

Tomáš Oberhuber

Faculty of Nuclear Sciences and Physical Engineering
Czech Technical University in Prague

Zdrojové kódy s příklady jsou na:

<https://gitlab.com/oberhuber.tomas/fjfi-num-src>.

Stahovat lze pomocí:

```
1 git clone git@gitlab.com:oberhuber.tomas/fjfi-num-src.git
```

nebo tlačítkem *Download*.

Rozbalíme příkazem

```
1 tar xvf fjfi-num-src*
```

```
2
```

```
3 cd fjfi-num-src*
```

- řešíme úlohu typu:

$$\frac{d\vec{u}}{dt} = \vec{f}(t, \vec{u}(t)) \text{ pro } t \in (0, T), \quad (1)$$

$$\vec{u}(0) = \vec{u}_{ini}, \quad (2)$$

kde $\vec{u} : \langle 0, T \rangle \rightarrow \mathbb{R}^n$, $\vec{f} : \langle 0, T \rangle \times \mathbb{R}^n \rightarrow \mathbb{R}^n$ a $\vec{u}_{ini} \in \mathbb{R}^n$ je okrajová (počáteční) podmínka.

- n se často také označuje jako **DOF**
 - **degrees of freedom** = stupně volnosti
 - znamená to, kolik parametrů použijeme k aproximaci řešení úlohy v daném čase t
- tuto úlohu lze řešit některou z Rungových-Kuttových metod
- pro jednoduchost nejprve zvolíme Eulerovu variantu

$$\vec{k}_1 = \tau \vec{f}(t, \vec{u}(t)) \quad (3)$$

$$\vec{u}(t + \tau) = \vec{u}(t) + \vec{k}_1 \quad (4)$$

Algoritmicky lze řešit naši úlohu zapsat takto:

procedure SOLVEODE($\vec{u}_{ini}$, $\vec{f}$, t_{stop} , τ_0)

$t := 0$

$\vec{u} := \vec{u}_{ini}$

while $t < t_{stop}$ **do**

$\tau := \min\{\tau_0, t_{stop} - t\}$

$\vec{u} := \vec{u} + \text{update}(\vec{u}, \vec{f}, \tau)$

$t := t + \tau$

end while

end procedure

Algoritmus 1: Algoritmus pro řešení ODR pomocí Rungových-Kuttových metod.

procedure UPDATE($\vec{u}$, $\vec{f}$, τ)

$\vec{k}_1 := \tau \vec{f}(t, \vec{u})$

return $\vec{k}_1$

end procedure

Algoritmus 2: Příklad implementace nejjednodušší
Rungovy-Kuttovy metody.

Řešení ODR

Naším cílem bude odvodit kód, ve kterém bude snadné měnit:

- úlohu, kterou řešíme
 - $\vec{f}$, $\vec{u}_{ini}$
- použitou Rungovu-Kuttovu metodu
 - v našem případě ji reprezentuje funkce `update`

Popis úlohy

Ve zdrojových kódech je každá úloha odvozována z objektu `ODEProblem`:

```
1 struct ODEProblem
2 {
3     virtual int getDegreesOfFreedom();
4
5     virtual void getRightHandSide( const double t,
6                                     const double* u,
7                                     double* fu );
8
9     virtual bool writeSolution( const double t,
10                                const int step,
11                                const double* u );
12 };
```

Kód lze nalézt v souboru `ode/ODEProblem.h`

Všechny metody jsou virtuální, protože popisují rozhraní obecné úlohy, podobně, jako tomu bylo v případě objektu `struct Function` popisujícím obecnou funkci v předchozí prezentaci.

Význam jednotlivých metod:

- `int getDegreesOfFreedom()`
 - vrací počet stupňů volnosti dané úlohy, tj. n pro $\vec{u} \in \langle 0, T \rangle \times \mathbb{R}^n$
- `void getRightHandSide( t, u, fu )`
 - napočítává pravou stranu rovnice (2)
 - vektor $\vec{u}$ a výsledek $\vec{f}(t, \vec{u})$ jsou reprezentovány formou ukazatele na pole
 - velikost těchto polí je vždy rovna počtu DOF, tj. n
- `bool writeSolution( t, step, u )`
 - v předem určených časových bodech provádí uložení stavu řešení u do souboru
 - t odpovídá času simulace
 - $step$ odpovídá celkovému počtu již uložených stavů

Popis Ruungovy-Kuttovy metody

Ve zdrojových kódech je každá Rungova-Kuttova metoda odvozována z objektu `ODESolver`:

```
1 struct ODESolver
2 {
3     virtual bool setup( const int degreesOfFreedom );
4
5     virtual bool solve( const double integrationTimeStep ,
6                         const double stopTime ,
7                         double* time ,
8                         ODEProblem* problem ,
9                         double* u );
10};
```

Kód lze nalézt v souboru `ode/ODESolver.h`

Metody jsou opět označeny jako virtuální, protože jde o popis obecného objektu.

Popis Rungovy-Kuttovy metody

Význam jednotlivých metod:

- `int setup( degreesOfFreedom )`
 - tato metoda slouží pro alokaci pomocných polí, zejména pro vektory $\vec{k}_1, \vec{k}_2 \dots$
 - jejich velikost opět odpovídá počtu DOFů
- `void solve( ... )`
 - napočítává řešení u úlohy `problem` od času daného hodnotou proměnné `time` s časovým krokem τ_0 rovnému `integrationTimeStep` až do času t_{stop} rovnému `stopTime`

Funkce pro řešení ODR pak může vypadat takto:

```
1 bool solve( const double initialTime ,
2             const double finalTime ,
3             const double timeStep ,
4             const double integrationTimeStep ,
5             ODEProblem* problem ,
6             ODESolver* solver ,
7             double* u )
8 {
9     solver->setup( problem->getDegreesOfFreedom() );
10    const int timeStepsCount =
11        ceil( max( 0.0, finalTime - initialTime ) / timeStep );
12    double time( initialTime );
13    for( int k = 1; k <= timeStepsCount; k++ )
14    {
15        double progress = (double) k / (double) timeStepsCount;
16        printf( "Solving time step %d / %d => %f %\n",
17               k, timeStepsCount, 100.0*progress );
18        if( ! solver->solve( integrationTimeStep ,
19                           time + timeStep, // stopTime
20                           &time ,
21                           problem ,
22                           u ) )
23            return false;
24        time += timeStep;
25        problem->writeSolution( time , k , u );
26    }
27    printf( "Done. \n");
28    return true;
29 }
```

Kód lze najít v souboru ode/ode-solve.h

Řešení ODR

Všimněme si, že se v kódu objevují dva různé časové kroky:

- `timeStep` - udává, v jakých intervalech se ukládá stav řešení ulohy $\vec{u}(t)$
- `integrationTimeStep` - udává integrační časový krok, který odpovídá parametru τ_0 v algoritmu 1

Implementace Rungovy-Kuttovy metody prvního řádu

Implementace Eulerova ODE řešiče vypadá takto:

Listing 1: "ode/Euler.h"

```
1 class Euler : public ODESolver
2 {
3     public:
4
5         // Constructor
6         Euler();
7
8         bool setup( const int degreesOfFreedom );
9
10        bool solve( const double integrationTimeStep ,
11                   const double stopTime ,
12                   double* time ,
13                   ODEProblem* problem ,
14                   double* u );
15
16        // Destructor
17        ~Euler();
18
19    protected:
20
21        double* k1;
22    };
```

Implementace Rungovy-Kuttovy metody prvního řádu

- objekt `Euler` je odvozen od `ODESolver`
- konstruktor `Euler()` se volá automaticky při vytvoření instance objektu
 - v našem případě se v něm nuluje ukazatel `k1`
- destruktorka `~Euler()` se volá automaticky při zániku instance objektu
 - v našem případě má na starost dealokaci dynamicky alokovaného pole `k1`

Implementace Rungovy-Kuttovy metody prvního řádu

Listing 2: "ode/Euler.cpp"

```
1 Euler::Euler()
2 {
3     k1 = 0;
4 }
5
6 bool Euler::setup( const int degreesOfFreedom )
7 {
8     k1 = new double[ degreesOfFreedom ];
9     if ( ! k1 )
10         return false;
11     return true;
12 }
13
14 Euler::~~Euler()
15 {
16     if ( k ) delete[] k;
17 }
```

- konstruktor `Euler()` nuluje ukazatel `k1`
- metoda `setup` alokuje do ukazatele `k1` pole o velikosti dané počtem DOFů
- destruktork `~Euler()` dealokuje alokovanou paměť

Implementace Rungovy-Kuttovy metody prvního řádu

Implementace
řešičů ODR

Riccatiho
rovnice

Experimentální
řád
konvergence

Harmonický
oscilátor

Adaptivní
volba
časového
kroku

Soupeřící
druhy

Lorenzovy
rovnice

Úloha n-těles

Listing 3: "ode/Euler.cpp"

```
1  bool Euler::solve( const double integrationTimeStep ,
2                      const double stopTime ,
3                      double* time ,
4                      ODEProblem* problem ,
5                      double* u )
6  {
7      const int dofs = problem->getDegreesOfFreedom();
8      while( *time < stopTime )
9      {
10         // compute current tau
11         double tau = min(integrationTimeStep , stopTime - *time );
12
13         // compute k1
14         problem->getRightHandSide( *time , u , k1 );
15
16         // update to the next time step
17         for( int i = 0; i < dofs; i++ )
18             u[ i ] += tau * k1[ i ];
19         *time += tau;
20     }
21     return true;
22 }
```


Implementace Rungovy-Kuttovy metody prvního řádu

Domácí úkol

- do souborů `ode/RungeKutta.h` a `ode/RungeKutta.cpp` dopište kód pro Rungovu-Kuttovu metodu druhého řádu
- zdrojové kódy vytisknete a odevzdejte

Jde o rovnici tvaru:

$$\frac{du(t)}{dt} = a_0(t) + a_1(t)u(t) + a_2(t)u^2(t).$$

Konkrétněji např.

$$\frac{du(t)}{dt} = t^{-4}e^t + u(t) + 2e^{-t}u^2(t). \quad (5)$$

Řešení této rovnice je:

$$u(t) = e^t \left[\frac{1}{\sqrt{2}t^2} \tan \sqrt{2} \left(c - \frac{1}{x} \right) - \frac{1}{2x} \right] \quad (6)$$

pro $c \in \mathbb{R}$.

Riccatiho rovnice

Tato úloha je implementována v souborech

ode/RiccatiProblem.h, RiccatiProblem.cpp a
ricatti.cpp

Listing 4: "ode/RicattiProblem.h"

```
1 class RiccatiProblem : public ODEProblem
2 {
3     public:
4
5         RiccatiProblem ();
6
7         int getDegreesOfFreedom ();
8
9         void getRightHandSide( const double& t,
10                                const double* _u,
11                                double* fu );
12
13         double getExactSolution( const double& t ,
14                                   const double& c = 1.0 );
15
16         bool writeExactSolution( const char* fileName,
17                                  const double& initialTime,
18                                  const double& finalTime,
19                                  const double& timeStep,
20                                  const double& c = 1.0 );
21
22         bool writeSolution( const double& t,
23                              int step,
24                              const double* u );
25
26         double getL1Error( const double timeStep );
27
28         double getL2Error( const double timeStep );
29
30         double getMaxError ();
31
32         double l1Error, l2Error, maxError;
33     };
```

Listing 5: "ode/RicattiProblem.cpp"

```
1 int RiccatiProblem::getDegreesOfFreedom()  
2 {  
3     return 1;  
4 }
```

- metoda `getDegreesOfFreedom()` vrací počet stupňů volnosti úlohy
- v našem případě je $u : \langle 0, T \rangle \rightarrow \mathbb{R}^1$, tj. $\text{DOF} = 1$

Listing 6: "ode/RicattiProblem.cpp"

```
1 void RiccatiProblem::getRightHandSide( const double& t ,  
2                                         const double* _u ,  
3                                         double* fu )  
4 {  
5     // create alias so that we can write u insted of _u[0]  
6     const double& u = _u[ 0 ];  
7     fu[ 0 ] = pow( t, -4.0 ) * exp( t ) + u +  
8               2.0 * exp( -t ) * u * u;  
9 }
```

- metoda `getRightHandSide()` napočítává pravou stranu vztahu (5), tj. výraz

$$t^{-4}e^t + u(t) + 2e^{-t}u^2(t)$$

Listing 7: "ode/RicattiProblem.cpp"

```
1 double RiccatiProblem::getExactSolution( const double& t ,  
2                                           const double& c )  
3 {  
4     const double sqrt_2 = sqrt( 2.0 );  
5     return exp( t ) * ( 1.0 / ( sqrt_2 * t * t ) *  
6                         tan( sqrt_2 * ( c - 1.0 / t ) ) -  
7                         1.0 / ( 2.0 * t ) );  
8 }
```

- metoda `getExactSolution()` přesné řešení (6), tj. výraz

$$u(t) = e^t \left[\frac{1}{\sqrt{2}t^2} \tan \sqrt{2} \left(c - \frac{1}{x} \right) - \frac{1}{2x} \right]$$

- budeme ho využívat k napočítání chyby našich numerických metod

Listing 8: "ode/RicattiProblem.cpp"

```
1  bool RiccatiProblem::writeExactSolution(  
2      const char*   fileName ,  
3      const double& initialTime ,  
4      const double& finalTime ,  
5      const double& timeStep ,  
6      const double& c )  
7  {  
8      std::fstream file ;  
9      file.open( fileName , std::ios::out );  
10     double t = initialTime ;  
11     while( t < finalTime )  
12     {  
13         file << t << " " << this->getExactSolution( t , c )  
14             << std::endl ;  
15         t = std::min( t + timeStep , finalTime ) ;  
16     }  
17 }
```

- metoda `getExactSolution()` vypisuje přesné řešení na intervalu $\langle initialTime, finalTime \rangle$ s krokem (rozlišením) `timeStep`
- výsledný graf lze zobrazit pomocí programu `gnuplot`

Formát dat pro program `gnuplot` při vykreslování funkce

$u : \mathbb{R} \rightarrow \mathbb{R}$ je:

```
1 t0 u(t0)
2 t1 u(t1)
3
4 ...
5
6 tn u(tn)
```

tj. hodnota proměnné t + mezera + příslušná funkční
hodnota + konec řádku.

Listing 9: "ode/RicattiProblem.cpp"

```
1 bool RiccatiProblem::writeSolution( const double& t,
2                                     int step,
3                                     const double* u )
4 {
5     const double& u = _u[ 0 ];
6     fstream file;
7     if( step == 0 )
8     {
9         /*
10          * In the first step, we want to rewrite the file
11          */
12         file.open( "riccati.txt", ios::out );
13         if( !file ) return false;
14     }
15     else
16     {
17         /*
18          * In later steps, we just append new time steps
19          */
20         file.open( "riccati.txt", ios::out | ios::app );
21         if( !file ) return false;
22     }
23     file << t << " " << u << endl;
24
25     /*
26      * Evaluate errors of the approximation
27      */
28     const double diff = fabs( getExactSolution(t) - u );
29     l1Error += diff;
30     l2Error += diff * diff;
31     maxError = std::max( maxError, diff );
32 }
```

- při vypisování prvního časového kroku (`step == 0`) chceme výstupní soubor přepsat
- proto ho otevíráme v módu `ios::out`
- při dalších časových krocích již chceme do tohoto souboru jen připisovat
- proto ho otevíráme v módu `ios::out || ios::app`

Implementace
řešičů ODR

Riccatiho
rovnice

Experimentální
řád
konvergence

Harmonický
oscilátor

Adaptivní
volba
časového
kroku

Soupeřící
druhy

Lorenzovy
rovnice

Úloha n-těles

```
1 double RiccatiProblem::getL1Error( const double timeStep )
2 {
3     return timeStep * l1Error;
4 }
5
6 double RiccatiProblem::getL2Error( const double timeStep )
7 {
8     return sqrt( timeStep * l2Error );
9 }
10
11 double RiccatiProblem::getMaxError()
12 {
13     return maxError;
14 }
```

Zbývající metody napočítávají celkové normy chyb.

Chyby aproximace

Jsou definovány jako:

$$\|e\|_{L_1\langle 0,T \rangle} := \int_0^T |e(t)| dt \approx \sum_{i=0}^N |e(i\Delta t)| \Delta t,$$

$$\|e\|_{L_2\langle 0,T \rangle} := \left(\int_0^T |e(t)|^2 dt \right)^{\frac{1}{2}} \approx \left(\sum_{i=0}^N |e(i\Delta t)|^2 \Delta t \right)^{\frac{1}{2}},$$

$$\|e\|_{L_\infty\langle 0,T \rangle} := \sup_{t \in \langle 0,T \rangle} |e(t)| \approx \max_{i \in 0 \dots N} |e(i\Delta t)|,$$

kde $\Delta t := \frac{T}{N}$ a

$$e(t) := u(t) - u_\tau(t),$$

pro u označující přesné řešení a u_τ přibližné řešení.

Samotný řešič Riccatiho rovnice pak vypadá takto:

```
1 #include <cstdlib>
2 #include "RiccatiProblem.h"
3 #include "Euler.h"
4 #include "ode-solve.h"
5
6 using namespace std;
7
8 const double initialTime( 0.0 );
9 const double finalTime( 0.15 );
10 const double timeStep( 1.0e-4 );
11 const double integrationTimeStep( 1.0e-4 );
12
13 int main( int argc, char** argv )
14 {
15     RiccatiProblem problem;
16     Euler integrator;
17
18     /**
19      * Set initial condition
20      */
21     double u = problem.getExactSolution( initialTime );
22
23     if ( ! solve( initialTime,
24                 finalTime,
25                 timeStep,
26                 integrationTimeStep,
27                 &problem,
28                 &integrator,
29                 &u ) )
30         return EXIT_FAILURE;
31
32     cout << "L1 error: " << problem.getL1Error( timeStep ) << endl
33          << "L2 error: " << problem.getL2Error( timeStep ) << endl
34          << "Max error: " << problem.getMaxError() << endl;
35
36     return EXIT_SUCCESS;
37 }
```

Riccatiho rovnice

Parametry na řádcích 8 - 11 mají tento význam:

- úloha je řešena na interval $\langle initialTime, finalTime \rangle$
- stav se ukládá v časech $initialTime + i \cdot timeStep$ pro $i = 0, 1, \dots$
- příslušná Rungova-Kuttova metoda aproximuje řešení úlohy s krokem $integrationTimeStep$
- na řádcích 15 a 16 vytváříme instance úlohy a příslušného řešiče.
- na řádku 21 nastavujeme počáteční/okrajovou podmínku.
- na řádcích 23 - 30 voláme řešič úlohy
- na řádcích 32 - 34 vypisujeme normy chyb aproximace

V prostředí linuxu pak psotupujeme následujícím způsobem:

```
1 cd fjfi --num--src*
2
3 make install
4
5 ~/.local/bin/riccati
6
7 gnuplot
8
9 plot 'riccati.txt'
```

Implementace
řešičů ODR

Riccatiho
rovnice

Experimentální
řád
konvergence

Harmonický
oscilátor

Adaptivní
volba
časového
kroku

Soupeřící
druhy

Lorenzovy
rovnice

Úloha n-těles

Domácí úkol

- zvolte vhodný interval, kde je možné napočítat numericky řešení Riccatiho rovnice
- napočítejte *experimentální řád konvergence* - EOC pro Eulerův a vámi implementovaný Rungův-Kuttův řešič
- napište report k numerické analýze
 - řešení rovnice
 - interval, na kterém ji řešíme
 - počáteční podmínka
 - parametry úlohy
 - zvolená numerická metoda
 - parametry numerické metody
 - tabulka EOC

Definition 1

Bud' $\tau_1 > \tau_2$, u přesné řešení úlohy, u_{τ_1} a u_{τ_2} přibližná řešení získaná integrací s časovými kroky τ_1 a τ_2 . Nechť $e_{\tau_1}(t) := u(t) - u_{\tau_1}(t)$ a $e_{\tau_2}(t) := u(t) - u_{\tau_2}(t)$. Dále $E_{\tau_1} := \|e_{\tau_1}\|_{L_*\langle 0, T \rangle}$, $E_{\tau_2} := \|e_{\tau_2}\|_{L_*\langle 0, T \rangle}$ ve vhodné normě. Pak definujeme

$$EOC(\tau_1, \tau_2) := \frac{\log(E_{\tau_1}/E_{\tau_2})}{\log(\tau_1/\tau_2)}.$$

Remark 2

Je-li $\tau_1/\tau_2 = 2$, tj $\tau_2 = \frac{1}{2}\tau_1$, pak

$$EOC(\tau_1, \tau_2) := \frac{\log(E_{\tau_1}/E_{\tau_2})}{\log(\tau_1/\tau_2)} = \log_2(E_{\tau_1}/E_{\tau_2}).$$

Pokud $E_{\tau_1}/E_{\tau_2} = 2 \Rightarrow EOC(\tau_1, \tau_2) = 1$.

Pokud $E_{\tau_1}/E_{\tau_2} = 4 \Rightarrow EOC(\tau_1, \tau_2) = 2$.

Pokud $E_{\tau_1}/E_{\tau_2} = 8 \Rightarrow EOC(\tau_1, \tau_2) = 3$.

Pokud $E_{\tau_1}/E_{\tau_2} = 16 \Rightarrow EOC(\tau_1, \tau_2) = 4$.

Example 3

Příklad EOC tabulky

N	τ	$L_1(0, T)$		$L_2(0, T)$		$L_\infty(0, T)$	
		Err.	EOC	Err.	EOC	Err.	EOC
17	0.5	2.1		4		19	
33	0.25	0.46	2.2	0.64	2.6	2.9	2.7
65	0.125	0.15	1.6	0.34	0.92	1.9	0.59
129	0.0625	0.06	1.3	0.16	1.1	1.1	0.74
257	0.03125	0.025	1.2	0.08	0.99	0.72	0.66
513	0.015625	0.012	1.05	0.041	0.96	0.42	0.77

Harmonický oscilátor

Řešíme úlohu:

$$\vec{F} = -k\vec{x}.$$

Z fyziky víme, že

$$\vec{F} = m\vec{a} = m\ddot{\vec{x}},$$

tj.

$$\ddot{\vec{x}} + \frac{k}{m}\vec{x} = 0.$$

My budeme řešit úlohu tzv. tlumeného oscilátoru v $\mathbb{R}^1$

$$\ddot{u}(t) + \epsilon u^2(t)\dot{u}(t) + u(t) = 0.$$

Harmonický oscilátor

Tuto rovnici druhého řádu převedeme na soustavu dvou rovnic prvního řádu:

$$\dot{u}_1 := u_2, \quad (7)$$

$$\dot{u}_2 := -u_1 - \epsilon u_1^2 u_2, \quad (8)$$

s počáteční podmínkou

$$\vec{u} |_{t=0} = (u_{1,ini}, u_{2,ini})^T.$$

Harmonický oscilátor

Řešič je implementován v souborech

ode/HyperbolicProblem.h,
ode/HyperbolicProblem.cpp a
ode/hyperbolic.cpp.

Listing 10: "ode/HyperbolicProblem.cpp"

```
1  int HyperbolicProblem::getDegreesOfFreedom()
2  {
3      return 2;
4  }
5
6  void HyperbolicProblem::getRightHandSide( const double& t,
7                                              const double* _u,
8                                              double* fu )
9  {
10     const double& u1 = _u[ 0 ];
11     const double& u2 = _u[ 1 ];
12     fu[ 0 ] = u2;
13     fu[ 1 ] = -u1 - epsilon * u1 * u1 * u2;
14 }
```

Listing 11: "ode/hyperbolic.cpp"

```
1  #include "HyperbolicProblem.h"
2  #include "Euler.h"
3  #include "ode-solve.h"
4
5  typedef HyperbolicProblem Problem;
6
7  const double initialTime( 0.0 );
8  const double finalTime( 100.0 );
9  const double timeStep( 1.0e-1 );
10 const double integrationTimeStep( 1.0 );
11
12 int main( int argc, char** argv )
13 {
14     HyperbolicProblem problem;
15     problem.setEpsilon( 0.0 );
16     Euler integrator;
17
18     double u[ 2 ] = { 0.0, 1.0 };
19
20     if( ! solve( initialTime,
21                 finalTime,
22                 timeStep,
23                 integrationTimeStep,
24                 &problem,
25                 &integrator,
26                 u ) )
27         return EXIT_FAILURE;
28     return EXIT_SUCCESS;
29 }
```


Harmonický oscilátor

Example 4

Zmenšujte intergační krok pro simulaci harmonického oscilátoru.

Mersonova-Rungova-Kuttova metoda

Implementace
řešičů ODR

Riccatiho
rovnice

Experimentální
řád
konvergence

Harmonický
oscilátor

Adaptivní
volba
časového
kroku

Soupeřící
druhy

Lorenzovy
rovnice

Úloha n-těles

procedure MERSONRUNGEKUTTA($\vec{u}, \vec{f}, t_{stop}, \tau_0$)

$t := 0$

$\vec{u} := \vec{u}_{ini}$

while $t \leq t_{stop}$ **do**

$\tau := \min\{\tau_0, t_{stop} - t\}$

$\vec{k}_1 := \tau \vec{f}(t, \vec{u})$

$\vec{k}_2 := \tau \vec{f}\left(t + \frac{1}{3}\tau, \vec{u} + \frac{1}{3}\vec{k}_1\right)$

$\vec{k}_3 := \tau \vec{f}\left(t + \frac{1}{3}\tau, \vec{u} + \frac{1}{6}\vec{k}_1 + \frac{1}{6}\vec{k}_2\right)$

$\vec{k}_4 := \tau \vec{f}\left(t + \frac{1}{2}\tau, \vec{u} + \frac{1}{8}\vec{k}_1 + \frac{3}{8}\vec{k}_3\right)$

$\vec{k}_5 := \tau \vec{f}\left(t + \tau, \vec{u} + \frac{1}{2}\vec{k}_1 - \frac{3}{2}\vec{k}_3 + 2\vec{k}_4\right)$

$e := \frac{1}{3} \left\| \frac{1}{5}\vec{k}_1 - \frac{9}{10}\vec{k}_3 + \frac{4}{5}\vec{k}_4 - \frac{1}{10}\vec{k}_5 \right\|_{\ell_\infty}$

if $e < \epsilon$ **then**

$\vec{u} := \vec{u} + \frac{1}{6}(\vec{k}_1 + 4\vec{k}_4 + \vec{k}_5)$

$t := t + \tau$

end if

$\tau := \min\left\{\tau \cdot \frac{4}{5} \left(\frac{\epsilon}{e}\right)^{\frac{1}{5}}, t_{stop} - t\right\}.$

end while

end procedure

Mersonova-Rungova-Kuttova metoda

Tato metoda je implementována v souborech
`ode/Merson.h` a `ode/Merson.cpp`.

- parametr ϵ se nastavuje pomocí metody
`setAdaptivity( eps )`
- tento parametr se nastavuje na hodnotu $10^{-3} - 10^{-12}$

Example 5

Proveďte simulaci harmonického oscilátoru s pomocí
Mersonovy metody.

Jde o úlohu typu:

$$\begin{aligned}\dot{u}_1 &= u_1 - au_1^2 - cu_1u_2, \\ \dot{u}_2 &= u_2 - bu_2^2 + du_1u_2,\end{aligned}$$

kde a, b, c a d jsou konstanty.

- jde o typ úloh zvané Volterrový-Lotkovy úlohy
- simuluje soupeření dvou živočišných druhů - predátor a oběť
- $u : \langle 0, T \rangle \rightarrow \mathbb{R}^2$, tj. DOFs = 2
- úloha je implementována v souborech
`SpeciesProblem.h`, `SpeciesProblem.cpp` a
`species.cpp`

Domácí úkol

Proveďte numerickou analýzu, tj. pokuste se najít zajímavou kombinaci parametrů a , b , c a d a napište opět report doplněný obrázky.

- řešená rovnice
- interval, na kterém ji řešíme
- počáteční podmínka
- parametry úlohy
- zvolená numerická metoda
- parametry numerické metody
- obrázky s výsledky

Lorenzovy rovnice

Řešíme úlohu:

$$\dot{u}_1 = \sigma(u_2 - u_1),$$

$$\dot{u}_2 = \rho u_1 - u_2 - u_1 u_3,$$

$$\dot{u}_3 = -\beta u_3 + u_1 u_2,$$

kde $\sigma, \rho, \beta > 0$ jsou konstanty.

- jde o jednoduchý model používaný v meteorologii
- úloha dala vzniknout teorii deterministického chaosu
- řešením může být tzv. Lorenzův atraktor

Lorenzovy rovnice

Domácí úkol

Proveďte numerickou studii Lorenzových rovnic a pokuste se najít Lorenzův atraktor.

Řešíme úlohu:

$$m_i \frac{d^2 \vec{p}_i}{dt} = \sum_{\substack{j=1 \\ j \neq i}}^n \frac{Gm_i m_j}{|\vec{p}_j - \vec{p}_i|} (\vec{p}_j - \vec{p}_i), i = 1, \dots, n$$

tj.

$$\frac{d^2 \vec{p}_i}{dt} = \sum_{\substack{j=1 \\ j \neq i}}^n \frac{Gm_j}{|\vec{p}_j - \vec{p}_i|} (\vec{p}_j - \vec{p}_i), i = 1, \dots, n$$

tj.

$$\frac{d\vec{v}_i}{dt} = \sum_{\substack{j=1 \\ j \neq i}}^n \frac{Gm_j}{|\vec{p}_j - \vec{p}_i|} (\vec{p}_j - \vec{p}_i), i = 1, \dots, n,$$

$$\frac{d\vec{p}_i}{dt} = \vec{v}_i, i = 1, \dots, n.$$

Úloha n-těles

- každá částice je popsána vektorem pozice $\vec{p}_i$ a rychlosti $\vec{v}_i$
- tj. $2d$ DOFů, kde d je dimenze prostoru
- celkem dostáváme $2dn$ DOFů
-

$$\vec{U} = \left(\underbrace{u_1, \dots, u_{dn}}_{\vec{v}_1, \dots, \vec{v}_n}, \underbrace{u_{dn+1}, \dots, u_{2dn}}_{\vec{p}_1, \dots, \vec{p}_n} \right)^T$$

- počáteční podmínka vlastně nastavuje pozici a rychlost každé částice na začátku
- úloha je implementovaná v souborech
`NBodyProblem.h`, `NBodyProblem.cpp` a
`nbody.cpp`

Úloha n-těles

Implementace
řešičů ODR

Riccatiho
rovnice

Experimentální
řád
konvergence

Harmonický
oscilátor

Adaptivní
volba
časového
kroku

Soupeřící
druhy

Lorenzovy
rovnice

Úloha n-těles

```
1 void NBodyProblem::setInitialCondition( double* u )
2 {
3     const int n = this->particlesCount;
4     for( int i = 0; i < this->particlesCount; i++ )
5     {
6         /****
7          * Initial velocity
8          */
9         u[ 2 * i ] = 0;
10        u[ 2 * i + 1 ] = 0;
11
12        /****
13         * Initial position
14         */
15        u[ n + 2 * i ] =
16            ( double ) ( rand() % 10000 ) / 100.0 - 50.0;
17        u[ n + 2 * i + 1 ] =
18            ( double ) ( rand() % 10000 ) / 100.0 - 50.0;
19
20        /****
21         * Particle mass
22         */
23        this->masses[ i ] = ( double ) ( rand() % 100 ) /
24            100.0 + 1.0;
25    }
26 }
```

Implementace
řešičů ODRRiccatiho
rovniceExperimentální
řád
konvergenceHarmonický
osilátorAdaptivní
volba
časového
krokuSoupeřící
druhyLorenzovy
rovnice

Úloha n-těles

```
1 void NBodyProblem::getRightHandSide( const double& t,
2                                       const double* _u,
3                                       double* fu )
4 {
5     const double epsilon = 1.0e-1;
6     const int n = this->particlesCount;
7     for( int i = 0; i < n; i++ )
8     {
9         /*
10          * Positions
11          */
12         fu[ 2 * ( n + i ) ] = _u[ 2 * i ];
13         fu[ 2 * ( n + i ) + 1 ] = _u[ 2 * i + 1 ];
14
15         /*
16          * Velocities
17          */
18         fu[ 2 * i ] = 0.0;
19         fu[ 2 * i + 1 ] = 0.0;
20         const double& q_i_x = _u[ 2 * ( n + i ) ];
21         const double& q_i_y = _u[ 2 * ( n + i ) + 1 ];
22         for( int j = 0; j < n; j++ )
23         {
24             if( i == j )
25                 continue;
26             const double& q_j_x = _u[ 2 * ( n + j ) ];
27             const double& q_j_y = _u[ 2 * ( n + j ) + 1 ];
28             const double q_ij_x = q_i_x - q_j_x;
29             const double q_ij_y = q_i_y - q_j_y;
30             double dist =
31                 sqrt( q_ij_x * q_ij_x + q_ij_y * q_ij_y + epsilon );
32             double coeff =
33                 this->g * this->masses[ j ] / ( dist * dist * dist );
34             fu[ 2 * i ] += coeff * ( q_j_x - q_i_x );
35             fu[ 2 * i + 1 ] += coeff * ( q_j_y - q_i_y );
36         }
37     }
38 }
```

```
1 bool NBodyProblem::writeSolution( const double& t,
2                                   int step,
3                                   const double* u )
4 {
5     /****
6      * Filename with step index
7      */
8     std::stringstream str;
9     str << "nbody-" << std::setw( 5 )
10        << std::setfill( '0' ) << step << ".txt";
11
12     /****
13      * Open file
14      */
15     std::fstream file;
16     file.open( str.str(), std::ios::out );
17     if( ! file )
18     {
19         std::cerr << "Unable to open the file "
20                 << str.str() << std::endl;
21         return false;
22     }
23
24     /****
25      * Write particles positions
26      */
27     const int n = this->particlesCount;
28     const int d = this->dimension;
29     for( int i = 0; i < n; i++ )
30     {
31         for( int j = 0; j < d; j++ )
32             file << u[ d * ( n + i ) + j ] << " ";
33         for( int j = 0; j < d; j++ )
34             file << u[ d * ( n + i ) + j ] + u[ d * i + j ]
35                 << " ";
36         file << std::endl;
37     }
38 }
```

Implementace
řešičů ODR

Riccatiho
rovnice

Experimentální
řád
konvergence

Harmonický
oscilátor

Adaptivní
volba
časového
kroku

Soupeřící
druhy

Lorenzovy
rovnice

Úloha n-těles

```
1 #include <cstdlib>
2 #include "NBodyProblem.h"
3 #include "Euler.h"
4 #include "Merson.h"
5 #include "ode-solve.h"
6
7 using namespace std;
8
9 const double initialTime( 0.0 );
10 const double finalTime( 1.0e3 );
11 const double timeStep( 4.0e-2 );
12 const double integrationTimeStep( 1.0e-6 );
13 const int particlesCount( 100 );
14
15 int main( int argc, char** argv )
16 {
17     NBodyProblem problem( particlesCount );
18
19     Merson integrator;
20     integrator.setAdaptivity( 1.0e-8 );
21
22     double* u = new double[ problem.getDegreesOfFreedom() ];
23     problem.setInitialCondition( u );
24
25     if( ! solve( initialTime,
26                 finalTime,
27                 timeStep,
28                 integrationTimeStep,
29                 &problem,
30                 &integrator,
31                 u ) )
32     {
33         delete[] u;
34         return EXIT_FAILURE;
35     }
36     delete[] u;
37     return EXIT_SUCCESS;
38 }
```