

# Programowanie, część IV

Rafał J. Wysocki

21 kwietnia 2010

# Równania ruchu

W fizyce często potrzebujemy przewidzieć zachowanie się jakiegoś układu.

# Równania ruchu

W fizyce często potrzebujemy przewidzieć zachowanie się jakiegoś układu.

W tym celu wyznaczamy zmiany jego konfiguracji w czasie przy zadanym stanie początkowym.

# Równania ruchu

W fizyce często potrzebujemy przewidzieć zachowanie się jakiegoś układu.

W tym celu wyznaczamy zmiany jego konfiguracji w czasie przy zadanym stanie początkowym.

Służą nam do tego równania różniczkowe, takie jak równanie Newtona:

$$\frac{d^2}{dt^2}\mathbf{q}(t) = \mathbf{f}\left(\frac{d}{dt}\mathbf{q}(t), \mathbf{q}(t), t\right) \quad (1)$$

gdzie  $\mathbf{q}(t)$  oznacza konfigurację układu w chwili czasu  $t$ , a funkcja  $\mathbf{f}$  reprezentuje siły działające w układzie (z uwzględnieniem mas składników układu).

# Srowadzenie do równania I rzędu

Wprawdzie równanie (1) jest II rzędu, ale można sprowadzić je do postaci równania I rzędu.

# Sprowadzanie do równania I rzędu

Wprawdzie równanie (1) jest II rzędu, ale można sprowadzić je do postaci równania I rzędu.

W tym celu wprowadźmy oznaczenia:

$$\mathbf{v}(t) = \frac{d}{dt}\mathbf{q}(t)$$

$$\mathbf{u}(t) = [\mathbf{q}(t); \mathbf{v}(t)]$$

$$\mathbf{g}(\mathbf{u}(t), t) = [\mathbf{v}(t); \mathbf{f}(\mathbf{v}(t), \mathbf{q}(t), t)]$$

# Srowadzenie do równania I rzędu

Wprowadźmy równanie (1) jest II rzędu, ale można sprowadzić je do postaci równania I rzędu.

W tym celu wprowadźmy oznaczenia:

$$\begin{aligned}\mathbf{v}(t) &= \frac{d}{dt}\mathbf{q}(t) \\ \mathbf{u}(t) &= [\mathbf{q}(t); \mathbf{v}(t)] \\ \mathbf{g}(\mathbf{u}(t), t) &= [\mathbf{v}(t); \mathbf{f}(\mathbf{v}(t), \mathbf{q}(t), t)]\end{aligned}$$

Wtedy następujące równanie jest równoważne równaniu (1).

$$\frac{d}{dt}\mathbf{u}(t) = \mathbf{g}(\mathbf{u}(t), t) \quad (2)$$

# Przybliżenie rozwiązania

Równanie (2) może być ściśle rozwiązane analitycznie tylko w (bardzo) ograniczonej liczbie przypadków.



# Przybliżenie rozwiązania

Równanie (2) może być ściśle rozwiązane analitycznie tylko w (bardzo) ograniczonej liczbie przypadków.

Na szczęście zwykle nie potrzebujemy go ściśle rozwiązywać. Nie musimy nawet znać rozwiązania dla każdej chwili czasu  $t$ .

# Przybliżenie rozwiązania

Równanie (2) może być ściśle rozwiązane analitycznie tylko w (bardzo) ograniczonej liczbie przypadków.

Na szczęście zwykle nie potrzebujemy go ściśle rozwiązywać. Nie musimy nawet znać rozwiązania dla każdej chwili czasu  $t$ .

Zazwyczaj wystarczy wyznaczyć **w przybliżeniu** konfigurację układu w pewnej liczbie chwil czasu  $t_0, t_1, t_2, \dots, t_n$  takich, że  $t_{i+1} = t_i + \Delta t$ , jeśli tylko odstęp czasu  $\Delta t$  jest rozsądnie mały.

# Przybliżenie rozwiązania

Równanie (2) może być ściśle rozwiązane analitycznie tylko w (bardzo) ograniczonej liczbie przypadków.

Na szczęście zwykle nie potrzebujemy go ściśle rozwiązywać. Nie musimy nawet znać rozwiązania dla każdej chwili czasu  $t$ .

Zazwyczaj wystarczy wyznaczyć **w przybliżeniu** konfigurację układu w pewnej liczbie chwil czasu  $t_0, t_1, t_2, \dots, t_n$  takich, że  $t_{i+1} = t_i + \Delta t$ , jeśli tylko odstęp czasu  $\Delta t$  jest rozsądnie mały.

Wystarczy zatem wyznaczyć ciąg wektorów  $\mathbf{u}_i$  takich, że dla każdego  $i$  zachodzi  $\|\mathbf{u}(t_i) - \mathbf{u}_i\| < \epsilon$ , gdzie *epsilon* jest zadany limitem odchylenia obliczonego wektora od „prawdziwej” konfiguracji układu w danej chwili czasu.

# Metoda Eulera

Zauważmy, że

$$\mathbf{u}(t_{i+1}) - \mathbf{u}(t_i) = \int_{t_i}^{t_{i+1}} \mathbf{g}(\mathbf{u}(\tau), \tau) d\tau \quad (3)$$

# Metoda Eulera

Zauważmy, że

$$\mathbf{u}(t_{i+1}) - \mathbf{u}(t_i) = \int_{t_i}^{t_{i+1}} \mathbf{g}(\mathbf{u}(\tau), \tau) d\tau \quad (3)$$

Jeżeli  $\mathbf{u}(t)$  jest funkcją o wartościach rzeczywistych, to w pierwszym przybliżeniu można oszacować całkę po prawej stronie równania (3) przez pole powierzchni prostokąta o wysokości  $\mathbf{g}(\mathbf{u}(t_i), t_i)$  i długości podstawy  $\Delta t$ .

# Metoda Eulera

Zauważmy, że

$$\mathbf{u}(t_{i+1}) - \mathbf{u}(t_i) = \int_{t_i}^{t_{i+1}} \mathbf{g}(\mathbf{u}(\tau), \tau) d\tau \quad (3)$$

Jeżeli  $\mathbf{u}(t)$  jest funkcją o wartościach rzeczywistych, to w pierwszym przybliżeniu można oszacować całkę po prawej stronie równania (3) przez pole powierzchni prostokąta o wysokości  $\mathbf{g}(\mathbf{u}(t_i), t_i)$  i długości podstawy  $\Delta t$ .

Stąd w ogólności otrzymujemy algorytm

$$\mathbf{u}_{i+1} = \mathbf{u}_i + \mathbf{g}(\mathbf{u}_i, t_i)\Delta t$$

zwany **metodą Eulera**.

# Wady metody Eulera

W każdym kroku metody Eulera odstępstwo od „prawdziwej” wartości  $\mathbf{u}(t_i)$  może być rzędu  $(\Delta t)^2$ :

$$\begin{aligned}\mathbf{u}(t_{i+1}) &= \mathbf{u}(t_i) + \Delta t \left. \frac{d\mathbf{u}}{dt} \right|_{t_i} + O((\Delta t)^2) \\ &= \mathbf{u}(t_i) + \mathbf{g}(\mathbf{u}(t_i), t_i) \Delta t + O((\Delta t)^2)\end{aligned}$$

# Wady metody Eulera

W każdym kroku metody Eulera odstępstwo od „prawdziwej” wartości  $\mathbf{u}(t_i)$  może być rzędu  $(\Delta t)^2$ :

$$\begin{aligned}\mathbf{u}(t_{i+1}) &= \mathbf{u}(t_i) + \Delta t \left. \frac{d\mathbf{u}}{dt} \right|_{t_i} + O((\Delta t)^2) \\ &= \mathbf{u}(t_i) + \mathbf{g}(\mathbf{u}(t_i), t_i) \Delta t + O((\Delta t)^2)\end{aligned}$$

Ponadto w ogólności jest ona **niestabilna**, czyli całkowite odstępstwo  $\mathbf{u}_i$  od  $\mathbf{u}(t_i)$  może stopniowo **narastać** wraz z liczbą wykonanych kroków.



# Wady metody Eulera

W każdym kroku metody Eulera odstępstwo od „prawdziwej” wartości  $\mathbf{u}(t_i)$  może być rzędu  $(\Delta t)^2$ :

$$\begin{aligned}\mathbf{u}(t_{i+1}) &= \mathbf{u}(t_i) + \Delta t \left. \frac{d\mathbf{u}}{dt} \right|_{t_i} + O((\Delta t)^2) \\ &= \mathbf{u}(t_i) + \mathbf{g}(\mathbf{u}(t_i), t_i) \Delta t + O((\Delta t)^2)\end{aligned}$$

Ponadto w ogólności jest ona **niestabilna**, czyli całkowite odstępstwo  $\mathbf{u}_i$  od  $\mathbf{u}(t_i)$  może stopniowo **narastać** wraz z liczbą wykonanych kroków.

W celu otrzymania bardziej praktycznej metody można spróbować użyć lepszego przybliżenia całki po prawej stronie równania (3).

# Poprawianie metody Eulera

Wysokość prostokąta, którego pola powierzchni używamy do oszacowania całki w równaniu (3), można wyznaczyć wykorzystując przybliżoną wartość funkcji  $\mathbf{g}(\mathbf{u}(t), t)$  w punkcie  $t_i + \Delta t/2$ .

# Poprawianie metody Eulera

Wysokość prostokąta, którego pola powierzchni używamy do oszacowania całki w równaniu (3), można wyznaczyć wykorzystując przybliżoną wartość funkcji  $\mathbf{g}(\mathbf{u}(t), t)$  w punkcie  $t_i + \Delta t/2$ .

Weźmy:

$$\mathbf{k}_1 = \mathbf{g}(\mathbf{u}_i, t_i)$$

$$\mathbf{k}_2 = \mathbf{g}(\mathbf{u}_i + \alpha \mathbf{k}_1 \Delta t, t_i + \beta \Delta t)$$

$$\mathbf{u}_{i+1} = \mathbf{u}_i + \omega_1 \mathbf{k}_1 \Delta t + \omega_2 \mathbf{k}_2 \Delta t$$

i wybierzmy stałe  $\alpha$ ,  $\beta$ ,  $\omega_1$ ,  $\omega_2$  tak, aby różnica między  $\mathbf{u}_{i+1}$  oraz  $\mathbf{u}(t_{i+1})$  była rzędu  $(\Delta t)^3$  (przy założeniu, że  $\mathbf{u}_i = \mathbf{u}(t_i)$ ).

# Poprawianie metody Eulera c. d.

Z rozwinięcia  $\mathbf{u}(t)$  w szereg Taylora w otoczeniu  $t_i$  mamy:

$$\begin{aligned}\mathbf{u}(t_{i+1}) &= \mathbf{u}(t_i) + \mathbf{g}(\mathbf{u}(t_i), t_i)\Delta t + \left. \frac{\partial \mathbf{g}}{\partial t} \right|_{t_i} \frac{(\Delta t)^2}{2} \\ &\quad + \left. \frac{\partial \mathbf{g}}{\partial \mathbf{u}} \right|_{t_i} \mathbf{g}(\mathbf{u}(t_i), t_i) \frac{(\Delta t)^2}{2} + O((\Delta t)^3)\end{aligned}$$

# Poprawianie metody Eulera c. d.

Z rozwinięcia  $\mathbf{u}(t)$  w szereg Taylora w otoczeniu  $t_i$  mamy:

$$\begin{aligned} \mathbf{u}(t_{i+1}) = & \mathbf{u}(t_i) + \mathbf{g}(\mathbf{u}(t_i), t_i)\Delta t + \left. \frac{\partial \mathbf{g}}{\partial t} \right|_{t_i} \frac{(\Delta t)^2}{2} \\ & + \left. \frac{\partial \mathbf{g}}{\partial \mathbf{u}} \right|_{t_i} \mathbf{g}(\mathbf{u}(t_i), t_i) \frac{(\Delta t)^2}{2} + O((\Delta t)^3) \end{aligned}$$

Rozwijając  $\mathbf{k}_2$  w szereg Taylora w otoczeniu  $t_i$  dostajemy:

$$\begin{aligned} \mathbf{u}_{i+1} = & \mathbf{u}_i + \mathbf{g}(\mathbf{u}_i, t_i)(\omega_1 + \omega_2)\Delta t + \left. \frac{\partial \mathbf{g}}{\partial t} \right|_{t_i} \beta\omega_2(\Delta t)^2 \\ & + \left. \frac{\partial \mathbf{g}}{\partial \mathbf{u}} \right|_{t_i} \mathbf{g}(\mathbf{u}_i, t_i)\alpha\omega_2(\Delta t)^2 + O((\Delta t)^3) \end{aligned}$$

# Metoda punktu środkowego (*ang. midpoint method*)

Oczekiwany efekt uzyskamy wybierając  $\omega_1 = 1 - \omega_2$ ,  $\beta\omega_2 = 1/2$  oraz  $\alpha\omega_2 = 1/2$ .

# Metoda punktu środkowego (*ang. midpoint method*)

Oczekiwany efekt uzyskamy wybierając  $\omega_1 = 1 - \omega_2$ ,  $\beta\omega_2 = 1/2$  oraz  $\alpha\omega_2 = 1/2$ .

W szczególności możemy wziąć  $\omega_1 = 0$ ,  $\omega_2 = 1$ ,  $\alpha = 1/2$  oraz  $\beta = 1/2$  i wtedy otrzymujemy algorytm:

$$\begin{aligned}\mathbf{k}_1 &= \mathbf{g}(\mathbf{u}_i, t_i) \\ \mathbf{k}_2 &= \mathbf{g}\left(\mathbf{u}_i + \frac{\Delta t}{2}\mathbf{k}_1, t_i + \frac{\Delta t}{2}\right) \Delta t \\ \mathbf{u}_{i+1} &= \mathbf{u}_i + \mathbf{k}_2\Delta t\end{aligned}$$

# Metoda punktu środkowego (*ang. midpoint method*)

Oczekiwany efekt uzyskamy wybierając  $\omega_1 = 1 - \omega_2$ ,  $\beta\omega_2 = 1/2$  oraz  $\alpha\omega_2 = 1/2$ .

W szczególności możemy wziąć  $\omega_1 = 0$ ,  $\omega_2 = 1$ ,  $\alpha = 1/2$  oraz  $\beta = 1/2$  i wtedy otrzymujemy algorytm:

$$\begin{aligned}\mathbf{k}_1 &= \mathbf{g}(\mathbf{u}_i, t_i) \\ \mathbf{k}_2 &= \mathbf{g}\left(\mathbf{u}_i + \frac{\Delta t}{2}\mathbf{k}_1, t_i + \frac{\Delta t}{2}\right) \Delta t \\ \mathbf{u}_{i+1} &= \mathbf{u}_i + \mathbf{k}_2\Delta t\end{aligned}$$

Nie musimy jednak poprzestawać na wykorzystywaniu dwóch pośrednich wartości  $\mathbf{g}(\mathbf{u}(t), t)$ .



# Wyprowadzenie metody Rungego-Kutty

Użyjmy czterech pośrednich wartości  $\mathbf{g}(\mathbf{u}(t), t)$ :

$$\mathbf{k}_1 = \mathbf{g}(\mathbf{u}_i, t_i)$$

$$\mathbf{k}_2 = \mathbf{g}(\mathbf{u}_i + \alpha_1 \mathbf{k}_1 \Delta t, t_i + \beta_1 \Delta t)$$

$$\mathbf{k}_3 = \mathbf{g}(\mathbf{u}_i + \alpha_2 \mathbf{k}_2 \Delta t, t_i + \beta_2 \Delta t)$$

$$\mathbf{k}_4 = \mathbf{g}(\mathbf{u}_i + \alpha_3 \mathbf{k}_3 \Delta t, t_i + \beta_3 \Delta t)$$

$$\mathbf{u}_{i+1} = \mathbf{u}_i + \Delta t \sum_{j=1}^4 \omega_j \mathbf{k}_j \quad (4)$$

# Wyprowadzenie metody Rungego-Kutty

Użyjmy czterech pośrednich wartości  $\mathbf{g}(\mathbf{u}(t), t)$ :

$$\begin{aligned}\mathbf{k}_1 &= \mathbf{g}(\mathbf{u}_i, t_i) \\ \mathbf{k}_2 &= \mathbf{g}(\mathbf{u}_i + \alpha_1 \mathbf{k}_1 \Delta t, t_i + \beta_1 \Delta t) \\ \mathbf{k}_3 &= \mathbf{g}(\mathbf{u}_i + \alpha_2 \mathbf{k}_2 \Delta t, t_i + \beta_2 \Delta t) \\ \mathbf{k}_4 &= \mathbf{g}(\mathbf{u}_i + \alpha_3 \mathbf{k}_3 \Delta t, t_i + \beta_3 \Delta t) \\ \mathbf{u}_{i+1} &= \mathbf{u}_i + \Delta t \sum_{j=1}^4 \omega_j \mathbf{k}_j\end{aligned}\quad (4)$$

Postępując analogicznie do poprzedniego przypadku możemy otrzymać zgodność prawej strony równania (4) z rozwinięciem Taylora  $\mathbf{u}(t)$  w otoczeniu  $t_i$  z dokładnością do wyrazów rzędu  $(\Delta t)^4$  przy założeniu, że  $\mathbf{u}_i = \mathbf{u}(t_i)$ .

# Metoda Rungego-Kutty (czwartego rzędu)

W tym celu trzeba wybrać  $\alpha_1 = \beta_1 = \alpha_2 = \beta_2 = 1/2$ ,  
 $\alpha_3 = \beta_3 = 1$ ,  $\omega_1 = \omega_4 = 1/6$  oraz  $\omega_2 = \omega_3 = 1/3$ .

# Metoda Rungego-Kutty (czwartego rzędu)

W tym celu trzeba wybrać  $\alpha_1 = \beta_1 = \alpha_2 = \beta_2 = 1/2$ ,  
 $\alpha_3 = \beta_3 = 1$ ,  $\omega_1 = \omega_4 = 1/6$  oraz  $\omega_2 = \omega_3 = 1/3$ .

Wówczas otrzymujemy algorytm:

$$\begin{aligned} \mathbf{k}_1 &= \mathbf{g}(\mathbf{u}_i, t_i) \\ \mathbf{k}_2 &= \mathbf{g}\left(\mathbf{u}_i + \frac{\Delta t}{2}\mathbf{k}_1, t_i + \frac{\Delta t}{2}\right) \\ \mathbf{k}_3 &= \mathbf{g}\left(\mathbf{u}_i + \frac{\Delta t}{2}\mathbf{k}_2, t_i + \frac{\Delta t}{2}\right) \\ \mathbf{k}_4 &= \mathbf{g}(\mathbf{u}_i + \mathbf{k}_3\Delta t, t_i + \Delta t) \\ \mathbf{u}_{i+1} &= \mathbf{u}_i + \Delta t \left( \frac{\mathbf{k}_1}{6} + \frac{\mathbf{k}_2}{3} + \frac{\mathbf{k}_3}{3} + \frac{\mathbf{k}_4}{6} \right) \end{aligned}$$

# Metoda Rungego-Kutty (czwartego rzędu) c. d.

W metodzie Rungego-Kutty czwartego rzędu różnica między  $\mathbf{u}_i$  oraz  $\mathbf{u}(t_i)$  w pojedynczym kroku jest rzędu  $(\Delta t)^5$ .

# Metoda Rungego-Kutty (czwartego rzędu) c. d.

W metodzie Rungego-Kutty czwartego rzędu różnica między  $\mathbf{u}_i$  oraz  $\mathbf{u}(t_i)$  w pojedynczym kroku jest rzędu  $(\Delta t)^5$ .

Ponadto metoda jest stabilna, czyli całkowite odstępstwo  $\mathbf{u}_i$  od „prawdziwych” wartości  $\mathbf{u}(t_i)$  jest ograniczone. Inaczej mówiąc istnieje dodatnia liczba  $\epsilon$  taka, że dla każdego  $i$  zachodzi  $\|\mathbf{u}_i - \mathbf{u}(t_i)\| < \epsilon$ .

# Metoda Rungego-Kutty (czwartego rzędu) c. d.

W metodzie Rungego-Kutty czwartego rzędu różnica między  $\mathbf{u}_i$  oraz  $\mathbf{u}(t_i)$  w pojedynczym kroku jest rzędu  $(\Delta t)^5$ .

Ponadto metoda jest stabilna, czyli całkowite odstępstwo  $\mathbf{u}_i$  od „prawdziwych” wartości  $\mathbf{u}(t_i)$  jest ograniczone. Inaczej mówiąc istnieje dodatnia liczba  $\epsilon$  taka, że dla każdego  $i$  zachodzi  $\|\mathbf{u}_i - \mathbf{u}(t_i)\| < \epsilon$ .

Wartość  $\epsilon$  zależy jednak od funkcji  $\mathbf{g}(\mathbf{u}(t), t)$  po prawej stronie równania różniczkowego.

# Założenia programu realizującego algorytm

Aby zapisać algorytm Rungego-Kutty w postaci programu zauważmy, że do przeprowadzenia obliczeń dla jednego kroku metody potrzebny jest wektor reprezentujący wartość  $\mathbf{u}_i$  oraz wektory, w których będzie można zapisać  $\mathbf{k}_1$ ,  $\mathbf{k}_2$ ,  $\mathbf{k}_3$  i  $\mathbf{k}_4$ .



## Założenia programu realizującego algorytm

Aby zapisać algorytm Rungego-Kutty w postaci programu zauważmy, że do przeprowadzenia obliczeń dla jednego kroku metody potrzebny jest wektor reprezentujący wartość  $\mathbf{u}_i$  oraz wektory, w których będzie można zapisać  $\mathbf{k}_1$ ,  $\mathbf{k}_2$ ,  $\mathbf{k}_3$  i  $\mathbf{k}_4$ .

Wynik obliczeń, czyli  $\mathbf{u}_{i+1}$ , można zapisać w tym samym miejscu, w którym przechowywana była wartość  $\mathbf{u}_i$  z poprzedniego kroku. Ponadto wektory służące do przechowywania  $\mathbf{k}_j$  mogą być wykorzystane w następnym kroku (rezerwowanie ich przed wykonaniem każdego kroku i zwalnianie po jego wykonaniu nie byłoby efektywne).

## Założenia programu realizującego algorytm

Aby zapisać algorytm Rungego-Kutty w postaci programu zauważmy, że do przeprowadzenia obliczeń dla jednego kroku metody potrzebny jest wektor reprezentujący wartość  $\mathbf{u}_i$  oraz wektory, w których będzie można zapisać  $\mathbf{k}_1$ ,  $\mathbf{k}_2$ ,  $\mathbf{k}_3$  i  $\mathbf{k}_4$ .

Wynik obliczeń, czyli  $\mathbf{u}_{i+1}$ , można zapisać w tym samym miejscu, w którym przechowywana była wartość  $\mathbf{u}_i$  z poprzedniego kroku. Ponadto wektory służące do przechowywania  $\mathbf{k}_j$  mogą być wykorzystane w następnym kroku (rezerwowanie ich przed wykonaniem każdego kroku i zwalnianie po jego wykonaniu nie byłoby efektywne).

Dlatego wygodnie jest tak skonstruować program, aby te wektory były polami pewnego obiektu.

## Założenia programu realizującego algorytm c. d.

Ułatwia to także wyznaczanie przybliżenia rozwiązania dla wielu równań jednocześnie (można zdefiniować odpowiedni obiekt dla każdego równania).

## Założenia programu realizującego algorytm c. d.

Ułatwia to także wyznaczanie przybliżenia rozwiązania dla wielu równań jednocześnie (można zdefiniować odpowiedni obiekt dla każdego równania).

Obiekt, którego będziemy używać do obliczeń, będzie zawsze przeprowadzał podobne operacje, niezależnie od tego jaka funkcja  $\mathbf{g}(\mathbf{u}(t), t)$  znajduje się po prawej stronie równania (zawsze trzeba będzie wyznaczyć  $\mathbf{k}_j$  na podstawie odpowiednich wzorów i obliczyć ich sumę).

## Założenia programu realizującego algorytm c. d.

Ułatwia to także wyznaczenie przybliżenia rozwiązania dla wielu równań jednocześnie (można zdefiniować odpowiedni obiekt dla każdego równania).

Obiekt, którego będziemy używać do obliczeń, będzie zawsze przeprowadzał podobne operacje, niezależnie od tego jaka funkcja  $\mathbf{g}(\mathbf{u}(t), t)$  znajduje się po prawej stronie równania (zawsze trzeba będzie wyznaczyć  $\mathbf{k}_j$  na podstawie odpowiednich wzorów i obliczyć ich sumę).

W związku z tym wygodnie jest tak zdefiniować klasę dla tego obiektu, aby funkcja  $\mathbf{g}(\mathbf{u}(t), t)$  była przekazywana do obiektu poprzez konstruktor.

# Wskaźniki do funkcji – deklaracja

## Wskaźnik do funkcji (*ang. function pointer*)

Jest to zmienna, w której zapisuje się adres funkcji. Dla tego rodzaju wskaźników typ danych wskazywanego obiektu określa:

- 1 jaki jest typ danych dla wyniku zwracanego przez funkcję,
- 2 jakie są typy danych argumentów tej funkcji oraz ich liczba.

# Wskaźniki do funkcji – deklaracja

## Wskaźnik do funkcji (*ang. function pointer*)

Jest to zmienna, w której zapisuje się adres funkcji. Dla tego rodzaju wskaźników typ danych wskazywanego obiektu określa:

- 1 jaki jest typ danych dla wyniku zwracanego przez funkcję,
- 2 jakie są typy danych argumentów tej funkcji oraz ich liczba.

Wskaźnik do funkcji zwracającej wynik typu `double` o dwóch argumentach – jednym typu `double` i drugim typu `int` – deklaruje się w następujący sposób:

```
double (*funkcja)(double, int);
```

Wtedy nazwą zmiennej dla tego wskaźnika jest `funkcja`.

## Wskaźniki do funkcji – nadawanie wartości

Mając następujący wskaźnik do funkcji:

```
double (*fun)(double); // Argument i wynik typu double.
```

można przypisać mu funkcję `sin()` z biblioteki standardowej:

```
fun = sin; // Po prawej stronie jest nazwa funkcji.
```



## Wskaźniki do funkcji – nadawanie wartości

Mając następujący wskaźnik do funkcji:

```
double (*fun)(double); // Argument i wynik typu double.
```

można przypisać mu funkcję `sin()` z biblioteki standardowej:

```
fun = sin; // Po prawej stronie jest nazwa funkcji.
```

Wtedy instrukcja

```
cout << fun(M_PI) << endl;
```

spowoduje wydrukowanie wartości funkcji `sin(M_PI)`.

# Klasa dla algorytmu Rungego-Kutty

Korzystając ze wskaźnika do funkcji można zdefiniować uniwersalną klasę dla algorytmu Rungego-Kutty.

```
class RungeKutta {
private:
    double t;                // bieżący czas
    vector<double> u;        // bieżąca konfiguracja
    vector<double> v2, v3, v4; // pomocnicze
    double dt;              // krok
    int n;                  // liczba współrzędnych
    void (*G)(const vector<double>& u, double t, vector<double>& out);
    // G() reprezentuje prawą stronę równania. Argument 'u' jest
    // bieżącą konfiguracją układu, a wynik działania funkcji na 'u'
    // jest zapisywany w wektorze 'out'.

public:
    RungeKutta(const vector<double> u0, double t0, double d,
               void (*fun)(const vector<double>&, double, vector<double>&));
    double getT(void) const { return t; }
    vector<double> getU(void) const { return u; }
    void wykonaj_krok(void);
};
```

# Konstruktor dla klasy RungeKutta

```
RungeKutta::RungeKutta(  
    const vector<double> u0, double t0, double d,  
    void (*fun)(const vector<double>&, double, vector<double>&))  
{  
    n = u0.size();  
    u = u0; // Początkowa konfiguracja.  
    v2.reserve(n); // Jawną deklaracją liczby współrzędnych.  
    v3.reserve(n);  
    v4.reserve(n);  
    t = t0;  
    dt = d;  
    G = fun;  
}
```

# Konstruktor dla klasy RungeKutta

```
RungeKutta::RungeKutta(  
    const vector<double> u0, double t0, double d,  
    void (*fun)(const vector<double>&, double, vector<double>&))  
{  
    n = u0.size();  
    u = u0; // Początkowa konfiguracja.  
    v2.reserve(n); // Jawna deklaracja liczby współrzędnych.  
    v3.reserve(n);  
    v4.reserve(n);  
    t = t0;  
    dt = d;  
    G = fun;  
}
```

Dla wektorów  $v_2$ ,  $v_3$ ,  $v_4$  trzeba jawnie zadeklarować liczbę współrzędnych (w przeciwnym razie w funkcji `wykonaj_krok` będą występować błędy związane z dostępem do pamięci).

# Konstruktor dla klasy RungeKutta

```
RungeKutta::RungeKutta(  
    const vector<double> u0, double t0, double d,  
    void (*fun)(const vector<double>&, double, vector<double>&))  
{  
    n = u0.size();  
    u = u0; // Początkowa konfiguracja.  
    v2.reserve(n); // Jawna deklaracja liczby współrzędnych.  
    v3.reserve(n);  
    v4.reserve(n);  
    t = t0;  
    dt = d;  
    G = fun;  
}
```

Dla wektorów  $v_2$ ,  $v_3$ ,  $v_4$  trzeba jawnie zadeklarować liczbę współrzędnych (w przeciwnym razie w funkcji `wykonaj_krok` będą występować błędy związane z dostępem do pamięci).

Obliczenia realizujące algorytm zawiera metoda `wykonaj_krok()`.

```
void RungeKutta::wykonaj_krok(void)
{
    const double jedna_trzecia = 1.0 / 3.0;
    double dt_2 = 0.5 * dt;
    int i;

    v3 = u;
    G(u, t, v2); // v2 zawiera wartość 'k1'
    for (i = 0; i < n; i++) {
        v2[i] *= dt_2;
        v4[i] = u[i] + v2[i]; // u + (h/2) * 'k1'
        u[i] += jedna_trzecia * v2[i]; // u += (h/6) * 'k1'
    }
    G(v4, t + dt_2, v2); // v2 zawiera wartość 'k2'
    for (i = 0; i < n; i++) {
        v2[i] *= dt;
        v4[i] = v3[i] + 0.5 * v2[i]; // u + (h/2) * 'k2'
        u[i] += jedna_trzecia * v2[i]; // u += (h/3) * 'k2'
    }
    G(v4, t + dt_2, v2); // v2 zawiera wartość 'k3'
    for (i = 0; i < n; i++) {
        v2[i] *= dt;
        v4[i] = v3[i] + v2[i]; // u + 'k3'
        u[i] += jedna_trzecia * v2[i]; // u += (h/3) * 'k3'
    }
    t += dt;
    G(v4, t, v2); // v2 zawiera wartość 'k4'
    for (i = 0; i < n; i++)
        u[i] += dt_2 * jedna_trzecia * v2[i]; // u += (h/6) * 'k4'
}
```

## Przykład: oscylator harmoniczny

Zdefiniowaną klasę RungeKutta można wykorzystać do wyznaczenia (w przybliżeniu) ruchu oscylatora harmonicznego. W tym celu trzeba przygotować funkcję, która zostanie przekazana do konstruktora obiektu.

# Przykład: oscylator harmoniczny

Zdefiniowaną klasę RungeKutta można wykorzystać do wyznaczenia (w przybliżeniu) ruchu oscylatora harmonicznego. W tym celu trzeba przygotować funkcję, która zostanie przekazana do konstruktora obiektu.

```
static void osc(const vector<double>& y, double x, vector<double>& out)
{
    static const double K = 1.0;

    out[0] = y[1];           // Pochodna położenia.
    out[1] = - K * y[0];    // Pochodna prędkości.

    (void)x; // Zbędny parametr.
}
```



## Przykład: oscylator harmoniczny – obliczenia

Mając do dyspozycji klasę `RungeKutta` oraz funkcję `osc()` zdefiniowane jak wyżej można przeprowadzić obliczenia.

# Przykład: oscylator harmoniczny – obliczenia

Mając do dyspozycji klasę RungeKutta oraz funkcję `osc()` zdefiniowane jak wyżej można przeprowadzić obliczenia.

```
vector<double> y;  
  
// Warunek początkowy.  
y.push_back(1.0); // położenie  
y.push_back(0.0); // prędkość  
  
RungeKutta oscylator(y, 0, 0.01, osc);  
  
for (int i = 0; i < KROKI; i++) {  
    double t = oscylator.getT();  
    y = oscylator.getU();  
    cout << t << "\t" << y[0] << "\t" << y[1] << endl;  
  
    oscylator.wykonaj_krok();  
}
```

## Przykład: oscylator harmoniczny – obliczenia

Mając do dyspozycji klasę RungeKutta oraz funkcję `osc()` zdefiniowane jak wyżej można przeprowadzić obliczenia.

```
vector<double> y;  
  
// Warunek początkowy.  
y.push_back(1.0); // położenie  
y.push_back(0.0); // prędkość  
  
RungeKutta oscylator(y, 0, 0.01, osc);  
  
for (int i = 0; i < KROKI; i++) {  
    double t = oscylator.getT();  
    y = oscylator.getU();  
    cout << t << "\t" << y[0] << "\t" << y[1] << endl;  
  
    oscylator.wykonaj_krok();  
}
```

Wynik zostanie zapisany na standardowe wyjście w postaci trzech kolumn liczb: czas, położenie i prędkość oscylatora.

# Abstrakcyjna klasa dla algorytmu Rungego-Kutty

Zamist postugiwać się wskaźnikiem do funkcji, można wykorzystać mechanizm dziedziczenia.

# Abstrakcyjna klasa dla algorytmu Rungego-Kutty

Zamist posługiwać się wskaźnikiem do funkcji, można wykorzystać mechanizm dziedziczenia.

Zaczynamy od zaprojektowania abstrakcyjnej klasy.

```
class RungeKutta {
private:
    double t;                // bieżący czas
    vector<double> u;        // bieżąca konfiguracja
    vector<double> v2, v3, v4; // pomocnicze
    double dt;              // krok
    int n;                  // liczba współrzędnych

protected:
    virtual void G(const vector<double>& in, double t, vector<double>& out) = 0;

public:
    RungeKutta(const vector<double> u0, double t0, double d);
    double getT(void) const { return t; }
    vector<double> getU(void) const { return u; }
    void wykonaj_krok(void);
};
```

# Konstruktor i metoda wykonaj\_krok()

Konstruktor dla klasy RungeKutta ma teraz mniej argumentów.

```
RungeKutta::RungeKutta(const vector<double> u0, double t0, double d)
{
    n = u0.size();
    u = u0; // Początkowa konfiguracja.
    v2.reserve(n); // Jawną deklaracja liczby współrzędnych.
    v3.reserve(n);
    v4.reserve(n);
    t = t0;
    dt = d;
}
```

# Konstruktor i metoda wykonaj\_krok()

Konstruktor dla klasy RungeKutta ma teraz mniej argumentów.

```
RungeKutta::RungeKutta(const vector<double> u0, double t0, double d)
{
    n = u0.size();
    u = u0; // Początkowa konfiguracja.
    v2.reserve(n); // Jawną deklaracja liczby współrzędnych.
    v3.reserve(n);
    v4.reserve(n);
    t = t0;
    dt = d;
}
```

Metoda wykonaj\_krok() pozostaje bez zmian w zapisie, choć zamiast funkcji wskazywanej przez wskaźnik wywołuje wirtualną metodę G().

## Przykład: oscylator harmoniczny – klasa pochodna

W celu zdefiniowania metody  $G()$  odpowiedniej dla naszych obliczeń trzeba stworzyć klasę pochodną od klasy RungeKutta.



# Przykład: oscylator harmoniczny – klasa pochodna

W celu zdefiniowania metody  $G()$  odpowiedniej dla naszych obliczeń trzeba stworzyć klasę pochodną od klasy RungeKutta.

```
#include "rk.h"

class Oscylator : public RungeKutta {
protected:
    void G(const vector <double> &y, double x, vector<double> &out);

public:
    Oscylator(const vector<double> u0, double t0, double d) :
        RungeKutta(u0, t0, d) {}
};

void Oscylator::G(const vector <double> &y, double x, vector<double> &out)
{
    static const double K = 1.0;

    out[0] = y[1]; // pochodna położenia
    out[1] = - K * y[0]; // pochodna prędkości

    (void)x;
}
```

# Przykład: oscylator harmoniczny – obliczenia

Obliczenia przeprowadzamy podobnie, jak w poprzednim przypadku, tylko teraz posługujemy się obiektem klasy `Oscylator`.

```
vector<double> y(1.0, 0.0);

// Warunek początkowy
y.push_back(1.0); // położenie
y.push_back(0.0); // prędkość

Oscylator oscylator(y, 0, 0.01);

for (int i = 0; i < KROKI; i++) {
    y = oscylator.getU();
    cout << oscylator.getT() << "\t"
         << y[0] << "\t" << y[1] << endl;

    oscylator.wykonaj_krok();
}
```

## Przykład: oscylator harmoniczny z tarciem

Dla innego modelu (np. oscylatora z tarciem) tworzymy inną klasę pochodną od klasy `RungeKutta`.

# Przykład: oscylator harmoniczny z tarciem

Dla innego modelu (np. oscylatora z tarciem) tworzymy inną klasę pochodną od klasy RungeKutta.

```
#include "rk.h"

class OscylatorTarcie : public RungeKutta {
protected:
    void G(const vector <double> &y, double x, vector<double> &out);

public:
    OscylatorTarcie(const vector<double> u0, double t0, double d) :
        RungeKutta(u0, t0, d) {}
};

void OscylatorTarcie::G(const vector <double> &y, double x, vector<double> &out)
{
    static const double K = 1.0;
    static const double f = 0.3;

    out[0] = y[1]; // pochodna położenia
    out[1] = - K * y[0] - f * y[1]; // pochodna prędkości

    (void)x;
}
```

## Przykład: huśtawka

Spróbujmy obliczyć jaka powinna być prędkość początkowa (sztywnej) huśtawki o długości  $l$ , na której umieścimy ciężar o masie  $m$ , aby osiągnęła ona wychylenie  $90^\circ$ .

## Przykład: huśtawka

Spróbujmy obliczyć jaka powinna być prędkość początkowa (sztywnej) huśtawki o długości  $l$ , na której umieścimy ciężar o masie  $m$ , aby osiągnęła ona wychylenie  $90^\circ$ .

Zadanie to można łatwo rozwiązać korzystając z zasady zachowania energii **jeżeli nie uwzględnia się tarcia**. My jednak uwzględnimy tarcie.

## Przykład: huśtawka

Spróbujmy obliczyć jaka powinna być prędkość początkowa (sztywnej) huśtawki o długości  $l$ , na której umieścimy ciężar o masie  $m$ , aby osiągnęła ona wychylenie  $90^\circ$ .

Zadanie to można łatwo rozwiązać korzystając z zasady zachowania energii **jeżeli nie uwzględnia się tarcia**. My jednak uwzględnimy tarcie.

Ponadto nie będziemy (bo nie możemy) korzystać z założenia, że wychylenie huśtawki jest „małe”.

# Opis ruchu

Położenie ciężaru na huśtawce zawsze znajduje się na okręgu o promieniu  $l$  i środku w punkcie zaczepienia huśtawki.



# Opis ruchu

Położenie ciężaru na huśtawce zawsze znajduje się na okręgu o promieniu  $l$  i środka w punkcie zaczepienia huśtawki.

Kąt wychylenia  $\alpha$  zdefiniujemy jako kąt między ramionami huśtawki oraz płaszczyzną jej stabilnej równowagi („pionem”). Wtedy odległość przebywana przez ciężar na huśtawce w czasie  $\Delta t$  jest równa zmianie kąta  $\alpha$  w czasie  $\Delta t$  pomnożonej przez długość ramienia huśtawki  $l$ .

# Opis ruchu

Położenie ciężaru na huśtawce zawsze znajduje się na okręgu o promieniu  $l$  i środku w punkcie zaczepienia huśtawki.

Kąt wychylenia  $\alpha$  zdefiniujemy jako kąt między ramionami huśtawki oraz płaszczyzną jej stabilnej równowagi („pionem”). Wtedy odległość przebywana przez ciężar na huśtawce w czasie  $\Delta t$  jest równa zmianie kąta  $\alpha$  w czasie  $\Delta t$  pomnożonej przez długość ramienia huśtawki  $l$ .

Na ciężar na huśtawce działa siła ciężkości, a właściwie jej składowa styczna do toru ruchu ciężaru, dana wzorem:

$$F_g(\alpha) = -mg \sin \alpha$$

gdzie  $g$  jest przyspieszeniem ziemskim.

# Równanie ruchu

Ponadto na huśtawkę działa tarcie. Będziemy przyjmować, że jest ono proporcjonalne do masy ciężaru na huśtawce i do jego prędkości (liniowej), a współczynnik proporcjonalności oznaczymy przez  $f$ .

# Równanie ruchu

Ponadto na huśtawkę działa tarcie. Będziemy przyjmować, że jest ono proporcjonalne do masy ciężaru na huśtawce i do jego prędkości (liniowej), a współczynnik proporcjonalności oznaczymy przez  $f$ .

Prędkość liniowa ciężaru na huśtawce jest równa prędkości kątowej huśtawki pomnożonej przez długość jej ramienia, natomiast prędkość kątowa huśtawki jest pochodną kąta  $\alpha$  względem czasu.

# Równanie ruchu

Ponadto na huśtawkę działa tarcie. Będziemy przyjmować, że jest ono proporcjonalne do masy ciężaru na huśtawce i do jego prędkości (liniowej), a współczynnik proporcjonalności oznaczymy przez  $f$ .

Prędkość liniowa ciężaru na huśtawce jest równa prędkości kątowej huśtawki pomnożonej przez długość jej ramienia, natomiast prędkość kątowa huśtawki jest pochodną kąta  $\alpha$  względem czasu.

Biorąc pod uwagę wszystkie powyższe obserwacje oraz drugą zasadę dynamiki, dostajemy następujące równanie ruchu na kąt  $\alpha$ :

$$\frac{d^2\alpha}{dt^2} = -\frac{g}{l} \sin \alpha - f \frac{d\alpha}{dt} \quad (5)$$

# Klasa pochodna od RungeKutta

Dla równania ruchu (5) można użyć następującej klasy pochodnej od RungeKutta:

```
class Pendulum : public RungeKutta {
private:
    double K; // iloraz g / l
    double f; // współczynnik tarcia

protected:
    void G(const vector <double> &y, double x, vector<double> &out);

public:
    Pendulum(const vector<double> u0, double t0, double d, double k, double r) :
        RungeKutta(u0, t0, d), K(k), f(r) {}
};

void Pendulum::G(const vector <double> &y, double x, vector<double> &out)
{
    out[0] = y[1]; // pochodna kąta wyhlenia
    out[1] = - K * sin(y[0]) - f * y[1]; // pochodna prędkości kątowej
    (void)x;
}
```

# Wyznaczanie „krytycznej” prędkości początkowej

Klasa ta pozwala nam znaleźć ruch wahadła dla ustalonej prędkości początkowej, ale nie wiemy z góry, czy osiągnie ono maksymalne wychylenie.

# Wyznaczanie „krytycznej” prędkości początkowej

Klasa ta pozwala nam znaleźć ruch wahadła dla ustalonej prędkości początkowej, ale nie wiemy z góry, czy osiągnie ono maksymalne wychylenie.

Możemy poszukać „krytycznej” prędkości metodą **bisekcji**, tzn. zgadnąć dwie prędkości początkowe takie, że dla jednej z nich maksymalne położenie jest przekraczane, a dla drugiej nie jest osiągnięte, a później sprawdzić co dzieje się dla wartości ze środka przedziału.



# Wyznaczanie „krytycznej” prędkości początkowej

Klasa ta pozwala nam znaleźć ruch wahadła dla ustalonej prędkości początkowej, ale nie wiemy z góry, czy osiągnie ono maksymalne wychylenie.

Możemy poszukać „krytycznej” prędkości metodą **bisekcji**, tzn. zgadnąć dwie prędkości początkowe takie, że dla jednej z nich maksymalne położenie jest przekraczane, a dla drugiej nie jest osiągnięte, a później sprawdzić co dzieje się dla wartości ze środka przedziału.

Następnie jedną z „odgadniętych” wartości zastępujemy ich średnią i powtarzamy. W ten sposób dla każdego kroku będziemy mieli dwukrotnie mniejszy przedział do przeszukania.