

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`

16. Mai 2013

Inhalt

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 muss 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 xzf 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://gmpilib.org/>. Ist diese nicht vorhanden müssen Makefiles entsprechend angepasst werden.

Typisches HDNUM Programm

```

1 // hallohdnum.cc
2 #include <iostream>           // notwendig zur Ausgabe
3 #include <vector>
4 #include "hdnum.hh"          // hdnum header
5
6 int main ()
7 {
8     hdnum::Vector<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`

Inhalt

2 Ein kleiner Programmierkurs

- Hallo Welt
- Variablen und Typen
- Entscheidung
- Wiederholung
- Funktionen

Programmierungsumgebung

- 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 Textdateien 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/numerik1_ws2011/
- 5 Klicke auf **Version 0.24** und bestätige
- 6 Kopiere Datei `hdnum-0.24.tgz` in das Verzeichnis:
`$ cp ~/Desktop/hdnum-0.24.tgz .`
- 7 Entpacken der Datei mit
`$ tar zxvf hdnum-0.24.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
```

(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 <complex> // header für komplexe Zahlen
4 int main ()
5 {
6     std::complex<double> y(1.0,3.0);
7     std::cout << y << std::endl;
8 }
```

- 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
4 #include <cmath> // für sqrt
5 int main ()
6 {
7     double x(0.0);
8     std::cout << "Gebe_eine_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 <cmath>
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;}
```

If-Anweisung

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

Für $x \in \mathbb{R}$ setze

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

- 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 }
```

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;
}
```

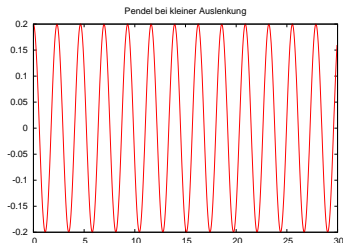
- Enthält der Block nur eine Anweisung dann kann man die geschweiften Klammern weglassen.
- Die *Schleifenvariable* ist so nur innerhalb des Schleifenkörpers sichtbar.
- Die **for**-Schleife kann auch mittels einer *while*-Schleife realisiert werden.

Pendel (analytische Lösung, **for**-Schleife)

```
1 // pendel.cc
2 #include <iostream> // header für Ein-/Ausgabe
3 #include <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    for (double t=0.0; t<=T; t=t+dt)
11    {
12        std::cout << t << "␣"
13                << phi0*cos(sqrt(9.81/l)*t)
14                << std::endl;
15    }
16 }
```

Visualisierung mit Gnuplot

- Gnuplot erlaubt einfache Visualisierung von Funktionen $f : \mathbb{R} \rightarrow \mathbb{R}$ und $g : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$.
- Für $f : \mathbb{R} \rightarrow \mathbb{R}$ 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`



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:

$$\frac{d^2\phi(t)}{dt^2} = -\frac{g}{l} \sin(\phi(t)) \quad \forall t > 0,$$
$$\phi(0) = \phi_0, \quad \frac{d\phi}{dt}(0) = u_0.$$

- 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 }
```

Funktionsaufruf und Funktionsdefinition

- In der Mathematik gibt es das Konzept der *Funktion*.
- In C++ auch.
- Sei $f : \mathbb{R} \rightarrow \mathbb{R}$, 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 Rückgabetyt `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     double l(1.34); // Pendellänge in Meter
```

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 }
```

Pendelsimulation mit Templates II

```
23
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 }
```

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.

Inhalt

3 Vektoren und Matrizen

- Vektoren
- Matrizen



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  
hdnum::Vector<double> y(10,3.14);    // 10 Elemente in  
hdnum::Vector<float> a;               // ein leerer Vek
```

- Speziellere Vektoren

```
hdnum::Vector<std::complex<double> >  
    cx(7, std::complex<double>(1.0,3.0));  
mpf_set_default_prec(1024); // Setze Genauigkeit für  
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[7:9]
z = x.sub(3,4); // z ist Kopie von x[3:6]
```

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]
unitvector(w,2); // kartesischer Einheitsvekt
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!

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 Mat  
hdnum::DenseMatrix<float> C(10,10,0.0);       // 10x10 Mat
```

- 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 initialis
```

- 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())  
A = B; // copy elements
```

- Ausschnitte von Matrizen (Untermatrizen)

```
hdnum::DenseMatrix<float> F(A.sub(1,2,3,4)); // 3x4 M
```

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 A
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<
{
    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;
    }
}
```

Inhalt

4 Gewöhnliche Differentialgleichungen

- 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 $f(t, x) : \mathbb{R} \times \mathbb{R}^d \rightarrow \mathbb{R}^d$.
- 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 I

(Datei `examples/modelproblem.hh`)



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 I

(Datei `examples/expliciteuler.hh`)

Lösung und Ergebnisausgabe

Die Lösung eines Differentialgleichungsmodells besteht nun aus

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



Hauptprogramm für Modellproblem I

(Datei `examples/modelproblem.cc`)

Literatur I



Rannacher, R.: *Einführung in die Numerische Mathematik (Numerik 0)*.

<http://numerik.iwr.uni-heidelberg.de/~lehre/notes>,
2006.



Stoer, J.: *Numerische Mathematik I*.

Springer, 9. Auflage, 2005.