



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

Streszczenie

Lorem ipsum.

Spis treści

1	Wstęp	1
1.1	Motywacja projektu	1
1.2	Cel pracy	1
2	Idea regulatora MPC	2
2.1	Wstęp	2
2.2	Sposób działania	2
2.3	Model obiektu	3
2.4	Kryterium jakości regulacji	4
2.5	Problem programowania kwadratowego	4
2.6	Pozostałe rodzaje regulatorów klasy MPC	5
2.7	Wady i zalety w porównaniu z regulatorem PID	6
3	Założenia projektowe i wykorzystane narzędzia	7
3.1	Założenia projektowe	7
3.2	Architektura systemu	7
3.2.1	Procesor - architektura ARM	7
3.2.2	Platforma STM	8
3.3	Narzędzia programistyczne	8
3.3.1	Języki programowania C/C++	8
3.3.2	Język programowania Python	9
3.3.3	Środowisko MATLAB	10
3.3.4	STM32CubeMX	10
3.3.5	CMake	11
3.3.6	Kompilator i linker	11
3.3.7	Regex	11
3.4	Przykład referencyjny	11
3.5	Sposób testowania	12
4	Implementacja rozwiązania	13
4.1	Ogólny schemat programu	13
4.2	Szczegóły implementacji - STM	13

4.2.1	Charakterystyczne cechy C++	13
4.2.2	Komunikacja	14
4.2.3	Klasa macierzy	15
4.2.4	Aglorytm	15
4.3	Szczegóły implementacji - PC	15
4.4	Problemy napotkane podczas realizacji	15
5	Przykładowe wyniki	25
5.1	Różne parametry układu i wartości zadane	25
5.2	Różne parametry regulatora	25
6	Podsumowanie	36
6.1	Wyniki	36
6.2	Wnioski	36
6.3	Pomysły na rozwój projektu	36
	Dodatki	37
A	Porównanie matlab stm	38

Rozdział 1. Wstęp

1.1 Motywacja projektu

Lorem ipsum.

1.2 Cel pracy

Lorem ipsum.

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 w 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 identyfikacji 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 przesuwym horyzoncie czasu. W algorytmie regulacji MPC obecny jest także tzw. horyzont sterowania N_c (gdzie $N_c \leq N_p$), po którego upływie przyrost sygnału sterującego wynosi zero. W ten sposób zapew-

nione 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 byćysterowany. 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 zwielokrotnienie 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 inhereśnie związany z predykcją wyjścia obiektu.

W przypadku skalarnym funkcję celu można opisać następującym równaniem:

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

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

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

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

$$H = \phi^T \phi + R_u \quad (2.4)$$

$$W = \phi^T (R_s - Fx_k) \quad (2.5)$$

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

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

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

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

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)

Tablica 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 ułatwiające 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.

Arithmetic Logic Unit (ALU) The ALU has two 32-bits inputs. The primary comes from the register file, whereas the other comes from the shifter. Status registers flags modified by the ALU outputs. The V-bit output goes to the V flag as well as the Count goes to the C flag. Whereas the foremost significant bit really represents the S flag, the ALU output operation is done by NORed to get the Z flag. The ALU has a 4-bit function bus that permits up to 16 opcode to be implemented.

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.

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 wbudowa-

nych. C domyślnie zapewnia narzędzia, które w sposób efektywny są kompilowane do kodu maszynowego. Taki kod ma porównywalną wydajność do programów napisanych jedynie za pomocą assemblera. Jest to powiązane ze względną niskopoziomowością języka C, która zrealizowana została m.in. za pomocą ręcznej alokacji dynamicznej pamięci, a także minimalnego wsparcia 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 przeznaczona pamięć dla tej zmiennej.

C++ jest to wieloparadygmataowy, 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 stercku, których 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 obiektowo język programowania. Zawiera on moduły, wyjątki, dynamiczne typowanie, wysokopoziom-

mowe 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 przenośność 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 wchodzi 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 wyrażen 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 algorytm 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 pokazano 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++

C++, singleton, hermetyzacja danych, clean code 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-

```
1  #ifndef MPC_STM_DATAPARSER_HPP
2  #define MPC_STM_DATAPARSER_HPP
3
4  #include <map>
5  #include <vector>
6  #include <string>
7
8  namespace Utils {
9      class DataParser {
10     public:
11         static bool isNewDataGoingToBeSend;
12
13         DataParser() = default;
14         ~DataParser() = default;
15         void ParseReceivedMsg(const std::string& msg);
16
17         [[nodiscard]] std::map<std::string,
18         std::vector<double>> GetStorage() const;
19
20         void ClearStorage();
21
22     private:
23         constexpr static const char* regexPattern =
24             R"([[:alpha:]]+)(': )(\[. +?\])";
25
26         std::map<std::string, std::vector<double>> storage;
27     };
28 }
29
30 #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 STL'a. Stało się to możliwe dzięki pokaznemu, jak na systemy wbudowane, rozmiarowi pamięci flash platformy Nucleo.

4.2.2 Komunikacja

Ramka, schemat, itp

4.2.3 Klasa macierzy

4.2.4 Aglorytm

Jak zostało zrealizowane założenie

4.3 Szczegóły implementacji - PC

4.4 Problemy napotkane podczas realizacji

HAL - przerwania, ramka UARTa Model układu - feasibility

```
1  #include ...
2
3  bool Utils::DataParser::isNewDataGoingToBeSend = false;
4
5  std::map<std::string, std::vector<double>>
   ↪  Utils::DataParser::GetStorage() const {
6      return storage;
7  }
8
9  void Utils::DataParser::ParseReceivedMsg(const std::string& msg) {
10     std::string valuesMatch;
11     std::regex pattern(regexPattern);
12     std::smatch fullMatch;
13     std::string::const_iterator iterator(msg.cbegin());
14
15     while (std::regex_search(iterator, msg.cend(), fullMatch, pattern))
16     ↪  {
17         valuesMatch = fullMatch[3].str();
18         std::replace(valuesMatch.begin(), valuesMatch.end(), ',', ' ');
19         valuesMatch.pop_back(); // trim ]
20         valuesMatch.erase(0, 1); // trim [
21         if (fullMatch[1].str().find('A') != std::string::npos) {
22             Utils::Misc::StringToDouble(valuesMatch, storage["A"]);
23         } else if (fullMatch[1].str().find('B') != std::string::npos) {
24             Utils::Misc::StringToDouble(valuesMatch, storage["B"]);
25         } else if (fullMatch[1].str().find('C') != std::string::npos) {
26             Utils::Misc::StringToDouble(valuesMatch, storage["C"]);
27         } else if (fullMatch[1].str().find("set") != std::string::npos)
28         ↪  {
29             Utils::Misc::StringToDouble(valuesMatch, storage["set"]);
30         } else if (fullMatch[1].str().find("control") !=
31         ↪  std::string::npos) {
32             Utils::Misc::StringToDouble(valuesMatch,
33             ↪  storage["control"]);
34         } else if (fullMatch[1].str().find("horizon") !=
35         ↪  std::string::npos) {
36             Utils::Misc::StringToDouble(valuesMatch,
37             ↪  storage["horizons"]);
38         }
39         iterator = fullMatch.suffix().first;
40     }
41 }
42
43 void Utils::DataParser::ClearStorage() {
44     storage.clear();
45 }
```

Listing 2: DataParser.cpp: Przykładowy plik źródłowy

```

1 namespace HAL {
2     class Peripherals {
3     public:
4         Peripherals(const Peripherals&) = delete;
5         Peripherals& operator=(const Peripherals&) = delete;
6         Peripherals(Peripherals&&) = delete;
7         Peripherals& operator=(Peripherals&&) = delete;
8
9         static Peripherals& GetInstance();
10    };
11 }

```

Listing 3: Peripherals.hpp: Wzorzec projektowy - singleton

```

1 HAL::Peripherals& HAL::Peripherals::GetInstance() {
2     static Peripherals instance;
3     return instance;
4 }

```

Listing 4: Peripherals.cpp: Wzorzec projektowy - singleton

```

1 void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
2     if(GPIO_Pin == B1_Pin) {
3         HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
4         Utils::DataParser::isNewDataGoingToBeSend = true;
5     }
6 }

```

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

```

1 void HAL::Peripherals::SendString(const uint8_t* string, uint16_t
   ↪ timeout) {
2     HAL_UART_Transmit(&handleUART, const_cast<uint8_t*>(string),
3     std::strlen(reinterpret_cast<const char*>(string)), timeout);
4 }

```

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

```

1  #include ...
2
3  class CMatrix {
4  protected:
5      uint32_t rows;
6      uint32_t columns;
7      double** matrix{};
8      void make_matrix();
9
10 public:
11     CMatrix();
12     CMatrix(uint32_t rows, uint32_t columns);
13     CMatrix(uint32_t rows, uint32_t columns, double** mat);
14     CMatrix(uint32_t rows, uint32_t columns, const double* mat);
15     explicit CMatrix(uint32_t rows, const std::string& value = "");
16     CMatrix(const CMatrix& M);
17     ~CMatrix();
18
19     [[nodiscard]] uint32_t GetRows() const { return rows; }
20     [[nodiscard]] uint32_t GetColumns() const { return columns; }
21     void GetSize(uint32_t& rows, uint32_t& columns) const;
22     [[nodiscard]] double& GetElement(uint32_t row, uint32_t column)
23         ↪ const;
24     void SetRow(uint32_t row, const CMatrix& v);
25     void SetColumn(uint32_t column, const CMatrix& v);
26     void SetValue(double value);
27     [[nodiscard]] double Det() const;
28     [[nodiscard]] CMatrix Inverse() const;
29
30     double* operator[](uint32_t row) const;
31     CMatrix& operator= (const CMatrix& m);
32     CMatrix operator+ (const CMatrix& m) const;
33     CMatrix& operator+= (const CMatrix& m);
34     CMatrix operator- (const CMatrix& m) const;
35     CMatrix& operator-= (const CMatrix& m);
36     CMatrix operator* (const CMatrix& m) const;
37     CMatrix& operator*= (const CMatrix& m);
38     CMatrix operator* (const double& scalar) const;
39     CMatrix& operator*= (const double& scalar);
40     CMatrix operator/ (const double& scalar) const;
41     CMatrix& operator/= (const double& scalar);
42     CMatrix operator^ (const uint32_t& exponent) const;
43     CMatrix operator-() const;
44     [[nodiscard]] CMatrix T() const;
45 };

```

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

```
1 double Control::MPC::FastGradientMethod(const std::string& msg) {
2     CMatrix J(1,1), J_prev(1,1);
3     Utils::Misc::StringToDouble(msg, sys.x);
4     for (uint32_t i = 0; i < 100; i++) {
5         opt.W = opt.fi.T() * ((opt.F * sys.x) - opt.Rs);
6         CalculateProjectedGradientStep();
7         J_prev = J;
8         J = sys.v.T() * opt.H * sys.v / 2 + sys.v.T() * opt.W;
9         if (std::fabs(J_prev[0][0] - J[0][0]) < eps) {
10             break;
11         }
12     }
13     return sys.v[0][0];
14 }
```

Listing 9: MPC.cpp: Implementacja szybkiej metody gradientowej

```
1 void Control::MPC::CalculateProjectedGradientStep() {
2     CVector gradient(sys.v.GetRows(), 1), v_old = sys.v;
3     gradient = opt.H * sys.v + opt.W;
4     sys.w = sys.v - gradient * eigenvalues.step;
5     for (uint32_t i = 0; i < sys.v.GetRows(); i++) {
6         if (sys.w[i][0] < controlValues.min) {
7             sys.v[i][0] = controlValues.min;
8         }
9         else if (sys.w[i][0] > controlValues.max) {
10             sys.v[i][0] = controlValues.max;
11         } else {
12             sys.v[i][0] = sys.w[i][0];
13         }
14     }
15     sys.w = sys.v + (sys.v - v_old) * eigenvalues.fastConvergence;
16 }
```

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


```

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 11: MPC.cpp: Implementacja zadawania nowych parametrów

```
1 void Control::MPC::CalculateOptimizationMatrices() {
2     opt.Rw *= opt.scalarRw;
3     opt.Rs.SetValue(opt.scalarRs);
4     CVector productMatrix(sys.C.GetColumns());
5     productMatrix = sys.C * (sys.A ^ (horizons.prediction -
6     ↪ horizons.control));
7     for (uint32_t i = horizons.prediction; i != 0; i--) {
8         for (uint32_t j = horizons.control; j != 0; j--) {
9             if (i == horizons.prediction) {
10                 if (j < horizons.control) {
11                     productMatrix *= sys.A;
12                 }
13                 opt.fi[i-1][j-1] = (productMatrix * sys.B)[0][0];
14             } else if (i < j && j < horizons.control) {
15                 opt.fi[i-1][j-1] = opt.fi[i][j];
16             }
17         }
18     }
19     opt.H = opt.fi.T() * opt.fi + opt.Rw;
20     productMatrix = sys.C * sys.A;
21     for (uint32_t i = 0; i < horizons.prediction; i++) {
22         opt.F.SetRow(i, productMatrix);
23         productMatrix *= sys.A;
24     }
25 }
```

Listing 12: MPC.cpp: Implementacja wyliczenia nowych macierzy optymalizacyjnych

```

1 projectPath = os.path.dirname(os.path.abspath(__file__))
2 changeParameters = True
3 systemParameters = {'A': [1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0,
    ↪ 1],
4                       'B': [0, 0, 1, 0],
5                       'C': [1, 1, 0, 0],
6                       'setPoint': [10],
7                       'controlExtremeValues': [-10, 10],
8                       'horizons': [30, 8]}
9 systemParametersToSend = str(systemParameters)[1:-1] + '\n\0'
10 A = np.array(systemParameters['A']).\
11     reshape(-1, int(np.sqrt(len(systemParameters['A']))))
12 B = np.array(systemParameters['B']).\
13     reshape(len(systemParameters['B']), -1)
14 C = np.array(systemParameters['C'])
15 x = np.zeros((len(systemParameters['B']), 1))
16 y = []
17 u = []
18 timer = []

```

Listing 13: Inicjalizacja danych w skrypcie Pythona

```

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

```

Listing 14: Główna pętla przeprowadzająca test HIL

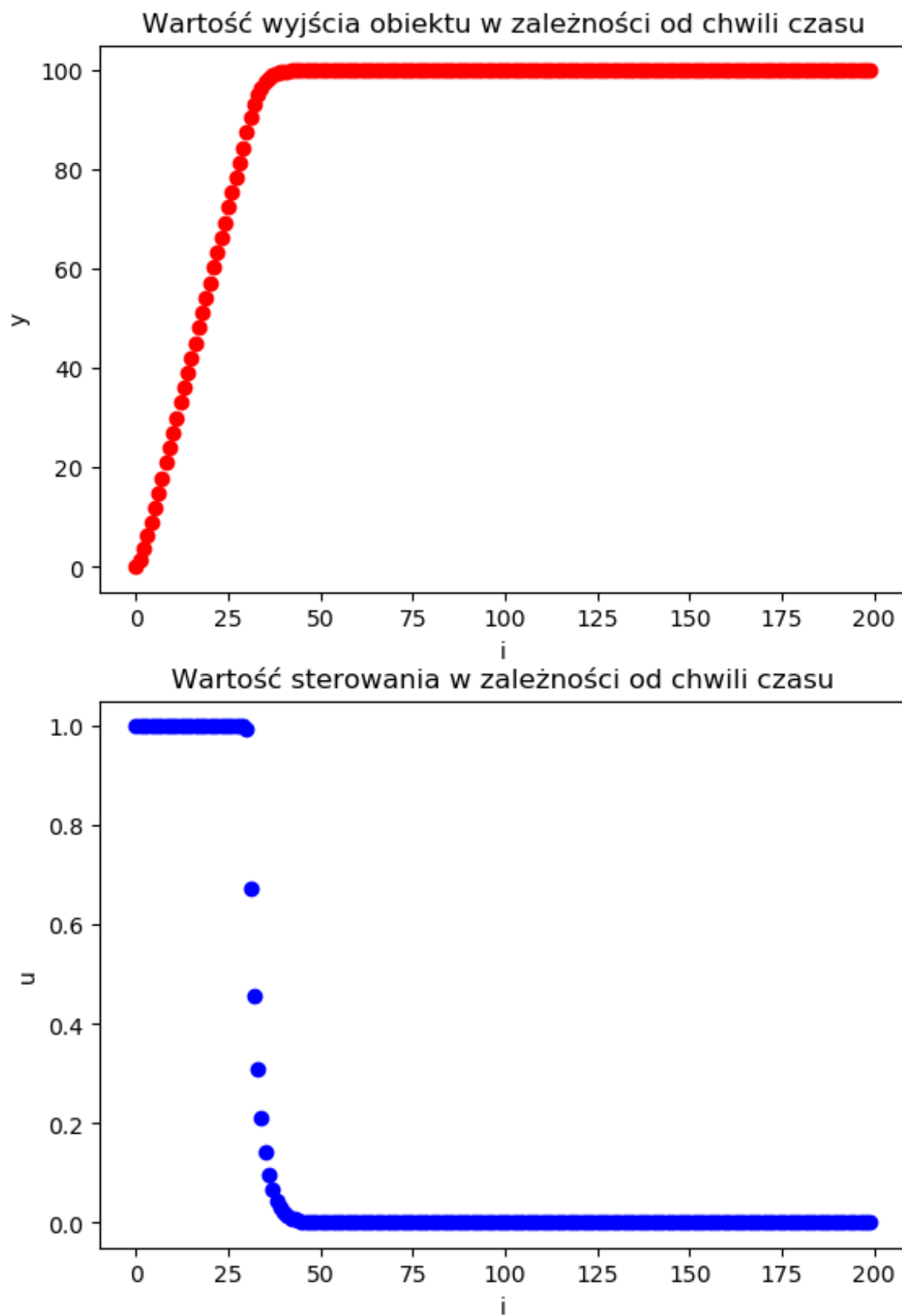
```
1 _, fig = plt.subplots(2, 1, figsize=(6.4, 9.6))
2 fig[0].set_title('Wartość wyjścia obiektu w zależności od chwili
   ↪ czasu')
3 fig[0].plot(y, 'ro'), fig[0].set_ylabel('y'), fig[0].set_xlabel('i')
4 fig[1].set_title('Wartość sterowania w zależności od chwili czasu')
5 fig[1].plot(u, 'bo'), fig[1].set_ylabel('u'), fig[1].set_xlabel('i')
6 plt.savefig(r'{0}\plots\horizons_{1}.png'.format
7             (projectPath, systemParameters['horizons']), format='png',
   ↪             bbox_inches='tight')
8
9 with open('log.txt', 'a') as logger:
10     logger.write(str(systemParameters['A']))
11     logger.write('Init time, Mean of timestamps, '
12                 ' Stdev of timestamps, Max of timestamps\n')
13     logger.write(f"{initTimer:.4} {statistics.mean(timer):.4}"
14                 f"{statistics.stdev(timer):.4} {max(timer):.4}\n")
```

Listing 15: Zapisywanie danych i wykresów

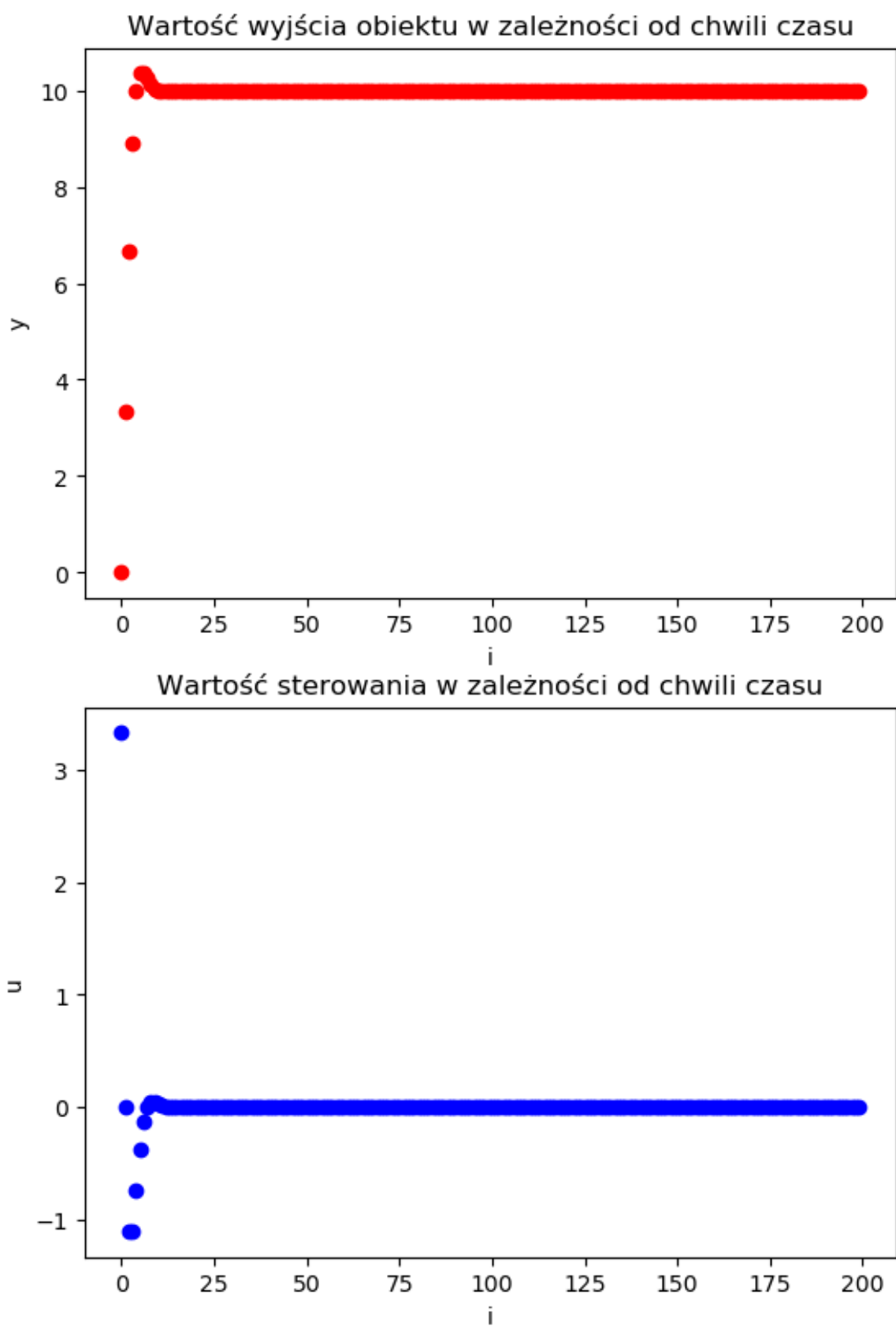
Rozdział 5. Przykładowe wyniki

5.1 Różne parametry układu i wartości zadane

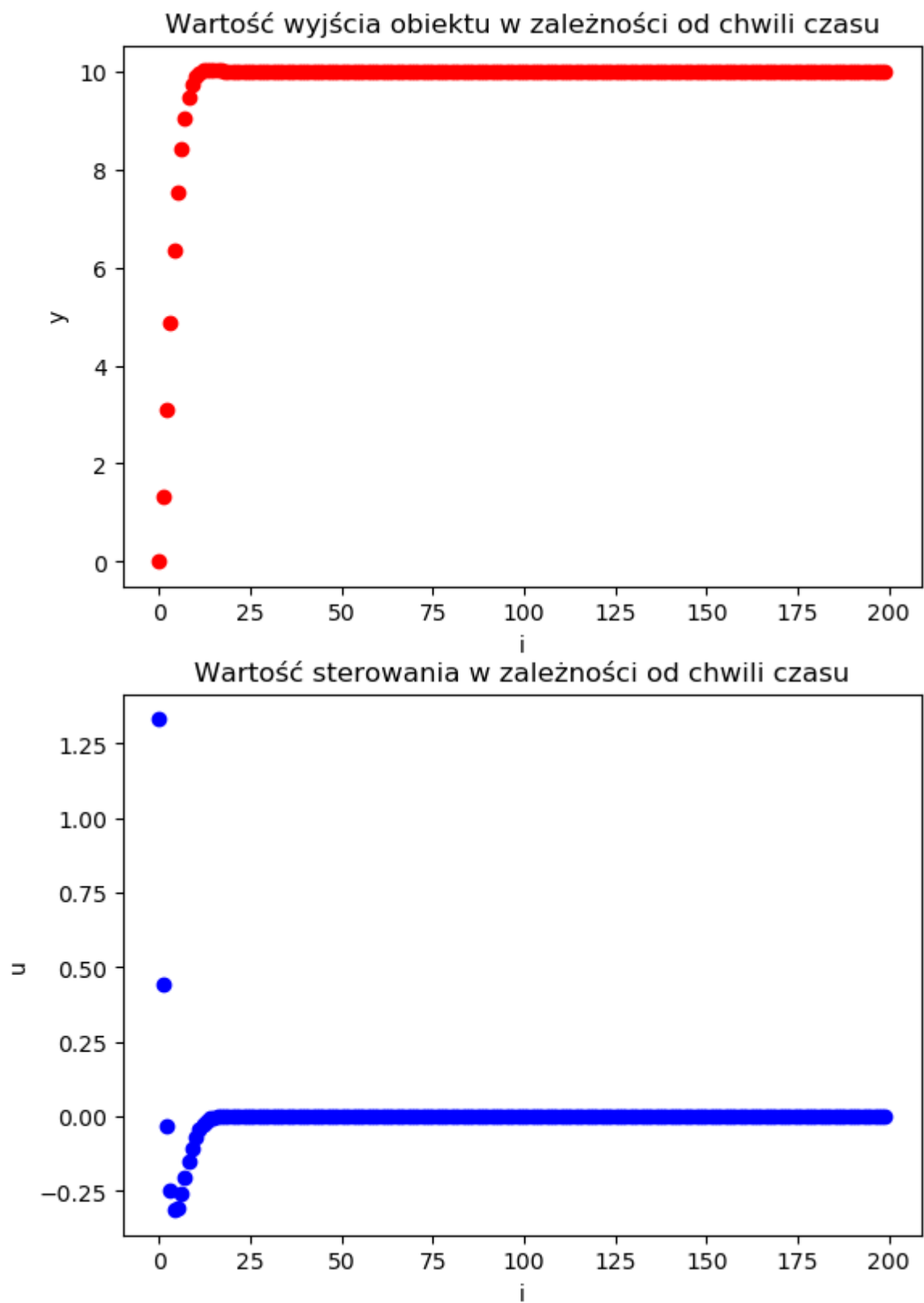
5.2 Różne parametry regulatora



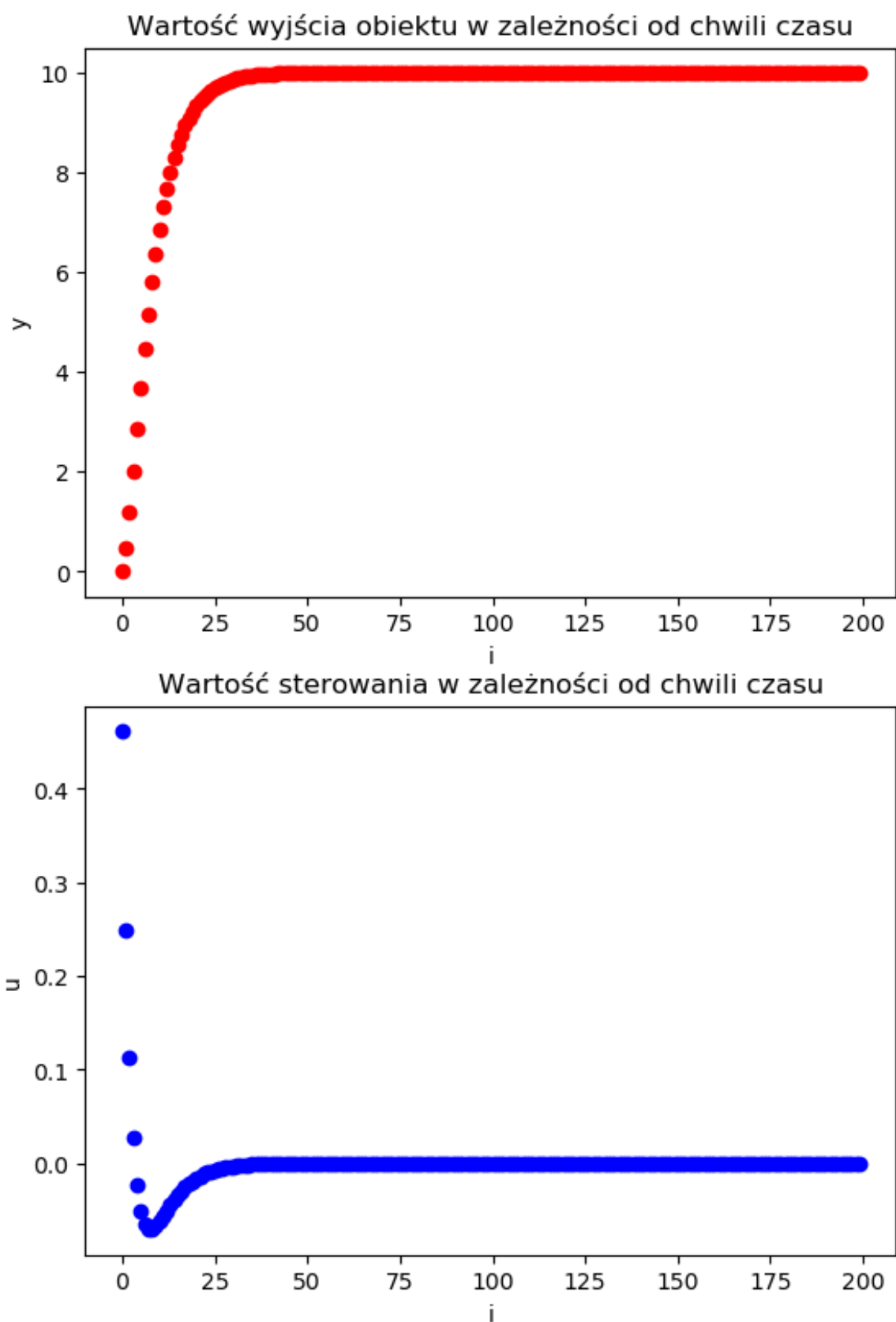
Rysunek 5.1: Nasycone sterowanie



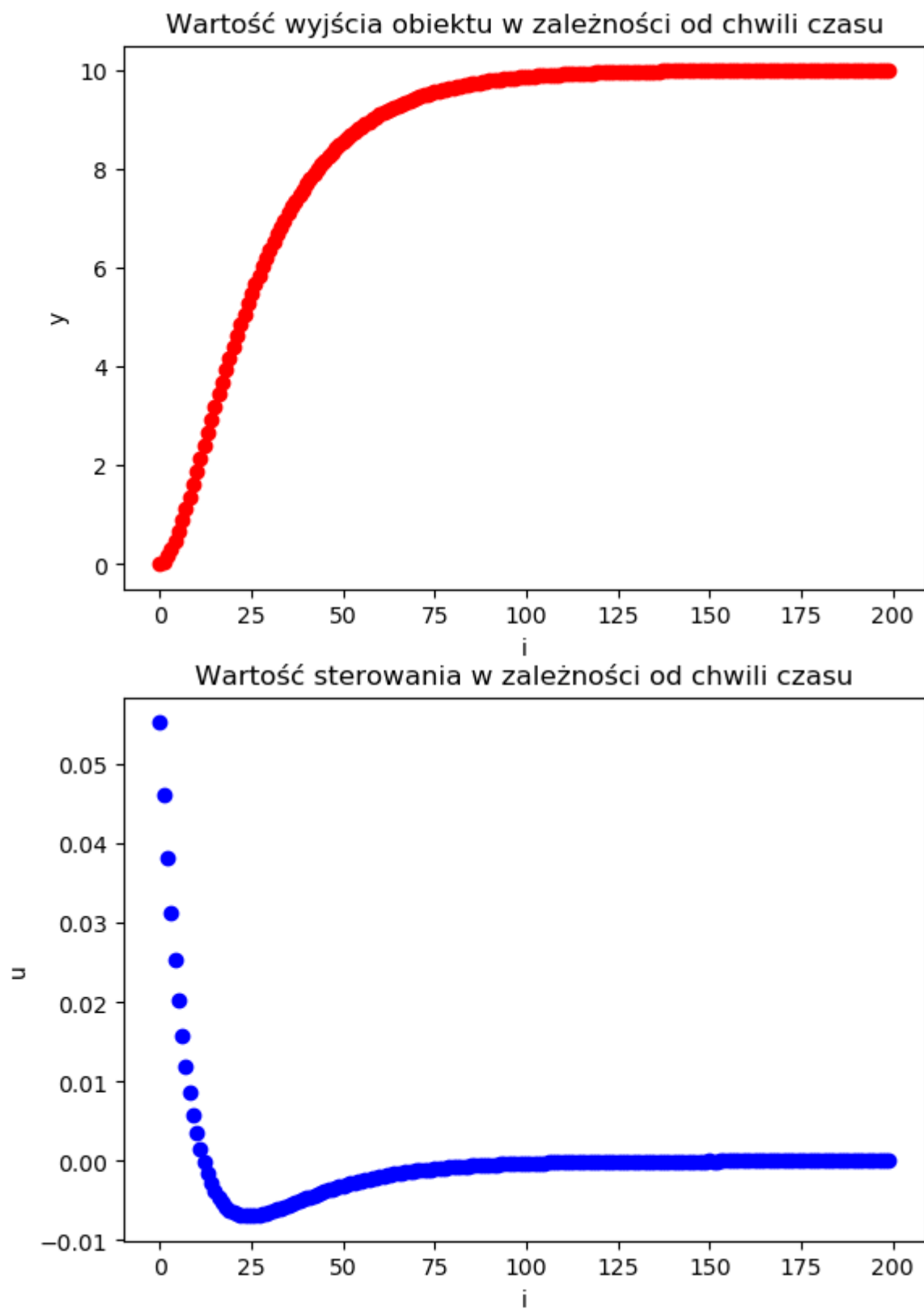
Rysunek 5.2: Horyzont predykcji: 3, horyzont sterowań: 3



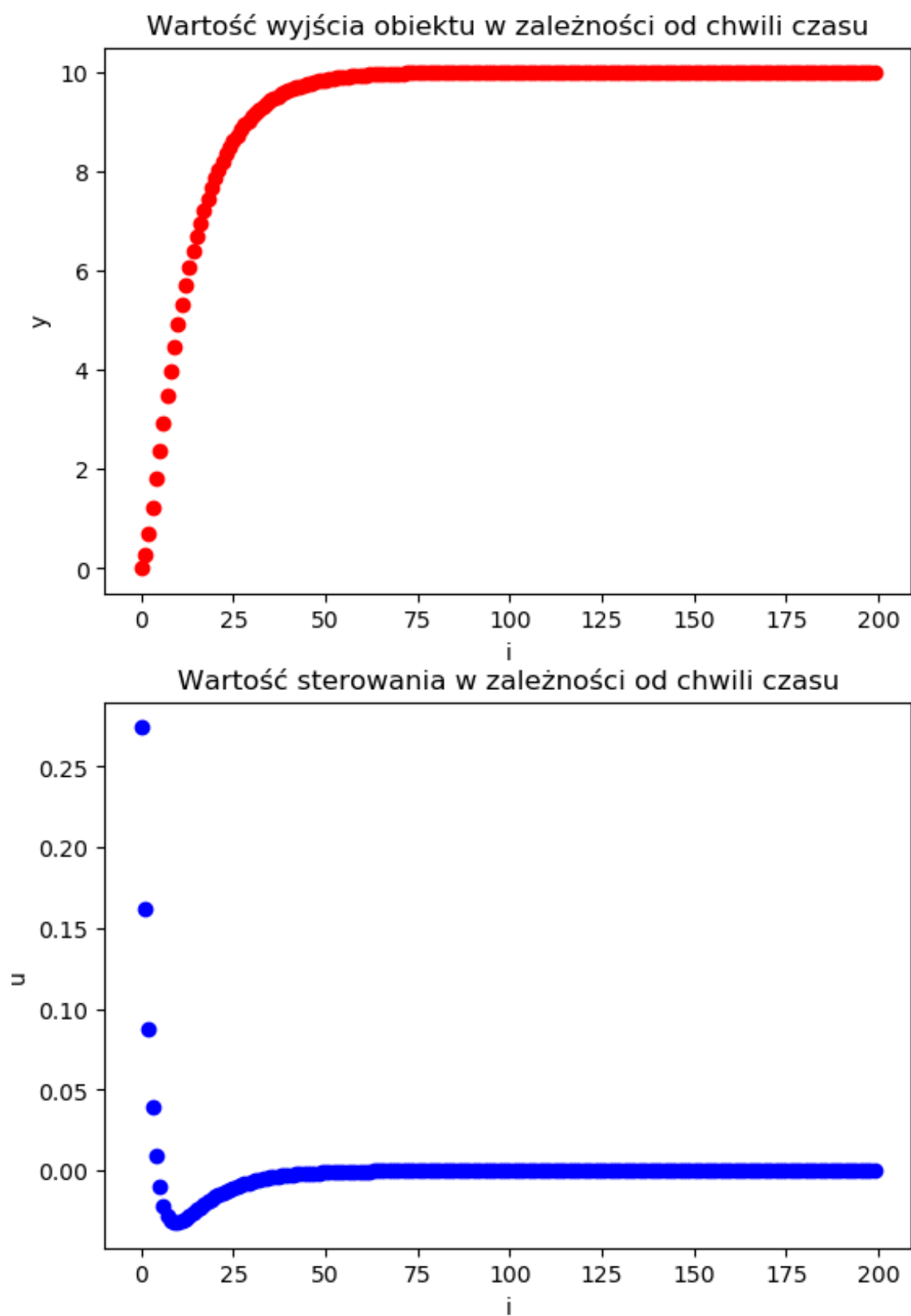
Rysunek 5.3: Horyzont predykcji: 5, horyzont sterowań: 3



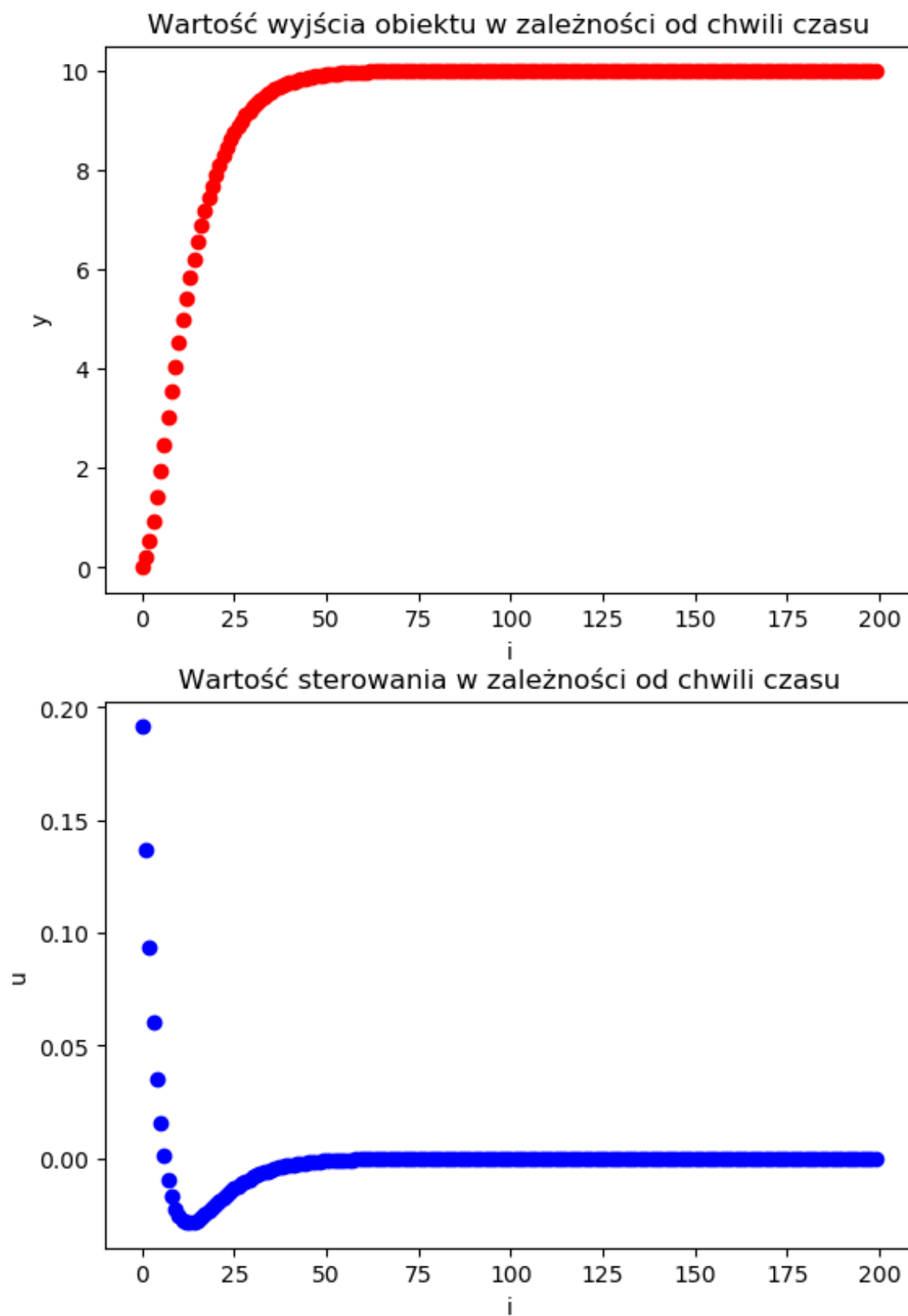
Rysunek 5.4: Horyzont predykcji: 10, horyzont sterowań: 3



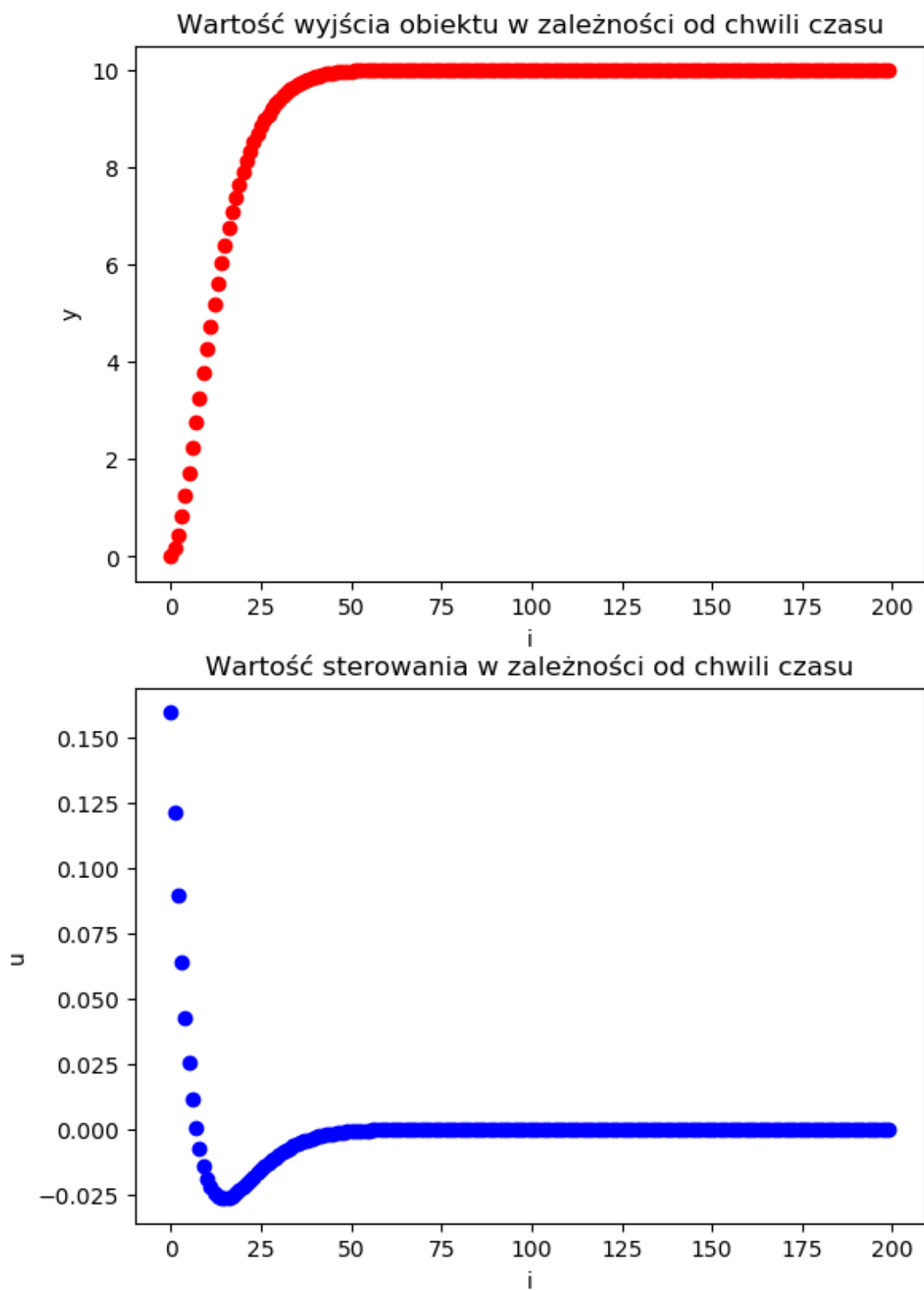
Rysunek 5.5: Horyzont predykcji: 15, horyzont sterowań: 2



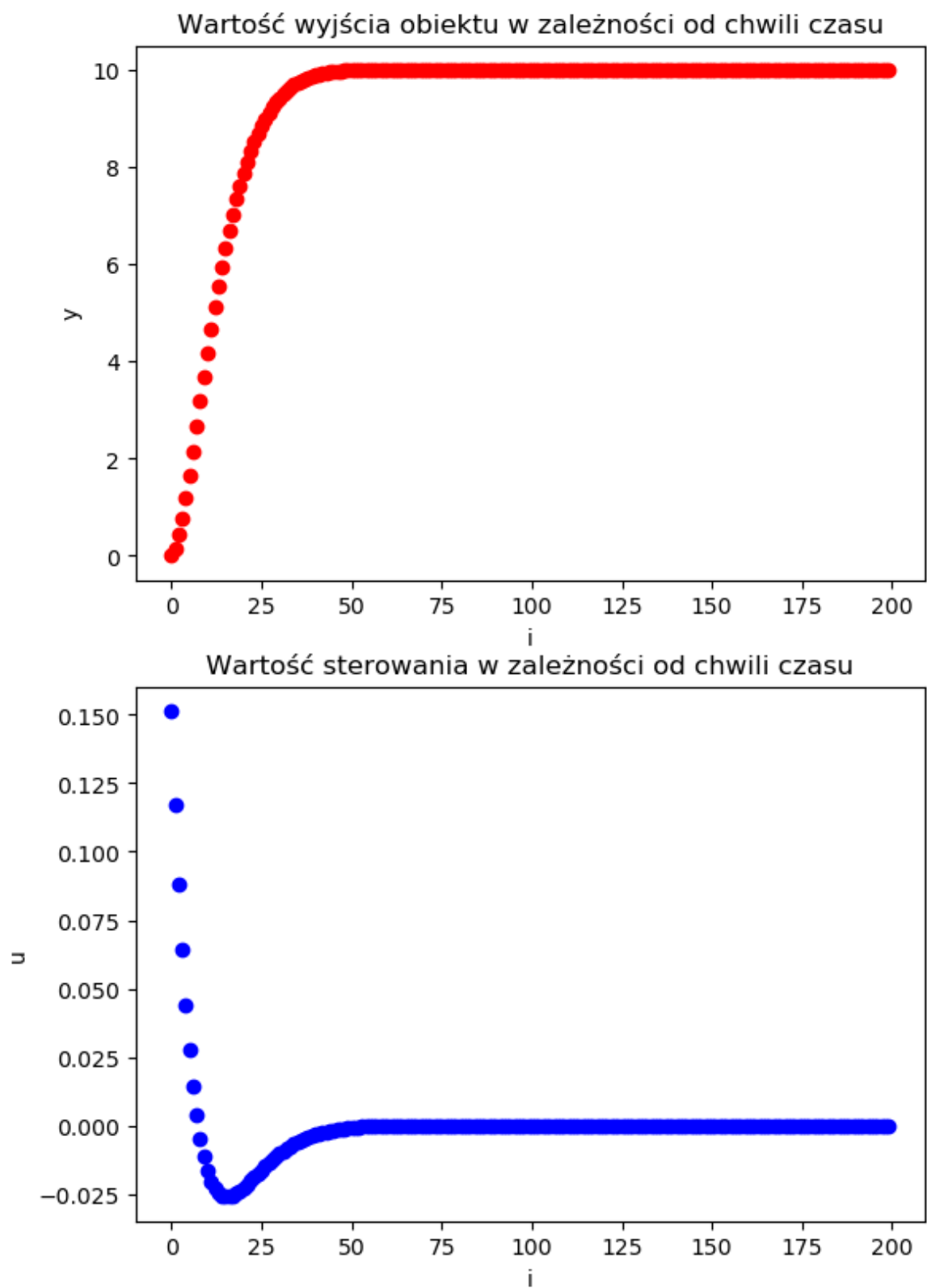
Rysunek 5.6: Horyzont predykcji: 15, horyzont sterowań: 3



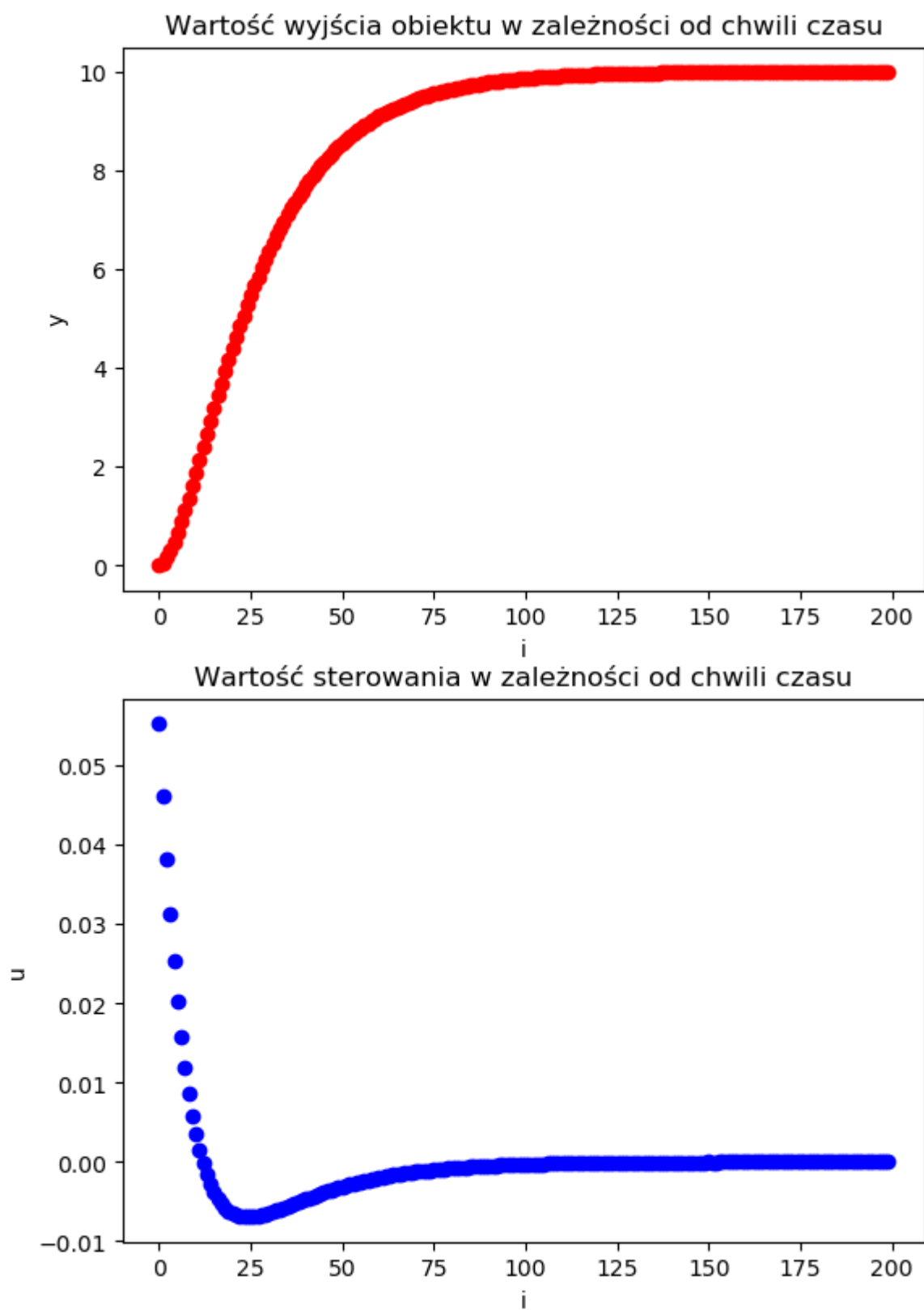
Rysunek 5.7: Horyzont predykcji: 15, horyzont sterowań: 5



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

Krótkie podsumowanie

6.2 Wnioski

Zgodność z założeniem

6.3 Pomysły na rozwój projektu

Implementacja linearyzacji

Dodatki

Dodatek A. Porównanie matlab stm

Spis rysunków

5.1	Nasycone sterowanie	26
5.2	Horyzont predykcji: 3, horyzont sterowań: 3	27
5.3	Horyzont predykcji: 5, horyzont sterowań: 3	28
5.4	Horyzont predykcji: 10, horyzont sterowań: 3	29
5.5	Horyzont predykcji: 15, horyzont sterowań: 2	30
5.6	Horyzont predykcji: 15, horyzont sterowań: 3	31
5.7	Horyzont predykcji: 15, horyzont sterowań: 5	32
5.8	Horyzont predykcji: 15, horyzont sterowań: 7	33
5.9	Horyzont predykcji: 15, horyzont sterowań: 8	34
5.10	Horyzont predykcji: 30, horyzont sterowań: 8	35

Spis tablic

2.1 Porównanie regulacji PID i MPC	6
--	---

Spis listingów

1	DataParser.hpp: Przykładowy plik nagłówkowy	14
2	DataParser.cpp: Przykładowy plik źródłowy	16
3	Peripherals.hpp: Wzorzec projektowy - singleton	17
4	Peripherals.cpp: Wzorzec projektowy - singleton	17
5	Peripherals.cpp: Implementacja wywołania zwrotnego (callback) podczas przerwania	17
6	Peripherals.cpp: Implementacja wysyłania danych z platformy STM do PC	17
7	Peripherals.cpp: Implementacja odbierania danych przez platformę STM	18
8	Matrix.hpp: Fragment pliku nagłówkowego zawierającego własną implementację macierzy	19
9	MPC.cpp: Implementacja szybkiej metody gradientowej	20
10	MPC.cpp: Implementacja obliczenia kolejnego kroku w algorytmie gradientowym	20
11	MPC.cpp: Implementacja zadawania nowych parametrów	21
12	MPC.cpp: Implementacja wyliczenia nowych macierzy optymalizacyjnych	22
13	Inicjalizacja danych w skrypcie Pythona	23
14	Główna pętla przeprowadzająca test HIL	23
15	Zapisywanie danych i wykresów	24

Bibliografia

- [1] Model predictive control. https://en.wikipedia.org/wiki/Model_predictive_control, 04.01.2020.
- [2] Power method. http://ergodic.ugr.es/cphys/LECCIONES/FORTRAN/power_method.pdf, 04.01.2020.
- [3] Understanding model predictive control. <https://www.mathworks.com/videos/series/understanding-model-predictive-control.html>, 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.