Tomáš Oberhuber

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

Videa na Youtube

Řešiče ODR v C++

Numerické řešení Riccatiho rovnice

Oscilátor, SIR a ostatní

Numerická matematika

Zdrojové kódy s příklady jsou na:
https://gitlab.com/oberhuber.tomas/fjfi-num-src-python.
Stahovat lze pomocí:

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

nebo tlačítkem *Download*.
Rozbalíme příkazem

```
1 tar xvf fjfi-num-src-python*
2 cd fjfi-num-src-python*
```

Řešení ODR

- řešíme úlohu typu:

$$\frac{d\vec{u}}{dt} = \vec{f}(t, \vec{u}(t)) \text{ pro } t \in (0, T\rangle), \quad (1)$$
$$\vec{u}(0) = \vec{u}_{ini}, \quad (2)$$

  kde $\vec{u} : \langle 0, T\rangle \to \mathbb{R}^n$, $\vec{f} : \langle 0, T\rangle \times \mathbb{R}^n \to \mathbb{R}^n$ 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

$$\vec{k}_1 = \tau \vec{f}(t, \vec{u}(t)) \quad (3)$$
$$\vec{u}(t + \tau) = \vec{u}(t) + \vec{k}_1 \quad (4)$$

Řešení ODR

Algoritmicky lze řešič naší úlohy 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**
    $\tau := min\{\tau_0, t_{stop} - t\}$
    $\vec{u} := \vec{u} +$ update(t,$\vec{u}$, $\vec{f}$, τ)
    $t := t + \tau$
  end while
end procedure

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

Řešení ODR

procedure UPDATE(t,$\vec{u}$, $\vec{f}$, 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 má každá úloha tvat:

```
1 class ODEProblem:
2     def __init__(self):
3
4     def get_degrees_of_freedom(self):
5         return ...
6
7     def function_f(self, t, u):
8         ...
```

Popis úlohy

Význam jednotlivých metod:

- `def` **`get_degrees_of_freedom`**`()`
  - vrací počet stupňů volnosti dané úlohy, tj. n pro $\vec{u} \in \langle 0, T \rangle \times \mathbb{R}^n$
- `def` **`function_f`**`( t, u )`
  - napočítává pravou stranu rovnice (2)
  - vektor $\vec{u}$ a výsledek $\vec{f}(t, \vec{u})$ jsou reprezentovány formou cisla nebo pole z Numpy
  - velikost těchto polí je vždy rovna počtu DOF, tj. n

Popis Rungovy-Kuttovy metody

Ve zdrojových kódech má každá Rungova-Kuttova metoda tvar:

```
1  class ODESolver:
2      def __init__(self):
3          self.k1 = None
4          self.k2 = None
5          ...
6
7      def setup(self, degrees_of_freedom):
8          self.k1 = np.zeros(degrees_of_freedom)
9          self.k2 = np.zeros(degrees_of_freedom)
10         ...
11
12     def solve(self, integration_time_step, stop_time, time, problem, x):
13         iteration = 0
14         while time < stop_time:
15             tau = min(integration_time_step, stop_time - time)
16             # Compute k1, k2, ... and update x
17             time += tau
18             iteration += 1
19         return time, x, True
```

Popis Rungovy-Kuttovy metody

Význam jednotlivých metod:

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

Řešení ODR

Funkce pro řešení ODR pak může vypadat takto:

```
def solve_loop(
    initial_time, final_time, time_step,
    integration_time_step, problem, solver, x
):

    solver.setup(problem.get_degrees_of_freedom())
    time_steps_count = math.ceil(max(0.0,
        final_time - initial_time) / time_step)
    time = initial_time
    solution = [(time, x)]  # Store the initial solution

    for step in tqdm(range(1, time_steps_count + 1),
         desc="Getting numerical solution"
    ):
        current_stop_time = time + min(time_step, final_time - time)
        time, x, success = solver.solve(
            integration_time_step, current_stop_time, time, problem, x
        )
        if not success:
            return False, 0.0, 0.0, 0.0  # Return error if solution fails
        solution.append((time, np.array(x)))  # Need to make a copy of x
    return solutions
```

Kód lze najít v souboru `ODE/ODE.py`

Řešení ODR

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

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

Implementace Rungovy-Kuttovy metody prvního řádu

Implementace Eulerova ODE řešiče vypadá takto:

```
1  class Euler:
2      def __init__(self):
3          self.k1 = None
4
5      def setup(self, degrees_of_freedom):
6          self.k1 = np.zeros(degrees_of_freedom)
7
8      def solve(self, integration_time_step, stop_time, time, problem, x):
9          iteration = 0
10         while time < stop_time:
11             tau = min(integration_time_step, stop_time - time)
12             self.k = problem.function_f(time, x)
13             x += tau * self.k1
14             time += tau
15             iteration += 1
16         return time, x, True
```

Implementace Rungovy-Kuttovy metody prvního řádu

- konstruktor `__init__(self)` se volá automaticky při vytvoření instance objektu
  - v našem případě se v něm deklaruje `k1`
- metoda `setup` nastavuje valikost pole `k1` na počet DOFů
- metoda `solve` reprezentuje samotnou Eluerovu metodu

Implementace Rungovy-Kuttovy metody druhého řádu

Domácí úkol

- do souboru `RungeKutta.py` dopište kód pro Rungovu-Kuttovu metodu druhého řádu
- zdrojové kódy vytisknete a odevzdejte

Riccatiho rovnice

Jde o rovnici tvaru:

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

Konkrétněji např.

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

Řešení této rovnice je:

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

pro $c \in \mathbb{R}$.
Tato úloha je implementována v souboru `ODE/Riccati.py`

Riccatiho rovnice

```
1 class Ricatti:
2     def get_degrees_of_freedom(self):
3         return 1
```

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

```
1 def function_f(self, t, u_):
2     u = u_[0]
3     fu = t ** (-4.0) * math.exp(t) + u + 2.0 * math.exp(-t) * u**2
4     return np.array(fu)
```

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

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

Riccatiho rovnice

```
def get_exact_solution(self, t, c=1):
    sqrt_2 = math.sqrt(2.0)
    return math.exp(t) * (
        1.0/(sqrt_2*t**2)*math.tan(sqrt_2*(c-1.0/t))-1.0/(2.0*t)
    )
```

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

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

- budeme ho využívat k napočítaní chyby našich numerických metod

Riccatiho rovnice

```
def get_exact_solutions(self, initial_time, final_time,
          time_step, c=1.0
):
    solutions = []
    t = initial_time
    while t < final_time:
        solutions.append((t, self.get_exact_solution(t, c)))
        t = min(t + time_step, final_time)
    return solutions
```

- metoda `get_exact_solutions` napočítává přesné řešení na intervalu $\langle inital\_time, final\_time \rangle$ s krokem (rozlišením) `time_step` do seznamu solutions

Riccatiho rovnice

```python
def plot_solution(self, exact_solutions, numerical_solutions, text):
    time_values_exact, solution_values_exact = zip(*exact_solutions)
    time_values_numerical, solution_values_numerical = zip(*numerical_solutions)

    plt.figure(figsize=(10, 6))
    plt.plot(
        time_values_exact, solution_values_exact,
        label=f"Exact Solution", color="r"
    )
    plt.plot(
        time_values_numerical,
        solution_values_numerical,
        label=f"Numerical Solution",
        color="orange",
    )
    plt.xlabel("Time")
    plt.ylabel("Solution")
    plt.title(f"{text} Solution of Riccati problem")
    plt.legend()
    plt.grid(True)
    plt.show()
```

- metoda `plot_solution` následně vykreslí numerické i přesné řešení pomoci Matplotlibu

Riccatiho rovnice

```python
if __name__ == "__main__":
    initial_time = 0.1
    final_time = 0.2
    time_step = 1.0e-3
    integration_time_step = 1.0e-4

    problem = Ricatti()

    integrator = Euler()
    # integrator = RK_second_order()
    # integrator = Merson()

    start_x = problem.get_exact_solution(initial_time)

    numerical_solutions = solve_loop(
        initial_time,
        final_time,
        time_step,
        integration_time_step,
        problem,
        integrator,
        [start_x],
    )
    if not numerical_solutions:
        print("Error: Solution failed.")
        exit(1)

    exact_solutions = problem.get_exact_solutions(initial_time,
        final_time, time_step)
    problem.plot_solution(exact_solutions, numerical_solutions,
        "Exact vs Numerical")
```

Chyby aproximace
Jsou definovány jako:

$$\begin{aligned}
\|e\|_{L_1\langle 0,T\rangle} &:= \int_0^T |e(t)|\,dt \approx \sum_{i=0}^{N} |e(i\Delta t)|\,\Delta t, \\
\|e\|_{L_2\langle 0,T\rangle} &:= \left(\int_0^T |e(t)|^2\,dt\right)^{\frac{1}{2}} \approx \left(\sum_{i=0}^{N} |e(i\Delta t)|^2\,\Delta t\right)^2, \\
\|e\|_{L_\infty\langle 0,T\rangle} &:= \sup_{t\in\langle 0,T\rangle} |e(t)| \approx \max_{i\in 0\ldots N} |e(i\Delta t)|\,,
\end{aligned}$$

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

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

pro u označující přesné řešení a u_τ přibližné řešení.

Riccatiho rovnice

Výpočet chyb pak vypadá takto:

```
max_error = 0.0
l1_error = 0.0
l2_error = 0.0
last_t = 0
diff = -1
for time, x in numerical_solutions:
    if diff != -1:
        tau = time - last_t
        l1_error += diff * tau
        l2_error += diff * diff * tau
        max_error = max(max_error, diff)
    exact_solution = problem.get_exact_solution(time)
    diff = abs(exact_solution - x[0])

print("L1 error:", l1_error)
print("L2 error:", math.sqrt(l2_error))
print("Max error:", max_error)
```

Riccatiho rovnice

Domácí úkol

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

EOC

Definition 1

Buď $\tau_1 > \tau_2$, 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:

$$\dot{u}_1 := u_2, \tag{7}$$
$$\dot{u}_2 := -u_1 - \epsilon u_1^2 u_2, \tag{8}$$

s počáteční podmínkou

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

Harmonický oscilátor

Řešič je implementován v souboru `ODE/Hyperbolic.py`

```
1  class HyperbolicProblem:
2      def __init__(self, epsilon=1.0):
3          self.epsilon = epsilon
4
5      def get_degrees_of_freedom(self):
6          return 2
7
8      def function_f(self, time, u, fu=None):
9          u1, u2 = u
10         return np.array([u2, -u1 - self.epsilon * u1**2 * u2])
```

Harmonický oscilátor

```python
if __name__ == "__main__":
    problem = HyperbolicProblem()

    integrator = Euler()
    # integrator = RK_second_order()
    # integrator = Merson()

    problem.epsilon = 0.0
    initial_time = 0.0
    final_time = 100.0
    time_step = 0.1
    integration_time_step = 0.1
    initial_conditions = [0.0, 10.0]

    numerical_solution = solve_loop(
        initial_time,
        final_time,
        time_step,
        integration_time_step,
        problem,
        integrator,
        initial_conditions,
    )
    problem.plot_solution(numerical_solution)
```

Harmonický oscilátor

Example 4

Zmenšujte intergační krok pro simulaci harmonického oscilátoru.

Mersonova-Rungova-Kuttova metoda

procedure MERSONRUNGEKUTTA($\vec{u}$, $\vec{f}$, t_{stop}, τ_0)

$t := 0$

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

$\tau := \tau_0$

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

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

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

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

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

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

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

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

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

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

$t := t + \tau$

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 `Merson.py`.

- parametr ϵ se nastavuje pomocí proměnné `self.adaptivity`
- tento parametr se nastavuje na hodnotu $10^{-3} - 10^{-12}$

Example 5

Proveďte simulaci harmonického oscilátoru s pomocí Mersonovy metody.

Soupeřící druhy

Jde o úlohu typu:

$$\begin{aligned} \dot{u}_1 &= a u_1 - b u_1 u_2, \\ \dot{u}_2 &= -c u_2 + d u_1 u_2, \end{aligned}$$

kde a, b, c a d jsou konstanty.

- jde o typ úloh zvané Volterrovy-Lotkovy úlohy
- simuluje soupeření dvou živočišných druhů - predátor a oběť
- $u : \langle 0, T \rangle \to \mathbb{R}^2$, tj. DOFs = 2
- úloha je implementována v souboru `VolterraLotka.py`

Soupeřící druhy

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:

$$\begin{aligned} \dot{u}_1 &= \sigma(u_2 - u_1), \\ \dot{u}_2 &= \rho u_1 - u_2 - u_1 u_3, \\ \dot{u}_3 &= -\beta u_3 + u_1 u_2, \end{aligned}$$

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

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

Lorenzovy rovnice

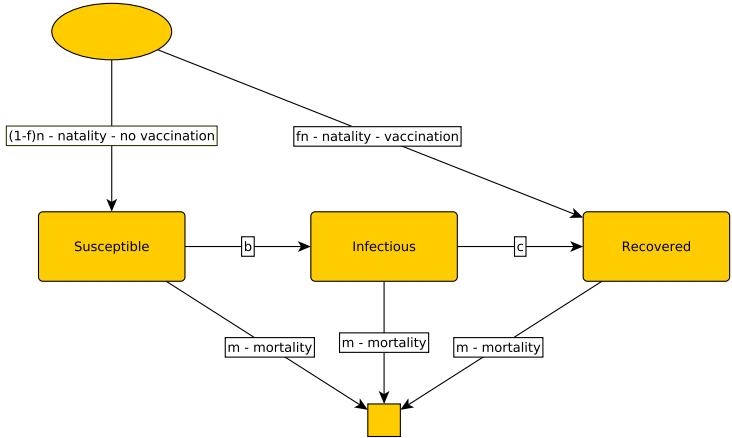
Domácí úkol
Proveďte numerickou studii Lorenzových rovnic a pokuste se najít Lorenzův atraktor.
Napište report podobně jako v předchozích úkolech.

Epidemiologie - SIR model

Tento modelo popisuje šíření infekčních nemocí.

- **S**usceptible - skupina lidí, kteří mohou být nakaženi
- **I**nfectious - skupina infikovaných lidí, kteří mohou šířit nákazu
- **R**ecovered - skupina imuních lidí díky očkování nebo překonání infekce, kteří nemohou být infikováni a nemohou šířit nákazu

Epidemiologie - SIR model



Epidemiologie - SIR model

$$\begin{aligned}
\frac{dS(t)}{dt} &= n(1-f)N - \frac{bI(t)S(t)}{N} - mS(t) \\
\frac{dI(t)}{dt} &= \frac{bI(t)S(t)}{N} - cI(t) - mI(t) \\
\frac{dR(t)}{dt} &= cI(t) + nfN - mR(t),
\end{aligned}$$

kde

- N je velikost populace,
- n je míra porodnosti,
- m je míra úmrtnosti, $m \approx n$,
- b je riziko nakažení neimuních jedinců,
- c je míra uzdravení se nakažených jedinců,
- f je míra pročkování populace
- $\frac{bI(t)S(t)}{N}$ udává počet jedinců nakažených za jednotku času.

Epidemiologie - SIR model

Domácí úkol:

- implementujte tento model
- proveďte výpočetní studii, tj. různá nastevní parametrů
- (na internetu najděte parametry pro reálné epidemie a porovnejte s výpočtem)

Úloha n-těles

Řešíme úlohu:

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

tj.

$$\frac{d^2\vec{p}_i}{dt} = \sum_{\substack{j=1 \\ j\neq i}}^{n} \frac{G m_j}{\left\|\vec{p}_j - \vec{p}_i\right\|} \left(\vec{p}_j - \vec{p}_i\right), i = 1, \ldots n$$

tj.

$$\begin{aligned} \frac{d\vec{v}_i}{dt} &= \sum_{\substack{j=1 \\ j\neq i}}^{n} \frac{G m_j}{\left\|\vec{p}_j - \vec{p}_i\right\|} \left(\vec{p}_j - \vec{p}_i\right), i = 1, \ldots n, \\ \frac{d\vec{p}_i}{dt} &= \vec{v}_i, i = 1, \ldots n. \end{aligned}$$

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 souboruh `NBody.py`.