Programowanie, część IV

Rafał J. Wysocki

Instytut Fizyki Teoretycznej, Wydział Fizyki UW

22 maja 2012

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:

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

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).

Sprowadzanie 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:

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

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:

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

Wtedy następujące równanie jest równoważne równaniu (1).

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

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, ..., t_n takich, że $t_{i+1} = t_i + \Delta t$, jeśli tylko odstęp czasu Δt 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, ..., t_n takich, że $t_{i+1} = t_i + \Delta t$, jeśli tylko odstęp czasu Δt 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 ϵ jest zadanym limitem odchylenia obliczonego wektora od „prawdziwej” konfiguracji układu w danej chwili czasu.

Metoda Eulera

Zauważmy, że

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

Metoda Eulera

Zauważmy, że

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

Jeżeli $\mathbf{u}(t)$ nie jest funkcją o wartościach wektorowych, tylko 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 Δt.

Metoda Eulera

Zauważmy, że

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

Jeżeli $\mathbf{u}(t)$ nie jest funkcją o wartościach wektorowych, tylko 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 Δt.

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 $(\Delta t)^2$:

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

Wady metody Eulera

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

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

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 $(\Delta t)^2$:

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

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:

$$\begin{aligned} \mathbf{k}_1 &= \mathbf{g}(\mathbf{u}_i, t_i) \\ \mathbf{k}_2 &= \mathbf{g}(\mathbf{u}_i + \alpha \mathbf{k}_1 \Delta t, t_i + \beta \Delta t) \\ \mathbf{u}_{i+1} &= \mathbf{u}_i + \omega_1 \mathbf{k}_1 \Delta t + \omega_2 \mathbf{k}_2 \Delta t \end{aligned}$$

i wybierzmy stałe α, β, ω_1, ω_2 tak, aby różnica między $\mathbf{u}_{i+1}$ oraz $\mathbf{u}(t_{i+1})$ była rzędu $(\Delta t)^3$ (przy założeniu, że $\mathbf{u}_i = \mathbf{u}(t_i)$ dla tego konkretnego i).

Poprawianie metody Eulera (c. d.)

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

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

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

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

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 $\mathbf{g}(\mathbf{u}(t), t)$:

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

Wyprowadzenie metody Rungego-Kutty

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

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

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 wystarczy 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 wystarczy 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 $\mathbf{u}_{i+1}$ oraz $\mathbf{u}(t_{i+1})$ w pojedynczym kroku (przy założeniu $\mathbf{u}_i = \mathbf{u}(t_i)$ dla danego i) jest rzędu $(\Delta t)^5$.

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

W metodzie Rungego-Kutty czwartego rzędu różnica między $\mathbf{u}_{i+1}$ oraz $\mathbf{u}(t_{i+1})$ w pojedynczym kroku (przy założeniu $\mathbf{u}_i = \mathbf{u}(t_i)$ dla danego i) 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 ϵ 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 $\mathbf{u}_{i+1}$ oraz $\mathbf{u}(t_{i+1})$ w pojedynczym kroku (przy założeniu $\mathbf{u}_i = \mathbf{u}(t_i)$ dla danego i) 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 ϵ taka, że dla każdego i zachodzi $\|\mathbf{u}_i - \mathbf{u}(t_i)\| < \epsilon$.

Wartość ϵ 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 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).

W związku z tym wygodnie jest zdefiniować klasę dla tego obiektu jako klasę abstrakcyjną, w której funkcja $\mathbf{g}(\mathbf{u}(t), t)$ będzie wirtualną metodą bez implementacji.

Klasa dla algorytmu Rungego-Kutty

Używając wirtualnej metody bez implementacji jako „modelu" funkcji $\mathbf{g}(\mathbf{u}(t), t)$ 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

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

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

Konstruktor dla klasy RungeKutta

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

Obliczenia realizujące algorytm zawiera metoda `.wykonaj_krok()`.

Konstruktor dla klasy RungeKutta

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

Obliczenia realizujące algorytm zawiera metoda `.wykonaj_krok()`.

Dla wektorów v2, v3, v4 trzeba jawnie zadeklarować liczbę współrzędnych, ponieważ jeśli tego nie zrobimy, to w metodzie `.wykonaj_krok()` będą występować błędy związane z dostępem do pamięci.

```
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

Przedstawioną klasę `RungeKutta` można wykorzystać do wyznaczenia (w przybliżeniu) ruchu oscylatora harmonicznego. W tym celu wystarczy zdefiniować (odpowiednią) klasę pochodną w stosunku do `RungeKutta`.

Przykład: oscylator harmoniczny

Przedstawioną klasę `RungeKutta` można wykorzystać do wyznaczenia (w przybliżeniu) ruchu oscylatora harmonicznego. W tym celu wystarczy zdefiniować (odpowiednią) klasę pochodną w stosunku do `RungeKutta`.

```
#include "RungeKutta.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 _dt) :
                RungeKutta(u0, t0, _dt) {}
};

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

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

Przykład: oscylator harmoniczny – obliczenia

Mając do dyspozycji klasy `RungeKutta` i `Oscylator` zdefiniowane jak wyżej można przeprowadzić obliczenia.

Przykład: oscylator harmoniczny – obliczenia

Mając do dyspozycji klasy `RungeKutta` i `Oscylator` zdefiniowane jak wyżej można przeprowadzić obliczenia.

```
vector<double> y;

// 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 – obliczenia

Mając do dyspozycji klasy `RungeKutta` i `Oscylator` zdefiniowane jak wyżej można przeprowadzić obliczenia.

```
vector<double> y;

// 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();
}
```

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

Oscylator harmoniczny z tarciem

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

Oscylator harmoniczny z tarciem

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

```
#include "RungeKutta.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)
{
        (void)x;
        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
}
```

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^o.

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^o.

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

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^o.

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 środku w punkcie zaczepienia huśtawki.

Kąt wychylenia α zdefiniujemy jako kąt między ramionami huśtawki oraz płaszczyzną jej stabilnej równowagi („pionem"). Wtedy odległość przebywana przez ciężar na huśtawce w czasie Δt jest równa zmianie kąta α w czasie Δt 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 α zdefiniujemy jako kąt między ramionami huśtawki oraz płaszczyzną jej stabilnej równowagi („pionem"). Wtedy odległość przebywana przez ciężar na huśtawce w czasie Δt jest równa zmianie kąta α w czasie Δt 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 α 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 α względem czasu.

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

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

Przekształcone równanie ruchu

Równanie (5) można przepisać w postaci układu równań I rzędu:

$$\frac{d\alpha}{dt} = \omega \tag{6}$$

$$\frac{d\omega}{dt} = -\frac{g}{l}\sin\alpha - f\omega$$

Przekształcone równanie ruchu

Równanie (5) można przepisać w postaci układu równań I rzędu:

$$\frac{d\alpha}{dt} = \omega \tag{6}$$

$$\frac{d\omega}{dt} = -\frac{g}{l}\sin\alpha - f\omega$$

Jeżeli wprowadzimy wektor $\mathbf{y}(t) = [\alpha(t), \omega(t)]$, to:

$$\frac{dy_0}{dt} = y_1$$

$$\frac{dy_1}{dt} = -\frac{g}{l}\sin y_0 - f\, y_1$$

Klasa pochodna w stosunku do 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)
{
        (void)x;
        out[0] = y[1];                          // pochodna kąta wyhylenia
        out[1] = - K * sin(y[0]) - f * y[1];  // pochodna prędkości kątowej
}
```

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 osiągane, 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 osiągane, 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.

Kryterium przekroczenia maksymalnego wychylenia

Zauważmy, że jeżeli znak prędkości kątowej zmieni się w trakcie ruchu przy $\alpha < \pi$, to maksymalne wychylenie nie zostanie osiągnięte w tym ruchu.

Kryterium przekroczenia maksymalnego wychylenia

Zauważmy, że jeżeli znak prędkości kątowej zmieni się w trakcie ruchu przy $\alpha < \pi$, to maksymalne wychylenie nie zostanie osiągnięte w tym ruchu.

```
bool przekroczone(double v0)
{
        if (v0 < 0)
                v0 = -v0;

        vector<double> y;
        y.push_back(0.0);
        y.push_back(v0);

        Pendulum oscylator(y, 0, 0.01, 1, 0.3);

        while (y[1] > 0 && y[0] < M_PI) {
                oscylator.wykonaj_krok();
                y = oscylator.getU();
        }

        return y[0] >= M_PI;
}
```

I etap obliczeń – liniowe zwiększanie prędkości początkowej

Obliczenia można podzielić na dwa etapy. W pierwszym z nich wybieramy ustaloną wartość prędkości początkowej (kątowej) $\omega_0 = \Omega$ i sprawdzamy, czy przy tej wartości maksymalne wychylenie jest przekroczone. Jeżeli tak, przechodzimy do II etapu obliczeń. Jeżeli nie, zwiększamy ω_0 o Ω i powtarzamy.

I etap obliczeń – liniowe zwiększanie prędkości początkowej

Obliczenia można podzielić na dwa etapy. W pierwszym z nich wybieramy ustaloną wartość prędkości początkowej (kątowej) $\omega_0 = \Omega$ i sprawdzamy, czy przy tej wartości maksymalne wychylenie jest przekroczone. Jeżeli tak, przechodzimy do II etapu obliczeń. Jeżeli nie, zwiększamy ω_0 o Ω i powtarzamy.

```
// I etap - sartujemy od v0 = V i będziemy ją w każdym kroku zwiększać
// o V, aż wahadło przekroczy maksymalne wychylenie.
double v0 = V;

while (!przekroczone(v0))
        v0 += V;
```

I etap obliczeń – liniowe zwiększanie prędkości początkowej

Obliczenia można podzielić na dwa etapy. W pierwszym z nich wybieramy ustaloną wartość prędkości początkowej (kątowej) $\omega_0 = \Omega$ i sprawdzamy, czy przy tej wartości maksymalne wychylenie jest przekroczone. Jeżeli tak, przechodzimy do II etapu obliczeń. Jeżeli nie, zwiększamy ω_0 o Ω i powtarzamy.

```
// I etap - sartujemy od v0 = V i będziemy ją w każdym kroku zwiększać
// o V, aż wahadło przekroczy maksymalne wychylenie.
double v0 = V;

while (!przekroczone(v0))
        v0 += V;
```

Po zakończeniu tego etapu wiemy, że poszukiwana wartość znajduję się w przedziale $[\omega_0 - \Omega, \omega_0]$.

II etap obliczeń – bisekcja

Przeszukujemy przedział $[\omega_0 - \Omega, \omega_0]$ metodą bisekcji. Robimy to tak, że dla dolnej granicy przedziału maksymalne wychylenie nie jest przekraczane podczas ruchu, a dla jego górnej granicy – jest. W każdym kroku sprawdzamy, co dzieje się w środku przedziału i przesuwamy do niego jedną lub drugą granicę (tę, dla której zachowanie się układu jest takie, jak dla środka przedziału).

II etap obliczeń – bisekcja

Przeszukujemy przedział $[\omega_0 - \Omega, \omega_0]$ metodą bisekcji. Robimy to tak, że dla dolnej granicy przedziału maksymalne wychylenie nie jest przekraczane podczas ruchu, a dla jego górnej granicy – jest. W każdym kroku sprawdzamy, co dzieje się w środku przedziału i przesuwamy do niego jedną lub drugą granicę (tę, dla której zachowanie się układu jest takie, jak dla środka przedziału).

```
// II etap - bisekcja.
double a = v0, b = v0 - V;

while (a - b > epsilon) {
        double c = 0.5 * (a + b);

        if (przekroczone(c))
                a = c;
        else
                b = c;
}
```

II etap obliczeń – bisekcja

Wynikiem obliczeń jest przedział zawierający poszukiwaną wartość.

II etap obliczeń – bisekcja

Wynikiem obliczeń jest przedział zawierający poszukiwaną wartość.

Długość tego przedziału zależy od narzuconej programowi dokładności (stała `epsilon` w kodzie powyżej).

II etap obliczeń – bisekcja

Wynikiem obliczeń jest przedział zawierający poszukiwaną wartość.

Długość tego przedziału zależy od narzuconej programowi dokładności (stała `epsilon` w kodzie powyżej).

Podobne obliczenia można zrobić dla przypadku, w którym wartość początkowego wychylenia huśtawki jest niezerowa.

Wahadło z wymuszaniem

Do omówionego modelu można bardzo łatwo dodać „popychanie” w postaci siły proporcjonalnej do pewnej funkcji zależnej od czasu.

Wahadło z wymuszaniem

Do omówionego modelu można bardzo łatwo dodać „popychanie" w postaci siły proporcjonalnej do pewnej funkcji zależnej od czasu.

Jeśli funkcją tą jest $\cos(x)$, to równanie ruchu ma postać:

$$\frac{d^2\alpha}{dt^2} = -\frac{g}{l}\sin\alpha - f\frac{d\alpha}{dt} + B_c\cos(\omega_c t + \varphi_c) \tag{7}$$

gdzie B_c jest amplitudą, ω_c jest częstością drgań, a φ_c jest przesunięciem fazowym „wymuszającego" oscylatora.

Wahadło z wymuszaniem

Do omówionego modelu można bardzo łatwo dodać „popychanie" w postaci siły proporcjonalnej do pewnej funkcji zależnej od czasu.

Jeśli funkcją tą jest $\cos(x)$, to równanie ruchu ma postać:

$$\frac{d^2\alpha}{dt^2} = -\frac{g}{l}\sin\alpha - f\frac{d\alpha}{dt} + B_c\cos(\omega_c t + \varphi_c) \tag{7}$$

gdzie B_c jest amplitudą, ω_c jest częstością drgań, a φ_c jest przesunięciem fazowym „wymuszającego" oscylatora.

W tym modelu można zaobserwować interesujące zachowanie się układu, takie jak rezonans lub chaotyczna ewolucja. Tak, jak poprzednio, można stworzyć dla niego klasę pochodną względem klasy `RungeKutta`.

Definicja klasy dla wahadła z wymuszaniem

Klasa pochodna od RungeKutta dla wahadła z wymuszaniem

```
class Pendulum : public RungeKutta {
  private:
        double K;  // iloraz g / l
        double f;  // współczynnik tarcia
        double B;  // amplituda wymuszającego oscylatora
        double W;  // częstość wymuszającego oscylatora
        double S;  // przesunięcie fazowe wymuszającego oscylatora

  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,
                 double b, double w, double s) :
                RungeKutta(u0, t0, d), K(k), f(r), B(b), W(w), S(s) {}
};

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

        out[0] = y[1];
        out[1] = - K * sin(y[0]) - f * y[1] + B * cos(W * x + S);
}
```

Wykres położenia w funkcji czasu dla oscylatora

Obliczywszy, na przykład, położenie oscylatora harmonicznego w pewnej liczbie chwil czasu t_k takich, że $t_i - t_{i-1} = \Delta t$, możemy wykorzystać bibliotekę *Qt* do utworzenia rysunku zawierającego wykres tej funkcji.

Wykres położenia w funkcji czasu dla oscylatora

Obliczywszy, na przykład, położenie oscylatora harmonicznego w pewnej liczbie chwil czasu t_k takich, że $t_i - t_{i-1} = \Delta t$, możemy wykorzystać bibliotekę *Qt* do utworzenia rysunku zawierającego wykres tej funkcji.

1. Tworzymy klasę pochodną od `RungeKutta` do reprezentowania oscylatorów harmonicznych oraz obiekt tej klasy reprezentujący oscylator o zadanych parametrach (K, położenie początkowe i prędkość początkowa).
2. Używamy tej klasy do wyznaczenia położenia oscylatora w zadanch chwilach czasu. Wyniki można umieścić w dwóch obiektach klasy `vector` (w jednym – czas, a w drugim – położenie oscylatora).
3. W ten sposób otrzymujemy zbiór współrzędnych punktów reprezentujących funkcję do narysowania.

Obliczanie zależności położenia od czasu

Otrzymane wektory można przekazać do konstruktora obiektu klasy pochodnej od `QWidget`, którego metoda `paintEvent()` będzie używać ich przy rysowaniu wykresu.

Obliczanie zależności położenia od czasu

Otrzymane wektory można przekazać do konstruktora obiektu klasy pochodnej od `QWidget`, którego metoda `paintEvent()` będzie używać ich przy rysowaniu wykresu.

```
vector<double> y;

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

Oscylator oscylator(1.0, y, 0, 0.01);

vector<double> t(KROKI);
vector<double> q(KROKI);

for (int i = 0; i < KROKI; i++) {
    y = oscylator.getU();
    t[i] = oscylator.getT();
    q[i] = y[0];
    oscylator.wykonaj_krok();
}
```

```
class Obrazek : public QWidget {
  private:
    vector<double> x;
    vector<double> y;
    double x0, y0, x1, y1;
    unsigned int rozmiar;

  protected:
    void paintEvent(QPaintEvent *event);

  public:
    Obrazek(vector<double>& a, vector<double>& b,
            QWidget *parent = NULL);
};
```

Rysowanie wykresu położenia w funkcji czasu

```
Obrazek::Obrazek(vector<double>& a,
                 vector<double>& b,
                 QWidget *parent) : QWidget(parent)
{
    x = a;
    y = b;
    rozmiar = x.size();
    if (y.size() < rozmiar)
        rozmiar = y.size();

    x0 = x[0];
    x1 = x[rozmiar - 1];
    y0 = y[0];
    y1 = y[rozmiar - 1];
    for (unsigned int j = 1; j < rozmiar; j++) {
        if (x[j] < x0)
            x0 = x[j];
        else if (x[j] > x1)
            x1 = x[j];

        if (y[j] < y0)
            y0 = y[j];
        else if (y[j] > y1)
            y1 = y[j];
    }
}
```

```
void Obrazek::paintEvent(QPaintEvent *event)
{
    (void)event;

    QPainter painter(this);
    double w = x1 - x0;
    double h = y1 - y0;

    painter.scale(width() / w, -height() / h);
    painter.translate(-x0, -y1);

    QPointF p1(x[0], y[0]);
    QPointF p2;
    for (unsigned int j = 1; j < rozmiar; j++) {
        p2 = QPointF(x[j], y[j]);
        painter.drawLine(p1, p2);
        p1 = p2;
    }
}
```

Algorytm eliminacji Gaussa

Przypuśćmy, że chcemy znaleźć rozwiązanie równania postaci $\widehat{A}\mathbf{x} = \mathbf{b}$, gdzie $\widehat{A}$ jest macierzą (o jednakowej liczbie wierszy i kolumn), a $\mathbf{b}$ jest wektorem o zadanych współrzędnych.

Algorytm eliminacji Gaussa

Przypuśćmy, że chcemy znaleźć rozwiązanie równania postaci $\widehat{A}\mathbf{x} = \mathbf{b}$, gdzie $\widehat{A}$ jest macierzą (o jednakowej liczbie wierszy i kolumn), a $\mathbf{b}$ jest wektorem o zadanych współrzędnych.

W ogólności można sprowadzić to równanie do równania $\widehat{U}\mathbf{x} = \mathbf{b}'$, gdzie $\widehat{U}$ jest macierzą trójkątną.

Algorytm eliminacji Gaussa

Przypuśćmy, że chcemy znaleźć rozwiązanie równania postaci $\widehat{A}\mathbf{x} = \mathbf{b}$, gdzie $\widehat{A}$ jest macierzą (o jednakowej liczbie wierszy i kolumn), a $\mathbf{b}$ jest wektorem o zadanych współrzędnych.

W ogólności można sprowadzić to równanie do równania $\widehat{U}\mathbf{x} = \mathbf{b}'$, gdzie $\widehat{U}$ jest macierzą trójkątną.

Wtedy rozwiązania oryginalnego równania można otrzymać poprzez tak zwane podstawienie wsteczne (*ang. backward substitution*).

Algorytm eliminacji Gaussa

Przypuśćmy, że chcemy znaleźć rozwiązanie równania postaci $\widehat{A}\mathbf{x} = \mathbf{b}$, gdzie $\widehat{A}$ jest macierzą (o jednakowej liczbie wierszy i kolumn), a $\mathbf{b}$ jest wektorem o zadanych współrzędnych.

W ogólności można sprowadzić to równanie do równania $\widehat{U}\mathbf{x} = \mathbf{b}'$, gdzie $\widehat{U}$ jest macierzą trójkątną.

Wtedy rozwiązania oryginalnego równania można otrzymać poprzez tak zwane podstawienie wsteczne (*ang. backward substitution*).

Algorytm Gaussa określa sposób wyznaczenia macierzy $\widehat{U}$ zwany eliminacją do przodu (*ang. forward elimination*).

Eliminacja do przodu

Aby możliwe było przeprowadzenie obliczeń tą metodą, macierz $\widehat{A}$ musi spełniać określone warunki. Załóżmy, że są one spełnione.

Eliminacja do przodu

Aby możliwe było przeprowadzenie obliczeń tą metodą, macierz $\widehat{A}$ musi spełniać określone warunki. Załóżmy, że są one spełnione.

W pierwszym kroku obliczamy macierz $\widehat{A}^{(1)}$, dla której:

1. Pierwszy wiersz jest identyczny z pierwszym wierszem macierzy $\widehat{A}$.
2. W każdym z pozostałych wierszy pierwszy element macierzowy jest zerem:

$$A_{ij}^{(1)} = A_{ij} - \frac{A_{i1}}{A_{11}} A_{1j}$$

Eliminacja do przodu

Aby możliwe było przeprowadzenie obliczeń tą metodą, macierz $\widehat{A}$ musi spełniać określone warunki. Załóżmy, że są one spełnione.

W pierwszym kroku obliczamy macierz $\widehat{A}^{(1)}$, dla której:

1. Pierwszy wiersz jest identyczny z pierwszym wierszem macierzy $\widehat{A}$.
2. W każdym z pozostałych wierszy pierwszy element macierzowy jest zerem:

$$A_{ij}^{(1)} = A_{ij} - \frac{A_{i1}}{A_{11}} A_{1j}$$

Jednocześnie obliczamy wektor $\mathbf{b}^{(1)}$ taki, że $b_1^{(1)} = b_1$ oraz (dla $i > 1$)

$$b_i^{(1)} = b_i - \frac{A_{i1}}{A_{11}} b_1$$

Eliminacja do przodu (c. d.)

Inaczej mówiąc mnożymy równanie 1 w oryginalnym układzie równań przez A_{i1}/A_{11} i odejmujemy wynik od równania i (powtarzając dla wszystkich $i > 1$).

Eliminacja do przodu (c. d.)

Inaczej mówiąc mnożymy równanie 1 w oryginalnym układzie równań przez A_{i1}/A_{11} i odejmujemy wynik od równania i (powtarzając dla wszystkich $i > 1$).

W drugim kroku równania 1 i 2 pozostawiamy bez zmian, zaś dla $i > 2$ równanie 2 mnożymy przez $A^{(1)}_{i2}/A^{(1)}_{22}$ i wynik odejmujemy od równania i.

Eliminacja do przodu (c. d.)

Inaczej mówiąc mnożymy równanie 1 w oryginalnym układzie równań przez A_{i1}/A_{11} i odejmujemy wynik od równania i (powtarzając dla wszystkich $i > 1$).

W drugim kroku równania 1 i 2 pozostawiamy bez zmian, zaś dla $i > 2$ równanie 2 mnożymy przez $A^{(1)}_{i2}/A^{(1)}_{22}$ i wynik odejmujemy od równania i.

W ten sposób otrzymujemy macierz macierz $\widehat{A}^{(2)}$, dla której:

1. Pierwszy wiersz jest identyczny z pierwszym wierszem macierzy $\widehat{A}$.
2. Drugi wiersz jest identyczny z drugim wierszem macierzy $\widehat{A}^{(1)}$.
3. W każdym z pozostałych wierszy pierwszy i drugi element macierzowy są zerami.

Eliminacja do przodu (c. d.)

W rezultacie mamy (dla $i > 2$):

$$A_{ij}^{(2)} = A_{ij}^{(1)} - \frac{A_{i2}^{(1)}}{A_{22}^{(1)}} A_{2j}^{(1)}$$

$$b_i^{(2)} = b_i^{(1)} - \frac{A_{i2}^{(1)}}{A_{22}^{(1)}} b_2^{(1)}$$

Eliminacja do przodu (c. d.)

W rezultacie mamy (dla $i > 2$):

$$A_{ij}^{(2)} = A_{ij}^{(1)} - \frac{A_{i2}^{(1)}}{A_{22}^{(1)}} A_{2j}^{(1)}$$

$$b_{i}^{(2)} = b_{i}^{(1)} - \frac{A_{i2}^{(1)}}{A_{22}^{(1)}} b_{2}^{(1)}$$

Powtarzając analogiczną procedurę dla kolejnych kolumn macierzy po lewej stronie równania w kroku k otrzymujemy macierz $\widehat{A}^{(k)}$, dla której:

1. Wiersze od 1 do k są takie, jak w macierzy $\widehat{A}^{(k-1)}$.
2. W każdym z pozostałych wierszy elementy macierzowe $A_{ij}^{(k)}$ dla $j < k$ są zerami.

Eliminacja do przodu (c. d.)

W trakcie wykonywania obliczeń elementy macierzowe macierzy $\widehat{A}$ mogą być zastępowane przez odpowiednie elementy macierzowe macierzy $\widehat{A}^{(k)}$ otrzymywanych w kolejnych krokach obliczeń (elementy macierzowe $\widehat{A}$, z wyjątkiem wiersza 1, nie są już potrzebne po przeprowadzeniu kroku 1, zaś elementy macierzowe $\widehat{A}^{(1)}$, z wyjątkiem wierszy 1 i 2, nie są potrzebne po przeprowadzeniu kroku 2 itd.).

Eliminacja do przodu (c. d.)

W trakcie wykonywania obliczeń elementy macierzowe macierzy $\widehat{A}$ mogą być zastępowane przez odpowiednie elementy macierzowe macierzy $\widehat{A}^{(k)}$ otrzymywanych w kolejnych krokach obliczeń (elementy macierzowe $\widehat{A}$, z wyjątkiem wiersza 1, nie są już potrzebne po przeprowadzeniu kroku 1, zaś elementy macierzowe $\widehat{A}^{(1)}$, z wyjątkiem wierszy 1 i 2, nie są potrzebne po przeprowadzeniu kroku 2 itd.).

Podobnie współrzędne wektora $\mathbf{b}$ mogą być zastępowane przez odpowiednie współrzędne wektorów $\mathbf{b}^{(k)}$ otrzymywanych w kolejnych krokach obliczeń.

Eliminacja do przodu (c. d.)

W trakcie wykonywania obliczeń elementy macierzowe macierzy $\widehat{A}$ mogą być zastępowane przez odpowiednie elementy macierzowe macierzy $\widehat{A}^{(k)}$ otrzymywanych w kolejnych krokach obliczeń (elementy macierzowe $\widehat{A}$, z wyjątkiem wiersza 1, nie są już potrzebne po przeprowadzeniu kroku 1, zaś elementy macierzowe $\widehat{A}^{(1)}$, z wyjątkiem wierszy 1 i 2, nie są potrzebne po przeprowadzeniu kroku 2 itd.).

Podobnie współrzędne wektora $\mathbf{b}$ mogą być zastępowane przez odpowiednie współrzędne wektorów $\mathbf{b}^{(k)}$ otrzymywanych w kolejnych krokach obliczeń.

Po przeprowadzeniu obliczeń dla wszystkich kolumn macierzy po lewej stronie równania otrzymujemy poszukiwaną trójkątną macierz $\widehat{U}$.

Eliminacja do przodu – kod

Jeżeli miejsce przechowywania elementu macierzowego $A_{ij}^{(k)}$ oznaczymy przez `a[i][j]`, to algorytm obliczeń dla eliminacji do przodu można zapisać w następujący sposób (`n` jest wymiarem macierzy $\widehat{A}$):

```
for (k = 1; k < n; k++)
    for (i = k + 1; i <= n; i++) {
        double alfa = a[i][k] / a[k][k];

        for (j = k; j <= n; j++)
            a[i][j] -= alfa * a[k][j];

        b[i] -= alfa * b[k];
    }
```

Podstawienie wstecz – kod

Podstawienie wstecz można przeprowadzić w taki sposób, że współrzędne rozwiązania zostaną zapisane w miejscach służących do przechowywania współrzędnych wektora **b**:

```
for (i = n; i >= 1; i--) {
    // Dla j > i b[j] jest współrzędną poszukiwanego wektora x[].
    for (int j = i + 1; j <= n; j++)
        b[i] -= a[i][j] * b[j];

    b[i] /= a[i][i];
}
```

Podstawienie wstecz – kod

Podstawienie wstecz można przeprowadzić w taki sposób, że współrzędne rozwiązania zostaną zapisane w miejscach służących do przechowywania współrzędnych wektora **b**:

```
for (i = n; i >= 1; i--) {
    // Dla j > i b[j] jest współrzędną poszukiwanego wektora x[].
    for (int j = i + 1; j <= n; j++)
        b[i] -= a[i][j] * b[j];

    b[i] /= a[i][i];
}
```

Powyższa pętla jest zapisana z założeniem, że `a[i][j]` oznacza miejsce przechowywania elementu macierzowego U_{ij}, a `b[i]` – mejsce przechowywania współrzędnej b'_i wektora $\mathbf{b}'$.

Eliminacja Gaussa – warunki

Aby można było przeprowadzić eliminację Gaussa, zmienna `a[k][k]` musi być różna od zera na początku kroku k.

Eliminacja Gaussa – warunki

Aby można było przeprowadzić eliminację Gaussa, zmienna `a[k][k]` musi być różna od zera na początku kroku k.

W praktyce, aby uniknąć znaczących błędów zaokrąglenia, żąda się, aby jej wartość bezwzględna była większa od pewnej zadanej stałej ϵ.

Eliminacja Gaussa – warunki

Aby można było przeprowadzić eliminację Gaussa, zmienna `a[k][k]` musi być różna od zera na początku kroku k.

W praktyce, aby uniknąć znaczących błędów zaokrąglenia, żąda się, aby jej wartość bezwzględna była większa od pewnej zadanej stałej ϵ.

Najwygodniej jest sprawdzać spełnienie tego warunku już w trakcie trwania obliczeń, przerywając je w przypadku, gdy nie jest on spełniony dla pewnego k.

Eliminacja Gaussa – warunki

Aby można było przeprowadzić eliminację Gaussa, zmienna `a[k][k]` musi być różna od zera na początku kroku k.

W praktyce, aby uniknąć znaczących błędów zaokrąglenia, żąda się, aby jej wartość bezwzględna była większa od pewnej zadanej stałej ϵ.

Najwygodniej jest sprawdzać spełnienie tego warunku już w trakcie trwania obliczeń, przerywając je w przypadku, gdy nie jest on spełniony dla pewnego k.

Układ równań początkowo nie spełniający wymaganych warunków może być przekształcony do postaci, w której będą one spełnione.

Algorytm eliminacji Gaussa-Jordana

Algorytm eliminacji Gaussa-Jordana jest bardzo podobny do algorytmu eliminacji Gaussa, ale wykorzystuje obserwację, że eliminację można przeprowadzić także w „górnych" wierszach macierzy, otrzymując ostatecznie macierz diagonalną, dzięki czemu nie ma potrzeby przeprowadzania podstawienia wstecz.

Algorytm eliminacji Gaussa-Jordana

Algorytm eliminacji Gaussa-Jordana jest bardzo podobny do algorytmu eliminacji Gaussa, ale wykorzystuje obserwację, że eliminację można przeprowadzić także w „górnych" wierszach macierzy, otrzymując ostatecznie macierz diagonalną, dzięki czemu nie ma potrzeby przeprowadzania podstawienia wstecz.

```
for (k = 1; k <= n; k++) {
    for (i = 1; i <= n; i++) {
        if (i == k)
            continue;

        alfa = a[i][k] / a[k][k];
        for (j = k; j <= n; j++)
            a[i][j] -= alfa * a[k][j];

        b[i] -= alfa * b[k];
    }
    x[k] = b[k] / a[k][k];
}
```

Wady algorytmu eliminacji Gaussa

Istnieją układy równań liniowych, dla których istnieje rozwiązanie, chociaż warunki konieczne do przeprowadzenia eliminacji Gaussa nie są przez nie spełniane w wyjściowej postaci.

Wady algorytmu eliminacji Gaussa

Istnieją układy równań liniowych, dla których istnieje rozwiązanie, chociaż warunki konieczne do przeprowadzenia eliminacji Gaussa nie są przez nie spełniane w wyjściowej postaci.

Ponadto jeśli elementy macierzowe w różnych wierszach macierzy równania znacznie różnią się od siebie (np. elementy macierzowe w wierszu k są o wiele rzędów wielkości większe od odpowiadających im elementów macierzowych w wierszu j), w czasie obliczeń mogą powstać znaczące błędy zaokrąglenia.

Wady algorytmu eliminacji Gaussa

Istnieją układy równań liniowych, dla których istnieje rozwiązanie, chociaż warunki konieczne do przeprowadzenia eliminacji Gaussa nie są przez nie spełniane w wyjściowej postaci.

Ponadto jeśli elementy macierzowe w różnych wierszach macierzy równania znacznie różnią się od siebie (np. elementy macierzowe w wierszu k są o wiele rzędów wielkości większe od odpowiadających im elementów macierzowych w wierszu j), w czasie obliczeń mogą powstać znaczące błędy zaokrąglenia.

Problemów tych można uniknąć poprzez odpowiednie przekształcanie układu równań podczas przeprowadzania obliczeń.

Normalizacja elementów macierzowych

Pozwala na redukcję błędów zaokrąglenia.

Normalizacja elementów macierzowych

Pozwala na redukcję błędów zaokrąglenia.

Polega na tym, że w każdym kroku obliczeń:

1. Dla każdego wiersza $i \geqslant k$ macierzy $\widehat{A}^{(k)}$ wyznacza się element macierzowy o największej wartości bezwzględnej.
2. Wszystkie elementy macierzowe w wierszu i macierzy $\widehat{A}^{(k)}$ oraz współrzędna i wektora $\mathbf{b}^{(k)}$ są dzielone przez wartość bezwzględną tego elementu macierzowego.

Normalizacja elementów macierzowych

Pozwala na redukcję błędów zaokrąglenia.

Polega na tym, że w każdym kroku obliczeń:

1. Dla każdego wiersza $i \geqslant k$ macierzy $\widehat{A}^{(k)}$ wyznacza się element macierzowy o największej wartości bezwzględnej.
2. Wszystkie elementy macierzowe w wierszu i macierzy $\widehat{A}^{(k)}$ oraz współrzędna i wektora $\mathbf{b}^{(k)}$ są dzielone przez wartość bezwzględną tego elementu macierzowego.

Po przeprowadzeniu tej operacji wszystkie elementy macierzowe $\widehat{A}^{(k)}$ w wierszach $i \geqslant k$ mają wartości bezwzględne nie przekraczające 1.

Wybieranie dzielnika o największym module

Wykorzystuje obserwację, iż zamiana miejscami dwóch równań w układzie równań liniowych nie ma wpływu na jego rozwiązanie.

Wybieranie dzielnika o największym module

Wykorzystuje obserwację, iż zamiana miejscami dwóch równań w układzie równań liniowych nie ma wpływu na jego rozwiązanie.

Polega na tym, że w każdym kroku obliczeń:

1. Spośród wierszy $i \geqslant k$ macierzy $\widehat{A}^{(k)}$ wyznacza się wiersz i_{max}, w którym element w kolumnie k ma największą wartość bezwzględną.
2. Zamienia się miejscami wiersze k oraz i_{max} macierzy $\widehat{A}^{(k)}$ i współrzędne k oraz i_{max} wektora $\mathbf{b}^{(k)}$.

Wybieranie dzielnika o największym module

Wykorzystuje obserwację, iż zamiana miejscami dwóch równań w układzie równań liniowych nie ma wpływu na jego rozwiązanie.

Polega na tym, że w każdym kroku obliczeń:

1. Spośród wierszy $i \geqslant k$ macierzy $\widehat{A}^{(k)}$ wyznacza się wiersz i_{max}, w którym element w kolumnie k ma największą wartość bezwzględną.
2. Zamienia się miejscami wiersze k oraz i_{max} macierzy $\widehat{A}^{(k)}$ i współrzędne k oraz i_{max} wektora $\mathbf{b}^{(k)}$.

Po przeprowadzeniu tej operacji element macierzowy $A_{kk}^{(k)}$ ma największą wartość bezwzględną ze wszystkich elementów macierzowych $\widehat{A}^{(k)}$ w kolumnie k. Jeśli jest on zerem, to wszystkie elementy macierzowe $\widehat{A}^{(k)}$ w kolumnie k są zerami.

Poprawiony algorytm eliminacji Gaussa

```
for (k = 1; k < n; k++) {
    int i, j;
    double r, w;

    // Normalizacja elementów macierzowych
    for (i = k; i <= n; i++) {
        r = fabs(a[i][k]);
        for (j = k + 1; j <= n; j++) {
            w = fabs(a[i][j]);
            if (w > r)
                r = w;
        }
        if (r == 0)
            throw "Dzielenie przez zero";

        b[i] /= r;
        for (j = k; j <= n; j++)
            a[i][j] /= r;
    }

    // Wybór dzielnika o największym module
    r = fabs(a[k][k]);
    i = k;
    for (j = k + 1; j <= n; j++) {
        w = fabs(a[j][k]);
        if (w > r) {
            r = w;
            i = j;
        }
    }

    if (i > k) {
        zamiana_wierszy(a[k], a[i]);
        zamiana(b[k], b[i]);
    }

    if (a[k][k] == 0)
        throw "Dzielenie przez zero";

    // Eliminacja
    for (i = k + 1; i <= n; i++) {
        r = a[i][k] / a[k][k];
        b[i] -= r * b[k];
        for (j = k; j <= n; j++)
            a[i][j] -= r * a[k][j];
    }
}
```

Układy równań nieliniowych

Układy równań nieliniowych w ogólności mają postać

$$\mathbf{F}(\mathbf{x}) = 0$$

gdzie $\mathbf{F}(\mathbf{x})$ jest dowolną funkcją $\mathbb{R}^n \longrightarrow \mathbb{R}^m$.

Układy równań nieliniowych

Układy równań nieliniowych w ogólności mają postać

$$\mathbf{F}(\mathbf{x}) = 0$$

gdzie $\mathbf{F}(\mathbf{x})$ jest dowolną funkcją $\mathbb{R}^n \longrightarrow \mathbb{R}^m$.

Aby obliczyć (przybliżone) rozwiązanie takiego układu równań, można posłużyć się rozwinięciem Taylora funkcji $\mathbf{F}(\mathbf{x})$ w otoczeniu pewnego punktu $\mathbf{x}_0$, który będzie stanowił początkowe przybliżenie rozwiązania:

$$\mathbf{F}(\mathbf{x}) \approx \mathbf{F}(\mathbf{x}_0) + \nabla\mathbf{F}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0)$$

gdzie $\nabla\mathbf{F}(\mathbf{x}_0)$ oznacza macierz Jacobiego funkcji w punkcie $\mathbf{x}_0$.

Algorytm Newtona

Jeśli $\mathbf{x}_\infty$ jest ścisłym rozwiązaniem naszego równania, to mamy:

$$0 \approx \mathbf{F}(\mathbf{x}_0) + \nabla\mathbf{F}(\mathbf{x}_0)(\mathbf{x}_\infty - \mathbf{x}_0) \tag{8}$$

Algorytm Newtona

Jeśli $\mathbf{x}_\infty$ jest ścisłym rozwiązaniem naszego równania, to mamy:

$$0 \approx \mathbf{F}(\mathbf{x}_0) + \nabla\mathbf{F}(\mathbf{x}_0)(\mathbf{x}_\infty - \mathbf{x}_0) \tag{8}$$

To oznacza, że

$$\mathbf{x}_\infty \approx \mathbf{x}_0 - [\nabla\mathbf{F}(\mathbf{x}_0)]^{-1}\mathbf{F}(\mathbf{x}_0) \tag{9}$$

Algorytm Newtona

Jeśli $\mathbf{x}_\infty$ jest ścisłym rozwiązaniem naszego równania, to mamy:

$$0 \approx \mathbf{F}(\mathbf{x}_0) + \nabla\mathbf{F}(\mathbf{x}_0)(\mathbf{x}_\infty - \mathbf{x}_0) \tag{8}$$

To oznacza, że

$$\mathbf{x}_\infty \approx \mathbf{x}_0 - [\nabla\mathbf{F}(\mathbf{x}_0)]^{-1}\mathbf{F}(\mathbf{x}_0) \tag{9}$$

Możemy zatem skonstruować ciąg punktów $\mathbf{x}_k$ taki, że

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\nabla\mathbf{F}(\mathbf{x}_k)]^{-1}\mathbf{F}(\mathbf{x}_k) \tag{10}$$

który będzie zbieżny do $\mathbf{x}_\infty$, o ile punkt $\mathbf{x}_0$ zostanie wybrany we właściwy sposób (musi on być „dostatecznie dobrym” przybliżeniem rozwiązania).

Algorytm Newtona (c. d.)

Zamiast obliczania macierzy $[\nabla \mathbf{F}(\mathbf{x}_k)]^{-1}$ w każdym punkcie można wykorzystać metody omówione wyżej i rozwiązywać równania liniowe

$$\nabla \mathbf{F}(\mathbf{x}_k)\mathbf{d}_k = -\mathbf{F}(\mathbf{x}_k) \tag{11}$$

a następnie wyznaczać $\mathbf{x}_{k+1}$ z wzoru

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{d}_k \tag{12}$$

Algorytm Newtona (c. d.)

Zamiast obliczania macierzy $[\nabla \mathbf{F}(\mathbf{x}_k)]^{-1}$ w każdym punkcie można wykorzystać metody omówione wyżej i rozwiązywać równania liniowe

$$\nabla \mathbf{F}(\mathbf{x}_k)\mathbf{d}_k = -\mathbf{F}(\mathbf{x}_k) \tag{11}$$

a następnie wyznaczać $\mathbf{x}_{k+1}$ z wzoru

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{d}_k \tag{12}$$

Zatem najpierw rozwiązujemy równanie (11) dla $\mathbf{x}_0$, podstawiamy wynik do równania (12), aby obliczyć $\mathbf{x}_1$ itd.

Algorytm Newtona (c. d.)

Zamiast obliczania macierzy $[\nabla \mathbf{F}(\mathbf{x}_k)]^{-1}$ w każdym punkcie można wykorzystać metody omówione wyżej i rozwiązywać równania liniowe

$$\nabla \mathbf{F}(\mathbf{x}_k)\mathbf{d}_k = -\mathbf{F}(\mathbf{x}_k) \tag{11}$$

a następnie wyznaczać $\mathbf{x}_{k+1}$ z wzoru

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{d}_k \tag{12}$$

Zatem najpierw rozwiązujemy równanie (11) dla $\mathbf{x}_0$, podstawiamy wynik do równania (12), aby obliczyć $\mathbf{x}_1$ itd.

Obliczenia można zakończyć np. wtedy, gdy $||\mathbf{x}_j - \mathbf{x}_{j-1}|| < \epsilon$ dla pewnego $k = j$ (i dla zadanej stałej ϵ).

Tablice dwuwymiarowe

Najprościej jest reprezentować macierz z wykorzystaniem tablicy dwuwymiarowej o elementach typu `double`.

Tablice dwuwymiarowe

Najprościej jest reprezentować macierz z wykorzystaniem tablicy dwuwymiarowej o elementach typu `double`.

Taką tablicę definiuje się określając liczbę jej wierszy i kolumn (`N` i `M`; mogą to być zmienne, jeżeli sama tablica jest zmienną lokalną):

```
double tablica[N][M];
```

Tablice dwuwymiarowe

Najprościej jest reprezentować macierz z wykorzystaniem tablicy dwuwymiarowej o elementach typu `double`.

Taką tablicę definiuje się określając liczbę jej wierszy i kolumn (`N` i `M`; mogą to być zmienne, jeżeli sama tablica jest zmienną lokalną):

```
double tablica[N][M];
```

Wtedy wiersze tablicy mają indeksy od 0 do $N-1$, a kolumny mają indeksy od 0 do $M-1$.

Tablice dwuwymiarowe

Najprościej jest reprezentować macierz z wykorzystaniem tablicy dwuwymiarowej o elementach typu `double`.

Taką tablicę definiuje się określając liczbę jej wierszy i kolumn (`N` i `M`; mogą to być zmienne, jeżeli sama tablica jest zmienną lokalną):

```
double tablica[N][M];
```

Wtedy wiersze tablicy mają indeksy od 0 do $N-1$, a kolumny mają indeksy od 0 do $M-1$.

W C++ nie ma możliwości zmiany sposobu indeksowania kolumn i wierszy tablicy dwuwymiarowej. Wynika to ze sposobu rozmieszczenia elementów tablicy w pamięci.

Struktura tablic dwuwymiarowych

W C++ (podobnie, jak w C) każdy wiersz tablicy dwuwymiarowej jest traktowany jako oddzielna tablica jednowymiarowa.

Struktura tablic dwuwymiarowych

W C++ (podobnie, jak w C) każdy wiersz tablicy dwuwymiarowej jest traktowany jako oddzielna tablica jednowymiarowa.

Zatem, jeżeli `tablica[][]` jest tablicą dwuwymiarową zdefiniowaną jak wyżej, to symbol `tablica[i]` oznacza wskaźnik zawierający adres elementu `tablica[i][0]`.

Struktura tablic dwuwymiarowych

W C++ (podobnie, jak w C) każdy wiersz tablicy dwuwymiarowej jest traktowany jako oddzielna tablica jednowymiarowa.

Zatem, jeżeli `tablica[][]` jest tablicą dwuwymiarową zdefiniowaną jak wyżej, to symbol `tablica[i]` oznacza wskaźnik zawierający adres elementu `tablica[i][0]`.

W związku z tym symbol `*(tablica[i])` oznacza element `tablica[i][0]`.

Struktura tablic dwuwymiarowych

W C++ (podobnie, jak w C) każdy wiersz tablicy dwuwymiarowej jest traktowany jako oddzielna tablica jednowymiarowa.

Zatem, jeżeli `tablica[][]` jest tablicą dwuwymiarową zdefiniowaną jak wyżej, to symbol `tablica[i]` oznacza wskaźnik zawierający adres elementu `tablica[i][0]`.

W związku z tym symbol `*(tablica[i])` oznacza element `tablica[i][0]`.

Po wykonaniu przypisania `ptr = tablica[i]`, gdzie `ptr` jest wskaźnikiem typu `double`, można posługiwać się wskaźnikiem `ptr` tak, jakby był on nazwą tablicy jednowymiarowej pokrywającej się z wierszem *i* tablicy dwuwymiarowej.

Algorytm Gaussa w wersji wskaźnikowej

```
for (k = 0; k < n - 1; k++) {
    double *a_k = a[k];

    for (i = k + 1; i < n; i++) {
        double *a_i = a[i];
        double alfa = a_i[k] / a_k[k];

        for (j = k; j < n; j++)
            a_i[j] -= alfa * a_k[j];

        b[i] -= alfa * b[k];
    }
}
```

```
for (k = 0; k < n - 1; k++) {
    double *a_k = a[k];

    for (i = k + 1; i < n; i++) {
        double alfa = a[i][k] / a_k[k];
        double *a_i_j = a[i] + j;
        double *a_k_j = a_k + j;

        for (j = k; j < n; j++)
            *a_i_j++ -= *a_k_j++ * alpha;

        b[i] -= b[k] * alfa;
    }
}
```

Algorytm Gaussa w wersji wskaźnikowej

```
for (k = 0; k < n - 1; k++) {
    double *a_k = a[k];

    for (i = k + 1; i < n; i++) {
        double *a_i = a[i];
        double alfa = a_i[k] / a_k[k];

        for (j = k; j < n; j++)
            a_i[j] -= alfa * a_k[j];

        b[i] -= alfa * b[k];
    }
}
```

```
for (k = 0; k < n - 1; k++) {
    double *a_k = a[k];

    for (i = k + 1; i < n; i++) {
        double alfa = a[i][k] / a_k[k];
        double *a_i_j = a[i] + j;
        double *a_k_j = a_k + j;

        for (j = k; j < n; j++)
            *a_i_j++ -= *a_k_j++ * alpha;

        b[i] -= b[k] * alfa;
    }
}
```

Niestety w tablicach dwuwymiarowych nie można zamieniać miejscami wskaźników zawierających adresy poszczególnych wierszy. To powoduje, że w poprawionym algorytmie eliminacji Gaussa zamianę wierszy macierzy trzeba przeprowadzać jako kopiowanie elementów macierzowych.

Problemy z rozmiarami tablic dwuwymiarowych

Z tablicamy dwuwymiarowymi w C++ wiąże się dodatkowy problem, polegający na tym, że mogą one być deklarowane jako zmienne statyczne lub jako zmienne lokalne (w funkcjach).

Problemy z rozmiarami tablic dwuwymiarowych

Z tablicamy dwuwymiarowymi w C++ wiąże się dodatkowy problem, polegający na tym, że mogą one być deklarowane jako zmienne statyczne lub jako zmienne lokalne (w funkcjach).

Jeżeli mają one być zmiennymi statycznymi, to ich rozmiary muszą być znane z wyprzedzeniem (tzn. przed skompilowaniem programu) lub trzeba nadawać im rozmiary „na zapas”.

Problemy z rozmiarami tablic dwuwymiarowych

Z tablicamy dwuwymiarowymi w C++ wiąże się dodatkowy problem, polegający na tym, że mogą one być deklarowane jako zmienne statyczne lub jako zmienne lokalne (w funkcjach).

Jeżeli mają one być zmiennymi statycznymi, to ich rozmiary muszą być znane z wyprzedzeniem (tzn. przed skompilowaniem programu) lub trzeba nadawać im rozmiary „na zapas".

Jeżeli mają one być zmiennymi lokalnymi, to pamięć na przechowywanie ich elementów macierzowych jest rezerwowana na stosie procesora. Powoduje to, że rozmiary takich tablic podlegają ograniczeniom.

Problemy z rozmiarami tablic dwuwymiarowych

Z tablicamy dwuwymiarowymi w C++ wiąże się dodatkowy problem, polegający na tym, że mogą one być deklarowane jako zmienne statyczne lub jako zmienne lokalne (w funkcjach).

Jeżeli mają one być zmiennymi statycznymi, to ich rozmiary muszą być znane z wyprzedzeniem (tzn. przed skompilowaniem programu) lub trzeba nadawać im rozmiary „na zapas".

Jeżeli mają one być zmiennymi lokalnymi, to pamięć na przechowywanie ich elementów macierzowych jest rezerwowana na stosie procesora. Powoduje to, że rozmiary takich tablic podlegają ograniczeniom.

Zatem pamięć do przechowywania elementów macierzowych macierzy najlepiej jest rezerwować na żądanie.

Macierze i rezerwowanie pamięci na żądanie

Kompilator C++ zawsze rozmieszcza wiersze tablic dwuwymiarowych w pamięci kolejno jeden za drugim, zgodnie z numeracją.

Macierze i rezerwowanie pamięci na żądanie

Kompilator C++ zawsze rozmieszcza wiersze tablic dwuwymiarowych w pamięci kolejno jeden za drugim, zgodnie z numeracją.

Można wykorzystać tę obserwację rezerwując na żądanie zmienne, które wspólnie będą reprezentować macierz:

```
double **wiersze, *elementy;

elementy = new double[N*M];  // Elementy macierzowe.
wiersze = new double *[N];   // Wskaźniki do wierszy.
for (int j = 0; j < N; j++)
    wiersze[j] = elementy + j*M;
```

Macierze i rezerwowanie pamięci na żądanie

Kompilator C++ zawsze rozmieszcza wiersze tablic dwuwymiarowych w pamięci kolejno jeden za drugim, zgodnie z numeracją.

Można wykorzystać tę obserwację rezerwując na żądanie zmienne, które wspólnie będą reprezentować macierz:

```
double **wiersze, *elementy;

elementy = new double[N*M];  // Elementy macierzowe.
wiersze = new double *[N];   // Wskaźniki do wierszy.
for (int j = 0; j < N; j++)
    wiersze[j] = elementy + j*M;
```

Wtedy symbol `wiersze[j][k]` oznacza element o indeksie k z wiersza o indeksie j macierzy, gdzie $j = 0 \ldots N-1$ oraz $k = 0 \ldots M-1$.

Macierze i rezerwowanie pamięci na żądanie (c. d.)

Rezerwując pamięć na żądanie w sposób podobny do przedstawionego można tak zmodyfikować wskaźniki, aby wiersze i kolumny macierzy były indeksowane począwszy od 1.

Macierze i rezerwowanie pamięci na żądanie (c. d.)

Rezerwując pamięć na żądanie w sposób podobny do przedstawionego można tak zmodyfikować wskaźniki, aby wiersze i kolumny macierzy były indeksowane począwszy od 1.

```
double **wiersze, *elementy;

elementy = new double[N*M];
wiersze = new double *[N];
wiersze--;
for (int j = 1; j <= N; j++)
    wiersze[j] = elementy + (j-1)*M - 1;
```

Macierze i rezerwowanie pamięci na żądanie (c. d.)

Rezerwując pamięć na żądanie w sposób podobny do przedstawionego można tak zmodyfikować wskaźniki, aby wiersze i kolumny macierzy były indeksowane począwszy od 1.

```
double **wiersze, *elementy;

elementy = new double[N*M];
wiersze = new double *[N];
wiersze--;
for (int j = 1; j <= N; j++)
    wiersze[j] = elementy + (j-1)*M - 1;
```

Wtedy symbol `wiersze[j][k]` oznacza element o indeksie k z wiersza o indeksie j macierzy, gdzie $j = 1 \ldots N$ oraz $k = 1 \ldots M$.

Klasa reprezentująca macierz

Można także zdefiniować klasę reprezentującą macierz:

```
class Macierz {
    double *elementy;
    int n, m;
  public:
    Macierz(int a, int b);
    ~Macierz(void);
    ...
    double * operator [](int i) const;
};

Macierz::Macierz(int a, int b)
{
    elementy = new double[a*b];
    n = a;
    m = b;
}
```

```
Macierz::~Macierz(void)
{
    delete [] elementy;
}

double * Macierz::operator [](int i) const
{
    if (i < 1 || i > n)
        throw "Przekroczony zakres";

    return elementy + (i-1)*m - 1;
}
```

Klasa reprezentująca macierz

Można także zdefiniować klasę reprezentującą macierz:

```
class Macierz {
    double *elementy;
    int n, m;
  public:
    Macierz(int a, int b);
    ~Macierz(void);
    ...
    double * operator [](int i) const;
};

Macierz::Macierz(int a, int b)
{
    elementy = new double[a*b];
    n = a;
    m = b;
}
```

```
Macierz::~Macierz(void)
{
    delete [] elementy;
}

double * Macierz::operator [](int i) const
{
    if (i < 1 || i > n)
        throw "Przekroczony zakres";

    return elementy + (i-1)*m - 1;
}
```

Wtedy, dla obiektu `M` klasy `Macierz`, symbol `M[j][k]` oznacza element o indeksie k z wiersza o indeksie j macierzy, gdzie $j = 1 \ldots N$ oraz $k = 1 \ldots M$.

Macierze i klasy (c. d.)

Aby uniknąć niepotrzebnych obliczeń w czasie wykonywania programu, można użyć pomocniczej tablicy `wiersze[]`, w której będą zapisywane adresy poszczególnych wierszy macierzy:

```
class Macierz {
    double **wiersze;
  public:
    Macierz(unsigned int a, unsigned int b);
    ~Macierz(void);
    ...
    double * operator [](int i) const;
};

Macierz::~Macierz(void)
{
    delete [] (wiersze[1] + 1);
    wiersze++;
    delete [] wiersze;
}
```

```
Macierz::Macierz(unsigned int a, unsigned int b)
{
    double *wsk;

    wsk = new double[a*b];
    wiersze = new double *[a];
    wiersze--;
    for (int j = 1; j <= a; j++)
        wiersze[j] = wsk + (j-1)*b - 1;
}

double * Macierz::operator [](int i) const
{
    return wiersze[i];
}
```

Macierze i klasy (c. d.)

Aby uniknąć niepotrzebnych obliczeń w czasie wykonywania programu, można użyć pomocniczej tablicy `wiersze[]`, w której będą zapisywane adresy poszczególnych wierszy macierzy:

```
class Macierz {
    double **wiersze;
  public:
    Macierz(unsigned int a, unsigned int b);
    ~Macierz(void);
    ...
    double * operator [](int i) const;
};

Macierz::~Macierz(void)
{
    delete [] (wiersze[1] + 1);
    wiersze++;
    delete [] wiersze;
}
```

```
Macierz::Macierz(unsigned int a, unsigned int b)
{
    double *wsk;

    wsk = new double[a*b];
    wiersze = new double *[a];
    wiersze--;
    for (int j = 1; j <= a; j++)
        wiersze[j] = wsk + (j-1)*b - 1;
}

double * Macierz::operator [](int i) const
{
    return wiersze[i];
}
```

Można także dodać kod sprawdzający przekroczenie zakresu indeksów.

Odwrotna notacja polska (ONP)

RPN (*ang. Reverse Polish Notation*)

ONP, zwana także notacją postfiksową (*ang. postfix*), jest sposobem zapisu wyrażeń arytmetycznych, w którym znak wykonywanej operacji umieszczony jest po operandach, np. $2\ \ 4\ +$ zamiast $2 + 4$.

Odwrotna notacja polska (ONP)

RPN (*ang. Reverse Polish Notation*)

ONP, zwana także notacją postfiksową (*ang. postfix*), jest sposobem zapisu wyrażeń arytmetycznych, w którym znak wykonywanej operacji umieszczony jest po operandach, np. $2\ \ 4\ +$ zamiast $2 + 4$.

ONP pozwala na całkowitą rezygnację z użycia nawiasów w wyrażeniach, ponieważ jednoznacznie określa kolejność wykonywanych działań.

Odwrotna notacja polska (ONP)

RPN (*ang. Reverse Polish Notation*)

ONP, zwana także notacją postfiksową (*ang. postfix*), jest sposobem zapisu wyrażeń arytmetycznych, w którym znak wykonywanej operacji umieszczony jest po operandach, np. $2\ \ 4\ +$ zamiast $2 + 4$.

ONP pozwala na całkowitą rezygnację z użycia nawiasów w wyrażeniach, ponieważ jednoznacznie określa kolejność wykonywanych działań.

ONP została zaproponowana w połowie lat 1950 przez Charlesa Hamblina, jako „odwrócenie" beznawiasowej notacji polskiej Jana Łukasiewicza.

Odwrotna notacja polska (ONP)

RPN (*ang. Reverse Polish Notation*)

ONP, zwana także notacją postfiksową (*ang. postfix*), jest sposobem zapisu wyrażeń arytmetycznych, w którym znak wykonywanej operacji umieszczony jest po operandach, np. $2\ \ 4\ +$ zamiast $2 + 4$.

ONP pozwala na całkowitą rezygnację z użycia nawiasów w wyrażeniach, ponieważ jednoznacznie określa kolejność wykonywanych działań.

ONP została zaproponowana w połowie lat 1950 przez Charlesa Hamblina, jako „odwrócenie" beznawiasowej notacji polskiej Jana Łukasiewicza.

Okazuje się, że algorytm obliczania wyrażeń w ONP jest stosunkowo prosty. Podobnie prosty jest algorytm przekształcania wyrażeń w notacji „tradycyjnej" (infiksowej) na ONP.

Algorytm obliczania wyrażeń w ONP

Potrzebny jest stos do przechowywania liczb.

1. Dla każdego elementu wyrażenia w ONP od lewej:
   1. Jeżeli bieżący element jest liczbą, umieść ją na stosie.
   2. Jeżeli bieżący element jest operatorem $(+, -, *, /)$, to:
      - Zdejmij dwie liczby ze stosu.
      - Przeprowadź obliczenia: druga liczba zdjęta ze stosu-operator-pierwsza liczba zdjęta ze stosu.
      - Umieść wynik na stosie.
2. Po zakończeniu pętli jedyna znajdująca się na stosie liczba jest wynikiem wyrażenia (o ile wyrażenie jest poprawne).

Obliczanie wyrażeń w ONP – przykład

Wyrażenie

$2\ 3\ +\ 4\ *\ 10\ -$

Przebieg obliczeń

Krok	**Na stosie**
1	2
2	2 3
3	5
4	5 4
5	20
6	20 10
7	10

Funkcja obliczająca wartość wyrażenia w ONP

```
double oblicz(string& wyr)
{
    stack<double> stos;
    istringstream input(wyr);
    string s;
    double r;

    while (input >> s) {
        istringstream is(s);

        if(is >> r) {
            stos.push(r);
            continue;
        }

        if (jest_operatorem(s[0])) {
            r = stos.top();
            stos.pop();

            r = wykonaj_obliczenia(r, stos.top(), s[0]);
            stos.pop();

            stos.push(r);
        }
    }
    r = stos.top();
    stos.pop();

    return r;
}


bool jest_operatorem(char c)
{
    return c == '+' || c == '-'
        || c == '*' || c == '/';
}
```

Funkcja obliczająca wartość wyrażenia w ONP (c. d.)

```
double wykonaj_obliczenia(double r, double d, char c)
{
    switch (c) {
    case '+':
        d += r;
        break;
    case '-':
        d -= r;
        break;
    case '*':
        d *= r;
        break;
    case '/':
        d /= r;
        break;
    }

    return d;
}
```

Algorytm zamiany zapisu wyrażenia na ONP (1)

Potrzebny jest stos do przechowywania symboli oraz łańcuch, w którym będzie zapisany wynik.

Algorytm zamiany zapisu wyrażenia na ONP (1)

Potrzebny jest stos do przechowywania symboli oraz łańcuch, w którym będzie zapisany wynik.

Dla każdego elementu wyrażenia w notacji infiksowej (z nawiasami) (1)

- Jeśli bieżący element jest liczbą, wstaw go oraz znak przerwy na koniec wynikowego łańcucha.
- Jeśli bieżący element jest operatorem, to:
  1. Powtarzaj, aż na wierzchołku stosu nie będzie operatora:
     - Zdejmij operator ze stosu.
     - Wstaw zdjęty ze stosu operator oraz znak przerwy na koniec wynikowego łańcucha.
  2. Wstaw bieżący element na stos.
- Jeśli bieżący element jest lewym nawiasem, wstaw go na stos.

Algorytm zamiany zapisu wyrażenia na ONP (2)

Dla każdego elementu wyrażenia w notacji infiksowej (2)

- Jeśli bieżący element jest prawym nawiasem, to:
  1. Zdejmij symbol ze stosu.
  2. Powtarzaj, dopóki zdjęty ze stosu symbol nie jest lewym nawiasem:
     - Wstaw zdjęty ze stosu symbol i znak przerwy na koniec wynikowego łańcucha.
     - Zdejmij symbol ze stosu.

Algorytm zamiany zapisu wyrażenia na ONP (2)

Dla każdego elementu wyrażenia w notacji infiksowej (2)

- Jeśli bieżący element jest prawym nawiasem, to:
  1. Zdejmij symbol ze stosu.
  2. Powtarzaj, dopóki zdjęty ze stosu symbol nie jest lewym nawiasem:
     - Wstaw zdjęty ze stosu symbol i znak przerwy na koniec wynikowego łańcucha.
     - Zdejmij symbol ze stosu.

Powtarzaj, aż stos będzie pusty

- Zdejmij symbol ze stosu.
- Wstaw symbol zdjęty ze stosu oraz znak przerwy na koniec wynikowego łańcucha.

Algorytm zamiany zapisu wyrażenia na ONP (2)

Dla każdego elementu wyrażenia w notacji infiksowej (2)

- Jeśli bieżący element jest prawym nawiasem, to:
  1. Zdejmij symbol ze stosu.
  2. Powtarzaj, dopóki zdjęty ze stosu symbol nie jest lewym nawiasem:
     - Wstaw zdjęty ze stosu symbol i znak przerwy na koniec wynikowego łańcucha.
     - Zdejmij symbol ze stosu.

Powtarzaj, aż stos będzie pusty

- Zdejmij symbol ze stosu.
- Wstaw symbol zdjęty ze stosu oraz znak przerwy na koniec wynikowego łańcucha.

Usuń ostatni znak (znak przerwy) z wynikowego łańcucha

Zamiana zapisu wyrażenia na ONP – przykład I

Przebieg obliczeń dla wyrażenia $((2+3)*5)-1$

Krok	Na stosie	Wynik
1	(	
2	((	
3	((	2
4	((+	2
5	((+	2 3
6	(	2 3 +
7	(*	2 3 +
8	(*	2 3 + 5
9		2 3 + 5 *
10	−	2 3 + 5 *
11	−	2 3 + 5 * 1
12		2 3 + 5 * 1 −

Modyfikacja związana z priorytetami operatorów

Jeżeli operatory mają różne priorytety, powyższy algorytm może nie dać prawidłowego wyniku dla wyrażenia bez nawiasów, więc trzeba go trochę zmienić.

Modyfikacja związana z priorytetami operatorów

Jeżeli operatory mają różne priorytety, powyższy algorytm może nie dać prawidłowego wyniku dla wyrażenia bez nawiasów, więc trzeba go trochę zmienić.

Dla każdego elementu wyrażenia w notacji infiksowej (z nawiasami)

- Jeśli bieżący element jest liczbą, wstaw go oraz znak przerwy na koniec wynikowego łańcucha.
- Jeśli bieżący element jest operatorem, to:
  1. Powtarzaj, aż na wierzchołku stosu nie będzie operatora o wyższym priorytecie:
     - Zdejmij operator ze stosu.
     - Wstaw zdjęty ze stosu operator oraz znak przerwy na koniec wynikowego łańcucha.
  2. Wstaw bieżący element na stos.

Zamiana zapisu wyrażenia na ONP – przykład II

Przebieg obliczeń dla wyrażenia $(2 * 3 + 5)/2$

Krok	Na stosie	Wynik
1	(	
3	(	2
4	(*	2
5	(*	2 3
6	(+	2 3 *
7	(+	2 3 * 5
8		2 3 * 5 +
9	/	2 3 * 5 +
10	/	2 3 * 5 + 2
11		2 3 * 5 + 2 /

Funkcja zamieniająca zapis wyrażenia na ONP

```
string infix_postfix(string& wyr)
{
    stack<char> stos;
    istringstream input(wyr);
    string s, postfix("");

    while (input >> s) {
        istringstream is(s);
        double r;

        if(is >> r) {
            postfix += s + ' ';
            continue;
        }
        char c = s[0];
        if (jest_operatorem(c)) {
            while (!stos.empty()) {
                char znak = stos.top();
                if (!jest_operatorem(znak))
                    break;
                if (!jest_silniejszy(znak, c))
                    break;
                postfix += znak;
                postfix += ' ';
                stos.pop();
            }
            stos.push(c);
        } else if (c == '(') {
            stos.push(c);
        } else if (c == ')') {
            char znak = stos.top();
            stos.pop();
            while (znak != '(') {
                postfix += znak;
                postfix += ' ';
                znak = stos.top();
                stos.pop();
            }
        }
    }

    while (!stos.empty()) {
        postfix += stos.top();
        stos.pop();
        postfix += ' ';
    }

    if (postfix.length() > 0)
        postfix.erase(postfix.length() - 1);

    return postfix;
}
```

Odczytywanie wyrażenia znak po znaku

Funkcja `infix_postfix()` przedstawiona wcześniej jest zaprojektowana z założeniem, że poszczególne elementy wyrażenia (tzn. liczby, operatory, nawiasy) będą rozdzielone znakami przerwy.

Odczytywanie wyrażenia znak po znaku

Funkcja `infix_postfix()` przedstawiona wcześniej jest zaprojektowana z założeniem, że poszczególne elementy wyrażenia (tzn. liczby, operatory, nawiasy) będą rozdzielone znakami przerwy.

W celu uniknięcia tego założenia „produkcyjna" wersja funkcji `infix_postfix()` musi odczytywać wejściowe wyrażenie znak po znaku.

Odczytywanie wyrażenia znak po znaku

Funkcja `infix_postfix()` przedstawiona wcześniej jest zaprojektowana z założeniem, że poszczególne elementy wyrażenia (tzn. liczby, operatory, nawiasy) będą rozdzielone znakami przerwy.

W celu uniknięcia tego założenia „produkcyjna" wersja funkcji `infix_postfix()` musi odczytywać wejściowe wyrażenie znak po znaku.

Niestety powoduje to komplikację polegająca na tym, ze znak `'-'` może być interpretowany jako:

- operator dwuargumentowy,
- część zapisu liczby ujemnej,
- zmiana znaku podwyrażenia w nawiasie,

którą trzeba uwzględnić.

Odczytywanie wyrażenia znak po znaku – odejmowanie

W tym celu można wykorzystać obserwację, że znak '-' oznacza operator dwuargumentowy w dwóch przypadkach:

1. Jeżeli znajduje się w wyrażeniu bezpośrednio za nawiasem zamykającym (prawym).
2. Jeżeli znajduje się w wyrażeniu bezpośrednio za liczbą.

Odczytywanie wyrażenia znak po znaku – odejmowanie

W tym celu można wykorzystać obserwację, że znak `'-'` oznacza operator dwuargumentowy w dwóch przypadkach:

1. Jeżeli znajduje się w wyrażeniu bezpośrednio za nawiasem zamykającym (prawym).
2. Jeżeli znajduje się w wyrażeniu bezpośrednio za liczbą.

Wystarczy zatem wprowadzić zmienną, która będzie otrzymywać wartość `false` po odczytaniu z wyrażenia wejściowego liczby lub prawego nawiasu oraz wartość `true` po odczytaniu jakiegokolwiek innego symbolu.

Odczytywanie wyrażenia znak po znaku – odejmowanie

W tym celu można wykorzystać obserwację, że znak `'-'` oznacza operator dwuargumentowy w dwóch przypadkach:

1. Jeżeli znajduje się w wyrażeniu bezpośrednio za nawiasem zamykającym (prawym).
2. Jeżeli znajduje się w wyrażeniu bezpośrednio za liczbą.

Wystarczy zatem wprowadzić zmienną, która będzie otrzymywać wartość `false` po odczytaniu z wyrażenia wejściowego liczby lub prawego nawiasu oraz wartość `true` po odczytaniu jakiegokolwiek innego symbolu.

Wtedy, jeżeli ta zmienna ma wartość `false` i z wyrażenia zostanie odczytany znak `'-'`, to ten znak należy traktować jako operator dwuargumentowy (tzn. odejmowanie).

Odczytywanie wyrażenia znak po znaku – zmiana znaku

W przypadkach, w których znak '-' nie reprezentuje operatora argumentowego, powinien on być interpretowany jako:

- część zapisu liczby ujemnej,
- zmiana znaku podwyrażenia w nawiasie.

Odczytywanie wyrażenia znak po znaku – zmiana znaku

W przypadkach, w których znak '-' nie reprezentuje operatora argumentowego, powinien on być interpretowany jako:

- część zapisu liczby ujemnej,
- zmiana znaku podwyrażenia w nawiasie.

Drugi przypadek ma miejsce wtedy, gdy pierwszy (nie będący przerwą) znak za znakiem '-' jest nawiasem otwierającym (lewym).

Odczytywanie wyrażenia znak po znaku – zmiana znaku

W przypadkach, w których znak '-' nie reprezentuje operatora argumentowego, powinien on być interpretowany jako:

- część zapisu liczby ujemnej,
- zmiana znaku podwyrażenia w nawiasie.

Drugi przypadek ma miejsce wtedy, gdy pierwszy (nie będący przerwą) znak za znakiem '-' jest nawiasem otwierającym (lewym).

W takim przypadku w wyrażeniu wynikowym (w ONP) powinien być zapisany symbol reprezentujący zmianę znaku ostatnio obliczonego wyrażenia (np. $\sim$).

Odczytywanie wyrażenia znak po znaku – zmiana znaku

W przypadkach, w których znak '-' nie reprezentuje operatora argumentowego, powinien on być interpretowany jako:

- część zapisu liczby ujemnej,
- zmiana znaku podwyrażenia w nawiasie.

Drugi przypadek ma miejsce wtedy, gdy pierwszy (nie będący przerwą) znak za znakiem '-' jest nawiasem otwierającym (lewym).

W takim przypadku w wyrażeniu wynikowym (w ONP) powinien być zapisany symbol reprezentujący zmianę znaku ostatnio obliczonego wyrażenia (np. $\sim$).

Np. dla wyrażenia $-(2+3)$ odpowiednikiem w ONP może być $2\ \ 3\ +\ \sim$

Przetwarzanie wyrażenia znak po znaku – liczby

Odczytując wejściowe wyrażenie znak po znaku należy odpowiednio uwzględniać liczby.

Przetwarzanie wyrażenia znak po znaku – liczby

Odczytując wejściowe wyrażenie znak po znaku należy odpowiednio uwzględniać liczby.

W tym celu można zastosować następujący algorytm:

1. Przypuśćmy, że znaleźliśmy znak (być może) będący początkiem liczby.
2. Znajdujemy najbliższy znak przerwy lub nawias zamykający za nim.
3. Próbujemy odczytać liczbę z wycinka łańcucha wejściowego między tymi dwoma znakami.
4. Jeżeli operacja zakończy się powodzeniem, kopiujemy ten wycinek w całości (z „doklejonym” znakiem przerwy) do wynikowego łańcucha.
5. Przechodzimy do pierwszego znaku położonego za tym wycinkiem.

Zamiana zapisu wyrażeń na ONP i standardowe funkcje

Wygodnie jest uwzględniać standardowe funkcje (np. $\sin(x)$, $\cos(x)$) występujące w wyrażeniach przy zamianie zapisu tych wyrażeń na ONP.

Zamiana zapisu wyrażeń na ONP i standardowe funkcje

Wygodnie jest uwzględniać standardowe funkcje (np. $\sin(x)$, $\cos(x)$) występujące w wyrażeniach przy zamianie zapisu tych wyrażeń na ONP.

W tym celu można wykorzystać obserwację, że w każdym przypadku zapis funkcji standardowej składa się z nazwy (1 lub więcej znaków alfanumerycznych), bezpośrednio po której znajduje się nawias otwierający (lewy).

Zamiana zapisu wyrażeń na ONP i standardowe funkcje

Wygodnie jest uwzględniać standardowe funkcje (np. $\sin(x)$, $\cos(x)$) występujące w wyrażeniach przy zamianie zapisu tych wyrażeń na ONP.

W tym celu można wykorzystać obserwację, że w każdym przypadku zapis funkcji standardowej składa się z nazwy (1 lub więcej znaków alfanumerycznych), bezpośrednio po której znajduje się nawias otwierający (lewy).

W związku z tym standardową funkcję można traktować podobnie, jak nawias otwierający poprzedzony znakiem '-', oznaczającym zmianę znaku wyrażenia, lecz zamiast symbolu reprezentującego zmianę znaku w wynikowym łańcuchu powinien być zapisywany symbol reprezentujący daną funkcje (np. s dla $\sin(x)$ lub c dla $\cos(x)$).

Łączenie zamiany zapisu wyrażeń na ONP i obliczania ich

Procedurę zamiany wyrażenia w zapisie infiksowym na ONP (zapis postfiksowy) można połączyć z procedurą obliczania wyrażeń w zapisie postfiksowym.

Łączenie zamiany zapisu wyrażeń na ONP i obliczania ich

Procedurę zamiany wyrażenia w zapisie infiksowym na ONP (zapis postfiksowy) można połączyć z procedurą obliczania wyrażeń w zapisie postfiksowym.

Wtedy łańcuch wynikowy pochodzący z pierwszej procedury staje się łańcuchem wejściowym dla drugiej procedury.

Łączenie zamiany zapisu wyrażeń na ONP i obliczania ich

Procedurę zamiany wyrażenia w zapisie infiksowym na ONP (zapis postfiksowy) można połączyć z procedurą obliczania wyrażeń w zapisie postfiksowym.

Wtedy łańcuch wynikowy pochodzący z pierwszej procedury staje się łańcuchem wejściowym dla drugiej procedury.

Oczywiście jeśli pierwsza procedura uwzględnia zamianę znaku wyrażeń oraz funkcje standardowe, to druga procedura musi rozpoznawać oznaczające je symbole „wyprodukowane” przez pierwszą procedurę.

Łączenie zamiany zapisu wyrażeń na ONP i obliczania ich

Procedurę zamiany wyrażenia w zapisie infiksowym na ONP (zapis postfiksowy) można połączyć z procedurą obliczania wyrażeń w zapisie postfiksowym.

Wtedy łańcuch wynikowy pochodzący z pierwszej procedury staje się łańcuchem wejściowym dla drugiej procedury.

Oczywiście jeśli pierwsza procedura uwzględnia zamianę znaku wyrażeń oraz funkcje standardowe, to druga procedura musi rozpoznawać oznaczające je symbole „wyprodukowane” przez pierwszą procedurę.

Dodatkowo, na potrzeby obliczania wartości funkcji dla różnych wartości zmiennej niezależnej, należy uwzględnić zapis zmiennej niezależnej w wyrażeniu wejściowym i w ONP.

Literatura


B. Stroustrup, *Język C++* (Wydawnictwo Naukowo-Techniczne, Warszawa 2002).


B. Eckel, *Thinking in C++. Edycja polska* (Wydawnictwo Helion, Gliwice 2002).


Pang Tao, *Metody obliczeniowe w fizyce* (Wydawnictwo Naukowe PWN, Warszawa 2001).


T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Wprowadzenie do algorytmów* (Wydawnictwa Naukowo-Techniczne, Warszawa, 2001).