Symulacja układu dynamicznego z regulatorem PD

Dokumentacja projektu

1. Cel projektu

Celem projektu była implementacja symulacji układu dynamicznego opisanego transmitancją drugiego rzędu oraz sterowanie tym układem przy użyciu regulatora PD. Wszystkie obliczenia zostały wykonane z użyciem manualnie zaimplementowanych metod numerycznych w języku Python, bez wykorzystania zaawansowanych bibliotek symulacyjnych.

2. Opis modelu układu

Układ został opisany transmitancją typu:

$$G(s) = \frac{a_1 s + a_0}{b_2 s^2 + b_1 s + b_0}$$

Został on przekształcony do postaci równań stanu:

$$\dot{x}_1 = x_2$$

$$\dot{x}_2 = \frac{-b_1 x_2 - b_0 x_1 + a_1 \dot{u} + a_0 u}{b_2}$$

Do symulacji zastosowano numeryczną metodę Eulera:

$$x(t + dt) = x(t) + \dot{x}(t) \cdot dt$$

3. Regulator PD

Zastosowano regulator PD (proporcjonalno-różniczkujący), którego równanie ma postać:

$$u(t) = K_p \cdot e(t) + K_d \cdot \frac{de(t)}{dt}$$

Błąd regulacji obliczany jest jako:

$$e(t) = r(t) - y(t)$$

a pochodna błędu została przybliżona numerycznie:

$$\frac{de(t)}{dt} \approx \frac{e(t) - e(t - dt)}{dt}$$

4. Generowanie sygnałów wejściowych

Zaimplementowano trzy typy sygnałów wejściowych:

- **Prostokątny** – na podstawie operatora modulo,
- **Trójkątny** – z użyciem wartości bezwzględnej,
- **Harmoniczny** – funkcja sinusoidalna.

5. Metody numeryczne

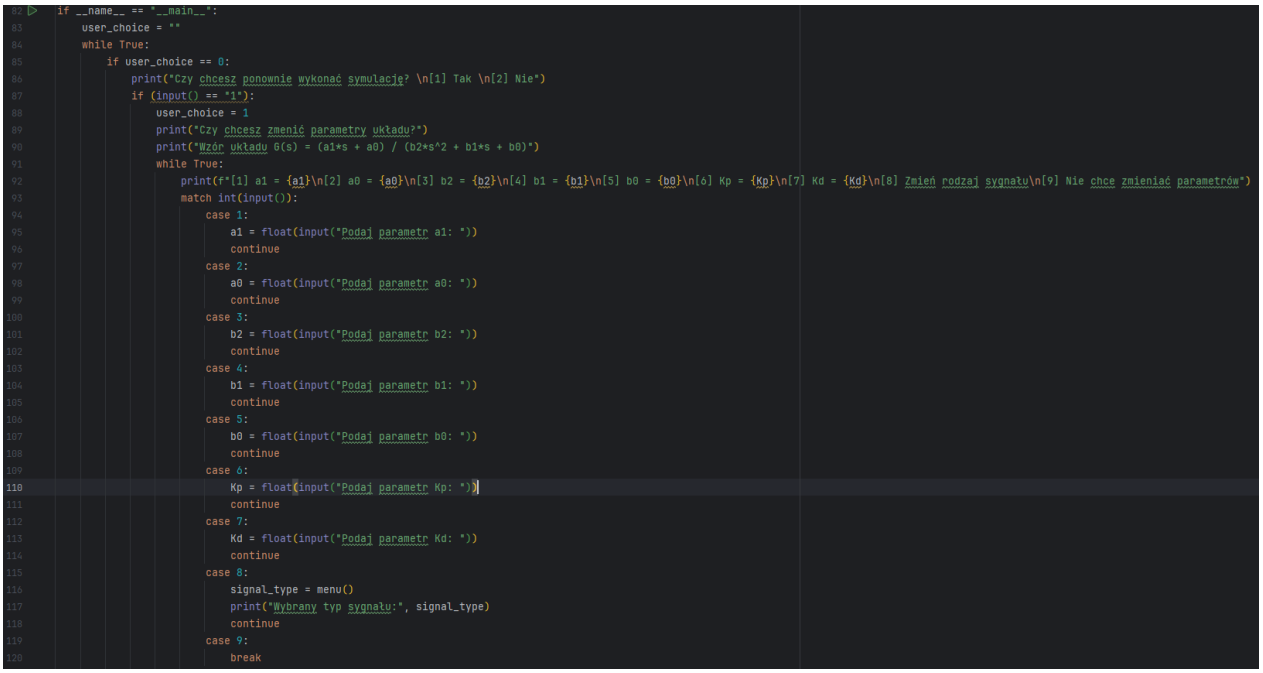
Symulację przeprowadzono przy użyciu metody Eulera – jednej z podstawowych metod całkowania równań różniczkowych. Jej prostota pozwala na szybkie wykonanie symulacji, choć kosztem dokładności i stabilności w przypadku bardziej złożonych układów.

6.Omównienie kodu

1) Stworzono prosty interfejs tekstowy z menu wyboru rodzaju sygnału oraz edycji parametrów układu i regulatora. Implementacja wyboru została wykonana w stylu `switch-case`, umożliwiając wygodne sterowanie przebiegiem symulacji.

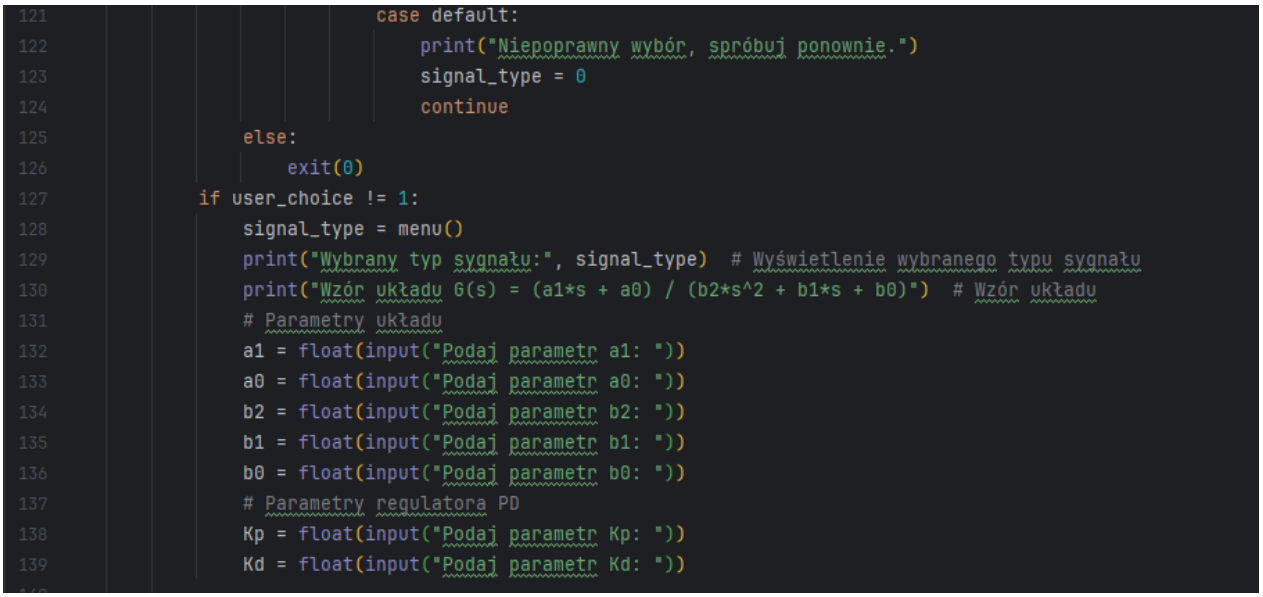
```
def menu():  2 usages  Tomasz Nazar
    menu_text = """
========= Wybierz typ sygnału wejściowego =========
\t[1] Prostokątny
\t[2] Trójkątny
\t[3] Harmoniczny
\t[q] Wyjście
================================================
"""
    # Wyświetlenie menu wyboru sygnału

    while True:
        choice = input(menu_text)
        if choice == '1':
            return "rectangular"
        elif choice == '2':
            return "triangular"
        elif choice == '3':
            return "harmonic"
        elif choice == 'q':
            exit(0)
        else:
            print("Niepoprawny wybór, spróbuj ponownie.")
```

Rysunek 1: Menu (1)



```
if __name__ == "__main__":
    user_choice = ""
    while True:
        if user_choice == 0:
            print("Czy chcesz ponownie wykonać symulację? \n[1] Tak \n[2] Nie")
            if (input() == "1"):
                user_choice = 1
                print("Czy chcesz zmenić parametry układu?")
                print("Wzór układu G(s) = (a1*s + a0) / (b2*s^2 + b1*s + b0)")
                while True:
                    print(f"[1] a1 = {a1}\n[2] a0 = {a0}\n[3] b2 = {b2}\n[4] b1 = {b1}\n[5] b0 = {b0}\n[6] Kp = {Kp}\n[7] Kd = {Kd}\n[8] Zmień rodzaj sygnału\n[9] Nie chce zmieniać parametrów")
                    match int(input()):
                        case 1:
                            a1 = float(input("Podaj parametr a1: "))
                            continue
                        case 2:
                            a0 = float(input("Podaj parametr a0: "))
                            continue
                        case 3:
                            b2 = float(input("Podaj parametr b2: "))
                            continue
                        case 4:
                            b1 = float(input("Podaj parametr b1: "))
                            continue
                        case 5:
                            b0 = float(input("Podaj parametr b0: "))
                            continue
                        case 6:
                            Kp = float(input("Podaj parametr Kp: "))
                            continue
                        case 7:
                            Kd = float(input("Podaj parametr Kd: "))
                            continue
                        case 8:
                            signal_type = menu()
                            print("Wybrany typ sygnału:", signal_type)
                            continue
                        case 9:
                            break
```

Rysunek 2: Menu (2)



```
                        case default:
                            print("Niepoprawny wybór, spróbuj ponownie.")
                            signal_type = 0
                            continue
            else:
                exit(0)
        if user_choice != 1:
            signal_type = menu()
            print("Wybrany typ sygnału:", signal_type)  # Wyświetlenie wybranego typu sygnału
            print("Wzór układu G(s) = (a1*s + a0) / (b2*s^2 + b1*s + b0)")  # Wzór układu
            # Parametry układu
            a1 = float(input("Podaj parametr a1: "))
            a0 = float(input("Podaj parametr a0: "))
            b2 = float(input("Podaj parametr b2: "))
            b1 = float(input("Podaj parametr b1: "))
            b0 = float(input("Podaj parametr b0: "))
            # Parametry regulatora PD
            Kp = float(input("Podaj parametr Kp: "))
            Kd = float(input("Podaj parametr Kd: "))
```

Rysunek 3: Menu (3)

2) Symulacja układu dynamicznego została zrealizowana w klasie `TransferFunctionSimulator`, wykorzystującej metodę Eulera do rozwiązania równań stanu. Obliczenia wykonywane są iteracyjnie, a odpowiedź układu $y(t)$ uzyskiwana jest na podstawie zadanego sygnału wejściowego $u(t)$. Dzięki tej implementacji możliwa jest szybka i elastyczna analiza zachowania układu drugiego rzędu.

```python
class TransferFunctionSimulator:  1 usage  Tomasz Nazar *
    def __init__(self, a1, a0, b2, b1, b0, dt=0.01, T=10):  Tomasz Nazar
        # Inicjalizacja parametrów układu
        self.a1, self.a0 = a1, a0
        self.b2, self.b1, self.b0 = b2, b1, b0
        self.dt = dt  # Krok czasowy symulacji
        self.T = T  # Całkowity czas symulacji
        self.time = np.arange(0, T, dt)  # Wektor czasu
        self.y = np.zeros_like(self.time)  # Inicjalizacja odpowiedzi układu
        self.u = np.zeros_like(self.time)  # Inicjalizacja sygnału wejściowego

    def simulate(self, u):  1 usage  Tomasz Nazar *
        self.u = u
        x1, x2 = 0, 0
        for i in range(1, len(self.time)):
            x1_dot = x2
            u_dot = (u[i] - u[i - 1]) / self.dt
            x2_dot = (-self.b1 * x2 - self.b0 * x1 + self.a1 * u_dot + self.a0 * u[i]) / self.b2
            x1 += x1_dot * self.dt
            x2 += x2_dot * self.dt
            self.y[i] = x1
        return self.time, self.y
```

Rysunek 4: Symulacja układu

3) Regulator PD został zaimplementowany w klasie `PDController`. Oblicza on wartość sterowania na podstawie bieżącego błędu oraz jego przybliżonej pochodnej, zgodnie ze wzorem $u(t) = K_p \cdot e(t) + K_d \cdot \frac{de(t)}{dt}$. Przechowywanie poprzedniego błędu umożliwia prostą aproksymację pochodnej.

Dodatkowo zaimplementowano funkcję `generate_signal`, umożliwiającą generowanie trzech typów sygnałów wejściowych: prostokątnego, trójkątnego oraz harmonicznego. Ich kształt zależy od parametrów takich jak amplituda, częstotliwość i okres trwania.

```python
class PDController:  1 usage
    def __init__(self, Kp, Kd):
        # Inicjalizacja parametrów regulatora PD
        self.Kp, self.Kd = Kp, Kd
        self.prev_error = 0  # Poprzedni błąd dla wyliczenia pochodnej

    def control(self, ref, y, dt):  1 usage
        # Obliczanie wartości sterowania na podstawie regulatora PD
        error = ref - y  # Obliczenie błędu
        de = (error - self.prev_error) / dt  # Przybliżona pochodna błędu
        self.prev_error = error  # Aktualizacja błędu
        return self.Kp * error + self.Kd * de  # Wyliczenie sygnału sterującego


def generate_signal(signal_type, time, amplitude=1, frequency=1, duration=1):  1 usage
    # Generowanie różnych typów sygnałów wejściowych
    if signal_type == "rectangular":
        return amplitude * ((time % (2 * duration)) < duration)  # Sygnał prostokątny
    elif signal_type == "triangular":
        return amplitude * (2 * np.abs((time / duration) % 2 - 1) - 1)  # Sygnał trójkątny
    elif signal_type == "harmonic":
        return amplitude * np.sin(2 * np.pi * frequency * time)  # Sygnał harmoniczny
    else:
        raise ValueError("Unknown signal type")  # Obsługa błędnego typu sygnału
```

Rysunek 5: Regulator PD

4) Główna część programu odpowiada za przeprowadzenie symulacji z zastosowaniem regulatora PD. Na początku inicjalizowany jest symulator układu oraz regulator, a następnie generowany jest sygnał referencyjny – jego typ można zmieniać (prostokątny, trójkątny, harmoniczny).

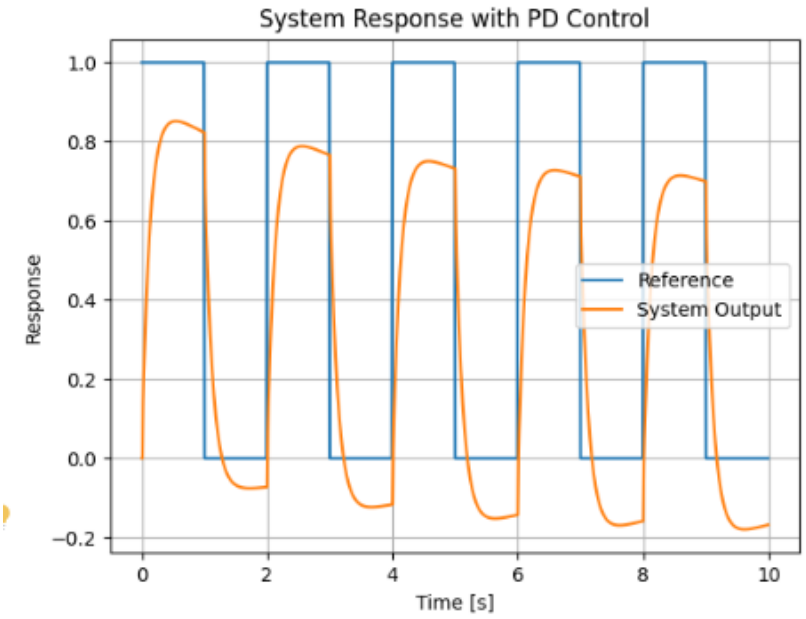
W pętli czasowej dla kolejnych kroków obliczana jest wartość sterowania na podstawie błędu regulacji, po czym wykonywana jest symulacja odpowiedzi układu. Wyniki w postaci odpowiedzi układu i sygnał

```
# Parametry symulacji
dt, T = 0.01, 10  # Krok czasowy i całkowity czas symulacji

simulator = TransferFunctionSimulator(a1, a0, b2, b1, b0, dt, T)  # Inicjalizacja symulatora układu
controller = PDController(Kp, Kd)  # Inicjalizacja regulatora PD

# Można zmienić na "rectangular" lub "triangular"
reference = generate_signal(signal_type, simulator.time)  # Generowanie sygnału referencyjnego

y_output = np.zeros_like(simulator.time)  # Inicjalizacja wektora odpowiedzi układu
u_control = np.zeros_like(simulator.time)  # Inicjalizacja sygnału sterującego

for i in range(1, len(simulator.time)):
    u_control[i] = controller.control(reference[i], y_output[i - 1], dt)  # Obliczenie wartości sterowania
    _, y_output = simulator.simulate(u_control)  # Symulacja układu dla sygnału sterującego

plt.figure()
plt.plot( *args: simulator.time, reference, label="Reference")  # Wykres sygnału referencyjnego
plt.plot( *args: simulator.time, y_output, label="System Output")  # Wykres odpowiedzi układu
plt.legend()
plt.xlabel("Time [s]")
plt.ylabel("Response")
plt.title("System Response with PD Control")
plt.grid()
plt.show()  # Wyświetlenie wykresu
user_choice = 0
```

Rysunek 6: Pętla i wyświetlanie wykresów

```
Aktualne parametry:

a1 = 0.01, a0 = 1.0

b2 = 1.0, b1 = 1.0, b0 = 1.0

Kp = 1.0, Kd = 7.0

Typ sygnału: rectangular


Czy chcesz ponownie wykonać symulację?

[1] Tak

[2] Nie
```

Rysunek 7: Przykładowy wykres prostokątny (1)



Rysunek 8: Przykładowy wykres prostokątny (2)

7. Wnioski z pracy projektowej

1. **Regulator PD** przy odpowiednich parametrach skutecznie zmniejsza uchyb regulacji i tłumi oscylacje odpowiedzi układu. Poprawia dynamikę systemu.

2. **Ujemne sprzężenie zwrotne**, zostało zastosowane w postaci klasycznego wzoru $e(t) = r(t) - y(t)$.

3. **Metoda Eulera**, mimo swojej prostoty i intuicyjności, cechuje się ograniczoną dokładnością i może prowadzić do błędów numerycznych przy większych krokach czasowych. Dla bardziej wymagających zastosowań warto rozważyć metody wyższego rzędu, takie jak Rungego-Kutty.

4. **Elastyczność symulatora** — możliwość łatwej zmiany parametrów transmitancji oraz rodzaju sygnałów wejściowych pozwala testować różne scenariusze i zachowania układu.

5. **Walor edukacyjny** — projekt na zrozumienie działania regulatora PD „od podstaw", w tym sposobu wyznaczania pochodnej błędu, działania transmitancji jako operatorowego modelu układu oraz problematyki numerycznego rozwiązywania równań różniczkowych.

6. **Możliwości rozbudowy** obejmują m.in. dodanie graficznego interfejsu użytkownika (GUI), wprowadzenie zapisu wyników do plików, rozszerzenie o inne typy regulatorów (PI, PID), wprowadzenie zakłóceń lub opóźnień, oraz zaimplementowanie dokładniejszych metod numerycznych.

7. **Zastosowanie praktyczne** — choć projekt ma charakter edukacyjny, to taka struktura (model + regulator + symulacja odpowiedzi) stanowi fundament rzeczywistych systemów automatyki, np. w przemyśle, robotyce czy mechatronice.

8. Zakończenie

Projekt stanowi praktyczne zastosowanie wiedzy z zakresu:

- modelowania układów dynamicznych,
- projektowania regulatorów,
- stosowania metod numerycznych w automatyce,