POLITECHNIKA GDAŃSKA

# Sprawozdanie z Projektu
# Metody Modelowania Matematycznego :
# Model wózka na sprężynach z wejściem w postaci siły i tarciem

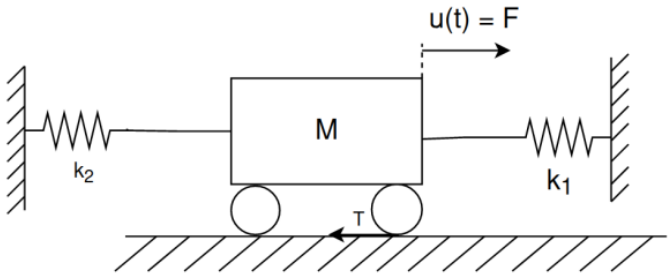
Maja Czarniecka 193327, Piotr Haber 193540

Gdańsk, 27.06.2025

## Spis treści

# 1 Opis zadania

Celem zadania było zaimplementowanie układu wózka na sprężynach w symulacji. Symulacja polega na wyrysowaniu przebiegów prędkości i położenia. Interfejs pozwala użytkownikowi wprowadzić parametry symulacji i na ich podstawie program symuluje przebiegi.




Rysunek 1: Model wózka na sprężynach

Układ musi spełniać równanie:

$$ma = u(t) - f_k(t) - T(t)$$

Przyjmując, że siła sprężystości $f_k$ jest proporcjonalna do odkształcenia sprężyny, a siła tarcia $T(t)$ proporcjonalna do prędkości, można zapisać:

$$v'(t) = \frac{1}{m}u(t) - \frac{k_1 + k_2}{m}x(t) - \mu v(t),$$

a zależność prędkości $v(t)$ wyrazić wzorem:

$$v(t) = x'(t)$$

Równania te pozwalają na wyznaczenie modelu stanowego.

## Model stanowy

Dla przeprowadzenia symulacji wyprowadzono model stanowy układu przedstawionego na Rysunku 1:

$$\begin{bmatrix} v'(t) \\ x'(t) \end{bmatrix} = \begin{bmatrix} \frac{\mu}{m} & -\frac{k1+k2}{m} \\ 1 & 0 \end{bmatrix} \begin{bmatrix} v(t) \\ x(t) \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u(t)$$

$$y(t) = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} v(t) \\ x(t) \end{bmatrix}$$

$x(t)$ - przemieszenie, $v(t)$ - prędkość, $u(t)$ - sygnał wejściowy, $y(t)$ - sygnał wyjściowy, $\mu$ - współczynnik proporcjonalności $\frac{T(t)}{v(t)}$

## Parametry symulacji

Zaimplementowany model widoczny jest na Rysunku 1. Parametry, którymi użytkownik może sterować, oraz ich zakresy przedstawione są w poniższej tabeli:

| Parametr | Wartość minimalna | Wartość maksymalna |
|---|---|---|
| **Parametry modelu** | | |
| Współczynnik sprężystości $k_1$ | 0.0 | 99.99 |
| Współczynnik sprężystości $k_2$ | 0.0 | 99.99 |
| Masa | 0.0 | 99.99 |
| Współczynnik tarcia | 0.0 | 99.99 |
| **Parametry symulacji** | | |
| Czas symulacji | 0.00 | 99.99 |
| Skok symulacji | 0.01 | 99.99 |
| Położenie początkowe | -99.99 | 99.99 |
| **Parametry siły wejściowej** $u(t)$ | | |
| Okres | 0.0 | 99.99 |
| Amplituda | 0.0 | 99.99 |
| Faza | -99.99 | 99.99 |
| Przesunięcie | -99.99 | 99.99 |
| Współczynnik wypełnienia | 0.0 | 1.0 |

Tabela 1: Zakresy parametrów symulacji

Zakresy wartości parametrów zostały dobrane losowo (z zachowaniem logiki takiej jak np. czas trwania symulacji nie może być ujemny), a ich zmiana nie wpłynie negatywnie na działanie programu, wymaga jedynie zmiany w kodzie źródłowym.

# 2 Metody symulacji

## 2.1 Metoda Eulera

Do wykonania symulacji wykorzystano 2 algorytmy do rozwiązywania równań różniczkowych. Pierwszym z nich był algorytm Eulera. Jest to algorytm rozwiązywania równań różniczkowych zwyczajnych przy znajomości warunków początkowych.

### Idea

Metoda Eulera opiera się na interpretacji geometrycznej równania różniczkowego. Polega na "przewidywaniu"wartości funkcji dla czasu $t_{n+1}$ przez wyznaczanie nachylenia funkcji oraz stycznej w punkcie $y_n$ a następnie obliczeniu punktu $y_{n+1}$.

### Opis matematyczny

Dany jest problem opisany równaniem różniczkowym:

$$\frac{dy}{dt} = f(t, y)$$

Znany jest również warunek początkowy $y(t_0) = y_0$. Funkcja $y(t)$ jest funkcją, której przebieg chcemy przybliżyć metodą Eulera. Metoda Eulera jest metodą iteracyjną, zatem w pierwszej iteracji $y_{n=0} = y(t_0)$.

Wybieramy arbitralny krok czasu $h$ i dla $n = 0, 1, 2, ..,$ i liczymy przybliżone wartości funkcji w następnych iteracjach:

$$y_{n+1} = y_n + hf(t_n, y_n) \tag{1}$$

Wartość $y_n$ jest przybliżoną wartością funkcji dla $t_n$

### Implementacja w projekcie

Do realizacji symulacji należało rozwiązać następujące równania różniczkowe:

$$\frac{dv(t)}{dt} = \frac{1}{m}u(t) - \frac{k_1 + k_2}{m}x(t) - \frac{\mu}{m}v(t)$$
$$\frac{dx(t)}{dt} = v(t)$$

Ze względu na to, że zamodelowany układ jest układem drugiego rzędu, należało wprowadzić odpowiednie zmiany w algorytmie.

Zamiast jednego bufora do którego dodajemy kolejne wartości $y_{n+1}$ powstają dwa bufory: `result_a` związany z funkcją `func_a`, i `result_b` - związany z funkcją `func_b`

W naszym przypadku `func_a` odpowiada funkcji $\frac{dx(t)}{dt}$ a `func_b` funkcji $\frac{dv(t)}{dt}$. Zatem bufor `result_a` będzie wynikowym przybliżeniem przebiegu $x(t)$ a bufor `result_b` - przybliżonym przebiegiem $v(t)$.

Kolejną różnicą między rozwiązywanym problemem a ideowym opisem matematycznym metody Eulera jest fakt, że w przypadku rozważanego modelu fukcja $f$ przyjmuje więcej parametrów.

```
void MainWindow::solveEuler(double initial_a, double initial_b, double step,
    double (*func_a)(double, double, double, Force),
    double (*func_b)(double, double, double, Force),
    std::vector<double>& result_a, std::vector<double>& result_b,
    Force input_force)
    {
        result_a.clear();
        result_b.clear();

        result_a.push_back(initial_a);
        result_b.push_back(initial_b);

        for (size_t i = 0; i < m_t.size() - 1; ++i)
        {
            double da_dt = func_a(m_t[i], result_a[i], result_b[i],
               input_force);
            double db_dt = func_b(m_t[i], result_a[i], result_b[i],
               input_force);

            double next_a = result_a[i] + step * da_dt;
            double next_b = result_b[i] + step * db_dt;

            result_a.push_back(next_a);
            result_b.push_back(next_b);
        }
    }
```

Listing 1: Implementacja algorytmu Eulera dla układu 2. rzędu w języku C++

## 2.2 Runge-Kutta 4. rzędu

### Idea

Drugą z wykorzystanych metod był algorytm Rungego-Kutty 4. rzędu (RK4). Metoda ta podobnie jak metoda Eulera "przewiduje"wartości funkcji dla czasu $t + \Delta t$, jednak robi to o wiele dokładniej. Za-

miast "przewidywania"wartości na podstawie tylko i wyłącznie aktualnej trajektorii funkcji, obliczane są współczynniki biorące pod uwagę przyszłe jej przebiegi. Następnie z odpowiednią wagą są one dodawane do wartości funkcji dla czasu $t$.

**Opis matematyczny**

Dany jest problem opisany równaniem różniczkowym:

$$\frac{dy}{dt} = f(t, y)$$

Znany jest również warunek początkowy $y(t_0) = y_0$. Funkcja $y(t)$ jest funkcją, której przebieg chcemy przybliżyć metodą RK4. Metoda RK4 jest metodą iteracyjną zatem w pierwszej iteracji $y_{n=0} = y(t_0)$.

Wybieramy arbitralny krok czasu $h$ i dla $n = 0, 1, 2, ..$, wyliczamy wcześniej wspomniane współczynniki:

$$\begin{aligned}
k_1 &= f(t_n, y_n), \\
k_2 &= f(t_n + \frac{h}{2}, y_n + h \cdot \frac{k_1}{2}), \\
k_3 &= f(t_n + \frac{h}{2}, y_n + h \cdot \frac{k_2}{2}), \\
k_4 &= f(t_n + h, y_n + h \cdot k_3)
\end{aligned}$$

Następnie korzystając z wyliczonych współczynników wyliczamy:

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

Ważnym jest aby pamiętać, że wartość $y_{n+1}$ jest **tylko przybliżeniem** wartości $y(t + h)$.
Wartym uwagi jest również fakt, że parametr $k_1$ jest krokiem Eulera, który wykorzystywany jest w pierwszej z użytych w projekcie metod.

### 2.2.1 Implementacja w projekcie

Tak samo jak dla metody Eulera zostało przyjęte nazewnictwo zmiennych literami `a` oraz `b`. Konwencja nazewnictwa, w której zmienne oznaczone literą `a` powiązane są z przebiegiem położenia wózka, a literą `b` z przebiegiem prędkości wózka jest utrzymana również dla obliczenia parametrów $k$.

```cpp
void MainWindow::solveRK4(double initial_a, double initial_b, double step,
    double (*func_a)(double, double, double, Force),
    double (*func_b)(double, double, double, Force),
    std::vector<double>& result_a, std::vector<double>& result_b,
    Force input_force)
    {
        result_a.clear();
        result_b.clear();

        result_a.push_back(initial_a);
        result_b.push_back(initial_b);

        double k1a, k1b, k2a, k2b, k3a, k3b, k4a, k4b;
        for (int i = 0; i < m_t.size(); i++)
        {
            k1a = func_a(m_t[i], result_a[i], result_b[i], input_force);
            k1b = func_b(m_t[i], result_a[i], result_b[i], input_force);

            k2a = func_a(m_t[i] + 0.5*step, result_a[i] + 0.5*step*k1a,
               result_b[i] + 0.5*k1b*step, input_force);
            k2b = func_b(m_t[i] + 0.5*step, result_a[i] + 0.5*step*k1a,
               result_b[i] + 0.5*k1b*step, input_force);

            k3a = func_a(m_t[i] + 0.5*step, result_a[i] + 0.5*step*k2a,
               result_b[i] + 0.5*step*k2b, input_force);
            k3b = func_b(m_t[i] + 0.5*step, result_a[i] + 0.5*step*k2a,
               result_b[i] + 0.5*step*k2b, input_force);

            k4a = func_a(m_t[i] + step, result_a[i] + step*k3a, result_b[i]
               + step*k3b, input_force);
            k4b = func_b(m_t[i] + step, result_a[i] + step*k3a, result_b[i]
               + step*k3b, input_force);

            result_a.push_back(result_a[i] + 1.0/6.0*step*(k1a + 2*k2a +
               2*k3a + k4a));
            result_b.push_back(result_b[i] + 1.0/6.0*step*(k1b + 2*k2b +
               2*k3b + k4b));
        }
    }
```

Listing 2: Implementacja algorytmu RK4 dla układu 2. rzędu w języku C++

# 3 Wnioski

## 3.1 Wpływ znaczących parametrów na symulację

### Współczynniki sprężystości

Zwiększanie współczynnika sprężystości powoduje zmniejszanie amplitudy oscylacji oraz zwiększenie częstotliwości drgań. Zauważalna jest również szybsza stabilizacja obiektu.

### Współczynnik tarcia

Wzrost współczynnika tarcia znacząco wpływa na zmniejszenie się amplitudy oraz częstotliwości oscylacji. Zauważalne jest też to, że obiekt stabilizuje się po wykonaniu mniejszej ilości drgań.

### Masa

Zwiększanie masy wózka powoduje przede wszystkim zmniejszenie amplitudy oscylacji na wyjściu układu - aby przesunąć wózek o tę samą odległość, potrzebna jest większa siła. Dla niektórych konfiguracji łatwo jest również zauważyć zmianę okresu drgań (np. dla pobudzenia wartością stałą)

### Skok symulacji

Zwiększanie skoku symulacji powoduje utratę dokładności wyników liczonych metodą Eulera. Im większy skok, tym bardziej niezgodne z rzeczywistymi przebiegami będą wyniki.

## 3.2 Porównanie metody Eulera i RK4

Realizacja projektu pozwoliła wykazać, że metoda Eulera, mimo że jest łatwa w zrozumieniu jak i w użyciu, wiąże się z pewnymi problemami. Zauważono, że metoda Eulera jest metodą niestabilną. Rozumiemy przez to, że dla długo trwającej symulacji wynik będzie coraz bardziej odbiegać od wartości rzeczywistej. Efekt ten można osłabiać, ustawiając o wiele mniejszy krok symulacji, zwiększając tym efektywnie jej dokładność, jednak prowadzi to do problemów. Mniejszy krok symulacji oznacza większą liczbę obliczeń wymaganych do zasymulowania tego samego okresu czasu. Wydłuża to pracę programu. Nie jest to problemem w symulatorze podobnym do tego zaimplementowanego w projekcie - użytkownik będzie musiał po prostu dłużej poczekać, jednak do symulacji w czasie rzeczywistym bardzo duża ilość obliczeń jest niepożądana.

Dodatkowo zauważono, że wpływ na te niedokładności ma nie tylko skok symulacji. Zmieniając parametry w taki sposób, aby tarcie miało mniejszy wpływ na sygnały wyjściowe, na przykład zwiększając amplitudę siły lub współczynniki sprężystości (charakter wyjścia zbliża się wtedy do niegasnących oscylacji), jesteśmy w stanie osiągnąć ten sam efekt - wynik symulacji metodą Eulera zaczyna mocno odbiegać od rzeczywistych przebiegów, a w niektórych przypadkach powodować nawet wynik sugerujący rosnące do nieskończoności oscylacje, mimo że układ w rzeczywistości nie będzie się tak zachowywać.

Dlatego metoda RK4 jest tą częściej stosowaną - parametry symulacji nie wpływają w znacznym stopniu na dokładność jej wyników.

Metoda RK4 jest metodą dokładniejszą, ze względu na to, że bierze ona pod uwagę nie tylko krok w kierunku stycznej do punktu, w którym aktualnie się znajdujemy - dodatkowo sprawdza ona zachowanie tej funkcji w "otoczeniu"tego punktu, i na podstawie tych kalkulacji, z dobraniem odpowiednich wag poszczególnych parametrów, z większą dokładnością jest w stanie wyznaczyć rozwiązanie.