Politechnika
Śląska

POLITECHNIKA ŚLĄSKA
WYDZIAŁ AUTOMATYKI, ELEKTRONIKI I INFORMATYKI
KIERUNEK AUTOMATYKA I ROBOTYKA

Projekt inżynierski

Sprzętowa implementacja regulatora MPC

Autor: Szymon Zosgórnik
Kierujący pracą: dr hab. inż., prof. PŚ Jarosław Śmieja

Gliwice, styczeń 2020

Załącznik Nr 2 do Zarz. Nr 97/08/09

O Ś W I A D C Z E N I E

Wyrażam zgodę/nie wyrażam* zgody na udostępnienie mojej pracy dyplomowej/rozprawy doktorskiej*

……., dnia ……..……

……..……

(podpis)

……

(poświadczenie wiarygodności podpisu przez Dziekanat)

** właściwe podkreślić*

Streszczenie

Tematem pracy jest sprzętowa implementacja algorytmu sterowania predykcyjnego (MPC). Regulacja MPC jest to proces złożony obliczeniowo. W związku z tym istnieje ryzyko braku odpowiedzi mikrokontrolera w wyznaczonym czasie. Postanowiono przyjąć pewne obostrzenia w celu sprawdzenia wpływu parametrów regulatora na jakość odpowiedzi układu. Założono znajomość dyskretnego modelu obiektu. Ponadto przyjęto jego liniowy charakter. Zaimplementowano stosowny algorytm regulacji predykcyjnej przy użyciu języka programowania C++ oraz platformy STM. Przeprowadzono testy Hardware in the loop (HIL) z wykorzystaniem skryptu napisanego w języku programowania Python mające na celu zweryfikowanie poprawności zaimplementowanego algorytmu. Zebrane obserwacje potwierdziły teorię regulacji predykcyjnej. Efekty pracy mogą stanowić punkt wyjścia dla rozwoju opracowanego przykładu o bardziej zaawansowane metody, takie jak identyfikacja sterowanego obiektu.

Spis treści

Rozdział 1. Wstęp

1.1 Motywacja projektu

Regulacja MPC (Model Predictive Control) jest szybko rozwijającą się częścią automatyki w ostatnich latach. Jak większość algorytmów regulacji sterowanie predykcyjne powinno funkcjonować w wbudowanych systemach czasu rzeczywistego. W związku z tym wymagana jest możliwość pracy przy ograniczonych zasobach sprzętowych.

1.2 Cel pracy

Celem pracy jest implementacja algorytmu sterowania predykcyjnego na wybranym mikrokontrolerze, a następnie przeprowadzenie weryfikacji jakości wyznaczonych sterowań oraz otrzymanych odpowiedzi układu regulacji na zadaną wartość. Na pełny zakres teorii regulacji MPC składa się wiele zagadnień. W badanym przypadku postanowiono ograniczyć to spektrum do jednego z najmniej złożonych przypadków, aby sprawdzić wpływ podstawowych parametrów regulatora predykcyjnego na przyjęte kryteria badań. Zakłada się znajomość modelu matematycznego dyskretnego obiektu regulacji oraz jego liniowy charakter.

Rozdział 2. Idea regulatora MPC

2.1 Wstęp

Model predictive control (MPC) jest to zaawansowana metoda sterowania, która polega na takim dobraniu sterowania, aby spełniało ono szereg ograniczeń. Od lat 80. XX wieku algorytm ten wykorzystywany jest przemyśle procesowym w zakładach chemicznych i rafineriach ropy naftowej. W ostatnich latach MPC znalazło zastosowanie także w elektrowniach i elektronice mocy. Sterowanie predykcyjne wykorzystuje dynamiczny model obiektu, najczęściej jest to empiryczny model pozyskany za pomocą indetyfikacji systemów. Główną zaletą MPC jest optymalizacja obecnego przedziału czasowego, biorąc pod uwagę przyszłe stany obiektu. Jest to osiągnięte poprzez optymalizację funkcji jakości w przedziale skończonego horyzontu czasowego, ale z wykorzystaniem jedynie sterowania wyliczonego dla obecnej chwili czasu. Proces ten jest powtarzany z każdą iteracją algorytmu rozwiązującego układ równań różniczkowych opisujących dany układ. Taki schemat regulacji powoduje, że istnieje możliwość przewidzenia przyszłych zdarzeń (występujących zgodnie z podanym modelem wartości zadanej) i podjęcia odpowiednich działań regulujących pracę układem we wcześniejszych chwilach. Sterowanie predykcyjne jest zazwyczaj zaimplementowane jako dyskretny regulator, lecz obecnie prowadzone są badania mające na celu uzyskanie szybszej odpowiedzi przy użyciu specjalnie do tego przygotowanych układów analogowych.

2.2 Sposób działania

Zasada pracy regulatora MPC polega na minimalizacji różnic między wartościami predykowanymi: $y_{k+i|k}$ w chwili obecnej k na przyszłą $k+i$, a wartościami zadanymi dla tych wyjść $r(i)$. Przez minimalizację tychże różnic rozumiana jest minimalizacja określonego kryterium jakości J. W następnej chwili czasu $(k+1)$ następuje kolejny pomiar sygnału na wyjściu obiektu, a cała procedura powtarzana jest z takim samym horyzontem predykcji N_p. W tym celu stosowana jest więc zasada sterowania repetycyjnego bazującego na przesuwnym horyzoncie czasu. W algorytmie regulacji MPC obecny jest także tzw. horyzont sterowania N_c (gdzie $N_c \leqslant N_p$), po którego upływie przyrost sygnału sterującego wynosi zero. W ten sposób zapewnio-

ne są własności całkujące układu regulacji predykcyjnej.
Algorytmy MPC cechują się następującymi wymogami i właściwościami:

- Wymaganie wyznaczenia wartości przyszłych sygnału sterującego.
- Sterowanie według zdefiniowanej trajektorii referencyjnej dla wielkości wyjściowej.
- Uwzględnienie przyszłych zmian wartości zadanej. Wcześniejsza reakcja regulatora na przyszłą zmianę wartości referencyjnej kompensuje negatywny wpływ opóźnienia na działanie układu.
- Stabilna regulacja obiektów, które nie są minimalnofazowe bez uwzględnienia tego faktu podczas syntezy regulatora.

Realizację metody sterowania predykcyjnego można zapisać w czterech następujących krokach:

1. Pomiar lub estymacja aktualnego stanu obiektu.
2. Obliczenie przyszłych próbek wyjść systemu.
3. Zaaplikowanie sygnałów sterujących tylko do następnej chwili czasu.
4. Powtórzenie algorytmu dla kolejnej chwili czasu.

2.3 Model obiektu

Do poprawności działania regulatora MPC niezbędna jest identyfikacja modelu obiektu, który ma być wysterowany. Obecnie wykorzystuje się model w postaci równań stanu, podczas gdy w przeszłości korzystano z modelu odpowiedzi skokowej. Takie podejście wymaga także zaprojektowania obserwatora stanu, używając do tego metod znanych z teorii sterowania. Model obiektu może być zarówno liniowy, jak i nieliniowy. Jednakże, użycie modelu nieliniowego prowadzi do nieliniowej optymalizacji, co powoduje zwieloktrotnienie trudności obliczeniowej. Przekłada się to na zwiększenie wymagań odnośnie częstotliwości taktowania procesora w implementacji sprzętowej. Wynika z tego stwierdzenie, że modele liniowe mają największe znaczenie praktyczne z uwagi na możliwość przeprowadzenia obliczeń w czasie rzeczywistym nawet bez wygórowanych wymagań hardware'owych. Rozwiązaniem tego problemu jest zastosowanie regultaora predykcyjnego w połączeniu z linearyzacją modelu obiektu w konkretnym punkcie pracy. Następnie wyznaczone

są sterowania tak jak dla liniowego przypadku. Tak zrealizowany algorytm gwarantuje jedynie rozwiązanie suboptymalne, jednak nie rzutuje to w żaden sposób na przydatność jego realizacji.

2.4 Kryterium jakości regulacji

Jak już wcześniej pokazano w rozdziale 2.2 w celu wyznaczenia wartości sterowań w obecnej i następnych chwilach wyznacza się minimum funkcji celu. Funkcja ta określa jakość pracy regulatora na horyzoncie predykcji. Można stwierdzić, że wartość sygnału sterującego jest wyznaczana poprzez minimalizację wskaźnika jakości regulacji, który jest inheretnie związany z predykcją wyjścia obiektu.
W przypadku skalarnym funkcję celu można opisać następującym równaniem:

$$\begin{cases} J = R_y \sum_{i=1}^{N_p} (r_k - y_{i|k})^2 + R_u \sum_{i=1}^{N_c} (u_{i|k} - u_{k-1})^2 \\ x_{k+1} = Ax_k + Bu_k \\ y_k = Cx_k \end{cases} \tag{2.1}$$

$x_k =$ wektor zmiennych stanu w chwili k

$y_k =$ zmienna wyjściowa w chwili$k$

$r_k =$ zmienna referencyjna w chwili$k$

$u_k =$ sterowanie w chwili k

$N_p =$ horyzont predykcji

$N_c =$ horyzont sterowań

$i|k =$ predykcja w chwili k odnosząca się do chwili i

$R_y =$ współczynnik wagowy wyjścia y

$R_u =$ współczynnik wagowy sterowania$u$

$A, B, C =$ macierze przestrzeni stanu

2.5 Problem programowania kwadratowego

$$J = \frac{1}{2} U^T H U + W^T U \tag{2.2}$$

$$U = \begin{bmatrix} u_{k|k} - u_{k-1} \\ u_{k+1|k} - u_{k-1} \\ \vdots \\ u_{k+N_c-1|k} - u_{k-1} \end{bmatrix}_{N_c \times 1} \tag{2.3}$$

$$H = \phi^T \phi + R_u \tag{2.4}$$

$$W = \phi^T (R_s - F x_k) \tag{2.5}$$

$$R_u = R_1 \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}_{N_c \times N_c} \tag{2.6}$$

$$\phi = \begin{bmatrix} CB & 0 & 0 & \cdots & 0 \\ CAB & CB & 0 & \cdots & 0 \\ CA^2B & CAB & CB & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ CA^{Np-1}B & CA^{Np-2}B & CA^{Np-3}B & \cdots & CA^{Np-Nc}B \end{bmatrix}_{N_p \times N_c} \tag{2.7}$$

$$F = \begin{bmatrix} CA \\ CA^2 \\ CA^3 \\ \vdots \\ CA^{Np}B \end{bmatrix}_{N_p \times 1} \tag{2.8}$$

$$R_s = r_k \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}_{N_p \times 1} \tag{2.9}$$

2.6 Pozostałe rodzaje regulatorów klasy MPC

- Nonlinear MPC
- Explicit MPC
- Robust MPC

2.7 Wady i zalety w porównaniu z regulatorem PID

Cecha	Regulator PID	Regulator MPC
ograniczenia wartości	brak	uwzględnione w projekcie: twarde albo miękkie
optymalność sterowania	brak	charakter optymalny
liczba wejść i wyjść	zazwyczaj SISO	MIMO
model matematyczny	nie jest konieczny	niezbędny (może być wyliczony iteracyjnie poprzez indetyfikację procesu)

Tabela 2.1: Porównanie regulacji PID i MPC

Rozdział 3. Założenia projektowe i wykorzystane narzędzia

3.1 Założenia projektowe

Jak pokazano w rozdziale 2 sterowanie predykcyjne jest bardzo złożonym procesem. Postanowiono więc przyjąć pewne obostrzenia ułatawiające skupienie się na samym sposobie regulacji, wpływie parametrów regulatora oraz obiektu na poprawność działania danego układu regulacji. W tym celu przyjęto następujące założenie projektowe - model matematyczny badanego układu jest znany, a także stały w czasie wykonywania się algorytmu. Ponadto obiekt ten jest dyskretnym układem liniowym. Założenie to pozwoliło na eliminację konieczności przeprowadzenia identyfikacji modelu. Do implementacji algorytmu MPC wykorzystano szybką metodę gradientową oraz ograniczono się do skalarnej wersji regulatora. Podczas oceny poprawności przeprowadzonej symulacji posłużono się następującymi kryteriami:

- Odpowiedź mikrokontrolera w czasie rzeczywistym.
- Poprawność wyznaczonych sterowań.
- Poprawność otrzymanej wartości wyjściowej.
- Uzyskanie wartości zadanej w skończonym czasie.

3.2 Architektura systemu

3.2.1 Procesor - architektura ARM

ARM (Advanced RISC Machine) jest to rodzina architektur procesorów typu RISC (Reduced Instruction Set Computing). Charakteryzuje się zmniejszoną liczbą instrukcji w porównaniu do CISC, co przekłada się na mniejsze zużycie energii. Wynika z tego zastosowanie architektury ARM w systemach wbudowanych. Dla każdego mikroprocesora jednostka sterująca (CU) jest sercem całego urządzenia. CU jest odpowiedzialne za operacje systemowe. Z tego powodu zaprojektowanie

jednostki sterującej jest najważniejszą częścią w tworzeniu nowego schematu mikrokontrolera. W przypadku architektury ARM CU jest zaimplementowany za pomocą maszyny stanów. Dodatkowo sygnały z jednostki sterującej są połączone z każdym podzespołem procesora, aby zapewnić kontrolę jego operacji. Jednostka arytmetyczno-logiczna (ALU) jest kolejną istotną częścią mikrokontrolera. Jej zadaniem jest prowadzenie obliczeń na liczbach całkowitych. W przypadku architektury ARM ALU posiada dwa 32 bitowe wejścia.

3.2.2 Platforma STM

STM32 jest to rodzina 32 bitowych mikrokontrolerów produkowana przez STMicroelectronics. Kontrolery są podzielone na odpowiednie serie, jednak łączy je bazowanie na 32 bitowym rdzeniu firmy ARM. Grupy te różnią się m.in. częstotliwością taktowania, obsługiwanymi urządzeniami peryferyjnymi, wsparciem dla arytmetyki zmiennoprzecinkowej, jak i możliwością cyfrowego przetwarzania sygnałów. Użyty w projekcie zestaw uruchomieniowy STM32 Nucleo F401RE jest wyposażony w mikrokontroler STM32F401, który zapewnia wsparcie dla wspomnianej wcześniej jednostki zmiennoprzecinkowej (FPU), jak również instrukcji cyfrowego przetwarzania sygnałów (DSP). Procesor ten jest oparty na architekturze ARM Cortex M4. Platforma Nucleo za to dostarcza elastyczne możliwości budowania oraz projektowania nowych rozwiązań sprzętowych, zarówno doświadczonym jak i początkującym, użytkownikom. Moduł ten łączy w sobie różne kombinacje aspektów wydajności oraz zużycia energii. W porównaniu do konkurencyjnych platform oferuje znacznie mniejsze zużycie energii podczas pracy z zasilaczem impulsowym. Poniżej zamieszczono pełną specyfikację.

- Mikrokontroler: STM32F401RET6
  - ⋆ Rdzeń: ARM Cortex M4 32-bit
  - ⋆ Częstotliwość taktowania: 84 MHz
  - ⋆ Pamięć programu Flash: 512 kB
  - ⋆ Pamięć SRAM: 96 kB
  - ⋆ Przetwornik analogowo-cyfrowy: 12-bitowy, 10-kanałowy
  - ⋆ Ilość Timerów: 10
  - ⋆ Ilość programowalnych wejść/wyjść: 81
  - ⋆ Interfejsy: 3x I2C, 3x USART, 4x SPI, USB 2.0 Full Speed
- Dwa typy złącz:

  - ⋆ Złącza dla nakładek kompatybilnych z Arduino Uno Rev3
  - ⋆ Standardowe piny STMicroelectronics Morpho, umożliwiające dostęp do wyprowadzeń mikrokontrolera
- Moduł zgodny z systemem mbed (mbed.org)
- Debugger ST-Link/V2 umieszczony na płytce z możliwością pracy jako oddzielne urządzenie z wyjściem SWD
- Możliwość zasilania poprzez złącze USB
- Wbudowane trzy diody LED:
  - ⋆ 1 x sygnalizująca napięcia zasilania
  - ⋆ 1 x sygnalizująca komunikację
  - ⋆ 1 x do dyspozycji użytkownika
- Dwa przyciski:
  - ⋆ 1 x RESET
  - ⋆ 1 x do dospozycji użytkownika
- Trzy różne interfejsy poprzez złącze miniUSB:
  - ⋆ Wirtualny port COM
  - ⋆ Pamięć masowa
  - ⋆ Port do programowania / debuggowania

3.3 Narzędzia programistyczne

3.3.1 Języki programowania C/C++

C jest to proceduralny, strukturalny, statycznie typowany język programowania, który znajduje zastosowanie w implementacji systemów operacyjnych i wbudowanych. C domyślnie zapewnia narzędzia, które w sposób efektywny są kompilowane do kodu maszynowego. Taki kod ma porównywalną wydajność do programów napisanych jedynie za pomocą asemblera. Jest to powiązane ze względną niskopoziomowością języka C, która zrealizowana została m.in. za pomocą ręcznej alokacji dynamicznej pamięci, a także minimalnego wspracia w czasie wykonywania programu. Pomimo swoich niewątpliwych zalet istnieją także wady zastosowania tylko

i wyłącznie tego języka. Przykładem niewłaściwych a dopuszczalnych praktyk programistycznych w języku C są możliwość wywołania niezadeklarowanej funkcji, czy też przyjmowanie przez funkcje zadeklarowane bez żadnego argumentu dowolnej ilości parametrów. Ponadto legalną operacją jest inicjalizacja tablicy literałem łańcuchowym składającym się z większej ilości elementów niż jest przeznaczonej pamięci dla tej zmiennej.

C++ jest to wieloparadygmatowy, statycznie typowany, kompilowalny język programowania zapewniający znacznie wyższy poziom abstrakcji w porównaniu do języka C. Na potrzeby pracy inżynierskiej wykorzystano najnowszy, w pełni dostępny, standard tego języka - C++17. Wspomniana abstrakcja danych została zrealizowana za pomocą zastosowania paradygmatu programowania obiektowego. Zastosowanie interfejsów i 'schowanie' implementacji programu umożliwia lepsze przeprowadzenie testów jednostkowych, a także budowę wieloplatformowych aplikacji. Podejście obiektowe umożliwia ponadto zaprowadzenie znacznie większego porządku w projektowaniu danego rozwiązania. Jednakże największym atutem takiego rozwiązania jest hermetyzacja danych, która zapobiega wprowadzeniu innych wartości do zmiennych przechowywanych w danej klasie w niepożądanym miejscu programu. W przypadku programowania rozwiązań przeznaczonych pod współpracę z systemami wbudowanymi użycie najnowszych możliwości nowszych standardów języka C++ nie jest czasem możliwe. Przykładem jest w tym przypadku obsługa wyjątków, która pochłania znaczne ilości mocy obliczeniowej, jak i potrzebnej pamięci. Zagrożeniem jest także używanie Standard Template Library (STL), biblioteki zawierającej standardowe algorytmy, kontenery i iteratory. Implementacja tejże biblioteki obfita jest w operacje na stercie, któtych użycie powinno być minimalizowane w świecie systemów wbudowanych ze względu na fragmentację niewielkiej ilości dostępnej pamięci oraz dłuższy czas jej alokacji i dealokacji. Ponadto STL zawiera operacje rzucające wcześniej wymienione wyjątki.

3.3.2 Język programowania Python

Python jest to interpretowany, interaktywny, zoorientowany obiketowo język programowania. Zawiera on moduły, wyjątki, dynamiczne typowanie, wysokopoziomowe typy danych i klasy. Python łączy w sobie godną uwagi moc z bardzo czystą składnią. Posiada on interfejs do bardzo dużej ilości biblioteki i wywołań systemowych. Dodatkowo jest możliwe jego rozszerzenie o własne biblioteki w C albo C++. Python jest także wykorzystywany jako dodatkowy język przy projektowaniu aplikacji, które potrzebują programowalnego interfejsu. Warta uwagi jest również przeno-

śność tego języka pomiędzy popularnymi systemami operacyjnymi takimi jak Linux, Mac, czy też Windows.

W projekcie inżynierkism skorzystano z kilku wysokopoziomowych bibliotek dostępnych w Pythonie. Do komunikacji z platformą STM została wykorzystana biblioteka *Serial*, która zawiera odpowiednią implementację wysyłania i odbierania danych za pośrednictwem portu szeregowego. Przydatna okazała się także biblioteka *NumPy* oferująca klasy macierzy, a także wydajne obliczenia numeryczne. Zastosowanie znalazła także biblioteka *Pyplot*, która jest zaopatrzona w funkcje pozwalające graficznie zaprezentować wyniki działania programu. Działanie dwóch ostatnich bibliotek jest bardzo zbliżone do analogicznych funkcji środowiska MATLAB.

3.3.3 Środowisko MATLAB

MATLAB (matrix laboratory) jest to wieloparadygmatowe i zamknięte środowisko służące do wykonywania obliczeń numerycznych rozwijane przez firmę Mathworks. MATLAB pozwala na bardzo wygodne dla użytkownika operacje na macierzach, graficzną reprezentację danych w formie wykresów, implementację dużej liczby algorytmów, a także możliwość stworzenia interfejsu użytkownika. Jednym z pakietów tego środowiska jest *Simulink* - graficzne środowisko programistyczne przeznaczone do modelowania, analizy oraz symulacji systemów dynamicznych. Rozszerzenie to oferuje integrację z resztą modułów MATLABa, co znacznie upraszcza proces tworzenia modelu. Podstawą funkcjonalności pakietu *Simulink* są programowalne bloki, które zawierają odpowiednie wywołania funkcji numerycznych. Biblioteka ta znalazła szerokie zastosowanie w automatyce i cyfrowym przetwarzaniu sygnałów.

3.3.4 STM32CubeMX

STM32CubeMX jest to graficzne narzędzie umożliwiające prostą konfigurację mikrokontrolerów oraz mikroprocesorów z rodziny STM32. Program ten jest w stanie wygenerować adekwatny system plików zawierających kod w C przeznaczony do użycia dla rodziny rdzeni Arm Cortex-M. Biblioteka, z której wywołań korzysta wygenerowany kod ma formę warstwy abstrakcji sprzętowej (HAL). Takie rozwiązanie umożliwia pominięcie operacji na rejestrach procesora w trakcie tworzenia aplikacji przez użytkownika.

3.3.5 CMake

CMake jest to otwartoźródłowa, wieloplatformowa rodzina narzędzi przeznaczonych do budowania oraz testowania oprogramowania. CMake jest używany do kontrolowania kompilacji kodów źródłowych, używając prostych, niezależnych od platformy i kompilatora plików konfiguracyjnych. Program ten generuje natywne pliki Makefile, a także przestrzenie robocze, które mogą być użyte w dowolnym zintegrowanym środowisku programistycznym.

3.3.6 Kompilator i linker

Do kompilacji i linkowania wykorzystano zestaw narzędzi programistycznych GNU przeznaczony dla systemów wbudowanych opartych o architekturę Arm (The GNU Embedded Toolchain Arm), w którego skład wchodzą m.in. kompilator języka C - arm-none-eabi-gcc, kompilator języka C++ - arm-none-eabi-g++, a także linker - ld. Zdecydowano się na ten zestaw narzędzi ze względu na otwartoźródłowość tego projektu.

3.3.7 Regex

Wyrażenia regularne (regex) są to sekwencje znaków, które definiują wzór wyszukiwania. Regex został rozwinięty jako technika zarówno w teoretycznej informatyce, jak i w teorii języków formalnych. Z reguły wyrażenia regularne są wykorzystywane w operacjach ’znajdź’ lub ’znajdź i zamień’, które zdefiniowane są na łańcuchach znakowych. Innym zastosowaniem ich jest weryfikacja poprawności podanych danych na wejściu. Istnieją dwie wiodące składnie do zapisu wyrażeń regularnych: pierwsza z nich wchodzi w skład standardu POSIX, a druga jest wzorowana na oryginalnej implementacji języka Perl.

3.4 Przykład referencyjny

Do oceny poprawności implementacji rozwiązania wykorzystano 2 przykłady referencyjne, które zostały przygotowane w 3.3.3 środowisku MATLAB. Jeden z nich posłużył jako zapoznanie się ze sposobem regulacji MPC. Został on wykonany w środowisku *Simulink*. Do weryfikacji poprawności obliczeń przeprowadzonych na platformie STM zaimplementowano analogiczny alogrytm jako skrypt środowiska MATLAB.

3.5 Sposób testowania

Testy zostały przeprowadzone jako symulacja Hardware in the loop (HIL). Jest to technika używana do rozwoju i testowania złożonych wbudowanych systemów czasu rzeczywistego. Symulacja HIL zapewnia efektywne możliwości przeprowadzenia testów danej platformy poprzez przekazanie kontroli nad modelem matematycznym tejże platformie. Właściwości sterowanego modelu są włączane do próby testowej poprzez dodanie adekwatnego modelu matematycznego reprezentującego opisywany układ dynamiczny. Implementacja symulacji HIL została wykonana w języku programowania Python ze względu na łatwą zdolność integracji różnych bibliotek, co pokazno w rozdziale 3.3.2.

Rozdział 4. Implementacja rozwiązania

4.1 Ogólny schemat programu

Przydałby się rysunek z przepływem informacji + jak działa STM w połączeniu z PC.

4.2 Szczegóły implementacji - STM

Zdecydowano się na implementację części programowej w języku C++, zważywszy na jego benefity omówione w rozdziale 3.3.1. W kolejnych podrozdziałach zaprezentowano szczegóły zaprogramowanego rozwiązania problemu. Omówiono także na przykładach zastosowane praktyki programistyczne wraz z możliwością ich rozwoju oraz usprawnienia działania.

4.2.1 Charakterystyczne cechy C++

W projekcie wykorzystano paradygmat programowania obiektowego ze względu na uproszczenie zarządzania dostępem do danych przez konkretne funkcje, a także w celu uporządkowania konkretnych rozwiązań. Dodatkowo takie podejście w łatwy sposób można rozszerzyć o nowe funkcjonalności. Nieodłącznym aspektem programistycznym są testy jednostkowe, ponieważ pozwalają one na wczesne wykrycie błędów danego oprogramowania. Implementacja obiektowości znacznie ułatwia testowanie kodu napisanego w C++. W tym celu można posłużyć się np. otwartoźródłową biblioteką Googletest. Na listingu 1 zaprezentowano przykładowy plik nagłówkowy jednej z klas użytych w implementacji rozwiązania. Podziałem na pola i metody publiczne oraz prywatne zrealizowano postulat hermetyzacji danych. Do pola *storage* spoza klasy można odwołać się tylko poprzez adekwatną metodę *Get*. Dzięki takiemu rozwiązaniu uniknięto niepożądanych zmian zawartości tego pola poprzez metody nienależące do klasy *DataParser*. Dodatkową organizację i porządek w zrealizowanym projekcie zapewniło użycie przestrzeni nazw (namespace).

Na listingu 2 przedstawiono plik źródłowy korespondujący z plikiem nagłówkowym klasy *DataParser*. Rozdzielenie deklaracji i definicji pól oraz metod umożli-

```cpp
#ifndef MPC_STM_DATAPARSER_HPP
#define MPC_STM_DATAPARSER_HPP

#include <map>
#include <vector>
#include <string>

namespace Utils {
    class DataParser {
    public:
        static bool isNewDataGoingToBeSend;

        DataParser() = default;
        ~DataParser() = default;
        void ParseReceivedMsg(const std::string& msg);

        [[nodiscard]] std::map<std::string,
        std::vector<double>> GetStorage() const;

        void ClearStorage();

    private:
        constexpr static const char* regexPattern =
        R"(([[:alpha:]]+)(': )(\[.+?\]))";

        std::map<std::string, std::vector<double>> storage;
    };
}

#endif //MPC_STM_DATAPARSER_HPP
```

Listing 1: DataParser.hpp: Przykładowy plik nagłówkowy

wia odwołanie się do tych publicznych poza zakresem danej klasy poprzez włączenie odpowiedniego pliku nagłówkowego. W celu zapewnienia bardziej elastycznego sposobu identyfikacji odebranych danych posłużono się omówionymi w rozdziale 3.3.7 wyrażeniami regularnymi. Zastosowanie wyszukiwania kluczowych wartości w przeprowadzaniu analizy odebranej wiadomości umożliwia przyszłą modyfikację tychże komunikatów. Aby uprościć implementację biblioteki *<regex>*, zdecydowano się na wykorzystanie STLa. Stało się to możliwe dzięki pokaźnemu, jak na systemy wbudowane, rozmiarowi pamięci flash platformy Nucleo. Całość została zrealizowana jako maszyna stanów.

W celu lepszego dostosowania aplikacji do standardu języka C++ postanowiono zastosować jeden z wzorców projektowych, a mianowicie Singleton. Przy-

```cpp
#include ...

bool Utils::DataParser::isNewDataGoingToBeSend = false;

std::map<std::string, std::vector<double>>
 ↪  Utils::DataParser::GetStorage() const {
    return storage;
}

void Utils::DataParser::ParseReceivedMsg(const std::string& msg) {
    std::string valuesMatch;
    std::regex pattern(regexPattern);
    std::smatch fullMatch;
    std::string::const_iterator iterator(msg.cbegin());

    while (std::regex_search(iterator, msg.cend(), fullMatch, pattern))
     ↪  {
        valuesMatch = fullMatch[3].str();
        std::replace(valuesMatch.begin(), valuesMatch.end(), ',', ' ');
        valuesMatch.pop_back(); // trim ]
        valuesMatch.erase(0, 1); // trim [
        if (fullMatch[1].str().find('A') != std::string::npos) {
            Utils::Misc::StringToDouble(valuesMatch, storage["A"]);
        } else if (fullMatch[1].str().find('B') != std::string::npos) {
            Utils::Misc::StringToDouble(valuesMatch, storage["B"]);
        } else if (fullMatch[1].str().find('C') != std::string::npos) {
            Utils::Misc::StringToDouble(valuesMatch, storage["C"]);
        } else if (fullMatch[1].str().find("set") != std::string::npos)
         ↪  {
            Utils::Misc::StringToDouble(valuesMatch, storage["set"]);
        } else if (fullMatch[1].str().find("control") !=
         ↪  std::string::npos) {
            Utils::Misc::StringToDouble(valuesMatch,
             ↪  storage["control"]);
        } else if (fullMatch[1].str().find("horizon") !=
         ↪  std::string::npos) {
            Utils::Misc::StringToDouble(valuesMatch,
             ↪  storage["horizons"]);
        }
        iterator = fullMatch.suffix().first;
    }
}

void Utils::DataParser::ClearStorage() {
    storage.clear();
}
```

Listing 2: DataParser.cpp: Przykładowy plik źródłowy

kład implementacji pokazano na listingach 3 oraz 4. Zdecydowano się na statyczną wersję tego wzorca ze względu na bezpieczeństwo w poprawnym funkcjonowaniu procesów. Rozwiązanie proponowane za pośrednictwem gotowego przykładu użycia interferjsu biblioteki HAL jest bardzo niebezpieczne z punktu widzenia dostępu do danych. Mianowicie, po inicjalizacji urządzeń peryferyjnych korzysta się w owym przykładzie z jednej struktury, która jest zmienną globalną. Przykład ten pokazuje, że rozwiązania tej konkretnej biblioteki nie należą do optymalnych. Z tego powodu postanowiono stworzyć osobną klasę, która powinna być inicjalizowana tylko raz. W takim rozwiązaniu problemu Singleton jest dobrym pomysłem na poprawę jakości kodu.

```
namespace HAL {
    class Peripherals {
    public:
        Peripherals(const Peripherals&) = delete;
        Peripherals& operator=(const Peripherals&) = delete;
        Peripherals(Peripherals&&) = delete;
        Peripherals& operator=(Peripherals&&) = delete;

        static Peripherals& GetInstance();
    };
}
```

Listing 3: Peripherals.hpp: Wzorzec projektowy - singleton

```
HAL::Peripherals& HAL::Peripherals::GetInstance() {
    static Peripherals instance;
    return instance;
}
```

Listing 4: Peripherals.cpp: Wzorzec projektowy - singleton

4.2.2 Komunikacja

W implementacji komunikacji po stronie platformy STM wykorzystano uniwersalny asynchroniczny nadajnik-odbiornik (UART). Jest to układ scalony, który wykorzystuje się w celu odbierania i przesyłania informacji poprzez port szeregowy. W tym celu wykorzystano gotowe, wysokopoziomowe rozwiązania biblioteki HAL. Aby możliwe było wysłanie nowych parametrów regulatora oraz układu wykorzystano wspomniany w rozdziale 3.2.2 programowalny przycisk dostępny dla użytkownika. Postanowiono zastosować do tego celu jedno z systemowych przerwań wraz

z wywołaniem zwrotnym widocznym na listingu 5. Taki sposób rozwiązania problemu pozwolił na wyeliminowanie zjawiska odpytywania (pollingu). W oczekiwaniu na interakcję użytkownika, procesor systematycznie nie sprawdza danego rejestru.

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
    if(GPIO_Pin == B1_Pin) {
        HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
        Utils::DataParser::isNewDataGoingToBeSend = true;
    }
}
```

Listing 5: Peripherals.cpp: Implementacja wywołania zwrotnego (callback) podczas przerwania

W przypadku implementacji funkcji stricte odpowiedzialnych za komunikację posłużono się wcześniej wspomnianym pollingiem. Skorzystanie z obsługi przerwań w tym przypadku nie było możliwe zważywszy na niewłaściwą interakcję między tym generowanym po wciśnięciu przecisku a tymi po wysłaniu czy też odebraniu wiadomości. Takie podejście prowadzi do nadmiernego zużycia zasobów procesora, jednakże umożliwiło ono tymczasowe rozwiązanie napotkanego problemu. Zgodnie z przyjętym paradygmatem programowania obiektowego napisano metody opakowujące biblioteczne funkcje obsługujące UARTa. Wspomniane funkcjonalności zaprezentowano na listingach 6 oraz 7. Implementacja wysyłania wiadomości okazała się prostolinijna w przeciwieństwie do odbioru wiadomości. Problemu przysporzył brak możliwości skutecznego nawiązania połączenia bez znajomości długości przychodzącego łańcucha znaków. Postanowiono więc wysyłać dwie wiadomości - pierwszą o z góry ustalonym rozmiarze 1 bajta zawierającą informację o długości następnej. Takie ograniczenie poskutkowało zmniejszeniem maksymalnej pojemności wiadomości do 255 bajtów, co jest wystarczającą wielkością do użycia w celach projektowych. Istnieje możliwość zwiększenia tego limitu poprzez ustalenie innej liczby bajtów w pierwszej przychodzącej wiadomości.

```
    void HAL::Peripherals::SendString(const uint8_t* string, uint16_t
    ↪ timeout) {
        HAL_UART_Transmit(&handleUART, const_cast<uint8_t*>(string),
        std::strlen(reinterpret_cast<const char*>(string)), timeout);
    }
```

Listing 6: Peripherals.cpp: Implementacja wysyłania danych z platformy STM do PC

```
1 void HAL::Peripherals::ReceiveString(const uint8_t* string, uint16_t
  ↪  timeout) {
2     uint8_t bufSize = 0;
3     HAL_UART_Receive(&handleUART, &bufSize, 1, timeout);
4     HAL_UART_Receive(&handleUART, const_cast<uint8_t*>(string),
      ↪  bufSize, 100);
5 }
```

Listing 7: Peripherals.cpp: Implementacja odbierania danych przez platformę STM

4.2.3 Klasa macierzy

Implementacja algorytmu MPC wymaga dużej ilości operacji na macierzach. W standardowej bibliotece języka programowania C++ nie są dostępne żadne kontenery ułatawiające przeprowadzenie takich działań. Dostępne są wprawdzie gotowe rozwiązania w postaci zewnętrznych bibliotek, np. *Eigen*. Jednakże nie zdecydowano się na włączenie żadnej z nich do projektu zważywszy na ograniczone zasoby platformy STM. Postanowiono więc zaimplementować własne rozwiązanie będące klasą opakowującą na dynamicznie alokowane tablice wskaźników. Na listingu 8 przedstawiono fragment pliku nagłówkowego omawianej solucji. Przedstawiono w nim najważniejsze metody będące niezbędne do poprawnego zaimplementowania algorytmu sterowania predykcyjnego. Skorzystano także z możliwości przeciążenia operatorów, którą oferuje język C++. W ten sposób opakowano odpowiadające im metody i sprawiono, że używanie tych funkcjonalności jest bardziej przejrzyste dla użytkownika. Zaimplementowano także rekursywną metodę obliczania wyznacznika macierzy bazującą na rozwinięciu Laplace'a. Potrzebny okazał się także być algorytm odwracania macierzy. W tym celu posłużono się metodą polegającą na znalezieniu macierzy dopełnień algebraicznych. Pewną niedogodnością zaproponowanego rozwiązania jest użycie wyjątków, których obsługa jest domyślnie wyłączona w platformie STM. Ich negatywny wpływ na systemy wbudowane został omówiony w rozdziale 3.3.1. Znalazły się one w projekcie, ponieważ ich zastosowanie znacznie ułatwiło debuggowanie testowej wersji algorytmu, która była sprawdzana na PC.

4.2.4 Aglorytm

W celu realizacji sterowania predykcyjnego posłużono się szybką metodą gradientu prostego. Do poprawnego funkcjonowania tej metody potrzebne są następujące dane.

- Wektor stanów x_k.

```
#include ...

class CMatrix {
protected:
    uint32_t rows;
    uint32_t columns;
    double** matrix{};
    void make_matrix();

public:
    CMatrix();
    CMatrix(uint32_t rows, uint32_t columns);
    CMatrix(uint32_t rows, uint32_t columns, double** mat);
    CMatrix(uint32_t rows, uint32_t columns, const double* mat);
    explicit CMatrix(uint32_t rows, const std::string& value = "");
    CMatrix(const CMatrix& M);
    ~CMatrix();

    [[nodiscard]] uint32_t GetRows() const { return rows; }
    [[nodiscard]] uint32_t GetColumns() const { return columns; }
    void GetSize(uint32_t& rows, uint32_t& columns) const;
    [[nodiscard]] double& GetElement(uint32_t row, uint32_t column)
    ↪  const;
    void SetRow(uint32_t row, const CMatrix& v);
    void SetColumn(uint32_t column, const CMatrix& v);
    void SetValue(double value);
    [[nodiscard]] double Det() const;
    [[nodiscard]] CMatrix Inverse() const;

    double* operator[](uint32_t row) const;
    CMatrix& operator= (const CMatrix& m);
    CMatrix operator+ (const CMatrix& m) const;
    CMatrix& operator+= (const CMatrix& m);
    CMatrix operator- (const CMatrix& m) const;
    CMatrix& operator-= (const CMatrix& m);
    CMatrix operator* (const CMatrix& m) const;
    CMatrix& operator*= (const CMatrix& m);
    CMatrix operator* (const double& scalar) const;
    CMatrix& operator*= (const double& scalar);
    CMatrix operator/ (const double& scalar) const;
    CMatrix& operator/= (const double& scalar);
    CMatrix operator^ (const uint32_t& exponent) const;
    CMatrix operator-() const;
    [[nodiscard]] CMatrix T() const;
};
```

Listing 8: Matrix.hpp: Fragment pliku nagłówkowego zawierającego własną implementację macierzy

- Początkowa wartość v_0 (przyjęto 0).
- Maksymalna liczba możliwych iteracji i_{max}.
- Maksymalna wartość własna L macierzy H, jak i minimalna μ.

Poniżej zamieszczono kolejne kroki tego algorytmu.

1. Ustawienie stanu poczatkowego: $v_{old} = y$, $w = y$.
2. Obliczenie v jako projekcji gradientowej z w o kroku $1/L$.
3. Obliczenie $w = v + \frac{\sqrt{L}-\sqrt{\mu}}{\sqrt{L}+\sqrt{\mu}}(v - v_{old})$.
4. Ustawienie $v_{old} = v$.
5. Powtórzenie algorytmu do osiągnięcia warunku stopu albo maksymalnej liczby iteracji.
6. Zwrócenie otrzymanego sterowania v.

Na listingu 9 znajduje się implementacja szybkiej metody gradientowej, a listing 10 zawiera sposób, w jaki zrealizowano obliczenie kolejnego kroku w algorytmie.

```
double Control::MPC::FastGradientMethod(const std::string& msg) {
    CMatrix J(1,1), J_prev(1,1);
    Utils::Misc::StringToDouble(msg, sys.x);
    for (uint32_t i = 0; i < 100; i++) {
        opt.W = opt.fi.T() * ((opt.F * sys.x) - opt.Rs);
        CalculateProjectedGradientStep();
        J_prev = J;
        J = sys.v.T() * opt.H * sys.v / 2 + sys.v.T() * opt.W;
        if (std::fabs(J_prev[0][0] - J[0][0]) < eps) {
            break;
        }
    }
    return sys.v[0][0];
}
```

Listing 9: MPC.cpp: Implementacja szybkiej metody gradientowej

W celu obliczenia minimalnej i maksymalnej wartości własnej macierzy H zastosowano iteracyjną metodę potęgową. Jej implementacja znajduje się na listingu 11. Algorytm ten umożliwia znalezienie maksymalnej, co do modułu, wartości własnej danej macierzy. Wartość minimalna może być obliczona poprzez znalezienie

```
void Control::MPC::CalculateProjectedGradientStep() {
    CVector gradient(sys.v.GetRows(), 1), v_old = sys.v;
    gradient = opt.H * sys.v + opt.W;
    sys.w = sys.v - gradient * eigenvalues.step;
    for (uint32_t i = 0; i < sys.v.GetRows(); i++) {
        if (sys.w[i][0] < controlValues.min) {
            sys.v[i][0] = controlValues.min;
        }
        else if (sys.w[i][0] > controlValues.max) {
            sys.v[i][0] = controlValues.max;
        } else {
            sys.v[i][0] = sys.w[i][0];
        }
    }
    sys.w = sys.v + (sys.v - v_old) * eigenvalues.fastConvergence;
}
```

Listing 10: MPC.cpp: Implementacja obliczenia kolejnego kroku w algorytmie gradientowym

wartości maksymalnej macierzy odwrotnej. Metoda potęgowa została zastosowana ze względu na swoją prostotę, łatwość w implementacji oraz konieczność wyznaczenia jedynie najbardziej skrajnych, co do modułu, wartości własnych danej macierzy.

W celu przetestowania zmian parametrów regulatora predykcyjnego w czasie wykonywania się programu konieczne okazało się zaimplementowanie nieobecnej w języku C++ funkcji realokacji pamięci dynamicznej. Posłużono się w tym celu tzw. obiektami funkcyjnymi. Na listingu 12 pokazano sposób, w jaki reinicjalizowane są parametry układu, jak i regulatora.

W celu wyznaczenia wskazanych w rozdziale 2.5 macierzy potrzebnych do rozwiązania problemu programowania kwadratowego zastosowano kod zaprezentowany na listingu 13.

4.3 Szczegóły implementacji - PC

Przygotowany do testowania algorytmu MPC skrypt w języku programowania Python składa się z trzech funkcjonalnych części. Pierwsza z nich została przedstawiona na listingu 14. Sekcja ta odpowiada za inicicjalizację parametrów, które zostaną wysłane do platformy STM oraz użyte na potrzeby wyliczenia kolejnych wartości wyjściowych układu. Zmienne przechowywane są w kolekcji standardowej biblioteki

```
double Utils::Misc::PowerMethod(const CMatrix& matrix, uint32_t
↪  maxIterations) {
    double greatest_eigenvalue_current = 0,
    ↪  greatest_eigenvalue_previous;
    std::random_device rd;
    std::mt19937 gen(rd());
    CVector b_k(matrix.GetRows(), 1), b_k1(matrix.GetRows(), 1);

    for (uint32_t i = 0; i < matrix.GetRows(); i++) {
        b_k[i][0] = std::generate_canonical<double, 10>(gen);
    }
    for (uint32_t i = 0; i < maxIterations; i++) {
        b_k1 = matrix * b_k;
        b_k = b_k1 / std::sqrt((b_k1 * b_k1.T())[0][0]);
        greatest_eigenvalue_previous = greatest_eigenvalue_current;
        greatest_eigenvalue_current = ((matrix * b_k) * b_k.T())[0][0]
        ↪  / (b_k * b_k.T())[0][0];
        if (std::fabs(greatest_eigenvalue_current -
        ↪  greatest_eigenvalue_previous) < 0.01) {
            break;
        }
    }
    return std::fabs(greatest_eigenvalue_current);
}
```

Listing 11: Misc.cpp: Implementacja metody potęgowej

Pythona - słowniku. Zabieg taki umożliwił odwołanie się do konkretnych parametrów za pomocą klucza stanowiącego nazwę danej wartości.

Kolejna sekcja odpowiedzialna jest za główną część skryptu, a mianowicie komunikację z mikrokontrolerem oraz obliczanie kolejnych wartości wyjściowych układu. Przesyłanie wiadomości zostało zrealizowane za pomocą interfejsu biblioteki *Serial*. Na listingu 15 zaprezentowano proponowaną implementację.

Na listingu 16 przedstawiono część skryptu odpowiedzialną za zapisywanie danych.

4.4 Problemy napotkane podczas realizacji

Podczas realizacji implementacji algorytmu MPC napotkano na kilka trudności, które wynikały głównie z ograniczeń biblioteki HAL. Problemy te opisano w poprzednich rozdziałach. W celu sprawnego testowania algorytmu podjęto decyzję, aby obliczenia na macierzach przeprowadzać na stercie. Rozwiązanie to znacznie ułatwiło pracę nad projektem, jednak obarczone było sporą utratą wydajności.

```
1  void Control::MPC::InitializeParameters(std::map<std::string,
   ↪  std::vector<double>> storage) {
2      uint32_t dimension = static_cast<uint32_t>(storage["C"].size());
3      controlValues.min = storage["control"][0];
4      controlValues.max = storage["control"][1];
5
6      horizons.prediction = storage["horizons"][0];
7      horizons.control = storage["horizons"][1];
8
9      sys.A(dimension, dimension, storage["A"].data());
10     sys.B(dimension, 1, storage["B"].data());
11     sys.C(1, dimension, storage["C"].data());
12     sys.x(dimension, 1);
13     sys.v(horizons.control, 1);
14     sys.w(horizons.control, 1);
15
16     opt.F(horizons.prediction, dimension);
17     opt.fi(horizons.prediction, horizons.control);
18     opt.Rw(horizons.control, horizons.control, "eye");
19     opt.Rs(horizons.prediction, 1);
20     opt.H(horizons.control, horizons.control);
21     opt.W(horizons.control, 1);
22     opt.scalarRs = storage["set"][0];
23     opt.scalarRw = 1;
24
25     CalculateOptimizationMatrices();
26
27     eigenvalues.max = Utils::Misc::PowerMethod(opt.H, 20);
28     eigenvalues.min = Utils::Misc::PowerMethod(opt.H.Inverse(), 20);
29     eigenvalues.step = 1 / eigenvalues.max;
30     eigenvalues.fastConvergence = (std::sqrt(eigenvalues.max) -
       ↪  std::sqrt(eigenvalues.min)) /
31                                   (std::sqrt(eigenvalues.max) +
                                      ↪  std::sqrt(eigenvalues.min));
32 }
```

Listing 12: MPC.cpp: Implementacja zadawania nowych parametrów

```
void Control::MPC::CalculateOptimizationMatrices() {
    opt.Rw *= opt.scalarRw;
    opt.Rs.SetValue(opt.scalarRs);
    CVector productMatrix(sys.C.GetColumns());
    productMatrix = sys.C * (sys.A ^ (horizons.prediction -
    ↪ horizons.control));
    for (uint32_t i = horizons.prediction; i != 0; i--) {
        for (uint32_t j = horizons.control; j != 0; j--) {
            if (i == horizons.prediction) {
                if (j < horizons.control) {
                    productMatrix *= sys.A;
                }
                opt.fi[i-1][j-1] = (productMatrix * sys.B)[0][0];
            } else if (i < j && j < horizons.control) {
                opt.fi[i-1][j-1] = opt.fi[i][j];
            }
        }
    }
    opt.H = opt.fi.T() * opt.fi + opt.Rw;
    productMatrix = sys.C * sys.A;
    for (uint32_t i = 0; i < horizons.prediction; i++) {
        opt.F.SetRow(i, productMatrix);
        productMatrix *= sys.A;
    }
}
```

Listing 13: MPC.cpp: Implementacja wyliczenia nowych macierzy optymalizacyjnych

```
projectPath = os.path.dirname(os.path.abspath(__file__))
changeParameters = True
systemParameters = {'A': [1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0,
 ↪  1],
                    'B': [0, 0, 1, 0],
                    'C': [1, 1, 0, 0],
                    'setPoint': [10],
                    'controlExtremeValues': [-10, 10],
                    'horizons': [30, 8]}
systemParametersToSend = str(systemParameters)[1:-1] + '\n\0'
A = np.array(systemParameters['A']).\
    reshape(-1, int(np.sqrt(len(systemParameters['A']))))
B = np.array(systemParameters['B']).\
    reshape(len(systemParameters['B']), -1)
C = np.array(systemParameters['C'])
x = np.zeros((len(systemParameters['B']), 1))
y = []
u = []
timer = []
```

Listing 14: Inicjalizacja danych w skrypcie Pythona

```
with serial.Serial('COM3', 115200, timeout=20) as ser:
    if changeParameters:
        ser.write(bytes([len(systemParametersToSend)]))
        ser.write(systemParametersToSend.encode())
        startTimer = time.time()
        print(ser.readline().decode('utf-8'))
        endTimer = time.time()
        initTimer = endTimer - startTimer
    for i in range(200):
        xToSend = np.array2string(
            x.flatten(), formatter={'float_kind': lambda number: "%.4f"
             ↪  % number})
        xToSend = xToSend[1:-1] + '\0'
        ser.write(bytes([len(xToSend)]))
        ser.write(xToSend.encode())
        startTimer = time.time()
        v = float(ser.readline().decode('utf-8'))
        endTimer = time.time()
        x = np.dot(A, x) + v * B
        y.append(np.dot(C, x).item(0))
        timer.append(endTimer - startTimer)
        u.append(v)
```

Listing 15: Główna pętla przeprowadzająca test HIL

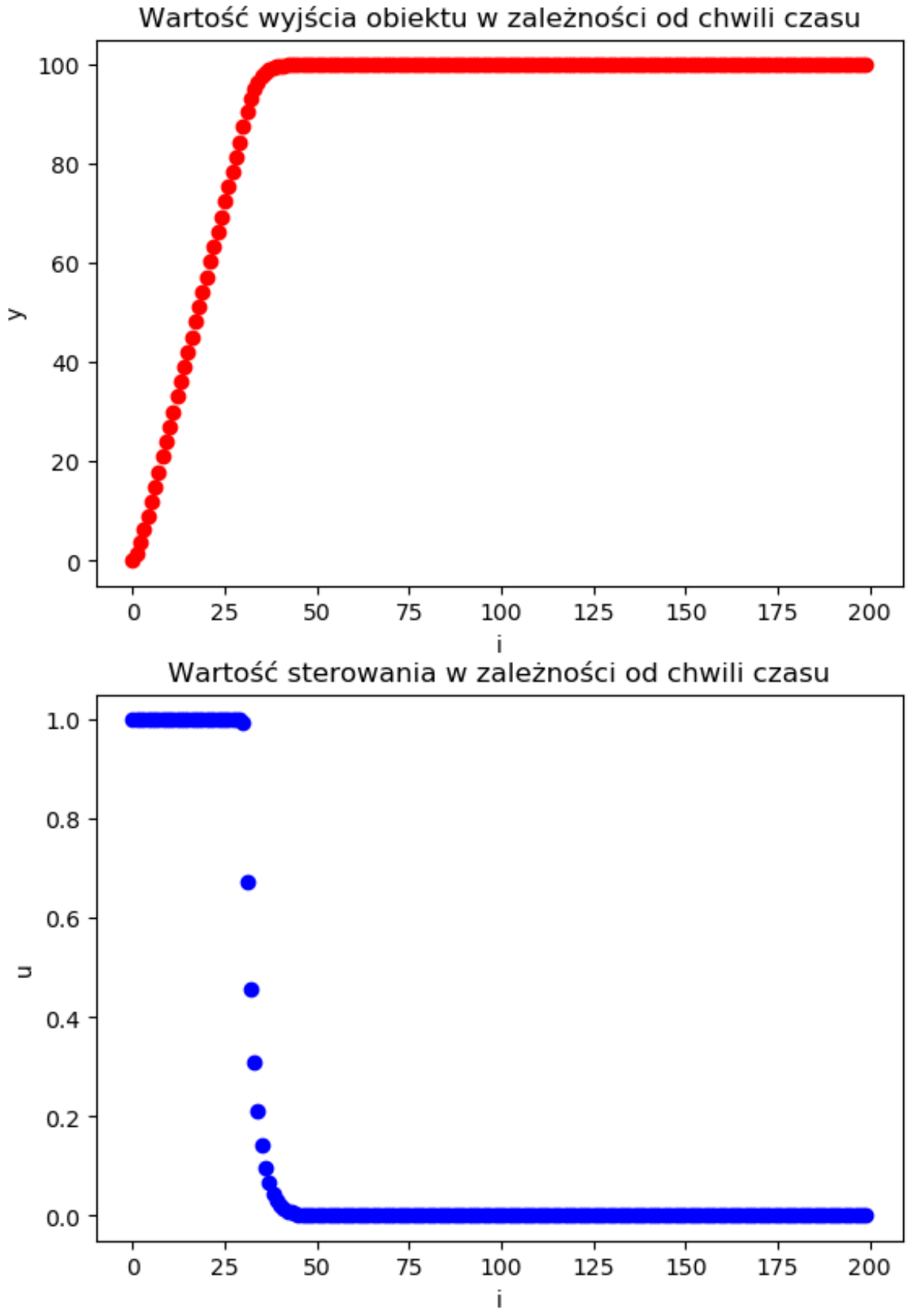
```
_, fig = plt.subplots(2, 1, figsize=(6.4, 9.6))
fig[0].set_title('Wartość wyjścia obiektu w zależności od chwili
↪  czasu')
fig[0].plot(y, 'ro'), fig[0].set_ylabel('y'), fig[0].set_xlabel('i')
fig[1].set_title('Wartość sterowania w zależności od chwili czasu')
fig[1].plot(u, 'bo'), fig[1].set_ylabel('u'), fig[1].set_xlabel('i')
plt.savefig(r'{0}\plots\horizons_{1}.png'.format
            (projectPath, systemParameters['horizons']), format='png',
            ↪  bbox_inches='tight')

with open('log.txt', 'a') as logger:
    logger.write(str(systemParameters['A']))
    logger.write('Init time, Mean of timestamps,'
                 ' Stdev of timestamps, Max of timestamps\n')
    logger.write(f"{initTimer:.4} {statistics.mean(timer):.4}"
                 f"{statistics.stdev(timer):.4} {max(timer):.4}\n")
```

Listing 16: Zapisywanie danych i wykresów

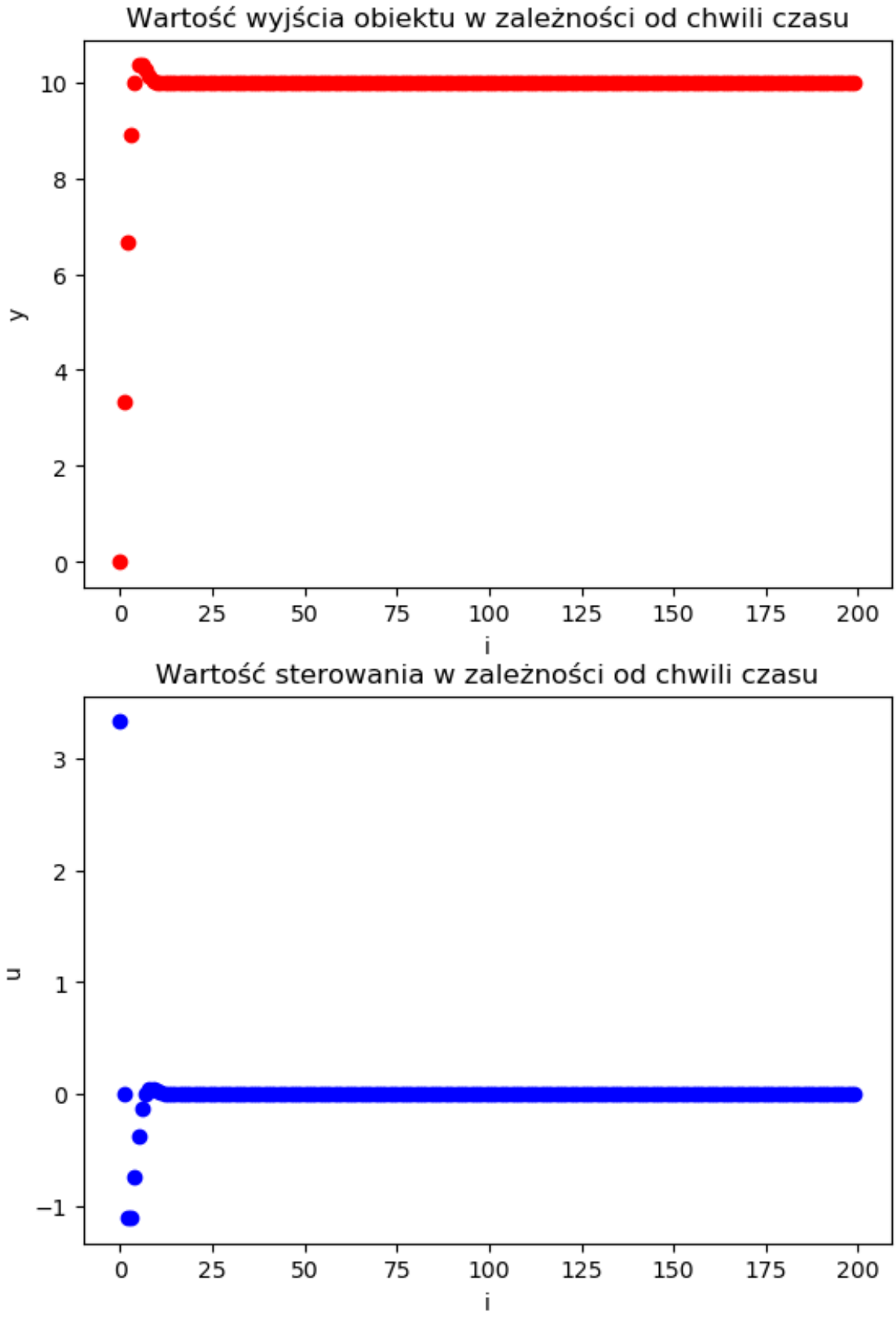
Rozdział 5. Przykładowe wyniki

5.1 Różne parametry układu i wartości zadane

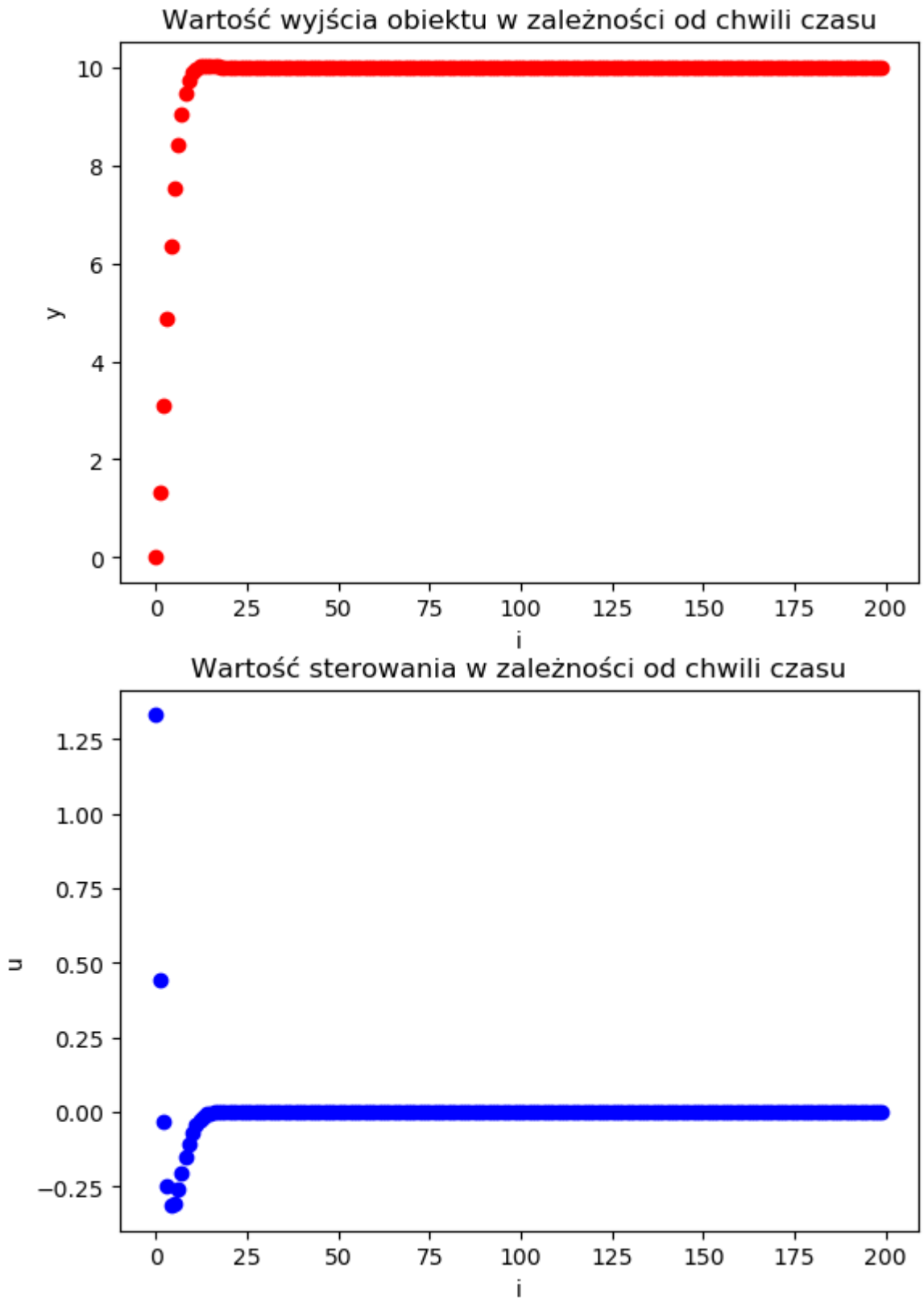
5.2 Różne parametry regulatora



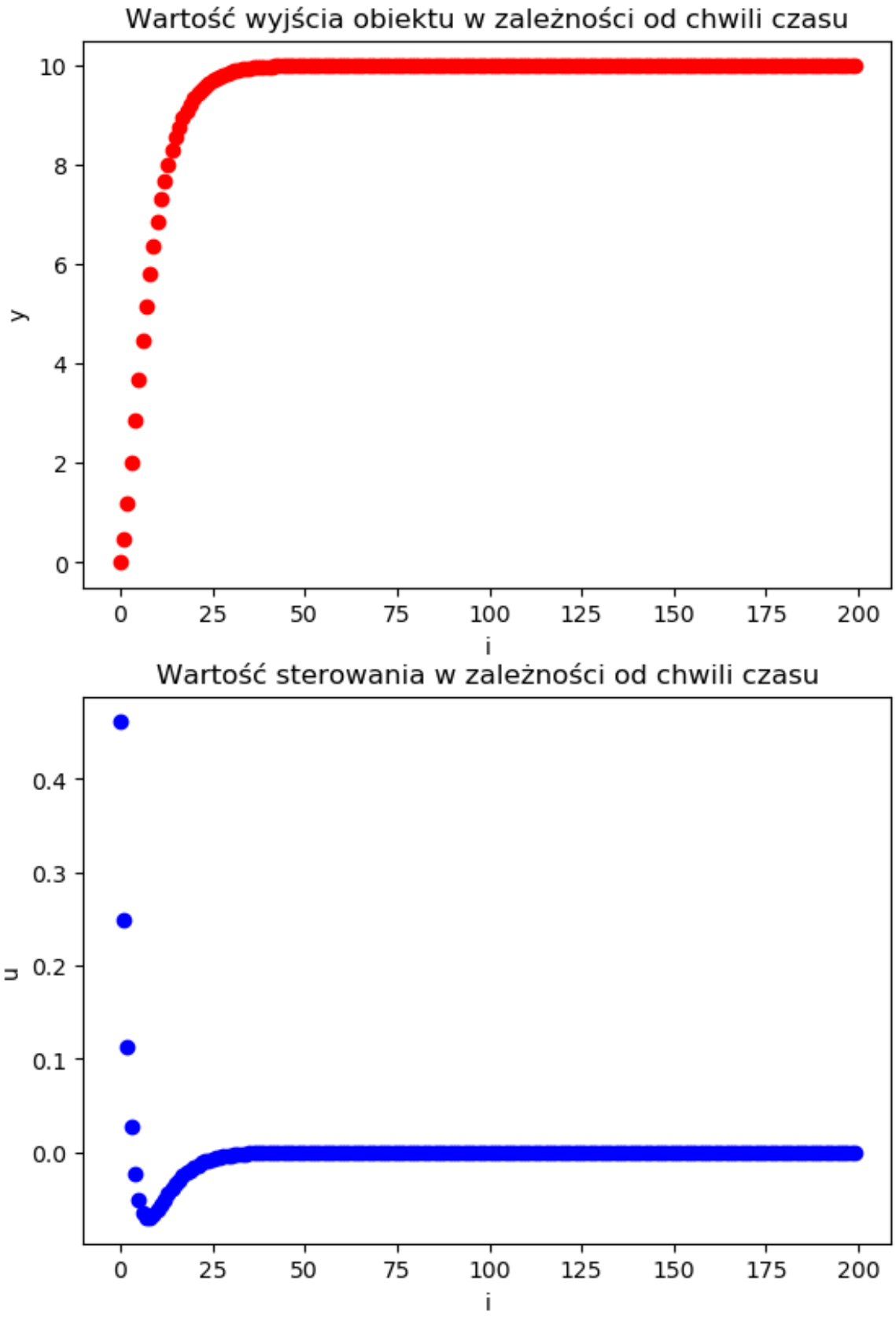
Rysunek 5.1: Nasycone sterowanie



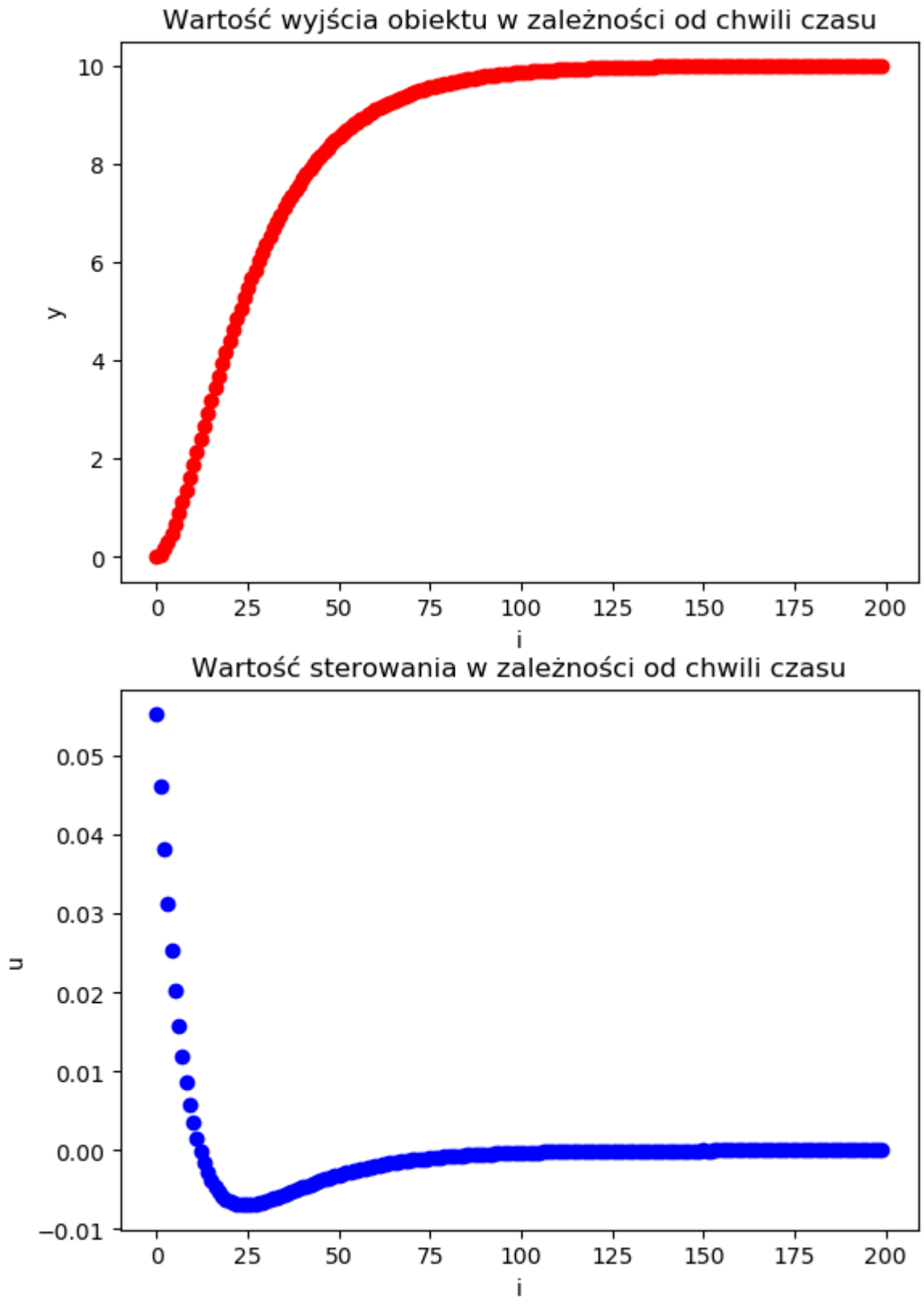
Rysunek 5.2: Horyzont predykcji: 3, horyzont sterowań: 3



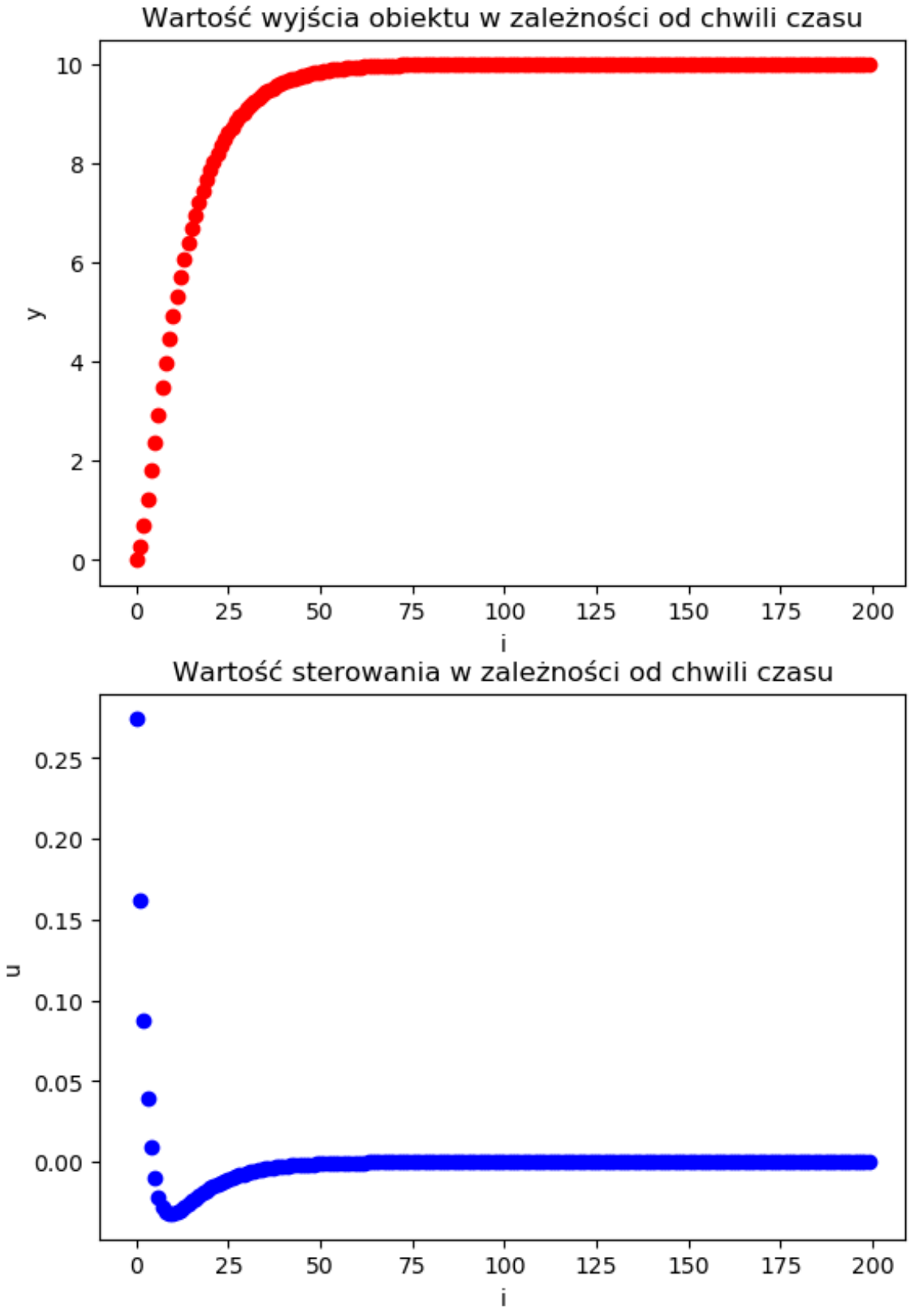
Rysunek 5.3: Horyzont predykcji: 5, horyzont sterowań: 3



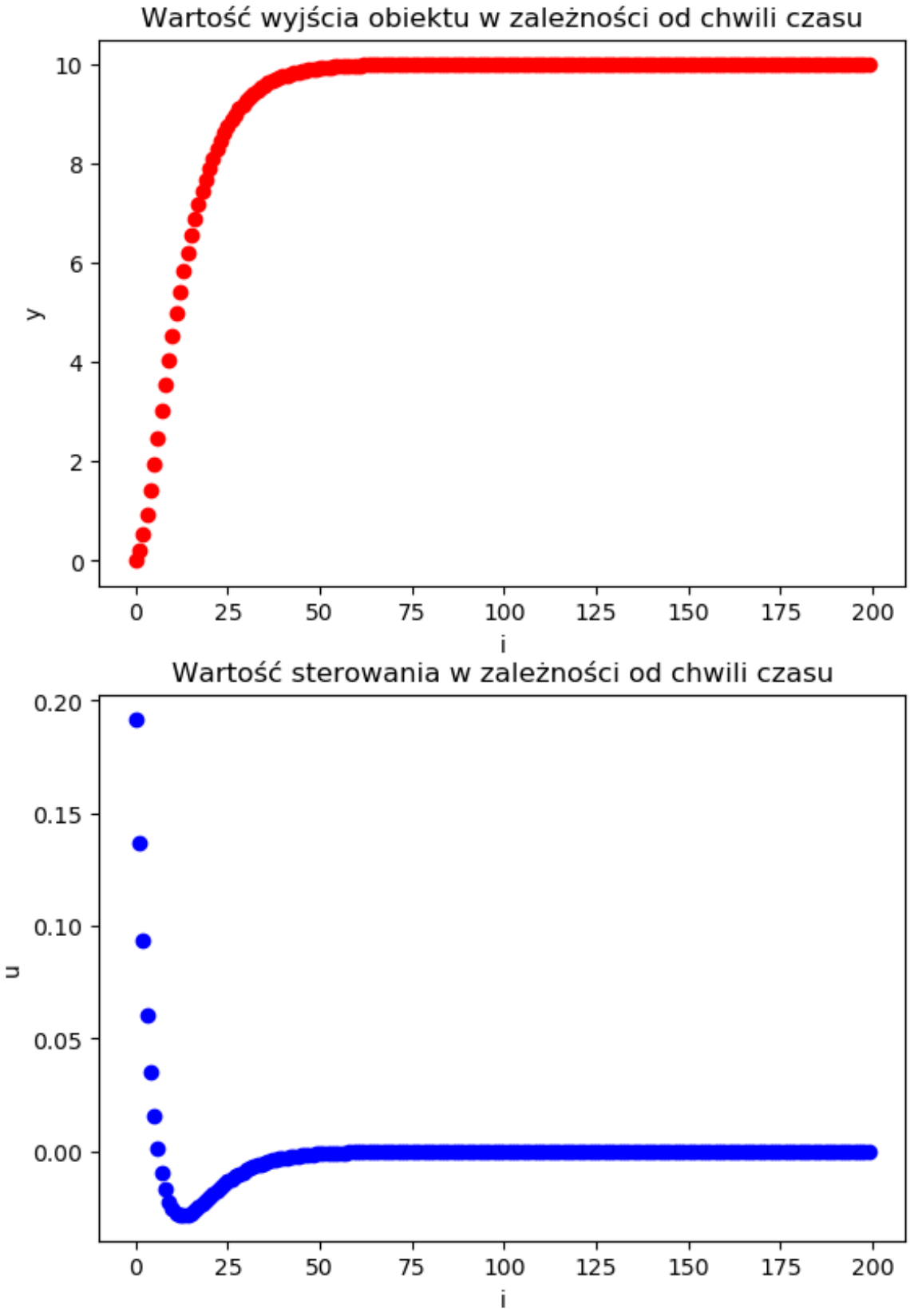
Rysunek 5.4: Horyzont predykcji: 10, horyzont sterowań: 3



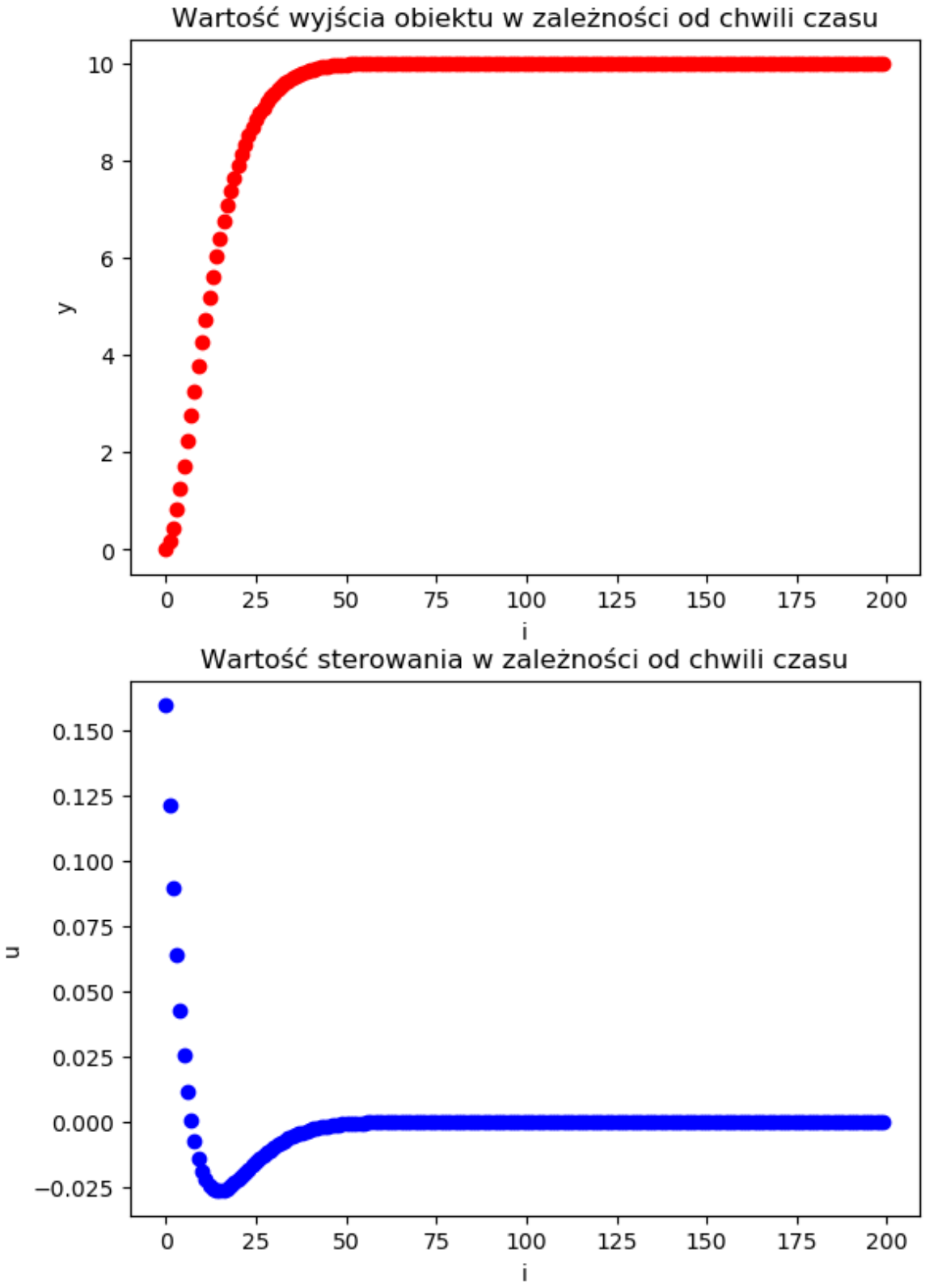
Rysunek 5.5: Horyzont predykcji: 15, horyzont sterowań: 2



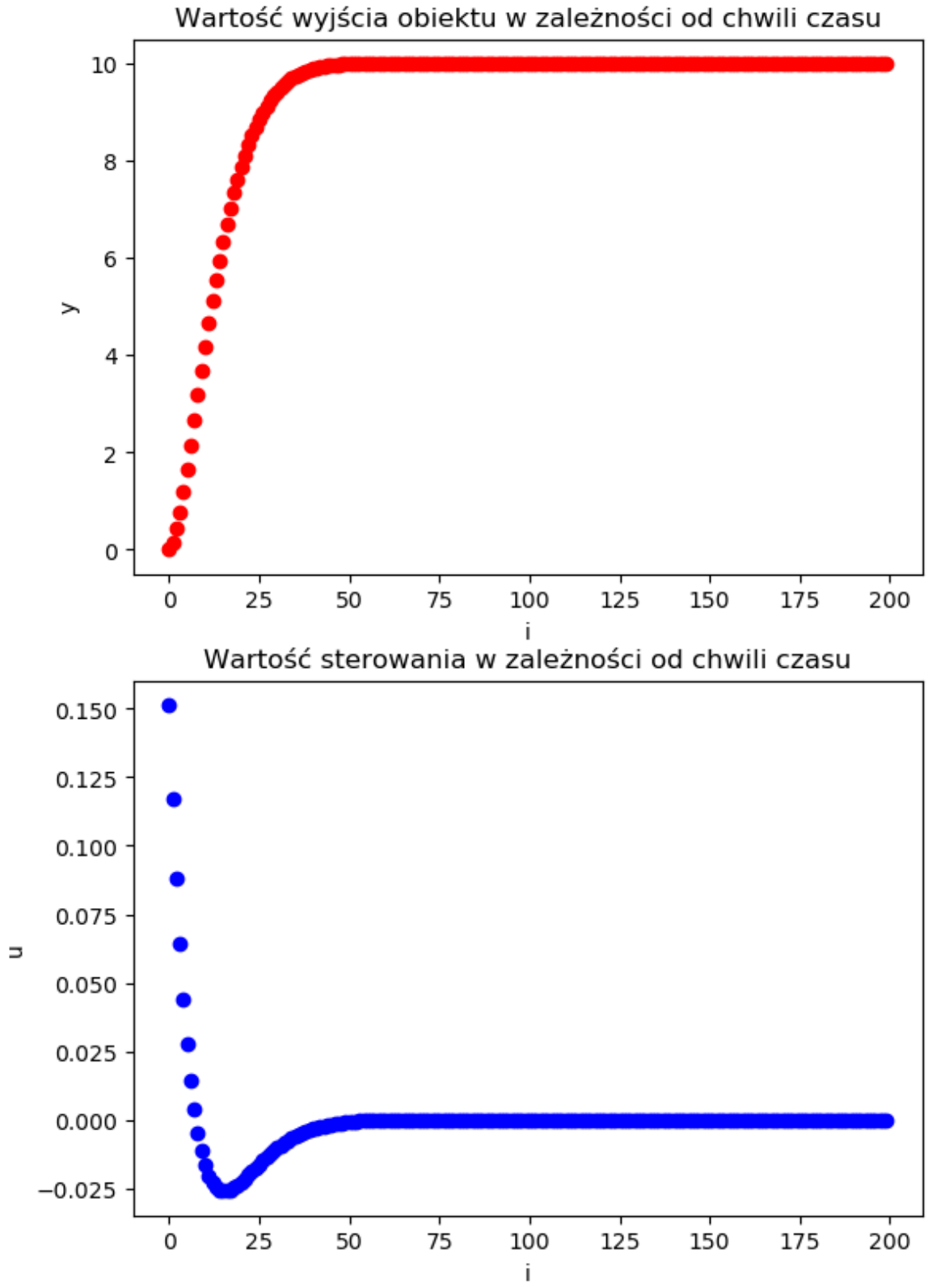
Rysunek 5.6: Horyzont predykcji: 15, horyzont sterowań: 3



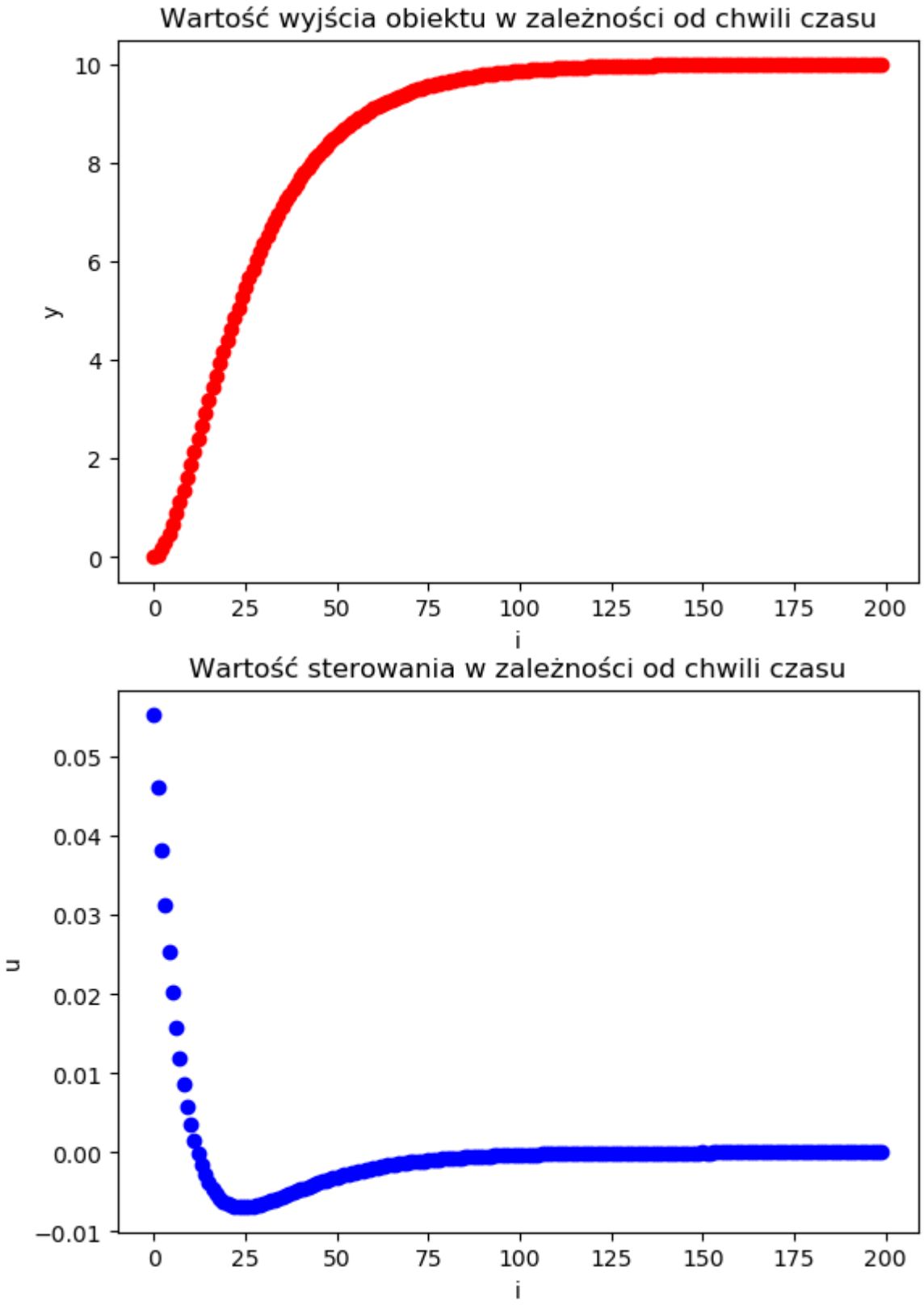
Rysunek 5.7: Horyzont predykcji: 15, horyzont sterowań: 5



Rysunek 5.8: Horyzont predykcji: 15, horyzont sterowań: 7



Rysunek 5.9: Horyzont predykcji: 15, horyzont sterowań: 8



Rysunek 5.10: Horyzont predykcji: 30, horyzont sterowań: 8

Rozdział 6. Podsumowanie

6.1 Wyniki

Zebrane przykładowe wyniki...

6.2 Wnioski

Realizacja postawionych w rozdziale 3.1 założeń pozwoliła na sprawdzenie wpływu horyzontu predykcji oraz sterowań na jakość otrzymanej regulacji. Zbyt mały horyzont predykcji w badanym przypadku skutkował sporą amplitudą kolejnych początkowych sterowań. Zbyt duży horyzont predykcji wpływał na zwiększenie się opóźnienia w układzie regulacji. Zwiększenie się horyzontu sterowań było ściśle powiązane z zwiększeniem się złożoności obliczeniowej problemu, a przede wszystkim konieczności użycia większej ilości pamięci. Ponadto potwierdzono cechę charakterystyczną sterowania predykcyjnego, a mianowicie minimalizację wartości pomiędzy kolejnymi sterowaniami. Im horyzont predykcji był większy, tym owa różnica była mniejsza.

6.3 Pomysły na rozwój projektu

Istnieją dwie płaszczyzny, na których można rozwinąć przygotowany projekt. Pierwsza z nich dotyczy doboru biblioteki do obsługi rejestrów mikrokontrolera platformy STM. Jedną z bardziej przyjaznych użytkowniki alternatywnych opcji do korzystania z Cube'a jest otwartoźródłowy projekt *libopencm3*. Oferowane rozwiązania tejże biblioteki są znacznie bardziej przejrzyste oraz lepiej udokumentowane. Z jej pomocą można bezproblemowo zastąpić komunikację opartą na opisanym w rozdziale 4.2.2 pollingu przerwaniami systemowymi. Kolejnym aspektem stricte programistycznym jest zmiana sposobu alokacji używanych w rozwiązaniu macierzy z dynamicznego na statyczny. Przeniesienie wykonywania największej części obliczeń ze sterty na stos poprawiłoby w znaczny sposób wydajność napisanego programu.

Drugą płaszczyzna jest związana z zakresem badanych funkcjonalności regulatora MPC. Warte rozważenia są w tym przypadku dwie możliwości rozwoju.

Pierwsza z nich zakłada rozszerzenie możliwości pracy regulatora o model nieliniowy przy pomocy techniki linearyzacji. Pozwoliłoby to zweryfikować poprawność pracy zaimplementowanego algorytmu MPC na szerszej dziedzinie obiektów. Docelowym rozwiązaniem jest realizacja indetyfikacji obiektu oraz rozwiązywanie zadania optymalizacji przy nieznanym modelu obiektu.

Dodatki

Dodatek A. Porównanie matlab stm

Spis rysunków

Spis tabel

Spis listingów

Bibliografia

[1] Understanding model predictive control. https://www.mathworks.com/videos/series/understanding-model-predictive-control.html. Dostęp 18.01.2020.

[2] Model predictive control. https://en.wikipedia.org/wiki/Model_predictive_control, 04.01.2020.

[3] Power method. http://ergodic.ugr.es/cphys/LECCIONES/FORTRAN/power_method.pdf, 04.01.2020.

[4] Stm32 nucleo-64 boards (mb1136) user manual. https://www.st.com/content/ccc/resource/technical/document/user_manual/98/2e/fa/4b/e0/82/43/b7/DM00105823.pdf/files/DM00105823.pdf/jcr:content/translations/en.DM00105823.pdf, 04.2019.

[5] Stm32f401xd stm32f401xe datasheet. https://www.espruino.com/datasheets/STM32F401xD.pdf, 2015.

[6] G. Wang et al. State-space model predictive control method for core power control in pressurized water reactor nuclear power stations. *Nuclear Engineering and Technology*, strony 3–4, 2016.

[7] Rolf Findeisen Markus Kögel. A fast gradient method for embedded linear predictive control. *Proceedings of the 18th World Congress The International Federation of Automatic Control*, strony 1362–1367, 28.08 - 02.09.2011.