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

16 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);                // Definiendo la cantidad de digitos
w.precision(16);            // Numero de posiciones decimales
std::cout << w << std::endl;// Imprimiendo w
std::cout <<cx << std::endl;// Imprimiendo complejos
std::cout <<mx << std::endl;// Funciona para mpf_class
```

Visualización

```
[   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
```

Funciones auxiliares

```
float d = norm(w);          // Norma Euclidea
d = w.two_norm();           // La misma norma
zero(w);                    // Igual que w=0.0
fill(w,(float)1.0);         // Igual que 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);            // Vector unitario cartesiano
gnuplot("test.dat",w);      // Salida a gnuplo: i w[i]
gnuplot("test2.dat",w,z);   // Saida a gnuplot: w[i] z[i]
```

Funciones

- Ejemplo: Suma de todas las componentes

```
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;
}
```

- Versión mejorada de la función con **template** y uso por referencia

```
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;
}
```

- El uso por referencia es mejor para objetos grandes.

2.2. Matrices

`hdnum::DenseMatrix<T>`

- `hdnum::DenseMatrix<T>` Es una plantilla o Template de clase.
- Convierte un elemento tipo `T` en una matriz.
- También es posible incluir complejos y su precisión.
- Las matrices son como las conocemos en matemáticas:
  - Dimensión de $m \times n$.
  - Inicia en elemento 0 hasta $m-1$ o, hasta $n-1$ numerados consecutivamente.
  - El conjunto de las matrices $m \times n$ constituyen un espacio vectorial.
  - Multiplicación de vectores y de matrices.
- Los siguientes ejemplos se pueden encontrar en el archivo `matrizen.cc`

Construcción y acceso

- Construcción con y sin inicialización

```
hdnum::DenseMatrix<float> B(10,10);      // 10x10 Matrix sin inicializar
hdnum::DenseMatrix<float> C(10,10,0.0); // 10x10 Matrix inicializada
```

- Acceso a elementos

```
for (int i=0; i<B.rowsize(); ++i)
  for (int j=0; j<B.colsize(); ++j)
    B[i][j] = 0.0;           // Ahora la matriz B áest inicializada
```

- El objeto matriz se elimina al final del ciclo.

Copia y asignación

- Constructor copia: Crea una matriz como copia de otra

```
hdnum::DenseMatrix<float> D(B); // D copia de B
```

- Asignación y tamaño de las copias

```
hdnum::DenseMatrix<float> A(B.rowsize(),B.colsize()); //Tamano correcto
A = B;                      // Copia
```

- Extractos de matrices (Submatrices)

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

Calculando con matrices

- Operaciones de espacio vectorial: Ojo, las matrices deben tener el tamaño correcto!)

```
A += B;            // A = A+B
A -= B;            // A = A-B
A *= 1.23;         // óMultiplicacin por escalar
A /= 1.23;         // óDivisin por escalar
A.update(1.23,B); // A = A + s*B
```

- Matrices, Vectores y multiplicación de matrices

```
hdnum::Vector<float> x(10,1.0); // Construimos dos vectores
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);                 // Hace x la primera columna de A
A.sr(x,1);                 // Hace x la primera fila de A
float d=A.norm_infty();    // Halla la norma de la suma de las filas
d=A.norm_1();              // Halla la norma de la suma de las columnas
```

Funciones de salida auxiliares

- Salida de matrices

```
std::cout << A.sub(0,0,3,3) << std::endl;// Impresion
A.iwidth(2);                // Digitos en la salida del indice
A.width(10);                // Numero total de digitos
A.precision(4);             // Posiciones decimales
std::cout << A << std::endl;// Nueva publicacion
```

- Algunas funciones auxiliares

```
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

Ejemplo de una función de una matriz A y de un vector b inicializados.

```
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;
    }
}
```

En la siguiente sección se tratan los solucionadores de sistemas de ecuaciones. Si el sistema de ecuaciones es lineal, se utiliza el método de descomposición LR o QR. En el caso no lineal, como por ejemplo en el archivo num1, que utiliza iteraciones de punto fijo, se utiliza el método de Newton.

2.3. Método LR

2.3.1. Breve explicación del algoritmo

El método de descomposición LR se utiliza para resolver un sistema de ecuaciones de la forma $Ax = b$. Se busca descomponer la matriz de coeficientes A que es cuadrada en dos matrices, una matriz L matriz triangular inferior y otra matriz R matriz triangular superior, de tal forma que el producto sea $A = LR$. Cuando es necesario realizar intercambio de filas ésto conduce a un sistema de la forma $PA = LR$. Los pivotes aseguran que los elementos de la Diagonal no son cero, de lo contrario no se puede usar el algorítmo.

Por un lado, se hace una distinción entre pivotamiento parcial, que asegura que el elemento más grande de la columna debajo de la diagonal en términos de cantidad se intercambie en la diagonal mediante permutaciones de fila. Con la rotación total, uno mira la matriz completa debajo de la diagonal y busca el elemento más grande en términos de cantidad para cambiarlo a la entrada de la diagonal actual usando operaciones de fila y columna. El elemento más grande en términos de valor absoluto se elige para reducir los errores numéricos.

2.3.2. Como hacer la descomposición LR

Para hacer un programa que resuelva un sistema $Ax = b$ utilizando la descomposición LR, se procede de la siguiente forma:

- Se crea el vector b y la matriz A. veamos el siguiente ejemplo:

```
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;
```

- Además, necesitamos los vectores x y p. Si se realiza un pivoteo total, se debe crear otro vector q. Para mejorar la condición de la matriz, las funciones `row_equilibrate` y `apply_equilibrate` se pueden necesitar para lo cual se requiere un vector adicional s:

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

- Como ya se mencionó en el punto anterior, el condicionamiento de la matriz A se puede mejorar desde el principio. Esto se hace usando las funciones `row_equilibrate` y `apply_equilibrate`. La aplicación de los comandos se puede ver en los siguientes ejemplos.
- Ahora se aplica una de las siguientes funciones a la matriz A y al vector de permutación p creado previamente. En nuestro ejemplo realizamos un pivoteo total, por lo que necesitamos el vector adicional q:

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

  La función `lr_partialpivot` se utilizará para el pivoteo parcial y la función `lr` se utilizará para la descomposición LR sin pivotar. (El vector de permutación adicional q no es necesario aquí.) Ahora podemos usar el sistema de ecuaciones para resolver distintos lados derechos.

- Para esto necesitamos preparar el lado derecho de la siguiente forma:

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

- Luego llamamos a la función `solveL`, que recibe la matriz A, el vector de la derecha por un vector y como parámetros en los que se almacena la solución del sistema de ecuaciones $Ly = b$. Para ahorrar espacio de almacenamiento se puede escribir el resultado en el vector ya existente b.
- Finalmente, se requiere la función `solveR`, que resuelve el sistema de ecuaciones $Rx = y$. La función necesita la matriz A, el vector de la derecha y (del sistema de ecuaciones $Ly = b$) así como el vector x, en el que se guarda el resultado final:

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

- Si se ha realizado un pivoteo total, las permutaciones que se almacenaron en el vector q (transformaciones de columna de A) deben aplicarse al resultado x usando `permute_backward`:

- La solución del sistema lineal de ecuaciones ahora se almacena en el vector `x`. En nuestro caso:

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

2.4. Explicaciones detalladas del algoritmo `lr.hh`

- **La función `lr`**: Al principio se comprueba para todas las funciones si existe una matriz cuadrada no vacía y si el vector p es compatible con la matriz dada. El primer bucle for busca una fila de la matriz cuyo elemento diagonal no sea igual a cero. A continuación, esta línea se compara con la de la corriente Todos los elementos diagonales intercambiados. Las permutaciones que resultan de la búsqueda pivote se almacenan en el vector p.

```
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];
    }
}
```

En el segundo ciclo *for*, se crea la matriz triangular con la matriz A permutada.

- **La función** `lr_partialpivot`: Parametros : Matriz A y vector de permutación p. Ésta función realiza un pivoteo parcial. Procede de la siguiente manera: Primero, el vector p se inicializa describiéndolo con los valores de 0 a $n-1$ (Donde $A \in \mathbb{R}^{n\times n}$). Luego, el elemento pivote (el elemento más grande en términos de magnitud) se busca en la columna actual debajo del elemento diagonal en la matriz A y la permutación requjerida para cambiarlo a la diagonal que se almacena en el vector p:

```
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
}
```

En el siguiente ciclo, las líneas k y j se intercambian de modo que el elemento pivote se encuentre en la diagonal.

- **La función** `lr_fullpivot`: Funciona de manera similar a la función anterior `lr_partialpivot`, pero necesita un vector adicional q para poder realizar un pivoteo total. No solo es posible intercambiar filas, sino también intercambiar columnas, que se almacenan en el vector q.

- **La función** `permute_forward`: El vector p ha almacenado las permutaciones necesarias. En esta función, las permutaciones de línea se transfieren al vector:

```
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;
  }
```

- **La función** `permute_backward`: Ésta función se utiliza al final del algorítmo LR para deshacer las permutaciones realizadas en la función `permute_forward` para el vector de la derecha.

- **La función** `row_equilibrate`: Esta función se utiliza antes del algoritmo actual para mejorar la condición de la matriz de (equilibrio). Los valores por los que se dividen las filas de la matriz se almacenan en el vector s:

```
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];
}
```

- **La función** `apply_equilibrate`: Los cambios que se hicieron a la matriz A también se aplican aquí al vector b para obtener la solución correcta.

- **La función** `solveL`: Paramétros : Vector x y el vector b. Ésta función resuelve la ecuación $Lx = b$. Aquí x se determina iterativamente de la siguiente forma: $x_i = b_i - \sum_{j=0}^{i-1} l_{ij} x_j$

```
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;
}
```

- **La función** `solveR`: Ésta función resuelve la ecuación $Rx = b$. Por lo tanto el vector x se determina de la siguiente manera: $x_i = b_i - \sum_{j=i+1}^{n-1} r_{ij} x_j$ (aquí está $R \in \mathbb{R}^{n \times n}$)

```
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. Método de iteración - archivo `newton.hh`

Ahora sabemos cómo resolver sistemas lineales de ecuaciones de la forma $Ax = b$. Pero, ¿qué se debe hacer si el sistema de ecuaciones no es lineal, por ejemplo, en el caso simple y unidimensional $x^2 = a$? En la lección aprenderá procedimientos que utilizan iteraciones de punto fijo para acercarse mucho a la solución. El archivo `newton.hh` proporciona herramientas útiles para resolver este tipo de ecuaciones y sistemas de ecuaciones. El más importante es el método de Newton, con el que se pueden resolver ecuaciones no lineales de la forma $F(x) = 0$. Pero primero consideramos la formulación concreta de un problema en una clase.

2.5.1. La clase `SquareRootProblem`

Para poder resolver un sistema no lineal de ecuaciones de la forma $f(x) = 0$, primero debemos crear una clase para nuestro problema. Además de un constructor adecuado e información sobre la dimensión del problema, esto requiere un método que proporcione el valor de la función y otro que proporcione la derivada de la función. Mostramos esto con el ejemplo de la clase 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;
  ```

  Las definiciones de tipo al principio no son métodos, pero son igualmente importantes. No sabemos de antemano qué tipo de datos se utilizará en última instancia. Las definiciones de tipo están ahí para que el solucionador pueda reconocer más tarde con qué tipo de número está trabajando realmente la clase.

- Constructor:

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

  Esto nos da la oportunidad de resolver varios problemas de la forma $x^2 = a$ pasando la a deseada al constructor.

- Dimensión:

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

- Valor de la función: $f(x) = x^2 - a$:

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

  Necesitamos esta forma especial porque solo podemos resolver problemas de la forma $f(x) = 0$.

- Derivación: $f'(x) = 2x$:

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

(La derivada debe calcularse manualmente.)

Ahora que hemos creado nuestra clase, debemos crear el objeto z.B del problema $x^2 = 5$, crear y resolver esto con el método de Newton. Para ello, procedemos de la siguiente manera:

- Crear el objeto del problema con `WurzelProblem` con el nombre "problem", que representa la ecuación $x^2 = 5$:

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

Ahora tenemos que crear un objeto de la clase Newton y establecer varios parámetros:

2.5.2. Clase `Newton`

- De la siguiente manera, puede crear una instancia de la clase `Newton` y establecer todos los parámetros:

```
Newton newton;                      // Crea un objeto newton
newton.set_maxit(20);               // Numero maximo de iteraciones
newton.set_verbosity(2);            // Detalle del gasto
newton.set_reduction(1e-100);       // Factor de reduccion
newton.set_abslimit(1e-100);        // Maximo error en valor absoluto
newton.set_linesearchsteps(3);      // Numero maximo de pasos para la busqueda
```

- Finalmente necesitamos un vector u, en el que se almacena la solución. Éste debe teber el mismo tamaño que nuestro problema:

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

- Aquí establecemos el valor inicial para el método de Newton en 17 Por supuesto, se puede seleccionar otro valor, pero debe asegurarse de que el valor inicial no esté demasiado lejos de la solución, ya que el método de Newton no es globalmente convergente.

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

- Ahora podemos aplicar el método lstinlinesolve de la clase de `Newton` a nuestro problema:

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

- Como solución a este problema raíz en particular, obtenemos el resultado:
  $u = 2{,}2361e + 00$

Se pueden resolver estos problemas no solo con el método de Newton, como ya se ha visto, sino también con la ayuda de la clase `Banach`.

2.5.3. Explicaciones detalladas de la clase Newton

La clase Newton consiste escencialmente de un método `solve` que se puede utilizar para resolver sistemas de ecuaciones no lineales. Además de éste método, también hay algunos parámetros de procedimiento, como el número máximo de iteraciones, que se pueden configurar en el constructor.

Para resolver, primero se comprueba si el residuo $r = F(x)$ ya es menor que el error `abslimit`. Si es así, el valor inicial ya es suficientemente aceptable y el proceso ha terminado. En otro caso, la dirección de búsqueda se determina utilizando la descomposición $LR \bigtriangledown f(x_k)^{-1} f(x_k) = z_k$ definitivamente. Usando un método de búsqueda de línea simple y un λ apropiado en $x_{k+1} = x_k - \lambda z_k$

```
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));                              // Norma calculada
}
if (newR<(1.0-0.25*lambda)*R)                   // Comprobacion de la convergencia
{
  x = y;
  R = newR;
  break;
}
else lambda *= 0.5;                             // Reduccion del factor de amortiguacion
if (R<=reduction*R0)                            // Comprobacion de la convergencia
{
  converged = true;
  return;
}
```

2.5.4. La clase `Banach`

- Soluciona un sistema no lineal de la forma $F(x) = 0$ usando el método de iteración del punto fijo $x = x - \sigma * F(x)$
- La función más importante es la función `solve`, que permite obtener la solución real
- Éste método hace uso del teorema del punto fijo de Banach.
- Una implementación concreta, utiliza la clase `Banach` no está incluida en ésta documentación ya que es similar a la clase `Newton` que está funcionando. Se puede ver un ejemplo en el archivo `wurzelbanach.cc`. La única diferencia está en el parámetro sigma, que también se debe tener en cuenta con Banach.

2.5.5. Implementación

```
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;
};
```

- Con Typedef ahorra papeleo y es más claro, en cuanto al tamaño.

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

- En el constructor, los valores se asignan a todos los parámetros privados,...

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

- ...que luego se pueden cambiar con las siguientes funciones. El parámetro "maxit" asegura que el solucionador que se explica más adelante no quede en un bucle infinito en el caso de que no haya convergencia en la iteración del punto fijo, en cuyo caso aborta e informa que no hay convergencia.

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

- Aquí se ajusta el parametro σ.

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

- Cuántos pasos debe tomar el solucionador, antes de abortar?, se puede configurar aquí.

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

- Control de salida: cuanto mayor sea el número establecido, más información se precisa sobre la convergencia en la consola. Lo que significan los números individuales, sin embargo, debe verse en el código fuente si es necesario.

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

- Tolerancia a fallos

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

- Factor de Reducción

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

- Con el método `solve` se puede resolver un modelo dado recurriendo a los miembros privados".

```
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());              // Residuo
  Vector<N> y(model.size());              // Soluciones temporales

  model.F(x,r);          // Calculo del residuo no lineal
  N R0(norm(r));         // Norma del resiudal inicial
  N R(R0);               // Norma de la corriente resiudal

  converged = false;

  // Maximo de iteraciones especificadas en la matriz
  for (size_type i=1; i<=maxit; i++)
  {
    if (R<=abslimit)     //Verifica el valor absoluto del residuo
    {
      converged = true;
      return;
    }
```

En caso de que el resultado preliminar no fuera lo suficientemente preciso ($\leqslant abslimit$), pasa a la siguiente iteración real y luego se usa un estándar para probar si el resultado es ahora lo suficientemente preciso y luego se regresa al comienzo del ciclo for. Si el resultado es lo suficientemente preciso, la función ha cumplido su propósito y termina.

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

  x = y;                           // óAdopcin de las nuevas iteraciones
  R = newR;                        // Almacenamiento estandar

  // Chequea la convergencia
  if (R<=reduction*R0 || R<=abslimit)
  {
    converged = true;
    return;
  }
}
```

- El valoor booleano se establece como falso por defecto en principio. Resuelve la función `solve` Si el sistema de ecuaciones tiene éxito, establece el valor en verdadero y este valor se retiene como un miembro privado de la clase. Esta función le dice si el método de punto fijo ya ha tenido éxito y, por lo tanto, también lo volverá a tener éxito.

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

3. Ecuaciones diferenciales ordinarias

El siguiente capítulo trata sobre el tema central de la conferencia. Numéricos 1, resolviendo ecuaciones diferenciales ordinarias. Repetir: se trata de una ecuación en la que se da una función así como las derivadas de la función y se intenta averiguar qué función cumple la ecuación. HDNUM proporciona algunas herramientas útiles para resolver este tipo de ecuaciones diferenciales. Muestra cómo preparar una ecuación diferencial para que pueda ser resuelta por un solucionador (cómo el funcionamiento individual se deja a la lección y sus demostraciones) y al mismo tiempo contiene varios de estos solucionadores. El hecho es que tanto las ecuaciones diferenciales como los solucionadores están empaquetados en clases. Estas clases deben tener ciertos métodos para ser compatibles entre sí. Comencemos con un ejemplo de ecuación diferencial:

3.1. Un buen ejemplo de DGL in HDNUM - `modelproblem.hh`

- Este archivo sólo contiene la clase `ModelProblem`, que contiene exactamente los métodos necesarios para la compatibilidad con todos los solucionadores de HDNUM. ¡Así que cada clase de ecuación diferencial debe tener exactamente estas declaraciones de método!
- La información completa sobre una ecuación diferencial está contenida en la implementación de los métodos.
- El archivo está escrito para que los objetos de la clase `ModelProblem` Los problemas modelo en el sentido de la lección son, pero pueden reescribirse para cualquier ecuación diferencial. Cabe señalar que todos los jefes de función de los métodos no se modifican. Esta es la única forma de mantener el nuevo DGL compatible con nuestros solucionadores.
- Un objeto de la clase de Modelproblem corresponde entonces a una ecuación diferencial a resolver.
- Si el archivo está incluido en el encabezado, los objetos de la clase de problema del modelo se pueden crear en el programa y luego resolver con el conocimiento de las siguientes secciones.

```
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;  //Valores iniciales
    void f (const T& t, const hdnum::Vector<N>& x,        //óFuncin f
      hdnum::Vector<N>& result) const;
    void f_x (const T& t, const hdnum::Vector<N>& x,  //Matriz Jacobiana de f
       hdnum::DenseMatrix<N>& result) const;

  private:
    N lambda;
};
```

- Las definiciones de tipo al principio no son métodos, pero merecen una breve explicación. Puede ver a partir de esto que es una clase de plantilla y nunca está claro desde el principio qué tipo de datos se está utilizando realmente. Las definiciones de tipo están ahí para que el solucionador pueda reconocer más tarde con qué tipo de número está trabajando realmente la clase de modelo. Los usamos para poder trabajar con tipos de datos muy precisos (precisión múltiple).

- El constructor inicializa los parámetros privados si es necesario. Pero no siempre tiene que haber tal cosa.

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

- Con esta función se determina qué dimensión tiene la ecuación diferencial a resolver.

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

- Los valores iniciales se establecen aquí. t_0 es el valor inicial en el tiempo, mientras que x_0 es el vector de los valores iniciales. Entonces, en el unidimensional, contiene solo una entrada.

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

- La función f contiene la ecuación diferencial real. El vector `result`, entonces la solución de la función `f` en el momento. `t` calculado.

```
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];
}
```

- Esta función representa la matriz jacobiana de la función `f` en `result` a disposición. Esto es requerido por solucionadores implícitos.

```
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;
}
```

- Cualquier parámetro que pueda ser necesario se encuentra en la parte privada de la clase.

3.2. Ejemplo de aplicación `modelproblem.hh`

El archivo `modelproblem_high_dim.hh` es una reformulación del archivo `modelproblem.hh` y pone la ecuación diferenial $u'(t) = \begin{pmatrix} 5 & -2 \\ -2 & 5 \end{pmatrix} * u(t)$ con valor inicial $u(t) = \begin{pmatrix} 1 \\ 3 \end{pmatrix}$ allí. Esto lo convierte en un ejemplo de la representación de un Ecuación diferencial.

3.3. El solucionador resuelve DGL - `modelproblem.cc`

- Este archivo es un excelente ejemplo de resolución de problemas Ecuaciones diferenciales.
- Muestra cómo combinar la clase de ecuación diferencial y la clase de solucionador de tal manera que la ecuación diferencial se resuelve y el resultado se escribe en un archivo de tal manera que se puede trazar.

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

Además de las bibliotecas, el encabezado también incluye el problema del modelo y un archivo para resolver la ecuación diferencial. En este caso, la ecuación diferencial debe resolverse con el Euler explícito.

```
int main ()
{
  typedef double Number;                  // Define el tipo de numero
  typedef ModelProblem<Number> Model;     // Define el tipo de modelo
  Model model(-1.0);                      // Objeto de la clase con lambda =-1
  typedef ExplicitEuler<Model> Solver;    // Elige el solucionador
  Solver solver(model);                   // Inicializar el solucionar con el modelo
  solver.set_dt(0.02);                    // Establecer intervalos de tiempo
  hdnum::Vector<Number> times;            // Vector para intervalos de tiempo
  hdnum::Vector<hdnum::Vector<Number>> states; // Vector ósolucin
  times.push_back(solver.get_time());     // Guarda la hora de inicio
  states.push_back(solver.get_state()); // Fuarda el valor inicial en el vector
  while (solver.get_time()<5.0-1e-6)      // Bucle para aflojar
    {
      solver.step();
      times.push_back(solver.get_time());     // Ahorrar tiempo
      states.push_back(solver.get_state());  // Ahorra el valor
    }

  gnuplot("mp2-ee-0.02.dat",times,states);   // La salida esta en la seccion
```

```
                                                    // explicado sobre Gnuplot
  return 0;
}
```

Un ejemplo alternativo es el archivo `modelproblem_high_dim.cc`. Pasado por Solver EE otros solucionadores `ode.hh` (Ver explicación a continuación) reemplazado, se puede resolver el DGL con diferentes medios y ver un ejemplo de que un DGL con todos nuestros solucionadores es compatible.

3.4. Que debe poder hacer un solucionador? - `expliciteuler.hh`

- Este archivo contiene la clase `ExplicitEuler`.
- La clase contiene todos los métodos que necesita un solucionador en nuestro contexto.
- Todos los solucionadores tienen al menos los métodos que `ExplicitEuler` tiene, tal vez algunos más.
- Con la ayuda de este archivo puede resolver todas las ecuaciones diferenciales que tienen la representación mencionada anteriormente en una clase.

```
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
};
```

- Primero, un breve comentario sobre las definiciones de tipo al principio: Inicialmente, el solucionador solo obtuvo una referencia a un modelo en el constructor. Sin embargo, todavía no está claro qué tipos de números se utilizan en el modelo y si se pueden llamar a sus funciones. Los typedefs se utilizan para determinar esto. Por lo tanto, el solucionador puede resolver DGL para cualquier tipo de número y solo cuando se compila se determina cuál se refiere realmente.
- El constructor almacena una referencia al modelo que se supone que debe resolver. Además, aquí se especifican los parámetros para el algoritmo de solución, como el tamaño de los pasos de tiempo, los valores iniciales o similares.

  ```
  template<class M>
  ExplicitEuler::ExplicitEuler (const M& model_)
  ```

```
  : model(model_), u(model.size()), f(model.size())
{
  model.initialize(t,u);
  dt = 0.1;
}
```

- Dado que Solver calcula la solución (tiempo) paso a (tiempo) paso, puede determinar qué tan grandes deben ser estos pasos. Cuanto mayores sean los pasos, menos preciso será el resultado, pero menor será el esfuerzo computacional.

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

- El algoritmo de solución real está en la función de paso. Decide cómo pasar de un paso al siguiente. Así que aquí está el algoritmo del Euler explícito.

```
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
}
```

- El vector solución calculado hasta ahora:

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

- El tiempo que se acaba de calcular:

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

- Actualiza dt (tamaño del paso):

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

3.5. Insertar Gnuplot en `ode.hh`

Por supuesto, un solucionador numérico no puede proporcionarnos una solución analítica de una ecuación diferencial en forma de función concreta. Pero puede decirnos cómo se ve la función de solución en muchos puntos. Para que podamos hacer algo con estas muchas tuplas numéricas, las visualizamos con Gnuplot. Las siguientes funciones de plantilla nos facilitan esto y escriben el resultado en el formato correcto en un archivo para que luego podamos trazarlo directamente.

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

   ¡Solo apto para DGL unidimensional! La función recibe un nombre de archivo (.dat) entre comillas, así como la hora y el vector de solución. La función asegura que los datos estén en una especie de tabla en un archivo con el nombre deseado. A continuación, este archivo se puede trazar.

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

   ¡Adecuado para DGL bidimensional! Pasas los mismos datos a la función que arriba más el segundo vector de solución. El resultado también es análogo. Sin embargo, al trazar, debe tener en cuenta las peculiaridades de la multidimensionalidad.

Como plantilla de ejemplo, el código de la sección anterior se puede ver al final.
Los comandos de Gnuplot más importantes en la terminal:

1. `gnuplot` - abre Gnuplot
2. `plot 'dateiname.dat'using 1:2` - grafica en dos dimensiones usando las columnas 1 vs 2
3. `plot 'dateiname.dat'using 1:2, 'dateiname.dat'using 1:3` - traza gráficas 1 vs 2, y 1 vs 3.
4. `splot 'dateiname.dat'using 1:2:3` - Grafica en tres dimensiones
5. `exit` - termina Gnuplot

3.6. Proceso de un paso- `ode.hh`

Ahora que hemos visto cómo se empaquetan exactamente una ecuación diferencial y un solucionador en una clase para que sean compatibles entre sí y cómo luego se resuelve la ecuación diferencial con el solucionador, podemos pasar a ver varios de estos solucionadores. En la conferencia conocerás los métodos implícitos y explícitos de Runge-Kutta como los ejemplos más importantes. El Euler explícito, que teníamos antes como ejemplo, también es parte de él. En el archivo `ode.hh` se implementan varios de estos solucionadores. Para que pueda resolver cualquier ecuación diferencial (nuevamente empaquetada en una clase, por supuesto) con cada solucionador, todas estas clases de solucionador tienen métodos con los mismos cabezales de función. Solo en la forma en que se implementan estas funciones difieren, lo que luego define el procedimiento individual. Además de los métodos de la clase discutidos anteriormente `ExplicitEuler` tener las clases en `ode.hh` algunas funciones adicionales. Los métodos con control de tamaño de paso también se modifican ligeramente.

3.6.1. Los procedimientos en ode.hh

- Método explícito de Runge-Kutta
  - `EE` - Método explícito de Euler
  - `ModifiedEuler`
  - `Heun2`
  - `Heun3`

- `Kutta3`
- `RungeKutta4`

- Método implícito de Runge-Kutta
  - `IE` - Método implícito de Euler
  - `DIRK` -Procedimiento implícito diagonal

- Control de tamaño del paso
  - `RKF45`
  - `RE` - Extrapolación de Richardson

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:

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

  wobei $F_i(z_1, ..., z_s) = z_i - h_n \sum_{j=1}^{s} a_{ij} f(t_{n-1} + c_j h_n, u_{n-1} + z_j)$ 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