Heidelberger Numerikbibliothek für die Lehre

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

18. Februar 2022

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_ws2011/`

- 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/num0` 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

```
// hallohdnum.cc
#include <iostream>     // notwendig zur Ausgabe
#include <vector>
#include "hdnum.hh"     // hdnum header

int main ()
{
  std::cout << "Numerik␣0␣ist␣ganz␣leicht!" << std::endl;
  std::cout << "1+1=" << 1+1 << std::endl;

  return 0;
}
```

- übersetzen im Verzeichnis `examples/num0` 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.

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.
- Dieser Programmierkurs ist auch Teil von HDNUM

Herunterladen von HDNUM

1. Einloggen
2. Herunterladen von HDNUM `git clone https://parcomp-git.iwr.uni-heidelberg.de/Teaching/hdnum.git`
3. Wechsle in das Verzeichnis `$ cd hdnum/examples/progkurs`
4. 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 `hallohdnum.cc` mit einem Editor: `$ gedit hallohdnum.cc`

```
// hallohdnum.cc
#include <iostream>    // notwendig zur Ausgabe
#include <vector>
#include "hdnum.hh"    // hdnum header

int main ()
{
  std::cout << "Numerik␣0␣ist␣ganz␣leicht!" << std::endl;
  std::cout << "1+1=" << 1+1 << std::endl;

  return 0;
}
```

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

- 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.

```
// zahlen.cc
#include <iostream>

int main ()
{
  unsigned int i; // uninitialisierte natuerliche Zahl
  double x(3.14); // initialisierte Fliessommazahl
  float y(1.0);   // einfache Genauigkeit
  short j(3);     // eine 'kleine' Zahl
  std::cout << "(i+x)*(y+j)=" << (i+x)*(y+j) << std::endl;

  return 0;
}
```

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.

```
// string.cc
#include <iostream>
#include <string>

int main ()
{
  std::string m1("Zeichen");
  std::string leer("␣␣␣");
  std::string m2("kette");
  std::cout << m1+leer+m2 << std::endl;

  return 0;
}
```

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

Mehr Zahlen

```
// mehrzahlen.cc
#include <iostream> // header für Ein-/Ausgabe
#include <complex>  // header für komplexe Zahlen

int main ()
{
  std::complex<double> y(1.0,3.0);
  std::cout << y << std::endl;

  return 0;
}
```

- GNU Multiprecision Library `http://gmplib.org/` erlaubt Zahlen mit vielen Stellen (hier 512 Stellen zur Basis 2).
- übersetzen mit: `$ g++ -I../../ -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

```
// eingabe.cc
#include <iostream> // header für Ein-/Ausgabe
#include <iomanip>  // für setprecision
#include <cmath>    // für sqrt

int main ()
{
  double x(0.0);
  std::cout << "Gebe␣eine␣Zahl␣ein:␣";
  std::cin >> x;
  std::cout << "Wurzel(x)=␣"
            << std::scientific << std::showpoint
            << std::setprecision(15)
            << sqrt(x) << std::endl;

  return 0;
}
```

- 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?

```
// whitespace.cc
#include <iostream> // includes auf eigener Zeile!
#include <iomanip>
#include <cmath>

int main(){
  double x(0.0);
  std::cout<<"Gebe eine lange Zahl ein: ";std::cin >> x;
  std::cout<<"Wurzel(x)= "<<std::scientific<<std::showpoint
           <<std::setprecision(16)<<sqrt(x)<< std::endl;

  return 0;
}
```

2.3 Entscheidung

`If`-Anweisung

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

  Für $x \in \mathbb{R}$ setze
  $$y = |x| = \begin{cases} x & \text{falls } x \geq 0 \\ -x & sonst \end{cases}$$

- 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:

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

- 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:

```
// pendelwhile.cc
#include <iostream> // header für Ein-/Ausgabe
#include <cmath>    // mathematische Funktionen
int main ()
{
  double l(1.34);    // Pendellänge in Meter
  double phi0(0.2);  // Amplitude im Bogenmaß
  double dt(0.05);   // Zeitschritt in Sekunden
  double T(30.0);    // Ende in Sekunden
  double t(0.0);     // Anfangswert

  while ( t<=T )
  {
    std::cout << t << "␣"
              << phi0*cos(sqrt(9.81/l)*t)
              << std::endl;
    t = t + dt;
  }

  return 0;
}
```

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)

```
// pendel.cc
#include <iostream> // header für Ein-/Ausgabe
#include <cmath>    // mathematische Funktionen

int main ()
{
  double l(1.34);    // Pendellänge in Meter
  double phi0(0.2); // Amplitude im Bogenmaß
  double dt(0.05);   // Zeitschritt in Sekunden
  double T(30.0);    // Ende in Sekunden
  for (double t=0.0; t<=T; t=t+dt)
  {
    std::cout << t << "␣"
              << phi0*cos(sqrt(9.81/l)*t)
              << std::endl;
  }

  return 0;
}
```

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`

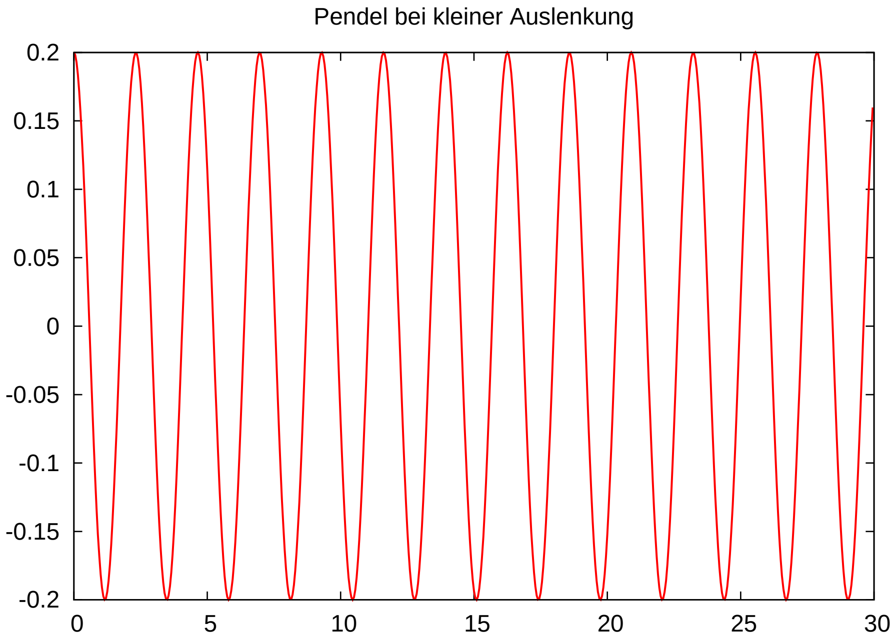

Abbildung 1: Das Pendel in Aktion. Gnuplot-Ausgabe des Programmes `pendel.cc`.

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

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

Pendel (expliziter Euler)

```
// pendelnumerisch.cc
#include <iostream> // header für Ein-/Ausgabe
#include <cmath>    // mathematische Funktionen

int main ()
{
  double l(1.34);   // Pendellänge in Meter
  double phi(3.0); // Anfangsamplitude in Bogenmaß
  double u(0.0);    // Anfangsgeschwindigkeit
  double dt(1E-4); // Zeitschritt in Sekunden
  double T(30.0);   // Ende in Sekunden
  double t(0.0);    // Anfangszeit

  std::cout << t << "␣" << phi << std::endl;
  while (t<T)
  {
    t = t + dt;          // inkrementiere Zeit
    double phialt(phi);// merke phi
    double ualt(u);     // merke u
    phi = phialt + dt*ualt;              // neues phi
    u = ualt - dt*(9.81/l)*sin(phialt);// neues u
    std::cout << t << "␣" << phi << std::endl;
  }

  return 0;
}
```

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

```
// funktion.cc
#include <iostream>

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

int main ()
{
  double x(2.0);
  std::cout << "f(" << x << ")=" << f(x) << std::endl;

  return 0;
}
```

- 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

```
// pendelmitfunktion.cc
#include <iostream> // header für Ein-/Ausgabe
#include <cmath>     // mathematische Funktionen

void simuliere_pendel (double l, double phi, double u)
{
  double dt    = 1E-4;
  double T     = 30.0;
  double t     = 0.0;

  std::cout << t << " " << phi << std::endl;
  while (t<T)
  {
    t = t + dt;
    double phialt(phi),ualt(u);
    phi = phialt + dt*ualt;
```

```
    u = ualt - dt*(9.81/l)*sin(phialt);
    std::cout << t << "␣" << phi << std::endl;
  }
}

int main ()
{
  simuliere_pendel(1.34,3.0,0.0);

  return 0;
}
```

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

```
// pendelmitfunktionstemplate.cc
#include <iostream> // header für Ein-/Ausgabe
#include <cmath>    // mathematische Funktionen

template<typename Number>
void simuliere_pendel (Number l, Number phi, Number u)
{
  Number dt(1E-4);
  Number T(30.0);
  Number t(0.0);
  Number g(9.81/l);

  std::cout << t << "␣" << phi << std::endl;
  while (t<T)
  {
    t = t + dt;
    Number phialt(phi),ualt(u);
    phi = phialt + dt*ualt;
    u = ualt - dt*g*sin(phialt);
    std::cout << t << "␣" << phi << std::endl;
  }
```

```
}

int main ()
{
  float l1(1.34);   // Pendellänge in Meter
  float phi1(3.0); // Anfangsamplitude in Bogenmaß
  float u1(0.0);    // Anfangsgeschwindigkeit
  simuliere_pendel(l1,phi1,u1);

  double l2(1.34);   // Pendellänge in Meter
  double phi2(3.0); // Anfangsamplitude in Bogenmaß
  double u2(0.0);    // Anfangsgeschwindigkeit
  simuliere_pendel(l2,phi2,u2);

  return 0;
}
```

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

HDNUM stellt Matrix und Vektorklassen zur Verfügung.

3.1 Vektoren

`hdnum::Vector<T>`

- `hdnum::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

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

- Speziellere Vektoren

```
hdnum::Vector<std::complex<double> >
  cx(7,std::complex<double>(1.0,3.0));
mpf_set_default_prec(1024); // Setze Genauigkeit für mpf_class
hdnum::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: Erstellen eines Vektors als Kopie eines anderen

```
hdnum::Vector<float> z(x); // z ist Kopie von x
```

- Zuweisung nach Initialisierung, beide Vektoren müssen die gleiche Größe haben!

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

- Ausschnitte von Vektoren

```
hdnum::Vector<float> w(x.sub(7,3));// w ist Kopie von x[7],...,x[9]
z = x.sub(3,4);              // z ist Kopie von 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]  1.2042000770568848e+01
[ 1]  1.2042000770568848e+01
[ 2]  1.2042000770568848e+01
[ 3]  1.2042000770568848e+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 (hdnum::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 (hdnum::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

`hdnum::DenseMatrix<T>`

- `hdnum::DenseMatrix<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

```
hdnum::DenseMatrix<float> B(10,10);      // 10x10 Matrix uninitialisiert
hdnum::DenseMatrix<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: Erstellen einer Matrix als Kopie einer anderen

```
hdnum::DenseMatrix<float> D(B); // D Kopie von B
```

- Zuweisung nach Initialisierung, beide Matrizen müssen gleiche Größe haben:

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

- Ausschnitte von Matrizen (Untermatrizen)

```
hdnum::DenseMatrix<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

```
hdnum::Vector<float> x(10,1.0); // make two vectors
hdnum::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 (hdnum::DenseMatrix<T> A, hdnum::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 HDNUM.

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/num1/modelproblem.hh)

```cpp
/** @brief Example class for a differential equation model

    The model is

    u'(t) = lambda*u(t), t>=t_0, u(t_0) = u_0.

    \tparam T a type representing time values
    \tparam N a type representing states and f-values
*/
template<class T, class N=T>
class ModelProblem
{
public:
  /** \brief export size_type */
  typedef std::size_t size_type;

  /** \brief export time_type */
  typedef T time_type;

  /** \brief export number_type */
  typedef N number_type;

  //! constructor stores parameter lambda
  ModelProblem (const N& lambda_)
    : lambda(lambda_)
```

```
  {}

  //! return number of components for the model
  std::size_t size () const
  {
    return 1;
  }

  //! set initial state including time value
  void initialize (T& t0, hdnum::Vector<N>& x0) const
  {
    t0 = 0;
    x0[0] = 1.0;
  }

  //! model evaluation
  void f (const T& t, const hdnum::Vector<N>& x, hdnum::Vector<N>& result) const
  {
    result[0] = lambda*x[0];
  }

  //! exact solution if known
  void exact_solution (const T& t, hdnum::Vector<N>& result) const
  {
    result.resize(size());
    result[0] = exp(lambda*t);
  }

  //! jacobian evaluation needed for implicit solvers
  void f_x (const T& t, const hdnum::Vector<N>& x, hdnum::DenseMatrix<N>& result) const
  {
    result[0][0] = lambda;
  }

private:
  N lambda;
};
```

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/num1/expliciteuler.hh`)

```
/** @brief Explicit Euler method as an example for an ODE solver

    The ODE solver is parametrized by a model. The model also
    exports all relevant types for time and states.
    The ODE solver encapsulates the states needed for the computation.

    \tparam M the model type
*/
template<class M>
class ExplicitEuler
{
public:
  /** \brief export size_type */
  typedef typename M::size_type size_type;

  /** \brief export time_type */
  typedef typename M::time_type time_type;

```

```
  /** \brief export number_type */
  typedef typename M::number_type number_type;

  //! constructor stores reference to the model
  ExplicitEuler (const M& model_)
    : model(model_), u(model.size()), f(model.size())
  {
    model.initialize(t,u);
    dt = 0.1;
  }

  //! set time step for subsequent steps
  void set_dt (time_type dt_)
  {
    dt = dt_;
  }

  //! do one step
  void step ()
  {
    model.f(t,u,f);    // evaluate model
    u.update(dt,f);    // advance state
    t += dt;           // advance time
  }

  //! get current state
  const hdnum::Vector<number_type>& get_state () const
  {
    return u;
  }

  //! get current time
  time_type get_time () const
  {
    return t;
  }

  //! get dt used in last step (i.e. to compute current state)
  time_type get_dt () const
  {
    return dt;
  }

private:
  const M& model;
  time_type t, dt;
  hdnum::Vector<number_type> u;
  hdnum::Vector<number_type> f;
};
```

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 `hdnum::Vector`.
- Visualisierung der Ergebnisse mit `gnuplot`.

Hauptprogramm für Modellproblem

(Datei `examples/num1/modelproblem.cc`)

```
#include <iostream>
#include <vector>
#include "hdnum.hh"

#include "modelproblem.hh"
#include "expliciteuler.hh"

int main ()
```

```
{
  typedef double Number;                    // define a number type

  typedef ModelProblem<Number> Model;   // Model type
  Model model(-1.0);                        // instantiate model

  typedef ExplicitEuler<Model> Solver; // Solver type
  Solver solver(model);                     // instantiate solver
  solver.set_dt(0.02);                       // set initial time step

  hdnum::Vector<Number> times;                // store time values here
  hdnum::Vector<hdnum::Vector<Number> > states; // store states here
  times.push_back(solver.get_time());   // initial time
  states.push_back(solver.get_state()); // initial state

  while (solver.get_time()<5.0-1e-6) // the time loop
    {
      solver.step();                        // advance model by one time step
      times.push_back(solver.get_time()); // save time
      states.push_back(solver.get_state()); // and state
    }

  gnuplot("mp2-ee-0.02.dat",times,states); // output model result

  return 0;
}
```

Literatur

[1] P. Deuflhard and A. Hohmann. *Numerische Mathematik I, Eine algorithmisch orientierte Einführung*. de Gruyter, 2002.

[2] K. Erikson, D. Estep, P. Hansbo, and C. Johnson. *Computational Differential Equations*. Cambridge University Press, 1996.

[3] G. Golub and J. M. Ortega. *Scientific Computing*. Teubner, 1996.

[4] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 2nd edition, 1989.

[5] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical Mathematics*. Springer, 2nd edition, 2007.

[6] R. Rannacher. Einführung in die Numerische Mathematik (Numerik 0). `http://numerik.iwr.uni-heidelberg.de/~lehre/notes`, 2006.

[7] R. Schaback and H. Wendland. *Numerische Mathematik*. Springer, 5th edition, 2005.

[8] H.-R. Schwarz and N. Köckler. *Numerische Mathematik*. Teubner, 5. edition, 2005.

[9] J. Stoer. *Numerische Mathematik I*. Springer, 9. edition, 2005.

[10] J. Stoer and R. Bulirsch. *Numerische Mathematik II*. Springer, 5. edition, 2005.