POLITECHNIKA POZNAŃSKA

WYDZIAŁ ELEKTRYCZNY

INSTYTUT AUTOMATYKI I INŻYNIERII INFORMATYCZNEJ

ZAKŁAD STEROWANIA I ELEKTRONIKI PRZEMYSŁOWEJ

PROJEKT PRZEJŚCIOWY

IMPLEMENTACJA ALGORYTMU REKURENCYJNEJ METODY NAJMNIEJSZYCH KWADRATÓW I ZMIENNYCH INSTRUMENTALNYCH W C++ DO PARAMETRYCZNEJ IDENTYFIKACJI OBIEKTU

PATRYK KOTLARZ

PROWADZĄCY:
DR INŻ. DARIUSZ JANISZEWSKI

POZNAŃ, 2.05.2016

Spis treści

I. WPROWADZENIE

1. WSTĘP

Praca jest projektem zaliczeniowym z przedmiotu „Projekt przejściowy" prowadzonym przez dr inż. Dariusza Janiszewskiego na kierunku automatyka i robotyka. Programy zostały pisane w języku C++ w środowiskach True Studio (1), Dev-C++ (2) oraz Visual Studio (3). Do testowania algorytmów zostało użyte środowisko Matlab (4). Do napisania dokumentu została wykorzystana formatka sprawozdania (5), której autorem jest dr inż. Dominik Łuczak.

2. CEL IDENTYFIKACJI OBIEKTÓW STEROWANIA I ZAŁOŻENIA PROJEKTOWE

Tematem projektu była implementacja algorytmu rekurencyjnej metody najmniejszych kwadratów (RLS) oraz pokrewnej rekurencyjnej metody zmiennych instrumentalnych (RIV) w języku programowania powszechnie używanym przy projektowaniu oprogramowania dla mikrokontrolerów. Są to numeryczne metody identyfikacji parametrów dyskretnej liniowego modelu (aproksymacja rzeczywistego obiektu) działające online (identyfikacja w trakcie trwania procesu) szczegółowiej opisane w dalszej części. Ze względu na ich rekurencyjny charakter nadają się w procesach sterowania za pomocą układu mikroprocesorowego, w którym nastawy cyfrowego regulatora można dostosować w trakcie pracy do zidentyfikowanego modelu. W owym projekcie nie została wykonana synteza regulatora ze względu na ograniczony wymiar pracy.

Wybrane zostały metody RLS oraz RIV ze względu na ich dobre opracowanie w programie studiów i literaturze oraz dużą powszechność stosowania w programach przeznaczonych stricte do identyfikacji obiektu na podstawie próbek sygnału wejściowego i wyjściowego (np. System Identification Toolbox (6) dla środowiska Matlab i Simulink).

Ze względu na to, iż język programowania powinien być dostępny dla dużej części mikrokontrolerów, początkowo został wybrany czysty język C. Przy pierwszych wersjach projektu pojawiły się problemy przy przydzielaniu i zwalnianiu pamięci, dlatego finalny projekt został napisany w języku C++, w którym przydzielanie i zwalnianie pamięci jest znacznie uproszczone i prostsze jest również pisanie pewnych operacji przez obiektowy charakter języka. Jednocześnie można zauważyć, że od pewnego czasu duża część popularnych mikrokontrolerów również ma wsparcie dla programowania w języku C++ od szybkich STM32 do nawet wolnych lecz tanich i popularnych mikrokontrolerów AVR. Dzięki temu implementacja wybranych algorytmów w języku C++ nie jest przeszkodą do użycia ich dalej do pracy w układach mikroprocesorowych.

Aby implementacje wybranych algorytmów można stwierdzić za udane, należałoby sprawdzić je w identyfikacji rzeczywistych układów dynamicznych. Ze względu na dużą pracochłonność w wykonaniu takiego układu jedynie w celu sprawdzenia poprawności identyfikacji, algorytmy zostały sprawdzone jedynie na zbiorach próbek otrzymanych w wyniku symulacji różnego rodzaju obiektów dynamicznych w środowisku Matlab.

II. ZAGADNIENIA TEORETYCZNE

1. REKURENCYJNA METODA NAJMNIEJSZYCH KWADRATÓW

Metoda ta wywodzi się z klasycznej metody najmniejszych kwadratów, której zadaniem jest znalezienie takich parametrów liniowego modelu ARX (AutoRegressive with eXogenous input), aby błąd średniokwadratowy pomiędzy rzeczywistą odpowiedzią obiektu a odpowiedzią zidentyfikowanego modelu był jak najmniejszy. Model ten opisany jest wzorem:

$$Y = \frac{a_1 z^{-1} + a_2 z^{-2} + \cdots + a_n z^{-nA}}{1 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_n z^{-nB}} \cdot U + \frac{1}{1 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_n z^{-nB}} \cdot E$$

Gdzie *Y, U* i *E* są odpowiednio transformatami Laurenta sygnału wyjściowego, wejściowego i zakłócenia (w drodze wejścia do obiektu), a *n* jest rzędem obiektu. Dzięki takiemu zapisowi po przekształceniu równania przez pozbycie się mianownika i wykonaniu odwrotnej transformaty Laurenta (w uproszczeniu $F \cdot z^{-n} \rightarrow f_{k-n}$) otrzymujemy różnicowe równanie, które może być użyte do zasymulowania odpowiedzi danego modelu ARX.

Model można zapisać prościej jako wektor parametrów *a* i *b*:

$$\theta = [b_1 \quad b_2 \quad \ldots \quad b_{nB} \quad a_1 \quad a_2 \quad \ldots \quad a_{nA}]^T$$

Taki zapis jest przydatny przy metodzie RLS, gdyż nowy wektor parametrów obliczany jest iteracyjnie:

$$\hat{\theta}_{k+1} = \hat{\theta}_k + \frac{P_k \underline{\varphi}_k}{\lambda + \underline{\varphi}_k^T P_k \underline{\varphi}_k} \varepsilon_k$$

Gdzie:

$$\varepsilon_k = y_k - \hat{y}_k = y_k - \underline{\varphi}_k^T \cdot \underline{\hat{\theta}}_k$$

$$\varphi_k = [-y_{k-1} \quad -y_{k-2} \quad \ldots \quad -y_{k-nB} \quad u_{k-1} \quad u_{k-2} \quad \ldots \quad u_{k-nA}]^T$$

$$P_{k+1} = \frac{1}{\lambda}\left(P_k - \frac{P_k \underline{\varphi}_k \underline{\varphi}_k^T P_k}{\lambda + \underline{\varphi}_k^T P_k \underline{\varphi}_k}\right)$$

Gdzie *k* jest aktualną liczbą zebranych próbek, λ jest współczynnikiem zapominania użytecznym w przypadku niestacjonarnych oraz nieliniowych obiektów ($0 < \lambda \leq 1$). Macierz P_k, jest estymatorem macierzy kowariancji wektora θ, gdzie jej początkową wartością powinna być macierz jednostkowa, lecz w celu szybszej identyfikacji zamiast jedności na przekątnej, stosuje się pewną, dużą wartość dodatnią.

2. REKURENCYJNA METODA ZMIENNYCH INSTRUMENTALNYCH

Metoda RIV różni się od RLS tym, iż identyfikuje model ARMAX (AutoRegressive Moving Average with eXogenous input) zamiast modelu ARX, który opisany jest wzorem:

$$Y = \frac{a_1 z^{-1} + a_2 z^{-2} + \cdots + a_n z^{-nA}}{1 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_n z^{-nB}} \cdot U + \frac{c_1 z^{-1} + c_2 z^{-2} + \cdots + c_n z^{-nC}}{1 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_n z^{-nB}} \cdot E$$

Jak widać, różni się tym, iż sygnał zakłócający przed wejściem na obiekt ma swoją własną dynamikę (której nie można zidentyfikować). Metoda RIV jak i RLS zakładają, iż sygnał *e(t)* jest szumem białym o nieznanej wariancji i zerowej wartości oczekiwanej.

Wzory na identyfikację metodą RIV różnią się nieznacznie:

$$\hat{\theta}_{k+1} = \hat{\theta}_k + \frac{P_k \underline{z}_k}{\lambda + \underline{\varphi}_k^T P_k \underline{z}_k} \varepsilon_k$$

$$\varepsilon_k = y_k - \hat{y}_k = y_k - \underline{\varphi}_k^T \cdot \underline{\hat{\theta}}_k$$

$$\varphi_k = [-y_{k-1} \quad -y_{k-2} \quad \dots \quad -y_{k-nB} \quad u_{k-1} \quad u_{k-2} \quad \dots \quad u_{k-nA}]^T$$

$$P_{k+1} = \frac{1}{\lambda}\left(P_k - \frac{P_k \underline{z}_k \underline{\varphi}_k^T P_k}{\lambda + \underline{\varphi}_k^T P_k \underline{z}_k}\right)$$

Gdzie wektor $\underline{z}_k$ musi być niezależny statystycznie od wektora sygnału nieznanego zakłócenia po przejściu przez własną dynamikę. Został więc uzależniony od próbek znanego sygnału wejściowego:

$$z_k = [u_{k-1} \quad u_{k-2} \quad \dots \quad u_{k-nB} \quad u_{k-nB-1} \quad u_{k-nB-2} \quad \dots \quad u_{k-nB-nA}]^T$$

3. IDENTYFIKACJA RZĘDU OBIEKTU

Identyfikacja rzędu obiektu polega na przyjęciu pewnego zakresu rzędu, identyfikacja osobno dla każdego rzędu oraz symulacja każdego z obiektów z sygnałem wejściowym. Wskaźnikiem jakości identyfikacji mógłby być błąd średniokwadratowy, lecz okazuje się, że zwiększanie rzędu minimalizuje owy wskaźnik. Aby znaleźć kompromis pomiędzy najlepszym odwzorowaniem rzeczywistego obiektu a wielkością rzędu obiektu zidentyfikowanego, posłużono się kryterium Akaike, który dla RIV oraz RLS może być zdefiniowany:

$$AIC = lnV_N + \frac{6n}{N}$$

Gdzie V_n jest wskaźnikiem średniokwadratowym, n jest rzędem obiektu, a N jest liczbą próbek użytych do identyfikacji. Wartość pierwszego minimum lokalnego wskaźnika AIC zaczynając od najmniejszego rzędu powinna dać dobry kompromis pomiędzy jakością odwzorowania sygnału wyjściowego a rzędem obiektu.

III. BADANIA

1. TESTOWANIE ALGORYTMÓW W ŚRODOWISKU MATLAB I SIMULINK

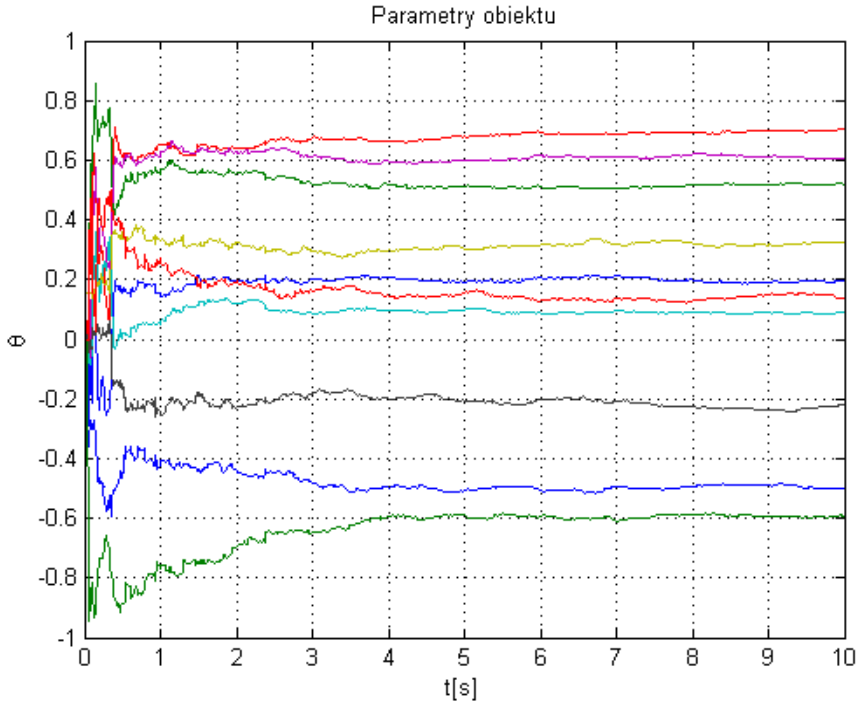
Przed implementacją algorytmów w docelowym języku należało sprawdzić końcową skuteczność działania algorytmów. Bez wchodzenia w szczegóły kodu w Matlabie podane zostały jedynie wyniki identyfikacji danymi metodami.

Zasymulowany został obiekt ARX 5-ego rzędu opisany wektorem parametrów:

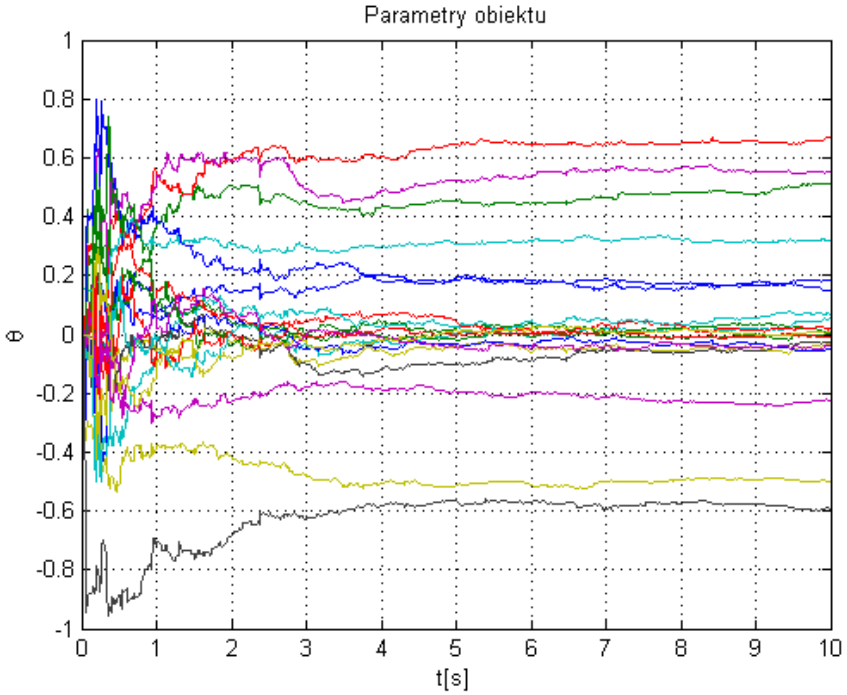
$$\theta = [0.2 \quad 0.5 \quad 0.7 \quad 0.1 \quad 0.6 \quad 0.3 \quad -0.2 \quad -0.5 \quad -0.6 \quad 0.15]^T$$

Gdzie sygnałem wymuszającym był znany co do wartości zaszumiony sygnał stały, do którego został dodany nieznany szum biały.

Z próbek wejścia i wyjścia zidentyfikowany został obiekt ARX 5-ego rzędu, którego wartości parametrów kształtowały się w czasie w sposób przedstawiony na wykresie:



Końcowe wartości są zbliżone do prawdziwych wartości parametrów oryginalnego obiektu ARX. Identyfikacja dla modelu ARX, gdzie zakładany jest rząd obiektu równy 10:



Można zauważyć, że nadmiarowe parametry identyfikowane są do wartości bliskiej zera. Metoda RIV również dla tego samego obiektu działa poprawnie. Wyniki identyfikacji dla innych obiektów zostaną przedstawione przy testowaniu zaimplementowanego już algorytmu.

2. IMPLEMENTACJA OPERACJI MACIERZOWYCH W JĘZYKU C++

Można zauważyć, że używane wzory dotyczą operacji macierzowych. Można byłoby do tego użyć gotowych bibliotek do mnożenia macierzy itp., lecz zostały one napisane od początku ze względu na ich całkowite zrozumienie i przewidywalność w działaniu oraz niewielkie skomplikowanie. Pomimo ułamków we wzorach okazuje się, że nie trzeba tworzyć funkcji odwracającej macierz (co mogłoby się skończyć problemami przy małych wyznacznikach), gdyż dzielenia dokonują się zawsze przez skalary.

Została stworzona klasa `MatrixIdent,` która reprezentuje macierz o dowolnych wymiarach. Posiada parametry mówiące o liczbie kolumn, liczbie wierszy oraz jednowymiarową tablicę wszystkich elementów tablicy. Zaimplementowane zostały standardowe konstruktory, destruktory, przeciążenia operatora przypisania. Dla uproszczenia kolejnych zapisów dobrym pomysłem okazały się przeciążenia operatorów dodawania i odejmowania od siebie macierzy, mnożenia macierzy przez siebie oraz przez skalary. Napisane zostały również metody transpozycji macierzy, skonwertowania macierzy do skalara, jeżeli jest ona wymiaru 1x1 oraz wyświetlenia macierzy w konsolowym oknie do celów testowych. Dopisana została też funkcja sprawdzająca, czy wszystkie wartości macierzy są określone (np. czy któryś z elementów przyjmuje wartość #IND itp.). Bazuje ona na znanym powszechnie sprawdzeniu warunku `if(a!=a)`… Kody nie zostały zamieszczone w tym dokumencie ze względu na to, iż nie są one skomplikowane, a możliwe do podejrzenia w projekcie końcowym.

Można zauważyć, iż macierze działają na typie danych opisanym jako `USER_DATATYPE.` Jest to dodatkowa możliwość dla użytkownika wybrania własnego typu danych używanym do obliczeń w celu np. redukcji pamięci. W tym celu należy zmienić makro preprocesora:

```
#define USER_DATATYPE long double
```

Nie jest to zbyt estetyczne rozwiązanie, ponieważ nasuwa się wprost napisanie całej klasy jako klasy szablonowej `template MatrixIdent<typename USER_DATATYPE>.` Ze względu na to, iż typ ten używany jest również w dalszych klasach służących do identyfikacji obiektu, wybrano właśnie takie rozwiązanie określenia typu danych. Wadą tego rozwiązania jest pozbycie się pewnej funkcjonalności klasy `MatrixIdent`, jeżeli okazałaby się ona przydatna w innych projektach. Poza tym mimo, iż można niektóre mikrokontrolery (np. STM32) programować w języku C++, często w trakcie kompilacji użytkownik jest powiadamiany przez kompilator, iż nie wszystkie funkcjonalności owego języka są dostępne.

3. IMPLEMENTACJA ALGORYTMU RLS I RIV W JĘZYKU C++

Za identyfikację oraz reprezentację obiektu odpowiedzialna jest klasa nazwana `SystemIdentification`. Jej deklaracja została przedstawiona w listingu poniżej:

```
class SystemIdentification {
public:
        MatrixIdent objectParam;
private:
        USER_DATATYPE* inputSamples;
        USER_DATATYPE* outputSamples;
        int samplesIterator;
        int inputSamplesIterator;
        MatrixIdent covMatrix;
        USER_DATATYPE forgettingFactor;
        int rank;
        int identMethod;
```

```
public:
        SystemIdentification();
        SystemIdentification(const SystemIdentification & mat);
        SystemIdentification(int rank, USER_DATATYPE forgettingFactor = 1,
                        USER_DATATYPE initialCovarianceMatrix = 1, int identMethod =
                                        SYSTEM_IDENTYFICATION_METHOD_RLS);
        ~SystemIdentification();

        int getRank();

        SystemIdentification& operator=(const SystemIdentification& sys);

        void identStep(USER_DATATYPE inputSample, USER_DATATYPE outputSample);
        void print();
        bool isCorrect();

private:
        void insertNewSamples(USER_DATATYPE inSample,
                        USER_DATATYPE outSammple);
        USER_DATATYPE getInputSample(int index); //
        USER_DATATYPE getOutputSample(int index);

        void RLSstep();
        void RIVstep();
};
```

Zaczynając od początku parametrów składowych klasy - macierz `objectParam` jest wprost wektorem Θ reprezentującym model dyskretny przedstawiony w opisie teoretycznym. Jego rozmiar jest zależny od rzędu obiektu (parametr `rank`) i wynosi dokładnie $2 \cdot rank\ x\ 1$. Macierz ta jest dostępna dla użytkownika.

Tablice `inputSamples` oraz `outputSamples` przechowują ostatnie próbki wejścia i wyjścia podane do identyfikacji. Liczba próbek jest zależna od metody identyfikacji (parametr `identMethod` przyjmujący wartość `SYSTEM_IDENTYFICATION_METHOD_RIV` lub `SYSTEM_IDENTYFICATION_METHOD_RLS`) co można zauważyć we wzorach podanych wcześniej:

- dla RLS dla obu tablic równa $rank + 1$
- dla RIV `inputSamples` ma rozmiar $2 \cdot rank + 1$, `outputSamples` ma rozmiar $rank + 1$

Gdy ich rozmiary są równe, parametr `samplesIterator` wskazuje indeks tablicy ostatnio dodanej próbki, gdy natomiast rozmiary nie są równe, o indeksie ostatnio dodanej próbki wejściowej mówi `inputSamplesIterator`. Macierz `covMatrix` jest macierzą kowariancji oznaczoną we wzorach jako P_k i jej rozmiar jest równy zawsze $2 \cdot rank\ x\ 2 \cdot rank$. Ostatni nieomówiony parametr `forgettingFactor`, jak wskazuje nazwa, jest współczynnikiem zapominania we wzorach oznaczonym jako λ.

Działanie konstruktorów w tym przypadku jest dosyć proste. Nadają wszystkim parametrom odpowiednie wartości, macierzom rozmiarym oraz przydzielają odpowiednią ilość pamięci dla tablic przetrzymujących wartości ostatnich próbek wejścia i wyjścia. Dla konstruktora domyślnego, przyjęty rząd obiektu jest równy 1. Konstruktor parametryczny przyjmuje jedną z wartości, której nie ma bezpośrednio przetrzymywanej w klasie, a jest nią `initialCovarianceMatrix`. We wstępie teoretycznym zostało wspomniane, iż macierz `covMatrix` na początku powinna być macierzą jednostkową, lecz dla szybszych rezultatów identyfikacji można ją przemnożyć przez pewną dużą wartość, którą jest właśnie w konstruktorze `initialCovarianceMatrix.`

Jak zostało wspomniane wcześniej, w klasie znajdują się tablice przetrzymujące ostatnie próbki wejściowe i wyjściowe. Do pobrania ostatnich próbek służą metody `getInputSample(int index)` oraz `getOutputSample(int index)`. Trudno mówić tutaj o indeksie próbki licząc od próbki

początkowej, dlatego przyjęta została konwencja, iż ostatnia dodana próbka ma indeks równy 0, próbka przedostatnia indeks -1 itd. Dzięki temu dalszy zapis algorytmu będzie zgodny z podanymi wcześniej rekurencyjnymi wzorami. Do wprowadzenia nowych próbek służy oczywiście niedostępna dla użytkownika funkcja `insertNewSamples(USER_DATATYPE,USER_DATATYPE)`.

Tak samo, jak w przypadku klasy `MatrixIdent` istnieje metoda nazwana `isCorrect()` odpowiedzialna również za sprawdzenie, czy w macierzach znajdujących się w owej klasie istnieje jakikolwiek element spełniający warunek `if(a!=a)`. Ta metoda nie jest używana nigdzie w klasie, dlatego identyfikacja odbywa się niezależnie od tego, czy któryś z identyfikowanych parametrów jest nieokreślony.

Najważniejszą metodą jest `identStep(USER_DATATYPE inputSample, USER_DATATYPE outputSample)`. Jak wynika również z nazwy, jest to kolejny krok identyfikacji dla podanych nowych wartości próbek wejściowych i wyjściowych. Dodaje nowe próbki do tablic i zależnie od typu identyfikacji wywołuje metodę `RLSstep()` lub `RIVstep()`. Metody te mają już wprost zaimplementowane wzory podane wcześniej.

Poniżej przedstawiona została metoda `RLSstep()`:

```
void SystemIdentification::RLSstep() {
	MatrixIdent fiVec(2 * rank, 1);

	int tempIterator = -1;
	for (int i = 0; i < rank; i++) {
		fiVec.insert(-1 * getOutputSample(tempIterator), i, 0);
		fiVec.insert(getInputSample(tempIterator), rank + i, 0);
		tempIterator--;
	}

	MatrixIdent fiVecT = fiVec.trans();

	//Error
	USER_DATATYPE error = getOutputSample(0)
		- (fiVecT * objectParam).toScalar();

	//Dominator
	USER_DATATYPE dominator = forgettingFactor
		+ (fiVecT * (covMatrix * fiVec)).toScalar();

	if (dominator == 0) dominator = 0.0001;

	objectParam = objectParam
		+ (covMatrix * fiVec * (error / dominator));

	covMatrix = (covMatrix)
		-((covMatrix)) * fiVec * fiVecT * (covMatrix)
		/ dominator;
}
```

Macierz `fiVec` reprezentuje wektor φ_k przedstawiony we wzorach. Dla metody RLS przetrzymuje ostatnie próbki wejściowe i zanegowane próbki wyjściowe. Wiele razy używana jest transpozycja owego wektora, więc aby nie wykonywać wiele razy tej samej operacji, nowy obiekt `fiVecT` jest równy `fiVec.trans()`. Zmienna `error` odpowiada za błąd pomiędzy rzeczywistą próbką wyjściową a próbką symulowaną przez operacje `fiVecT * objectParam`. Ponieważ wynik tej operacji jest również obiektem klasy `MatrixIdent`, należałoby napisać metodę przeciążającą operator rzutowania danej klasy do typu `USER_DATATYPE`. Niestety takie rozwiązanie okazało się niepoprawne przez dwuznaczność w kompilacji, która będzie opisane dalej, dlatego napisana została metoda transformująca macierz do skalara (gdzie tak naprawdę zwraca jedynie element (1,1) danej macierzy).

Jak można zauważyć w obu wzorach na nowy wektor parametrów obiektów oraz macierz kowariancji, występuje identyczny mianownik, dlatego został on obliczony raz i nazwany jako `dominator`. Aby nie wystąpił problem dzielenia przez 0 można byłoby wyrzucić wyjątek przerywający obliczenia, lecz trudno stwierdzić, jak pominięcie jednej próbki mogłoby zadziałać dla identyfikacji z małym krokiem próbkowania. Dla uproszczenia, jeżeli mianownik ułamka będzie równy 0, program przypisze mu wartość 0.0001. Nie jest to zbyt precyzyjne, lecz dla rzeczywistych obiektów, gdzie istnieje duża ilość szumów istnieje małe prawdopodobieństwo, że właśnie ta zmienna przyjmie wartość równą 0.

Obliczenia w dwóch ostatnich linijkach kodów są wprost wzorami podanymi wcześniej. Dzięki przeciążeniu operatorów arytmetycznych, zapisanie wzorów jest wystarczająco przejrzyste.

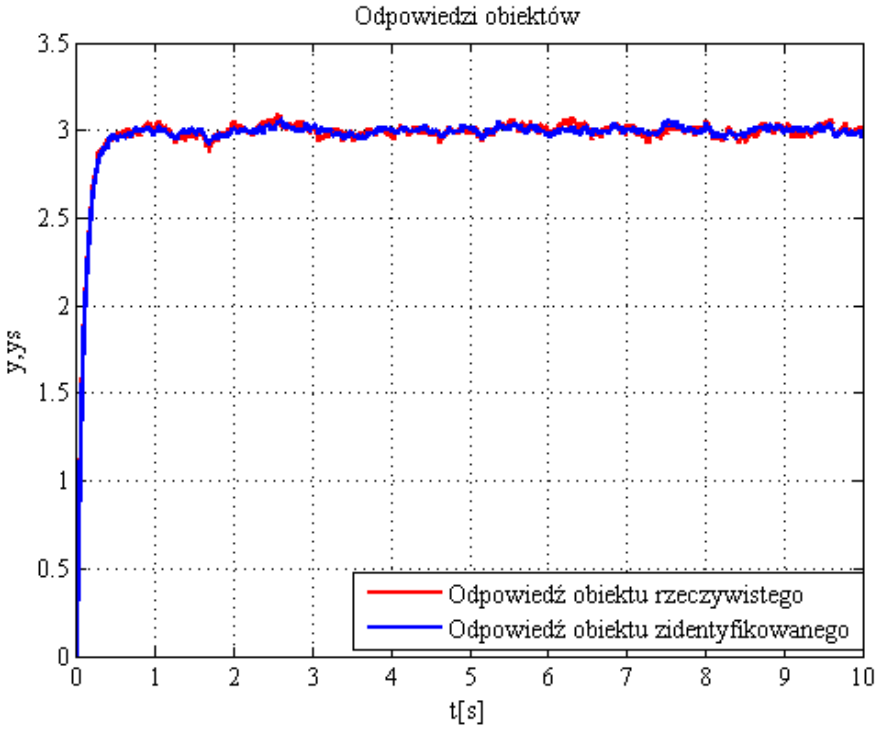
Nawiązując do wcześniejszego problemu w dwuznaczności w kompilacji, miała ona miejsce przy dzieleniu macierzy przez skalar (`(covMatrix) / dominator`). Przeciążenie operatora rzutowania na typ `USER_DATATYPE` wywołałoby problem w kompilacji, gdyż powyższe działanie można byłoby zinterpretować jako rzutowanie macierzy na typ `USER_DATATYPE` lub wykonanie przeciążonego operatora dzielenia (lub mnożenia, problem ten nie znika) macierzy przez typ zmiennoprzecinkowy. Jest to wyjaśnienie istnienia metody `toScalar()` rozwiązującej ten problem.

Kod metody `RIVstep()` jest bardzo podobny, dlatego nie ma potrzeby zamieszczania jego listingu. Jedyną różnicą jest pojawienie się wektora z_k, który jest podobnej budowy do wektora φ_k, lecz zależy tylko od próbek wejściowych. Patrząc na podane wcześniej wzory można dojść do wniosku, że należy zapamiętać dwukrotnie więcej próbek wejściowych dla danej metody, co również jest wyjaśnieniem istnienia osobnego iteratora dla tablicy (większych rozmiarów) przetrzymujących owe próbki.

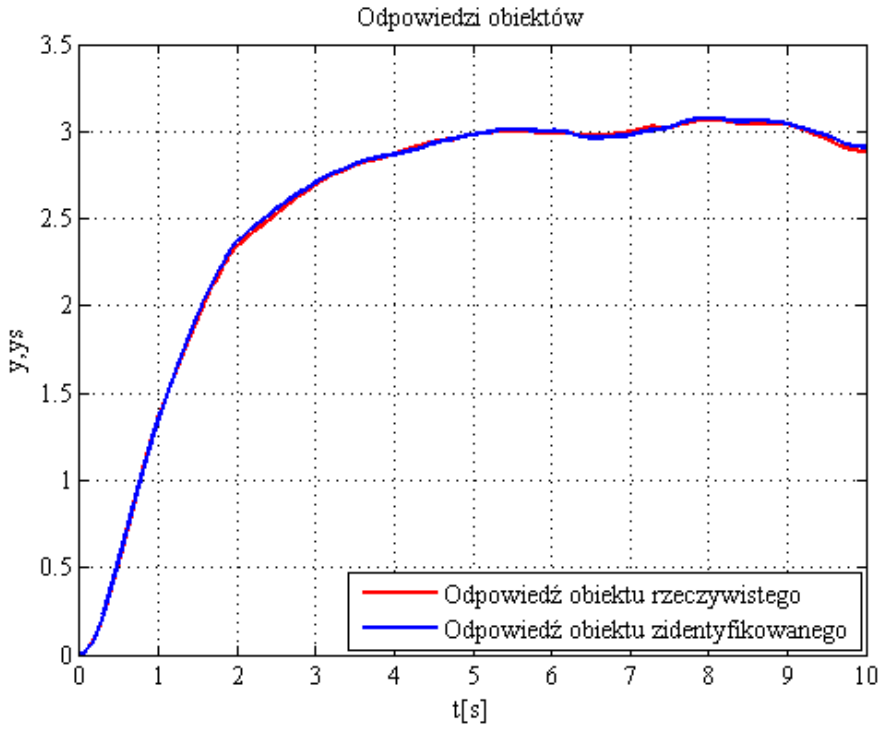
4. TESTOWANIE POPRAWNOŚCI DZIAŁANIA ALGORYTMÓW NA PRZYKŁADOWYCH DANYCH SYMULACYJNYCH

W tym rozdziale zostały zamieszczone opisy przykładowych obiektów (wzory lub schematy), ich odpowiedzi na sygnały zadane oraz odpowiedzi obiektów pochodzących z identyfikacji zaimplementowanymi metodami (zapis z Matlaba do pliku CSV, odczytanie programu w C++ próbek z pliku, a następnie identyfikacja).

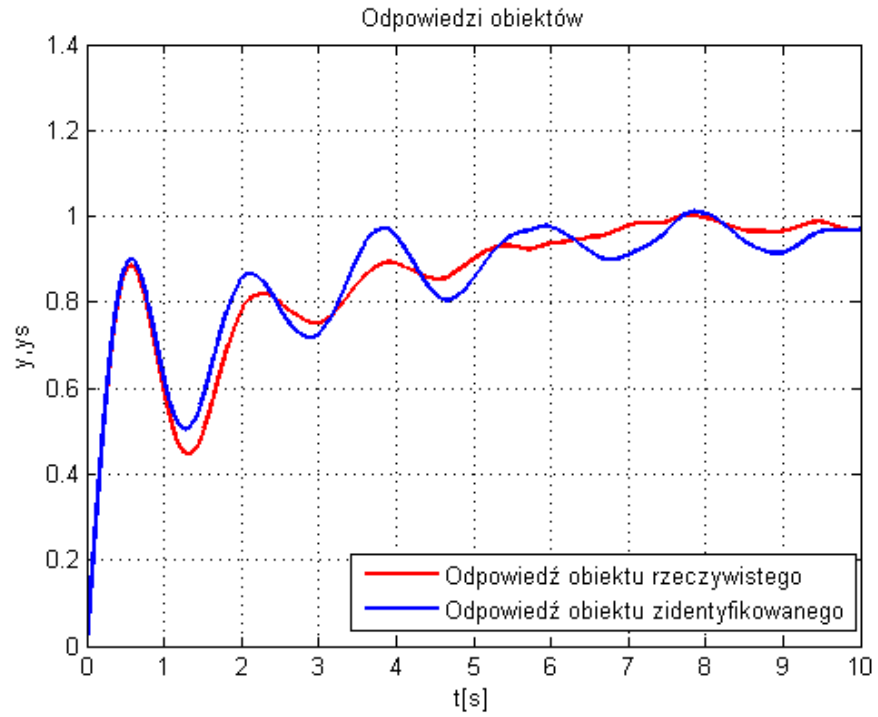
Obiekt inercyjny z zaszumieniem sygnałem skoku i dodatkowym zaszumieniem w drodze wejścia:



To samo dla obiektu dwuinercyjnego:



Obiekt nieliniowy – idealne wahadło matematyczne z regulatorem PID oraz dodatkowym szumem w drodze sygnału sterującego:



Powyższe identyfikacje dotyczyły metody RLS, dużej wartości `initialCovarianceMatrix` oraz nadmiarowego zakładanego rzędu. Pomimo tego, iż metoda RIV powinna być lepsza, czasem zidentyfikowane obiekty okazywały się niestabilne przez zbyt dużą wartość `initialCovarianceMatrix`. Analiza działania metody została przedstawiona razem z identyfikacją rzędu w dalszej części.

5. OPRACOWANIE DZIAŁANIA I IMPLEMENTACJA ALGORYTMU IDENTYFIKACJI RZĘDU OBIEKTU

Teoretyczne działanie algorytmu zostało przedstawione wcześniej, lecz należałoby przejść do jego implementacji. W tym celu została stworzona klasa `RankIdent` odpowiedzialna za identyfikację rzędu oraz jednocześnie identyfikację parametrów obiektu. Idea opiera się na przyjęciu maksymalnego zakresu rzędu obiektu, identyfikacji parametrów każdego rzędu poniżej, symulacja każdego obiektu z rzeczywistymi próbkami wyjściowymi oraz wyliczenie wskaźnika z kryterium Akaike. Aby zminimalizować nakład obliczeń, po podaniu wartości próbek wejścia i wyjścia zostanie wykonana identyfikacja dla każdego rzędu, a inna metoda udostępniona dla użytkownika wykona resztę czasochłonnych działań w celu znalezienia najlepszego rzędu obiektu.

Deklaracja klasy `RankIdent` została przedstawiona poniżej:

```
class RankIdent {
private:
        SystemIdentification* systemTab;
        USER_DATATYPE* outputSamples;
        USER_DATATYPE* inputSamples;
        int maxRank;
        int identMethod;
        bool finished;
        int currentIndex;
        int maxSamplesNumber;
        USER_DATATYPE forgettingFactor;
        int initialCovarianceMatrix;
public:
        RankIdent();
        RankIdent(const RankIdent&);
        RankIdent(int maxRank = 5, USER_DATATYPE forgettingFactor = 1,
                USER_DATATYPE initialCovarianceMatrix = 1,
                int identMethod = SYSTEM_IDENTYFICATION_METHOD_RLS, int maxSamples = 1000);
        ~RankIdent();

        bool isFinished();
        SystemIdentification getBestSystem();
        bool identStep(USER_DATATYPE inputSample, USER_DATATYPE outputSample);
        void print();
        void printAllAIC();
        bool isCorrect();

public:
        USER_DATATYPE getAIC(int rank);
        USER_DATATYPE getInputSample(int index);
        USER_DATATYPE getOutputSample(int index);
};
```

Zgodnie z opisem istnieje tablica reprezentująca poszczególne modele obiektu o kolejnych rzędach (`*systemTab`). Tak jak poprzednio istnieją tablice przechowujące próbki wejściowe i wyjściowe, lecz żadne z nich w tym przypadku nie zostają zapominane. W tym celu należałoby lepiej stworzyć listę, która w przypadku dokładania nowych elementów jest mniej problematyczna niż tablica, lecz ze względu na implementację na mikrokontrolerach, autor postanowił nie używać bibliotek STL, a przyjąć pewną maksymalną liczbę próbek (`maxSamplesNumber`), po której dalsza identyfikacja rzędu nie będzie wykonywana. Stan maksymalnego wypełnienia próbkami przechowywany jest we fladze `finished` oraz zwracany po wykonaniu metody `identStep(USER_DATATYPE, USER_DATATYPE)`.

Poniżej znajduje się listing najważniejszej metody `getAIC(int rank)`:

```
USER_DATATYPE RankIdent::getAIC(int rank) {
        if (rank < 0 || rank > this->maxRank)
                return 0;

        //Simulation
        int outLength = this->currentIndex;
        USER_DATATYPE* simulatedOutput = new USER_DATATYPE[outLength];
        for (int i = 0; i < outLength; i++)
                simulatedOutput[i] = 0;

        for (int outIterator = 0; outIterator < outLength; outIterator++) {
                for (int currOut = -1; currOut >= (-1 * rank); currOut--) {
                        if ((outIterator + currOut) >= 0) {
                                simulatedOutput[outIterator] += -1
                                        * simulatedOutput[outIterator + currOut]
                                        * this->systemTab[rank - 1].objectParam.elementAt(
                                        -1 * currOut - 1, 0);
                        }
                }

                for (int currIn = -1; currIn >= (-1 * rank); currIn--) {
                        simulatedOutput[outIterator] += this->getInputSample(
                                outIterator + currIn)
                                * this->systemTab[rank - 1].objectParam.elementAt(
                                -1 * currIn - 1 + rank, 0);
                }

        }




        //Calculating Mean Square Error
        USER_DATATYPE mse = 0;
        for (int i = 0; i < outLength; i++) {
                simulatedOutput[i] -= this->getOutputSample(i);
                mse += simulatedOutput[i] * simulatedOutput[i];
        }
        delete[] simulatedOutput;

        //Calculating Akaike's Information Criterion
        return log((double)mse) + 6 * rank / outLength;
}
```

Na początku należy wyznaczyć wektor symulowanych próbek wyjściowych. Długość jego jest równa indeksowi ostatnio zebranych próbek wejściowych i wyjściowych. Zewnętrzna pętla `for` oblicza próbkę o indeksie `outIterator`. Na nią składają się poprzednie próbki wejściowe i wyjściowe. Pierwsza wewnętrzna pętla sumuje według rekurencyjnego opisu obiektu zanegowane (przeniesienie we wzorze próbek wyjściowych na drugą stronę) zasymulowane próbki wyjściowe przemnożone przez odpowiadający im w danym kroku parametr z wektora opisującego obiekt pod warunkiem, że indeks nie jest mniejszy od zera. Następna pętla robi dokładnie to samo dla niezanegowanych rzeczywistych próbek wejściowych. Mając wektor odpowiedzi rzeczywistego obiektu oraz jego przybliżonego modelu, można obliczyć wskaźnik średniokwadratowy, a następnie wprost ze wzoru wskaźnik z kryterium Akaike.

Aby znaleźć optymalny rząd obiektu została stworzona metoda znajdująca pierwsze minimum lokalne wskaźnika AIC spośród wszystkich rzędów zaczynając od pierwszego do maksymalnego wybranego przez użytkownika. Metoda zwraca obiekt klasy `SystemIdentification`.

Odnosząc się do wszystkich deklaracji klas, nie zostały opisane metody `print`. Były one pomocnicze przy tworzeniu projektu. Ich zadaniem jest jedynie proste wyświetlanie macierzy oraz parametrów obiektów w konsoli.

6. TESTOWANIE ALGORYTMU IDENTYFIKACJI RZĘDU OBIEKTU

W poniższych przykładach zostały pokazane działania zaimplementowanego algorytmu identyfikacji rzędu:

a) RLS

Identyfikacja obiektu inercyjnego, tabela współczynników AIC:

Rząd	AIC
1	2.337
2	2.55995
3	2.27779
4	2.30382
5	2.1612
6	2.10679
7	2.07888
8	2.0569
9	2.0605
10	2.05776

Dla obiektu drugiego rzędu:

Rząd	AIC
1	8.21524
2	0.981347
3	1.22842
4	1.14037
5	1.28435
6	1.23505
7	1.25259
8	1.23794
9	1.08239
10	1.00202

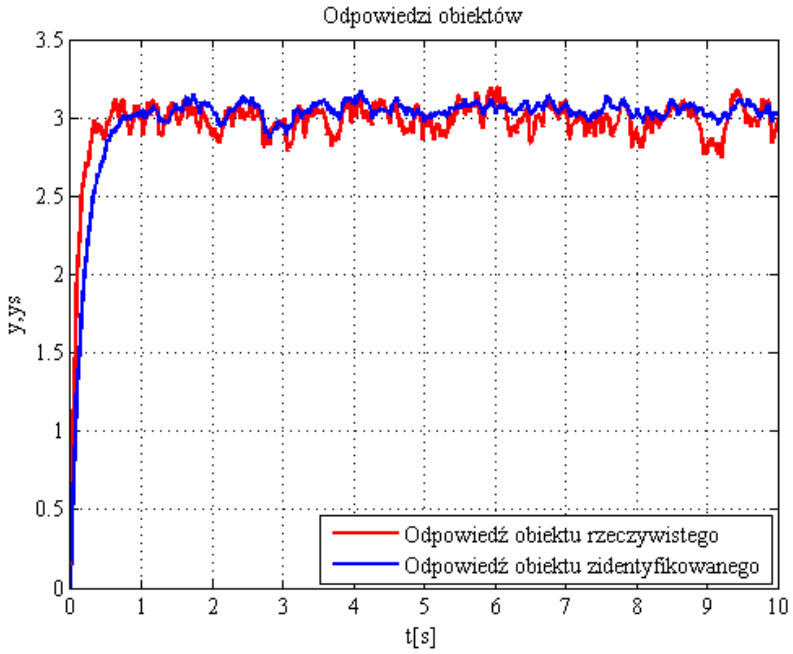
Dla obiektu nieliniowego – idealnego wahadła z regulatorem PID:

Rząd	AIC
1	6.51196
2	0.403621
3	0.539025
4	0.619376
5	0.581156
6	1.23569
7	0.535564
8	2.69093
9	3.05346
10	4.11287

b) RIV

Obiekt inercyjny – tabela AIC i symulacja:

Rząd	AIC
1	2.90198
2	-1.#IND
3	1.#INF
4	1.#INF
5	-1.#IND
6	239.652
7	-1.#IND
8	1.#INF
9	616.391
10	-1.#IND



Reszta obiektów jest identyfikowana na podobnym poziomie, jak w metodzie RLS, dlatego nie zostały zamieszczone wyniki. Można zobaczyć, że dla większych rzędów wskaźnik przybiera wartości nieokreślone. Po analizie okazało się, że owe obiekty były niestabilne.

IV. PODSUMOWANIE I WNIOSKI

1. WNIOSKI

Identyfikacja odbywa się na bardzo dobrym poziomie dla obiektów liniowych jak widać na przedstawionych wykresach symulacji. Wynik identyfikacji jest obiektem liniowym, dlatego identyfikacja wybranego przez autora nieliniowego układu regulacji kąta idealnego wahadła jest jedynie jego aproksymacją liniową. Trudno określić wynik identyfikacji, ponieważ należałoby do zidentyfikowanego obiektu dobrać regulator a następnie sprawdzić jakość regulacji rzeczywistego obiektu. Dalsza synteza regulatora za pomocą owej biblioteki oraz implementacja mikroprocesorowego układu samostrojenia się jest dobrym tematem do dalszej rozbudowy projektu

W projekcie istnieją rzeczy, które można byłoby poprawić pod względem programistycznym. Jedną z nich jest wspomniana wcześniej budowa szablonów klas zamiast mało estetycznego zmieniania dyrektyw preprocesora. Zmienić można byłoby również organizację klas - można zauważyć, iż dla identyfikacji rzędu próbki są dublowane w pamięci dla każdego obiektu `SystemIdentification` oraz dodatkowo w tablicy klasy `RankIdent`. Pozostaje również poprawa optymalizacji działania, ponieważ dla 10 000 próbek oraz identyfikacji rzędu od 1 do 30 (co nie ma większego sensu, gdyż jak widać nawet obiekt nieliniowy okazał się być aproksymowany do obiektu liniowego drugiego rzędu), cały proces trwał nawet kilkanaście minut na komputerze z procesorem o częstotliwości pracy 2,4 GHz.

V. BIBLIOGRAFIA

1. **Atollic.** ARM Development Tools IDE | TrueStudio from Atollic. [Online] [Zacytowano: 02 05 2016.] http://timor.atollic.com/.
2. **Orwell.** Dev-C++ Blog. [Online] [Zacytowano: 02 05 2016.] http://orwelldevcpp.blogspot.com/.
3. **Microsoft Corporation.** Visual Studio — Microsoft Developer Tools. [Online] [Zacytowano: 07 05 2016.] https://www.visualstudio.com/.
4. **The MathWorks.** MATLAB - MathWorks. *MathWorks – Makers of MATLAB and Simulink.* [Online] [Zacytowano: 02 05 2016.]
5. **Łuczak Dominik.** *Szablon sprawozdania.* [Online] 2014. www.zsep.cie.put.poznan.pl.
6. **The MathWorks.** System Identification Toolbox – MATLAB. *MathWorks – Makers of MATLAB and Simulink.* [Online] [Zacytowano: 07 05 2016.] http://www.mathworks.com/products/sysid/.