HDNUM
CPP Review Proyecto para Latinoamérica

Peter Bastian
Universität Heidelberg,
Traducido por John Jairo Leal G,
Universidad Nacional de Colombia,
Línea de investigación en Modelamiento Matemático

4 de junio de 2021

Resumen

Este manual de Hdnum, está basado en la propuesta presentada por la universidad de Heidelberg Alemania en el curso de Lima, Perú en 2020, para introducirse en el manejo de las soluciones de ecuaciones diferenciales ordinarias y parciales.

Índice

1. Introducción

Haremos un breve resumen de HDNUM

1.1. Que es HDNUM?

La biblioteca numérica de Heildeberg HDNUM, es una biblioteca basada en C++ para realizar ejercicios prácticos en clases magistrales, que incluye métodos numéricos para resolver ecuaciones diferenciales ordinarias, la actual versión está disponible en :

`https://parcomp-git.iwr.uni-heidelberg.de/Teaching/hdnum`

Que tiene un control de vesiones administrada por `git` . Spezifische Versionen können auf der jeweiligen Vorlesungswebseite veröffentlicht werden.

Los objetivos en el desarrollo de HDNUM fueron i) la facilidad de uso (incluida la instalación simple), ii) la demostración de la programación orientada a objetos en la solución de métodos numéricos, así como la posibilidad de realizar cálculos con cualquier grado de precisión sobre la base de Gnu Multiple Precision Biblioteca. HDNUM ofrece actualmente las siguientes funciones:

1) Clases para vectores y matrices

2) Solución de sistemas de ecuaciones lineales

3) Solución de sistemas de ecuaciones no lineales

4) Solución de la ecuación de Poisson utilizando diferencias finitas

1.2. Instalación

HDNUM es una biblioteca de ”solo encabezado” y no requiere ninguna instalación más que descargar los archivos. La versión actual se puede encontrar usando el siguiente comando para descargarla:

```
$ git clone https://parcomp-git.iwr.uni-heidelberg.de/Teaching/hdnum.git
```

Para realizar esto se requiere el programa `git`, el cual está disponible gratuitamente para todos los sistemas operativos. Alternativamente también hay un archivo comprimido que se puede descargar `tar` en la página del evento.

```
$ tar zxvf hdnum-XX.tgz
```

Los siguientes archivos y subdirectorios se pueden encontrar en el directorio instalado o descomprimido:

- `hdnum.hh`: Este archivo encabezado debe estar integrado en los programa `C++` para poder utilizar HDNUM.
- El directorio `mystuff` está destinado a sus programas, pero, por supuesto, puede utilizar cualquier otro directorio. Lo único importante es que el compilador tenga el archivo `hdnum.hh` que encontró. En el registro mystuff ya es un programa de muestra para comenzar de inmediato. Este programa se puede compilar de la siguiente forma:

```
$ cd mystuff
$ g++ -I.. -o example example.cc
```

  Otra forma es utilizar el comando `make`, el cual ejecutará el archivo `Makefile` que está en la carpeta, y el cual compilará el programa ejemplo, generando el archivo ejecutable `example` que lo puede correr con `./example`.

  Estos comandos requieren que el compilador GNU C ++ esté instalado en su sistema. En Windows o para otros compiladores, debe adaptar los comandos en consecuencia.

- El directorio `examples` en la carpeta HDNUM contiene muchos ejemplos ordenados para el curso de programación, `num0` y `num1`.
- El directorio `src` en la carpeta HDNUM contiene el código fuente de la biblioteca HDNUM. Estos archivos son utilizados por hdnum.hh cuando sea invocado.
- El directorio `programmierkurs` en la carpeta HDNUM contiene el código fuente de este documento
- El directorio `tutorial` en la carpeta HDNUM contiene las diapositivas del curso de programación.

GNU Biblioteca de Multiple Precisión

HDNUM puede realizar cálculos con gran precisión. Esto requiere la biblioteca GNU Multiple Precision Library (GMP), que puede obtener de forma gratuita para muchos sistemas. Para poder utilizar GMP debe poner en el archivo `hdnum.hh` la línea:

```
#define HDNUM_HAS_GMP 1
```

Además, las opciones del compilador pueden ser necesarias para que el compilador pueda encontrar las bibliotecas y los archivos de encabezado GMP. Entonces puede verse así:

```
$ g++ -I.. -I/opt/local/include -o example example.cc -L/opt/local/lib -lgmpxx -lgmp
```

2. Algebra Lineal

2.1. Vectores

`hdnum::Vector<T>`

- `hdnum::Vector<T>` es una plantilla de Clase.
- Convierte cualquier (número) tipo de datos `T` en un vector.

- También son posibles números complejos y muy precisos.
- Los vectores se comportan como los conocemos por las matemáticas:
  - Tiene n componentes.
  - Inicia en el elemento 0 hasta el elemento $n-1$ numerados consecutivamente.
  - Adición y multiplicación por un escalar.
  - Producto escalar y norma Euclidiana.
  - Multiplicación vector-matriz.
- Los siguientes ejemplos se pueden encontrar en el archivo `vektoren.cc`

Construcción y Acceso

- Construcción con y sin Inicialización

```
hdnum::Vector<float> x(10);         // Vector con 10 elementos
hdnum::Vector<double> y(10,3.14);   // Vector con 10 elementos inicializado
hdnum::Vector<float> a;             // Vector sin elementos
```

- Vectores más específicos

```
hdnum::Vector<std::complex<double> >
  cx(7,std::complex<double>(1.0,3.0));
mpf_set_default_prec(1024); // Establece precision para  mpf_class
hdnum::Vector<mpf_class> mx(7,mpf_class("4.44"));
```

- Acceso a un elemento

```
for (std::size_t i=0; i<x.size(); i=i+1)
  x[i] = i;                  // Acceso a cada elemento
```

- El objeto vectorial se elimina al final del ciclo for.

Copia y Asignación

- Constructor copia: crea una copia de otro vector

```
hdnum::Vector<float> z(x); // z es copia de x
```

- Asignación, ¡El tamaño también cambia!

```
b = z;               // b copia los datos de z
a = 5.4;             // asignacion a todos los elementos
hdnum::Vector<double> w;   // Vector sin elementos
w.resize(x.size()); // Redimiensiona el vector
w = x;               // Copia los elementos
```

- Extracto de vectores

```
hdnum::Vector<float> w(x.sub(7,3));// w es una copia de x[7],...,x[9]
z = x.sub(3,4);              // z es una copia de x[3],...,x[6]
```

Cálculos y operaciones

- Operaciones de espacio vectorial y producto escalar

```
w += z;             // w = w+z
w -= z;             // w = w-z
w *= 1.23;          // Multiplicacion por escalar
w /= 1.23;          // Division por escalar
w.update(1.23,z);   // w = w + a*z
float s;
s = w*z;            // Producto escalar
```

- Mostrando las salidas

```
std::cout << w << std::endl;// Salida por pantalla
w.iwidth(2);                // Digitos en la salida del indice
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 fuer complex
std::cout <<mx << std::endl;// geht auch fuer 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

```
float d = norm(w);          // Euklidsche Norm
d = w.two_norm();           // das selbe
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;
}
```

- Verbesserte Version mit **Funktionentemplate** und by-const-reference Übergabge

```
template<class T>
T sum (const 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;
}
```

- By-value Übergabe ist be großen Objekten vorzuziehen

2.2. Matrices

`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`

Construcción y acceso

- 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.

Copia y asignación

- Copy-Konstruktor: Erstellen einer Matrix als Kopie einer anderen

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

- Zuweisung, kopiert auch Größe mit

```
hdnum::DenseMatrix<float> A(B.rowsize(),B.colsize()); // korrekte Groesse
A = B;                      // kopieren
```

- Ausschnitte von Matrizen (Untermatrizen)

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

Calculando con matrices

- Vektorraumoperationen (Vorsicht: Matrizen sollten passende Größe haben!)

```
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
A.sc(x,1);                 // mache x zur ersten Spalte von A
A.sr(x,1);                 // mache x zur ersten Zeile von A
float d=A.norm_infty();    // Zeilensummennorm
d=A.norm_1();              // Spaltensummennorm
```

Funciones de salida auxiliares

- Ausgabe von Matrizen

```
std::cout << A.sub(0,0,3,3) << std::endl;// öschne 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);
```

Salida de muestra

```
                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
```

Funciones con argumentos matriciales

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;
    }
}
```

Im Folgenden geht es um Löser für Gleichungssysteme. Ist das zu lösende Gleichungssystem linear, so verwendet man eine LR- oder QR-Zerlegung. Im nichtlinearen Fall, der beispielsweise bei den in Numerik 1 behandelten impliziten Runge-Kutta Verfahren auftritt, macht man sich Fixpunktiterationen, wie z.B. im Newtonverfahren zunutze.

La siguiente sección trata sobre solucionadores de sistemas de ecuaciones. Sirven para resolver un sistema de ecuaciones lineales, se usa un LR o la descomposición QR. En el caso no lineal, que es el caso de los métodos Numéricos 1 tratados implícitamente ocurre el procedimiento de Runge-Kutta, hace uno usa iteraciones de punto fijo, por ejemplo, en el método de Newton.

2.3. LR Desmontaje

2.3.1. Breve explicación del algoritmo

Die LR-Zerlegung wird angewandt, um ein Gleichungssystem der Form $Ax = b$ zu lösen. Dabei wird die reguläre, quadratische Matrix A in eine linke untere Dreiecksmatrix L und in eine rechte obere Dreiecksmatrix R Zerlegt, sodass $A = LR$. Zusätzlich ist meistens eine Pivotisierung erforderlich, womit man schließlich auf das System $PA = LR$ kommt. Durch diese Pivotisierung ist sichergestellt, dass das Diagonalelement der Matrix ungleich null ist, denn sonst könnten wir unseren Algorithmus nicht anwenden.

Dabei unterscheidet man zum einen die Partialpivotisierung, bei der sichergestellt wird, dass das betragsmäßig größte Element der Spalte unterhalb der Diagonale durch Zeilenpermutationen auf die Diagonale getauscht wird. Bei der Totalpivotisierung betrachtet man die ganze Matrix unterhalb der Diagonalen und sucht dort das betragsmäßig größte Element, um es durch Zeilen- und Spaltenoperationen auf den aktuellen Diagonaleintrag zu tauschen. Das betragsmäßig größte Element wird gewählt, um numerische Fehler zu verringern.

2.3.2. LR Como hacer el desmontaje

Will man ein Programm schreiben, dass ein lineares Gleichungssystem der Form $Ax = b$ mittels LR-Zerlegung der Matrix A löst, ist wie folgt vorzugehen:

- Zu Beginn erstellt man den Rechteseitevektor b und die Matrix A. Dies sieht zum Beispiel so aus:

```
Vector<number> b(3);
b[0]=15;
b[1]=73;
b[2]=12;

DenseMatrix <number> A(3,3);
A[0][0]=2;   A[0][1]=1;   A[0][2]=7;
A[1][0]=8;   A[1][1]=8;   A[1][2]=33;
A[2][0]=-4;  A[2][1]=10;  A[2][2]=4;
```

- Zusätzlich benötigen wir Vektoren x und p. Falls eine Totalpivotisierung durchgeführt wird, ist ein weiterer Vektor q zu erstellen. Um die Kondition der Matrix zu

verbessern können die Funktionen `row_equilibrate` sowie `apply_equilibrate` angewendet werden, für die ein zusätzlicher Vektor s benötigt wird:

```
Vector<number> x(3,0.0);
Vector<number> s(3);
Vector<std::size_t> p(3);
Vector<std::size_t> q(3);
```

- Wie bereits im vorangegangenen Punkt erwähnt, kann zu Beginn die Kondition der Matrix A verbessert werden. Dies geschieht mithilfe der Funktionen `row_equilibrate` und `apply_equilibrate`. Wie diese angewendet werden, kann man in den nachfolgenden Punkten sehen.

- Nun wendet man eine der folgenden Funktionen auf die Matrix A und den zuvor erstellten Permutationsvektor p an. In unserem Beispiel führen wir eine Totalpivotisierung durch, weshalb wir den zusätzlichen Vektor q benötigen:

  ```
  row_equilibrate (A,s);
  lr_fullpivot(A,p,q);
  ```

  Für eine Partialpivotisierung ist die Funktion `lr_partialpivot`, für eine LR-Zerlegung ohne Pivotisierung die Funktion `lr` zu verwenden. (Hierbei ist der zusätzliche Permutationsvektor q nicht notwendig.) Jetzt können wir das Gleichungssystem für viele unterschiedliche rechte Seiten lösen.

- Dafür müssen wir die rechte Seite wie folgt vorbereiten:

  ```
  apply_equilibrate(s,b);
  permute_forward(p,b);
  ```

- Daraufhin rufen wir die Funktion `solveL` auf, die als Parameter die Matrix A, den Rechteseitevektor b und einen Vektor y bekommt, in dem die Lösung des Gleichungssystems $Ly = b$ gespeichert wird. Um Speicherplatz zu sparen können wir das Ergebnis gleich in den bereits vorhandenen Vektor b schreiben.

- Schließlich wird noch die Funktion `solveR` benötigt, die das Gleichungssystem $Rx = y$ löst. Die Funktion braucht als Parameter die Matrix A, den Rechteseitevektor y (des Gleichungssystems $Ly = b$) sowie den Vektor x, in dem das endgültige Ergebnis gespeichert wird:

  ```
  solveL(A,b,b);
  solveR(A,x,b);
  ```

- Falls eine Totalpivotisierung vorgenommen wurde, müssen zum Schluss die Permutationen, die im Vektor q (Spaltentransvormationen von A) gespeichert wurden, auf das Ergebnis x mittels `permute_backward` angewendet werden:

- Die Lösung des linearen Gleichungssystems ist nun im Vektor x gespeichert. In unserem Fall:

  ```
  x[0] = 1
  x[1] = 2
  x[2] = 3
  ```

2.4. Explicaciones detalladas del algoritmo `lr.hh`

- **Die Funktion** `lr`: Zu Beginn wird bei allen Funktionen überprüft, ob eine quadratische, nichtleere Matrix vorliegt und ob der Vektor p mit der gegebenen Matrix kompatibel ist. In der ersten for-Schleife wird eine Zeile der Matrix gesucht, deren Diagonalelement ungleich null ist. Anschließend wird diese Zeile mit der des aktuellen Diagonalelements getauscht. Die Permutationen, welche durch die Pivotsuche entstehen, werden im Vektor p gespeichert.

```
for (std::size_t k=0; k<A.rowsize()-1; ++k)
{
  // finde Pivotelement und vertausche Reihen
  for (std::size_t r=k; r<A.rowsize(); ++r)
    if (A[r][k]!=0)
    {
      p[k] = r;   // speichere Permutation im Schritt k
      if (r>k)    //tausche komplette Reihe falls r!=k
        for (std::size_t j=0; j<A.colsize(); ++j)
        {
          T temp(A[k][j]);
          A[k][j] = A[r][j];
          A[r][j] = temp;
        }
      break;
    }
    if (A[k][k]==0) HDNUM_ERROR("matrix␣is␣singular");
    // Moifikation
    for (std::size_t i=k+1; i<A.rowsize(); ++i)
    {
      T qik(A[i][k]/A[k][k]);
      A[i][k] = qik;
        for (std::size_t j=k+1; j<A.colsize(); ++j)
          A[i][j] -= qik * A[k][j];
    }
}
```

  In der zweiten for-Schleife wird dann die obere Dreiecksmatrix mit dem permutierten A erstellt.

- **Die Funktion** `lr_partialpivot`:Parameter: Matrix A sowie Permutationsvektor p. Diese Funktion führt eine partielle Pivotisierung durch. Dabei geht sie wie folgt vor: Zunächst wird der Vektor p initialisiert, indem er mit den Werten 0 bis $n-1$ (wobei $A \in \mathbb{R}^{n\times n}$) beschrieben wird. Danach wird in der Matrix A das Pivotelement (betragsmäßig größte Element) in der aktuellen Spalte unterhalb des Diagonalelementes gesucht und die erforderliche Permutation um dieses auf die Diagonale zu tauschen im Vektor p gespeichert:

```
for (std::size_t k=0; k<A.rowsize()-1; ++k)
{
   // finde Pivotelement
  for (std::size_t r=k+1; r<A.rowsize(); ++r)
    if (abs(A[r][k])>abs(A[k][k]))
    p[k] = r; // speichert Permutation im Schritt k
}
```

  In der darauffolgenden Schleife werden die Zeilen k und j getauscht, sodass das Pivotelement auf der Diagonalen liegt.

- **Die Funktion** `lr_fullpivot`: Diese Funktion geht ähnlich wie die Funktion `lr_partialpivot` vor, allerdings braucht sie einen zusätzlichen Vektor q, um eine Totalpivotisierung

durchzuführen. Hierbei sind nicht nur Zeilen- sondern auch Spaltenvertauschungen möglich, welche im Vektor q gespeichert werden.

- **Die Funktion** `permute_forward`: Der Vektor p hat die notwendigen Permutationen gespeichert. In dieser Funktion werden die Zeilenpermutationen auf den Vektor b übertragen:

```
for (std::size_t k=0; k<b.size()-1; ++k)
  if (p[k]!=k)
  {
     T temp(b[k]);
     b[k] = b[p[k]];
     b[p[k]] = temp;
  }
```

- **Die Funktion** `permute_backward`: Diese Funktion wird am Ende der Lr-Zerlegung angewendet, um die in der Funktion `permute_forward` vorgenommenen Permutationen beim Rechteseitevektor wieder rückgängig zu machen.

- **Die Funktion** `row_equilibrate`: Diese Funktion wird vor dem eigentlichen Algorithmus angewendet, um die Kondition der Matrix zu verbessern (Equilibration). Die Werte, durch die die Zeilen der Matrix dividiert werden, sind im Vektor s gespeichert:

```
for (std::size_t k=0; k<A.rowsize(); ++k)
{
  s[k] = T(0.0);
    for (std::size_t j=0; j<A.colsize(); ++j)
      s[k] += abs(A[k][j]);
  if (s[k]==0) HDNUM_ERROR("row␣sum␣is␣zero");
    for (std::size_t j=0; j<A.colsize(); ++j)
      A[k][j] /= s[k];
}
```

- **Die Funktion** `apply_equilibrate`: Die Veränderungen, die an der Matrix A durchgeführt wurden, werden hier ebenfalls auf den Vektor b angewandt, um die Lösung nicht zu verfälschen.

- **Die Funktion** `solveL`: Parameter: Vektor x und Rechteseitevektor b. Diese Funktion löst die Gleichung $Lx = b$. Dabei wird x folgendermaßen, iterativ bestimmt: $x_i = b_i - \sum_{j=0}^{i-1} l_{ij} x_j$

```
for (std::size_t i=0; i<A.rowsize(); ++i)
{
  T rhs(b[i]);
  for (std::size_t j=0; j<i; j++)
    rhs -= A[i][j] * x[j];
  x[i] = rhs;
}
```

- **Die Funktion** `solveR`: Diese Funktion löst die Gleichung $Rx = b$. Dabei wird x folgendermaßen bestimmt: $x_i = b_i - \sum_{j=i+1}^{n-1} r_{ij} x_j$ (hierbei ist $R \in \mathbb{R}^{n \times n}$)

```
for (int i=A.rowsize()-1; i>=0; --i)
{
  T rhs(b[i]);
  for (std::size_t j=i+1; j<A.colsize(); j++)
    rhs -= A[i][j] * x[j];
  x[i] = rhs/A[i][i];
}
```

2.5. Iterationsverfahren - die Datei `newton.hh`

Jetzt wissen wir, wie wir lineare Gleichungssysteme der Form $Ax = b$ lösen können. Was ist jedoch zu tun, wenn das Gleichungssystem nicht linear ist, z.B. im simplen, eindimensionalen Fall $x^2 = a$? In der Vorlesung lernt man Verfahren, die sich Fixpunktiterationen zunutze machen, um der Lösung sehr nahe zu kommen. Die Datei `newton.hh` stellt hilfreiche Werkzeuge bereit, um derartige Gleichungen und Gleichungssystem zu lösen. Das wichtigste ist das Newtonverfahren, mit dem man nichtlineare Gleichungen der Form $F(x) = 0$ lösen kann. Zunächst betrachten wir jedoch die konkrete Formulierung eines Problems in einer Klasse.

2.5.1. Die Klasse `SquareRootProblem`

Um ein nichtlineares Gleichungssystem der Form $f(x) = 0$ lösen zu können müssen wir zu Beginn eine Klasse für unser Problem erstellen. Diese benötigt neben einem geeigneten Konstruktor und den Angaben über die Dimension des Problems eine Methode, die den Funktionswert bereitstellt, und eine andere, welche die Ableitung der Funktion bereitstellt. Wir zeigen dies beispielhaft an der Klasse SquareRootProblem:

```
class WurzelProblem
{
   public:
    typedef std::size_t size_type;
    typedef N number_type;
    WurzelProblem (number_type a_);
    std::size_t size () const;
    void F (const Vector<N>& x, Vector<N>& result) const;
    void F_x (const Vector<N>& x, DenseMatrix<N>& result) const;

  private:
    number_type a;
};
```

- Typedef:

  ```
  typedef std::size_t size_type;
  typedef N number_type;
  ```

  Bei den Typedefs am Anfang handelt es sich zwar nicht um Methoden, diese sind jedoch ebenso wichtig. Wir wissen von vornherein nicht, welcher Datentyp letztlich verwendet wird. Die Typedefs sind da, damit der Solver später erkennen kann, mit welchem Zahlentyp die Klasse eigentlich arbeitet.

- Konstruktor:

  ```
  WurzelProblem::WurzelProblem (number_type a_)
  : a(a_)
  {}
  ```

  Damit haben wir die Möglichkeit unterschiedliche Probleme der Form $x^2 = a$ zu lösen, indem wir dem Konstruktor das gewünschte a übergeben.

- Dimension:

  ```
  std::size_t Wurzelproblem::size () const
  {
    return 1;
  }
  ```

- Funktionswert: $f(x) = x^2 - a$:

```
void Wurzelproblem::F (const Vector<N>& x, Vector<N>& result) const
{
  result[0] = x[0]*x[0] - a;
}
```

  Wir benötigen diese spezielle Form, da wir nur Probleme der Form $f(x) = 0$ lösen können.

- Ableitung: $f'(x) = 2x$:

```
void Wurzelprobelem::F_x (const Vector<N>& x,
    DenseMatrix<N>& result) const
{
  result[0][0] = number_type(2.0)*x[0];
}
```

  (Die Ableitung muss manuell berechnet werden.)

Nachdem wir unsere Problemklasse erstellt haben, können wir nun ein Objekt dieser Klasse z.B. das Problem $x^2 = 5$ erstellen und dieses mit dem Newtonalgorithmus lösen. Dazu gehen wir wie folgt vor:

- Objekt der Klasse `WurzelProblem` mit dem Namen „problem“ erstellen, welches die Gleichung $x^2 = 5$ repräsentiert:

```
WurzelProblem<Number> problem(5.0);
```

Nun müssen wir ein Objekt der Klasse Newton erstellen und diverse Parameter setzen:

2.5.2. Die Klasse `Newton`

- Folgendermaßen kann man eine Instanz der Klasse `Newton` erstellen und alle Parameter setzen:

```
Newton newton;                      // Ein Newtonobjekt
newton.set_maxit(20);               // maximale Anzahl der Iteratioenen
newton.set_verbosity(2);            // Ausfuehrlichkeit der Ausgaben
newton.set_reduction(1e-100);       // Reduktionsfaktor
newton.set_abslimit(1e-100);        // maximaler absoluter Fehler
newton.set_linesearchsteps(3);      // Wie viele Schritte fuer Linesearch
```

- Schließlich benötigen wir noch einen Vektor u, in dem die Lösung gespeichert wird. Dieser muss die selbe Größe wie unser Problem haben:

```
Vector<Number> u(problem.size());
```

- Den Startwert für das Newton Verfahren setzten wir hier auf 17. Es kann natürlich auch ein anderer Wert gewählt werden, man muss allerdings beachten, dass der Startwert nicht zu weit von der Lösung entfernt ist, da das Newton-Verfahren nicht global konvergent ist.

```
u[0]=17.0;
```

- Jetzt können wir die Methode `solve` der Klasse Newton auf unser Problem anwenden:

```
newton.solve(problem,u);
```

- Wir bekommen als Lösung dieses speziellen Wurzelproblems das Ergebnis: $u = 2{,}2361e + 00$

Man kann solche Probleme nicht nur mit dem Newton-Verfahren lösen, wie bereits gesehen, sondern auch mittels der Klasse `Banach`.

2.5.3. Ausführliche Erläuterungen zur Klasse Newton

Die Klasse Newton besteht im Wesentlichen aus einer Methode `solve` welche zum Lösen von nichtlinearen Gleichungen benutzt werden kann. Neben dieser Methode gibt es noch einige Verfahrens-Parameter wie die maximale Iterationsanzahl, welche im Konstruktor gesetzt werden können.

Beim Lösen wird zuerst überprüft ob das Residuum $r = F(x)$ bereits kleiner oder gleich der Schranke `abslimit` ist. Ist dies der Fall, so ist der Startwert bereits gut genug und wir sind fertig. Im anderen Fall wird mittels der LR-Zerlegung die nächste Suchrichtung $\bigtriangledown f(x_k)^{-1} f(x_k) = z_k$ bestimmt. Mittels einer simplen Line-Search Methode wird ein geeignetes λ bestimmt und $x_{k+1} = x_k - \lambda z_k$ gesetzt.

```
for (size_type k=0; k<linesearchsteps; k++)
{
  y = x;
  y.update(-lambda,z);                          // y = x-lambda*z
  model.F(y,r);                                 // r = F(y)
  N newR(norm(r));                              // berechnet Norm
}
if (newR<(1.0-0.25*lambda)*R)                   // pruefe Konvergenz
{
  x = y;
  R = newR;
  break;
}
else lambda *= 0.5;                             // reduziert Daempgungsfaktor
if (R<=reduction*R0)                            // pruefe Konvergenz
{
  converged = true;
  return;
}
```

2.5.4. Die Klasse `Banach`

- Löst ein nichtlineares System der Form $F(x) = 0$ mittels Fixpunktiteration $x = x - \sigma * F(x)$
- Die wichtigste Funktion ist die Funktion `solve`, die die eigentliche Lösung durchführt.
- Bei diesem Verfahren macht man sich den banachschen Fixpunktsatz zunutze.
- Eine konkrete Implementierung, wie ein Problem mit der Klasse `Banach` gelöst wird ist nicht in dieser Dokumentation enthalten, da dies sehr ahnlich zur Lösung mit der Klasse `Newton` funktioniert. Ein Beispiel sieht man in der Datei `wurzelbanach.cc`. Der einzige Unterschied besteht im Verfahrensparameter sigma, den man bei Banach noch zusätzlich beachten muss.

2.5.5. Implementierung

```
class Banach
{
  typedef std::size_t size_type;
  public:
    Banach ()
      : maxit(25), linesearchsteps(10), verbosity(0),
        reduction(1e-14), abslimit(1e-30),  sigma(1.0), converged(false);
    void set_maxit (size_type n);
    void set_sigma (double sigma_);
    void set_linesearchsteps (size_type n);
    void set_verbosity (size_type n);
    void set_abslimit (double l);
    void set_reduction (double l);
    template<class M>
    void solve (const M& model, Vector<typename M::number_type> x) const;
    bool has_converged () const;

  private:
    size_type maxit;
    size_type linesearchsteps;
    size_type verbosity;
    double reduction;
    double abslimit;
    double sigma;
    mutable bool converged;
};
```

- Mit folgender Typdefinition spart man sich Schreibarbeit und es ist klarer, dass damit eine Größe gemeint ist.

  ```
  typedef std::size_t size_type;
  ```

- Im Konstruktor werden allen privaten Parametern der Klasse Werte zugewiesen,...

  ```
  Banach::Banach ()
    : maxit(25), linesearchsteps(10), verbosity(0),
    reduction(1e-14), abslimit(1e-30),  sigma(1.0), converged(false)
  {}
  ```

- ...die man dann mit den folgenden Funktionen nachträglich noch ändern kann. Der Parameter „maxit“ sorgt dafür, dass der später noch erläuterte Solver in keine Endlosschleife gerät, falls die Fixpunktiteration nicht konvergiert, sondern in diesem Fall abbricht und meldet, dass keine Konvergenz vorliegt.

  ```
  void Banach::set_maxit (size_type n)
  {
    maxit = n;
  }
  ```

- Hier legt man den Verfahrensparameter σ fest.

  ```
  void Banach::set_sigma (double sigma_)
  {
    sigma = sigma_;
  }
  ```

- Wie viele Schritte soll der Solver machen, bevor er abbricht? Die kann man hier festlegen.

```
void Banach::set_linesearchsteps (size_type n)
{
  linesearchsteps = n;
}
```

- Ausgabekontrolle: Je höher die gesetzte Zahl ist, desto genauere Informationen zur Konvergenz werden auf der Konsole ausgegeben. Was die einzelnen Zahlen bedeuten sollte man sich allerdings bei Bedarf im Quellcode ansehen.

```
void Banach::set_verbosity (size_type n)
{
  verbosity = n;
}
```

- Fehlertoleranz

```
void Banach::set_abslimit (double l)
{
  abslimit = l;
}
```

- Reduktionsfaktor

```
void Banach::set_reduction (double l)
{
  reduction = l;
}
```

- Mit der Methode `solve` kann dann ein gegebenes Model unter Rückgriff auf die 'private Members' gelöst werden.

```
template<class M>
void Banach::solve (const M& model, Vector<typename M::number_type> x) const
{
  typedef typename M::number_type N;
  Vector<N> r(model.size());              // Residuum
  Vector<N> y(model.size());              // temporaere Loesungen

  model.F(x,r);          // berechne das nichtlineare Residuum
  N R0(norm(r));         // Norm des Anfangsresiduums
  N R(R0);               // Norm des aktuellen Residuums

  converged = false;

  // maximal so viele Iterationen wie in Matrix festgelegt sind
  for (size_type i=1; i<=maxit; i++)
  {
    if (R<=abslimit)     //pruefe Absolutbetrag des Residuums
    {
      converged = true;
      return;
    }
```

Falls das vorläufige Ergebnis noch nicht genau genug war ($\leqslant abslimit$), geht es in die nächste Iteration, bei der zunächst der eigentliche Iterationsschritt ausgeführt wird und anschließend mittels Norm getestet wird, ob das Ergebnis nun genau genug ist und man anschließend wieder zum Beginn der for-Schleife springt. Ist das Ergebnis genau genug, hat die Funktion ihren Zweck erfüllt und wird beendet.

```
  // next iterate
  y = x;
  y.update(-sigma,r);              // y = x-sigma*z
  model.F(y,r);                    // r = F(y)
  N newR(norm(r));                 // Norm berechnen

  x = y;                           // Annahme der neuen Iterierten
  R = newR;                        // Normspeicherung

  // check convergence
  if (R<=reduction*R0 || R<=abslimit)
  {
    converged = true;
    return;
  }
}
```

- Der bool-Wert, den folgende Funktion zurück gibt, wird am Anfang immer auf false gesetzt. Löst die Funktion `solve` das Gleichungssystem erfolgreich, so setzt sie den Wert auf true und als private Member der Klasse bleibt dieser Wert dann auch erhalten. Somit sagt einem diese Funktion, ob das Fixpunktverfahren schon mal erfolgreich war und damit auch wieder erfolgreich sein wird.

```
bool Banach::has_converged () const
{
  return converged;
}
```

3. Gewöhnliche Differentialgleichungen

Im folgenden Kapitel soll es um das zentrale Thema der Vorlesung Numerik 1 gehen, das Lösen von gewöhnlichen Differentialgleichungen. Zur Wiederholung: das ist eine Gleichung bei der eine Funktion, sowie auch Ableitungen der Funktion vorkommen und man versucht herauszufinden, welche Funktion die Gleichung erfüllt. HDNUM stellt einige hilfreiche Werkzeuge zum Lösen von solchen Differentialgleichungen zur Verfügung. Es zeigt, wie eine Differentialgleichung aufzubereiten ist, damit sie ein Solver (wie die im einzelnen Funktionieren sei der Vorlesung und ihren Beweisen überlassen) lösen kann und beinhaltet zugleich mehrere solcher Solver. Fakt ist, dass sowohl Differentialgleichungen, als auch Solver in Klassen verpackt sind. Diese Klassen müssen bestimmte Methoden haben, damit sie untereinander kompatible sind. Fangen wir doch einmal mit einem Beispiel für eine Differentialgleichung an:

3.1. Das Paradebeispiel für eine DGL in HDNUM - `modelproblem.hh`

- Diese Datei beinhaltet lediglich die Klasse `ModelProblem`, welche genau die Methoden enthält, die für die Kompatibilität mit jedem Solver aus HDNUM nötig sind. Also muss jede Differentialgleiungsklasse genau diese Methodendeklarationen aufweißen!
- Die komplette Information über eine Differentialgleichung ist in der Implementierung der Methoden enthalten.
- Die Datei ist so geschrieben, dass Objekte der Klasse `ModelProblem` Modelprobleme im Sinne der Vorlesung sind, kann aber für jede beliebige Differentalgleichung umgeschrieben werden. Dabei ist zu beachten, dass alle Funktionsköpfe der Methoden nicht verändert werden. Nur so bleibt die neue DGL mit unseren Solvern kompatibel.

- Ein Objekt der Klasse Modelproblem entspricht dann einer zu lösenden Differentialgleichung.
- Ist die Datei im Header eingebunden, kann man im Programm Objekte der Klasse Modelproblem erstellen und mit dem Wissen der nächsten Abschnitte dann auch lösen.

```
template<class T, class N=T>
class ModelProblem
{
  public:
    typedef std::size_t size_type;
    typedef T time_type;
    typedef N number_type;

    ModelProblem (const N& lambda_)
      : lambda(lambda_);

    std::size_t size () const;
    void initialize (T& t0, hdnum::Vector<N>& x0) const;  //Anfangswerte
    void f (const T& t, const hdnum::Vector<N>& x,        //Funktion f
      hdnum::Vector<N>& result) const;
    void f_x (const T& t, const hdnum::Vector<N>& x,  //Jacobi Matrix von f
       hdnum::DenseMatrix<N>& result) const;

  private:
    N lambda;
};
```

- Bei den Typedefs am Anfang handelt es sich zwar nicht um Methoden, diese sind jedoch auch eine kurze Erklärung wert. Man sieht daran gut, dass es sich um eine Template-Klasse handelt und nie von vornherein klar ist, welcher Datentyp dann eigentlich verwendet wird. Die Typedefs sind da, damit der Solver später erkennen kann, mit welchem Zahlentyp die Modellklasse eigentlich arbeitet. Wir verwenden sie, damit uns die Möglichkeit bleibt, mit sehr genauen Datentypen (multiple precision) zu arbeiten.
- Der Konstruktor initialisiert falls benötigt private Parameter. Solche muss es aber nicht immer geben.

  ```
  template <class T, class N=T>
  ModelProblem::ModelProblem (const N& lambda_)
    : lambda(lambda_)
  {}
  ```

- Mit dieser Funktion legt man fest, welche Dimension die zu lösende Differentialgleichung hat.

  ```
  template <class T, class N=T>
  std::size_t ModelProblem::size () const
  {
    return 1;
  }
  ```

- Hier legt man die Anfangswerte fest. t_0 ist der zeitliche Anfangswert, während x_0 der Vektor der Anfangswerte ist. Im eindimensionalen enthält er also nur einen Eintrag.

```
template <class T, class N=T>
void ModelProblem::initialize (T& t0, hdnum::Vector<N>& x0) const
{
  t0 = 0;
  x0[0] = 1.0;
}
```

- Die Funktion f beinhaltet die eigentliche Differentialgleichung. Dabei wird der Vektor `result`, also die Lösung der Funktion `f` zum Zeitpunkt `t` berechnet.

```
template <class T, class N=T>
void ModelProblem::f (const T& t, const hdnum::Vector<N>& x,
  hdnum::Vector<N>& result) const
{
  result[0] = lambda*x[0];
}
```

- Diese Funktion stellt die Jacobi-Matrix der Funktion `f` in `result` zur Verfügung. Diese wird von impliziten Solvern benötigt.

```
template <class T, class N=T>
void ModelProblem::f_x (const T& t, const hdnum::Vector<N>& x,
  hdnum::DenseMatrix<N>& result) const
{
  result[0] = lambda;
}
```

- Im privaten Teil der Klasse stehen eventuell benötigte Parameter.

3.2. Anwendungsbeispiel für `modelproblem.hh`

Die Datei `modelproblem_high_dim.hh` ist eine Umformulierung der Datei `modelproblem.hh` und stellt die Differentialgleichung $u'(t) = \begin{pmatrix} 5 & -2 \\ -2 & 5 \end{pmatrix} * u(t)$ mit Anfangswert $u(t) = \begin{pmatrix} 1 \\ 3 \end{pmatrix}$ da. Damit ist sie ein Beispiel für die Darstellung einer mehrdimensionalen Differentialgleichung.

3.3. Der Solver löst die DGL - `modelproblem.cc`

- Diese Datei ist ein Musterbeispiel zum Lösen von gewöhnlichen Differentialgleichungen.
- Sie zeigt, wie man Differentialgleichungsklasse und Solverklasse so kombiniert, dass die Differentialgleichung gelöst und das Ergebnis derart in eine Datei geschrieben wird, dass man es plotten kann.

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

Im Header wird neben Bibliotheken auch das Modelproblem, sowie eine Datei zur Lösung der Differentialgleichung eingebunden. In diesem Fall soll die Differentialgleichung mit dem expliziten Euler gelöst werden.

```
int main ()
{
  typedef double Number;                  // Definiert Zahlentyp
  typedef ModelProblem<Number> Model;     // Definiert Modeltyp
  Model model(-1.0);                      // Objekt der Klasse mit lambda=-1
  typedef ExplicitEuler<Model> Solver;    // Waehle einen Solver
  Solver solver(model);                   // initialisiere Solver mit Model
  solver.set_dt(0.02);                    // Setze Zeitabstaende
  hdnum::Vector<Number> times;            // Vektor fuer Zeitabstaende
  hdnum::Vector<hdnum::Vector<Number>> states; // Loesungsvektor
  times.push_back(solver.get_time());     // Anfangszeit in Vektor speichern
  states.push_back(solver.get_state()); // Anfangswert in Vektor speichern
  while (solver.get_time()<5.0-1e-6)      // Schleife zum Loesen
    {
      solver.step();
      times.push_back(solver.get_time());     // Zeit speichern
      states.push_back(solver.get_state());  // Wert speichern
    }

  gnuplot("mp2-ee-0.02.dat",times,states);    // Ausgabe wird im Abschnitt
                                              // ueber Gnuplot erklaert
  return 0;
}
```

Ein Alternativbeispiel ist die Datei `modelproblem_high_dim.cc`. Indem man beim Solver EE durch andere Solver aus `ode.hh` (Erklärung siehe unten) ersetzt, kann man die DGL mit verschiedenen Mitteln lösen und sieht dabei gleich ein Beispiel, dass eine DGL mit allen unseren Solvern kompatibel ist.

3.4. Was muss ein Solver können? - `expliciteuler.hh`

- Diese Datei enthält die Klasse `ExplicitEuler`.
- In der Klasse gibt es alle Methoden, die ein Solver in unserem Kontext braucht.
- Alle Solver haben mindestens die Methoden, die `ExplicitEuler` hat, eventuell noch ein paar mehr.
- Mit Hilfe dieser Datei kann man alle Differentialgleichungen lösen, die die bereits erwähnte Darstellung in einer Klasse besitzen.

```
template<class M>
class ExplicitEuler
{
  public:
    typedef typename M::size_type size_type;
    typedef typename M::time_type time_type;
    typedef typename M::number_type number_type;

    ExplicitEuler (const M& model_)
      : model(model_), u(model.size()), f(model.size());
    void set_dt (time_type dt_);
    void step ();
    const hdnum::Vector<number_type>& get_state () const;
    time_type get_time () const;
    time_type get_dt () const;

  private:  //Die private Member sind bei jedem Solver ähnlich.
    const M& model;      //Referenz auf das Model ist IMMER vorhanden
    time_type t, dt;     //Zeitlichen Variablen
```

```
    hdnum::Vector<number_type> u;  //Vektor zur Speicherung von Zeitschritten
    hdnum::Vector<number_type> f;  //mindestens einem Vektor
                  // zur Speicherung von Loesungen
};
```

- Zuerst noch eine kurze Bemerkung zu den Typedefs am Anfang: Der Solver hat zuanächst im Konstruktor nur eine Referenz auf ein Model bekommen. Damit ist aber noch nicht klar, mit welchen Zahlentypen im Model gearbeitet wird und ob man die Funktionen davon aufrufen kann. Um dies festzusetzen dienen die Typedefs. Somit kann der Solver DGLs für beliebige Zahlentypen lösen und erst beim Kompilieren wird festgelegt, welcher eigentlich gemeint ist.
- Der Konstruktor speichert eine Referenz zu dem Model, das er lösen soll. Außerdem werden hier Parameter für den Lösungsalgorithmus wie die Größe der Zeitschritte, Anfangswerte, oder ähnliches festgelegt.

```
template<class M>
ExplicitEuler::ExplicitEuler (const M& model_)
  : model(model_), u(model.size()), f(model.size())
{
  model.initialize(t,u);
  dt = 0.1;
}
```

- Da Solver die Lösung (Zeit-)Schritt für (Zeit-)Schritt berechnen, kann man festlegen, wie groß diese Schritte sein sollen. Je größer die Schritte, desto ungenauer das Ergebnis, aber desto geringer der Rechenaufwand.

```
template<class M>
void ExplicitEuler::set_dt (time_type dt_)
{
  dt = dt_;
}
```

- Der eigentliche Lösungsalgorithmus steht in der Funktion step. Sie entscheidet, wie man vom einem zum nächsten Schritt gelangt. Hier steht also der Algorithmus des expliziten Eulers.

```
template<class M>
void ExplicitEuler::step ()
{
  model.f(t,u,f);   // berechnet Wert von f an der Stelle t
  u.update(dt,f);   // naechster Funktionswert ist alter Wert+dt*f(t)
  t += dt;          // die Zeit wird um dt nach vorne gesetzt
}
```

- Der bisher errechnete Lösungsvektor:

```
template<class M>
const hdnum::Vector<number_type>& ExplicitEuler::get_state () const
{
  return u;
}
```

- Der Zeitpunkt, der gerade berechnet wurde:

```
template<class M>
time_type ExplicitEuler::get_time () const
{
  return t;
}
```

- Das aktuelle dt (Schrittweite):

```
template<class M>
time_type ExplicitEuler::get_dt () const
{
  return dt;
}
```

3.5. Einschub: Gnuplot in `ode.hh`

Ein numerischer Solver kann uns natürlich keine analytische Lösung einer DGL in Form einer konkreten Funktion liefern. Er kann uns aber sagen, wie die Lösungsfunktion an ganz vielen Punkten aussieht. Damit wir mit diesen vielen Zahltupeln etwas anfangen können, visualisieren wir sie mit Gnuplot. Folgende Template-Funktionen machen uns dies sehr leicht und schreiben das Ergebnis im richtigen Format in eine Datei, sodass wir es dann direkt plotten können.

1. ```
   void gnuplot (const std::string& fname, const std::vector<T> t,
    const std::vector<Vector<N> > u)
   ```

   Nur für eindimensionale DGL geeignet! Man übergibt der Funktion einen Dateinamen (.dat) in Anführungsstrichen, sowie Zeit und Lösungsvektor. Die Funktion sorgt dafür, dass die Daten in einer Art Tabelle in einer Datei mit dem gewünschten Namen stehen. Diese Datei kann man dann plotten.

2. ```
   void gnuplot (const std::string& fname, const std::vector<T> t,
    const std::vector<Vector<N> > u, const std::vector<T> dt)
   ```

   Für zweidimensionale DGL geeignet! Man übergibt der Funktion die gleichen Daten wie oben und zusätzlich noch den zweiten Lösungsvektor. Das Ergebnis ist ebenfalls analog. Man beachte beim plotten dann allerdings die Eigenheiten der Mehrdimensionalität.

Als Beispielvorlage kann der Code im vorhergehenden Abschnitt am Ende gesehen werden.

Die wichtigsten Gnuplotbefehle im Terminal:

1. `gnuplot` - öffnet Gnuplot
2. `plot 'dateiname.dat'using 1:2` - plottet im zweidimensionalen unter Verwendung der Zeilen eins und zwei
3. `plot 'dateiname.dat'using 1:2, 'dateiname.dat'using 1:3` - plottet im zweidimensionalen zwei Graphen
4. `splot 'dateiname.dat'using 1:2:3` - plottet im dreidimensionalen
5. `exit` - beendet gnuplot

3.6. Einschrittverfahren - `ode.hh`

Nachdem wir uns jetzt angeschaut haben, wie genau eine Differentialgleichung und ein Solver in eine Klasse verpackt werden müssen, damit sie untereinander kompatibel sind und wie man mit dem Solver dann die Differentialgleichung löst, können wir dazu übergehen uns mehrere solcher Solver anzuschauen. In der Vorlesung lernt man dazu die impliziten

und expliziten Runge-Kutta Verfahren als wichtigste Beispiele kennen. Der explizite Euler den wir zuvor schon als Beispiel hatten gehört auch dazu. In der Datei `ode.hh` sind mehrerer solche Solver implementiert. Damit man eine beliebige Differentialgleichung (natürlich wieder in einer Klasse verpackt) mit jedem Solver lösen kann, haben diese Solverklassen alle Methoden mit den jeweils gleichen Funktionsköpfen. Lediglich in der Art wie diese Funktionen dann implementiert sind unterscheiden sie sich, was dann das einzelne Verfahren ausmacht. Zusätzlich zu den Methoden der zuvor behandelten Klasse `ExplicitEuler` haben die Klassen in `ode.hh` noch einige zusätzliche Funktionen. Die Verfahren mit Schrittweitensteuerung sind ebenfalls leicht abgewandelt.

3.6.1. Die Verfahren in ode.hh

- Explizite Runge-Kutta Verfahren
  - `EE` - expliziter Euler
  - `ModifiedEuler`
  - `Heun2`
  - `Heun3`
  - `Kutta3`
  - `RungeKutta4`
- Implizite Runge-Kutta Verfahren
  - `IE` - impliziter Euler
  - `DIRK` - Diagonal implizites Verfahren
- Schrittweitensteuerung
  - `RKF45`
  - `RE` - Richardsonextrapolation

3.7. Das allgemeine Runge-Kutta-Verfahren - `RungeKutta`

Diese Klasse ist dazu gebaut, um eine Differentialgleichung mit einem beliebigen expliziten oder impliziten Runge-Kutta-Verfahren zu Lösen. Die Differentialgleichung muss dabei auf die gleiche Weise wie bisher in einer Klasse implementiert sein.

3.7.1. Bedienung der Klasse `RungeKutta`

Der einzige Unterschied zur Handhabung einer anderen Solverklasse besteht darin, dass dem Konstruktor zusätzlich noch das Butcher-Tableau des gewünschten Verfahrens übergeben werden muss. Der Funktionskopf im Namespace hdnum sieht folgendermaßen aus:

```
M& model, DenseMatrix<number_type> A_, Vector<number_type> b_,
Vector<number_type> c_)
```

Die Matrix `A_` und die Vektoren `b_` und `c_` kommen direkt aus dem Butcher Tableau. Alles Weitere ist dann analog zu den anderen Solverklassen. N ist ein Templateparameter der Klasse. Möchte man statt dem Newtonverfahren das Banachverfahren zur Lösung von nichtlinearen Gleichungssystemen verwenden, so ist Banach ein zweiter Templateparameter den man beim Erzeugen eines Objektes davor schreibt. In diesem Fall macht es auch Sinn dem Konstruktor als weiteres Argument am Schluss noch einen *number_type sigma_* zu übergeben. Der entsprechende Konstruktor ist implementiert. Das könnte dann folgendermaßen aussehen:

```
Solver(model, A, b, c, 0.5)
```

Dabei müsste dann model ein Modelproblem vom Typ Model sein und A eine $n \times n$ Matrix, sowie b und c n-dimensionale Vektoren. Das Sigma im Banachverfahren wäre in diesem Fall dann 0,5.

Die Algorithmen hinter Funktion `void step`

- Die Funktion `step` unterscheidet von Anfang an, ob es sich um ein explizites oder implizites Verfahren handelt. (Die Testfunktion erkennt dies am Butchertableau).
- Im expliziten Fall sind alle Werte bekannt und in privaten Variablen gespeichert, um $u_n^h = u_{n-1}^h + h_n(b_1k_1 + ... + b_sk_s)$ mittels $k_1 = f(t_{n-1}, u_{n-1}^h)$, $k_i = f(t_{n-1} + c_ih_n, u_{n-1}^h + h_n \sum_{j=1}^{i-1} a_{ij}k_j)$ zu berechnen. Dabei wird die Funktion f von der Problemklasse bereitgestellt
- Im impliziten Fall gilt es $k_i = f(t_{n-1} + c_ih_n, u_{n-i}^h + h_n \sum_{j=1}^{s} a_{ij}k_j)$ für $i = 1, ..., s$ zu lösen und damit $u_n^h = u_{n-1}^h + h_n \sum_{i=1}^{s} b_ik_i$ zu bestimmen. Numerisch ist es jedoch einfacher, zunächst $z_i := h_n \sum_{j=1}^{s} a_{ij}k_j$ für $i = 1, ..., s$ zu berechnen und dann die k_i über $K = h_n^{-1}A^{-1}Z$ zu bestimmen. Dabei sind K und Z Vektoren aus Vektoren.

  Falls b^T gleich der letzten Zeile von A ist, kann man sich die Berechnung der k_i sparen und direkt $u_n^h = u_{n-1} + z_s$ berechnen. Die nichtlinearen Gleichungssysteme bei der Berechnung der z_i werden wahlweise mit dem Banach- oder Newtonverfahren gelöst, für die eine Problemklasse erstellt wird.
- Sowohl für das Banach- als auch für das Newtonverfahren benötigt man eine bestimmte Problemklasse, die das zu lösende Problem modelliert. In unserem Fall erfüllt diesen Zweck die Klasse `ImplicitRungeKuttaStepProblem`. Diese wird im Konstruktor mit allen wichtigen Größen der Klasse `RungeKutta` initialisiert. Wichtig zu wissen ist jedoch, dass Banach- und Newtonverfahren keine Nullstellen von Funktionen, die Vektoren von Vektoren als Argument haben, berechnen können. Deshalb muss man Z als einen Vektor der Größe $n * s$ auffassen und erst danach wieder auf s Vektoren der Größe n zurückrechnen.
- Das Herzstück der Klasse `ImplicitRungeKuttaStepProblem` sind die Funktionen `void F` und `void F_x`. In der ersten wird die Funktion modelliert, die annuliert wird, wenn die richtigen z_i getroffen sind, während die zweite Funktion nur im Newtonverfahren benötigt wird und die Jacobimatrix der ersten Funktion bereitstellt.
- Die Funktion `F` sieht dabei folgendermaßen aus:

$$F : \mathbb{R}^{n*s} \to \mathbb{R}^{n*s}, \begin{pmatrix} z_1 \\ ... \\ ... \\ z_s \end{pmatrix} \mapsto \begin{pmatrix} F_1(z_1, ..., z_s) \\ ... \\ ... \\ F_s(z_1, ..., z_s) \end{pmatrix}$$

  wobei $F_i(z_1, ..., z_s) = z_i - h_n \sum_{j=1}^{s} a_{ij} f(t_{n-1} + c_jh_n, u_{n-1} + z_j)$ für $i = 1, ..., s$.
- Die zu berechnende Jacobimatrix ist eine Blockmatrix aus $s \times s$ Blöcken der Größe

$n \times n$. Dabei gilt für den (i,j)-ten Block:

$$J_{ij} = \frac{\partial F_i}{\partial z_j}(z_1, ..., z_s) = \frac{\partial}{\partial z_j}(z_i - h_n \sum_{k=1}^{s} a_{ik} f(t_{n-1} + c_k h_n, u_{n-1} + z_k)) \quad (1)$$

$$= \delta_{ij} I - h_n \sum_{k=1}^{s} a_{ik} \frac{\partial}{\partial z_j} f(t_{n-1} + c_k h_n, u_{n-1} + z_k) \quad (2)$$

$$= \delta_{ij} I - h_n a_{ij} \frac{\partial f}{\partial z_j}(t_{n-1} + c_j h_n, u_{n-1} + z_j) \quad (3)$$

$\frac{\partial f}{\partial z_j}$ erhalten wir dabei aus der Funktion `f_x` der Differentialgleichungsklasse.

3.7.2. Konsistenzordnungstests mit `void ordertest`

Mit dieser Funktion kann man die Konsistenzordnung eines allgemeinen Runge-Kutta-Verfahrens, dessen Butchertableau man kennt, bestimmen. Dazu ist es jedoch nötig, in der Klasse der Differentialgleichung die exakte Lösung in der eine Funktion u anzugeben. Ein Beispiel dazu findet man in der Datei `modelproblem.hh`. Der Funktionskopf von `ordertest` sieht folgendermaßen aus:

```
template<class M, class S> void ordertest(const M&
model, S solver, Number T, Number h_0, int L)
```

Dabei beschreibt `model` eine gewöhnliche Differentialgleichung, `solver` ist ein Löser und T der Zeitpunkt, der für den Konsistenzordnungstest verwendet werdern soll. h_0 ist die initiale Schrittweite und L die Anzahl, wie oft h_0 bei der Berechnung halbiert werden soll. Auf der Konsole wird dann in der i-ten Zeile der Fehler im i-ten Schritt, sowie die damit berechnete Konsistenzordnung ausgegeben. Ein kurzes Anwendungsbeispiel gibt es in der Datei `model_ordertest.cc`.

Berechnung der Konsistenzordnung

- Für die Konsistenzordnung α gilt: $||u - u_h|| = Ch^\alpha$
- $E_{n_1,n_2} = \frac{||u(T)-u_{h_1}(T)||}{||u(T)-u_{h_2}(T)||} = \frac{Ch_1^\alpha}{Ch_2^\alpha} = \left(\frac{h_1}{h_2}\right)^\alpha$, wobei $h_i = \frac{h_0}{2^i}$ gewählt wird.
- $\alpha = \frac{logE_{n_1,n_2}}{log\left(\frac{h_1}{h_2}\right)}$
- Im Fall, dass T nicht direkt von einem Zeitschritt getroffen wird, also $u_{h_i}(T)$ nicht direkt berechnet wird, muss man den Berechnungsalgorithmus anpassen. Dabei unterscheidet man mehrere Fälle. Wird T fast getroffen (Abstand kleiner als vorgegebenes ϵ), so nimmt man diesen Wert, das heißt man vergrößert den letzten Schritt um maximal ϵ, sodass man T genau trifft. Andernfalls verändert man die Schrittweite der letzten ein oder zwei Schritte um T genau zu treffen.

3.8. Anwendungsbeispiele

Im Ordner `examples/num1` sind einige interessante Anwendungsbeispiele gegeben, bei denen man sehen kann, wie die Verfahren aus der Vorlesung in anderen Naturwissenschaften verwendet werden.

3.8.1. Hodgkin-Huxley-Modell

Das Hodgkin-Huxley-Modell kommt aus der Neurobiologie und beschreibt die Vorgänge an der Zellmembran einer Nevenzelle bei der Reizweiterleitung. Für genauere Erklärungen siehe `https://de.wikipedia.org/wiki/Hodgkin-Huxley-Modell`.

3.8.2. n-body Problem

Das n-body Problem ist ein Problem der Astrophysik, bei dem es um die Bewegungen von Himmelskörpern geht. Für genauere Erklärungen siehe `https://en.wikipedia.org/wiki/N-body_problem`.

3.9. Van der Pol Oszillator

Dabei handelt es sich um ein Schwingungsbeispiel, dass in unserem Fall ein gutes Beispiel für eine steife Differentialgleichung ist. Genaueres dazu gibt es unter `https://de.wikipedia.org/wiki/Van-der-Pol-System` bei Wikipedia.

Apéndice A Kleiner Programmierkurs

Apéndice B Unix Kommandos

In der folgenden Tabelle sind die wichtigsten Kommandos fürs Terminal (das schwarze Fenster) zusammengestellt. Alle Worte in Großbuchstaben sind Platzhalter.

Kommando	Auswirkungen
cd	gehe ins home-Verzeichnis
cd ORDNERNAME	gehe in einen Ordner, dieser muss im Ordner enthalten sein, in dem man sich gerade befindet
cd ..	gehe einen Ordner höher
ls	zeigt an, was sich in dem Ordner befindet, in dem man gerade ist
tar cvf GEWÜNSCHTERNAME.tar Inhalt1.cc Inhalt2.cc ... Inhaltn.cc	erstellt ein Tar-Archiv
tar xvf TARNAME.tar	entpackt das Tar
g++ -std=c++11 -o DATEINAME DATEINAME.cc	kompilieren (-std=c++11 braucht nicht jeder)
./DATEI	Ausführen der Datei

Referencias