);     // merke u
20     phi = phialt + dt*ualt;              // neues phi
21     u = ualt - dt*(9.81/l)*sin(phialt);// neues u
22     std::cout << t << " " << phi << std::endl;
23   }
24
25   return 0;
26 }
```

Funktionsaufruf und Funktionsdefinition

- In der Mathematik gibt es das Konzept der *Funktion*.
- In C++ auch.
- Sei $f : \mathbb{R} \to \mathbb{R}$, z.B. $f(x) = x^2$.
- Wir unterscheiden den *Funktionsaufruf*

```
double x,y;
y = f(x);
```

- und die *Funktionsdefinition*. Diese sieht so aus:

  Ergebnistyp Funktionsname (*Argumente*)
  { *Funktionsrumpf* }

- Beispiel:

```
double f (double x)
{
  return x*x;
}
```

Komplettbeispiel zur Funktion

```
 1 // funktion.cc
 2 #include <iostream>
 3
 4 double f (double x)
 5 {
 6   return x*x;
 7 }
 8
 9 int main ()
10 {
11   double x(2.0);
12   std::cout << "f(" << x << ")=" << f(x) << std::endl;
13
14   return 0;
15 }
```

- Funktionsdefinition muss vor Funktionsaufruf stehen.
- Formales Argument in der Funktionsdefinition entspricht einer Variablendefinition.
- Beim Funktionsaufruf wird das Argument (hier) *kopiert*.
- `main` ist auch nur eine Funktion.

Weiteres zum Verständnis der Funktion

- Der Name des formalen Arguments in der Funktionsdefinition ändert nichts an der Semantik der Funktion (Sofern es überall geändert wird):

```
double f (double y)
{
  return y*y;
}
```

- Das Argument wird hier kopiert, d.h.:

```
double f (double y)
{
  y = 3*y*y;
  return y;
}

int main ()
{
  double x(3.0),y;
  y = f(x); // ändert nichts an x !
}
```

Weiteres zum Verständnis der Funktion

- Argumentliste kann leer sein (wie in der Funktion `main`):

```
double pi ()
{
  return 3.14;
}

y = pi(); // Klammern sind erforderlich!
```

- Der Rückgabetyp `void` bedeutet „keine Rückgabe“

```
void hello ()
{
  std::cout << "hello" << std::endl;
}

hello();
```

- Mehrere Argument werden durch Kommata getrennt:

```
double g (int i, double x)
{
  return i*x;
}
std::cout << g(2,3.14) << std::endl;
```

Pendelsimulation als Funktion

```
1  // pendelmitfunktion.cc
2  #include <iostream> // header für Ein-/Ausgabe
3  #include <cmath>    // mathematische Funktionen
4
5  void simuliere_pendel (double l, double phi, double u)
6  {
7    double dt   = 1E-4;
8    double T    = 30.0;
9    double t    = 0.0;
10
11   std::cout << t << "␣" << phi << std::endl;
12   while (t<T)
13   {
14     t = t + dt;
15     double phialt(phi),ualt(u);
16     phi = phialt + dt*ualt;
17     u = ualt - dt*(9.81/l)*sin(phialt);
18     std::cout << t << "␣" << phi << std::endl;
19   }
20 }
21
22 int main ()
23 {
24   simuliere_pendel(1.34,3.0,0.0);
25
26   return 0;
```

Aufgabe 4

1. Schreibt eine Funktion, die die Fakultät einer (natürlichen) Zahl berechnet und ausgibt.
2. Plottet die Funktion mit Gnuplot im Bereich [1,10].

Funktionsschablonen

- Oft macht eine Funktion mit Argumenten verschiedenen Typs einen Sinn.
- `double f (double x) {return x*x;}` macht auch mit `float`, `int` oder `mpf_class` Sinn.
- Man könnte die Funktion für jeden Typ definieren. Das ist natürlich sehr umständlich. (Es darf mehrere Funktionen gleichen Namens geben, sog. *overloading*).
- In C++ gibt es mit Funktionsschablonen (engl.: *function templates*) eine Möglichkeit den Typ variabel zu lassen:

```
template<typename T>
T f (T y)
{
  return y*y;
}
```

- `T` steht hier für einen beliebigen Typ.

Pendelsimulation mit Templates I

```
1  // pendelmitfunktionstemplate.cc
2  #include <iostream> // header für Ein-/Ausgabe
3  #include <cmath>    // mathematische Funktionen
4
5  template<typename Number>
6  void simuliere_pendel (Number l, Number phi, Number u)
7  {
8    Number dt(1E-4);
9    Number T(30.0);
10   Number t(0.0);
11   Number g(9.81/l);
12
13   std::cout << t << "␣" << phi << std::endl;
14   while (t<T)
15   {
16     t = t + dt;
17     Number phialt(phi),ualt(u);
18     phi = phialt + dt*ualt;
19     u = ualt - dt*g*sin(phialt);
20     std::cout << t << "␣" << phi << std::endl;
21   }
22 }
23
```

Pendelsimulation mit Templates II

```
24 int main ()
25 {
26   float l1(1.34);   // Pendellänge in Meter
27   float phi1(3.0); // Anfangsamplitude in Bogenmaß
28   float u1(0.0);    // Anfangsgeschwindigkeit
29   simuliere_pendel(l1,phi1,u1);
30
31   double l2(1.34);   // Pendellänge in Meter
32   double phi2(3.0); // Anfangsamplitude in Bogenmaß
33   double u2(0.0);    // Anfangsgeschwindigkeit
34   simuliere_pendel(l2,phi2,u2);
35
36   return 0;
37 }
```

Headerdateien

- Das tolle an Funktionen ist, dass man sie wiederverwenden kann.
- Es gibt ganze Sammlungen an Funktionen, sogenannte Funktionsbibliotheken
- Werden in separaten Headerdateien gespeichert. (Dateiendung .h,.hh, o.ä.)
- Um Funktionen (und anderes) aus einer Headerdatei zu verwenden:
- #include "Dateiname"
- Es gibt eine Standardbibliothek von Headerdateien (z.b. cmath,string,iostream)

Headerdateien

```
#include <iostream> // Standardbibliothek (Name in <>)
#include <cmath>
#include "hdnum.hh" // Eigene Headerdatei (Dateiname in " ")

int main()
{
    float zahl = 5.0;
    zahl = sqrt(zahl); //sqrt aus cmath
    std::cout << zahl << std::endl;
    // std::cout, std::endl aus iostream
    hdnum::vector(2,1.0); // Aus hdnum.hh
}
```

Headerdateien

- Nur #include in die Datei zu schreiben reicht nicht!
- Beim kompilieren muss noch der Ort der Headerdatei angegeben werden
- bei dem kompilierbefehl -I<Verzeichnis> einfügen
- z.b. g++ -o hallohdnum -I../../ hallohdnum.cc
- ../../ bedeutet Oberverzeichnis des Oberverzeichnisses (2x ../)

Aufgabe 5

1. Erstellt eine Headerdatei für eure Fakultätsfunktion aus Aufgabe 3. (z.b. fakultaet.hh)
2. Schreibt eure Funktionsdefinition in die Headerdatei, und kommentiert sie in der Hauptdatei aus.
3. Bindet eure headerdatei mit #include ein und führt eure Funktion in der Datei aus.

Klassen

- Sehr grobe Darstellung von Klassen für die Praktische Verwendung.
- Klassen erlauben einem C++ Programmierer eigene Datentypen zu definieren.
- Es gibt unter anderem Klassenvariablen und Klassenmethoden.
- Klassenvariablen sind die Variablen aus denen der neue Datentyp zusammengesetzt ist.
- Können elementare Datentypen wie int sein oder aber auch andere Klassen.
- Klassenmethoden sind Funktionen mit denen Klassenvariablen manipuliert werden können.

Klassen I

```
1  // classes.cc
2  #include <iostream> // header für Ein-/Ausgabe
3  #include <cmath>    // mathematische Funktionen
4
5  class Vector3d // einfaches Beispiel für eine Klasse
6  {
7  public:
8    float x; //Klassenvariablen
9    float y;
10   float z;
11
12   Vector3d(float par1, float par2, float par3)
13   {
14     x = par1;
15     y = par2;
16     z = par3;
17   }
18
19   float getNorm() //Klassenmethode (Norm des Vektors)
20   {
21     return sqrt(x*x + y*y + z*z);
22   }
23
```

Klassen II

```
24    float dot(Vector3d U) // Skalarprodukt mit anderem Vektor
25    {
26      return x*U.x + y*U.y + z*U.z;
27    }
28 };
29
30
31 int main ()
32 {
33    Vector3d V(1.0,2.0,3.0); // Erstellen einer Variable der Klasse Vector3
34    Vector3d W(-1.0,-1.0,1.0); // Erstellen einer anderen Variable der
   Klasse Vector3d
35    std::cout << V.x << std::endl; // Zugriff auf Variable x von V
36    std::cout << W.x << std::endl; // Zugriff auf Variable x von W
37    std::cout << V.getNorm() << std::endl; // Norm von V
38    std::cout << W.getNorm() << std::endl; // Norm von W
39    std::cout << V.dot(W) << std::endl;     // Skalarprodukt zwischen
   V und W
40    std::cout << W.dot(V) << std::endl;     // Skalarprodukt zwischen
   W und V (das selbe)
41
42    return 0;
43 }
```

Klassen III

- Klassenvariablen können grundsätzlich nur durch Klassenmethoden modifiziert oder gelesen werden!
- Ausnahme: Variablen die als "public"deklariert wurden.
- Verschiedene Variablen einer Klasse (wie eben V und W) teilen sich ihre Untervariablen nicht.
- Zugriff auf Klassenelemente erfolgt über den . Operator: Also wie eben V.getNorm(), V.x, W.x

Klassenbibliotheken

- Klassen muss man (zum Glück) oft nicht selbst programmieren.
- Es gibt bereits fertige Klassenbibliotheken.
- Im folgenden beschäftigen wir uns mit der HDNum-Bibiliothek für den Numerik-Kurs

HDNUM

- C++ kennt keine Matrizen, Vektoren, Polynome, ...
- Wir haben C++ erweitert um die **Heidelberg Educational Numerics Library**, kurz **HDNum**.
- Alle in der Vorlesung behandelten Beispiele sind dort enthalten.
- Dieser Programmierkurs ist auch Teil von HDNUM

HDNUM

- HDNum realisiert Vektoren, Matrizen usw. als Klassen-Templates
- Klassen-Templates analog zu Funktions-Templates
- d.h. Vektoren, Matrizen mit Elementen verschiedener Datentypen
- Klassenmethoden für Lineare Algebra, z.b. Matrizenmultiplikation, Skalarprodukte
- Einbinden über #include "hdnum.hh"
- + Angabe des Verzeichnisses der Datei hdnum.hh über -I<Verzeichnis> beim kompilieren
- Detaillierte Anleitung im repository unter hdnum/tutorial

HDNUM Vektoren

```
 1 // vektoren.cc
 2 #include <iostream>     // notwendig zur Ausgabe
 3 #include "hdnum.hh"     // hdnum header
 4
 5 int main ()
 6 {
 7   hdnum::Vector<float> x(10);          // Vektor mit 10 float-Elementen
 8   hdnum::Vector<float> y(10,3.14);   // 10 Elemente initialisiert
 9   hdnum::Vector<float> a;              // uninitialisierter float-Vektor
10
11   x[5] = 5.0; // Zugriff erfolgt wie bei Arrays
12   x = 5.0; // Alle Werte sind 5.0 (geht bei Arrays nicht!)
13
14   a = x + y; // Wäre bei Arrays auch nicht so einfach
15   float d = x*y; //Skalarprodukt!
16
17   return 0;
18 }
```

HDNUM Matrizen

```
1  // matrizen.cc
2  #include <iostream>     // notwendig zur Ausgabe
3  #include "hdnum.hh"     // hdnum header
4
5  int main ()
6  {
7    // Konstruktion
8    hdnum::DenseMatrix<float> A(10,10);       // 10x10 Matrix uninitialisi
9    hdnum::DenseMatrix<float> B(10,10,1.0); // 10x10 Matrix uninitialisi
10   hdnum::DenseMatrix<float> C(10,10,2.0); // 10x10 Matrix initialisier
11
12   A[1][3] = 3.14; // Zuweisung auf A13
13   A = B + C;      // Addition
14   C.mm(A,B);      // C = A*B (matmul)
15
16   hdnum::Vector<float> x(10,1.0);
17   hdnum::Vector<float> y(10,1.0);
18   A.mv(y,x);     //y = A*x (matrix-vector)
19
20   return 0;
21 }
```

HDNUM Ausgabe

```
1  // ausgabe.cc
2  #include <iostream>     // notwendig zur Ausgabe
3  #include "hdnum.hh"     // hdnum header
4
5  int main ()
6  {
7    hdnum::Vector<float> x(10,1.0);
8    hdnum::DenseMatrix<float> M(10,10,1.0);
9
10   std::cout << x << std::endl; // Ausgabe von 7 Nachkommastellen (defaul
11   std::cout << M << std::endl;
12
13   x.iwidth(2);                 // Stellen in Indexausgabe
14   x.width(20);                 // Anzahl Stellen gesamt
15   x.precision(16);             // Anzahl Nachkommastellen
16
17   M.iwidth(2);                 // Stellen in Indexausgabe
18   M.width(20);                 // Anzahl Stellen gesamt
19   M.precision(16);             // Anzahl Nachkommastellen
20
21   std::cout << x << std::endl; // Ausgabe von mehr Nachkommastellen
22   std::cout << M << std::endl;
23
24   return 0;
25 }
```

Beispiel: Gram-Schmidt-Orthogonalisierung mit HDNUM

- Gegeben: n (linear unabhängige) Vektoren
- Gesucht: n orthogonale Vektoren, die den selben Unterraum aufspannen
- Projektionen der anderen Vektoren werden raussubtrahiert
- Entsprechend der Formel:

$$w_j = v_j - \sum_{i=1}^{j-1} \frac{\langle w_i, v_j \rangle}{\langle w_i, w_i \rangle} w_i$$

Gram-Schmidt Orthogonalisierung I

```
 1 // gaussseidel.cc
 2 #include <iostream>     // notwendig zur Ausgabe
 3 #include <vector>
 4 #include "hdnum.hh"     // hdnum header
 5
 6 int main ()
 7 {
 8   /// Anfangsvektoren (1,2,3) (4,5,6) (7,8,9)
 9   hdnum::Vector<float> v1(3);
10   fill(v1, (float) 1.0,(float) 1.0);
11   hdnum::Vector<float> v2(3);
12   fill(v2, (float) 4.0,(float) 1.0);
13   hdnum::Vector<float> v3(3);
14   fill(v3, (float) 7.0,(float) 1.0);
15   hdnum::Vector<float> w1 = v1;
16   hdnum::Vector<float> w2;
17   hdnum::Vector<float> w3;
18
19   hdnum::Vector<float> proj = w1;
20   proj *= (w1*v2)/(w1*w1);
21
22   w2 = v2 - proj;
23
```

Gram-Schmidt Orthogonalisierung II

```
24    proj = w1;
25    proj *= w1*v3/(w1*w1);
26
27    w3 = v3 - proj;
28
29    proj = w2;
30    proj *= w2*v3/(w2*w2);
31
32    w3 -= proj;
33
34    v1.precision(2);
35    std::cout << v1 << std::endl;
36    std::cout << v2 << std::endl;
37    std::cout << v3 << std::endl;
38
39    std::cout << w1 << std::endl;
40    std::cout << w2 << std::endl;
41    std::cout << w3 << std::endl;
42
43    std::cout << w1*w2 << std::endl;
44    std::cout << w2*w3 << std::endl;
45    std::cout << w1*w3 << std::endl;
46
```

Gram-Schmidt Orthogonalisierung III

```
47    return 0;
48 }
```

Debugging

- Was mache ich wenn mein Programm nicht läuft?
- Grundsätzliches:
  1. Fehlermeldung lesen!
  2. Fehlermeldung bei google eingeben (Copy-Paste) falls unklar.
  3. Falls ihr gar nicht weiter kommt: Kommilitonen/Tutoren um Hilfe bitten.

Debugging II

- Was mache ich wenn mein Programm läuft, aber nicht das macht was ich erwarte?
  1. Geht das Programm nochmal Zeile für Zeile durch.
  2. Schaut euch den Wert eurer Variablen mit std::cout an.
  3. Insbesondere Variablen in If-Statements oder Schleifen.
  4. Falls möglich, Ergebnis plotten. (grundsätzlich sinnvoll)
  5. Falls ihr gar nicht weiter kommt: Kommilitonen/Tutoren um Hilfe bitten.

Debugging III

- Typische Fehler
  1. Semikolon vergessen
  2. Klammer vergessen
  3. Groß- und Kleinschreibung
  4. Sonstige Tippfehler
  5. Variable nicht deklariert/initialisiert
  6. Falscher Datentyp in Funktion eingesetzt
  7. und vieles mehr...