Heidelberger Numerikbibliothek für die Lehre

Peter Bastian
Universität Heidelberg
Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Im Neuenheimer Feld 368, D-69120 Heidelberg
email: Peter.Bastian@iwr.uni-heidelberg.de

7. Mai 2010

Die Heidelberger Numerikbibliothek wurde begleitend zu den Vorlesungen *Einführung in die Numerik* und *Numerik* in der Programmiersprache C++ entwickelt und stellt einfach zu benutzende Klassen für grundlegende Aufgaben in der Numerik bis hin zur Lösung von gewöhnlichen Differentialgleichungen zur Verfügung. In fast allen Klassen ist der benutzte Zahlentyp parametrisierbar so dass auch hochpräzise Rechnungen durchgeführt werden können.

Inhaltsverzeichnis

1 Einführung

Was ist HDNUM

- HDNUM ist eine kleine Sammlung von C++ Klassen, die die Implementierung numerischer Algorithmen aus der Vorlesung erleichtern soll.
- Die aktuelle Version gibt es unter

  `http://conan.iwr.uni-heidelberg.de/teaching/numerik1_ss2010/`

- Einige Ziele bei der Entwicklung von HDNUM waren:
  - Einfache Installation: Es mur nur eine Header-Datei eingebunden werden.
  - Einfache Benutzung der Klassen: Z.B. keine dynamische Speicherverwaltung.
  - Möglichkeit der Rechnung mit verschiedenen Zahl-Datentypen.
  - Effiziente Realisierung der Verfahren möglich: Z.B. Block-Algorithmen in der linearen Algebra.

Installation

- Datei `hdnum-x.yy.tgz` (komprimiertes tar archive) herunterladen.
- Archiv mit `tar zxf hdnum-x.yy.tgz` entpacken.
- Das Verzeichnis enthält unter anderem:
  - Das Verzeichnis `src` mit dem Quellcode der Klassen (muss Sie nicht interessieren).
  - Das Verzeichnis `examples` mit den Beispielanwendungen (die sollten Sie sich ansehen).
  - Das Verzeichnis `tutorial`: Quelle für dieses Dokument.
  - Die Datei `hdnum.hh`, die zentrale Header-Datei, die in alle Anwendungen eingebunden werden muss.
- Das Verzeichnis `hdnum/examples` enthält ein simples Makefile zum Übersetzen der Programme.
- Die Beispiele erfordern die Installation der GNU multiprecision library `http://gmplib.org/`. Ist diese nicht vorhanden müssen Makefiles entsprechend angepasst werden.

Typisches HDNUM Programm

```
1  // hallohdnum.cc
2  #include <iostream>     // notwendig zur Ausgabe
3  #include "hdnum.hh"     // hdnum header
4  using namespace hdnum; // Namen ohne hdnum:: verwenden
5
6  int main ()
7  {
8    Array<float> a(10,3.14); // Feld mit 10 init. Elementen
9    a[3] = 1.0;              // Zugriff auf Element 3
10 }
```

- Übersetzen im Verzeichnis `examples` mit GMP installiert:

  ```
  g++ -I.. -o hallohdnum hallohdnum.cc -lm -lgmpxx -lgmp
  ```

- und ohne GMP:

  ```
  g++ -I.. -o hallohdnum hallohdnum.cc -lm
  ```

- oder einfach

  ```
  make
  ```

- oder falls kein GMP installiert ist

  ```
  make nogmp
  ```

2 Ein kleiner Programmierkurs

2.1 Hallo Welt

Programmierumgebung

- Wir benutzen die Programmiersprache C++.
- Wir behandeln nur die Programmierung unter LINUX mit den GNU compilern.
- Windows: On your own.
- Wir setzen Grundfertigkeit im Umgang mit LINUX-Rechnern voraus:
  - Shell, Kommandozeile, Starten von Programmen.
  - Dateien, Navigieren im Dateisystem.
  - Erstellen von Textdatein mit einem Editor ihrer Wahl.
- Idee des Kurses: „Lernen an Beispielen“, keine rigorose Darstellung.
- Blutige Anfänger sollten zusätzlich ein Buch lesen (siehe Literaturliste).

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. Interpretiere Ergebnisse. Dazu benutzen wir weitere Programme wie **gnuplot** oder **grep**.
5. Falls Ergebnis nicht korrekt, gehe nach 1!

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.

Herunterladen von HDNUM

1. Einloggen
2. Erzeuge neues Verzeichnis mit `$ mkdir kurs`
3. Wechsle in das Verzeichnis mit `$ cd kurs`
4. Gehe zur Webseite `http://conan.iwr.uni-heidelberg.de/teaching/numerik0_ws2009/`
5. Klicke auf **Version 0.10, Stand 12.10.2009** und bestätige
6. Kopiere Datei `hdnum-0.10.tgz` in das Verzeichnis: `$ cp ~/Desktop/hdnum-0.10.tgz .`
7. Entpacken der Datei mit `$ tar zxvf hdnum-0.10.tgz`
8. Wechsle in das Verzeichnis `$ cd hdnum/examples`
9. Anzeigen der Dateien mittels `$ ls`

Wichtige UNIX-Befehle

- `ls --color -F` - Zeige Inhalt des aktuellen Verzeichnisses
- `cd` - Wechsle ins Home-Verzeichnis
- `cd <verzeichnis>` - Wechsle in das angegebene Verzeichnis (im aktuellen Verzeichnis)
- `cd ..` - Gehe aus aktuellem Verzeichnis heraus
- `mkdir < verzeichnis>` - Erstelle neues Verzeichnis
- `cp <datei1> <datei2>` - Kopiere datei1 auf datei2 (datei2 kann durch Verzeichnis ersetzt werden)
- `mv <datei1> <datei2>` - Benenne datei1 in datei2 um (datei2 kann durch Verzeichnis ersetzt werden, dann wird datei1 dorthin verschoben)
- `rm <datei>` - Lösche datei
- `rm -rf <verzeichnis>` - Lösche Verzeichnis mit allem darin

Hallo Welt !

Öffne die Datei `hallowelt.cc` mit einem Editor: `$ gedit hallowelt.cc`

```
1 // hallowelt.cc (Dateiname als Kommentar)
2 #include <iostream>  // notwendig zur Ausgabe
3
4 int main ()
5 {
6   std::cout << "Numerik␣0␣ist␣leicht:" << std::endl;
7   std::cout << "1+1=" << 1+1 << std::endl;
8 }
```

- `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:

```
$ g++ -o hallowelt hallowelt.cc
$ ./hallowelt
```

- Dies sollte dann die folgende Ausgabe liefern:

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

2.2 Variablen und Typen

(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  int main ()
4  {
5    unsigned int i; // uninitialisierte natürliche Zahl
6    double x(3.14); // initialisierte Fließkommazahl
7    float y(1.0);   // einfache Genauigkeit
8    short j(3);     // eine "kleine" Zahl
9    std::cout << "(i+x)*(y+j)=" << (i+x)*(y+j) << std::endl;
10 }
```

Andere Typen

- C++ kennt noch viele weitere Typen.
- Typen können nicht nur Zahlen sondern viele andere Informationen repräsentieren.
- Etwa Zeichenketten: `std::string`
- Oft muss man dazu weitere Headerdateien angeben.

```
1  // string.cc
2  #include <iostream>
3  #include <string>
4  int main ()
5  {
6    std::string m1("Zeichen");
7    std::string leer("␣␣␣");
8    std::string m2("kette");
9    std::cout << m1+leer+m2 << std::endl;
10 }
```

- Jede Variable *muss* einen Typ haben. Strenge Typbindung.

Mehr Zahlen

```
1  // mehrzahlen.cc
2  #include <iostream> // header für Ein-/Ausgabe
3  #include <gmpxx.h>  // header für GNU multiprecision
4  #include <complex>  // header für komplexe Zahlen
5  int main ()
6  {
7    mpf_class x("3.14159265358979323846264338327950288 41",512);
8    std::complex<double> y(1.0,3.0);
9    std::complex<mpf_class> z(x,x);
10   std::cout << x << "␣" << y << "␣" << z << std::endl;
11 }
```

- GNU Multiprecision Library `http://gmplib.org/` erlaubt Zahlen mit vielen Stellen (hier 512 Stellen zur Basis 2).
- Übersetzen mit: `$ g++ -o mehrzahlen mehrzahlen.cc -lgmpxx -lgmp`
- Komplexe Zahlen sind Paare von Zahlen.
- `complex<>` ist ein Template: Baue komplexe Zahlen aus jedem anderen Zahlentyp auf (später mehr!).

Mehr Ein- und Ausgabe

```
1  // eingabe.cc
2  #include <iostream> // header für Ein-/Ausgabe
3  #include <iomanip>  // für setprecision etc.
4  #include <gmpxx.h>  // header für GNU multiprecision
5  int main ()
6  {
7    mpf_class x("0.0",512);
8    std::cout << "Gebe␣eine␣lange␣Zahl␣ein:␣";
9    std::cin >> x;
10   std::cout << "Wurzel(x)=␣"
11             << std::scientific << std::showpoint
12             << std::setprecision(15)
13             << sqrt(x) << std::endl;
14 }
```

- Eingabe geht mit `std::cin >> x;`

- Standardmäßig werden nur 6 Nachkommastellen ausgegeben. Das ändert man mit `std::setprecision`.
- Dazu muss man die Headerdatei `iomanip` einbinden.
- Die Wurzel berechnet die Funktion `sqrt`.

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 "math.h"
5 int main(){double x(0.0);
6 std::cout<<"Gebe␣eine␣lange␣Zahl␣ein:␣";std::cin >> x;
7 std::cout<<"Wurzel(x)=␣"<<std::scientific<<std::showpoint
8 <<std::setprecision(16)<<sqrt(x)<< std::endl;}
```

2.3 Entscheidung

`If`-Anweisung

- Aus der Mathematik kennt man eine „Zuweisung“ der folgenden Art.

  Für $x \in \mathbb{R}$ setze

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

- 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!

2.4 Wiederholung

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

```
1  // pendelwhile.cc
2  #include <iostream> // header für Ein-/Ausgabe
3  #include <math.h>   // 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  }
```

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.

Pendel (analytische Lösung, `for`-Schleife)

```
1  // pendel.cc
2  #include <iostream> // header für Ein-/Ausgabe
3  #include <math.h>   // 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   for (double t=0.0; t<=T; t=t+dt)
11     std::cout << t << "␣"
12               << phi0*cos(sqrt(9.81/l)*t)
13               << std::endl;
14 }
```

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.

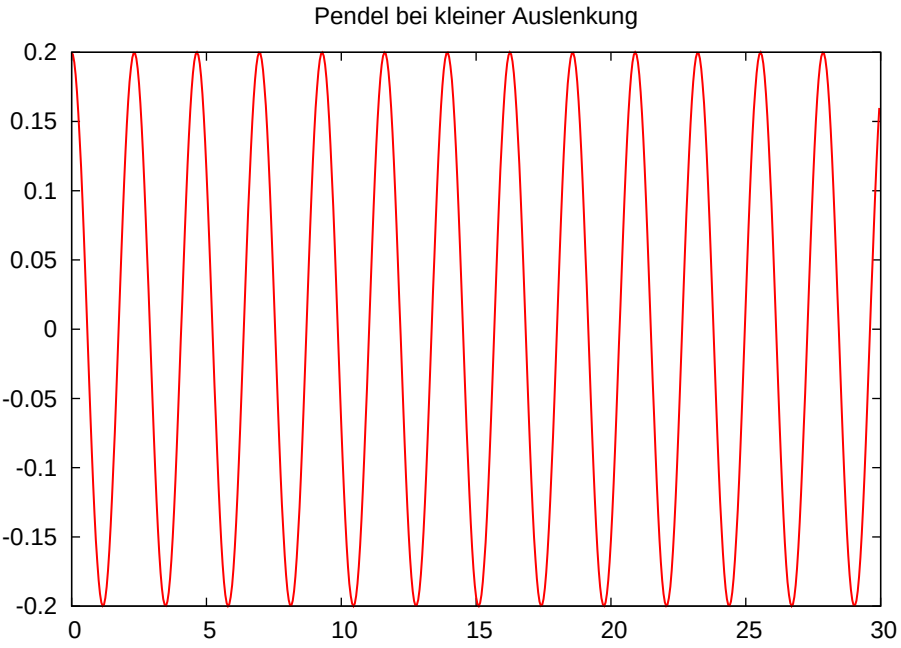

Abbildung 1: Das Pendel in Aktion. Gnuplot-Ausgabe des Programmes `pendel.cc`.

- Umlenken der Ausgabe eines Programmes in eine Datei: `$ ./pendel > pendel.dat$`
- Starte `gnuplot` `gnuplot> plot "pendel.dat"with lines`

Geschachtelte Schleifen

- Ein Schleifenkörper kann selbst wieder eine Schleife enthalten, man spricht von *geschachtelten* Schleifen.
- Beispiel:

```
for (int i=1; i<=10; i=i+1)
  for (int j=1; j<=10; j=j+1)
    if (i==j)
      std::cout << "i␣gleich␣j:␣" << std::endl;
    else
      std::cout << "i␣ungleich␣j!" << std::endl;
```

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:

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

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

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

Pendel (expliziter Euler)

```
1  // pendelnumerisch.cc
2  #include <iostream> // header für Ein-/Ausgabe
3  #include <math.h>   // 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 }
```

2.5 Funktionen

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 }
```

- 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 <math.h>   // mathematische Funktionen
4
5  void simuliere_pendel (double l, double phi, double u) {
6    double dt   = 1E-4;
7    double T    = 30.0;
8    double t    = 0.0;
9
10   std::cout << t << "␣" << phi << std::endl;
11   while (t<T) {
12       t = t + dt;
13       double phialt(phi),ualt(u);
14       phi = phialt + dt*ualt;
15       u = ualt - dt*(9.81/l)*sin(phialt);
16       std::cout << t << "␣" << phi << std::endl;
17     }
```

```
18 }
19
20 int main () {
21   double l(1.34); // Pendellänge in Meter
22   double phi(3.0);// Anfangsamplitude in Bogenmaß
23   double u(0.0);  // Anfangsgeschwindigkeit
24   simuliere_pendel(l,phi,u); }
```

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

```
1  // pendelmitfunktionstemplate.cc
2  #include <iostream> // header für Ein-/Ausgabe
3  #include <math.h>   // mathematische Funktionen
4
5  template<typename Number>
6  void simuliere_pendel (Number l, Number phi, Number u) {
7    Number dt(1E-4);
8    Number T(30.0);
9    Number t(0.0);
10   Number g(9.81/l);
11
12   std::cout << t << "␣" << phi << std::endl;
13   while (t<T) {
14       t = t + dt;
15       Number phialt(phi),ualt(u);
16       phi = phialt + dt*ualt;
17       u = ualt - dt*g*sin(phialt);
18       std::cout << t << "␣" << phi << std::endl;
19     }
20 }
21
22 int main () // geht leider nicht mit GNU MP :-(
23 {
24   float l1(1.34);  // Pendellänge in Meter
25   float phi1(3.0); // Anfangsamplitude in Bogenmaß
```

```
26   float u1(0.0);    // Anfangsgeschwindigkeit
27   simuliere_pendel(l1,phi1,u1);
28
29   double l2(1.34);  // Pendellänge in Meter
30   double phi2(3.0); // Anfangsamplitude in Bogenmaß
31   double u2(0.0);   // Anfangsgeschwindigkeit
32   simuliere_pendel(l2,phi2,u2);
33 }
```

Referenzargumente

- Das Kopieren der Argumente einer Funktion kann verhindert werden indem man das Argument als *Referenz* definiert:

```
void f (double x, double& y)
{
  y = x*x;
}

double x(3), y;
f(x,y); // y hat nun den Wert 9, x ist unverändert.
```

- Statt eines Rückgabewertes kann man auch ein (zusätzliches) Argument modifizieren.
- Insbesondere kann man so den Fall mehrerer Rückgabewerte realisieren.
- Referenzargumente bieten sich auch an wenn Argumente „sehr groß“ sind und damit das kopieren sehr zeitaufwendig ist.
- Der aktuelle Parameter im Aufruf *muss* dann eine Variable sein.

3 Vektoren und Matrizen

3.1 Vektoren

`Vector<T>`

- `Vector<T>` ist ein Klassen-Template.
- Es macht aus einem beliebigen (Zahl-)Datentypen `T` einen Vektor.
- Auch komplexe und hochgenaue Zahlen sind möglich.
- Vektoren verhalten sich so wie man es aus der Mathematik kennt:
  - Bestehen aus n Komponenten.
  - Diese sind von 0 bis $n-1$ (!) durchnummeriert.
  - Addition und Multiplikation mit Skalar.

  - Skalarprodukt und Norm (noch nicht implementiert).
  - Matrix-Vektor-Multiplikation

- Die folgenden Beispiele findet man in `vektoren.cc`

Konstruktion und Zugriff

- Konstruktion mit und ohne Initialisierung

```
Vector<float> x(10);          // Vektor mit 10 Elementen
Vector<double> y(10,3.14);    // 10 Elemente initialisiert
Vector<float> a;              // ein leerer Vektor
```

- Speziellere Vektoren

```
Vector<std::complex<double> >
  cx(7,std::complex<double>(1.0,3.0));
mpf_set_default_prec(1024); // Setze Genauigkeit für mpf_class
Vector<mpf_class> mx(7,mpf_class("4.44"));
```

- Zugriff auf Element

```
for (std::size_t i=0; i<x.size(); i=i+1)
  x[i] = i;                   // Zugriff auf Elemente
```

- Vektorobjekt wird am Ende des umgebenden Blockes gelöscht.

Kopie und Zuweisung

- Copy-Konstruktor hat **Referenzsemantik**!

```
Vector<float> z(x); // Kopie ist eine Referenz auf gleiche Daten
z[2] = 1.24;        // hat den gleichen Effekt wie x[2] = 1.24 !
```

- Erstellen einer echten Kopie

```
Vector<float> b(copy(x)); // b ist echte Kopie von a,x,z
```

- Zuweisung funktioniert ganz normal, **aber** beide Vektoren müssen die gleiche Größe haben!

```
b = z;                // b kopiert die Daten aus z
a = 5.4;              // Zuweisung an alle Elemente
Vector<double> w;     // leerer Vektor
w.resize(x.size()); // make correct size
w = x;                // copy elements
```

- Ausschnitte von Vektoren

```
Vector<float> w(x.sub(7,3));// w referenziert x[7],...,x[9]
z = x.sub(3,4);             // z referenziert x[3],...,x[6]
```

Rechnen und Ausgabe

- Vektorraumoperationen und Skalarprodukt

```
w += z;              // w = w+z
w -= z;              // w = w-z
w *= 1.23;           // skalare Multiplikation
w /= 1.23;           // skalare Division
w.update(1.23,z);    // w = w + a*z
float s;
s = w*z;             // Skalarprodukt
```

- Ausgabe auf die Konsole

```
std::cout << w << std::endl;// schöne Ausgabe
w.iwidth(2);                // Stellen in Indexausgabe
w.width(20);                // Anzahl Stellen gesamt
w.precision(16);            // Anzahl Nachkommastellen
std::cout << w << std::endl;// nun mit mehr Stellen
std::cout <<cx << std::endl;// geht auch für complex
std::cout <<mx << std::endl;// geht auch für mpf_class
```

Beispielausgabe

```
[   0]     1.204200e+01
[   1]     1.204200e+01
[   2]     1.204200e+01
[   3]     1.204200e+01

[ 0] 1.20420007705688 48e+01
[ 1] 1.20420007705688 48e+01
[ 2] 1.20420007705688 48e+01
[ 3] 1.20420007705688 48e+01
```

Hilfsfunktionen

```
zero(w);                    // das selbe wie w=0.0
fill(w,(float)1.0);         // das selbe wie w=1.0
fill(w,(float)0.0,(float)0.1); // w[0]=0, w[1]=0.1, w[2]=0.2, ...
unitvector(w,2);            // kartesischer Einheitsvektor
gnuplot("test.dat",w);      // gnuplot Ausgabe: i w[i]
gnuplot("test2.dat",w,z);   // gnuplot Ausgabe: w[i] z[i]
```

Funktionen

- Beispiel: Summe aller Komponenten

```
double sum (Vector<double> x) {
  double s(0.0);
  for (std::size_t i=0; i<x.size(); i=i+1)
    s = s + x[i];
  return s;
}
```

- Mit **Funktionentemplate**:

```
template<class T>
T sum (Vector<T> x) {
  T s(0.0);
  for (std::size_t i=0; i<x.size(); i=i+1)
    s = s + x[i];
  return s;
}
```

- **Vorsicht**: Call-by-value erzeugt **keine** Kopie!

3.2 Matrizen

`Matrix<T>`

- `Matrix<T>` ist ein Klassen-Template.
- Es macht aus einem beliebigen (Zahl-)Datentypen `T` eine Matrix.
- Auch komplexe und hochgenaue Zahlen sind möglich.
- Matrizen verhalten sich so wie man es aus der Mathematik kennt:
  - Bestehen aus $m \times n$ Komponenten.
  - Diese sind von 0 bis $m-1$ bzw. $n-1$ (!) durchnummeriert.
  - $m \times n$-Matrizen bilden einen Vektorraum.
  - Matrix-Vektor und Matrizenmultiplikation.
- Die folgenden Beispiele findet man in `matrizen.cc`

Konstruktion und Zugriff

- Konstruktion mit und ohne Initialisierung

```
Matrix<float> B(10,10);      // 10x10 Matrix uninitialisiert
Matrix<float> C(10,10,0.0); // 10x10 Matrix initialisiert
```

- Zugriff auf Elemente

```
for (int i=0; i<B.rowsize(); ++i)
  for (int j=0; j<B.colsize(); ++j)
    B[i][j] = 0.0;           // jetzt ist B initialisiert
```

- Matrixobjekt wird am Ende des umgebenden Blockes gelöscht.

Kopie und Zuweisung

- Copy-Konstruktor hat **Referenzsemantik**!

```
Matrix<float> D(B); // D identisch mit B! Keine Kopie
D[0][0] = 3.14;     // ändert auch B[0][0]
```

- Erstellen einer echten Kopie

```
Matrix<float> E(copy(B)); // E ist echte Kopie
```

- Zuweisung ganz normal, **aber** beide Matrizen müssen gleiche Größe haben:

```
Matrix<float> A(B.rowsize(),B.colsize()); // make correct size
A = B;                    // copy elements
```

- Ausschnitte von Matrizen (Untermatrizen)

```
Matrix<float> F(A.sub(1,2,3,4));// 3x4 Mat ab (1,2)
```

Rechnen mit Matrizen

- Vektorraumoperationen

```
A += B;            // A = A+B
A -= B;            // A = A-B
A *= 1.23;         // Multiplikation mit Skalar
A /= 1.23;         // Division durch Skalar
A.update(1.23,B); // A = A + s*B
```

- Matrix-Vektor und Matrizenmultiplikation

```
Vector<float> x(10,1.0); // make two vectors
Vector<float> y(10,2.0);
A.mv(y,x);               // y = A*x
A.umv(y,x);              // y = y + A*x
A.umv(y,(float)-1.0,x);  // y = y + s*A*x
C.mm(A,B);               // C = A*B
C.umm(A,B);              // C = C + A*B
```

Ausgabe und Hilfsfunktionen

- Ausgabe von Matrizen

```
std::cout << A.sub(0,0,3,3) << std::endl;// schöne Ausgabe
A.iwidth(2);                 // Stellen in Indexausgabe
A.width(10);                 // Anzahl Stellen gesamt
A.precision(4);              // Anzahl Nachkommastellen
std::cout << A << std::endl;// nun mit mehr Stellen
```

- einige Hilfsfunktionen

```
identity(A);
spd(A);
fill(x,(float)1,(float)1);
vandermonde(A,x);
```

Beispielausgabe

```
                0            1            2            3
  0   4.0000e+00 -1.0000e+00 -2.5000e-01 -1.1111e-01
  1  -1.0000e+00  4.0000e+00 -1.0000e+00 -2.5000e-01
  2  -2.5000e-01 -1.0000e+00  4.0000e+00 -1.0000e+00
  3  -1.1111e-01 -2.5000e-01 -1.0000e+00  4.0000e+00
```

Funktion mit Matrixargument

Beispiel einer Funktion, die eine Matrix A und einen Vektor b initialisiert.

```
template<class T>
void initialize (Matrix<T> A, Vector<T> b)
{
  if (A.rowsize()!=A.colsize() || A.rowsize()==0)
    HDNUM_ERROR("need␣square␣and␣nonempty␣matrix");
  if (A.rowsize()!=b.size())
    HDNUM_ERROR("b␣must␣have␣same␣size␣as␣A");
  for (int i=0; i<A.rowsize(); ++i)
    {
      b[i] = 1.0;
      for (int j=0; j<A.colsize(); ++j)
        if (j<=i) A[i][j]=1.0; else A[i][j]=0.0;
    }
}
```

4 Gewöhnliche Differentialgleichungen

4.1 Differentialgleichungsmodelle und Löser

Gewöhnliche Differentialgleichungen in HDNUM

- Erlaube Lösung beliebiger Modelle mit beliebigen Lösern.
- Erlaube variable Typen für Zeit und Zustand.
- Trenne folgende Komponenten:
  - Differentialgleichungsmodell (inklusive Anfangsbedingung),
  - Lösungsverfahren,
  - Steuerung und Zeitschleife.

Differentialgleichungsmodell

Ein Differentialgleichungsmodell ist gegeben durch

- Typen für Zeit und Zustandskomponenten variabel.
- Größe des Systems d.
- Anfangszustand (t_0, u_0).
- Funktion $f(t, x) : \mathbb{R} \times \mathbb{R}^d \to \mathbb{R}^d$.
- Optional die Jacobimatrix $f_x(t, x)$ (wird für implizite Verfahren benötigt).
- Für Zustand und Jacobimatrix verwenden wir Vektor- und Matrixklassen aus HD-NUM.

Als nächstes ein Beispiel für das Modellproblem

$$u'(t) = \lambda u(t), \quad t \geq t_0, \quad u(t_0) = u_0, \quad \lambda \in \mathbb{R}, \mathbb{C}.$$

Modellproblem (Datei `examples/modelproblem.hh`)

```
1 /** @brief Example class for a differential equation model
2
3     The model is
4
5     u'(t) = lambda*u(t), t>=t_0, u(t_0) = u_0.
6
7     \tparam T a type representing time values
8     \tparam N a type representing states and f-values
9 */
10 template<class T, class N=T>
11 class ModelProblem
12 {
13 public:
14   /** \brief export size_type */
15   typedef std::size_t size_type;
16
17   /** \brief export time_type */
18   typedef T time_type;
19
20   /** \brief export number_type */
21   typedef N number_type;
22
23   //! constructor stores parameter lambda
24   ModelProblem (const N& lambda_)
```

```
25     : lambda(lambda_)
26   {}
27
28   //! return number of componentes for the model
29   std::size_t size () const
30   {
31     return 1;
32   }
33
34   //! set initial state including time value
35   void initialize (T& t0, Vector<N>& x0) const
36   {
37     t0 = 0;
38     x0[0] = 1.0;
39   }
40
41   //! model evaluation
42   void f (const T& t, const Vector<N>& x, Vector<N>& result) const
43   {
44     result[0] = lambda*x[0];
45   }
46
47   //! jacobian evaluation needed for implicit solvers
48   void f_x (const T& t, const Vector<N>& x, Matrix<N>& result) const
49   {
50     result[0] = lambda;
51   }
52
53 private:
54   N lambda;
55 };
```

Differentialgleichungslöser

- Differentialgleichungsmodell ist ein Template-Parameter.
- Typen für Zeit und Zustand werden aus Differentialgleichungsmodell genommen.
- Kapselt aktuellen Zustand und aktuelle Zeit (und evtl. weitere Zustände).
- Methode `step` führt einen Schritt des Verfahrens durch.

Als nächstes ein Beispiel für den expliziten Euler.

Expliziter Euler (Datei `examples/expliciteuler.hh`)

```
1 /** @brief Explicit Euler method as an example for an ODE solver
2
3     The ODE solver is parametrized by a model. The model also
4     exports all relevant types for time and states.
5     The ODE solver encapsulates the states needed for the computation.
6
7     \tparam M the model type
8 */
9 template<class M>
10 class ExplicitEuler
11 {
12 public:
13   /** \brief export size_type */
14   typedef typename M::size_type size_type;
15
16   /** \brief export time_type */
17   typedef typename M::time_type time_type;
18
19   /** \brief export number_type */
20   typedef typename M::number_type number_type;
21
22   //! constructor stores reference to the model
23   ExplicitEuler (const M& model_)
24     : model(model_), u(model.size()), f(model.size())
25   {
26     model.initialize(t,u);
```

```
27      dt = 0.1;
28    }
29
30    //! set time step for subsequent steps
31    void set_dt (time_type dt_)
32    {
33      dt = dt_;
34    }
35
36    //! do one step
37    void step ()
38    {
39      model.f(t,u,f);    // evaluate model
40      u.update(dt,f);    // advance state
41      t += dt;           // advance time
42    }
43
44    //! get current state
45    const Vector<number_type>& get_state () const
46    {
47      return u;
48    }
49
50    //! get current time
51    time_type get_time () const
52    {
53      return t;
54    }
55
56    //! get dt used in last step (i.e. to compute current state)
57    time_type get_dt () const
58    {
59      return dt;
60    }
61
62  private:
63    const M& model;
64    time_type t, dt;
65    Vector<number_type> u;
66    Vector<number_type> f;
67  };
```

Lösung und Ergebnisausgabe

Die Lösung eines Differentialgleichungsmodells besteht nun aus

- Instantieren der entsprechenden Objekte für Modell und Löser.
- Zeitschrittschleife bis zur gewünschten Endzeit.
- Speicherung und Ausgabe der Ergebnisse in einem `std::vector`.
- Visualisierung der Ergebnisse mit `gnuplot`.

Hauptprogramm für Modellproblem (Datei `examples/modelproblem.cc`)

```
1 #include <iostream>
2 #include <vector>
3 #include <gmpxx.h>
4 #include "hdnum.hh"
5
6 using namespace hdnum;
7
8 #include "modelproblem.hh"
9 #include "expliciteuler.hh"
10
11 int main ()
12 {
13   typedef double Number;                  // define a number type
14
15   typedef ModelProblem<Number> Model;     // Model type
16   Model model(-1.0);                      // instantiate model
17
18   typedef ExplicitEuler<Model> Solver; // Solver type
```

```
19   Solver solver(model);                  // instantiate solver
20   solver.set_dt(0.1);                    // set initial time step
21
22   std::vector<Number> times;             // store time values here
23   std::vector<Vector<Number> > states;   // store states here
24   times.push_back(solver.get_time());    // initial time
25   states.push_back(copy(solver.get_state())); // initial state
26
27   while (solver.get_time()<10.0-1e-6) // the time loop
28     {
29       solver.step();                     // advance model by one time step
30       times.push_back(solver.get_time()); // save time
31       states.push_back(copy(solver.get_state())); // and state
32     }
33
34   gnuplot("modelproblem.dat",times,states); // output model result
35
36   return 0;
37 }
```