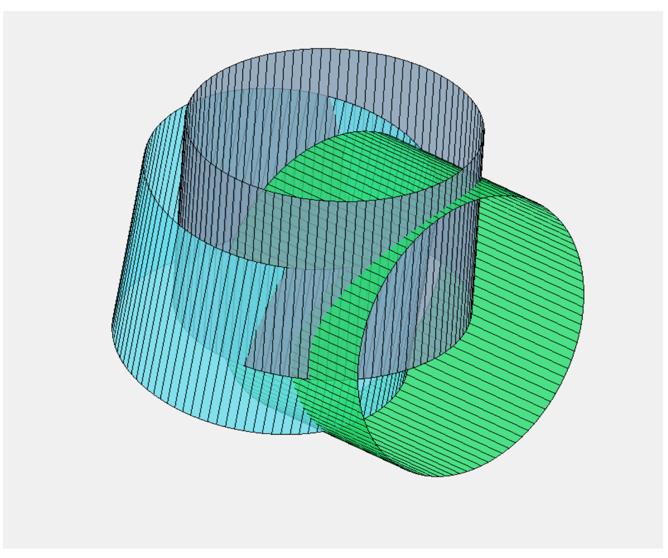
MATLAB

Podstawy użytkowania z przykładami

Materiały dydaktyczne

Tomasz Bajorek



POLITECHNIKA RZESZOWSKA

im. Ignacego Łukasiewicza

Wydział Budowy Maszyn i Lotnictwa

Rzeszów 2020

Spis treści

Wprowadzenie ... 9
1. Środowisko programu ... 11
  1.1. Menu ... 12
  1.2. Tryby pracy w środowisku *MATLAB-a* ... 13
    1.2.1. Tryb interakcyjny ... 14
    1.2.2. Tryb wsadowy ... 14
    1.2.3. Tryb symulacyjny ... 14
    1.2.4. Tryb projektowania aplikacji wizualnej *GUI* ... 14
  1.3. Narzędzia pomocy ... 14
  1.4. Przydatne polecenia systemu plików ... 15
2. Podstawowe elementy języka ... 16
  2.1. Znaki używane w języku *MATLAB-a* ... 16
  2.2. Typy danych ... 17
  2.3. Słowa kluczowe ... 18
  2.4. Stałe ... 18
    2.4.1. Stałe liczbowe ... 18
    2.4.2. Stałe tekstowe ... 19
    2.4.3. Stałe logiczne ... 20
  2.5. Zmienne ... 20
3. Podstawy użytkowania ... 21
  3.1. Praca w trybie interakcyjnym ... 21
  3.2. Instrukcja przypisania ... 22
  3.3. Wyrażenia obliczeniowe ... 27
    3.3.1. Proste wyrażenia z wykorzystaniem operatorów arytmetycznych ... 27
    3.3.2. Standardowe funkcje arytmetyczne ... 29
    3.3.3. Przykłady poprawnych wyrażeń obliczeniowych ... 31
    3.3.4. Zaokrąglenia liczb dziesiętnych ... 32
    3.3.5. Wyrażenia logiczne ... 33
4. Tworzenie i wykonywanie skryptów (*M-plików*) ... 35
  4.1. Wymagania i uwagi praktyczne ... 35

4.2. Kontrola błędów 36
4.3. Pierwszy skrypt 37
4.4. Komentarze w skrypcie 38
5. Instrukcje wejścia i wyjścia 40
5.1. Interakcyjne wprowadzanie danych - instrukcja wejścia 40
5.2. Wyprowadzanie tekstów informacyjnych i formatowanie wyników - instrukcje wyjścia 40
6. Macierze 43
6.1. Generowanie stałej macierzowej 43
6.2. Dostęp do elementu macierzy i fragmentu macierzy 47
6.3. Operacje macierzowe i tablicowe 50
6.3.1. Łączenie macierzy 50
6.3.2. Suma macierzy 50
6.3.3. Iloczyn macierzy 52
6.3.4. Potęgowanie macierzy 54
6.3.5. Dzielenie macierzy 55
6.3.6. Macierz transponowana i sprzężona 57
6.3.7. Wyrażenia logiczne wykonywane na macierzach 57
6.3.8. Funkcje działające na macierzach liczbowych 58
6.3.9. Przykłady zastosowań działań macierzowych 63
6.3.10. Macierze wielowymiarowe 66
7. Inne typy danych 68
7.1. Tablice znaków 68
7.2. Tablice komórkowe (*cell arrays*) 69
7.3. Struktury i tablice struktur 74
7.4. Tabele 75
8. Obsługa plików 80
8.1. Pliki tekstowe *ASCII* 80
8.2. *MAT*-pliki 81
8.3. Obsługa plików graficznych 82
9. Instrukcje strukturalne 83
9.1. Instrukcja warunkowa *if* 83

9.2. Instrukcja wyboru *switch* ... 87
9.3. Instrukcje iteracyjne ("pętle") ... 88
9.3.1. Pętla *for* ... 88
9.3.2. Pętla *while* ... 92
9.4. Instrukcja *try... catch...* ... 94
9.5. Wykorzystanie instrukcji strukturalnych do analizy i modyfikacji macierzy ... 95
9.5.1. Sumowanie warunkowe elementów macierzy ... 95
9.5.2. Zliczanie warunkowe elementów macierzy ... 101
9.5.3. Wyszukiwanie liczb w wektorze ... 102
10. Tworzenie wykresów ... 104
10.1. Wykresy dwuwymiarowe ... 104
10.1.1. Funkcja *plot* ... 104
10.1.2. Funkcja *fplot* ... 108
10.1.3. Inne funkcje tworzące wykresy ... 109
10.2. Wykresy trójwymiarowe ... 110
10.2.1. Krzywe w przestrzeni trójwymiarowej ... 110
10.2.2. Powierzchnie w przestrzeni trójwymiarowej ... 111
10.3. Przykładowe zadanie ... 116
11. Definiowanie funkcji przez użytkownika ... 119
11.1. Funkcja anonimowa ... 119
11.2. Funkcja w zewnętrznym *M-pliku* ... 120
12. Wykorzystanie pakietu *Symbolic Toolbox* ... 127
12.1. Możliwości pakietu ... 127
12.2. Deklaracja zmiennych symbolicznych ... 127
12.3. Tworzenie funkcji i macierzy symbolicznych ... 127
12.4. Podstawienia wartości liczbowych do wyrażeń symbolicznych ... 128
12.5. Operacje na macierzach o elementach symbolicznych ... 129
12.6. Symboliczne rozwiązywanie równań ... 130
12.7. Obliczanie pochodnych ... 135
12.8. Obliczanie całek nieoznaczonych i oznaczonych ... 136
12.9. Granice ciągów i funkcji ... 137
12.10. Wykresy funkcji symbolicznych ... 137

12.11. Równania różniczkowe 139
12.12. Przykładowe zastosowania działań symbolicznych 143
12.12.1. Badanie przebiegu funkcji 143
12.12.2. Obliczanie pól powierzchni 146
12.12.3. Obliczanie długości krzywych 148
12.12.4. Obliczenia objętości 149
13. Zastosowania *MATLAB-a* w statystyce 151
13.1. Zestawienie użytecznych funkcji 151
13.2. Suma i średnia arytmetyczna 152
13.3. Sortowanie zbioru 152
13.4. Mediana 153
13.5. Wariancja i odchylenie standardowe 154
13.6. Krzywa *Gaussa* 155
13.7. Operacje na zbiorach 157
14. Wybrane problemy obliczeniowe 158
14.1. Oscylator harmoniczny z tłumieniem 158
14.2. Analiza rzutu ukośnego 159
14.3. Zagadnienia kinematyki 162
14.4. System ze sprzężeniem zwrotnym 176
14.5. Metoda *Monte Carlo* 177
14.5.1. Wyznaczanie przybliżonej wartości liczby π 177
14.5.2. Wyznaczanie przybliżonej wartości całki oznaczonej 178
14.6. Transformacje geometryczne 179
15. Aproksymacja i interpolacja 182
15.1. Aproksymacja 182
15.2. Interpolacja 183
16. Podstawowe zasady korzystania z pakietu *Simulink* 187
17. Programowanie obiektowe 191
17.1. Zasady ogólne 191
17.2. Definicja klasy 191
17.3. Przykładowe programy zarządzania obiektami 193

18. Tworzenie interfejsu graficznego *GUI* 199
18.1. Inicjacja narzędzia *GUI* (*Graphic User Interface*) 199
18.2. Projektowanie aplikacji 201
18.3. Dodanie funkcjonalności do aplikacji 201
18.4. Pierwsza aplikacja *GUI* 202
18.5. Dostęp do właściwości komponentów w kodzie aplikacji 204
18.6. Umieszczanie własnych zmiennych w strukturze *handles* 205
18.7. Przykłady aplikacji *GUI* 206
19. Przykład prostej aplikacji bazy danych 213
20. Zadania 218
Bibliografia 230
Alfabetyczny skorowidz podstawowych funkcji w *MATLAB-ie* 231

Wprowadzenie

MATLAB© jest interakcyjnym środowiskiem, służącym do wykonywania obliczeń oraz analiz naukowych i inżynierskich.

Nazwa programu pochodzi od *MATrix LABoratory* (*laboratorium macierzowe*). Autorem programu jest firma *Mathworks Inc.* (*www.mathworks.com*).

Instalacja *MATLAB-a* i jego rozszerzeń wymaga licencji - *Politechnika Rzeszowska* posiada licencję akademicką dla pracowników i studentów uczelni. Informacje na stronie internetowej:

https://matlab.prz.edu.pl

wspomagają pobranie i instalację programu. Proces ten wymaga rejestracji i aktywacji konta użytkownika, przy pomocy uczelnianego konta pocztowego (w domenie *prz.edu.pl*) na stronie internetowej producenta pod adresem:

mathworks.com/mwaccount/register

MATLAB można określić mianem języka programowania wysokiego poziomu. Głównym typem danych w języku jest macierz zawierająca elementy składowe (liczby, znaki, struktury). W podstawowej wersji *MATLAB-a* użytkownik ma możliwość wykonywania złożonych operacji macierzowych i tablicowych, środowisko implementuje wiele algorytmów matematycznych, posiada też zaawansowaną obsługę grafiki.

Zasadniczą zaletą *MATLAB-a* jest możliwość szybkiego uzyskania rezultatów analiz i obliczeń oraz ich przedstawienia w postaci wykresów różnych typów, dwu- lub trójwymiarowych. Jest to znakomite narzędzie do szybkiej analizy i wizualizacji danych.

MATLAB umożliwia pracę w trybie interakcyjnym, lecz szybkość i efektywność pracy w *MATLAB-ie* wzrasta po utworzeniu zoptymalizowanych *M-plików* (lub *Mex-plików*), skryptów w formie tekstu nieformatowanego, które zawierają zestawy instrukcji i mogą być wykonane w całości. Skrypty, które łączą kod, dane wejściowe i wyjściowe, są tworzone wykonywalnym edytorze. *M-pliki* umożliwiają opracowanie własnych programów do realizacji prostych i złożonych algorytmów obliczeniowych użytkownika.

W szerszych wersjach *MATLAB* posiada wiele pakietów narzędziowych, umożliwiających realizację wyspecjalizowanych algorytmów. W *MATLAB-ie* możliwe jest również programowanie zorientowane obiektowo (klasy, metody, klasy abstrakcyjne, dziedziczenie) oraz tworzenie aplikacji z interfejsem graficznym, sterowanych zdarzeniami.

Architektura pakietu posiada poniższe cechy:

- język *MATLAB-a* i jego funkcje zewnętrzne są zawarte w wielu katalogach instalacyjnych programu,
- środowisko użytkownika (ang. *working environment*) zawiera zestaw narzędzi ułatwiających korzystanie z aplikacji, jest wyposażone w okna graficzne dostosowane do specyfiki pracy,
- obiektowo zorientowana grafika, która służy do wizualizacji danych i wyników obliczeń oraz umożliwia edycję obrazów, animację i efekty dźwiękowe;
- elementem składowym *MATLABA-a* jest *GUI* (ang. *Graphics User Interface*) - możliwość tworzenia interfejsu graficznego użytkownika, który daje możliwość pracy interakcyjnej z wykorzystaniem okienek edycyjnych i elementów sterujących (przycisków, suwaków, menu itp.),

- biblioteka matematyczna w wersji podstawowej oraz biblioteki wymagające osobnej licencji składają się z *M-plików* umieszczonych w wielu katalogach,
- *toolboxy*, czyli dodatkowe biblioteki, to około 20 wyspecjalizowanych pakietów oprogramowania, które umożliwiają analizę i symulację cyfrową zawansowanych problemów inżynierskich i naukowych,
- dzięki współpracy społeczności sieciowej możliwa jest instalacja zewnętrznych programów i aplikacji bezpośrednio z poziomu *MATLAB*-a.

Pakiety narzędziowe (*toolboxy*) to między innymi:

- *Symbolic Toolbox* - pakiet do operacji na symbolach: umożliwia przekształcenia wyrażeń, badanie funkcji, wyznaczania pochodnych i całek, granic ciągów, tworzenie wykresów funkcji symbolicznych,
- *Simulink* - pakiet do modelowania układów dynamicznych i symulacji przebiegów sygnałów; umożliwia interaktywne tworzenie wielopoziomowych systemów, w formie schematów złożonych z funkcjonalnych bloków,
- *Optimization Toolbox* – pakiet, który umożliwia wyszukiwanie parametrów, minimalizujących lub maksymalizujących cele, przy jednoczesnym spełnieniu ograniczeń,
- *Neural Network Toolbox* - tworzenie i analiza sieci neuronowych,
- *Signal Processing Toolbox* – umożliwia przetwarzanie sygnałów, projektowanie i analizę filtrów cyfrowych, estymację widmową (analiza szybką transformacją *Fouriera - FFT*),
- *Control System Toolbox* – zawiera narzędzia analizy systemów sterowania i regulacji z pomocą przekształcenia *Laplace'a* i *Fouriera*, umożliwia uzyskaniem odpowiedzi czasowych i częstotliwościowych układów.

Program posiada obszerną i przystępnie napisaną dokumentację w języku angielskim, przykłady i system pomocy, istnieje też wiele pozycji książkowych po polsku, grupy dyskusyjne i tutoriale internetowe.

MATLAB jest wykorzystywany nie tylko w zastosowaniach związanych z techniką, ale też w ekonomii, medycynie, meteorologii i wielu innych dziedzinach. *MATLAB* jest powszechnie nauczany na świecie, gdyż jest stosunkowo łatwy do nauki i stosowania w praktyce inżynierskiej. Stał się jednym z najczęściej używanych narzędzi w badaniach naukowych. Jak podaje firma *MathWorks*, aktualnie program ma ponad 3 miliony użytkowników na świecie.

Pierwsza wersja programu *MATLAB* powstała w 1980 roku w języku *Fortran*, od 1983 roku program jest tworzony w języku *C*. Najbardziej aktualne wersje noszą identyfikatory 2019a, 2019b i 2020a. W niniejszych opisach oparto się głównie na najnowszej wersji, z uwzględnieniem uwag na temat znaczących różnic w stosunku do poprzednich wydań.

Celem niniejszego opracowania jest dostarczenie studentom *Wydziału Budowy Maszyn i Lotnictwa Politechniki* Rzeszowskiej wiedzy na temat podstaw pracy w środowisku nowoczesnej platformy obliczeniowej i analitycznej, jaką niewątpliwie jest *MATLAB*.

Pozycja nie ma charakteru monografii, dzięki poszerzonym informacjom i przykładom może dać jedynie inspirację do dalszego rozwijania własnych umiejętności w zakresie tworzenia programów komputerowych, realizujących analizy i obliczenia inżynierskie z wykorzystaniem zaawansowanej platformy. Podręcznik może być wykorzystywany w wybranym zakresie na zajęciach laboratoryjnych z przedmiotów: "Informatyka" i "Metody obliczeniowe i podstawy programowania" i innych.

1. Środowisko programu

Po uruchomieniu *MATLAB-a* użytkownik ma do dyspozycji zestaw wewnętrznych okien. Są to:

Command Window - okno zwierające wiersz poleceń, służy do interakcyjnego dialogu, pisania i wykonywania pojedynczych poleceń programowych i systemowych oraz wyświetlania tekstowych rezultatów działań użytkownika,

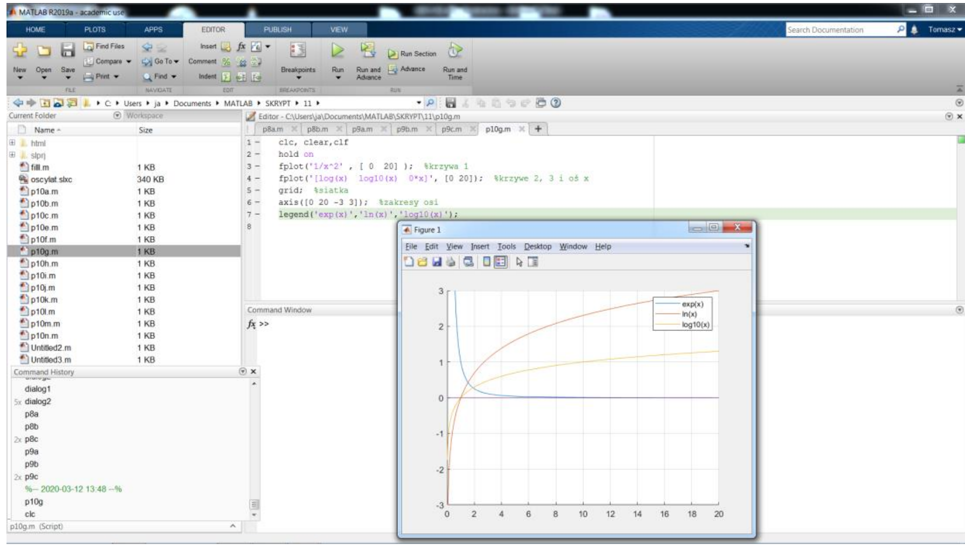
Workspace - obszar roboczy, w którym znajduje się lista zmiennych utworzonych przez użytkownika lub przez program, elementów przechowujących dane,

Current Folder (*Current Directory*) - lista plików i folderów katalogu bieżącego, roboczego katalogu, w którym użytkownik przechowuje pliki skryptów (tekstów programów źródłowych w języku *MATLAB-a*), pliki z danymi do programów i pliki, w których zapisywane są wyniki obliczeń,

Editor - okno edytora plików tekstowych, które pojawia się w trakcie opracowywania bądź modyfikacji skryptu, czyli pliku z ciągiem poleceń realizujących algorytm obliczeniowy,

Command History - okno ukazujące historię instrukcji (poleceń) wykonanych przez użytkownika w wierszu poleceń *Command Window*,

Figure - okno ukazujące się po wykonaniu instrukcji wykonującej utworzenie wykresu; może również służyć do prezentacji obrazów różnych formatów.



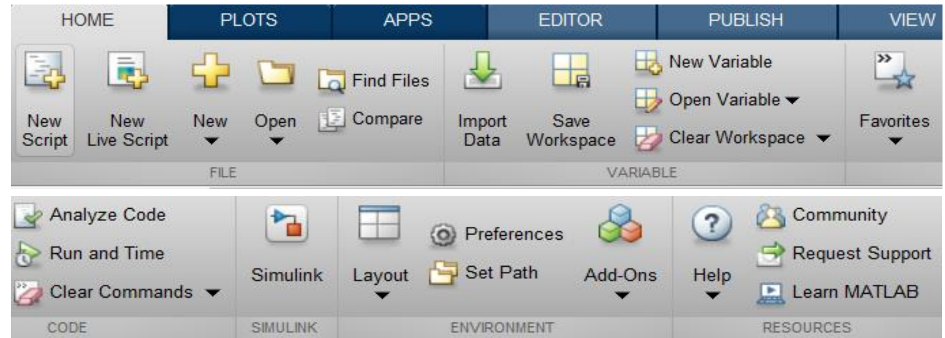
Rys.1.1. Przykładowy układ okien *MATLAB-a* w najnowszych edycjach programu

Można w łatwy sposób zarządzać zestawem okien (ich widocznością i rozmieszczeniem), korzystając z menu *Home/Layout* (w starszych wersjach *MATLAB-a* menu: *Desktop/Desktop Layout*). Można ustalać parametry każdego okna z osobna, wybrać projekt domyślny (*default*), lub zapisać własny projekt okien, dopasowany do potrzeb użytkownika.

1.1. Menu

Menu *MATLAB-a* posiada kilka wstęg (*ribbons*):

Home (rys.1.2) - wstęga zawiera pozycje do tworzenie nowych i otwierania istniejących plików skryptowych, zarządzania obszarem roboczym, analizy kodu, ustawiania zestawu okien wewnętrznych programu (*Layout*), dostępu do pomocy i inne.

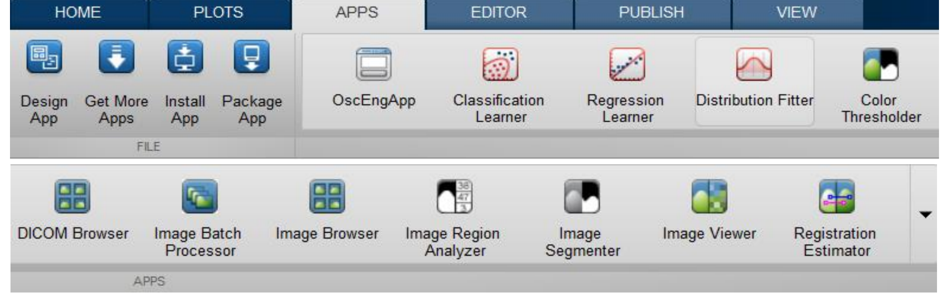


Rys.1.2. Menu - wstęga *Home*

Plots (rys.1.3) - to zestaw opcji zarządzania wykresami, po wykonaniu programu można wskazać w menu zmienną macierzową, dla której ma być utworzony wykres, a także typ wykresu.

Rys.1.3. Menu - wstęga *Plots*

Apps (rys.1.4) - opcje dostępne w tym zestawie umożliwiają pobieranie i instalację aplikacji celowych i obsługę aplikacji wbudowanych (np. zarządzania obrazami i wideo),

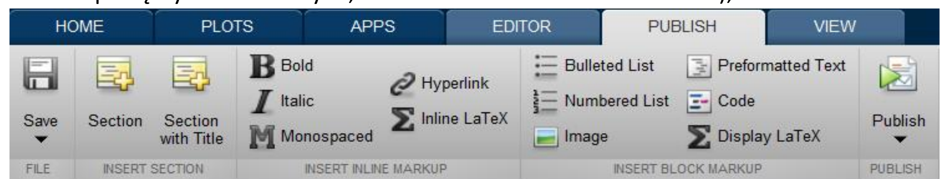


Rys.1.4. Menu - wstęga *Apps*

Editor (rys.1.5) - wstęga pojawia się w trakcie opracowywania skryptu - tekstowego pliku programu; zawiera opcje zarządzania plikami, uruchamiania programu i mechanizmy kontroli jego działania,

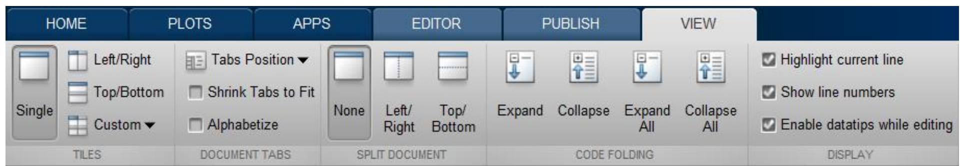
Rys.1.5. Menu - wstęga *Editor*

Publish (rys.1.6) - opcje dla celów publikacji (np. wspomaganie tworzenia dokumentów *HTML* i hiperłączy internetowych, format *LateX* dokumentów i inne),



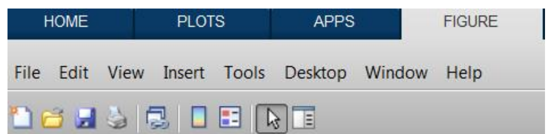
Rys.1.6. Menu - wstęga *Publish*

View (rys.1.7) - opcje widoku okien dla wielu plików edytora, numeracja wierszy, podświetlanie linii, podpowiedzi ekranowe itp..



Rys.1.7. Menu *MATLAB-a* - wstęga *View*

Figure (rys.1.8) - wstęga pojawia się po zadokowaniu okna *Figure*; menu okna zwiera wiele dodatkowych opcji zarządzania wykresami i obrazami, narzędzia obrotu i skalowania, dostępu do właściwości elementów wykresu: kształtów, krzywych, osi itp.



Rys.1.8. Menu *MATLAB-a* - wstęga *Figure*

Warto poświęcić trochę czasu na zaznajomienie się z interfejsem *MATLAB-a*, by sprawnie korzystać z możliwości środowiska.

1.2. Tryby pracy w środowisku *MATLAB-a*

W *MATLAB-ie* możliwe są podstawowe tryby pracy: **interakcyjny** i **wsadowy** oraz dodatkowe tryby specjalistyczne, wykorzystywane do modelowania wizualnego i tworzenia aplikacji komponentowych.

1.2.1. Tryb interakcyjny

W tym trybie, pojedyncze polecenie (instrukcja) jest wpisywane przez użytkownika w wierszu polecenia okna *Command Window*, po znaku zachęty **>>**:

>> *treść polecenia*

i zatwierdzane jest do wykonania klawiszem *Enter*.

1.2.2.Tryb wsadowy

W trybie wsadowym w oknie *Editor* są opracowywane przez użytkownika pliki tekstowe (skrypty - tzw. *M-pliki*, koniecznie z rozszerzeniem **.m)**, zawierające zestaw poprawnych składniowo i znaczeniowo instrukcji *MATLAB-a*, które można wykonać w ciągłej sekwencji. *M-pliki* mogą także zawierać bloki działań, wykonywanych w aplikacjach obiektowych w reakcji na określone zdarzenia.

Zapis ciągu poleceń jako prostych programów, składających się z kilku instrukcji, a także bardziej złożonych realizacji skomplikowanych algorytmów, dzięki możliwości edycji, kontroli błędów, wielokrotnemu testowaniu z różnymi parametrami wejściowymi, stanowi ułatwienie pracy w środowisku *MATLAB-a.* Podstawy korzystania z edytora *M-plików* i zasad pracy wsadowej zostaną przedstawione w rozdz. 4.

1.2.3. Tryb symulacyjny

Oprócz wymienionych wyżej, podstawowych trybów, *MATLAB* jest wyposażony w narzędzia symulacji. Istnieje możliwość wykonania modeli układów fizycznych (mechanicznych, elektronicznych i innych) oraz ich analizy przy pomocy specjalnego pakietu narzędziowego *Simulink*, w którym problem obliczeniowy przedstawiony jest w postaci schematu wizualnych bloków sygnałowych, przetwarzających i rejestrujących. *Simulink* wykorzystuje *MATLAB-a* w środowisku graficznym, gdzie na bazie modeli wykonuje się symulacje procesów i analizę systemów w czasie rzeczywistym.

1.2.4. Tryb projektowania aplikacji wizualnej *GUI*

MATLAB posiada zestaw narzędzi do projektowania formularza z wizualnymi elementami sterującymi i prezentacyjnymi (przyciski, pola wyboru, pola tekstowe, suwaki, wykresy itp.). Następnie tworzone są bloki funkcyjne dla obsługi elementów zaprojektowanego formularza, które są powiązane ze zdarzeniami dotyczącymi tych elementów, czyli działaniami wykonywanymi na elementach sterujących.

1.3. Narzędzia pomocy

Materiały w języku angielskim dostępne są z menu: *Home/Help/Documentation* lub z wykorzystaniem klawisza *F1.*

Jeżeli zostanie wskazane myszką (w *Command Window* lub *M-pliku*) słowo kluczowe lub nazwa wbudowanej funkcji *MATLAB-a*, można uzyskać pomoc kontekstową na wskazany temat, z przykładami zastosowań przy pomocy klawisza *F1* lub opcji *Help* w menu podręcznym.

Poniżej wybrane polecenia *Command Window*, wspomagające uzyskanie pomocnej informacji:

help - lista dostępnych tematów pomocy,
helpwin - pomoc w interaktywnym oknie,
helpdesk - pomoc w przeglądarce,

help *temat* - pomoc na wybrany temat.

Przykładowo, wyświetlenie pomocy na temat funkcji matematycznej *log*:

```
>> help log
log    Natural logarithm.
  log(X) is the natural logarithm of the elements of X
  Complex results are produced if X is not positive.
```

log(x) jest logarytmem naturalnym
Gdy X nie będzie dodatnie wyniki będą zespolone

W sieci można łatwo znaleźć materiały w języku polskim, tutoriale wideo oraz zadać pytanie na grupach dyskusyjnych, zajmujących się problemami pracy w *MATLAB-ie*.

1.4. Przydatne polecenia systemu plików

Dla ułatwienia pracy z katalogami i plikami (skryptami w *M-plikach*, plikami danych, odrębnymi plikami definiującymi funkcje użytkownika itp.), *MATLAB* dostarcza możliwość używania poleceń systemu plików.

Najczęściej stosowane polecenia przedstawia poniższe zestawienie:

dir lub **ls**	- wyświetla listę plików i podkatalogów katalogu roboczego,
cd *katalog*	- zmiana katalogu roboczego na podrzędny o podanej nazwie,
cd ..	- zmiana katalogu roboczego na nadrzędny,
delete *nazwa*	- usuwanie pliku,
copyfile *źródło cel*	- kopiowanie pliku
rmdir *nazwa1 nazwa2*	- usuwanie katalogu,
movefile *źródło cel*	- zmiana nazwy (lob przenoszenie) plików i katalogów,
mkdir *nazwa_katalogu*	- tworzenie nowego katalogu,
which *funkcja*	- ścieżka do katalogu przeszukiwań w którym znajduje się funkcja MATLAB-a.

Polecenia te mogą być wykonywane bezpośrednio w wierszu poleceń *Command Window*, jak też wykorzystywane w treści skryptów (*M-plików*).

2. Podstawowe elementy języka

2.1. Znaki używane w języku *MATLAB-a*

Poniższa tabela zestawia wyjaśnienia dotyczące kontekstu używania znaków specjalnych w *MATLAB-ie*.

Tabela 2.1. Znaki wykorzystywane w *MATLAB*-ie

Znak	Opis
[ ]	W nawiasach prostokątnych umieszcza się elementy macierzy: wartości liczbowe, wyrażenia, znaki lub symbole,
{ }	Nawiasy klamrowe są używane przy definiowaniu tzw. macierzy komórkowych oraz do określania indeksów elementu takiej macierzy,
( )	Nawiasy okrągłe są używane do: • wyszczególnienia indeksów macierzy homogenicznej, • do określenia wyższego od domyślnego priorytetu operacji w wyrażeniach obliczeniowych, • do wyszczególnienia argumentów funkcji,
:	Dwukropek jest wykorzystywany dla: • oddzielenia dwóch (a:b) lub trzech elementów (a:b:c) w definicji ciągu elementów, • określenia zakresu indeksów macierzy, na przykład X(2,:) reprezentuje wszystkie elementy drugiego wiersza macierzy, • przekształcenia macierzy wielokolumnowej w jednokolumnową, sklejając jej kolumny - należy pisać: X(:),
.	Kropka: • poprzedza część ułamkową liczby dziesiętnej - nie przecinek!, • oddziela nazwę struktury od nazwy jej pola (np. *osoba.nazwisko*),
,	Przecinek rozdziela: • indeksy macierzy wielowymiarowej, np. X(3,4) , • elementy wiersza macierzy (alternatywnie można oddzielać spacją), • argumenty funkcji, np. *power*(3, 4), *rand*(4, 7) itp., • kolejne instrukcje (zamiast zmiany wiersza),
;	Średnik oddziela: • wiersze macierzy - zamiennie ze zmianą wiersza (*Enter*), • instrukcje – blokuje tzw. "echo" instrukcji poprzedzającej średnik,
...	Trzy kropki na końcu wiersza informują o kontynuacji zapisu instrukcji w następnym wierszu,
%	Znak % poprzedza komentarz (działa do końca wiersza) - dowolny tekst wyjaśniający lub dezaktywujący instrukcję,

<table>
<tr><td>spacja</td><td>Spacja oddziela elementy wiersza macierzy (zamiennie z przecinkiem).
Spacja jest zabroniona:
• wewnątrz liczby dziesiętnej,
• wewnątrz nazw (identyfikatorów zmiennych, plików, funkcji),
Spacja jest wymagana:
• po słowach kluczowych rozpoczynających instrukcje if, for, while, switch,
W innych sytuacjach jest dopuszczalna (także ciąg spacji)</td></tr>
<tr><td>=</td><td>Operator przypisania, nadaje zmiennej wartość obliczonego wyrażenia</td></tr>
<tr><td>==</td><td>Dwuznakowy operator porównania "czy równe", łączy dwa wyrażenia - jeżeli mają tę samą wartość, zwracana jest prawda logiczna</td></tr>
<tr><td>~=</td><td>Dwuznakowy operator porównania: "czy nierówne", wynikiem jest prawda logiczna, jeżeli połączone tym operatorem wyrażenia mają różną wartość</td></tr>
<tr><td>>
<
>=
<=</td><td>Pozostałe operatory porównania (relacji), porównanie dwóch wyrażeń w aspekcie wielkości liczbowej (także kolejności alfabetycznej znaków)</td></tr>
<tr><td>~</td><td>(tylda) Operator logiczny negacji</td></tr>
<tr><td>&</td><td>(ampersand) Operator logiczny koniunkcji</td></tr>
<tr><td>|</td><td>(kreska pionowa) Operator logiczny alternatywy</td></tr>
<tr><td>Enter</td><td>• wysyła polecenie do wykonania w Command Window,
• oddziela instrukcje w M-pliku,
• oddziela wiersze macierzy - zamiennie ze znakiem średnika ;</td></tr>
</table>

2.2. Typy danych

Podstawowym elementem przechowującym dane jest macierz. Nawet pojedyncza liczba jest umieszczana w macierzy zawierającej jeden element.

Każda dana w programie jest określonego typu. Macierze mogą zawierać dane różnych typów, są to przede wszystkim:

double - liczby,
char - znaki,
string - ciągi znaków (napisy),
logical - wartości logiczne.

Należy tu także wspomnieć o typach specjalnych:

cell - komórki,
struct - struktury,
sym - typ symboliczny.

Typy te zostaną omówione w kolejnych rozdziałach.

2.3. Słowa kluczowe

Poniższa tabela zawiera kompletną listę słów kluczowych, wykorzystywanych w języku *MATLAB-a*.

Tabela 2.2. Zestawienie słów kluczowych w *MATLAB-ie*

if	rozpoczyna instrukcję warunkową
else	wykorzystywane w połączeniu ze słowem kluczowym *if*
elseif	wykorzystywane w połączeniu ze słowem kluczowym *if*
end	kończy instrukcje strukturalne, także symbolizuje ostatni element w macierzy bądź tablicy
for	rozpoczyna powtarzanie bloku instrukcji ***n*** razy
while	rozpoczyna powtarzanie bloku instrukcji, liczba powtórzeń wynika ze zdefiniowanego warunku
switch	wybór bloku instrukcji do wykonania
case	wykorzystywane w połączeniu ze słowem kluczowym *switch*
otherwise	wykorzystywane w połączeniu ze słowem kluczowym *switch*
break	zakończenie pętli *for* lub *while*
return	powrót do funkcji wywołującej
try	rozpoczyna sekwencję testowanych instrukcji pod katem błędów
catch	używane łącznie ze słowem kluczowym *try*

Słowa kluczowe nie mogą być wykorzystywane dla ustalania nazw elementów programowych, tworzonych przez użytkownika (zmiennych, plików).

Wyjaśnienia zastosowań słów kluczowych znajdą się w kolejnych rozdziałach.

2.4.Stałe

2.4.1. Stałe liczbowe

MATLAB używa standardowego zapisu liczb dziesiętnych, **z kropką jako separatorem dziesiętnym**.

W zapisie stałoprzecinkowym, przykładowe stałe liczbowe to:

456 -56.6 0.23 -.9 (zero w części całkowitej można pominąć)

Nie wolno umieszczać spacji wewnątrz zapisu liczby.

W stałej liczbowej może wystąpić litera ***e*** (lub ***E***) - oznaczająca notację naukową (zmiennoprzecinkową), w ogólnej postaci:

$$\pm m.n \begin{Bmatrix} e \\ E \end{Bmatrix} \pm p$$

gdzie *m*, *n*, *p* to ciągi cyfr (element ***.n*** można pominąć).

Przykłady z wyjaśnieniami:

2.5e5 - oznacza wartość 2.5×10^{5}
-4.78E-12 - oznacza wartość -4.78×10^{-12}
2e5 - oznacza wartość 200000

MATLAB jako jedno z nielicznych środowisk obliczeniowych może wykonywać działania na **liczbach zespolonych** (typu *double* z atrybutem *complex*), o postaci:

część_rzeczywista + *część_urojona* **i**

lub

część_rzeczywista + część_*urojona* **j**

Identyfikator **i** lub **j** służy do zapisu części urojonej liczby zespolonej (**i** = $\sqrt{-1}$, jednostka urojona).
Przykładowo, stałe zespolone mogą być zapisane:

4+7 i
-3.5 - 2.45 j

Podstawowa dokładność obliczeń wartości liczbowych w *MATLAB-ie* to 16 cyfr dziesiętnych (typ *double*).

W *MATLAB-ie* można korzystać ze standardowych stałych, które przedstawia tabela 2.3. Można zapoznać się z zakresami bezwzględnych wartości liczb zmiennoprzecinkowych jakich używa *MATLAB*.

Tabela 2.3. Wbudowane stałe w *MATLAB-ie*

pi	liczba π = 3.14159265... π może być także obliczone jako: 4*atan(1) lub imag(log(-1))
eps	dokładność mantysy liczb zmiennoprzecinkowych = 2.2204×10^{-16}
realmin	najmniejsza dodatnia liczba zmiennoprzecinkowa = 2.2251×10^{-308}
realmax	największa liczba zmiennoprzecinkowa = $1.7977\times10^{+308}$
intmin	najmniejsza liczba całkowita = -2147483648
intmax	największa liczba całkowita = 2147483647 (2^{31} = ponad 2 miliardy)
inf	*Infinity* - nieskończoność (∞)
NaN	brak liczby (*Not-a-Number*), wynik nieokreślony (np. dzielenia przez 0)
i lub j	jednostka urojona

Stałe są funkcjami bezargumentowymi (można je pisać w formie: *pi()*, *eps()*, *inf()* itd.)

W powyższych nazwach musi być przestrzegana wielkość liter.

2.4.2.Stałe tekstowe

Ciągi znaków są ważnym typem danych, wiele obiektów świata rzeczywistego jest opisywanych przy pomocy tekstów, na przykład nazwisko, nazwa towaru, nazwa pliku.

Stałe tekstowe - otoczone apostrofami (') - są to znaki umieszczone w kolejnych komórkach tablic znakowych (typ *char*). Przykładowe stałe:

'Politechnika'

Całe napisy - pisane cudzysłowie " - ciągi znaków (typ *string*):

"Jan Kowalski"

mogą być też przechowywane w komórkach tablic.

Teksty można przechowywać (zapamiętywać), są też argumentami funkcji służących do wyprowadzenia użytecznej informacji, komunikatów i opisów.

2.4.3. Stałe logiczne

Stałe logiczne w języku *MATLAB-a* to:

1 - prawda logiczna (*logical 1*)

0 - fałsz logiczny (*logical 0*)

2.5. Zmienne

Podstawowym elementem przechowującym dane jest **zmienna**. Zmienną nazywamy identyfikowany przez unikalną nazwę obszar pamięci programu, przechowujący wartość określonego typu. Zmienne są inicjowane automatycznie przy pierwszym nadaniu wartości przez użytkownika lub program, nie wymagają wcześniejszej deklaracji.

Reguły obowiązujące przy **nadawaniu nazw (identyfikatorów) zmiennych**:

- nazwa musi zaczynać się od litery,
- kolejne znaki nazwy mogą być literami, cyframi lub znakami podkreślenia, - niedozwolone są inne znaki, w tym polskie litery diakrytyczne ą, ę, ć itd.,
- nie wolno umieszczać spacji "wewnątrz" nazwy,
- duże i małe litery w nazwach są odróżnialne.

Przy tworzeniu identyfikatorów zmiennych należy unikać nazw wbudowanych stałych i funkcji (jak *pi*, *eps*, *sin*, *cos*, *abs*, *log* itp.), spowoduje to przesłonięcie ich znaczenia i odtąd nie będzie można korzystać z danej stałej czy funkcji w jej pierwotnym znaczeniu.

Powyższe zasady dotyczą także nadawania nazw *M-plikom*.

Najczęstszym sposobem utworzenia zmiennej i nadania jej wartości jest **instrukcja przypisania**. Zostanie ona opisana w kolejnym rozdziale.

3. Podstawy użytkowania

3.1. Praca w trybie interakcyjnym

W trybie interakcyjnym przeprowadza się zwykle pojedyncze, kalkulatorowe obliczenia, wykonuje polecenia systemu plików, uzyskania pomocy ekranowej i inne polecenia organizacyjne.

Poniżej zamieszczono przykłady pojedynczych poleceń.

Proste obliczenie arytmetyczne:

```
>> 3*4.5
ans =
    13.5000
```

Po wykonaniu polecenia obliczeniowego, wynik, oprócz pojawienia się w oknie *Command Window*, jest przechowywany przez *MATLAB-a* w elemencie pamięci zwanym **zmienną**. Jeżeli użytkownik nie nazwie zmiennej, program do przechowania rezultatu wykorzysta domyślną zmienną o nazwie *ans* (ang. *answer*), można tę zmienną wykorzystywać w kolejnych obliczeniach.

Usunięcie zawartości okna *Command Window*, pozostałej po poprzednich działaniach użytkownika:

```
>> clc
```

Ustalenie formatu widoczności 15 miejsc dziesiętnych dla liczb:

```
>> format long
```

Utworzenie katalogu o podanej nazwie:

```
>> mkdir cwiczenie1
```

Po akceptacji poprawnego polecenia, w oknie *Command Window* pojawia się odpowiednia informacja, adekwatna do treści polecenia. Gdy *MATLAB* napotka błąd składniowy, znaczeniowy bądź obliczeniowy, wówczas wyświetla w oknie *Command Window* odpowiedni komunikat w języku angielskim, czcionką w kolorze czerwonym.

Czasem pojawi się sugestia poprawnej wersji, na przykład:

```
>> PI
Undefined function or variable 'PI'.
            Niezdefiniowana zmienna PI
 Did you mean:
            Czy chodziło ci o?
>> pi
```

W przypadku popełnienia błędu, można w oknie *Command Window*, przy pomocy klawiszy nawigacyjnych ↑↓, "przewijać" listę ostatnio wykonywanych poleceń.

W menu *Home/Layout/Command History* można ustawić tryb *Popup*, wówczas w trakcie pisania poleceń pojawia się rozwijane okienko historii poleceń. Wybrane i ponowione polecenie można edytować i zaakceptować do wykonania.

Istnieje możliwość rejestracji przebiegu sesji, ciągu poleceń wykonanych w *Command Window* przez użytkownika. W tym celu przed rozpoczęciem sesji należy wykonać polecenie:

>> **diary** *nazwa_pliku*

W bieżącym katalogu zostanie utworzony plik o podanej nazwie, dziennik rejestracji naszej pracy. Start rejestracji obywa się samoczynnie.

Po zakończeniu działań polecenie:

>> **diary off**

zatrzymuje zapis dziennika.

Polecenie wznowienia rejestracji to:

>> **diary on**

Dziennik zapisany w pliku tekstowym można przeglądać i edytować dowolnym edytorem *ASCII* (np. *Notatnikiem Windows* lub edytorem *MATLAB-a*).

Sposób wyświetlania zależy od aktualnego parametru polecenia *format*. Przykładowo, jeżeli zostanie napisane polecenie (w *Command Window* lub w skrypcie):

format short - format domyślny, liczby będą wyświetlane z 4-ma miejscami dziesiętnymi,
format long - liczby będą wyświetlane z 15 miejscami dziesiętnymi,
format bank - liczby będą wyświetlane z 2 miejscami dziesiętnymi,
format rat - liczby w formacie ułamków zwykłych.

Przykład zastosowania polecenia:

```
>> format long
>> pi
ans =
  3.141592653589793
```

3.2. Instrukcja przypisania

W celu nabycia umiejętności tworzenia zmiennych przechowujących dane oraz wykonywania prostych i złożonych obliczeń, należy zapoznać się z instrukcją przypisania.

Instrukcja przypisania służy do inicjacji zmiennej - utworzenia zmiennej oraz umieszczenia jej w obszarze roboczym *Workspace,* oraz **nadania tej zmiennej wartości określonego typu** (lub zmiany wartości dla wcześniej zainicjowanej zmiennej).

Instrukcja przypisania jest najczęściej używanym poleceniem w programowaniu. Można wpisywać i wykonywać przypisania w wierszu poleceń *Command Window*, istnieje też możliwość utrwalenia sekwencji instrukcji w pliku.

Ogólna postać instrukcji przypisania to:

nazwa_zmiennej = wyrażenie

Należy pamiętać, że operator **=** służy do nadawania wartości - w odróżnieniu od operatora relacji **==**, który służy do porównywania wyrażeń.

Najprostsza postać instrukcji przypisania to nadanie zmiennej wartości liczbowej. Po wykonaniu instrukcji przypisania *MATLAB* reaguje "echem" ekranowym - pokazuje efekt polecenia.

W *MATLAB-ie* nie trzeba określać typu zmiennej, typ nadawany jest na podstawie typu wartości przypisanej do zmiennej.

Efekt przypisania w wierszu poleceń *Command Window*:

```
>> alfa = 31.5
alfa=
      31.5
>>
```

Utworzona została zmienna o nadanej przez użytkownika nazwie *alfa*, została jej nadana wartość typu liczbowego (*double*).

Echo nie pojawia się, gdy tekst polecenia jest zakończony znakiem średnika **;**

```
>> beta = -12;
>>
```

Zmienna *beta* z nadaną wartością także została utworzona (dołączona do obszaru roboczego *Workspace*).

Należy przypomnieć, że jeżeli jako polecenie wpisana zostanie tylko wartość liczbowa lub wyrażenie obliczeniowe, *MATLAB* dokona przypisania wyniku do systemowej zmiennej *ans* (ang. *answer*).

```
>> 2.9+5.1
ans=
      8
>>
```

Można zapoznać się z listą aktualnych zmiennych obszaru roboczego w oknie *Workspace* (rys.3.1).

Current Folder | Workspace

Name	Value	Class
alfa	31.5000	double
ans	8	double
beta	-12	double

Rys.3.1. Lista zmiennych w obszarze roboczym (*Workspace*)

Jeżeli polecenie zawiera tylko nazwę zmiennej (bez nadania wartości), *MATLAB* wyświetli w *Command Window* aktualną wartość zmiennej, o ile wartość ta została wcześniej określona (czyli istnieje w obszarze roboczym *Workspace*):

```
>> beta
beta =
    -12
```

Zmienną o wcześniej zdefiniowanej wartości można wykorzystać w wyrażeniu przypisywanym nowej zmiennej. Nie wolno używać w wyrażeniach zmiennej o nieznanej wartości.

Przykładowo:

```
>> a= 6.7
a =
    6.7
```

```
>> b=2*a
b =
    13.4
>> c=2*x
Undefined function or variable 'x'.
```

Niezdefiniowana funkcja lub zmienna 'x'

Poniżej zestawiono przykłady ilustrujące znaczenie poszczególnych postaci instrukcji przypisania:

x= 23.5	nadanie zmiennej ***x*** wartości liczbowej
x=2*5.1	nadanie zmiennej ***x*** wartości prostego wyrażenia obliczeniowego,
x=sin(pi/3)	nadanie zmiennej ***x*** wartości wyrażenia wykorzystującego funkcję matematyczną
x=10	zmiana poprzedniej wartość zmiennej na inną wartość,
x=x+2	zwiększenie wartości zmiennej ***x*** o 2, uwaga: poprzednia wartość zmiennej ***x*** musi być określona
x=3*x	trzykrotne zwiększenie wartości zmiennej ***x***, poprzednia wartość zmiennej ***x*** musi być określona
x=b	powielenie wartości zmiennej ***b*** w zmiennej ***x*** ("*niech x ma tę samą wartość jak **b***") - zmienna ***x*** może być wcześniej określona lub po raz pierwszy inicjowana, wartość zmiennej ***b*** powinna być wcześniej określona

Istnieje możliwość nadawania zmiennym wartości stałych typu znakowego (*char*):

```
>>napis = 'Wydział Budowy Maszyn i Lotnictwa'
napis=
  'Wydział Budowy Maszyn i Lotnictwa'
```

a także wartości logicznych (*logical*), przykładowo:

```
>>pytanie = sin(4.5)>tan(4.8)
pytanie =
  logical
    1
```

Utworzona została zmienna o nazwie *pytanie*, która bada wartość logiczną porównania dwóch wyrażeń arytmetycznych, w efekcie zmienna przyjmuje wartość logiczną *1* (czyli *prawda logiczna*).

Jak już wspomniano, podstawowym elementem przechowującym dane jest macierz (ang. *matrix*). Każdą macierz, nawet jednokomórkową, reprezentuje element programu zwany zmienną. Zmienna może przechowywać dane różnych typów.

Przy pomocy instrukcji przypisania można utworzyć wiele zmiennych różnych typów. Ich listę wraz z rozmiarem i typem danych (*class*) można znaleźć w oknie *Workspace* lub wyświetlić ich listę wykonując polecenie *whos*:

whos

Podstawowe typy to wymienione już wyżej:

double - typ liczbowy,
char - typ znakowy,
string - typ tekstowy (ciąg znaków),
logical - typ logiczny,
cell - typ komórkowy,
struct - typ struktury,
sym - typ symboliczny.

Każda zmienna posiada określony typ.

Jeżeli w *Command Window* zostanie wykonany poniższy ciąg kolejnych instrukcji przypisania:

```
>>a = 5
>>b = 'ABC'
>>c=" Politechnika"
>>d = 2 > 4
>>e = {1, 2, 3, 'x' }
>>f.wiek = 3
>>g= 4+3i
```

a następnie polecenie:

```
>>whos
```

pojawi się poniższa informacja:

Name	Size	Bytes	Class	Attributes
a	1x1	8	double	
b	1x3	10	char	
c	1x1	174	string	
d	1x1	1	logical	
e	1x4	474	cell	
f	1x1	184	struct	
g	1x1	16	double	complex

Utworzono 7 zmiennych o jednoliterowych nazwach. Można się przekonać, że każda z tych zmiennych jest macierzą o odpowiednich wymiarach (*Size*), zajmuje określone miejsce w pamięci i **jest innego typu** (*Class*):

a - typ liczbowy,
b - typ znakowy,
c - typ tekstowy,
d - typ logiczny,
e - typ komórkowy,
f - element struktury,
g - typ liczbowy (zespolony).

Identyczna lista znajdzie się w oknie *Workspace* (rys.3.2).

Name	Value	Size	Class
a	5	1x1	double
b	' ABC '	1x5	char
c	" Politechnika"	1x1	string
d	0	1x1	logical
e	*1x4 cell*	1x4	cell
f	*1x1 struct*	1x1	struct
g	4.0000 + 3.00...	1x1	double (complex)

Rys.3.2. Aktualna lista zmiennych w oknie *Workspace*

Jeżeli zmienna będzie złożona z wielu elementów składowych (macierz, tablica komórek), po dwukrotnym kliknięciu w zmienną w oknie *Workspace*, otwiera się okienko w formie arkusza, z wyszczególnionymi wartościami składowymi w komórkach. Po kliknięciu w zmienną ***e,*** otrzymamy arkusz jak na rys.3.3.

e

1x4 cell

	1	2	3	4
1	1	2	3	'x'
2				

Rys.3.3. Wartości elementów zmiennej *e* w arkuszu.

Informację o typie dowolnej zmiennej można też uzyskać przy pomocy poznanej funkcji *class*, a więc, bazując na powyższym zestawie zmiennych, otrzymamy:

```
>> class(a)
  ans=
    'double'
```

Można wykonać w *Command Window* użyteczny ciąg instrukcji przypisania. Poniższy przykład pokazuje realizację obliczenia pola powierzchni okręgu, przy zadanej wartości promienia:

```
>> r = 5
r =
    5
>> pole=pi*r^2
  pole =
      78.5398
```

Utworzono zmienną ***r*** o wartości 5, następnie wykorzystano wartość tej zmiennej w odpowiednim wyrażeniu obliczeniowym. Wartości zmiennych *r* i *pole* przechowywane są w obszarze roboczym *Workspace* w celu ewentualnego, późniejszego wykorzystania.

Usunięcie zmiennej (zmiennych) z obszaru roboczego *Workspace* wykonuje się poleceniem *clear*:

clear - usuwa wszystkie zmienne,

clear *zmienna1 zmienna2 ...itd.* - usuwa zmienne o wyszczególnionych nazwach.

można też usuwać zmienne interaktywnie w oknie *Workspace*, stosując menu kontekstowe zmiennej.

Wykorzystując instrukcję przypisania, można nadawać zmiennym wartości bardziej złożonych wyrażeń, zasady ich konstruowania przedstawi kolejny rozdział.

3.3. Wyrażenia obliczeniowe

3.3.1. Proste wyrażenia z wykorzystaniem operatorów arytmetycznych

Chcąc wykonywać w *MATLAB-ie* proste i bardziej złożone obliczenia, należy zaznajomić się z zasadami tworzenia wyrażeń obliczeniowych. Wartość obliczonego wyrażenia może zostać przechowana w zmiennej, przy wykorzystaniu poznanej instrukcji przypisania.

Wyrażeniem może być:

stała

zmienna

wyrażenie **operator** *wyrażenie*

funkcja (*wyrażenie*)

Dwie ostatnie postacie definicji *wyrażenia* są rekurencyjne, to znaczy definiując *wyrażenie* wykorzystują pojęcie definiowane. Stąd wynika logiczny wniosek, że wiele wyrażeń można łączyć operatorami w długim ciągu, a także, jeżeli wyrażenie może być argumentem funkcji, to tym argumentem może być stała, zmienna, ciąg wyrażeń połączonych operatorami bądź inna funkcja.

Dostępne w języku *MATLAB-a* **operatory arytmetyczne** (macierzowe):

+	operator dodawania (także powielenie znaku)
–	operator odejmowania (także zmiana znaku)
*****	operator mnożenia
/	operator dzielenia
^	operator potęgowania (zamiennie do funkcji *power*)

Domyślny priorytet operacji jest następujący:

- następnie wykonywane są funkcje, poczynając od najbardziej wewnętrznych; jeżeli argumentami funkcji są wyrażenia z operatorami, to obowiązuje poniższy priorytet operatorów.

Domyślny priorytet operatorów:

- operacje potęgowania ^ ma najwyższy priorytet,
- następnie zmiany znaku przed liczbą (zmienną),
- potem operacje multiplikatywne (mnożenia i dzielenia - od lewej do prawej),
- na końcu operacje addytywne (dodawania i odejmowania - od lewej do prawej).

W wyrażeniach mogą też być stosowane nawiasy okrągłe **( )**, których celem jest zmiana domyślnej kolejności wykonania operacji. Najwcześniej wykonywane są operacje w nawiasach, poczynając od najbardziej wewnętrznych,

W odróżnieniu od notacji matematycznej, wyrażenie obliczeniowe należy zapisać w jednym wierszu, w tekście nieformatowanym nie da się stosować indeksów górnych

dla potęg, "piętra" ułamka, specjalnych symboli jak np. pierwiastek w ce $\sqrt{}$. Poprawne konstruowanie wyrażeń wymaga pewnej wprawy.

Należy zwrócić uwagę, że operacja potęgowania ma domyślnie wyższy priorytet niż operacja zmiany znaku, a zatem wyrażenie obliczeniowe:

```
- 2^2
```

daje wynik -4 (a nie 4 jak w *MS Excel*, gdzie operacja zmiany znaku jest wykonywana przed operacją potęgowania).

Wyrażenie musi być obliczalne, to znaczy wszystkie jego elementy powinny być określone, w szczególności zmienne wykorzystywane w wyrażeniu muszą mieć wcześniej określone wartości.

W przypadku niepoprawnie skonstruowanego wyrażenia wynik będzie nieprawidłowy lub *MATLAB* wyświetli komunikat o błędzie.

Najczęściej popełniane błędy w wyrażeniach:

<table>
<tr><td rowspan="3">brak operatora mnożenia</td><td colspan="2">Przykładowo:
(x-1)(y+1)
Pojawi się komunikat o błędzie:
Error: Unexpected MATLAB expression.
Błąd: nieoczekiwane wyrażenie
Nowsze wersje MATLAB-a w przypadku błędu sugerują poprawkę:
did you mean?
czy miałeś na myśli?
(x-1)*(y+1)</td></tr>
</table>

<table>
<tr><td rowspan="3">brak lub źle użyte nawiasy</td><td colspan="2">Przy mnożeniu (dzieleniu) wielomianów trzeba pamiętać o nawiasach na wielomianach:</td></tr>
<tr><td>Matematyka</td><td>MATLAB</td></tr>
<tr><td>$\frac{x+2}{1-y}$</td><td>(x+2)/(1-y)</td></tr>
<tr><td>niedomknięte nawiasy</td><td colspan="2">Zawsze powinno się sprawdzić liczbę nawiasów otwartych i zamkniętych oraz ich właściwe umiejscowienie, uwzględniające kolejność działań.
W przypadku błędu pojawi się komunikat o błędzie:
Invalid expression. When calling a function or indexing a variable, use parentheses. Otherwise, check for mismatched delimiters.
Niepoprawne wyrażenie. Podczas wywoływania funkcji lub indeksowania zmiennej użyj nawiasów. W przeciwnym razie sprawdź niedopasowane ograniczniki.
lub w starszych wersjach MATLAB-a:
Unbalanced or unexpected parenthesis or bracket.
Niedomknięty lub nieoczekiwany nawias.</td></tr>
</table>

niepoprawnie zapisany iloczyn w mianowniku ułamka	Matematyka	MATLAB
	$\frac{1}{x\ y}$	1/(x*y) lub 1/x/y co wynika z prostych zasad operacji na ułamkach
dwa sąsiadujące operatory	Błędne są zapisy: 3*^2 lub 5+/2. Error: Unexpected MATLAB operator *Nieoczekiwany opreator* Niemniej jednak poprawne jest: 5+-3 oraz 11/+3 ponieważ tu operatory + i – użyte są w kontekście zmiany znaku lub jego powielenia.	
użycie niezdefiniowanej zmiennej jako argumentu w wyrażeniu	Przykładowo: >>a=5*x Error: Undefined function or variable 'x' *Błąd: Niezdefiniowana funkcja lub zmienna 'x'*	

Przykłady prostych, poprawnych wyrażeń arytmetycznych z wykorzystaniem operatorów i nawiasów (obliczalnych, o ile wcześniej określono wartości zmiennych *x* i *y*):

```
(2+3*x)/(2.45-1.1)
(-x^2+4.5)*3
-((y-1)*(y+1)-x)/2
```

3.3.2. Standardowe funkcje arytmetyczne

W wyrażeniach można wykorzystywać standardowe funkcje stosowane w matematyce. Tabele 3.4 i 3.5 prezentują najczęściej stosowane funkcje.

Tabela 3.4. Podstawowe standardowe funkcje matematyczne

sin(*w*) **cos**(*w*) **tan**(*w*) **cot**(*w*)	funkcje trygonometryczne - argument ***w*** powinien być wyrażony w mierze łukowej kąta, czyli w radianach! Są też dostępne funkcje odwrotne (*asin*, *acos* itp.) oraz funkcje hiperboliczne.
sind(*w*) **cosd**(*w*) **tand**(*w*) **cotd**(*w*)	funkcje trygonometryczne - wymagają argumentu ***w***, wyrażonego w mierze stopniowej
log(*w*)	logarytm naturalny - w matematyce: $\ln w$
log10(*w*)	logarytm dziesiętny - w matematyce: $\log w$
exp(*w*)	funkcja wykładnicza - e^{w}
sqrt(*w*)	pierwiastek kwadratowy: $\sqrt{w}$
abs (*w*)	wartość bezwzględna
fix (*w*)	zaokrąglenie wyrażenia *w* do liczby całkowitej w kierunku zera

floor(*w*)	zaokrąglenie wyrażenia *w* do liczby całkowitej w kierunku -∞
ceil (*w*)	zaokrąglenie wyrażenia *w* do liczby całkowitej w kierunku +∞
round(*w*)	zaokrąglenie wyrażenia *w* do najbliższej liczby całkowitej
round (*w*, *N*)	zaokrąglenie wyrażenia *w* do *N* cyfr dziesiętnych
rand	bezargumentowy generator liczby losowej dziesiętnej z przedziału (0, 1)
rem (*a*, *b*)	reszta z dzielenia *a* przez *b*
power(*a*, *b*)	potęgowanie a^b - alternatywnie do operatora potęgowania: ^
nthroot(*a*, *b*)	pierwiastek stopnia b

Tabela 3.5. Funkcje działań na liczbach i zmiennych zespolonych: z=a+b**i**

abs (*z*)	wartość bezwzględna liczby zespolonej, czyli $\sqrt{a^2 + b^2}$
real (*z*)	część rzeczywista liczby zespolonej: a
imag (*z*)	część urojona liczby zespolonej: b
angle (*z*)	kąt wyznaczony przez $\mathrm{arctg}(\frac{b}{a})$ - w *MATLAB-ie*: atan(b/a)
conj (*z*)	liczba sprzężona do ***z*** (przeciwny znak części urojonej b)

Nazwy wszystkich powyższych funkcji należy pisać małymi literami.

Jak już nadmieniono, można uzyskać listę dostępnych w *MATLAB-ie* funkcji matematycznych, wraz z opisem i przykładami, pisząc w *Command Window* polecenie:

help elfun - interaktywny spis funkcji elementarnych

help *nazwa_funkcji* - pomoc na temat danej funkcji

Oprócz wymienionych sposobów uzyskania pomocy, w nowszych wersjach *MATLAB-a* można skorzystać z podpowiedzi w rozwijanym okienku pomocy kontekstowej, w którym można przeglądnąć listę dostępnych argumentów funkcji. Trzeba w tym celu napisać w edytorze lub w *Command Window* nazwę funkcji i otworzyć nawias (i dalej *More help..*).

Jak wynika z zestawienia tabeli 2.3, istnieją funkcje:

- bezargumentowe (jak np. *rand*),
- jednoargumentowe (przeważająca większość),
- wieloargumentowe (jak np. funkcja *power*).

Trzeba pamiętać, że argument (lub argumenty) funkcji matematycznych powinny być umieszczone w nawiasach okrągłych, bezpośrednio po nazwie funkcji.

W przypadku funkcji bezargumentowych nawiasy można pominąć - można też zastosować "puste" nawiasy, bez zawartości, np. *rand*( ).

Jak zostanie opisane w dalszej części, istnieją funkcje, w których można używać różnej liczby argumentów, co daje różne rezultaty.

Przykładem może być funkcja *rand*, której użycie w postaci:

rand - zwraca jedną liczbę losową,

zaś w postaciach:

rand(3) - zwraca zbiór losowych liczb rzeczywistych z przedziału (0, 1), umieszczonych w macierzy kwadratowej o rozmiarach 3×3,

rand (4, 5) - jak wyżej, lecz w macierzy prostokątnej o rozmiarach 4×5.

3.3.3. Przykłady poprawnych wyrażeń obliczeniowych

Funkcje matematyczne używane są w bardziej złożonych wyrażeniach obliczeniowych. Najistotniejsze cechy konstruowania poprawnych wyrażeń omawia tabela 3.6.

Tabela 3.6. Uwagi dotyczące poprawnego pisania wyrażeń zawierających funkcje

	Matematyka	MATLAB
Argumenty funkcji matematycznych powinny być zawsze w nawiasach okrągłych	$sin\ x$	sin(x)
Pierwiastki *n*-tego stopnia oblicza się jako potęgę, z wykładnikiem będącym odwrotnością rzędu pierwiastka, można tez wykorzystywac funkcję **nthroot**	$\sqrt[3]{x}$	x^(1/3) lub power(x, 1/3) lub nthroot(x, 3)
	$\sqrt{x}$	x^(1/2) lub power(x, 1/2) lub sqrt(x)
Podnoszenie funkcji do potęgi	sin^3x	sin(x)^3 lub wykorzystanie zagnieżdżania funkcji: power(sin(x), 3)
Częstym błędem jest niezrozumienie zapisu funkcji wykładniczej		exp(1) to liczba *Eulera* = 2.718281....
	e^x	exp(x)
	$e^{-2tg\ x}$	exp(-2*tan(x))
Logarytmy	log(x) to funkcja logarytmu naturalnego, log10(x) to funkcja logarytmu dziesiętnego! Logarytmy o dowolnej podstawie *p* można obliczać korzystając z przekształcenia matematycznego: $log_p x = \frac{ln\ x}{ln\ p}$	
	$log_3\ x$	log(x)/log(3)
Należy uważać na miarę kąta funkcji trygonometrycznych	Dla kątów w mierze stopniowej wykorzystuje się funkcje *sind, cosd, tand, cotd* Wyrażenia: sin(pi/2) oraz sind(90) dadzą ten sam wynik: 1 Funkcje odwrotne, jak: *asind, acosd* itd., zwracają kąt w stopniach.	

Poniżej przykładowe, bardziej złożone wyrażenia i ich poprawne wersje w języku *MATLAB-a*:

Matematyka	MATLAB
$\frac{3tg^3x - 2\ln x}{3 - \sqrt[3]{x^3 + 3}}$	(3*tan(x)^3-2*log(x))/(3-(x^3+3)^(1/3)) lub (3*power(tan(x),3)-2*log(x))/(3-power(x*x*x+3,1/3))
$\frac{3sin^3(3x^3) + 1}{e^{\sqrt{x}} + \frac{1}{1 + \frac{1}{x}}}$	(3*sin(3*x^3)^3+1)/(exp(sqrt(x))+1/(1+1/x)) lub (3*power(sin(3*x^3), 3)+1)/(exp(sqrt(x))+1/(1+1/x))

Wartość zmiennej *x* musi być wcześniej określona, aby powyższe wyrażenia dało się poprawnie obliczyć. Należy zwrócić uwagę na zagnieżdżanie funkcji - w ostatnim przykładzie funkcja *sqrt(x)* jest argumentem funkcji wykładniczej e^x - oraz na poprawne użycie nawiasów.

3.3.4. Zaokrąglenia liczb dziesiętnych

Chcąc zaokrąglić dziesiętną wartość liczbową do wymaganej dokładności, można zastosować następujący algorytm:

round(*x**10^*N*)/10^*N*

gdzie *N* jest liczbą cyfr dziesiętnych wymaganego zaokrąglenia.

Wyjaśnienie przepisu: kropka dziesiętna, na skutek pomnożenia wartości *x* przez 10^*N* jest przesuwana w prawo o *N* pozycji, następnie wykonywane jest zaokrąglenie do najbliższej liczby całkowitej (*round*), w końcu kropka wraca na poprzednie miejsce po podzieleniu liczby całkowitej przez 10^*N*.

Przykładowe użycie:

```
>>format long
>> x=pi/2
x =
  1.570796326794897
>>w=round(x*10000)/10000
w=
   1.570800000000000
```

W nowszych wersjach *MATLAB-a* można stosować funkcję *round* z dwoma argumentami:

round(*w*, *n*)

gdzie:

w - wyrażenie, którego wartość zaokrąglamy,

n - liczba cyfr dziesiętnych.

Przekładowe użycie:

```
>>z=round(pi, 4)
z=
 3.141600000000000
```

3.3.5.Wyrażenia logiczne

Relacje to proste wyrażenia logiczne, łączące dwa wyrażenia arytmetyczne operatorem relacji (porównania). Odbywa się porównanie w aspekcie wielkości liczbowej lub kolejności alfabetycznej znaków.

W *MATLAB-ie* występują następujące **operatory** relacji, łączące wyrażenia arytmetyczne lub znakowe:

<	czy mniejsze niż
<=	czy mniejsze lub równe
>	czy większe niż
>=	czy większe lub równe
==	czy równe
~=	czy nierówne

Przykłady relacji:

4>5   sin(pi/2)<=cos(pi/7)   sin(pi/2)= =1   log10(5) ~ =1

Rezultatem relacji jest:

prawda logiczna (*logical* 1)

lub

fałsz logiczny (*logical* 0).

Operatory ">" i "<" badają także kolejność alfabetyczną znaków, przykładowo:

```
>>'k'>'a'
ans =
   logical
   1
```

Ustalenie kolejności alfabetycznej dłuższych tekstów wymaga zastosowania tablicy komórkowej, będzie o tym mowa w rozdziale dotyczącym tego typu danych.

Dwa (lub więcej) wyrażeń logicznych można połączyć przy pomocy operatorów logicznych **koniunkcji** lub **alternatywy** lub zastosować operator **negacji** (przed wyrażeniem logicznym).

Operatory logiczne:

&	operator koniunkcji (iloczyn logiczny)
|	operator alternatywy (suma logiczna)
~	operator negacji

Można też stosować notację podwójnych znaków: && i || (jak w języku *C, JavaScript* i innych).

Priorytet operatorów jest następujący:

- najwcześniej wykonywana jest negacja ~,
- potem operatory relacji: < , <= , > , >= , == , ~=,
- następnie iloczyny logiczne &,
- na końcu sumy logiczne |.

Można też korzystać z następujących funkcji logicznych:

<table>
<tr><td>and (x, y)</td><td>iloczyn logiczny - równoważne z: x&y</td></tr>
<tr><td>or (x, y)</td><td>suma logiczna - równoważne z: x|y</td></tr>
<tr><td>xor(x, y)</td><td>suma wyłączająca</td></tr>
<tr><td>not (x)</td><td>negacja - równoważne z: ~x</td></tr>
</table>

W razie konieczności zmiany kolejności operacji logicznych, należy używać nawiasów okrągłych.

Oto przykłady poprawnych wyrażeń logicznych:

```
x>10 & x<20         - prawda logiczna, jeżeli x należy do przedziału (10,20)
x<0 | x>100         - prawda logiczna, jeżeli x jest poza przedziałem [0, 100]
~(x>0 & x<100)      - prawda logiczna, jeżeli x jest poza przedziałem (0, 100)
```

a także inne:

```
~(4>3)     (x>0|y<5)&z==1     or(x==4, y>10)     and(sin(pi)>0, log(x)<=5)
```

Należy unikać używania operatora **==** dla sprawdzania równości wyrażeń o wartościach dziesiętnych.

Wyrażenia logiczne stosuje się najczęściej w instrukcjach warunkowych *if* oraz pętlach *while*, co będzie opisane w kolejnych rozdziałach.

Należy pamiętać, że:

<table>
<tr><th>Operacja logiczna</th><th>Wynik operacji</th></tr>
<tr><td>prawda & prawda</td><td>prawda</td></tr>
<tr><td>prawda & fałsz</td><td>fałsz</td></tr>
<tr><td>fałsz & fałsz</td><td>fałsz</td></tr>
<tr><td>prawda | prawda</td><td>prawda</td></tr>
<tr><td>prawda | fałsz</td><td>prawda</td></tr>
<tr><td>fałsz | fałsz</td><td>fałsz</td></tr>
<tr><td>~prawda</td><td>fałsz</td></tr>
<tr><td>~fałsz</td><td>prawda</td></tr>
</table>

4. Tworzenie i wykonywanie skryptów (*M-plików*)

4.1. Wymagania i uwagi praktyczne

Jak nadmieniono wyżej, ***M-pliki*** (skrypty) to pliki tekstowe, zawierające sekwencję poleceń do wykonania w trybie wsadowym. W *M-pliku* mogą zostać opracowane ciągi instrukcji, prowadzące od inicjacji danych wejściowych, poprzez obliczenia wyrażeń, analizy rozwidlające działania, aż do prezentacji wyników.

Instrukcje przedstawione w poprzednim rozdziale, wpisywane dotychczas pojedynczo w wierszu poleceń *Command Window*, mogą być teraz utrwalone w pliku jako logiczna sekwencja działań i operacji, w celu jej wielokrotnego użycia.

Edycja skryptu może zostać zainicjowana następującymi sposobami:

- wykorzystując menu *Home/New Script* (w starszych wersjach *MATLAB-a* menu *File/New/M-file* lub po kliknięciu ikony paska narzędzi *New M-file*),
- wykorzystując menu *Home/New* - wybór z listy: *Script*,
- w oknie *Current Folder* - prawy klik myszką- *New/Script*.

Po wykonaniu jednej z powyższych operacji pojawia się puste okno *Editor*, w którym można napisać tekst naszego programu. Utworzony w oknie edytora skrypt trzeba zapisać w pliku z wybraną nazwą (z rozszerzeniem **.m**) w wybranym katalogu.

Zasady nazywania *M-plików* podlegają tym samym zasadom co nazwy zmiennych, czyli **pierwszy znak to litera**, następne znaki to sekwencja kombinacji liter, cyfr i znaków podkreślenia (**bez spacji**). Nazwa pliku może być długa: od 1 do 63 znaków.

Duże i małe litery są w nazwach odróżnialne.

Nazwa pliku powinna być (tak jak nazwy zmiennych) różna od nazw stałych i funkcji standardowych *MATLAB-a*. Próbę nazwania pliku słowem kluczowym języka (jak *if*, *for*, *case*, *while*, *switch, else, end* i innych) środowisko samo wychwyci jako błąd i zareaguje odpowiednim ostrzeżeniem.

Przykłady poprawnych nazw *M-plików*:

zadanie1.m   mojeDane.m   cw1_a.m

i niepoprawnych:

~~5plik.m~~   ~~moj skrypt.m~~   ~~ćw3.m~~

Oczywiście nie wolno zapomnieć o rozszerzeniu nazwy (.m), choć *MATLAB* domyślnie je sam dodaje.

Zapisany (archiwizowany) skrypt można w dowolnym momencie otworzyć w oknie *Editor*:

- podwójnym kliknięciem w pozycję pliku, z listy w oknie bieżącego foldera (*Current Folder*),
- z menu *Home/Open* - wybór w oknie eksploracji,
- przeciągnięciem z okna *Current Folder* do okna *Editor.*

W edytorze istnieje możliwość opracowywania kilku plików na kartach, przełączania ich widoku przy pomocy zakładek, a także rozmieszczania w oknie (wstęga menu *View*).

Utworzone pliki (także te archiwizowane w katalogu roboczym) można uruchomić, czyli wymusić całościowo kolejne wykonanie zapisanych w *M-pliku* poleceń (instrukcji języka).

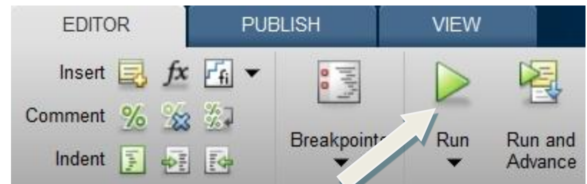
> **Instrukcje zapisane w skrypcie (*M-pliku*) wykonywane są kolejno, od pierwszej do ostatniej (jeżeli nie zrobiliśmy błędu!), chyba, że charakter instrukcji wskazuje na inną kolejność.**

Należy uważać na logiczną kolejność instrukcji: wprowadzenie danych, w kolejnych instrukcjach wykonanie obliczeń wyrażeń, które wykorzystują te dane.

Puste wiersze, umieszczone pomiędzy kolejnymi instrukcjami w skrypcie, nie mają znaczenia.

Poniżej sposoby wykonania skryptu:

- przy pomocy ikony *Run* w menu *Editor* (rys. 4.1),



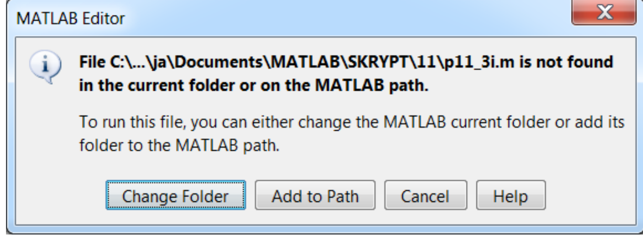
Rys.4.1. Ikona uruchamiania skryptu (*Run*)

- klawiszem *F5* (jeżeli aktywne jest okno *Editor*).

Testując wykonanie *M-pliku* można wykorzystywać *Debugger.* Pracę z *Debuggerem* rozpoczyna ustawienie "pułapki" (*breakpoint*) na dowolnej instrukcji programu (funkcji). Najprościej można to zrobić klikając w *Edytorze* na lewym marginesie edytowanego skryptu, na poziomie wybranej instrukcji. Pojawi się tam czerwony punkt, zaznaczenie miejsca chwilowego zatrzymania wykonywania skryptu. Skrypt po uruchomieniu wykonuje się do najbliższego *breakpoint-u*. Po jego wstrzymaniu można kontrolować aktualne wartości zmiennych. Kontynuacja działania skryptu odbywa się przez kliknięcie przycisku *Continue.*

4.2. Kontrola błędów

Wykonywany skrypt powinien się znajdować w katalogu roboczym, widocznym w oknie *Current Folder*. Możliwe jest też wykonanie skryptów spoza katalogu bieżącego, lecz katalog ten powinien być dołączony do ścieżki przeszukiwań *MATLAB-a* (*path*). Jeżeli w oknie *Editor* znajduje się tekst skryptu z innego katalogu, wówczas próba jego wykonania wywoła monit sugerujący zmianę katalogu bieżącego (rys. 4.2).



Rys.4.2. Błąd próby wykonania pliku, znajdującego się poza katalogiem roboczym

Błędy składniowe w skrypcie mogą być widoczne już w trakcie pisania instrukcji w edytorze (podkreślenia "wężykiem" w kolorze czerwonym).

Należy zaznaczyć, że wykonanie skryptu jest interpretowane, czyli skrypt zatrzyma się po napotkaniu pierwszego błędu, po jego poprawieniu może zostać znaleziony następny błąd w jednej z kolejnych instrukcji, itd.

Najczęstsze błędy popełniane przy konstruowaniu wyrażeń z opisano w rozdz. 3.3.

Możliwe typy błędów to między innymi:

- źle skonstruowane wyrażenie (brak lub niedomknięcie nawiasów, niezdefiniowana zmienna, nieznana nazwa funkcji itp.),
- nieprawidłowa konstrukcja instrukcji strukturalnej (np. brak słowa kluczowego *end*),
- nieprawidłowe indeksowanie macierzy (np. różne liczby elementów w wierszach, próba dostępu do elementu spoza rozmiaru macierzy, nieprawidłowe wymiary macierzy dla danej operacji itp.),
- nieprawidłowa liczba argumentów funkcji,
- zły typ danych w kontekście argumentu wyrażenia lub funkcji (np. typ znakowy zamiast liczbowego).

Mogą też wystąpić błędy wynikające z braku możliwości wykonania polecenia, na przykład braku lub złej lokalizacji pliku, z którego skrypt chce odczytać dane.

Po uruchomieniu skryptu każdy z tych błędów zostanie opisany anglojęzycznym komunikatem w oknie *Command Window*. Proces systematycznego eliminowania błędów nosi nazwę *debugging*-u ("odpluskwiania"), *debugger* jest programem lub procesem, służącym do analizy i identyfikacji błędów.

4.3. Pierwszy skrypt

Zadanie: Napisać prosty skrypt dla obliczeń pola powierzchni trójkąta, przy zadanych długościach podstawy i wysokości. Wypisać listę użytych zmiennych.

Należy utworzyć nowy *M-plik* w edytorze o treści:

Treść skryptu	Rezultat wykonania skryptu w *Command Window*

<pre>clc   %wyczyść Command Window
clear   %usuń zmienne
a = 2.5   %przypisz wartości do zmiennej
h = 4.7   %przypisz wartości do zmiennej
pole = a*h/2   %oblicz wyrażenie i przypisz do zmiennej
whos   %lista zmiennych</pre>
	<pre>a =
	   2.5000
	h =
	   4.7000
	pole =
	   5.8750
	
	  Name Size Bytes Class
	  a 1x1 8 double
	  h 1x1 8 double
	  pole 1x1 8 double</pre>

Próba jego pierwszego wykonania (*Run* lub klawisz *F5*) będzie skutkować żądaniem zapisu w *M-pliku*. Należy ustalić nazwę pliku z rozszerzeniem **.m** (przestrzegając omówionych wyżej zasad tworzenia nazw) i następnie ponownie wykonać skrypt.

Kolejne przypisania zostały napisane w osobnych wierszach, co zwiększa czytelność skryptu, choć można je pisać w jednym wierszu, oddzielając instrukcje przecinkami.

Można w instrukcjach przypisania zmienić dane wejściowe i wykonać *M-plik* ponownie. Po każdej modyfikacji skryptu, nie trzeba go już zapisywać, kolejne wykonanie jest poprzedzone automatycznym nadpisaniem treści pliku.

Niekiedy *MATLAB* przechowuje w katalogu pliki z rozszerzeniem *.asv* (ang. *autosave*), zawierające zapasowe wersje *M-plików* sprzed ostatniej poprawki. Można je zachować lub usunąć.

Dobrym zwyczajem jest rozpoczynanie skryptu, jak w zamieszczonym przykładzie, od poleceń:

clc - usunięcie zawartości okna *Command Window* po wcześniejszych operacjach,

clear - wszystkie zmienne zostaną usunięte z obszaru roboczego *Workspace*, inaczej skrypt może korzystać ze zmiennych i ich wartości nadanych w poprzednich wykonaniach tego lub innego skryptu.

Jak już wspomniano, wybrane instrukcje można kończyć znakiem średnika (;), co spowoduje brak echa ekranowego danej instrukcji. Można zatem zmodyfikować nasz skrypt do postaci:

Skrypt	Wynik

clc, clear
a = 2.5;
h = 4.7;
pole = a*h/2
	pole =
	5.8750

Jeżeli zrobimy błąd literowy w nazwie zmiennej pojawia się komunikat:

Skrypt	Wynik

clc, clear
a = 2.5;
h = 4.7;
pole = a*H/2
	Undefined function or variable 'H'.
	Error in *plik* (line 4)
	pole=a*H/2
	>>

W wyrażeniu użyto niezdefiniowanej zmiennej ***H*** i wyrażenie nie jest obliczalne.

4.4.Komentarze w skrypcie

Pisząc skrypt zawierający wiele instrukcji, warto poświęcić trochę czasu dla opisania intencji stosowania kolejnych instrukcji. Służą do tego komentarze, dowolne teksty pisane **po znaku % do końca wiersza**. Nie są one analizowane przez interpreter skryptów. Komentarze zastosowano już w pierwszym skrypcie, w poprzednim podrozdziale.

Rolą komentarzy jest:

- objaśnienie użyteczności poszczególnych operacji,
- dezaktywacja instrukcji bez konieczności jej usuwania (można w każdej chwili przywrócić jej działanie usuwając znak %).

Skrypt	Wynik
clc, clear a = 2.5 % ustalenie danych h = 4.7 pole = a*h/2; % brak echa, bo na końcu znak ; pole % wypisujemy wartość zmiennej pole % pole2 = pole*3	a= 2.5 h= 4.7 pole= 5.87500

Działanie ostatniej instrukcji skryptu zostało dezaktywowane.

Przydatne w częstej pracy ze skryptami w edytorze mogą być skróty klawiaturowe:

CTRL+R – dodanie znaków komentarza (%) do zaznaczonych wielu wierszy skryptu,

CTRL+T – usunięcie komentarzy do zaznaczonych wielu wierszy skryptu.

5. Instrukcje wejścia i wyjścia

5.1. Interakcyjne wprowadzanie danych - instrukcja wejścia

Istnieje możliwość zatrzymania biegu programu w celu wprowadzenia danych przez użytkownika. Służy do tego funkcja *input*:

zmienna = **input**(*'tekst zachęty'*)

Z chwilą napotkania tej instrukcji program wstrzymuje wykonywanie, wypisuje na ekranie *Command Window* tekst zachęty będący argumentem funkcji i oczekuje na wprowadzenie odpowiedniej wartości liczbowej przez użytkownika. Po jej poprawnym wpisaniu i akceptacji klawiszem *Enter*, wartość ta przypisywana jest do zmiennej, następnie wykonywane są kolejne instrukcje skryptu.

Można zmodyfikować nasz skrypt:

```
clc, clear
a = input('Podaj podstawę trójkąta:');
h = input('Podaj wysokość trójkąta:');
pole = a*h/2
```

```
Podaj podstawę trójkąta:2.5
Podaj wysokość trójkąta:4.7
pole =
    5.8750
```

Należy zwracać uwagę na poprawność wprowadzanej liczby (całkowitej, dziesiętnej), w przeciwnym razie otrzymamy komunikat o błędzie.

Dla wprowadzenia danej typu tekstowego do zmiennej wykorzystuje się poniższą wersję funkcji *input*:

zmienna = **input**(*'tekst zachęty', 's'*)

Innym sposobem przekazania danych do programu jest odczyt z pliku (rozdz.8). Istnieje również możliwość wprowadzania danych tekstowych przy pomocy okienka dialogowego, o czym będzie mowa w rozdz.7.3, po poznaniu definicji struktury.

5.2. Wyprowadzanie tekstów informacyjnych i formatowanie wyników - instrukcje wyjścia

Do wyprowadzenia tekstu informacyjnego w oknie *Command Window* służy funkcja *disp*, o postaci:

disp (*'stała tekstowa'*)

Po zastosowaniu powyższego polecenia wypisywany jest w *Command Window* tekst argumentu funkcji *disp* i następuje zmiana wiersza.

Wartość zmiennej jest automatycznie wypisywana w *Command Window* jako echo instrukcji przypisania (chyba, że instrukcja przypisania zostanie zakończona średnikiem.

Można również, do wypisania wartości zmiennej wykorzystać poniższe zapisy:

zmienna
disp (*zmienna*)

Poniżej przykład ilustrujący możliwości wyświetlenia wartości zmiennej:

Kod		Wynik
clc,clear		a =
a = 5	% tu jest echo	5
a	% wypisz nazwę i wartość zmiennej	a =
disp(a)	% wypisz wartość zmiennej (bez jej nazwy)	5
a = 7;	% brak echa	5

Wyprowadzenie tekstów informacyjnych, wraz z obliczonymi wartościami zmiennych, umożliwia funkcja *fprintf* :

fprintf (*format*, *lista_zmiennych*)

gdzie *format* jest dowolnym tekstem (otoczonym apostrofami), który może zawierać specyfikacje pól. Specyfikacja pola rozpoczyna się znakiem %. W miejsce specyfikacji pola wpisywana będzie wartość kolejnych zmiennych z podanej listy, w odpowiednim formacie.

Najczęściej stosowane specyfikacje pól dla danych różnych typów:

%**s** - dla zmiennej typu tekstowego,

%**d** - dla zmiennej liczbowej typu całkowitego,

%**f** - dla zmiennej liczbowej dziesiętnej w formacie stałoprzecinkowym (6 miejsc dziesiętnych),

%**e** - dla zmiennej liczbowej dziesiętnej w formacie zmiennoprzecinkowym.

Dla specyfikacji **%f** można użyć zapisu:

%*m.n***f**

gdzie:

m - liczba całkowita, liczba znaków użytych do zapisu liczby (ewentualnie uzupełniona spacjami poprzedzającymi), jeżeli ***m*** będzie równe 0 lub niewielkie, liczba zajmie minimalną znaków, tyle, ile wynika z dokładności ***n***.

n - dokładność wyświetlania wartości liczbowej, czyli liczba miejsc dziesiętnych.

Dla specyfikacji **%s** istnieje możliwość zapisu w postaci:

%*m***s**

gdzie *m* - minimalna liczba znaków do zapisu tekstu (ewentualnie uzupełniana spacjami poprzedzającymi).

W tekście *format*, oprócz tekstu informacyjnego i specyfikacji formatu danych, mogą być także stosowane kody specjalne:

\n - kod zmiany wiersza,

\t - kod znaku tabulacji,

'' - (dwa apostrofy) - kod reprezentuje pojedynczy znak apostrofu ',

%% - (dwa znaki %) - kod reprezentuje pojedynczy znak % .

Przykład skryptu, z zastosowaniem poznanych zasad:

Skrypt	Wynik

clc, clear
disp('Obliczanie pola trójkąta');
a = input('Podaj podstawę trójkąta:');
h = input('Podaj wysokość trójkąta:');
pole = a*h/2;
fprintf('Pole trójkąta:%10.5f\n', pole)
fprintf(' Wykonane zadanie w 100%%\n');
	Obliczanie pola trójkąta
	Podaj podstawę trójkąta:2.5
	Podaj wysokość trójkąta:4.7
	Pole trójkąta: 5.87500
	Wykonane zadanie w 100%
	>>

Format **%10.5f** oznacza, że liczba dziesiętna w zmiennej *pole*, reprezentującej pole trójkąta, jest wyświetlona z dokładnością 5 miejsc dziesiętnych, poprzedzona jest zatem trzema spacjami (wszystkich pozycji formatu ma być 10, czyli: 5 cyfr dziesiętnych + 1 cyfra + 1 kropka + dodatkowo 3 spacje). Kod "\n" powoduje, że napis pojawia się w nowym wierszu. Jak pokazuje powyższy przykład, funkcja *fprintf* może być wykorzystana do wyprowadzenia samego tekstu, bez wartości obliczonych wyrażeń.

Przy pomocy funkcji *fprintf* można formatować wyświetlanie wartości wielu zmiennych w jednym wierszu:

Skrypt	Wynik
clc, clear a = 4.6; b = ' mm'; fprintf('a= %f %s \n', a, b)	a =4.600000 mm

lecz ten sam rezultat uzyskamy:

Skrypt	Wynik
clc, clear a = 4.6; b = ' mm'; fprintf('a= %f ', a) fprintf(' %s\n', b)	a =4.600000 mm

Kod *\n* może być użyty w dowolnej pozycji formatu, na przykład dla zmiany wiersza przed wyprowadzeniem tekstu:

```
fprintf(' \nRezultaty ')
```

6. Macierze

Odtąd zamieszczone przykłady oparto na trybie pracy wsadowej. Przedstawiana będzie treść opracowywanego skryptu w *M-pli*ku i rezultat jego wykonania.

6.1. Generowanie stałej macierzowej

Jak wspomniano, *MATLAB* jest środowiskiem zorientowanym macierzowo, dane są przechowywane w macierzach (tablicach wielowymiarowych), niezależnie od użytego typu danych.

Macierz jest złożonym elementem, zawierającym wiele danych wewnętrznych tego samego typu (własność zwana **homogenicznością**), a zatem macierz może zawierać wiele liczb lub wiele znaków (lub także wiele innych macierzy wewnętrznych, lecz takich samych rozmiarów i zawierających te same typy elementów), ale nie może zawierać równocześnie i liczb i tekstów.

Macierz (*matrix*) albo tablica (*array*) może być **jednowymiarowa** (**wektor**) lub **dwuwymiarowa**, prostokątna (wiersze, kolumny). Możliwe jest również tworzenie macierzy **wielowymiarowych**. W **każdym wymiarze** macierz posiada odpowiedni **rozmiar** (np. rozmiary: 5 wierszy i 4 kolumny).

Elementy macierzy są indeksowane, w celu określenia pozycji elementu w macierzy. Wektor posiada jeden indeks, macierz dwuwymiarowa dwa indeksy.

Indeksy elementów macierzy to kolejne liczby naturalne (1, 2, 3, ...itd, ...)

Odwołanie do elementu o indeksie zero, ujemnym lub indeksu spoza aktualnego rozmiaru zwróci błąd.

Macierz można utworzyć:

- wykorzystując operator konstrukcji **[ ]**,
- wykorzystując indeksowanie,
- wykorzystując funkcje tworzące macierze specjalne,
- wykonując operacje na innych macierzach, wynik operacji będzie przypisany do nowej macierzy,
- wczytując macierz z zewnętrznego pliku.

Stosując operator konstrukcji, liczby (także wyrażenia obliczeniowe) w wierszu macierzy oddziela się spacjami lub przecinkami, wiersze pomiędzy sobą średnikami (;) lub znakami nowego wiersza (*Enter*).

Przykładowo:

[1 2 3 ; 4 , 5.5, 6] definiuje macierz: $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5.5 & 6 \end{bmatrix}$

[2.3 ; -5 ; -7] definiuje macierz: $\begin{bmatrix} 2.3 \\ -5 \\ -7 \end{bmatrix}$

Należy uważać, aby:

- w zapisie liczby nie stosować wewnętrznej spacji (może to spowodować podział na dwie odrębne liczby),
- w każdym wierszu macierzy była taka sama liczba elementów (macierz musi być prostokątna).

Przykładowo, utworzone zostaną macierze o wymiarach 3×3 i przypisane do zmiennych:

Treść skryptu	Rezultat skryptu

clc, clear
T1 = [8, 1, 6 ; 3, 5, 7 ; 4 9 2]
%można też tak:
T2 = [1.2   3.8 5.3
          6.1 4.8   2.1
          1.6, -3.7, 3.4]
% lista zmiennych, rozmiar i typ
whos
	T1 =
	  8 1 6
	  3 5 7
	  4 9 2
	T2 =
	    1.2000 3.8000 5.3000
	    6.1000 4.8000 2.1000
	    1.6000 -3.7000 3.4000
	Name Size Bytes Class
	T1 3×3 72 double
	T2 3×3 72 double

Po zdefiniowaniu zawartości macierzy, w dowolnym miejscu skryptu można wyświetlić jej pełną zawartość, pisząc:

clc, clear
M = [8, 1 6 ; 3 5, 7 ; 4 9 2];
%.................
 	
% wiele instrukcji niżej
M   %wypisz nazwę i wartości elementów macierzy
%albo
disp(M)   %wypisz tylko wartości elementów macierzy
	M =
	  8 1 6
	  3 5 7
	  4 9 2
	
	  8 1 6
	  3 5 7
	  4 9 2

Poznana funkcja *fprintf* pozwala również na wypisanie wszystkich elementów macierzy:

clc,clear
M = [1,2; 3 ,4]   %macierz 2x2
fprintf('Elementy:%d\n', M);
	M =
	    1 2
	    3 4
	Elementy:1
	Elementy:3
	Elementy:2
	Elementy:4

Jak widać, w przypadku macierzy dwuwymiarowej, jej elementy są wyprowadzane kolumnami.

Łatwo można wygenerować wektor zawierający ciąg liczb w postępie arytmetycznym, wykorzystując poniższe zapisy:

wektor = [*pierwszy_element* **:** *przyrost* **:** *ograniczenie*]

wektor = *pierwszy_element* **:** *przyrost* **:** *ograniczenie*

lub

wektor = *pierwszy_element* **:** *ograniczenie* (domyślnie *przyrost* = 1)

Przykładowo:

clc, clear M = 1 :3 :12 N = 0 :0.1 :0.6 P = 0.5 :-0.2 : -0.5	M = 1 4 7 10 N = 0 0.1000 0.2000 0.3000 0.4000 0.5000 0.6000 P = 0.5 0.3000 0.1000 -0.1000 -0.3000 -0.5000

Jak pokazuje przykład, ciąg malejący dla kolejnych wartości elementów uzyskuje się dzięki ujemnej wartości *przyrostu*, lecz wówczas należy uważać, żeby *ograniczenie* miało wartość mniejszą niż *pierwszy_element*.

Pierwszy_element, *przyrost* i *ograniczenie* mogą być zmiennymi o uprzednio określonych wartościach lub obliczalnymi wyrażeniami:

clc, clear a = 6; b = 10; d = (b-a)/8; M = a:d:b	M = 6.00 6.50 7.00 7.50 8.00 8.50 9.00 9.50 10.00

Pominięcie *przyrostu* zakłada jego domyślną wartość równą 1:

clc, clear M = -5 : 5	M = -5 -4 -3 -2 -1 0 1 2 3 4 5

Można poniższym sposobem generować równocześnie dwa wiersze macierzy:

clc, clear M = [1:10; 11: 20]	M = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

uważając jednak, żeby w każdym wierszu zachować tę samą liczbę elementów.

Istnieje również możliwość użycia funkcji *linspace*, o postaci:

linspace (*pierwszy_element*, *ostatni_element*)

która generuje wektor 100-elementowy, wartości kolejnych elementów są w nim rozmieszczone równomiernie pomiędzy elementem pierwszym a ostatnim.

Można też zadecydować o liczbie elementów wektora:

linspace (*pierwszy_element*, *ostatni_element*, *N*)

gdzie *N* jest liczbą elementów wektora, jak w przykładzie:

clc, clear M = linspace(0, 10, 5) %100-elementowy wektor, co 1 K = linspace(1, 100); %wektor K nie jest wyświetlany	M = 0 2.5000 5.0000 7.5000 10.0000

MATLAB oferuje pewne funkcje specjalne dla działań na macierzach i tablicach, niektóre z nich przedstawia tabela 6.1.

Tabela 6.1. Funkcje generujące specjalne rodzaje macierzy

ones(*n*) lub **ones**(*n*, *m*)	utworzenie macierzy wypełnionej jedynkami ($n \times n$ lub $n \times m$),
zeros(*n*) lub **zeros**(*n*, *m*)	utworzenie macierzy wypełnionej zerami ($n \times n$ lub $n \times m$),

magic(*n*)	utworzenie "kwadratu magicznego" o zadanym wymiarze (te same sumy elementów w wierszach, kolumnach i obydwu przekątnych),
eye(*n*) lub **eye**(*n*, *m*)	macierz jednostkowa $n \times n$ lub $n \times m$, jedynki na przekątnej głównej, zera poza przekątną,
rand(*n*) **rand**(*n*, *m*)	tworzenie macierzy ($n \times n$ lub $n \times m$), liczby pseudolosowe z przedziału (0,1), o rozkładzie równomiernym.
randn(*n*) **randn**(*n*, *m*)	jak wyżej lecz rozkład normalny

Oto przykład działania funkcji *ones* i *magic*:

clc, clear M = ones(2, 5) N = magic(4)	M = 1 1 1 1 1 1 1 1 1 1 N = 16 2 3 13 5 11 10 8 9 7 6 12 4 14 15 1

Generowanie macierzy losowej o elementach całkowitych wykonywane jest według przepisu:

macierz = round(rand(*n*, *m*)**szerokość_przedziału* + *przesunięcie*)

Przykładowy skrypt wykonuje losowanie macierzy o rozmiarach 3×2, zawierającej liczby całkowite, należące do przedziału (-50, 50):

clc, clear M = round(rand(3, 2)*100-50)	M = 31 41 41 13 -37 -40

Każde wykonanie skryptu generuje inną macierz losową.

Można też do losowania liczb całkowitych stosować funkcję *randi*:

macierz = randi([*A*, *B*], *m*, *n*)

gdzie A i B to granice przedziału, n i m to rozmiary macierzy.

Przypisanie do zmiennej łańcucha znaków generuje wektor wierszowy z pojedynczymi znakami w komórkach (włączając spacje). A zatem poniższe zapisy są równoważne:

```
napis1 = 'MATLAB'
napis2 = ['MATLAB']
napsi3 = ['M', 'A', 'T', 'L', 'A', 'B']
```

clc, clear napis = 'Politechnika Rzeszowska' %pokaż zmienne whos	napis = 'Politechnika Rzeszowska' Name Size Bytes Class Attributes napis 1x23 46 char

Utworzenie macierzy homogenicznej z napisami o różnej długości nie jest możliwe, umożliwiają to tablice łańcuchów (*string array*):

clc, clear napis1 = "Politechnika Rzeszowska"; napis2 = "Rzeszów"; M = [napis1 napis2]	M = 1×2 string array "Politechnika Rzeszowska" "Rzeszów"

W tablicy umieszczono łańcuchy znaków o różniej długości, stałe tekstowe są zawarte w cudzysłowach.

W celu przechowywania danych różnych typów stosuje się tablice komórkowe (rozdz.7.2).

6.2. Dostęp do elementu macierzy i fragmentu macierzy

Chcąc wykorzystać w wyrażeniu wartość wybranego elementu macierzy, podaje się jego indeks lub indeksy (współrzędne) - bezpośrednio po nazwie zmiennej macierzowej, w nawiasach okrągłych, według zasady:

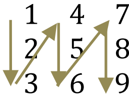
zmienna_wektorowa (*indeks*)

zmienna_macierzowa (*indeks_wiersza* , *indeks kolumny*)

Możliwe jest również wykorzystanie pojedynczego indeksu dla macierzy dwuwymiarowej:

zmienna_macierzowa (*indeks*)

gdzie numerowanie kolejnych elementów tylko jedną wartością indeksu odbywa się kolumnami, na przykład dla macierzy o rozmiarach 3×3 według zasady:



Wyjaśnia to poniższy przykład:

clc, clear N = magic(4) x = N(2, 2) %lub alternatywnie: y = N(6) fprintf('Element (4,3):%d\n', N(4,3))	N = 16 2 3 13 5 (11) 10 8 9 7 6 12 4 14 15 1 x = 11 y= 11 Element (4,3): 15

Błędem zakończy się próba wykorzystania w wyrażeniu elementu macierzy, posiadającego indeks lub indeksy spoza aktualnego rozmiaru macierzy.

W powyższym przykładzie, gdyby kolejną instrukcję stanowiło przypisanie:

```
z = sin(N(5, 1))
```

ukaże się komunikat o błędzie:

```
Index in position 1 exceeds array bounds (must not exceed 4).
Error in ==> macierze at nr_linii
```

Indeks w pozycji przekracza rozmiary (nie może przekraczać 4).

Błąd w 'nr_linii' wierszu pliku

Istnieje również możliwość wyodrębnienia z macierzy określonego, prostokątnego fragmentu i przypisywania do innej, mniejszej macierzy. Stosuje się tu wartość konkretnego indeksu lub zapis dla zakresu indeksów:

n	- wiersz (kolumna) o podanym indeksie
n **:** *m*	- wiersze (kolumny) w zakresie od *n*-tego do *m*-tego elementu,
n **: end**	- wiersze (kolumny) od *n*-tego do ostatniego elementu,
:	- wszystkie wiersze (kolumny).

Przykłady z wyjaśnieniami:

```
M( 2,2:4)        % drugi wiersz, od drugiej do czwartej kolumny,
M( 4,:)          % wszystkie kolumny czwartego wiersza,
M( 5:end, 3)     % od piątego do ostatniego wiersza trzeciej kolumny.
```

Metodę tę ilustruje poniższy skrypt:

Skrypt	Wynik
clc, clear %Macierz magiczna M = magic(4) %"Wycinamy" z tej macierzy fragment N = M(2:3, 2:end) % .. cały drugi wiersz W = M(2, :)	M = 16 2 3 13 5 11 10 8 9 7 6 12 4 14 15 1 N = 11 10 8 7 6 12 W = 5 11 10 8

Wykorzystując tę metodę można dopisywać nowe wiersze do macierzy:

Skrypt	Wynik
clc, clear M = [1 2 3]; %wektor M(2, :) = [4 5 6] %drugi wiersz	M = 1 2 3 4 5 6

Dwukropek jako pojedynczy indeks służy też do przekształcenia macierzy dwuwymiarowej w wektor:

clc, clear M = [9 7 ; 6 12] M2 = M(:)	M = 9 7 6 12 M2 = 9 7 6 12

Redukcję macierzy (usunięcie wiersza, kolumny lub zakresów wierszy i kolumn) wykonać można stosując macierz pustą: [ ] .

Przykładowo:

clc, clear M = [1 1 1 1 ; 2 2 2 2 ; 3 3 3 3] M(2, :) = [ ]; %usuwamy drugi wiersz M	M = 1 1 1 1 2 2 2 2 3 3 3 3 M = 1 1 1 1 3 3 3 3

Oczywiście wolno usuwać tylko kompletne wiersze lub kolumny.

Rozmiary macierzy mogą być łatwo powiększane. Dopuszczalne jest definiowanie nowego elementu o wartościach indeksów poza aktualnymi rozmiarami macierzy, spowoduje to odpowiednie uzupełnienie macierzy zerami.

Ilustruje to poniższy przykład:

clc, clear M = [1 1; 1 1] %Tworzymy element poza macierzą M(4, 4) = 100	M = 1 1 1 1 M = 1 1 0 0 1 1 0 0 0 0 0 0 0 0 0 100

Przykład uproszczonej metody dopisywania do wektora nowych elementów:

clc, clear M = [ ] %pusty wektor M(end+1) = 10 M(end+1) = 5 M(end+1) = 1000	M = [ ] M = 10 M = 10 5 M = 10 5 1000

6.3.Operacje macierzowe i tablicowe

6.3.1. Łączenie macierzy

Do połączenia macierzy można wykorzystać konstruktor macierzy, umieszczając macierze składowe we nawiasach kwadratowych [ ].

Poniżej przykłady:

clc, clear M = ones(4); N = 3*ones(4); W = [M, N]	W = 1 1 1 1 3 3 3 3 1 1 1 1 3 3 3 3 1 1 1 1 3 3 3 3 1 1 1 1 3 3 3 3

clc, clear M = ones(2,3); N = zeros(3,3); W = [M ; N]	W = 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0

Łącząc macierze wzdłuż kolumn, należy uważać na zgodność liczby wierszy w macierzach, natomiast dopisując drugą macierz w kolejnych wierszach trzeba zachować tę samą liczbę kolumn. Powyższym sposobem można łączyć więcej niż dwie macierze.

W przypadku niezgodności wymiarów pojawi się komunikat o błędzie:

```
Error using vertcat. Dimensions of arrays being concatenated are not consistent.
```

Błąd przy użyciu vertcat. Wymiary łączonych tablic nie są spójne.

6.3.2. Suma macierzy

Znak + to operator sumowania dwóch (lub więcej) macierzy. Zgodnie z zasadami matematyki w wyrażeniu $A+B$ macierze A i B muszą mieć **te same wymiary i te same rozmiary** w każdym wymiarze. Sumowanie odbywa się pozycyjnie - sumowane są elementy o tych samych indeksach wektorów lub o tej samej parze indeksów w macierzach dwuwymiarowych.

Oczywiście można sumować wiele macierzy wyrażeniem: $A+B+C$ itd., każda macierz będąca składnikiem sumy musi spełniać powyższe wymagania.

Podobne zasady dotyczą odejmowania macierzy.

Przykład dodawania małych macierzy:

clc, clear M = [1 2 ; 3 4] N = [5 6 ; 7 8] W = M+N	M = 1 2 3 4 N = 5 6 7 8 W = 6 8 10 12

Podstawowym wyjątkiem dopuszczającym niezgodność wymiarów i rozmiarów dwóch macierzy, które są składnikami sumy, jest dodawanie macierzy jednoelementowej (skalar) do dowolnej macierzy. Wykonane tu zostanie dodanie elementu macierzy jednowymiarowej do każdego elementu drugiej macierzy:

clc, clear M = [1 1; 2 2] N = 3 W = M+N	M = 1 1 2 2 N = 3 W = 4 4 5 5

W najnowszych wersjach Matlaba wprowadzono pewne rozszerzenia dopuszczalności obliczania sumy dwóch macierzy. Są to operacje:

- suma dwóch wektorów kolumnowego i wierszowego (operacja przemienna), obydwa wektory o dowolnej długości – sumowanie odbywa się według poniższych zasad:

$$[a_1 \quad a_2 \quad a_3 \ \dots] + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \dots \end{bmatrix} = \begin{bmatrix} a_1+b_1, & a_2+b_1, & a_3+b_1 & \dots \\ a_1+b_2, & a_2+b_2, & a_3+b_2 & \dots \\ a_1+b_3, & a_2+b_3, & a_3+b_3 & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \dots \end{bmatrix} + [b_1 \quad b_2 \quad b_3 \ \dots] = \begin{bmatrix} a_1+b_1, & a_1+b_2, & a_1+b_3 & \dots \\ a_2+b_1, & a_2+b_2, & a_2+b_3 & \dots \\ a_3+b_1, & a_3+b_2, & a_3+b_3 & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

- suma dowolnej macierzy dwuwymiarowej oraz:

- wektora kolumnowego o tej samej liczbie wierszy co macierz dwuwymiarowa,
- wektora wierszowego o tej samej liczbie kolumn co macierz dwuwymiarowa,

Operacje są również przemienne i wykonywane są według zasad, które można opisać ogólnie:

$$\begin{bmatrix} a_{11}, & a_{12}, & a_{13} \dots \\ a_{21}, & a_{22}, & a_{23} \dots \\ a_{31}, & a_{32}, & a_{33} \dots \\ \dots & \dots & \dots & \dots \end{bmatrix} + [b_1 \quad b_2 \quad b_3 \dots] = \begin{bmatrix} a_{11}+b_1, & a_{12}+b_2, & a_{13}+b_3 \dots \\ a_{21}+b_1, & a_{22}+b_2, & a_{23}+b_3 \dots \\ a_{31}+b_1, & a_{32}+b_2, & a_{33}+b_3 \dots \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

$$\begin{bmatrix} a_{11}, & a_{12}, & a_{13} \dots \\ a_{21}, & a_{22}, & a_{23} \dots \\ a_{31}, & a_{32}, & a_{33} \dots \\ \dots & \dots & \dots & \dots \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \dots \end{bmatrix} = \begin{bmatrix} a_{11}+b_1, & a_{12}+b_1, & a_{13}+b_1 \dots \\ a_{21}+b_2, & a_{22}+b_2, & a_{23}+b_2 \dots \\ a_{31}+b_3, & a_{32}+b_3, & a_{33}+b_3 \dots \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

$$[a_1 \quad a_2 \quad a_3 \dots] + \begin{bmatrix} b_{11}, & b_{12}, & b_{13} \dots \\ b_{21}, & b_{22}, & b_{23} \dots \\ b_{31}, & b_{32}, & b_{33} \dots \\ \dots & \dots & \dots & \dots \end{bmatrix} = \begin{bmatrix} a_1+b_{11}, & a_2+b_{12}, & a_3+b_{13} \dots \\ a_1+b_{21}, & a_2+b_{22}, & a_3+b_{23} \dots \\ a_1+b_{31}, & a_2+b_{32}, & a_3+b_{33} \dots \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \dots \end{bmatrix} + \begin{bmatrix} b_{11}, & b_{12}, & b_{13} \dots \\ b_{21}, & b_{22}, & b_{23} \dots \\ b_{31}, & b_{32}, & b_{33} \dots \\ \dots & \dots & \dots & \dots \end{bmatrix} = \begin{bmatrix} a_1+b_{11}, & a_1+b_{12}, & a_1+b_{13} \dots \\ a_2+b_{21}, & a_2+b_{22}, & a_2+b_{23} \dots \\ a_3+b_{31}, & a_3+b_{32}, & a_3+b_{33} \dots \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

Wykonanie przykładów pozostawiamy czytelnikowi.

6.3.3. Iloczyn macierzy

A*B	* *(gwiazdka)* to operator mnożenia macierzowego (*matrix opreator*) - iloczyn *Cauchy'ego* - operacja jest dozwolona gdy **macierz *A* ma tyle kolumn ile macierz B ma wierszy** - obliczane są sumy iloczynów wierszy macierzy *A* przez kolumny macierzy *B*,
A.*B	.* (*kropka gwiazdka*) to dwuznakowy operator (.*) mnożenia tablicowego (elementowego – *array opreator*) - iloczyn *Hadamarda* - operacja jest dozwolona gdy macierze ***A* i *B* mają te same wymiary i rozmiary**, powstaje macierz z pojedynczych iloczynów elementów o tych samych indeksach.

Częstym błędem jest niezrozumienie matematycznej zasady mnożenia macierzowego. Oto prosty schemat wyjaśniający tę regułę:

$$\begin{bmatrix} a_{11}, & a_{12}, & a_{13}, & \dots \\ a_{21}, & a_{22}, & a_{23}, & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix} * \begin{bmatrix} b_{11}, & b_{12,} & \dots \\ b_{21}, & b_{22}, & \dots \\ b_{31}, & b_{32}, & \dots \\ \dots & \dots & \dots \end{bmatrix} =$$

$$\begin{bmatrix} a_{11}*b_{11}+a_{12}*b_{21}+a_{13}*b_{31}+\cdots, & a_{11}*b_{12}+a_{12}*b_{22}+a_{13}*b_{32}+\cdots, & \dots \\ a_{21}*b_{11}+a_{22}*b_{21}+a_{23}*b_{31}+\cdots, & a_{21}*b_{12}+a_{22}*b_{22}+a_{23}*b_{32}+\cdots, & \dots \\ \dots & \dots & \dots \end{bmatrix}$$

Jeżeli A jest macierzą $n \times m$, a B macierzą $m \times p$, wzór na dowolny element macierzy wynikowej jest następujący:

$$w_{i,j} = \sum_{k=1}^{m} A_{i,k} \, B_{k,j}$$

dla każdej pary i, j - pamiętając że i=1..n, j=1..p.

W przypadku błędu niewłaściwych rozmiarów macierzy w iloczynie, pojawi się komunikat:

Inner matrix dimensions must agree.

Wewnętrzne wymiary macierzy muszą się zgadzać

lub

Incorrect dimensions for matrix multiplication.

Niepoprawne wymiary mnożenia macierzy

Podobna informacja pojawi się przy niespełnieniu warunków mnożenia tablicowego.

Poniżej przykład skryptu dla mnożenia macierzowego:

clc, clear M = [1 2; 3 4] N = [2 0; 4 -1] W = M*N	M = 1 2 3 4 N = 2 0 4 -1 W = 10 -2 22 -4

i mnożenia tablicowego:

clc, clear M = [1 2 ; 3 4; 6 -8] N = [2 0; 4 -1; 9 0] Wtab = M.*N	M = 1 2 3 4 6 -8 N = 2 0 4 -1 9 0 Wtab = 2 0 12 -4 54 0

Mnożenie macierzowe nie jest przemienne, jeżeli dozwolone jest mnożenie macierzy $A*B$, to mnożenie $B*A$ może być niedopuszczalne, a jeżeli jest dozwolone, to zwykle daje inny rezultat. Przykładowo:

clc, clear
M = [1 2; 3 4]
N = [2 ; 4]
W = M*N
V = N*M %nie wolno!
	M =
	    1 2
	    3 4
	N =
	    2
	    4
	W =
	    10
	    22
	Error using *
	Incorrect dimensions for matrix multiplication. Check that the number of columns in the first matrix matches the number of rows in the second matrix. To perform elementwise multiplication, use '.*'.
	Error in p6_3e (line 5)
	V = N*M %nie wolno!

W przypadku mnożenia macierzy przez stałą (skalar, czyli macierz jednoelementową), można alternatywnie stosować operator macierzowy ***** lub tablicowy **.***.

Działania symboliczne na macierzach (rozdz. 12.5) pozwolą na powtórzenie i ugruntowanie wiedzy na temat operacji macierzowych i tablicowych.

6.3.4.Potęgowanie macierzy

A^*n*	**^** (*daszek*) - to operator potęgowania macierzowego, • **wykładnik potęgi *n* powinien być skalarny** (pojedyncza wartość, czyli w *MATLAB-ie* macierz jednoelementowa), • potęgowana macierz musi być **kwadratowa**, • jeżeli *n* jest liczbą naturalną to: *A*^2 jest tożsame z *A***A*, *A*^3 jest tożsame z *A***A***A* itd., • *A*^-1 to macierz odwrotna, • *A*^(1/2) to taka macierz *B*, że B^2=*A*, • inne wykładniki niecałkowite (dodatnie i ujemne) - operacje rzadko stosowane opisane złożonymi wzorami, rezultaty często mają wartości zespolone.
A .^ B	**.^** (*kropka daszek*) - to dwuznakowy operator potęgowania tablicowego (elementowego) - **macierz potęgowana może być dowolna**; - jeżeli wykładnik potęgi jest skalarem, każdy element podnoszony jest do tej potęgi; - jeżeli wykładnik potęgi jest macierzą, to musi być ona tych samych rozmiarów co macierz potęgowana i wówczas wynikiem jest macierz powstała z elementów macierzy *A* podniesionych do odpowiednich potęg - elementów macierzy *B* o tych samych indeksach. Można też dla potęgowania tablicowego wykorzystywać funkcję *power.*

Przykład ilustrujący podnoszenie macierzy kwadratowej do potęgi całkowitej:

clc, clear M = [2.5 4; -3 0.2] N = M^2 % równoważne z…. W = M*M %potęgowanie tablicowe X1 = power(M,3) X2 = M.^3	M = 2.5000 4.0000 -3.0000 0.2000 N = -5.7500 10.8000 -8.1000 -11.9600 W = -5.7500 10.8000 -8.1000 -11.9600 X1 = 15.6250 64.0000 -27.0000 0.0080 X2 = 15.6250 64.0000 -27.0000 0.0080

oraz przykład potęgowania tablicowego dwóch wektorów:

clc, clear A = [2 2 2] B = [2 3 4] W = A.^B	A = 2 2 2 B = 2 3 4 W = 4 8 16

6.3.5. Dzielenie macierzy

Poniższe zestawienie wyjaśnia działanie operatorów dzielenia macierzy.

A/B	**/** (*slash*) to operator dzielenia macierzowego prawostronnego, **macierze *A* i *B* muszą mieć tyle samo kolumn, operacja odpowiada rozwiązaniu równania *x**B=A***, (czyli x=A*B^-1 jeżeli B jest macierzą kwadratową),
A./B	**./** (*kropka slash*) to dwuznakowy operator dzielenia tablicowego (elementowego) prawostronnego - **obie macierze muszą być tych samych wymiarów i rozmiarów**, obliczenie ilorazów elementów o tych samych indeksach,
A\B	**** (*backslash*) to operator dzielenia macierzowego lewostronnego, **macierze *A* i *B* muszą mieć tyle samo wierszy**, odpowiada rozwiązaniu równania *A**x=*B* jeżeli macierz *A* jest kwadratowa *nxn*, a *B* wektorem kolumnowym o *n* wierszach, (stosowane często zamiast operacji ***A*^-1****B*** (jeżeli A jest macierzą kwadratową)
A.\B	**.** (*kropka-backslash*) to dwuznakowy operator dzielenia tablicowego lewostronnego - **obie macierze muszą być tych samych wymiarów i rozmiarów**, elementowe (tablicowe) obliczenie odwrotności ilorazów elementów o tych samych indeksach

W przypadku dzielenia macierzy przez skalar (jedną liczbę) należy zastosować operatory prawostronne (**/** lub **./**).

Oto przykład dzielenia macierzowego prawostronnego dwóch macierzy wraz ze sprawdzeniem, czy jest ono równoważne z mnożeniem przez macierz odwrotną (jeżeli dzielnik jest macierzą kwadratową):

```
clc
clear
% dzielenie prawostronne
A = [1  3  5  3]
B = [ 2  0  6  7
      2  3  8  1
      5 -1  5  0
      2  4  3  7]
C = A/B
%sprawdzenie
D = A*B^-1
czyA=C*B
```

```
A =
    1    3    5    3

B =
    2    0    6    7
    2    3    8    1
    5   -1    5    0
    2    4    3    7

C =
   0.0693    0.5743   -0.1683    0.2772

D =
   0.0693    0.5743   -0.1683    0.2772

czyA =
   1.0000    3.0000    5.0000    3.0000
```

oraz przykład dzielenia macierzowego lewostronnego:

```
clc, clear
A = [1 2 3;  2 1 0; -2 6 7 ]
B = [4; 3 ; 4]
C = A\B

% jeżeli A jest macierzą kwadratową,

% czy jest równoważne z:
D = A^-1*B
% można sprawdzić, że A*C=B
```

```
A =
    1    2    3
    2    1    0
   -2    6    7

B =
    4
    3
    4

C =
   1.3333
   0.3333
   0.6667

D =
   1.3333
   0.3333
   0.6667
```

UWAGA: Nowe wersje MATLAB-a dopuszczają dodawanie oraz mnożenie, dzielenie i potęgowanie tablicowe dwóch macierzy (jeżeli jedna jest wektorem o pewnej liczbie

elementów – jest to jednak pewne uproszczenie raczej informatyczne niż matematyczne.

6.3.6. Macierz transponowana i sprzężona

Wyznaczenie macierzy transponowanej (lub sprzężonej, w przypadku gdy elementy macierzy mają wartości zespolone) wykonuje się następująco:

M.' lub **transpose**(M)	**.'** (*kropka apostrof*) – dwuznakowy operator transpozycji macierzy - zamiana miejscami wierszy i kolumn (obrócenie macierzy względem przekątnej głównej)
M'	**'** (*apostrof*) – operator sprzężenia macierzy o elementach zespolonych (zamiana wierszy i kolumn oraz zmiana znaku części urojonych)

Poniżej przykłady zastosowania:

<table>
<tr><td>

```
clc, clear
M = [2  4  1 6; 2  0  -9  3]
disp(' Transponujemy macierz');
Mt = M.'
```

</td><td>

```
M =
    2    4    1    6
    2    0   -9    3

Transponujemy macierz
Mt =
    2    2
    4    0
    1   -9
    6    3
```

</td></tr>
<tr><td>

```
clc, clear
N = [ 2+1i, 3-2i;-1+1i  4+3i ]
% macierz sprzężona
Nsp = N'
%macierz transponowana
Ntr = N.'
```

</td><td>

```
N =
   2 + 1i        3  - 2i
  -1 + 1i        4 + 3i

Nsp =
   2 -  1i      -1 - 1i
   3 + 2i        4 - 3i

Ntr =
   2 + 1i      -1 + 1i
   3 -  2i       4 + 3i
```

</td></tr>
</table>

6.3.7. Wyrażenia logiczne wykonywane na macierzach

Dla dwóch macierzy o tych samych wymiarach i rozmiarach można tworzyć wyrażenia logiczne. Wynikiem operacji jest macierz zer i jedynek logicznych, jako rezultat wyrażenia logicznego, wyznaczonego dla par elementów o tych samych indeksach. Możliwe też jest wykorzystanie macierzy jednoelementowej (skalar) w wyrażeniu logicznym.

Oto przykład, który ilustruje rezultaty porównania dwóch macierzy i porównania elementów macierzy do liczby:

<table>
<tr>
<td>

```
clc,clear
%porównanie macierzy
A = [0  3  6  9 11]
B = [8  2 -4  3  0]
X = A>B&B>0

%porównanie z wartością liczbową
Y = A>5
```

</td>
<td>

```
A =
   0    3    6    9   11

B =
   8    2   -4    3    0

X =
  1×5 logical array
   0   1   0   1   0

Y =
  1×5 logical array
   0   0   1   1   1
```

</td>
</tr>
</table>

6.3.8. Funkcje działające na macierzach liczbowych

MATLAB zawiera wiele funkcji dla operacji na macierzach liczbowych. Tabela 6.2 zestawia najczęściej używane, wbudowane funkcje *MATLAB-a*, które wspomagają operacje na macierzach.

Tabela 6.2. Wybrane funkcje operacji na macierzach

inv(*A*) lub *A*^-1	macierz odwrotna do macierzy kwadratowej
det(A)	wyznacznik macierzy kwadratowej
transpose(*A*) lub A.'	macierz transponowana (obrót o 180 stopni względem przekątnej głównej)
eig(*A*)	wektor kolumnowy wartości własnych macierzy kwadratowej
rot90(*A*)	obrót macierzy o 90 stopni w prawo (1-szy wiersz staje się ostatnią kolumną)
length(*A*)	zwraca największy z rozmiarów macierzy (jeżeli *A* jest wektorem, zwracana jest liczba jego elementów)
sum (*A*)	wektor wszystkich sum elementów w kolumnach macierzy *A*, jeżeli *A* jest wektorem, to zostaje obliczona suma jego elementów
sum(sum(*A*)) lub **sum**(*A*, 'all')	suma elementów macierzy dwuwymiarowej - suma sum kolumn suma wszystkich elementów dowolnej macierzy - 'all'

max(*A*)	wektor elementów maksymalnych w każdej kolumnie macierzy *A* (jeżeli *A* jest wektorem, zwracany jest jego największy element)
max(max(*A*)) lub **max**(*A*,[ ], 'all')	największy element w macierzy dwuwymiarowej
min(*A*)	wektor elementów minimalnych w kolumnach macierzy *A*, jeżeli *A* jest wektorem, zwracany jest jego najmniejszy element
min(min(*A*)) lub **min**(*A*,[ ], 'all')	najmniejszy element w macierzy dwuwymiarowej
sort(*A*)	sortowanie każdej kolumny macierzy *A* rosnąco, też sortowanie rosnąco wektora wierszowego
diag(*A*)	tworzy wektor kolumnowy z przekątnej głównej macierzy *A*
numel(*A*)	liczba elementów macierzy *A*
reshape(*A*,*n*,*m*)	rekonfiguracja macierzy *A* - tworzy nową macierz o rozmiarach $n \times m$
size(*A*)	zwraca wektor rozmiarów macierzy w każdym wymiarze dla macierzy dwuwymiarowej: **[*liczba_wierszy liczba kolumn*]**
horzcat (*A*, *B, ...*)	połączenie kilku macierzy poziomo (dopisywanie kolumn, macierze składowe muszą mieć tyle samo wierszy)
vertcat *(A*, *B, ...*)	łączenie kilku macierzy pionowo (dopisywanie wierszy, macierze składowe muszą mieć tyle samo kolumn)
find(*A*)	zwraca wektor indeksów elementów niezerowych macierzy
find(~*A*)	zwraca wektor indeksów elementów zerowych macierzy
find(*warunek*)	zwraca wektor indeksów elementów macierzy *A* , które spełniają *warunek*, Przykłady warunków: A==*liczba* A>*liczba* A~=*liczba* A>*liczba*&A<*liczba* i inne Dla macierzy dwuwymiarowej macierzy A, znalezione indeksy stanowią kontynuację kolejnymi kolumnami
prod(*A*)	zwraca iloczyn wszystkich elementów wektora, dla macierzy wynikiem jest wektor iloczynów dla każdej kolumny

unique(*A*)	zwraca wektor unikalnych wartości macierzy *A*
tril(*A*)	macierz trójkątna dolna (zeruje elementy powyżej przekątnej głównej macierzy *A*) - ang. *triangle low*
triu(*A*)	macierz trójkątna górna (zeruje elementy poniżej przekątnej głównej macierzy *A*) - ang. *triangle up*

Przedstawiono tu najprostsze postacie wykorzystania powyższych funkcji. Wiele z nich posiada dodatkowe argumenty, uwzględniające różne kryteria operacji. Zainteresowanych odsyłamy do dokumentacji.

Poniżej zamieszczono przykłady ilustrujące zastosowanie wybranych funkcji w skryptach oraz rezultaty ich wykonania.

Macierz odwrotna do macierzy kwadratowej spełnia równanie:

$$M * M^{-1} = I$$

gdzie *I* to macierz jednostkowa, czyli macierz wypełniona jedynkami na przekątnej głównej i zerami poza przekątną.

Poniżej przykład obliczenia macierzy odwrotnej do macierzy kwadratowej dwoma sposobami - potęgowaniem macierzowym lub z wykorzystaniem funkcji *inv*:

```
clc, clear
M = [2.5   4   1
       -3   0.2  2
        5  1.4   9]
Mo1 = inv(M)

% lub alternatywnie
Mo2 = M^-1
```

```
M =
    2.5000    4.0000    1.0000
   -3.0000    0.2000    2.0000
    5.0000    1.4000    9.0000

Mo1 =
   -0.0071   -0.2466    0.0556
    0.2637    0.1247   -0.0570
   -0.0371    0.1176    0.0891

Mo2 =
   -0.0071   -0.2466    0.0556
    0.2637    0.1247   -0.0570
   -0.0371    0.1176    0.0891
```

oraz sprawdzenie, czy macierz pomnożona przez własną macierz odwrotną daje w wyniku macierz jednostkową:

```
clc, clear
M = [2.5  4 2; 1 -3  0.2; 4 -3 3.5]
K = M*inv(M)
```

```
M =
    2.5000    4.0000    2.0000
    1.0000   -3.0000    0.2000
    4.0000   -3.0000    3.5000
K =
    1.0000         0         0
         0    1.0000         0
         0         0    1.0000
```

Macierz osobliwa to macierz o wyznaczniku równym zero (przypadek ten zachodzi, jeżeli występuje zależność liniowa wierszy lub kolumn).

Oto przykład ilustrujący taki przypadek:

clc, clear M = [1 2 3; 2 4 6; 1 1 1] wyznacznik= det(M) W = inv(M)	M = 1 2 3 2 4 6 1 1 1 wyznacznik = 0 Warning: Matrix is singular to working precision. W = Inf Inf Inf Inf Inf Inf Inf Inf Inf

Pojawiło się ostrzeżenie (ang. *warning*). Dwa pierwsze wiersze macierzy *M* są w zależności liniowej, elementy macierzy odwrotnej mają wartość nieokreśloną *Inf* (*Infinity* - nieskończoność).

Wcześniej przedstawiono przykład wyznaczenia macierzy transponowanej przy pomocy dwuznakowego operatora (**.'**). Funkcja *transpose* działa identycznie.

clc, clear M = [1 1 1; 2 2 2 ; 3 3 3] %transpozycja Mt = transpose(M)	M = 1 1 1 2 2 2 3 3 3 Mt = 1 2 3 1 2 3 1 2 3

Przykłady użycia funkcji *size, sum*, *max* i *min*:

clc, clear M=[2 4 1; 2 0 -9] rozmiary = size(M) %sumy kolumn sumyKol = sum(M) % całkowita suma suma = sum(sum(M)) % zamiennie z %suma = sum(M, 'all') %maksumum i minimum maks = max(max(M)) % zamiennie z %maks = max(M,[ ], 'all') minim = min(min(M)) % zamiennie z %minim = min(M,[ ], 'all')	M = 2 4 1 2 0 -9 rozmiary 2 3 sumyKol = 4 4 -8 suma = 0 maks = 4 minim = -9

Należy zwrócić uwagę na zagnieżdżanie funkcji (*sum*, *min*, *max*) w przypadku macierzy dwuwymiarowej lub wykorzystanie drugiego argumentu: 'all'.

Prosty przykład zastosowania funkcji *diag*, używana w celu wydzielenia z macierzy dwuwymiarowej wektora przekątnej:

clc, clear M = round(10*rand(4)) %wektor przekątnej głównej d = diag(M)	M = 2 8 1 6 5 1 4 3 4 4 2 7 5 10 1 8 d = 2 1 2 8

Stosując funkcję *reshape* dla macierzy dwuwymiarowej należy uważać, aby iloczyn nowych rozmiarów był zgodny z iloczynem rozmiarów pierwotnych (ta sama liczba elementów w macierzy pierwotnej i wynikowej).

Przykładowo:

clc, clear
%Macierz 2x4
M = [2 4 1 2; 2 0 -3   8]
rozmiary_M=size(M)
 	
%zmieniamy kształt
Mw = reshape(M, 1, 8)
rozmiary_Mw = size(Mw)
	M =
	    2 4 1 2
	    2 0 -3 8
	rozmiary_M =
	  2 4
	Mw =
	    2 2 4 0 1 -3 2 8
	rozmiary_Mw =
	  1 8

Do łączenia macierzy można wykorzystać standardowe funkcje *horzcat* i *vertcat*, zmiennie z przedstawioną w poprzednim rozdziale metodą:

$W_1=[M_1, N_1]$ i $W_2=[M_2; N_2]$

Przykład zastosowania funkcji *horzcat*:

clc, clear
M = [1 1; 1 1]
N = [2 2; 2 2]
P = horzcat(M, N)
%lub
P = [M, N]
	M =
	    1 1
	    1 1
	N =
	    2 2
	    2 2
	P =
	    1 1 2 2
	    1 1 2 2

Rezultat wykorzystania funkcji *find*:

clc, clear
A = [2 4 0 2 2 0 0   8]
% indeksy elementów z przedziału [2, 3]
n = find(A>=2&A<=3)
% indeksy elementów zerowych
m = find(~A)
	A =
	    2 4 0 2 2 0 0 8
	n =
	    1 4 5
	m =
	    3 6 7

oraz przykład zastosowania funkcji *unique*:

clc, clear
A = [2 4 0 2 2 0 0   8]
%wektor bez powtórzeń
B = unique(A)
	A =
	    2 4 0 2 2 0 0 8
	
	B =
	    0 2 4 8

6.3.9. Przykłady zastosowań działań macierzowych

Tabelaryzacja wartości funkcji

Często zachodzi konieczność przygotowania danych dyskretnych zmiennej niezależnej w wektorze i żądania obliczenia określonej funkcji na tych danych i umieszczenia ich

w innym wektorze, na przykład w celu narysowania wykresu. Należy obliczyć wartości funkcji operującej na elementach macierzy (najczęściej wektora).

Przykładowo:

clc, clear X = 0:0.2:1 Y = exp(X)	X = 0 0.2000 0.4000 0.6000 0.8000 1.0000 Y = 1.0000 1.2214 1.4918 1.8221 2.2255 2.7183

Można scalić obydwa wektory w jednej, dwuwierszowej macierzy.

clc, clear X = 0:0.2:1; M = [X; exp(X)]	M = 0 0.2000 0.4000 0.6000 0.8000 1.0000 1.0000 1.2214 1.4918 1.8221 2.2255 2.7183

Przy złożonych wyrażeniach, wykonywanych na serii danych umieszczonych w wektorze, należy pamiętać o używaniu operatorów:

.* - tablicowego (elementowego) mnożenia,
./ - tablicowego dzielenia,
.^ - tablicowego potęgowania.

Przykładowo dla funkcji:

$$f(x) = \frac{\sin x \ cos^2 x}{1+\sqrt[3]{x+1}}$$

po zdefiniowaniu serii danych w wektorze *X*, należy obliczyć wartości funkcji w wektorze *Y* następującym przypisaniem:

```
Y = sin(X).*cos(X).^2 ./(1+(X+1).^(1/3))
```

lub

```
Y = sin(X).*cos(X).^2./(1+power(X+1,1/3))
```

Należy podkreślić, że funkcja *power* wykonuje tablicowe potęgowanie macierzy, czyli ma działanie równoważne z operatorem (.^) .

Rozwiązywanie układu równań liniowych

Rachunek macierzowy można wykorzystać przy rozwiązywaniu układu równań liniowych. Przykładowo, dla układu trzech równań o trzech niewiadomych o postaci (uprzednio uszeregowanych według zmiennych i przeniesionych wyrazach wolnych na prawe strony równań):

$$\begin{aligned} 2x + 3y - 4z &= 5 \\ x + y - z &= 3{,}5 \\ -2{,}5y - z &= 2 \end{aligned}$$

należy utworzyć macierz kwadratową, zawierającą współczynniki przy niewiadomych (pamiętając o wstawieniu wartości zero tam gdzie zmienna nie występuje):

```
A = [2  3  -4 ; 1  1  -1  ; 0  -2.5  -1]
```

następnie zdefiniować wektor kolumnowy z wyrazów wolnych:

```
B = [ 5 ; 3.5  ;  2]
```

Wektor rozwiązań wyznaczy instrukcja (macierz odwrotna do macierzy *A* pomnożona przez *B*):

```
X= A^-1*B
```

lub

```
X= inv(A)*B
```

lub

```
X= A\B
```

Oto skrypt rozwiązujący powyższy układ równań:

clc, clear A = [2 3 -4 ; 1 1 -1 ; 0 -2.5 -1] B = [5 ; 3.5 ; 2] X = A^-1*B sprawdzenie=A*X	A = 2.0000 3.0000 -4.0000 1.0000 1.0000 -1.0000 0 -2.5000 -1.0000 B = 5.0000 3.5000 2.0000 X = 5.0000 -1.0000 0.5000 sprawdzenie = 5.0000 3.5000 2.0000

Powinno się także sprawdzić poprawność rozwiązania osobno dla każdego równania. Przykładowo, podstawiając rozwiązania do pierwszego równania:

```
spr_X1=A(1,1)*X(1)+ A(1,2)*X(2)+A(1,3)*X(3) - B(1)
```

powinniśmy uzyskać wartość 0.

Zamiast operacji:

```
X= A^-1*B
```

można wykorzystać funkcję *linsolve*:

```
X= linsolve(A,B)
```

Rozwiązywanie równania *n*-tego stopnia (miejsca zerowe wielomianu)

Dla rozwiązania równania o postaci ogólnej:

$$a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \ldots + a_1 x + a_0 = 0$$

należy utworzyć wektor zawierający kolejne współczynniki przy niewiadomych w kolejności malejących potęg, uważając na wstawienie wartości zerowych dla brakujących potęg zmiennej *x*:

A= [$a_n \quad a_{n-1} \quad a_{n-2}, \; \ldots \;, a_0$]

a następnie wykorzystać wbudowaną funkcję *roots*:

X = **roots**(*A*)

która zwraca wektor rozwiązań *X*.

Przykładowo dla równania:

$$5x^4 + 3x^2 + 2x - 1 = 0$$

rozwiązanie przedstawiono poniżej:

Kod	Wynik
clc, clear A = [5 0 3 2 -1] X = roots(A) % sprawdzenie S = A(1)*X(1)^4+A(3)*X(1)^2+A(4)*X(1)-1	A = 5 0 3 2 -1 X = 0.1741 + 0.9512i 0.1741 - 0.9512i -0.6682 0.3201 S = 1.0658e-14 - 7.9936e-15i

Jak widzimy niektóre miejsca zerowe funkcji mogą przyjąć wartości zespolone. Sprawdzenie dało wynik bliski zeru (bardzo małe wartości zespolone).

6.3.10. Macierze wielowymiarowe

W łatwy sposób można wygenerować macierz o losowych elementach, która jest trójwymiarowa (wiersze, kolumny, warstwy).

Przedstawia to poniższy przykład *M-pliku*:

Kod	Wynik
clc, clear A = rand(3,3,3)	A(:,:,1) = 0.7922 0.0357 0.6787 0.9595 0.8491 0.7577 0.6557 0.9340 0.7431 A(:,:,2) = 0.3922 0.7060 0.0462 0.6555 0.0318 0.0971 0.1712 0.2769 0.8235 A(:,:,3) = 0.7922 0.0357 0.6787 0.9595 0.8491 0.7577 0.6557 0.9340 0.7431

Macierz jest wyprowadzana kolejnymi warstwami trzeciego wymiaru.

Na macierzach trójwymiarowych dopuszczalne są operacje tablicowe, oczywiście przy spełnieniu warunków co do zgodności wymiarów i rozmiarów.

Poniższy przykład pokazuje sposób tworzenia macierzy trójwymiarowej 2×2×2 ze stałymi elementami, wykorzystując zakresy indeksów oraz przedstawia rezultat potęgowania tablicowego takiej macierzy:

Code	Output
clc, clear A(1:2,1:2,1:2)=3 B = A.^3	A(:,:,1) = 3 3 3 3 A(:,:,2) = 3 3 3 3 B(:,:,1) = 27 27 27 27 B(:,:,2) = 27 27 27 27

Można też utworzyć macierz trójwymiarową następująco: stworzyć macierz dwuwymiarową pierwszej warstwy, a następnie dodawać do niej kolejne warstwy.

Poniższy skrypt ilustruje tę zasadę, a także pokazuje sposób dostępu do elementów takiej macierzy z wykorzystaniem trzech indeksów:

Code	Output
clc, clear M = [1 2 ; 3 4]; M(:,:,2) = [5 6; 7 8]; M(:,:,3) = [9 10; 11 12] fprintf('Element M(2,2,3) = %d\n', M(2,2,3));	M(:,:,1) = 1 2 3 4 M(:,:,2) = 5 6 7 8 M(:,:,3) = 9 10 11 12 Element M(2,2,3) = 12

Można również tworzyć macierz trójwymiarową, generując kolejne elementy:

Code	Output
clc, clear M(1,1,1) = 1; M(1,2,1) = 2; M(2,1,1) = 3; M(2,2,1) = 4; M(1,1,2) = 5; M(1,2,2) = 6; M(2,1,2) = 7; M(2,2,2) = 8; M	M(:,:,1) = 1 2 3 4 M(:,:,2) = 5 6 7 8

7. Inne typy danych

7.1. Tablice znaków

Napisy, czyli łańcuchy znaków, to wektory znaków, jednowierszowe tablice, zawierające w komórkach pojedyncze znaki. Na takich wektorach można wykonywać pewne operacje.

Poniższa tabela przedstawia funkcje operujące na wektorach zawierających znaki.

Tabela 7.1. Funkcje operacji tekstowych

char(*A*, *B*)	funkcja łączy wierszowo wektory znaków, dopełniając spacjami do równej długości (w każdym wierszu tyle samo znaków, każdy znak w komórce macierzy),
char(*A*)	zmiana typu macierzy *A* (także wyrażeń symbolicznych) na typ tekstowy,
char(*liczba*)	znak o podanym kodzie ASCII
double(*znak*)	kod ASCII znaku
findstr(*znak*,*A*)	funkcja znajduje pozycję (indeks) *znaku* w wektorze znaków *A,*
strcmp(*tekst1*, *tekst2*)	funkcja porównuje dwa teksty (wektory znaków) - jeżeli są identyczne, zwraca logiczne 1,
strcat(*tekst1*, *tekst2,...*)	funkcja wykonuje połączenie łańcuchów znaków,
lower(*tekst*)	funkcja zamienia w tekście duże litery na małe,
upper(*tekst*)	funkcja zamienia w tekście małe litery na duże.

Przykład wyjaśniający stosowanie wybranych funkcji operujących na ciągach znaków:

```
clc, clear
M = char('Politechnika', 'Rzeszowska')

%zamień na duże litery
duze = upper(M)

%pozycje litery 's' w drugim wierszu
poz_s = findstr('s',M(2,:))

%porównanie tekstów
porownaj = strcmp('Janek','Tadek')
```

```
M =
  2×12 char array
    'Politechnika'
    'Rzeszowska  '
duze =
  2×12 char array
    'POLITECHNIKA'
    'RZESZOWSKA  '
poz_s =
    4    8
porownaj=
  logical
    0
```

i znaczenie działania innych funkcji:

clc, clear %łączenie tekstów razem = strcat('Jan',' Tadeusz', ' Kowalski') %szukanie litery 'a' x = findstr('a',razem) kod = double('c') znak = char(kod)	razem= 'Jan Tadeusz Kowalski' x = 2 6 16 kod = 99 znak = 'c'

Zmienna *razem* jest wektorem o 20 komórkach (20 znaków, wliczając 2 spacje).

Podział tekstów na fragmenty odbywa się podobnie jak w macierzach liczbowych, używamy poznanych notacji, określających zakresy indeksów poszczególnych znaków.

Przykład wyodrębniania fragmentów tekstów:

clc, clear M = ['Politechnika Rzeszowska'] pierwsza_litera = M(1) t1 = M(1:12) t2 = M(14:end)	M = 'Politechnika Rzeszowska' pierwsza_litera = 'P' t1 = 'Politechnika' t2 = 'Rzeszowska'

7.2. Tablice komórkowe (*cell arrays*)

Tablica komórkowa (ang. *cell array*) to struktura heterogeniczna, czyli umożliwiająca **przechowywanie wielu danych o zróżnicowanych typach**.

Poniżej kilka zasad:

- tablica komórkowa musi być prostokątna, czyli każdy wiersz powinien posiadać tyle samo komórek,
- komórka takiej tablicy może zawierać dowolne typy danych (np. liczby, macierze liczbowe, wektory znaków, inne tablice komórkowe),
- każda komórka w tablicy komórkowej identyfikowana jest indeksem lub indeksami,
- indeksy w nawiasach klamrowych { } to dostęp do komórki w tablicy komórkowej.

Przykładowo:

clc, clear oceny = {'fizyka', 3.5; 'informatyka', 4.0; 'mechanika', 5.0} przedmiot1 = oceny{1,1} przedmiot2 = oceny{2,1}	oceny = 3×2 cell array {'fizyka' } {[3.5000]} {'informatyka'} {[4]} {'mechanika' } {[5]} przedmiot1 = 'fizyka' przedmiot2 = 'informatyka'

W tablicy występują parami dane skalarne (macierze jednokomórkowe) i ciągi znaków (wektory znakowe), w trzech wierszach.

Poniżej przykład kodu z formatowaniem składowych tablicy komórkowej:

```
clc, clear
oceny = {'fizyka', 3.5; 'informatyka', 4.0; 'mechanika',5.0};
%wypisanie elementów
fprintf('Przedmiot: %12s  Ocena:%f\n', oceny{1,1},oceny{1,2} )
fprintf('Przedmiot: %12s  Ocena:%f\n', oceny{2,1},oceny{2,2} )
fprintf('Przedmiot: %12s  Ocena:%f\n', oceny{3,1},oceny{3,2} )
```

oraz rezultat wykonania skryptu:

```
Przedmiot:       fizyka  Ocena:3.500000
Przedmiot:  informatyka  Ocena:4.000000
Przedmiot:    mechanika  Ocena:5.000000
```

Poniżej jeszcze jeden przykład macierzy komórkowej, pokazujący sposób reprezentacji daty w postaci tablicy homogenicznej o trzech liczbach.

```
clc, clear
studenci = {1, 'Jan','Abacki',[ 1 10 1998]
            2, 'Tadeusz','Babacki',[4 11  1999]
            3,'Anna','Cabacka',[7 1 2000]}
fprintf('Data urodzenia studenta nr:%d to %d.%d.%d\n',...
        studenci{3,1}, studenci{3,4}(1), studenci{3,4}(2),studenci{3,4}(3))
```

W celu kontynuacji kodu długiej instrukcji w następnym wierszu, zastosowano trzy kropki (...) w miejscu przeniesienia.

Rezultat skryptu:

```
studenci =
                        3×4 cell array
                         {[1]}   {'Jan'    }   {'Abacki' }    {1×3 double}
                         {[2]}   {'Tadeusz'}   {'Babacki'}    {1×3 double}
                         {[3]}   {'Anna'   }   {'Cabacka'}    {1×3 double}
Data urodzenia studenta nr:3 to 7.1.2000
```

W powyższym przykładzie dostęp do komórki tablicy komórkowej odbywa się przez indeksy w nawiasach klamrowych {}, a następnie do elementu macierzy wewnętrznej przez indeksy w nawiasach okrągłych (), jak niżej:

*zmienna***{***indeksy_komórki***}(***indeksy_elementu_tablicy***)**

Można też bardziej zróżnicować typy danych w tablicy komórkowej:

Kod	Wynik
clc, clear tablica = {1.5,'Napis'; 3+4i, [1 2; 3,4]} %suma elementów komórki {2,2} suma = sum(tablica{2,2},'all')	tablica = 2×2 cell array {[1.5000]} {'Napis' } {[3.0000 + 4.0000i]} {2×2 double} suma= 10

Tablica komórkowa o rozmiarach 2×2 zawiera liczbę rzeczywistą, tekst, liczbę zespoloną i małą tablicę homogeniczną z liczbami.

Tablice komórkowe można przechowywać w innych tablicach komórkowych, dbając o odświeżanie zawartości. Poniższy skrypt ilustruje ten fakt:

Kod	Wynik
clc,clear a = {1, 'A'}; %dwie małe tablice komórkowe b = {2, 'B'}; M = {a; b}; %wstawiamy do tablicy M L = M{1}{2} a = {3,'C'} %zmiana tablicy składowej L = M{1}{2} %nie aktualizuje w dużej tablicy M ={a; b}; % ponownie wstawiamy L = M{1}{2} % teraz aktualna tablica M	el1 = 'A' a = 1×2 cell array {[3]} {'C'} L = 'A' L= 'C'

Po zmianie zawartości tablicy składowej, tablica ta powinna być ponownie wstawiona do dużej tablicy.

Tablice komórkowe są przydatne w algorytmach wyszukiwania podanego ciągu znaków w zbiorach tekstów. Opisana w poprzednim rozdziale funkcja *strcmp* w wersji:

strcmp*(wektor_znaków, tablica_komórkowa_wektorów_znaków)*

zwraca wektor logicznych zer i jedynek. Jedynki są w pozycjach, dla których porównanie wypadło pozytywnie. Przykładowo:

Kod	Wynik
clc, clear imiona ={'Anna', 'Ewa', 'Ola', 'Ula'}; x = 'Ola'; wektor =strcmp(imiona,x)	wektor = 1×4 logical array 0 0 1 0

MATLAB udostępnia również funkcję *any*, która sprawdza, czy w wektorze są niezerowe wartości. Jeżeli takie są, zwracana jest *prawda logiczna*:

wartość_logiczna = **any**(*wektor*)

clc, clear imiona={'Anna', 'Ewa', 'Ola', 'Ula'}; x = 'Ola'; wektor=strcmp(imiona,x) sprawdz=any(wektor) x = 'Asia'; wektor=strcmp(imiona,x) sprawdz = any(wektor)	wektor = 1×4 logical array 0 0 1 0 sprawdz = logical 1 wektor = 1×4 logical array 0 0 0 0 sprawdz = logical 0

Indeks znalezionej pozycji uzyskuje się poznaną funkcją *find*. Zasadę tę wyjaśnia poniższy przykład:

clc, clear
imiona = {'Anna', 'Ewa', 'Ola', 'Iza'};
x = input('Podaj imię:','s');
wektor = strcmp(imiona,x);
sprawdz = any(wektor);
nr = find(wektor);
if ~sprawdz
disp('Nie ma')
else
fprintf('Znaleziono w pozycji: %d\n', nr)
end
	Podaj imię:Ewa
	Znaleziono w pozycji: 2

Wykorzystanie okna dialogowego, uruchamianego funkcją *inputdlg* (w prostej postaci jest to funkcja dwuargumentowa), wymaga zastosowania tablicy komórkowej:

zmienna = **inputdlg**(*p*, *t*)

gdzie argumenty to:

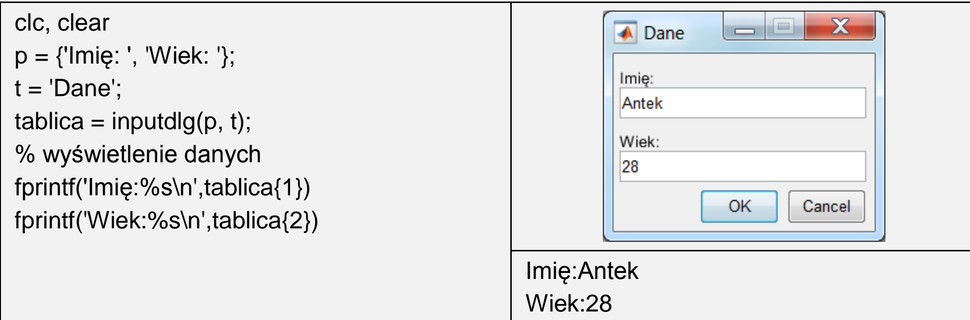
p - tablica komórkowa z tekstami opisującymi okienka edycyjne dla danych

t - tekst tytułu okna dialogowego.

Dla argumentu ***p*** wybrano tablicę komórkową, ponieważ może ona przechowywać wartości tekstowe - ciągi znaków o różnej długości (tablice homogeniczne nie mogą, gdyż różnica długości tekstów przeczy zasadzie homogeniczności, mogą przechowywać tylko pojedyncze znaki).

Zmienna będąca rezultatem działania funkcji, to również tablica komórkowa zawierająca teksty wpisane przez użytkownika w polach edycyjnych.

Wykonanie poniższego *M-pliku* spowoduje pojawienie się okna dialogowego, w którym można wpisać odpowiednie teksty. Rezultat jest następujący:




```
clc, clear
p = {'Imię: ', 'Wiek: '};
t = 'Dane';
tablica = inputdlg(p, t);
% wyświetlenie danych
fprintf('Imię:%s\n',tablica{1})
fprintf('Wiek:%s\n',tablica{2})
```

```
Imię:Antek
Wiek:28
```

Należy zaznaczyć, że podawana w okienku wartość wieku jest typu tekstowego (dokładniej jest to wektor znaków). Chcąc ją interpretować w skrypcie jako liczbę (wykonywać dowolne obliczenia arytmetyczne, np. obliczać średnią wieku kilku osób), należy wykonać konwersję typu tekstowego na liczbowy (*double*), wykorzystując funkcję *str2double*.

zmienna_liczbowa = **str2double**(*wyrażenie_tekstowe*)

Funkcja *str2double* będzie przydatna w aplikacjach okienkowych *GUI* (rozdz.18).

Kontynuacja powyższego skryptu wyjaśnia typy danych (polecenie *whos*) i pokazuje także sposób dostępu do pojedynczych znaków tekstu (w takiej samej notacji jak do elementów wektora):

```
%..... kontynuacja
imie = tablica{1}

%pierwszy znak
pierwsza_litera = imie(1)

wiek_kom= tablica(2);
%konwersja do typu liczbowego
wiek = str2double(wiek_kom)
whos        %jakie mamy zmienne
```

```
imie =
   'Antek'
pierwsza_litera =
   'A'
wiek =
    28
  Name             Size  Bytes  Class Attributes
  imie              1x5    10    char
  pierwsza_litera   1x1     2    char
  druga_litera      1x1     2    char
  tablica           2x1   238    cell
  t                 1x4     8    char
  p                 1x2   248    cell
  wiek_kom          1x1   116    cell
  wiek              1x1     8    double
```

Zmienna *wiek_kom* jest komórką (*cell*), zmienna *wiek* po konwersji jest typu liczbowego (*double*).

Elementy tablic komórkowych, zawierających łańcuchy znaków o różnej długości, można łatwo sortować w kolejności alfabetycznej przy pomocy funkcji *sort*:

Kod	Wynik
clc, clear nazwiska = {'Cabacki', 'Abacki','Babacki'} posortowane=sort(nazwiska)	nazwiska = 1×3 cell array {'Cabacki'} {'Abacki'} {'Babacki'} posortowane = 1×3 cell array {'Abacki'} {'Babacki'} {'Cabacki'}

7.3. Struktury i tablice struktur

Elementami struktury są dane złożone z wielu elementów (zwanych **polami**), mogących przechowywać wartości różnego typu i, co istotne, identyfikowanych nie przez indeksy liczbowe, lecz przez unikalne **identyfikatory** (nazwy). W innych językach, w tym w językach obsługi baz danych, struktury są nazywane rekordami.

Definicja pojedynczej struktury może być dokonana przy użyciu funkcji *struct* według zapisu:

zmienna=**struct**(*nazwa_pola1*, *wartość1*, *nazwa_pola2*, *wartość2*, ... *itd.*)

czyli na przykład:

student = **struct**(nazwisko, 'Nowak', imie, 'Andrzej', wiek, 22)

Można też bezpośrednio tworzyć strukturę, używając notacji kropkowej (tzw. kwalifikowanej), przykładowo:

```
student.nazwisko = 'Kowalski'
student.imie = 'Jan'
student.wiek = 20
```

i tak dalej.

Struktura o nazwie *student* zawiera trzy pola: *nazwisko* i *imie*- typu znakowego (*char*), i *wiek* - typu liczbowego (*double*).

Dostęp do pól jest podobny, korzysta się z zapisu kropkowego:

Kod	Wynik
clc, clear student = struct('nazwisko','Nowak','nr_indeksu',12345); %wypisanie elementów fprintf('%s %d\n', student.nazwisko, student.nr_indeksu)	Nowak 12345

Na dowolnym etapie skryptu można poszerzyć strukturę, dodając w niej nowe pole i jego wartość:

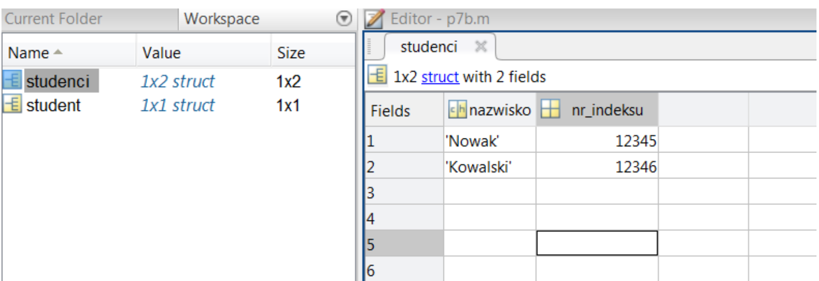
```
student.stypendium=500
```

Jeżeli struktury posiadają tę samą liczbę pól i takie same identyfikatory pól, można je umieścić w macierzy homogenicznej (najczęściej w wektorze), jeżeli się różnią typem danych, to umieszczamy je w tablicy komórkowej.

Ilustruje to przykładowy skrypt:

clc, clear
student .nazwisko = 'Nowak';
student.nr_indeksu=12345;
%przepisanie do elementu tablicy homogenicznej
studenci(1) = student;
 	
%teraz tworzymy strukturę z innymi danymi
student.nazwisko = 'Kowalski';
student.nr_indeksu = 12346;
%przepisanie do drugiej komórki tablicy
studenci(2) = student;
 	
%wypisanie elementów
fprintf('%s %d\n', studenci(1).nazwisko, studenci(1).nr_indeksu)
fprintf('%s %d\n', studenci(2).nazwisko, studenci(2).nr_indeksu)
	Nowak 12345
	Kowalski 12346

Po dwukrotnym kliknięciu w nazwę zmiennej w *Workspace* można obejrzeć układ składowych i aktualną zawartość komórek tablicy struktur (rys. 7.1).



Rys.7.1. Widok tablicy struktur w arkuszu danych

Oprócz zastosowań wektorów struktur w aplikacjach bazodanowych, są one rezultatami rozwiązań układów równań różniczkowych (rozdz.12.11).

W dalszej części będzie opisana instrukcja iteracyjna, która ułatwi generowanie macierzy liczbowych (także struktur umieszczanych w wektorze), ich analizę i inne wielokrotne operacje.

W rozdz.19 zamieszczono przykład aplikacji bazy danych, w którym zastosowano struktury i tablicę struktur oraz przedstawiono podstawowe operacje na tym typie danych.

7.4. Tabele

Ciekawym typem danych są tabele (typ *table*). Można je wykorzystywać w aplikacjach bazodanowych, przy analizie danych pomiarowych itp.

Tabelę można utworzyć wieloma sposobami. Najprostszym jest złożenie tabeli z wektorów kolumnowych o tej samej liczbie wierszy.

Przykład:

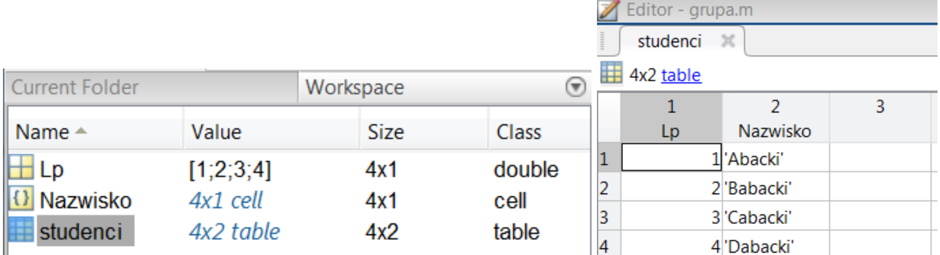
```
clc, clear
Lp=[1;2;3;4];
Nazwisko = {'Abacki';'Babacki';'Cabacki';'Dabacki'};

%złożenie w tabelę
studenci = table(Lp,Nazwisko)
```

```
studenci =
  4×2 table
    Lp    Nazwisko
    __    _________
    1     'Abacki'
    2     'Babacki'
    3     'Cabacki'
    4     'Dabacki'
```

Została wykorzystana funkcja *table*, która tworzy element typu *table*, złożony z dwóch kolumn danych różnych typów: wektora danych numerycznych (4x1 *double*) i wektora komórek (4x1 *cell*).

Jeśli istnieje rzadka konieczność zróżnicowania typów danych w jednej z kolumn, można zastosować tablice komórek, np.: Lp={'x';2;3;4}

Zmienne w *Workspace* i arkusz danych pokazuje rys.7.2.



Rys.7.2. Zmienne i arkusz dla tabeli *studenci*

Kolumny tabeli otrzymały nazwy zmiennych składowych.

Operacje na typie *table* nie są trudne, lecz wymagają nieco wprawy. Poniżej kilka użytecznych operacji na tabelach.

Dodawanie nowych kolumn odbywać się może w następujący sposób:

```
%... kontynuacja
Kierunek={'MPDI';'MPZI';'MPZI';'MMDU'};          %nowe dane
studenci(:,3) = Kierunek;                        %wstawienie
studenci.Properties.VariableNames(3)={'Kierunek'};  %ustalenie nazwy kolumny
Data_ur = {datetime(1951,10,6);
        datetime(1951,10,7);
        datetime(1951,10,8);
        datetime(1951,10,9)};                    %nowe dane
studenci(:,4) = Data_ur;                         %wstawienie
studenci.Properties.VariableNames(4) = {'Data_ur'}  %ustalenie nazwy kolumny
```

W skrypcie wykorzystano funkcję *datetime*, służącą do definiowania daty.

Rezultat skryptu:

```
studenci =
 4×3 table
   Lp    Nazwisko     Kierunek
   __    _________    ________
   1     'Abacki'     'MPDI'
   2     'Babacki'    'MPZI'
   3     'Cabacki'    'MPZI'
   4     'Dabacki'    'MMDU'
studenci =
 4×4 table
   Lp    Nazwisko     Kierunek   Data_ur
   __    _________    ______     ______________
   1     'Abacki'     'MPDI'     06-Oct-1951
   2     'Babacki'    'MPZI'     07-Oct-1951
   3     'Cabacki'    'MPZI'     08-Oct-1951
   4     'Dabacki'    'MMDU'     09-Oct-1951
```

Dodawanie nowych wierszy może się odbyć według schematu:

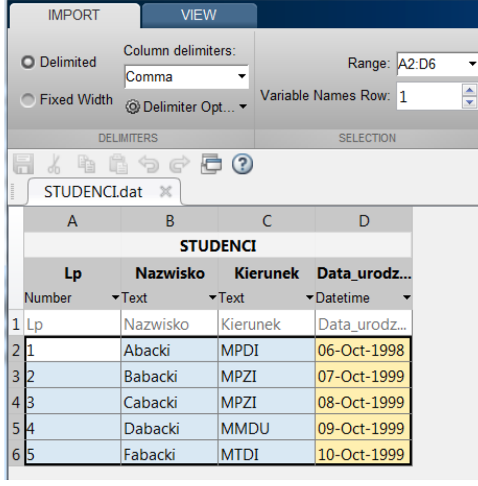
studenci = [studenci;{5,'Fabacki','MTDI',datetime(1999,10,10)}]

lecz trzeba uważać na typy danych każdej kolumny.

Zapis do pliku i odczyt tabeli z pliku wspomagają funkcje *writetable* i *readtable*:

```
%... kontynuacja
writetable(studenci,'STUDENCI.dat')
clear
NOWA=readtable('STUDENCI.dat')
```

Dużo możliwości daje podgląd pliku w oknie *IMPORT*, uruchamianym kliknięciem w nazwę pliku z zapisanymi danymi w oknie katalogu roboczego (rys.7.3).



Rys.7.3. Podgląd pliku w oknie *IMPORT*

Zaznaczenie interesującego wycinka danych w tym okienku i wybranie opcji *Import Selection* Import Selection, co skutkuje powstaniem w *Workspace* zmiennej o nazwie takiej jak plik, zawierającej tabelę z wybranymi danymi.

Odwołanie do interesujących danych w tabeli może być zróżnicowany.

Znanym sposobem:

%... kontynuacja skryptu…	fragment1 =
%wybieramy 1-szy wiersz	1×4 table
fragment1 = NOWA(1,:)	Lp Nazwisko Kierunek Data_ur
	__ ________ ________ ____________
	1 'Abacki' 'MPDI' 06-Oct-1998

jest tworzona tabela, o zawartości określonej zakresami indeksów.

Stosuje się też style zapisu z odwołaniem do nazw kolumn:

```
fragment2 = NOWA(:,'Nazwisko')
```

lub alternatywnie zapis kropkowy:

```
fragment2 = NOWA.Nazwisko(:)
```

co stanowi znaczne ułatwienie

Dla kilku kolumn:

```
fragment3 = NOWA(:,{'Nazwisko','Data_ur'})
```

Można też umieszczać indeksy w nawiasach klamrowych, ale tylko pojedyncze kolumny (albo zakresy kolumn z danymi tego samego typu), ponieważ wówczas tworzona jest tablica typu zgodnego z typem danych kolumny.

Z pierwszej kolumny wyodrębniona zostanie macierz liczbowa, z czwartej kolumny tablica elementów typu *datetime.*

%... kontynuacja kodu….	fragment4 =
% wybieramy 1-szą kolumnę	1
fragment4 = NOWA{1:2, 1}	2
	fragment5 =
%wybieramy 4-tą kolumnę	2×1 datetime array
fragment5 = NOWA{1:2, 4}	06-Oct-1998
	07-Oct-1999
%wybieramy 2 i 3 kolumnę	fragment6 =
fragment6 = NOWA{1:2, 2:3}	2×2 cell array
	{'Abacki' } {'MPDI'}
	{'Babacki'} {'MPZI'}

Kolumny drugą i trzecią można było scalić, ponieważ dane są typu *cell*:

fragment4	[1;2]	2x1	double
fragment5	*2x1 datetime*	2x1	datetime
fragment6	*2x2 cell*	2x2	cell

Rys.7.4. Informacje o kolumnach tabeli w oknie *Workspace*

Informację o typach danych w pojedynczych komórkach można uzyskać w poniższy sposób:

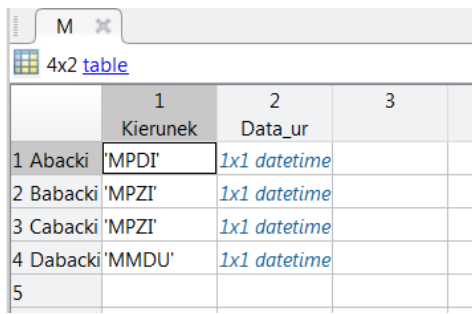
```
%... kontynuacja…
for k=1:4
   typy{k}=class(NOWA{:,k});
 end
typy
```

```
typy =
  1×4 cell array

    {'double'}    {'cell'}    {'cell'}    {'datetime'}
```

Na koniec jeszcze jedna ciekawa zasada, możliwość tworzenia tabeli z nazwami wierszy pobranymi z tablicy:

```
%... kontynuacja…
M = table(Kierunek,Data_ur, 'RowNames',Nazwisko)
```

Efekt w arkuszu pokazuje rys.7.5.



M

4x2 table

	1 Kierunek	2 Data_ur	3
1 Abacki	'MPDI'	*1x1 datetime*	
2 Babacki	'MPZI'	*1x1 datetime*	
3 Cabacki	'MPZI'	*1x1 datetime*	
4 Dabacki	'MMDU'	*1x1 datetime*	
5			

Rys.7.5. Nazwy wierszy pobrane z tablicy

Odwołanie wykonuje się wygodnie:

```
M('Abacki',:)                    %wszystkie dane Abackiego
M({'Abacki', 'Babacki'},:)       %wszystkie dane z listy
```

lub przez nazwę kolumny:

```
M('Cabacki','Data_ur')           %data urodzenia Cabackiego
```

a także w wersji kropkowej:

```
M.Data_ur('Dabacki')             %data urodzenia Dabackiego
```

Tabele są wygodnym typem, dają wiele możliwości analizy danych. Jak wynika z powyższych rozważań i przykładów, wariant dostępu do elementów tabeli i innych operacji użytkownik musi sobie sam wypracować.

Przydatną funkcjonalnością MATLAB-a jest możliwość odczytu plików *MS Excel* o rozszerzeniach *.xls* i *.xlsx*. Działanie to umożliwia poznana wyżej funkcja *readtable*:

zmienna = readtable('plik.xlsx')

8. Obsługa plików

8.1. Pliki tekstowe *ASCII*

W *MATLAB-ie* możliwy jest zapis macierzy do nowotworzonego lub istniejącego (przez nadpisanie treści) pliku tekstowego w kodzie *ASCII* (tekst nieformatowany) i odczyt tej macierzy z pliku. Wykorzystywane są w tym celu funkcje *save* (*zachowaj*) i *load* (*załaduj*).

Zapis pojedynczej zmiennej (macierzy) odbywa się według schematu:

save('*nazwa_pliku*', '*nazwa_zmiennej*', '-ASCII')

lub prościej:

save *nazwa_pliku* *nazwa_zmiennej* -ASCII

Można też zapisać do pliku więcej niż jedną macierz, pod warunkiem, że posiadają tę samą liczbę kolumn. W instrukcji *save* tworzy się wówczas listę zmiennych.

Przykładowy skrypt:

Skrypt	Wynik
clc,clear A =rand(2,6) B =round(rand(6)*50) save wyniki.txt A B -ASCII	A = 0.8147 0.1270 0.6324 0.2785 0.9575 0.1576 0.9058 0.9134 0.0975 0.5469 0.9649 0.9706 B = 35 1 6 26 19 24 3 32 7 21 23 25 31 9 2 22 27 20 8 28 33 17 10 15 30 5 34 12 14 16 4 36 29 13 18 11

Wynikiem działań skryptu będzie powstanie w katalogu roboczym *Current Folder* nowego pliku o nazwie *wyniki.txt*. Rozszerzenie *txt* pliku nie jest obowiązkowe (tylko dla celów *Windows*), istotna jest opcja *ASCII*.

Plik danych może być też efektem urządzeń pomiarowych, może też być utworzony przez użytkownika. Należy jedynie zadbać, by struktura pliku była macierzowa, czyli liczba danych w wierszach powinna być jednakowa.

Chcąc odczytać zawartość pliku tekstowego, stosuje się funkcję *load* w wersjach:

load(*nazwa_pliku*, '-ASCII')

load(*nazwa_pliku*)

load *nazwa_pliku*

Oczytane dane są widoczne w zmiennej o identyfikatorze zgodnym z nazwą pliku (bez rozszerzenia).

Oto odczyt poprzednio utworzonego pliku:

```
clc,clear
load wyniki.txt
wyniki
```

```
wyniki =
   0.8147   0.1270   0.6324   0.2785   0.9575   0.1576
   0.9058   0.9134   0.0975   0.5469   0.9649   0.9706
  35.0000   1.0000   6.0000  26.0000  19.0000  24.0000
   3.0000  32.0000   7.0000  21.0000  23.0000  25.0000
  31.0000   9.0000   2.0000  22.0000  27.0000  20.0000
   8.0000  28.0000  33.0000  17.0000  10.0000  15.0000
  30.0000   5.0000  34.0000  12.0000  14.0000  16.0000
   4.0000  36.0000  29.0000  13.0000  18.0000  11.0000
```

MATLAB umożliwia podgląd i edycję pliku tekstowego (w oknie *Editor* lub dowolnym, zewnętrznym edytorem ASCII) metodami takimi samymi jak obsługa *M-plików*.

8.2. *MAT*-pliki

Pliki z rozszerzeniem ***.mat*** to pliki typu binarnego, umożliwiające zapis wielu zmiennych różnych typów (macierzy różnych rozmiarów, tablic komórkowych itp.). Użytkownik ma zatem możliwość zapisu dowolnych zestawów zmiennych, istniejących aktualnie w obszarze roboczym *Workspace*.

Zapis i odczyt *MAT*-plików odbywa się podobnie jak dla plików tekstowych *ASCII*, z użyciem funkcji *save* i *load*.

Poniżej alternatywne schematy zapisu:

save('*plik*.**mat**', '*zmienna1*', '*zmienna2*' , ..., '*zmiennaN*')
save *plik*.**mat** *zmienna1* *zmienna2* ..... *zmiennaN*
save *plik* *zmienna1* *zmienna2* ..... *zmiennaN*

i odczytu:

load('*plik*.**mat**')
load *plik*.**mat**
load *plik*

Istnieje też możliwość dopisania wartości zmiennych do istniejącego pliku, korzystając z opcji -*append (dołącz)*:

save ('*plik*.**mat**', '*zmienna1*', '*zmienna2*' , ..*itd*... , '*zmiennaN*', '**-append**')
save *plik zmienna1* *zmienna2* ...*itd*... **-append**

W pliku zostają zapisanie wartości zmiennych oraz ich nazwy. Rozszerzenie *mat* jest domyślne i nie musi być używane w nazwie pliku.

Poniżej przykład *M-pliku*. w którym zapisano wartości kilku różnych zmiennych oraz, po usunięciu zmiennych, odczytano je z *mat*-pliku:

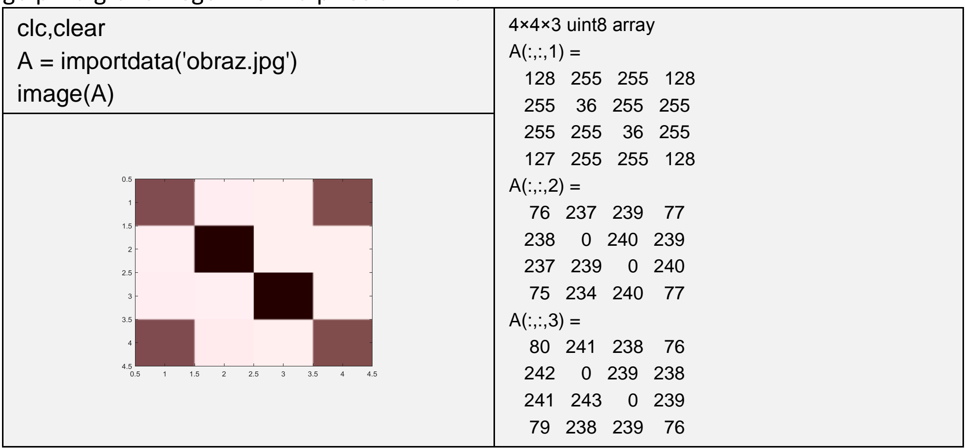
```
clc,clear
a = 6; b = [1 2 3 4]  %liczba i wektor
c = {'Janek', 8}      %macierz komórkowa
d.x = 3               %element struktury
d.y = 'Kowalski'      %element struktury
save dane a b c d     %zapis do pliku
clear                 %usuwamy zmienne
load dane             %odczyt z pliku
disp('Efekt polecenia whos');
whos                  %jakie mamy zmienne
```

```
b =
   1    2    3    4
c =
  1×2 cell array
    {'Janek'}    {[8]}
d =
  struct with fields:
    x: 3
d =
  struct with fields:
    x: 3
    y: 'Kowalski'
Efekt polecenia whos
Name   Size    Bytes   Class  Attributes
  a      1x1        8  double
  b      1x4       32  double
  c      1x2      242    cell
  d      1x1      370   struct
```

MALAB udostępnia również bardziej zaawansowane metody obsługi plików, z wykorzystaniem między innymi funkcji: *fopen*, *fprintf, fwrite, fread* i *fclose*.

8.3. Obsługa plików graficznych

Zaawansowane działania na plikach graficznych umożliwia funkcja *importdata*, która umieszcza dane o obrazie w macierzy trójwymiarowej. Indeksy wierszy i kolumn tej macierzy to współrzędne pikseli, trzy warstwy reprezentują składowe *RGB* koloru. Wyświetlenie obrazu wykonuje funkcja *image*. Poniżej prosty przykład otwarcia niewielkiego pliku graficznego z 16-ma pikselami wraz z :



```
clc,clear
A = importdata('obraz.jpg')
image(A)
```

```
4×4×3 uint8 array
A(:,:,1) =
  128  255  255  128
  255   36  255  255
  255  255   36  255
  127  255  255  128
A(:,:,2) =
   76  237  239   77
  238    0  240  239
  237  239    0  240
   75  234  240   77
A(:,:,3) =
   80  241  238   76
  242    0  239  238
  241  243    0  239
   79  238  239   76
```

MATLAB zawiera również inne funkcje obsługi plików graficznych różnych typów.

9. Instrukcje strukturalne

Instrukcje strukturalne, to takie instrukcje, w skład których wchodzi inna, dowolna instrukcja (bądź wiele instrukcji).

9.1. Instrukcja warunkowa *if*

Jeżeli algorytm naszego programu wymaga zbadania (sprawdzenia, przetestowania) jakiejś sytuacji obliczeniowej, wówczas stosuje się **instrukcję warunkową** *if..else..end*. Jej celem jest rozgałęzienie wykonania akcji programu na alternatywne ciągi operacji.

Instrukcja służy do sprawdzenia warunków i alternatywnego wykonywania różnych grup instrukcji, gdy dany warunek będzie *prawdą* (*logiczne 1*) lub *fałszem* (*logiczne 0*).

Postać ogólna instrukcji warunkowej:

if *wyrażenie_logiczne1*

instrukcje_1 (wykonywane, gdy *wyrażenie_logiczne1* jest prawdą logiczną)

elseif *wyrażenie_logiczne2*

instrukcje_2 (wykonywane, gdy *wyrażenie_logiczne2* jest prawdą logiczną)

elseif *wyrażenie_logiczne3*

instrukcje_3 (wykonywane, gdy *wyrażenie_logiczne3* jest prawdą logiczną)

*...itd. - bloków **elseif** może nie być, lecz także może być ich wiele...*

else

instrukcje_n (wykonywane, gdy wszystkie wyrażenia logiczne są fałszywe)

end

Znaczenie słów kluczowych w instrukcji warunkowej:

if - **jeżeli** prawdą jest, że ...

elseif - w innym przypadku gdy ...

else - w pozostałych przypadkach ...

end - koniec instrukcji.

Działanie instrukcji polega na **sprawdzaniu wartości kolejnych wyrażeń logicznych** (warunków).

- jeżeli kolejne analizowane wyrażenie logiczne będzie miało wartość *prawda* (*logiczne 1*) wykonane zostaną instrukcje tego bloku i instrukcja warunkowa się kończy, czyli program przechodzi do wykonania następnej instrukcji skryptu (po słowie kluczowym *end*).
- jeżeli analizowane wyrażenie logiczne będzie miało wartość *fałsz* (*logiczne 0*) instrukcja przechodzi do badania następnego wyrażenia logicznego,
- jeżeli żadne z wyrażeń logicznych nie będzie posiadało wartości *prawda*, wykonany zostanie blok instrukcji po słowie kluczowym *else*, **po tym słowie kluczowym nie umieszcza się wyrażenia logicznego**.

Bloki decyzyjne *elseif* i *else* są opcjonalne (mogą nie wystąpić), zatem w instrukcji:

if *wyrażenie_logiczne*
instrukcje_1 (wykonywane, gdy wyrażenie logiczne ma wartość *prawda*)
else
instrukcje_2 (wykonywane, gdy jest wyrażenie logiczne ma wartość *fałsz*)
end

sprawdzane jest tylko jedno wyrażenie logiczne i, w zależności od jego wartości logicznej, wykonywane są alternatywne bloki instrukcji wewnętrznych.
Natomiast instrukcja warunkowa w uproszczonej postaci:

if *wyrażenie_logiczne*
instrukcje (wykonywane, gdy wyrażenie logiczne ma wartość *prawda*)
end

wykonuje instrukcje wewnętrzne gdy wyrażenie ma wartość *prawda*, w przeciwnym wypadku skrypt przechodzi do wykonania następnej instrukcji po instrukcji warunkowej.

Pierwszy przykład ma na celu wypisanie alternatywnych komunikatów, informujących o możliwości znalezienia bądź nie, rozwiązań rzeczywistych równania kwadratowego.

Wykonywane są kolejne operacje:

- instrukcje przypisania określają pewne wartości współczynników *a*, *b* i *c* równania kwadratowego,
- obliczana jest wartość zmiennej *delta*,
- przy pomocy instrukcji warunkowej *if* analizowane są kolejne warunki, w przypadku wartości *prawda* któregokolwiek z nich, wypisane są odpowiednie komunikaty,
- gdy obydwa warunki mają wartość *fałsz* wykonywane są instrukcje bloku *else*.

```
clc,clear
% współczynniki
a = 1; b = 6; c = 3;
%obliczenie
delta = b^2-4*a*c
if delta<0                 % analiza warunków
   disp ('delta jest ujemne')
elseif delta == 0
   disp('delta równe 0')
else
  disp('delta większe od 0')
end;
```

```
delta =
   24
delta większe od 0
```

Poniżej zamieszczono rozbudowaną wersję powyższego skryptu, w której użytkownik podaje wartości współczynników równania. Następnie instrukcja warunkowa, w zależności od wartości wprowadzonych danych i wynikającej stąd wartości zmiennej *delta*,

wyświetla komunikat o braku rozwiązań rzeczywistych lub oblicza i wypisuje wartość rozwiązania podwójnego, bądź wartości dwóch różnych rozwiązań.

Oto ten skrypt:

clc,clear
disp('Rozwiąż równanie kwadratowe');
a = input('Podaj a:');
b = input('Podaj b:');
c = input('Podaj c:');
delta = b^2-4*a*c;
if delta<0
    disp ('Brak rozwiązań')
elseif delta == 0
    x = -b/2/a;
    fprintf('x=%f\n', x)
else
   x1 = (-b+sqrt(delta))/2/a;
   x2 = (-b-sqrt(delta))/2/a;
   fprintf('x1=%16.5f\n', x1)
   fprintf('x2=%16.5f\n', x2)
end;
disp('KONIEC')
	Rozwiąż równanie kwadratowe
	Podaj a:1
	Podaj b:6
	Podaj c:3
	x1= -0.5505
	x2= -5.4495
	KONIEC

Pisząc kolejne wiersze instrukcji strukturalnej *if*, można zauważyć samoczynne wykonywanie przez edytor wcięć bloków instrukcji wewnętrznych, co zwiększa czytelność i ułatwia zrozumienie działania bloków instrukcji warunkowej.

Jeżeli zadaniem jest sprawdzenie przynależności wartości zmiennej do określonego przedziału, w wyrażeniu logicznym wykorzystuje się operator koniunkcji *&*:

clc,clear
x = input('Podaj jakąś liczbę:');
if   x>=0 & x<10
    disp ('Liczba jest w przedziale [0,10)')
else
    disp ('Liczba jest ujemna lub >=10')
end;
	Podaj jakąś liczbę:6
	Liczba jest w przedziale [0,10)

Należy zwrócić uwagę, że nawiasy na relacjach nie są potrzebne, gdyż operatory relacji mają wyższy priorytet niż operator koniunkcji.

Niekiedy zachodzi konieczność zbadania przynależności wartości wyrażenia do wielu przedziałów liczbowych i wyświetlenia alternatywnych komunikatów. należy wówczas zastosować wielokrotne bloki *elseif*.

Oto przykład:

```
clc,clear
x=input('Podaj jakąś liczbę:');

%Badanie przynależności do przedziału
if  x<0
   disp('Mniej niż zero')
elseif  x<1
   disp('Przedział [0, 1)')
elseif  x<2
   disp('Przedział [1, 2)')
elseif  x<3
   disp('Przedział [2, 3)')
elseif  x<4
   disp('Przedział [3, 4)')
elseif  x<5
   disp('Przedział [4, 5)')
else
   disp('>=5')
end
```

```
Podaj jakąś liczbę:4.6
Przedział [4, 5)
```

Kod działa poprawnie, mimo, że w poszczególnych wyrażeniach logicznych nie podano obydwu granic przedziałów, lecz tylko granicę górną. Wynika to z faktu, że kolejne warunki badają stan po wyeliminowaniu poprzednich przedziałów (wyżej analizowane wyrażenia logiczne miały wartość *fałsz*).

Jeszcze jeden przykład, w którym sprawdzono, czy podawana przez użytkownika liczba jest parzysta:

```
clc,clear
%Badanie parzystości liczby

x = input('Podaj liczbę całkowitą:');

%Analiza
if  rem(x, 2) == 0
   disp ('Liczba jest parzysta')
else
   disp ('Liczba jest nieparzysta')
end;
```

```
Podaj liczbę całkowitą:87
Liczba jest nieparzysta
```

W skrypcie wykorzystana została funkcja *rem* (reszta z dzielenia obydwu argumentów funkcji). Sprawdzono, czy reszta z dzielenia przez 2 jest równa zero i wyświetlono adekwatny do rezultatu logicznego komunikat.

9.2. Instrukcja wyboru *switch*

Instrukcja *switch..case..end* jest "przełącznikiem" bloków instrukcji, wartość testowanego wyrażenia (liczbowego lub znakowego) decyduje o tym, który blok jest wykonywany.

Postać instrukcji:

switch *wyrażenie*
  case *wartość_1*
    instrukcje_1 (wykonywane, gdy wyrażenie ma *wartość_1*)
  case *wartość_2*
    instrukcje_2 (wykonywane, gdy wyrażenie ma *wartość_2*)
      …itd. – bloków case może być wiele...
  otherwise
    instrukcje_n (wykonywane dla pozostałych przypadków)
end

Działanie instrukcji polega na obliczeniu wartości wyrażenia i wykonania tych instrukcji, dla których wyrażenie to jest równe wartości wymienionej w nagłówku danego bloku *case*. Jeżeli wyrażenie nie jest równe żadnej wartości z wymienianych, wykonane zostaną instrukcje bloku *otherwise*. Blok *otherwise* jest opcjonalny.

Znaczenie słów kluczowych w instrukcji:

switch - oblicz *wyrażenie* i "przełącz" na odpowiednie instrukcje,
case - w przypadku gdy *wyrażenie* ma daną wartość, wykonaj ...
otherwise - w pozostałych przypadkach wykonaj ...
end - koniec instrukcji.

Poniżej prosty przykład:

Kod	Wynik

clc,clear
n = input('Podaj swój wybór: 1 2 lub 3 :','s');
switch n
case '1'
   disp ('Wybrałeś 1')
case '2'
   disp ('Wybrałeś 2')
otherwise
   disp ('Wybrałeś 3 lub inny znak')
end
	Podaj swój wybór: 1,2 lub 3: 2
	Wybrałeś 2

Powyższy przełącznik analizuje zmienną typu znakowego (*char*) – cyfry są interpretowane jako znaki – aby umożliwić prawidłowe działanie kodu przy dowolnych znakach podawanych przez użytkownika.

Wybór jednej pozycji z listy wartości tekstowych może być zrealizowany przy pomocy tablicy komórkowej.

Sposób realizacji zadania pokazuje następny przykład:

Program	Wynik

clc,clear
%podawanie wartości tekstowej
f = input('Podaj nazwę funkcji:','s');
 	
%decyzja
switch f
   case {'sin','cos','tan','cot'}
      disp('Trygonometryczna')
   case {'log','log10'}
      disp('Logarytmiczna')
   otherwise
      disp('Inna')
end
	Podaj nazwę funkcji:log
	Logarytmiczna

9.3. Instrukcje iteracyjne ("pętle")

Instrukcje iteracyjne wykorzystywane są do wielokrotnego powtarzania bloku instrukcji.

9.3.1. Pętla *for*

Iteracja ***for*** – zwana też "pętlą" *for* – wykorzystywana jest zwykle wtedy, gdy konieczne jest **wielokrotne wykonanie pewnej sekwencji instrukcji** i liczba powtórzeń jest określona.

Ogólna postać instrukcji:

for *zmienna_licznik* = *wektor_wartości*

instrukcje_wewnętrzne

end

Instrukcja wymaga zmiennej sterującej (nazwijmy ją ***licznikiem*** pętli), która w kolejnych powtórzeniach przyjmuje kolejno wartości określone w wektorze. Instrukcje wnętrza pętli mogą wykorzystywać wartość *licznika*.

Oto prosty przykład:

Program	Wynik

clc,clear
for k=[1 3 5 7 9]
   x = k^2
end
	x =
	   1
	x =
	   9
	x =
	   25
	x =
	   49
	x =
	   81

Wykorzystując w definicji *licznika* wieloelementowy wektor ciągu o stałym przyroście elementów, można tabelaryzować wartości funkcji:

Kod	Wynik

clc,clear
%tabelaryzacja funkcji sin(x) (0, 90) co 10 stopni
for x = 0:10:90
    y = sind(x); %dla kąta w stopniach funkcja sind
    fprintf('sin(%2d stopni)=%f\n', x, y);
end
	sin(0 stopni)=0.000000
	sin(10 stopni)=0.173648
	sin(20 stopni)=0.342020
	sin(30 stopni)=0.500000
	sin(40 stopni)=0.642788
	sin(50 stopni)=0.766044
	sin(60 stopni)=0.866025
	sin(70 stopni)=0.939693
	sin(80 stopni)=0.984808
	sin(90 stopni)=1.000000

Podobnie jak poprzednio, edytor *MATLAB-a* dodaje wcięcia instrukcji wewnętrznych pętli, dla poprawienia czytelności.

Pętla *for* może służyć do sekwencyjnego konstruowania elementów wektora o indeksach podawanych przy pomocy licznika pętli. Należy jednak pamiętać o wymaganym zakresie zmienności indeksów wektora (czyli kolejne liczby naturalne).

Wartości *licznika* mogą być wykorzystane w różnych celach:

Kod	Wynik

clc,clear
for k=1:10
    M(k)=k^2;
end
M
	M =
	    1 4 9 16 25 36 49 64 81 100

Zmienna sterująca *k* wystąpiła tu w podwójnej roli: posłużyła do indeksowania elementów wektora i do obliczania wartości jego elementów.

Powyższy kod, w którym macierz jest tworzona przy pomocy pętli, może zostać łatwo zastąpiony następującym:

Kod	Wynik

clc,clear
X = 1:10
Y1(X) = X.^2
Y2(X) = X.^3
	X=
	    1 2 3 4 5 6 7 8 9 10
	Y1 =
	    1 4 9 16 25 36 49 64 81 100
	Y2 =
	    1 8 27 64 125 216 343 512 729 1000

Zamiast seryjnych obliczeń w pętli, zastosowano wektor ***X*** i obliczony na jego podstawie, przy pomocy operacji tablicowego potęgowania, wektory ***Y1*** i ***Y2***. Taka metoda nosi nazwę **wektoryzacji**.

Można również tworzyć macierz dwuwymiarową, konieczne jest tu jednak zagnieżdżanie dwóch pętli (macierz prostokątna ma dwa wymiary).

Przykładowo:

Kod	Wynik

clc,clear
for wiersz = 1:3
  for kolumna = 1:5
    M(wiersz, kolumna) = 5; %twórz element
  end
end
M
	M =
	5 5 5 5 5
	5 5 5 5 5
	5 5 5 5 5

Działanie kodu polega na trzykrotnym wykonaniu instrukcji pętli zewnętrznej (wiersze macierzy), a instrukcją tą jest pętla wewnętrzna, która buduje kolumny danego wiersza. Stąd uzyskuje się $3\times5=15$ wykonań instrukcji definiującej element macierzy.

Wektoryzacja powyższego kodu miałaby prostą postać:

```
M(1:3,1:5) = 5
```

albo

```
M = 5*ones(3,5)
```

Trzeba pamiętać o "domykaniu" pętli *for* słowem kluczowym *end*, inaczej pojawi się komunikat o błędzie:

At least one END is missing

Co najmniej jeden brakujący END

Do generowania macierzy o elementach losowych wykorzystać można funkcję *rand.* Używanie pętli nie jest tu konieczne, ponieważ funkcja *rand* jest wyposażona w argumenty, którymi można określić rozmiar losowanej macierzy, a odpowiednimi wyrażeniami zmodyfikować zakres i typ losowanych liczb.

Przykładowo:

Kod	Wynik
clc,clear % losujemy macierz 2x4 % z liczbami całkowitymi z przedziału (-10, 10) M = round(20*rand(2, 4)-10)	M = 9 6 -2 6 0 -7 8 9

Przy pomocy zagnieżdżanych pętli *for* można tworzyć macierz dwuwymiarową o elementach będących wyrażeniami, wykorzystującymi zmienne liczników pętli.

Kod	Wynik

clc,clear
%zagnieżdżane pętle
for w = 1:2
  for k = 1:3
    M(w, k) = 2*w+k^2;
  end
end
M
	M =
	3 6 11
	5 8 13

Zrozumienie działania zagnieżdżanych pętli może być łatwiejsze, jeżeli zastosowana zostanie funkcja *pause*:

pause

która zatrzymuje bieg programu – ponowne uruchomienie wykonuje się naciśnięciem dowolnego klawisza klawiatury (przy aktywnym oknie *Command Window*), natomiast:

pause (*n*)

to zatrzymanie programu na ***n*** sekund.

Wykonanie skryptu:

Kod	Wynik

clc,clear
% element wypisywany co 1 sekundę
for w = 1:2
  for k = 1:3
    M(w, k) = 2*w+k^2
    pause(1)
  end
end
	M =
	3
	(tu będzie 1 sek. przerwy)
	M =
	3 6
	(tu będzie 1 sek. przerwy)
	M =
	3 6 11
	(tu będzie 1 sek. przerwy)
	M =
	3 6 11
	5 0 0
	.....*itd*

spowoduje konstruowanie w oknie *Command Window* kolejnych elementów macierzy, w odstępach co jedną sekundę.

Funkcja *pause(n)* może być wykorzystywana do animacji.

Oto przykładowy kod:

```
clc,clear
hold on
axis([0 20 0 400])
for  w=0:20
  plot(w,w.^2,'ro')
  pause(0.1)
end
```

którego działanie polega na dorysowywaniu jednego punktu wykresu w postaci czerwonych kółeczek (argument 'ro'), w odstępach co 0.1 sekundy. Funkcja rysująca *plot* będzie zaprezentowana szczegółowo w kolejnym rozdziale.

Pętla wewnętrzna może mieć niekiedy zmienianą wartość zakresów wektora licznika, na przykład uzależnioną od licznika pętli zewnętrznej:

Kod	Wynik

clc,clear
for wiersz = 1:3
  for kolumna = 1:wiersz
    M(wiersz, kolumna) = 5;
  end
end
M
	M =
	5 0 0
	5 5 0
	5 5 5

W powyższym przykładzie pętla wewnętrzna dla licznika *wiersz* równego 1 wykona się jeden raz (*kolumna*=1:1), dla równego 2 – dwa razy (*kolumna*=1:2), a dla równego 3 – trzy razy (*kolumna*=1:3). Definiowane jest zatem tylko 6 elementów macierzy, pozostałe zostaną automatycznie uzupełnione zerami (macierz musi być prostokątna).

Można też wykonać powyższe działanie bez użycia iteracji:

M = tril(5*ones(3))

Funkcja *tril* tworzy dolny trójkąt z macierzy z "piątkami".

9.3.2. Pętla *while*

Instrukcja *while* (ang. *dopóki*) to pętla warunkowa, również realizująca **wielokrotne wykonanie bloku instrukcji**.

Poniżej postać ogólna instrukcji:

while *wyrażenie_logiczne*
instrukcje_wewnętrzne (wykonywane, gdy *wyrażenie* ma wartość *prawda*)
end

Rozpoczynając działanie, pętla *while* bada wartość *wyrażenia logicznego*. Jeżeli będzie ono miało wartość *prawda*, wykonywane będą instrukcje wewnętrzne. Po ich wykonaniu następuje kolejne badanie *wyrażenia logicznego*, itd. Jeżeli po kolejnych powtórzeniach *wyrażenie logiczne* osiągnie wartość *fałsz*, pętla kończy działanie, program przechodzi do wykonania następnej instrukcji.

A zatem, działanie pętli *while* polega na wielokrotnym wykonywaniu bloku instrukcji wewnętrznych, dopóty, **dopóki wyrażenie logiczne ma wartość *prawda*** (*logiczne 1*)., gdy wyrażenie osiągnie wartość *fałsz* (*logiczne 0*) pętla się kończy.

Pętla *while* jest stosowana w algorytmach, w których nie jest znana liczba powtórzeń, wymagane jest wielokrotne wykonywanie bloku instrukcji, aż do osiągnięcia konkretnego celu, opisanego wyrażeniem logicznym.

Należy uważać na następujące przypadki:

- przed zastosowaniem instrukcji trzeba zadbać, żeby wyrażenie logiczne było obliczalne,
- wartość wyrażenia logicznego jest każdorazowo sprawdzana przed wykonaniem instrukcji wewnętrznych pętli, stąd też możliwy jest przypadek, że wyrażenie to, już przy pierwszym sprawdzeniu będzie miało wartość *fałsz* i instrukcje wnętrza pętli nigdy się nie wykonają,
- instrukcje wewnętrzne muszą mieć wpływ na wartość wyrażenia logicznego, inaczej pętla będzie wykonywała instrukcje wewnętrzne nieskończenie wiele razy i nastąpi zawieszenie programu.

Przerwanie zawieszonego programu można spowodować naciskając kombinację klawiszy *CTRL*+C lub *CTRL+Break* (przy aktywnym oknie *Command Window*) lub użyć narzędzia *Quit Debugging* w menu *Editor*.

W poniższym przykładzie zastosowano pętlę *while* w celu wielokrotnego połowienia wartości zmiennej *x*, która na wstępie ma wartość 1. Po każdym kroku pętli wypisywana

jest nowa wartość *x*. Instrukcje wykonawcze pętli są ponawiane, dopóki wartość *x* jest większa od 0.01:

Kod	Wynik

clc,clear
x = 1;
% dziel przez 2 wielokrotnie
while x>0.01
    x = x/2;   %(1)
    fprintf('%f\n',x);   %(2)
end
	0.500000
	0.250000
	0.125000
	0.062500
	0.031250
	0.015625
	0.007813

Można zauważyć, że ostatnia wypisana wartość nie spełnia warunku (po przepołowieniu zażądaliśmy wypisania wartości *x*, a następnie ponownego sprawdzenie warunku). Gdybyśmy w powyższym kodzie zamienili miejscami instrukcje (1) i (2), to pierwsza wypisana wartość byłaby 1, a ostatnia 0.015625.

Przykład pętli "nieskończonej":

Kod	Wynik

clc, clear
%Petla nieskończona !
x = 1;
disp('Początkowo x=1');
while x>0
    x = x+1;
    disp(x)
end
	Początkowo x=1
	1
	2
	3
	
	..itd....wypisywane będą liczby naturalne...
	...aż do intmax czyli 2147483647

Pętla *while* jest często wykorzystywana do testu poprawności wprowadzania danych przez użytkownika. W przypadku niewłaściwych danych pętla ponawia działanie, żądając korekty. Dopóki dane są nieprawidłowe (wyrażenie logiczne musi zawierać warunek niepoprawności), żądanie jest ponawiane.

Wersja kodu:

Kod	Wynik
clc,clear haslo = input('Podaj hasło:','s'); while ~strcmp(haslo,'Matlab') haslo = input('Podaj poprawnie: ','s'); end fprintf('Nareszcie, hasło to: %s\n',haslo);	Podaj hasło:123 Podaj poprawnie:abc Podaj poprawnie:xyz Podaj poprawnie:Matlab Nareszcie, hasło to:Matlab

Wyjaśnienie: Program wczytuje podany przez użytkownika tekst (argument 's' funkcji *input*). Warunek tworzy funkcja *strcmp* porównująca dwa teksty. Funkcja zwraca *prawdę* gdy teksty są identyczne, trzeba zanegować wynik funkcji: jeśli *nieprawdą* jest zgodność tekstów, ponów prośbę o hasło. Gdy *status* jest *prawdą* pętla się kończy.

Funkcja *strcmp* rozróżnia wielkość liter.

Dokładny test poprawności podawanej przez użytkownika pojedynczej liczby jest bardziej skomplikowany.

Poniżej kod dla dociekliwych:

Kod	Wynik

clc,clear
x = input('Podaj liczbę:','s');
test=all(ismember(x, '0123456789+-.eEdD'));
while test==0
  x = input('Podaj poprawnie:','s');
  test = all(ismember(x, '0123456789+-.eEdD'));
end
liczba = str2double(x);
fprintf('Nareszcie: %f\n',liczba);
	Podaj liczbę:A
	Podaj poprawnie:5/6
	Podaj poprawnie:4e5
	Nareszcie: 400000.000000

Niekiedy w kodzie wnętrza pętli *for* lub *while* może być przydatna instrukcja:

break

która wymusza zakończenie działania pętli i wykonanie następnych instrukcji.

Wyjaśnia to prosty przykład:

Kod	Wynik

clc, clear
x = 1;
while x>0
  x = x+1;
  disp(x)
  if x>6
    break %zakończ pętlę
  end
end
disp('KONIEC')
	2
	3
	4
	5
	6
	7
	KONIEC

9.4. Instrukcja *try... catch...*

Instrukcja *try... catch...* ma zastosowanie dla wykonania działań programowych w przypadku, gdy podejrzewamy fragment programu o błąd. Może ona ułatwić detekcję błędu.

Postać ogólna instrukcji:

try
instrukcje_1 (instrukcje testowane)
catch
instrukcje_2 (wykonywane, gdy w *instrukcjach_1* jest błąd)
end

Jeżeli instrukcje testowane nie wykażą błędu, instrukcje bloku *catch* nie będą wykonane. W przypadku błędu, wykonane są *instrukcje_2*, zawierające odpowiedni komunikat lub inną akcję.

Prosty przykład:

Kod	Wynik

clc, clear
try
  M = [1 2 3; 5 6]
catch
  disp('Złe wymiary')
end
	Złe wymiary

Trzeba tu dodać, że instrukcja nie testuje błędów składniowych instrukcji (np. niedomknięte nawiasy), lecz jedynie błędy wykonania (nieznana zmienna lub funkcja, brak pliku, niewłaściwy typ danych, źle wymiarowana macierz itp.).

9.5. Wykorzystanie instrukcji strukturalnych do analizy i modyfikacji macierzy

W kolejnych podrozdziałach pokazano metody wykorzystania instrukcji iteracyjnych w zastosowaniu do wybranych operacji na macierzach. Podano jednocześnie alternatywne rozwiązania jakie stoją do dyspozycji użytkownika *MATLAB-a*, pozwalające na skrócony zapis z wykorzystaniem odpowiednich notacji i funkcji macierzowych.

9.5.1.Sumowanie warunkowe elementów macierzy

Jak odnotowano wyżej, sumę wszystkich elementów macierzy można łatwo wyznaczyć stosując funkcję *sum*:

Kod	Wynik

clc, clear
M = rand(3)
suma = sum(sum(M));
fprintf('Suma elementów=%f\n',suma);
	M =
	  0.8147 0.9134 0.2785
	  0.9058 0.6324 0.5469
	  0.1270 0.0975 0.9575
	Suma elementów=5.273665

Funkcja *sum* z jednym argumentem znajduje wektor sum cząstkowych kolumn, stąd zagnieżdżanie funkcji dla wyznaczenia sumy całkowitej. Można też stosować zamiennie zapis:

suma = **sum** (*M*, 'all')

Dla wektora (macierzy jednowymiarowej) jest prościej:

Kod	Wynik
clc, clear M = 1:6 suma = sum(M); fprintf('Suma elementów wektora=%d\n',suma);	M = 1 2 3 4 5 6 Suma elementów wektora=21

Jeżeli celem jest sumowanie elementów liczbowych w wybranym zakresie indeksów macierzy, z pomocą może przyjść instrukcja iteracyjna i warunkowa.

Przykład sumowania elementów wybranej kolumny:

Kod	Wynik
clc, clear M = [1 2 3; 4 5 6; 7 8 9] suma = 0; %wyzeruj sumę for w = 1:3 %wszystkie wiersze suma = suma+M(w,1);%zwiększ sumę o element (w,1) end fprintf('Suma 1-szej kolumny=%d\n',suma);	M = 1 2 3 4 5 6 7 8 9 Suma 1-szej kolumny=12

lub suma pewnego zakresu wierszy:

Kod	Wynik

clc, clear
M=[1 2 3; 4 5 6; 7 8 9]
suma = 0 ; %wyzeruj sumę
for w = 1:2 %wiersz od 1 do 2
  for k = 1:3 %wszystkie 3 kolumny
   suma = suma+M(w,k); %zwiększ sumę o element (w,k)
  end
end
fprintf('Suma wierszy 1 i 2 = %d\n', suma);
	M =
	1 2 3
	4 5 6
	7 8 9
	Suma wierszy 1 i 2 = 21

Sumowanie odbyło się w zagnieżdżanej pętli *for*, dla ustalonego zakresu indeksów. Należy zwrócić uwagę na konieczność **inicjacji zmiennej obliczającej sumę i wstępnego nadania jej wartości 0**.

Poznane wcześniej zasady umożliwiają wykonanie tej samej operacji bez użycia pętli:

suma = sum(sum (*M*(1:2, :)))

lub

suma = sum(*M* (1:2, :), 'all')

Jeżeli celem jest zsumowanie elementów macierzy, które spełniają pewien warunek:

Kod	Wynik

clc, clear
M=[1 2 -3; 4 -5 2;1 2 -4]
suma = 0; %wyzeruj sumę
for w = 1:3 %wszystkie wiersze
    for k = 1:3 %wszystkie kolumny
        if M(w,k)>0
           %zwiększ sumę o element
           suma = suma+M(w,k);
        end
    end
end
fprintf('Suma elementów dodatnich =%d\n', suma);
	M =
	1 2 -3
	4 -5 2
	1 2 -4
	Suma elementów dodatnich =12

Konieczne jest sprawdzanie warunku dla każdego elementu, stąd zastosowanie zagnieżdżanych pętli i wewnętrznej instrukcji warunkowej.

Poniżej przykład sumowania elementów o wartościach należących do wybranego przedziału:

Skrypt	Wynik

clc, clear
M = [5 5 0 ; 0 10 5;-10 1 5]
suma_w = 0;
for w = 1:3
   for k = 1:3
      if M(w,k)>4&M(w,k)<6
         suma_w = suma_w+M(w,k);
      end
   end
end
fprintf('Suma elementów (4,6) =%d\n', suma_w);
	M =
	   5 5 0
	   0 10 5
	 -10 1 5
	Suma elementów (4, 6) =20

Wyrażenia logiczne, zawierające macierze jako argumenty, dają w rezultacie macierz zer i jedynek logicznych. Zatem w poniższym skrypcie:

Skrypt	Wynik

clc, clear
M = [5 5 0 ; 0 10 5;-10 1 5]
M2= M>0
	M =
	   5 5 0
	   0 10 5
	 -10 1 5
	M2 =
	 3×3 logical array
	  1 1 0
	  0 1 1
	  0 1 1

Powyższy mechanizm można wykorzystać dla wyznaczenia sumy elementów macierzy spełniających warunek, stosując instrukcję:

suma_warunkowa = sum (*M* (*M*>4&*M*<6), 'all')

lub

suma_warunkowa = sum (*M.** (*M*>4&*M*<6), 'all')

W ten sposób obliczana jest suma wszystkich elementów macierzy, przy filtrowanych indeksach sumowanych elementów, według utworzonego warunku.

Sumowanie elementów leżących na przekątnej głównej można wykonać wykorzystując warunek równości indeksów wiersza i kolumny:

Kod	Wynik

`clc, clear`
`M=[10  1   1 ; 1  10  1 ; 1  1  20]`
`suma = 0;`
`for w = 1:3`
`   for k = 1:3`
`      if    w ==k`
`         suma = suma+M(w,k);`
`      end`
`   end`
`end`
`fprintf('Suma przekątnej głównej=%d\n', suma);`
	`M =`
	`   10     1     1`
	`    1    10     1`
	`    1     1    20`
	`Suma przekątnej głównej=40`

lub prościej przy pomocy pojedynczej pętli, sumowane są elementy o tych samych indeksach wiersza i kolumny (*n*, *n*):

Kod	Wynik

`clc, clear`
`M = [10  1  1 ; 1  10  1 ; 1  1  20]`
`suma=0;`
`for n = 1:3`
`   suma = suma+M(n,n);`
`end`
`fprintf('Suma przekątnej głównej=%d\n', suma);`
	`M =`
	`   10     1     1`
	`    1    10     1`
	`    1     1    20`
	`Suma przekątnej głównej=40`

lub jeszcze prościej, wykorzystując elementy wektora przekątnej głównej macierzy, uzyskanego przy pomocy funkcji *diag*, wyrażeniem:

$$suma_przekatnej = \mathbf{sum}(\mathbf{diag}(M))$$

a już najprościej – stosując funkcję *trace,* sumującą bezpośrednio elementy przekątnej głównej macierzy:

$$suma_przekatnej = \mathbf{trace}(M)$$

Przykładowe wykorzystanie funkcji powyższych metod:

Kod	Wynik

`clc, clear`
`M = [10  1  1 ; 1  10  1 ; 1  1  20]`
`suma1=sum(diag(M))`
`suma2=trace(M)`
	`M =`
	`   10     1     1`
	`    1    10     1`
	`    1     1    20`
	
	`suma1=`
	`   40`
	
	`suma2 =`
	`   40`

Poniżej realizacja obliczenia iteracyjnego sumy elementów przeciwprzekątnej dla macierzy kwadratowej:

Kod	Wynik

clc, clear
M = [10 1 1 ; 1 10 1 ; 1 1 20]
 	
suma=0;
liczba_kolumn = size(M,2);
for wiersz = 1:3
  suma = suma+M(wiersz, liczba_kolumn+1-wiersz);
end
 	
fprintf('Suma przeciwprzekątnej=%d\n', suma);
	M =
	   10 1 1
	    1 10 1
	    1 1 20
	
	Suma przeciwprzekątnej=12

Wykorzystano tu wyrażenie:

$$indeks_kolumny = liczba_kolumn + 1 - indeks_wiersza$$

opisujące zmienność indeksu kolumny (malejąco) wraz ze wzrastającym indeksem wierszy.

Poniżej pokazano wcześniejszy przykład po modyfikacji. Sumowane są tu elementy przeciwprzekątnej dla macierzy kwadratowej, przy podawanym przez użytkownika rozmiarze macierzy i zakresie losowanych liczb naturalnych:

Kod	Wynik

clc,clear
N = input('Podaj rozmiar macierzy kwadr.: ');
zakres = input('Podaj zakres liczb naturalnych: ');
M = round(zakres*rand(N))
 	
sumpp = 0;
for n = 1:N
   sumpp = sumpp+M(n, N+1-n);
 end
fprintf('Suma przeciwprzekątnej=%d\n',sumpp);
	Podaj rozmiar macierzy kwadr.: 3
	Podaj zakres liczb naturalnych: 10
	M =
	     8 0 7
	    10 8 8
	     7 9 7
	Suma przeciwprzekątnej=22

Jest też inny sposób obliczenia sumy elementów przeciwprzekątnej:

$$suma_przeciwprzek = \mathbf{trace}\,(\mathbf{rot90}\,(M))$$

w którym oblicza się sumę elementów przekątnej głównej, lecz po uprzednim obróceniu macierzy o kąt 90 stopni zgodnie z ruchem zegara, przy użyciu funkcji *rot90*.

Niekiedy zakres licznika pętli wewnętrznej może być zależny od licznika pętli zewnętrznej. Określając zakres zmienności licznika pętli wewnętrznej stosujemy wówczas wyrażenia, które zawierają zmienną licznika pętli zewnętrznej.

Poniższy skrypt wykonuje sumowanie elementów macierzy kwadratowej położonych poniżej przekątnej, wraz z tą przekątną, czyli tak zwany trójkąt dolny macierzy:

```
clc,clear
M = [2  1  1 ; 2  2  1 ; 2   2   2]

suma = 0;
for w = 1:3
   for k = 1:w
      suma = suma+M(w, k);
   end
 end
fprintf('Suma trójkąta dolnego=%d\n', suma);
```

```
M =
    2    1    1
    2    2    1
    2    2    2
Suma trójkąta dolnego=12
```

Powyższy algorytm iteracyjny można zastąpić krótkim zapisem:

sumaTdolnego = **sum**(**tril**(*M*), 'all')

gdzie, jak wspomniano, funkcja *tril* wyodrębnia z macierzy trójkąt dolny, zerując pozostałe elementy.

Zadanie zsumowania wartości elementów "krawędziowych" macierzy wymaga przemyślenia, jaki sposób wybrać. Można oczywiście obliczyć sumę czterech sum (pierwszy i ostatni wiersz, pierwsza i ostatnia kolumna) i nie zapomnieć o odjęciu elementów "narożnych", gdyż każdy z nich wystąpił dwukrotnie w wartości sumy.

Prostszy jednak jest poniższy algorytm, w którym od sumy wszystkich elementów można odjąć sumę elementów "wnętrza" macierzy:

```
clc,clear
M = [1  1  1  1; 1  2  2  1; 1  2  2  1; 1  1  1  1]
suma_wnetrza = 0;  %zerujemy sumę "wnętrza"
rozmiary = size(M)

%od 2-go do przedostatniego...
for w = 2:size(M,1)-1
   for k = 2:size(M,2)-1
      suma_wnetrza = suma_wnetrza+M(w, k);
   end
 end
sumak = sum(M,'all')-suma_wnetrza;
fprintf('Suma krawędzi=%d\n',sumak);
```

```
M =
    1    1    1    1
    1    2    2    1
    1    2    2    1
    1    1    1    1
rozmiary =
    4    4
Suma krawędzi=12
```

Do ograniczeń indeksów wykorzystano funkcję *size*, zwracającą wektor rozmiarów macierzy, czyli dla przypomnienia:

size(*M*,1) - określa liczbę wierszy macierzy *M*,

natomiast:

size(*M*,2) - określa liczbę kolumn macierzy *M*.

Można również wyzerować wewnętrzne elementy macierzy:

```
M(2:end-1, 2:end-1) = 0
```

a następni obliczyć sumę jej elementów.

Najkrótsza, choć nie tak prosta, realizacja powyższego zadania przedstawia się następująco:

suma_krawedzi = sum(*M*,'all') - sum(*M*(2:size(*M*,1)-1,2:size(*M*,2) -1),'all')

Od sumy całkowitej odjęto sumę wnętrza macierzy, które jest opisane indeksami od pierwszego od przedostatniego (od 2 do *rozmiar*-1).

Jak widać z powyższych rozważań, nawet tak stosunkowo proste zadanie daje możliwość wyboru spośród kilku algorytmów prowadzących do prawidłowego rozwiązania. Warto poświęcić chwilę na zastanowienie się czy nie ma sposobu prostszego lub też szybszego.

9.5.2. Zliczanie warunkowe elementów macierzy

Zadanie policzenia elementów macierzy spełniających pewien warunek wymaga **inicjacji zmiennej i nadania jej zerowej wartości**. Zmienna ta będzie reprezentowała liczbę liczonych elementów. W zagnieżdżanych pętlach przeprowadza się badanie warunku instrukcją *if*.

Jeżeli warunek jest *prawdą*, wartość zmiennej liczącej zostaje zwiększona o 1:

```
clc,clear
M = [1 -1 -1; 1 -2  1; 1 -2  2 ]
ile = 0;  %zerujemy zmienną liczącą
for w = 1:size(M,1)
   for k = 1:size(M,2)
      if M(w,k)>0
        ile = ile+1;     %zwiększ o 1
      end
   end
 end
fprintf('Elementów dodatnich jest =%d\n', ile);
```

```
M =
    1   -1   -1
    1   -2    1
    1   -2    2
Elementów dodatnich jest =5
```

Jeżeli zadanie polega na policzeniu liczby elementów, które znajdują się w określonych przedziałach, można korzystać z zapisów:

iled = sum(*M*>0,'all')

ilez = sum(*M* ==0,'all')

ileu = sum(*M*<0,'all')

a także dla wyznaczenia liczby elementów należących do przedziału:

ile = sum(*M*>-3&*M*<3,'all')

Wykorzystano tu opisaną wyżej zasadę, że wyrażenie logiczne z użyciem macierzy zwraca macierz zer i jedynek logicznych, obliczenie sumy jedynek znajduje liczbę elementów spełniających warunek.

Można również zastosować iterację *for* i odpowiednio sformułowaną instrukcję warunkową *if*:

```
clc,clear
M = [1 -1 -1; 0 -2  1;0 -2  2 ]
% trzy zmienne liczące
ilezer = 0; iledod = 0; ileuj = 0;
for w = 1:size(M,1)
   for k = 1:size(M,2)
      if    M(w,k)>0
        iledod = iledod+1;
      elseif   M(w,k) ==0
        ilezer = ilezer+1;
      else
        ileuj = ileuj+1;
      end
   end
 end
fprintf('Elementów dodatnich jest =%d\n',iledod);
fprintf('Elementów ujemnych jest =%d\n',ileuj);
fprintf('Elementów zerowych jest =%d\n',ilezer);
```

```
M =
    1   -1   -1
    0   -2    1
    0   -2    2
Elementów dodatnich jest =3
Elementów ujemnych jest =4
Elementów zerowych jest =2
```

Poszczególne warunki można badać w innej kolejności.

Należy przypomnieć, że w przypadku zliczania w pętli elementów należących do przedziału (*a*, *b*) stosuje się zapis warunku:

M(w,k)>a&M(w,k)<b

9.5.3.Wyszukiwanie liczb w wektorze

Jeżeli istnieje wektor z wieloma liczbami i celem jest sprawdzenie, czy dana liczba znajduje się w wektorze, ile razy występuje i jakie są indeksy elementów zawierających tę liczbę, można wykorzystać opisaną już funkcję *find*.

Poniżej prosty przykład wyjaśniający zastosowanie funkcji *find* – wyszukiwanie w macierzy indeksów elementów, których wartości są większe od 10:

```
clc,clear
M = [ 19  18  1  15  5  8  11   19   8   20]
%indeksy elementów >10
INDEKSY = find(M>10)
fprintf('Liczba wystąpień liczb >10:%d\n', ...
       length(INDEKSY))
%indeksy elementów równych 0
zera = find(M==0)
```

```
M =
   19  18  1  15   5   8  11  19   8  20
INDEKSY =
    1     2     4     7     8    10
Liczba wystąpień liczb >10: 6
zera =
   1×0 empty double row vector
```

W macierzy nie ma wartości zerowych – otrzymano pusty wektor.

Można też zrealizować powyższe zadanie, wykorzystując pętlę *for*:

Kod	Wynik

clc,clear
M = [19 18 1 15 5 8 11   19   8   20]
% inicjujemy indeks macierzy INDEKSY
m = 1;
for k = 1:length(M)
   if M(k)>10
      INDEKSY(m) = k;
      m = m+1; % następny indeks
   end
end
disp('Indeksy elementów >10');
disp(INDEKSY)
	M =
	19 18 1 15 5 8 11 19 8 20
	Indeksy elementów >10
	1 2 4 7 8 10

Z powyższych przykładów wynika, że wiele algorytmów wymagających analizy macierzy liczbowych, przeprowadzić można przy pomocy zagnieżdżanych pętli i instrukcji warunkowej. Zorientowanie *MATLAB-a* na operacje macierzowe daje możliwość zastosowania specjalistycznych funkcji pozwalających na uproszczenie analiz macierzy.

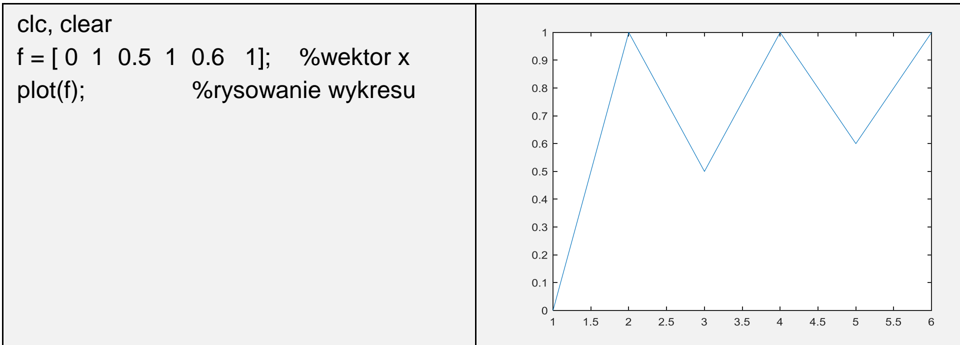
10. Tworzenie wykresów

10.1. Wykresy dwuwymiarowe

10.1.1. Funkcja *plot*

Funkcja *plot* wymaga przygotowania danych w wektorach. W najprostszej wersji funkcja posiada jeden argument, wektor danych:

plot(*f*)



```
clc, clear
f = [ 0  1  0.5  1  0.6   1];    %wektor x
plot(f);               %rysowanie wykresu
```


Zmienną zależną (oś pionowa) stanowią wartości elementów wektora, zmienną niezależną (oś pozioma) określają indeksy tego wektora, kolejne liczby naturalne. Dla wektora jednoelementowego zostanie narysowany jeden punkt.

Jeżeli zastosowane zostaną dwa argumenty funkcji:

plot(*x*, *y*)

to *x* i *y* muszą być wektorami o tej samej długości, zawierającymi współrzędne punktów wykresu. Współrzędne te są określone przez kolejne pary liczb, pobierane z obydwu wektorów – elementy o tych samych indeksach. Poszczególne punkty są łączone odcinkami prostej.

Dla wyboru parametrów wykresu można dodać kolejny argument funkcji *plot*:

plot(*x*, *y*, '*S*')

gdzie *S* jest ciągiem znaków.

Znaki te są symbolami typu linii, koloru i oznakowania punktów.

Typy linii:

-	*linia ciągła*	--	*linia kreskowana*	:	*linia kropkowana*

Kolory:

b	*niebieski*	g	*zielony*	r	*czerwony*
y	*żółty*	k	*czarny*	m	*magenta*

Oznakowania punktów wykresu:

+	*plus*	*	*gwiazdka*
o	*kólko*	s	*kwadrat*

Poniżej przykładowe oznaczenia:

```
plot(x, y, 'r')        % linia koloru czerwonego
plot(x, y, '-ko')      % linia ciągła i kółka w punktach, kolor czarny
plot(x, y, '*')        % tylko gwiazdki w punktach
plot(x, y, '-g')       % linia kreskowana, koloru zielonego
```

Ciągi znaków w oznaczeniach cech linii powinny być zamknięte w apostrofach. W przypadku braku oznaczeń, *MATLAB* wybiera parametry domyślne.

Więcej informacji wraz z przykładami pojawi się w *Command Window* po wydaniu polecenia:

help plot

Jeżeli celem jest narysowanie wykresu dwóch lub więcej krzywych w tym samym układzie współrzędnych, stosuje się dodatkowe argumenty funkcji (pary wektorów dla każdej krzywej – (x1,y1), (x2, y2) ... , według zasady:

plot (*x1*, *y1*, *x2*, *y2*,...*itd.*)

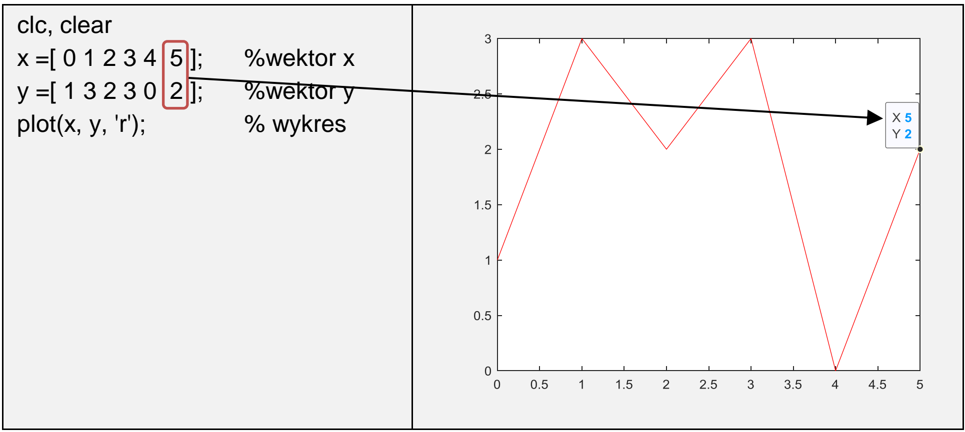
lub

plot (*x1*, *y1*, '*s1*', *x2*, *y2*, '*s2*', ..*itd.*)

gdzie *s1* i *s2* to, jak uprzednio, ciągi znaków określające parametry linii, punktów i kolorów.

Zazwyczaj wektor *x* jest wspólny dla wielu krzywych, lecz niekoniecznie.

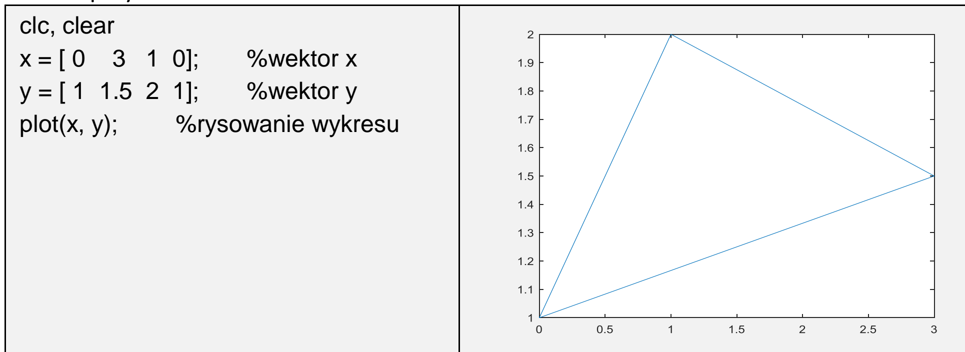
Poniżej skrypt, który tworzy prosty wykres, na podstawie danych umieszczonych w dwóch wektorach:



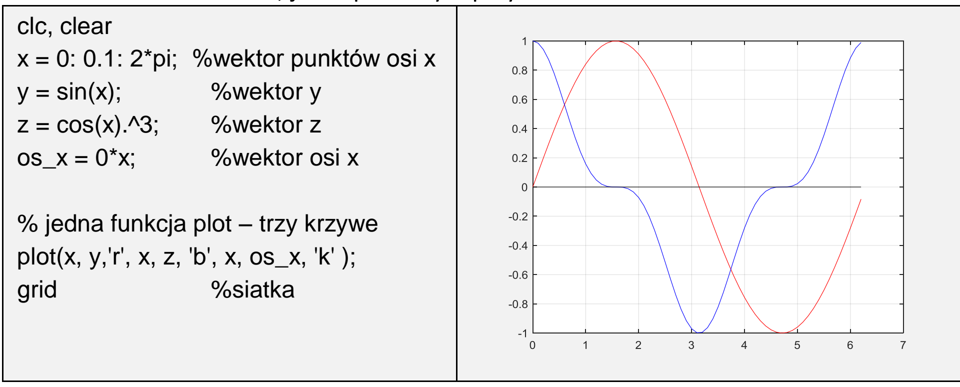
Rezultatem skryptu będzie pojawienie się okna *Figure*, w którym rysowana jest krzywa łamana, łącząca odcinkami prostej punkty o odpowiednich współrzędnych.

Można utworzyć wykres konturu zamkniętego, należy wówczas domknąć kontur, powtarzając współrzędne punktu początkowego i końcowego.

Oto przykład:



Rysowanie "gładkich" wykresów funkcji matematycznych wymaga "gęstej" osi *x* (wektor o wielu elementach, według postępu arytmetycznego) oraz wykonania funkcji na elementach wektora *x*, jak w poniższym przykładzie:



Należy zwrócić uwagę na zastosowanie **operatów tablicowych** dla wyznaczenia wektora zmiennej zależnej, na podstawie elementów wektora zmiennej niezależnej. Dodatkowo dodano do wykresu oś *x* (wykres dla wektora z elementami zerowymi).

W *MATLAB-ie* istnieją funkcje wspomagające rysowanie wykresów (tabele 10.1, 10.2 i 10.3).

Tabela 10.1. Funkcje stosowane przed wykonaniem funkcji rysującej *plot*:

hold on	instrukcję stosuje się, aby wielokrotne użycia funkcji *plot* dorysowywały nowe krzywe do jednego układu współrzędnych, nie usuwając poprzednich, nazwijmy to "zamrożeniem" układu współrzędnych
hold off	wyłącza dorysowywanie krzywych do wykresu, nowa krzywa zastąpi poprzednią, "odmrożenie" układu
figure(*n*)	tworzenie okna *Figure* o numerze *n*, dla innego wykresu
subplot(*m, n, p*)	Podział okna *Figure* w celu narysowania wykresów w jednym oknie lecz w kilku różnych układach współrzędnych *m*, *n* – liczba wierszy i kolumn podziału, *p* – numer aktualnego wykresu składowego.

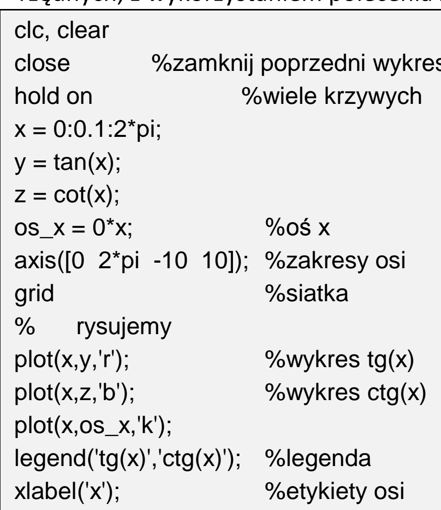
Tabela 10.2. Funkcje używane przed lub po instrukcji rysującej *plot:*

clf	wymuszenie usunięcia krzywych z okna *Figure*
close	usuwanie okna *Figure*
grid **grid on** **grid off**	dodaje lub usuwa siatkę do wykresu
title(*'text'*)	dodaje tytuł wykresu
xlabel(*'tekst'*)	dodaje etykietę dla osi zmiennej niezależnej
ylabel(*'tekst'*)	dodaje etykietę do osi zmiennej zależnej

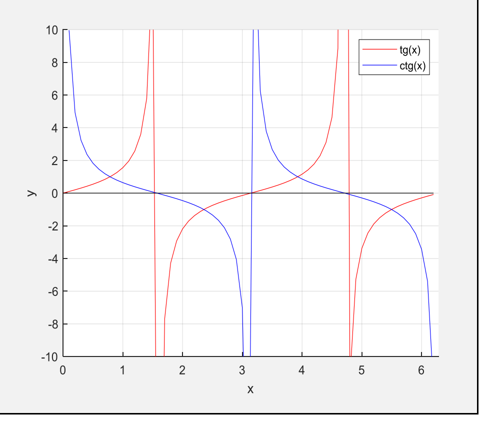
Tabela 10.3. Funkcje używane po wykonaniu funkcji rysującej (*plot* lub innej):

legend(*'tekst1', 'tekst2', ...itd.*)	dodaje legendę do wykresu (liczba tekstów jest równa liczbie krzywych).
axis([*xmin xmax ymin ymax*])	ustala zakresy osi *x* i *y*
axis equal	wymusza jednakowe skalowanie obydwu osi

Poniżej zamieszczono przykład rysowania wielu krzywych w jednym układzie współrzędnych, z wykorzystaniem polecenia *hold on* oraz kilku instrukcji *plot*:

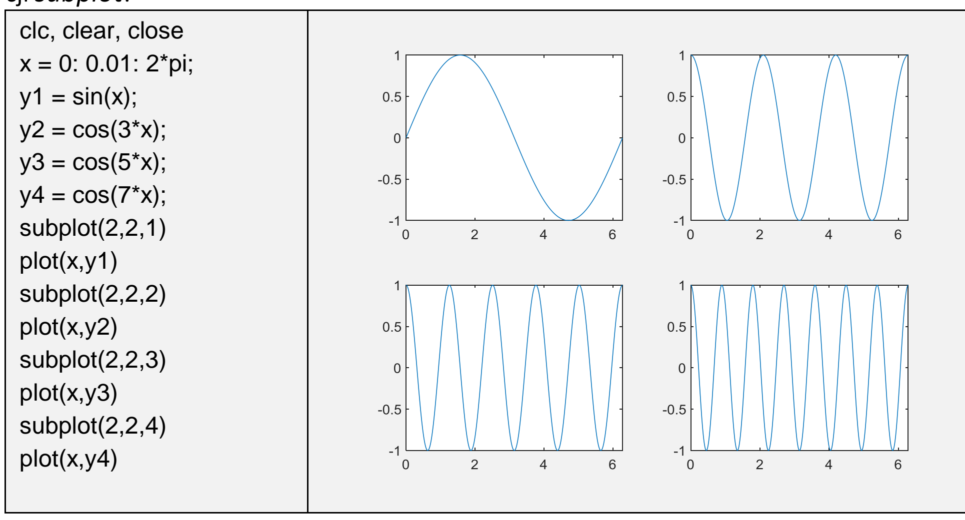


```
clc, clear
close          %zamknij poprzedni wykres
hold on                  %wiele krzywych
x = 0:0.1:2*pi;
y = tan(x);
z = cot(x);
os_x = 0*x;              %oś x
axis([0  2*pi  -10  10]); %zakresy osi
grid                     %siatka
%      rysujemy
plot(x,y,'r');           %wykres tg(x)
plot(x,z,'b');           %wykres ctg(x)
plot(x,os_x,'k');
legend('tg(x)','ctg(x)'); %legenda
xlabel('x');             %etykiety osi
ylabel('y');
```



Po wykonaniu wykresu, w okienku *Figure* dostępne są ikony narzędzi do przesuwania i skalowania wykresu (lupa). W menu *Edit* okna *Figure*, opcja *Copy Figure* pozwala na skopiowanie wykresu do schowka i wykorzystanie go w opracowywanej przez użytkownika dokumentacji.

Poniżej przykład podziału okna na kilka układów współrzędnych z wykorzystaniem funkcji *subplot*:



```
clc, clear, close
x = 0: 0.01: 2*pi;
y1 = sin(x);
y2 = cos(3*x);
y3 = cos(5*x);
y4 = cos(7*x);
subplot(2,2,1)
plot(x,y1)
subplot(2,2,2)
plot(x,y2)
subplot(2,2,3)
plot(x,y3)
subplot(2,2,4)
plot(x,y4)
```

10.1.2. Funkcja *fplot*

Funkcja *fplot* służy do szybkiego rysowania wykresu i nie wymaga uprzedniego przygotowania wektorów danych. Postać ogólna funkcji (w starszych wersjach MATLAB-a, akceptowana w nowych wydaniach):

fplot (' *f(x)* ') - x ∈ (-5, 5)

fplot (' *f(x)* ', [x_{min} x_{max}]) - dla jednej krzywej

fplot (' [$f_1(x)$, $f_2(x)$, ...] ', [x_{min} x_{max}]) - dla wielu krzywych

Zamiast identyfikatora *x* można użyć innej nazwy zmiennej. Opisana wyrażeniem funkcja (lub wektor zawierający wyrażenia funkcyjne) powinny być domknięty przy pomocy apostrofów.

Poniższy skrypt ilustruje możliwości zastosowania funkcji *fplot* oraz funkcji pomocniczych, tworzących siatkę, ustalających zakresy osi i legendę dla krzywych wykresu:

```
clc, clear,close
hold on  %będzie kilka krzywych
%krzywa 1
fplot('1/x^2' , [ 0  20] );
%krzywe 2, 3 i oś x
fplot('[log(x)  log10(x)  0*x]', [0 20]);

grid;                    %siatka
axis([0 20 -3 3]);       %zakresy osi
legend('1/x^2', 'ln(x)', 'log10(x)');
```

Gęstość punktów osi *x* i kolory krzywych są domyślne. *MATLAB* umożliwia ich modyfikację dzięki zastosowaniu dodatkowych argumentów. specyfikacje są podobne do przytoczonych wyżej. Przykładowo:

```
fplot ('exp(x)',[-3 0],'+r')
```

Przy tworzeniu wyrażeń opisujących krzywe wykresu w funkcji *fplot*, nie jest wymagane stosowanie operatorów tablicowych (.*, ./ i .^).

Stosując funkcję *fplot* w najnowszych wersjach *MATLAB-a*, w oknie *Command Window* pojawia się ostrzeżenie (ang. *warning*), że kolejne wydania programu będą wymagały zastosowania tzw. *funkcji anonimowej*, w przykładowej postaci:

```
fplot (@(x) (sin(x).^2+1), [0 2*pi])
```

Opis definiowania i stosowania funkcji anonimowej zamieszczony zostanie w rozdz.11.1.

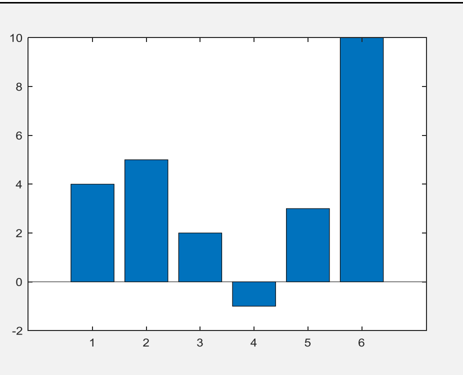
Jako argument funkcji *fplot* może też być wykorzystywana funkcja opisana symbolicznie, co zostanie omówione w rozdz. 12.

10.1.3. Inne funkcje tworzące wykresy

MATLAB udostępnia wiele funkcji rysujących wykresy płaskie.

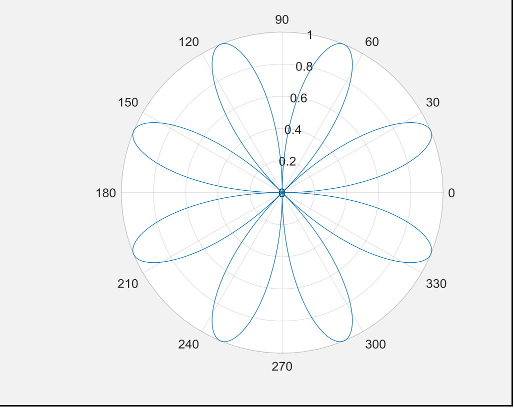
- funkcja *bar*, tworząca wykres słupkowy wektora:

```
clc,clear
x = [ 4  5  2  -1  3  10];
bar(x);
```



- funkcja *polarplot*, tworząca wykresy we współrzędnych biegunowych:

```
clc, clear
fi = 0:0.01:2*pi;
r = sin(4*fi);
polarplot(fi,r);
```

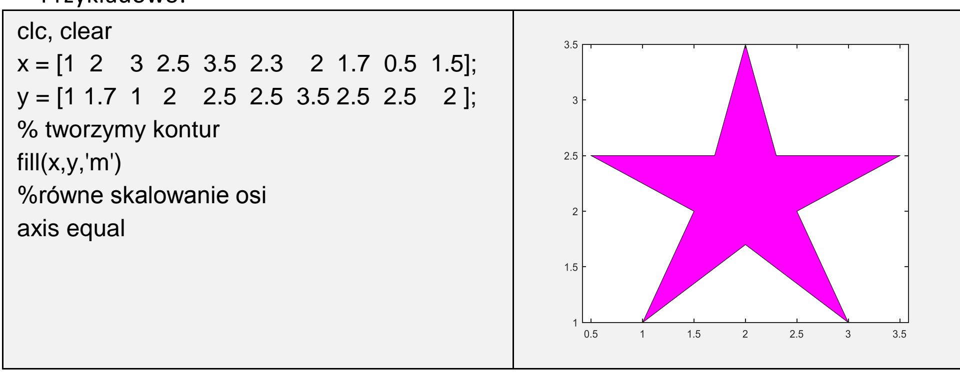


- funkcja *fill*, służąca do tworzenia konturów zamkniętych, o postaci ogólnej:

  fill (*x*, *y*, *kolor*)

Funkcje *polarplot* i *fill* wymagają dwóch wektorów z danymi. Argument *kolor* (identycznie jak dla funkcji *plot*: *'k', 'r', 'b', 'g'* i inne) jest kolorem wypełnienia konturu. Funkcja *fill* automatycznie domyka kontur, łącząc punkty o współrzędnych pierwszej pary danych z wektorów ***x*** i ***y*** z ostatnią.

Przykładowo:



```
clc, clear
x = [1  2    3  2.5  3.5  2.3    2  1.7  0.5  1.5];
y = [1 1.7  1   2    2.5  2.5  3.5 2.5  2.5    2 ];
% tworzymy kontur
fill(x,y,'m')
%równe skalowanie osi
axis equal
```


Istnieją możliwości ustalania grubości, koloru i stylu linii, dodawanie etykiet na wykresach itp. W celu poszerzenia umiejętności w tym zakresie odsyłamy czytelnika do dokumentacji *MATLAB-a*.

10.2. Wykresy trójwymiarowe

10.2.1. Krzywe w przestrzeni trójwymiarowej

Tworzenie wykresów krzywych w przestrzeni trójwymiarowej umożliwia funkcja *plot3*, o postaci:

plot3 (*x*, *y*, *z*)

Przykładowo dla krzywej danej równaniami:

$$y = e^{-0.1x} sin(5x)$$

$$z = e^{-0.1x} cos(5x)$$

należy, przed zastosowaniem funkcji *plot3*, utworzyć wektor zmiennej niezależnej o równomiernym rozkładzie elementów, a następnie zdefiniować dwie zmienne, przypisując im odpowiednie funkcje zmiennej niezależnej. Wektor *z* wyznacza dane dla osi pionowej, przy innej kolejności argumentów wykres będzie odpowiednio obrócony.

Utworzony *M-plik* pokazuje sposób wykorzystania funkcji *plot3*:

```
clc,clear,close
x = 0: pi/50: 10*pi;
y = exp(-0.1*x).*sin(5*x);
z = exp(-0.1*x).*cos(5*x);
% linia krzywa w przestrzeni
% oś x będzie pionowa
plot3(y , z , x);
grid
```

Tu również zachodzi konieczność stosowania operatorów tablicowych przy mnożeniu wektorów.

10.2.2. Powierzchnie w przestrzeni trójwymiarowej

Powierzchnie w przestrzeni definiowane są równaniem funkcji dwóch zmiennych. *MATLAB* dostarcza dla narysowania wykresów przestrzennych poniższe funkcje rysujące:

mesh (*x*, *y*, *z*)

lub:

surf (*x, y, z*)

Funkcja *mesh* rysuje siatkę powierzchni, funkcja *surf* wypełnia oczka siatki kolorami.

Przed narysowaniem wykresu należy utworzyć "podstawę" *x, y* wykresu w postaci odpowiednich macierzy, wykorzystując funkcję *meshgrid*:

[*x y*] = **meshgrid**(*wektorX*, *wektorY*)

gdzie: *wektorX* i *wektorY* – dwa wektory rozkładu równomiernego dla osi płaszczyzny poziomej,

x i *y* – dwie specjalne macierze, wymagane do określenia funkcji dwóch zmiennych (metoda przypisania rezultatu funkcji *meshgrid* do dwóch zmiennych).

Przykładowa instrukcja:

```
[x  y] = meshgrid( 0:0.1:2*pi,  0:0.1:2*pi )
```

tworzy sieć punktów dla osi *x* i *y* o równomiernym rozkładzie z krokiem co 0.1, w przedziałach (0, 2π).

Korzystanie z funkcji *meshgrid* ułatwia tworzenie dwóch macierzy dla obydwu argumentów funkcji *z(x,y)*, inaczej wymagane byłyby macierze, które trzeba tworzyć poniższym, bardziej złożonym przepisem:

```
px = 0: 0.1: pi;
py = 0: 0.1: 2*pi;                %wektory  dla osi x i  y
x = ones(length(py),1)*px
y = (ones(length(px),1)*py)'
```

W następnej kolejności tworzona jest definicja funkcji dwóch zmiennych:

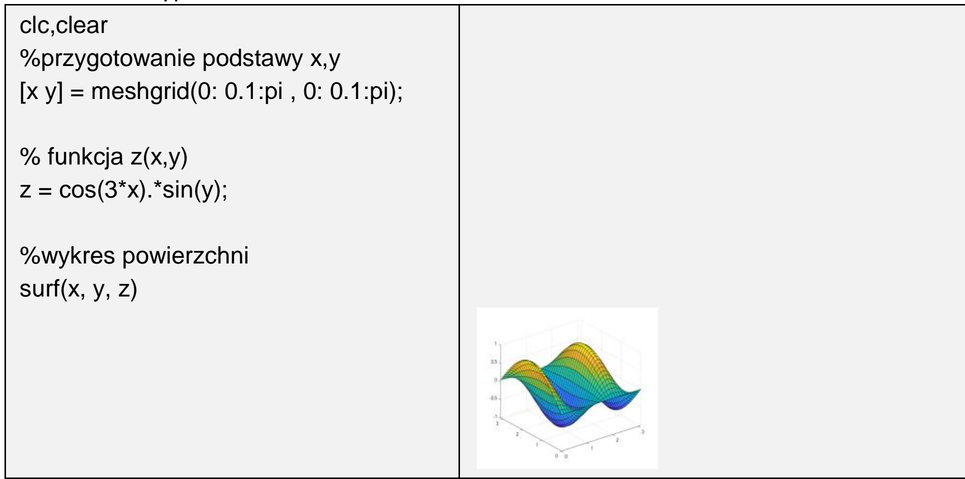
$$z = f(x, y)$$

i wykonywany jest wykres, z wykorzystaniem funkcji *mesh* bądź *surf.*

Przykładowo, dla powierzchni danej równaniem:

$$z = cos3x \;\; siny$$

utworzono skrypt:



```
clc,clear
%przygotowanie podstawy x,y
[x y] = meshgrid(0: 0.1:pi , 0: 0.1:pi);

% funkcja z(x,y)
z = cos(3*x).*sin(y);

%wykres powierzchni
surf(x, y, z)
```

Należy ponownie zwrócić uwagę, że w wyrażeniu *f(x,y)* konieczne jest używanie operatorów tablicowych mnożenia (.*), dzielenia (./) i potęgowania (.^).

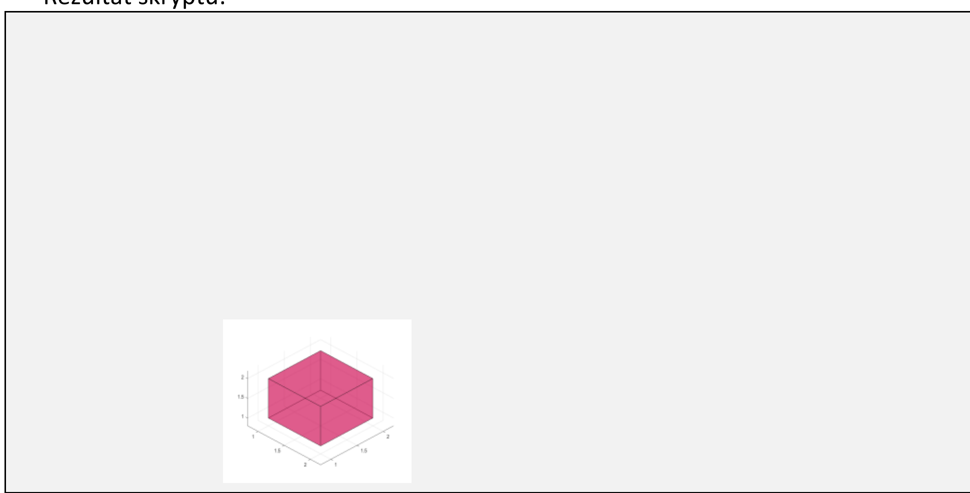
Jeszcze ciekawy przykład realizacji wykresu *torusa*, który zostawiamy czytelnikowi do własnej analizy.

```
clc, clear
R = 40;
r = 15;
%dwa zakresy kątów pełnych
u = 0: 10: 360;
v = 0: 10: 360;
[u,v] = meshgrid(u, v);
%trzy funkcje (u ,v)
x = (R+r*cosd(v)).*cosd(u);
y = (R+r*cosd(v)).*sind(u);
z = r*sind(v);
grid

%rysuj wykres
surf(x, y, z);
axis equal %równe skalowanie osi
```

Poniżej kolejny przykład grafiki trójwymiarowej, ilustrujący poszczególne etapy żmudnego konstruowania obrazu figury geometrycznej, złożonej z wielokątów:

```
clc,clear
hold on
 [x y] = meshgrid(1:2,1:2)                %tworzymy siatkę podstawy wykresu
%tworzymy kolejne ściany
surf(x, y, ones(2))                       %ściana dolna
surf(x, y, 2*ones(2))                     %ściana górna
[x y] = meshgrid(1:2, [1 1])
surf(x, y, [1  1 ; 2  2])                 %ściana lewy przód
[x y] = meshgrid(1:2, [2  2])
surf(x, y, [1  1 ; 2  2])                 %ściana lewy tył
[x y] = meshgrid([1 1], 1:2)
surf(x, y, [1  2 ; 1  2 ])                %ściana prawy tył
[x y] = meshgrid([2 2], 1:2)
surf(x, y, [1  2; 1  2 ])                 %ściana prawy przód
colormap([0.8  0  0.3])                   % 3 składowe RGB kolorów
axis([0.8, 2.2, 0.8, 2.2, 0.8, 2.2])      %zakresy osi x y z
grid
view(45,45),alpha(0.4)                    %kąty perspektywy i przezroczystość
```

Rezultat skryptu:



W skrypcie ustalono pespektywę patrzenia na wykres, wykorzystując funkcję *view*:

view(*stopnie1, stopnie2*)

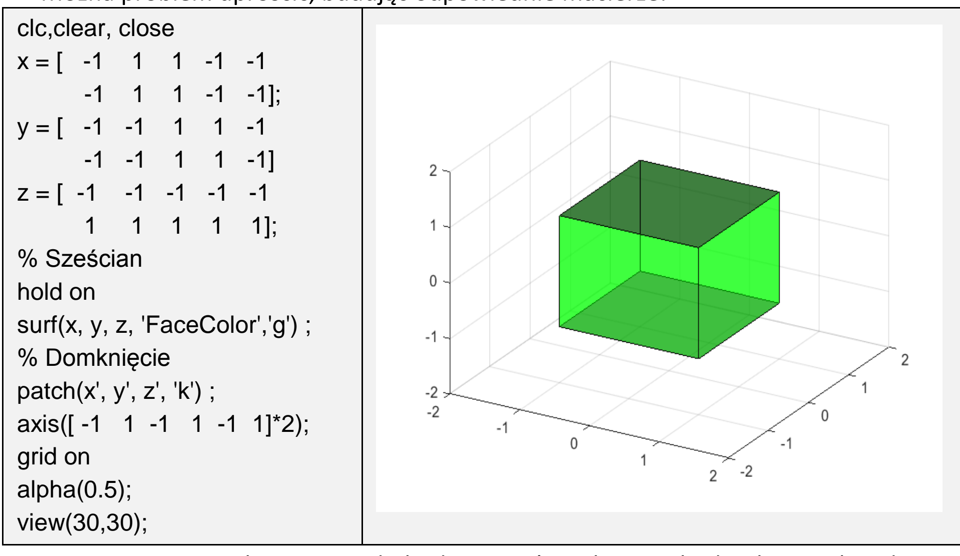
Funkcją *colormap* ustalono kolor ścian (wektor składowych [*red green blue*], każda składowa to liczba dziesiętna w przedziale od 0 do 1), natomiast funkcja *alpha* określa współczynnik przezroczystości:

alpha(*m*)

gdzie $m \in (0, 1)$.

Po narysowaniu wykresu można korzystać z narzędzi skalowania i obracania w oknie *Figure.*

Można problem uprościć, budując odpowiednie macierze:



```
clc,clear, close
x = [  -1    1    1   -1   -1
       -1    1    1   -1   -1];
y = [  -1   -1    1    1   -1
       -1   -1    1    1   -1]
z = [ -1    -1   -1   -1   -1
        1    1    1    1    1];
% Sześcian
hold on
surf(x, y, z, 'FaceColor','g') ;
% Domknięcie
patch(x', y', z', 'k') ;
axis([ -1   1  -1   1  -1  1]*2);
grid on
alpha(0.5);
view(30,30);
```

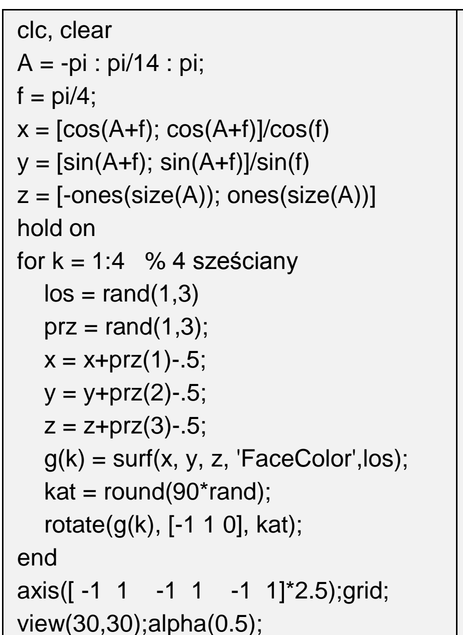
Macierze *x*, *y*, *z* wykorzystano do budowania ścian bocznych, domknięcie kwadratem górnym i dolnym wykonano funkcją *patch,* która rysuje wielokąty (użyto tych samych macierzy po transpozycji).

Analizując postać macierzy *x*, *y*, *z* zauważamy okresowość jej elementów. Można zatem wykorzystać funkcje trygonometryczne:

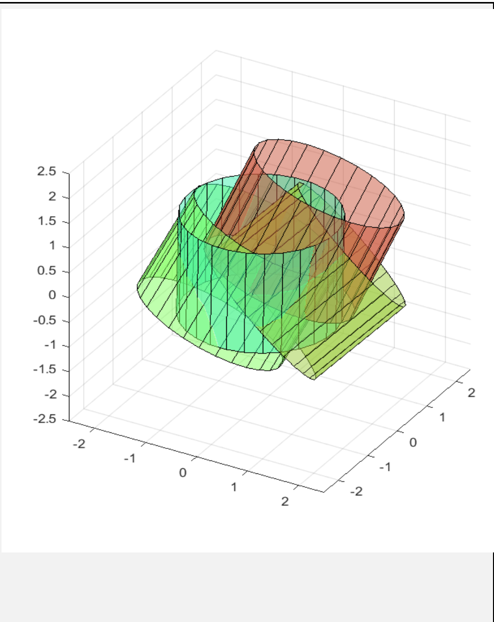
```
clc, clear, close
A = -pi : pi/2 : pi;         % Narożniki
faza = pi/4;                 % Faza
x = [cos(A+faza); cos(A+faza)] / cos(faza)
y = [sin(A+faza); sin(A+faza)] / sin(faza)
z = [-ones(size(A)); ones(size(A))]
```

Na podstawie powyższego przepisu można zastosować iterację do narysowania kilku sześcianów, o losowych przesunięciach, losowych kolorach ścian i losowych kątach obrotu.

Zmiana przyrostu elementów wektora *A* spowoduje utworzenie graniastosłupa o podstawie wielokąta foremnego, niewielki przyrost przybliży figurę do walca. Ilustruje to następny kod skryptu i jego rezultat graficzny.



```
clc, clear
A = -pi : pi/14 : pi;
f = pi/4;
x = [cos(A+f); cos(A+f)]/cos(f)
y = [sin(A+f); sin(A+f)]/sin(f)
z = [-ones(size(A)); ones(size(A))]
hold on
for k = 1:4   % 4 sześciany
    los = rand(1,3)
    prz = rand(1,3);
    x = x+prz(1)-.5;
    y = y+prz(2)-.5;
    z = z+prz(3)-.5;
    g(k) = surf(x, y, z, 'FaceColor',los);
    kat = round(90*rand);
    rotate(g(k), [-1 1 0], kat);
end
axis([ -1  1    -1  1    -1  1]*2.5);grid;
view(30,30);alpha(0.5);
```



Każdą macierz dwuwymiarową można przedstawić w postaci powierzchni trójwymiarowej, w której indeksy macierzy są współrzędnymi ***x*** i ***y***, a wartości elementów są współrzędnymi ***z***, przykładowo:

```
clc,clear
M = [0    0   0   0  0
       0  14 14 14  0
       0  14 14 14  0
       0  14 14 14  0
       0    0   0   0  0]
surf(M)
%zakresy trzech osi
axis([0  6  0  6  0  20])
view(145,30)
```

```
M =
    0     0     0     0     0
    0    14    14    14     0
    0    14    14    14     0
    0    14    14    14     0
    0     0     0     0     0
```

10.3.Przykładowe zadanie

Poniżej przykład, w którym problem iteracyjny (wielokrotne powtórzenia) będzie zawierał zastosowanie także instrukcji warunkowej, operacji na wektorach oraz reprezentacji graficznej rezultatu działań.

Zadanie polega na wykonaniu obliczeń i skonstruowaniu graficznej prezentacji fraktala zwanego *smokiem Heighwaya.*

Algorytm wyrażony pseudokodem, czyli opisany słownie, jest następujący:

1. Założyć dowolne wartości początkowe wektorów x(1) i y(1).
2. Wielokrotnie wykonać następujące operacje:

- wybrać z jednakowym prawdopodobieństwem jeden z dwóch poniższych układów równań:

  x(k+1) = -0.4*x(k) -1
  y(k+1) = -0.4*y(k)+0.1

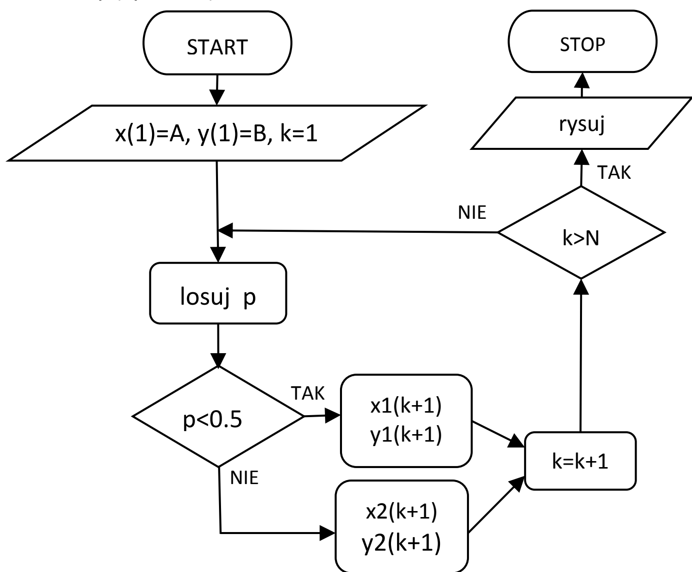
  lub

  x(k+1) = 0.76*x(k) – 0.4*y(k)
  y(k+1) = 0.4*x(k)+0.76*y(k)

  i obliczyć wartości następnych elementów wektorów: x(k+1) i y(k+1), na podstawie elementów poprzednich,
- przechować obliczone wartości w wektorze.

3. Narysować wykres y(x).

Można wspomóc zrozumienie algorytmu i proces jego realizacji w *MATLAB-ie*, tworząc schemat blokowy (rys.10.1).



Rys.10.1. Schemat blokowy zadania

Po ustaleniu danych:

- początkowych wartości dwóch wektorów *x* i *y*,
- zakresu licznika iteracji,

losowana jest liczba *p*, przy pomocy funkcji *rand*.

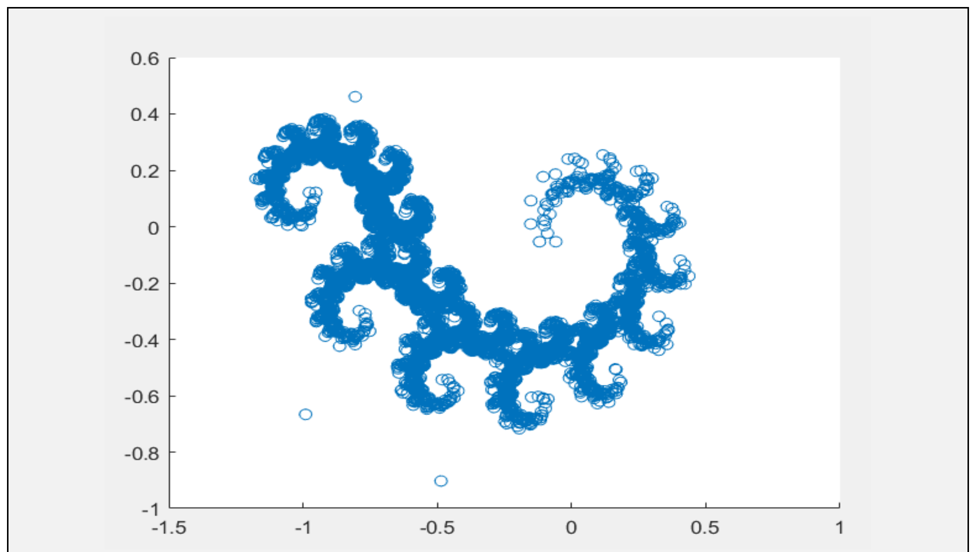
Schemat blokowy zawiera dwa elementy decyzyjne. Jeden z nich bada przedział wylosowanej liczby ***p***, tu zastosowana zostanie instrukcja warunkowa. Drugi blok decyzyjny schematu sprawdza zakres licznika, za to odpowiada pętla. Instrukcje wewnętrzne pętli mają za zadanie generowanie kolejnych elementów wektorów, według podanych wyrażeń, w zależności od zakresu wylosowanej liczby.

Operacja jest ponawiana dla kolejnej wylosowanej liczby ***p***. Po wyczerpaniu licznika iteracji następuje narysowanie wykresu według wartości wygenerowanych wektorów.

Kod i rezultat graficzny przedstawiono poniżej:

```
clc,clear, close
% dane
x(1) = 10; y(1) = 10;
N = 50000;
% iteracja ("pętla")
for k = 1: N
    p = rand;
    if p<0.5
      x(k+1) = -0.4*x(k)-1;
      y(k+1) = -0.4*y(k)+0.1;
    else
      x(k+1) = 0.76*x(k)-0.4*y(k);
      y(k+1) = 0.4*x(k)+0.76*y(k);
    end
end
%wykres
plot(x,y,'o')
axis([-1.5 1 -1 0.6])
```

Poszczególne punkty wykresu wykonano kółkami, aby uniknąć ich łączenia odcinkami prostej. Po pięćdziesięciu tysiącach iteracji *smok Heighwaya* wystarczająco ładnie się prezentuje.


0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1.5
-1
-0.5
0
0.5
1

11. Definiowanie funkcji przez użytkownika

Pisząc użyteczne skrypty, wykorzystuje się wiele funkcji standardowych *MATLAB-a* (funkcje matematyczne, funkcje operacji na macierzach, tworzenia wykresów, funkcje wejścia/wyjścia, systemu plików i inne). Istnieje też możliwość tworzenia definicji własnych funkcji i ich późniejszego, wielokrotnego wykorzystania.

11.1. Funkcja anonimowa

Funkcja anonimowa to sposób definiowania wyrażeń, które obliczane są na podstawie danych wejściowych. Funkcja anonimowa jest definiowana lokalnie w skrypcie i może w nim wykorzystana, a poza nim tylko jako argument funkcji zewnętrznej.

Definicja funkcji anonimowej jest następująca:

nazwa_funkcji = @**(***lista_argumentów***)** *wektor_wyrażeń_wyjściowych*

Lista argumentów zawiera nazwy zmiennych, oddzielane przecinkami. Wyrażenia wyjściowe, będące rezultatami funkcji, zawierają zmienne z listy argumentów.

Definicja prostej funkcji, której rezultatem będzie obliczenie pola powierzchni okręgu:

```
poleOkregu = @(R)pi*R^2
```

Można też pisać wyrażenie obliczeniowe w nawiasie, co jest bardziej czytelne:

```
poleOkregu = @(R)(pi*R^2)
```

W dalszych instrukcjach tego samego *M-pliku* można tę funkcję wielokrotnie wykorzystać, z różnymi wartościami argumentu (reprezentującego promień okręgu), także w wyrażeniach:

```
clc,clear
%definicja funkcji anonimowej
poleOkregu = @(R)pi*R^2 %definicja

%zastosowania funkcji anonimowej
pole1 = poleOkregu(2.1)
pole2 = poleOkregu(5.6)
%różnica pól dwóch okręgów
roznica_pol = poleOkregu(6)- poleOkregu(2);
fprintf('Ok6-Ok2 :%7.2f\n', roznica_pol);
```

```
poleOkregu =
    function_handle with value:
   @(R)pi*R^2
pole1 =
  13.8544
pole2 =
  98.5203
Ok6-Ok2 : 100.53
```

Jest możliwość dodatkowego dołączenia parametrów w definicji anonimowej. Pokazuje to poniższy skrypt:

```
clc,clear
a = 3;
b = 4.5;
f = @(x)a*sin(x)+b*cos(x)^2 %definicja
wynik = f(pi/4)
```

```
f =
  function_handle with value:
    @(x)a*sin(x)+b*cos(x)^2
wynik =
    4.3713
```

Jeżeli wymagamy, aby funkcja miała kilka argumentów i kilka wyrażeń obliczeniowych, wyrażenia te definiuje się jako elementy wektora.

Kod	Wynik

clc,clear
%definicja funkcji anonimowej
prostokat = @(a,b) [a*b   2*(a+b)] ;
%wykorzystanie funkcji
wynik = prostokat(3, 4);
fprintf('Pole prostokąta o bokach 3 i 4 = %d\n',...
                wynik(1));
fprintf('Obwód prostokąta =%d\n', wynik(2));
	Pole prostokąta o bokach 3 i 4 =12
	Obwód prostokąta =14

W powyższym przykładzie funkcja posiada dwa argumenty (boki prostokąta) i zwraca dwa rezultaty w wektorze, pierwszym jest powierzchnia prostokąta, a drugim długość obwodu.

Jak wspomniano, funkcja *fplot* rysująca wykresy, jako argument może zawierać alternatywnie tekstowy zapis funkcji, funkcję anonimową lub symboliczną, przyszłe wydania *MATLAB-a* będą już wymagały tylko funkcji anonimowej lub symbolicznej. Poniżej wzorce zastosowań funkcji anonimowej jako argumentu funkcji *fplot*:

```
fplot(@(x)sin(x) )                              - przedział x∈(-5,5),
fplot(@(x)[sin(x), cos(x)] )                    - dwie krzywe,
fplot(@(x)[sin(x), cos(x)] , [0  2*pi] )        - dodatkowo zakres zmiennej x.
```

11.2. Funkcja w zewnętrznym *M-pliku*

Funkcje wbudowane *MATLAB-a* są *M-plikami* znajdującymi się w katalogach instalacyjnych programu. Funkcje te są opisane specjalnymi algorytmami (np. sumowaniem pewnej liczby wyrazów rozwinięcia funkcji w szereg).

Jeżeli definicja naszej autorskiej funkcji zostanie zapisana w *M-pliku* umieszczonym we własnym katalogu roboczym, to w tym samym katalogu można utworzyć skrypt lub skrypty korzystające z tej funkcji.

Użytkownik może zdefiniować własne funkcje w zewnętrznych *M-plikach*, umieścić je w katalogach przeszukiwań *MATLAB-a*. Jak już wyżej nadmieniono, listę ścieżek do katalogów przeszukiwań uzyskać można wpisując w *Command Window* polecenie *path*:

```
>>path
```

Można pliki definiujące funkcje przenieść do jednego z katalogów przeszukiwań lub stworzyć własny katalog i dołączyć go do ścieżek przeszukiwań przykładowymi poleceniami:

```
>>mkdir  E:/moje_funkcje
>>path(path, 'E:/moje_funkcje')
```

Po tej operacji można będzie już korzystać z pliku funkcyjnego w każdym, tworzonym przez nas skrypcie.

Plik z własną definicją funkcji tworzy się tak, jak inne *M-pliki*. Łatwiej jednak zastosować pozycję menu wstęgi *Home/New/Function* (w starszych wersjach *MATLAB-a* kliknąć prawym klawiszem myszki w oknie *Current Directory*, z menu podręcznego wybrać *New/M-file* i nadać plikowi nazwę według zasad *MATLAB-a*).

Po tych czynnościach i otwarciu *M-pliku*, w oknie *Editor* pojawi się szkielet definicji funkcji w postaci:

```
function [outputArg1, outputArg2] = Untitled (inputArg1, inputArg2)
   % Untitled Summary of this function goes here
   %   Detailed explanation goes here
   outputArg1 = inputArg1;
   outputArg2 = inputArg2;
end
```

lub w starszych wersjach *MATLAB-a*:

```
function [ output_args ] = Untitled ( input_args )
 % Untitled Summary of this function goes here
 %   Detailed explanation goes here
```

Zamieszczone poniżej nagłówka komentarze w języku angielskim można przeczytać i skasować, bądź umieścić własne.

Zasady tworzenia definicji funkcji:

- nagłówek definicji funkcji zawiera słowo kluczowe *function*,
- trzeba ustalić nazwę funkcji według znanych zasad (w miejsce domyślnej nazwy *Untitled*),
- argumenty wejściowe (*inputArgs*), zwane **argumentami formalnymi**, to zmienne reprezentujące dane przekazywane do funkcji – lista w nawiasach okrągłych, trzeba ustalić ich nazwy,
- argumenty wyjściowe (*outputArgs*) to zmienne rezultatów funkcji (umieszczone w wektorze), również trzeba ustalić ich nazwy,
- w treści funkcji powinny zostać umieszczone instrukcje prowadzące do obliczeniowego powiązania zmiennych rezultatów (*outputArgs*) z danymi (*inputArgs*); można w tym celu wykorzystywać wyrażenia arytmetyczne, pętle, analizy warunkowe, definiować zmienne pomocnicze, korzystać z innych funkcji własnych itp.,
- słowo kluczowe *end* nie jest konieczne.

Zdefiniujmy prostą funkcję, dla obliczania objętości kuli o promieniu ***r*** :

```
function [ob] = Kula ( r )
ob = 4/3*pi*r^3;
```

Trzeba zapisać plik nadając mu nazwę *Kula.m* (**plik musi mieć taką samą nazwę jak funkcja** – także uwzględniając wielkość liter).

Posiadając definicję funkcji można utworzyć nowy skrypt, którym sprawdzi działanie funkcji *Kula*:

clc, clear wynik = Kula(5)	wynik = 523.5988

albo wprowadzając dodatkowo interakcję z użytkownikiem, w celu podania danej:

Kod	Wynik
clc, clear R = input('Podaj promień kuli:'); wynik = Kula(R); fprintf('Objętość tej kuli = %f\n',wynik);	Podaj promień kuli:5 Objętość tej kuli = 523.598776

Skrypt wykorzystuje zdefiniowaną funkcję z **argumentami aktualnymi**. Mogą nimi być stałe, zmienne o znanej wartości lub obliczalne wyrażenia.

Oto jak działa komunikacja skryptu korzystającego z funkcji z plikiem definicji funkcji:

- po wykryciu w skrypcie żądania wykonania funkcji, poszukiwany jest plik tej funkcji w katalogu bieżącym lub katalogach przeszukiwań,
- następnie odbywa się przekazanie wartości argumentów wejściowych (argumentów aktualnych) ze skryptu do funkcji, do jej lokalnych argumentów formalnych, w kolejności jak na liście,
- po obliczeniu rezultatów, ich wartości są zwracane do miejsca wywołania funkcji.

Wykorzystanie utworzonej funkcji odbywa się w identyczny sposób jak funkcji wbudowanych *MATLAB-a*. Wartość obliczoną przez funkcję można użyć w wyrażeniu (zgodnym z typem obliczonej wartości).

Czasem może istnieć definicja funkcji, która nie posiada argumentów wyjściowych (np. funkcje rysujące wykres, drukujące itp.). Wówczas wykonanie takiej funkcji w skrypcie jest realizowane przy pomocy osobnej instrukcji:

nazwa_funkcji (*argumenty_aktualne*)

Argumenty aktualne (wykonania funkcji) i argumenty formalne (w definicji) powinny być **zgodne**:

- co do ich **liczby** (w każdym razie nie może być więcej argumentów aktualnych niż formalnych, czasem może być mniej, ale to trzeba uwzględnić w definicji funkcji),
- co do ich **kolejności** (czyli znaczenia, jaką wielkość reprezentują),
- co do **typu** (jeżeli argumentem jest np. skalar czy macierz, czasem tekst).

Rezultat wykonania funkcji może być przypisywany do zmiennej (jak w naszym przykładzie) lub wykorzystywany w złożonych obliczeniach. Funkcja może być wykorzystana wielokrotnie, z różnymi wartościami argumentów aktualnych.

Modyfikacja definicji funkcji, uwzględniająca obliczanie objętości oraz pola powierzchni kuli:

```
function [ob, pole] = Kula( r )
ob = 4/3*pi*r^3;
pole = 4*pi*r^2;
```

Po każdej modyfikacji plik definicyjny funkcji musi zostać zapisany.

Funkcja zwraca dwa rezultaty, zatem istnieje konieczność przechowania wyników wykonania funkcji w dwóch zmiennych (przypisanie do wektora zmiennych).

Czynimy to odpowiednim przypisaniem, jak w poniższym przykładzie:

clc, clear R = input('Podaj promień kuli: '); %przypisanie wyników do dwóch zmiennych [o, p] = Kula (R); fprintf('Objętość tej kuli = %0.4f\n', o); fprintf('Pole powierzchni kuli = %0.4f\n', p);	Podaj promień kuli: 3.5 Objętość tej kuli = 179.5944 Pole powierzchni kuli = 153.9380

Należy zwrócić uwagę, że istotna jest tu znajomość kolejności zmiennych rezultatów funkcji w definicji, aby przy korzystaniu z funkcji nie pomylić wartości objętości kuli z jej polem powierzchni. A zatem należy uważać na liczbę argumentów wejściowych, ich typ i kolejność, a także liczbę i kolejność rezultatów.

Jeżeli wymagana jest definicja funkcji o wielu rezultatach, wygodniej argumenty wyjściowe funkcji umieścić w wektorze, jak w poniższym w przykładzie:

```
function [wyniki] = Kula ( r )
wyniki(1) = 4/3*pi*r^3;
wyniki(2) = 4*pi*r^2;
```

Wykorzystanie tak zdefiniowanej funkcję wymaga indeksowania wektora rezultatów:

clc, clear R = input('Podaj promień kuli:'); % K będzie wektorem wyników K = Kula (R); % dostęp do elementów wektora K fprintf('Objętość tej kuli = %f\n', K(1)); fprintf('Pole powierzchni kuli = %f\n', K(2));	Podaj promień kuli:6.7 Objętość tej kuli = 1259.833108 Pole powierzchni kuli = 564.104377

Poniżej zamieszczono przykład funkcji wieloargumentowej, służącej do obliczania pola powierzchni trójkąta o zadanych długościach trzech boków, według wzoru *Herona*:

$$\text{pole} = \sqrt{p(p-a)(p-b)(p-c)}$$

gdzie p jest połową obwodu trójkąta.

Definicję funkcji w pliku *Heron.m*:

```
function [pole] = Heron(a, b, c)
po = (a+b+c)/2;
pole = sqrt(po*(po-a)*(po-b)*(po-c));
```

Skrypt wykorzystujący zdefiniowaną funkcję może mieć postać jak poniżej:

clc, clear
disp('Pole trójkąta według Herona');
bok1 = input('Podaj długość boku 1:');
bok2 = input('Podaj długość boku 2:');
bok3 = input('Podaj długość boku 3:');
wynik = Heron(bok1, bok2, bok3);
fprintf('Pole powierzchni to %0.3f\n', wynik);
	Pole trójkąta według Herona
	Podaj długość boku 1: 2
	Podaj długość boku 2: 3
	Podaj długość boku 3: 4
	Pole powierzchni to 2.905

Istnieje jednak warunek poprawności konstruowania trójkąta: suma każdej pary boków musi być większa od boku trzeciego. Należy uwzględnić ten fakt, badając powyższy warunek w definicji funkcji, przy pomocy instrukcji warunkowej *if*:

```
function [pole] = Heron(a, b, c)
po = (a+b+c)/2;
if   a+b>=c&a+c>=b&b+c>=a     %warunek możliwej konstrukcji trójkąta
   pole = sqrt(po*(po-a)*(po-b)*(po-c));
else
   pole = -1;
end
```

Wyrażenie logiczne to koniunkcja trzech relacji logicznych. Założono dla uproszczenia, że w przypadku braku możliwości zbudowania trójkąta, zmienna *pole* przyjmie wartość -1.

Zawsze powinno się dokładnie przetestować poprawność definicji funkcji, wykonując skrypt korzystający z funkcji dla kilku krytycznych zestawów danych:

```
clc, clear
disp('Pole trójkąta według Herona');
bok1 = input('Podaj długość boku 1:');
bok2 = input('Podaj długość boku 2:');
bok3 = input('Podaj długość boku 3:');
wynik = Heron(bok1, bok2, bok3);
fprintf('Pole powierzchni to %0.3f\n', wynik);
if  wynik<=0
   fprintf('\nTo nie jest trójkąt\n');
else
   fprintf('Pole powierzchni to: %0.3f\n', wynik);
end
```

```
Pole trójkąta według Herona
Podaj długość boku 1: 1
Podaj długość boku 2: 1
Podaj długość boku 3: 3

To nie jest trójkąt
```

W powyższym przykładzie kolejność argumentów wejściowych, reprezentujących długości boków trójkąta, nie ma znaczenia. Istnieją jednak sytuacje, gdzie jest to istotne. Przykładowo, definiując funkcję obliczającą objętość stożka przy zadanym promieniu podstawy i wysokości, ważna jest kolejność danych:

Definicja funkcji w pliku *stozek.m*:

```
function [ ob ] = stozek(r, h)
ob = pi*r^2*h/3;
```

Wykorzystanie zdefiniowanej funkcji w skrypcie:

```
clc, clear
disp('Objętość stożka');
R = input('Podaj promień podstawy:');
H = input('Podaj wysokość:');
wynik = stozek(R, H);
fprintf('Objętość stożka:%0.3f\n', wynik);
```

```
Objętość stożka
Podaj promień podstawy:3
Podaj wysokość:4
Objętość stożka:37.699
```

Argumenty wykonania funkcji *stozek* powinny być użyte w kolejności (*promień, wysokość*), inaczej wyniki będą nieprawidłowe.

Mogą istnieć definicje funkcji bezargumentowych, a także, jak już wyżej wspomniano, funkcje o zmiennej liczbie argumentów. Istnieje możliwość wykorzystania dostępnej w definicji funkcji zmiennej *nargin,* zawierającej liczbę argumentów. Wewnątrz definicji można testować wartość tej zmiennej.

Poniższy przykład definiuje funkcję, która podana bez argumentów utworzy jeden zakład lotto (6 liczb), a jeżeli uwzględnimy argument ***N*** (liczba naturalna), to wylosowane będzie ***N*** zakładów:

```
function [ Z ] = lotto(N)
switch nargin
 case 0
   Z = ceil(49*rand(1,6));
 case 1
   Z = ceil(49*rand(N,6));
end;
```

W definicji funkcji *lotto* zbadano wartość zmiennej *nargin* przy pomocy instrukcji przełącznika *switch*. Zastosowano funkcję *ceil*, zaokrąglającą wyrażenie do liczby całkowitej w górę, aby uniknąć losowania wartości 0.

Oto wykorzystanie zdefiniowanej funkcji w skrypcie:

clc, clear %jeden zakład LOTTO w1 = lotto % cztery zakłady LOTTO w2 = lotto(4)	w1 = 37 13 25 35 44 48 w2 = 27 42 46 31 29 38 7 12 18 24 27 37 8 40 10 18 45 19 13 12 13 41 15 28

Została wykorzystana funkcja *lotto* bez argumentów – można też pisać: *lotto*( ) – oraz z jednym argumentem.

Ostatni wylosowany zestaw liczb zawiera dwukrotne wystąpienie liczby 13, uniknięcie powtórzeń losowanych liczb wymagałoby utworzenia bardziej złożonej definicji funkcji, wyposażonej w algorytm usuwania powtórzeń lub z zastosowaniem funkcji *unique*.

Algorytm mógłby być zrealizowany następująco:

- sześć wylosowanych liczb jest gromadzonych w wektorze *L*,
- jeżeli *prawdą* jest warunek:

  length(unique(*L*))<length(*L*)

  czyli liczba unikalnych elementów wektora jest mniejsza od liczby elementów wylosowanych, to wektor zostaje pominięty, inaczej dopisywany jest jako nowy wiersz macierzy losowań,
- liczba wierszy macierzy może być argumentem naszej funkcji.

Próbę realizacji tego algorytmu z zastosowaniem definicji własnej funkcji pozostawiamy czytelnikowi.

Niekiedy zachodzi konieczność przekazywania do funkcji innych funkcji jako argumentów. Poniższy przykład wyjaśnia problem.

Jeżeli istnieje zewnętrzna definicja funkcji:

```
function  z = suma_kwadratow(f1, f2)
 % suma kwadratów dwóch argumentów
 z = f1^2+f2^2;
```

i celem jest wykonanie tej funkcji z argumentami funkcyjnymi, wykorzystujemy tu poznaną definicję funkcji anonimowej. Oto *M-plik* wykonawczy:

```
clc, clear
FUN1 = @(x)(cos(x));   %definicja funkcji anonimowej
FUN2 = @(x)(sin(x));   %definicja funkcji anonimowej
los = rand;
wynik = suma_kwadratow(FUN1(los), FUN2(los))
```

Zdefiniowane lokalnie w skrypcie dwie funkcje anonimowe *FUN1* i *FUN2* są argumentami aktualnymi wykonania zewnętrznej funkcji *suma_kwadratow* i są do niej przekazane. Można się przekonać, że wykonanie tej funkcji dla dowolnej, losowej wartości funkcji *sin(x)* i *cos(x)* da wartość jedynki trygonometrycznej.

Oczywiście funkcję *suma_kwadratów* można wykonać z innymi definicjami funkcji anonimowych jako argumentami, a także z argumentami liczbowymi.

Na koniec należy wspomnieć, że niekiedy definiując własną funkcję w osobnym pliku, należy wymusić (przy pewnym spełnionym warunku) zakończenie jej działania przed wykonaniem kolejnych instrukcji wchodzących w skład tej funkcji. Wówczas stosuje się polecenie:

return

które powoduje powrót do skryptu wykonującego funkcję, do miejsca wywołania.

12. Wykorzystanie pakietu *Symbolic Toolbox*

12.1. Możliwości pakietu

Korzystanie z zestawu narzędzi dla celów obliczeń symbolicznych wymaga dodatkowej instalacji pakietu ***Symbolic Toolbox*** (pakiet posiada osobną licencję). Obliczenia symboliczne wykonują operacje na zmiennych (typu symbolicznego *sym*), które **nie posiadają wartości**.

Wybrane możliwości pakietu *Symbolic Toolbox* to:

- przekształcenia i uproszczenia wyrażeń arytmetycznych,
- operacje na macierzach o elementach symbolicznych,
- wyznaczanie symbolicznych rozwiązań równań,
- podstawianie danych liczbowych do wyrażeń symbolicznych,
- obliczanie granic ciągów i funkcji,
- wyznaczanie pochodnych funkcji symbolicznych,
- wyznaczanie całek nieoznaczonych i obliczanie całek oznaczonych funkcji symbolicznych,
- rysowanie wykresów funkcji definiowanych symbolicznie.

12.2. Deklaracja zmiennych symbolicznych

Rozpoczynając pracę z wykorzystaniem pakietu *Symbolic Toolbox*, należy wstępnie zadeklarować potrzebne zmienne symboliczne. Późniejsze wyrażenia definiowane będą z wykorzystaniem tych zmiennych.

Deklaracja zmiennych elementarnych dokonywana jest przy pomocy funkcji *syms*:

syms ('*nazwa1*', '*nazwa2*', '*nazwa3*' ... *itd*.)

lub w wersji uproszczonej:

syms *nazwa nazwa2 nazwa3* ... *itd*.

Jeżeli istnieje potrzeba deklaracji wielu zmiennych symbolicznych do wypełnienia dużej macierzy, można zastosować poniższą funkcję *sym* (tylko w nowszych wersjach *MATLAB-a*), która ułatwia tworzenie wielu nazw dla elementów macierzy:

macierz = **sym** ('*litera*', [*liczba_wierszy*, *liczba_kolumn*])

12.3. Tworzenie funkcji i macierzy symbolicznych

Po zadeklarowaniu zmiennych symbolicznych, można definiować własne funkcje symboliczne, wiążąc te zmienne w wyrażenia tak, jakby to były wyrażenia arytmetyczne. Można również wypełniać macierze elementami symbolicznymi, a następnie wykonywać na nich działania, jak na przykład mnożenia, potęgowania, obliczenia wyznacznika, macierzy odwrotnej, wartości własnych itp.

Przykład skryptu:

clc,clear % deklarujemy dwie zmienne syms x y % definiujemy funkcje f = x^2 g = sin(x)*cos(3*y) % wypisz jakie mamy zmienne whos	f = x^2 g = sin(x)*cos(3*y) Name Size Bytes Class f 1x1 8 sym g 1x1 8 sym x 1x1 8 sym y 1x1 8 sym

Istnieje możliwość podstawiania wartości liczbowych do tak zdefiniowanych funkcji symbolicznych, rysowania ich wykresów, obliczania symbolicznie pochodnych i całek itp.

Macierz z elementami symbolicznymi generowana jest podobnie jak macierze liczbowe:

clc,clear syms a b c d M = [a b; c d]	M = [a, b] [c, d]

Przykładowy rezultat szybkiego utworzenia macierzy z nazwami seryjnymi:

clc,clear M = sym('a' , [3, 4])	M = [a1_1, a1_2, a1_3, a1_4] [a2_1, a2_2, a2_3, a2_4] [a3_1, a3_2, a3_3, a3_4]

12.4. Podstawienia wartości liczbowych do wyrażeń symbolicznych

W celu zastąpienia zmiennych symbolicznych wartościami liczbowymi, z pomocą przychodzi funkcja *subs*:

wyrażenie = **subs**(*wyrażenie_symboliczne*)

Jeżeli po zdefiniowaniu funkcji symbolicznej nastąpi nadanie zmiennym symbolicznym wartości liczbowych, funkcja *subs* podstawi je do wyrażenia. Wyrażenie wynikowe będzie również typu symbolicznego.

Wyjaśnia to poniższy przykład:

clc, clear syms a b %a i b symboliczne y = a*b; %wyrażenie symboliczne a = 2.5; %teraz a będzie liczbą w1 = subs(y) %podstawiamy do y a =5; %teraz a i b będą liczbami b =4.5; w2 = subs(y) %podstawiamy do y % konwersja w2 do typu liczbowego w3 = double(w2) whos %lista zmiennych i ich typów	y = a*b w1 = (5*b)/2 w2 = 45/2 w3 = 22.5000 Name Size Bytes Class a 1x1 8 double b 1x1 8 double w1 1x1 8 sym w2 1x1 8 sym w3 1x1 8 double y 1x1 8 sym

Rezultat polecenia *whos* pokazuje, że po podstawieniach wyrażenia przypisane zmiennym *w1* i *w2* pozostają typu symbolicznego. Jeżeli pakiet *Symbolic Toolbox* przedstawia liczby symboliczne w postaci wyrażeń ułamkowych, stosuje się funkcję *double*, która zmienia typ na liczbowy (*double*).

zmienna_liczbowa = **double**(*ułamek_symboliczny*)

Można także podstawiać dane liczbowe do macierzy o elementach symbolicznych, co pokazuje poniższy przykład:

clc, clear syms a b c d M = [a b; c d] %teraz a i d będą liczbami a = 5 d = 9 M2 = subs(M) w = M2(1,1)*M2(2,2)	M = [a, b] [c, d] a = 5 d = 9 M2 = [5, b] [c, 9] w = 45

12.5. Operacje na macierzach o elementach symbolicznych

Zamieszczone niżej przykłady pozwalają lepiej zrozumieć operacje wykorzystywane w rachunku macierzowym, na podstawie wyznaczonych symbolicznie wzorów ogólnych.

Obliczenie iloczynu macierzowego dwóch macierzy:

clc, clear %określamy zmienne symboliczne syms a b c d e f g h % wstawiamy jako elementy macierzy M1 = [a b; c d] M2 = [e f ; g h] W = M1*M2	M1 = [a, b] [c, d] M2 = [e, f] [g, h] W = [a*e+b*g, a*f+b*h] [c*e+d*g, c*f+d*h]

Obliczenie iloczynu tablicowego dwóch macierzy:

clc, clear syms a b c d e f g h M1 = [a b; c d] M2 = [e f ; g h] W = M1.*M2	M1 = [a, b] [c, d] M2 = [e, f] [g, h] W = [a*e, b*f] [c*g, d*h]

Obliczenie wyznacznika macierzy kwadratowej:

clc, clear syms a b c d e f g h i M = [a b c ; d e f ; g h i] wyznacznik = det(M)	M = [a, b, c] [d, e, f] [g, h, i] wyznacznik = a*e*i-a*f*h-d*b*i+c*d*h+g*b*f- c*e*g

Można też wykonywać symbolicznie inne operacje, np. obliczanie macierzy odwrotnej, dzielenie lewostronne dwóch macierzy, wyznaczanie wartości własnych macierzy (funkcja *eig*(A)) itp. Na macierzach zawierających elementy symboliczne dozwolone jest wykonywanie większości poznanych wcześniej funkcji operujących na macierzach liczbowych, takich jak *diag*, *reshape*, *size*, *length*, *tril* i innych.

12.6. Symboliczne rozwiązywanie równań

W celu znalezienia ogólnych wzorów na rozwiązania funkcji z jedną niewiadomą (znalezienie miejsc zerowych), stosuje się funkcję *solve* o postaci:

rozwiazanie = **solve**(*f* , *zmienna*)

gdzie: *f* – funkcja symboliczna, *zmienna* – zmienna niezależna.

Można też stosować w nowych wydaniach *MATLAB-a* zapis:

rozwiazanie = **solve**(*w* , *zmienna*)

gdzie: *w* jest równaniem o postaci:

$f_1 == f_2$ (f_1 i f_2 to funkcje symboliczne)

Jeżeli jest zdefiniowana funkcja jednej zmiennej (bez parametrów) lub jedną ze zmiennych jest ***x***, można pominąć drugi argument funkcji *solve*:

rozwiazanie = **solve**(*f*)

Przykładowa realizacja:

clc, clear syms a b c x f = a*x^2+b*x+c rozw = solve(f)	f = a*x^2 + b*x + c rozw = -(b + (b^2 – 4*a*c)^(1/2))/(2*a) -(b – (b^2 – 4*a*c)^(1/2))/(2*a)

Otrzymano ogólne wzory na miejsca zerowe równania drugiego stopnia. Funkcja *solve* znajduje wartości, dla których *f(x)=0*. Przy rozwiązywaniu równań wyższych stopni rezultaty mogą być bardziej złożonymi wyrażeniami.

Korzystając z funkcji *solve* można też rozwiązywać równania nieliniowe. Przykładowy skrypt ilustruje metody zastosowania funkcji *solve* (*MATLAB* 2019b):

clc, clear syms x w1 = sind(x) == cosd(x) kat = solve(w1) w2 = sind(x)-cosd(x) kat2 = solve(w2)	w1 = sind(x) == cosd(x) kat = 45 w2 = sin((pi*x)/180) – cos((pi*x)/180) kat2 = 45

Otrzymano wartość kąta równą 45 stopni, współrzędną *x* punktu przecięcia funkcji *sin(x)* i *cos(x)*.

Niektóre rozwiązania mogą być wyrażeniami zespolonymi. Można zażądać tylko wartości rzeczywistych, stosując poniższą wersję funkcji *solve*:

rozwiazanie = **solve**(*f*, 'Real', true)

Przykład zastosowania w skrypcie:

```
clc, clear
syms  x a
f =  sin(x) - a*cos(x)
%Tylko rozwiązania rzeczywiste
S = solve(f, 'Real', true)
spr1 = sin(S(1))-a*cos(S(1))
spr2 = simplify(spr1) %uprość wyrażenie
```

oraz rezultat jego wykonania:

```
f =
   sin(x) – a*cos(x)
S =
   2*atan(((a^2 + 1)^(1/2) – 1)/a)
 - 2*atan(((a^2 + 1)^(1/2) + 1)/a)
spr1 =
   sin(2*atan(((a^2 + 1)^(1/2) – 1)/a)) – a*cos(2*atan(((a^2 + 1)^(1/2) – 1)/a))
spr2 =
    0
```

Rozwiązania zwracane są w macierzy *S* o elementach typu symbolicznego.

W podanym przykładzie wykonano sprawdzenie pierwszego z rozwiązań *S(1)*, przez podstawienie rozwiązania do równania.

Oczekiwany rezultat sprawdzenia (czyli 0) otrzymano dopiero po przekształceniu wyrażenia (uproszczeniu) przy pomocy funkcji *simplify*:

wyrażenie2 = **simplify**(*wyrażenie*)

Przy pomocy funkcji *solve* można rozwiązywać również układ równań. Sposób zapisu różni się w zależności od wersji *MATLAB-a*:

Kod	Wynik

%*MATLAB* 2007
clc, clear
syms x y
R = solve('x+2*y-4 = 0', 'x+y - 1 = 0')
x1 = R.x
x2 = R.y
 	
%*MATLAB* 2019
clc, clear
syms x y
R = solve(x+2*y-4, x+y-1)
%albo
%R = solve(x+2*y==4, x== -y+1)
x1 = R.x
x2 = R.y
	R =
	  struct with fields:
	    x: [1×1 sym]
	    y: [1×1 sym]
	
	x1 =
	 -2
	
	x2 =
	 3

W obydwu przypadkach otrzymano rozwiązanie jako strukturę o dwóch polach: *x* i *y*. Wartości pól uzyskuje się stosując zapis kropkowy (kwalifikowany).

W przypadku występowania w równaniu lub układzie równań funkcji trygonometrycznych, rozwiązań może być nieskończenie wiele. *MATLAB* znajduje rozwiązania najbliższe zeru, w przedziale (-π, π).

Przykładowo:

clc,clear syms x y R = solve(y-sin(x)==0,y-cos(2*x)==0) x = R.x*180/pi %przeliczenie na stopnie y = R.y y = double(y) %konwersja do typu double	x = -90.0000 30.0000 150.0000 y = -1 1/2 1/2 y = -1.0000 0.5000 0.5000

Rezultaty obliczeń to współrzędne punktów przecięcia funkcji *sin(x)* i *cos(2x)*, czyli (-90°, -1), (30°, 0.5), (150°, 0.5). Można to pokazać na wykresach funkcji symbolicznych, co zostanie opisane w jednym z kolejnych rozdziałów.

Oprócz funkcji *simplify,* która wykonuje upraszczanie wyrażeń symbolicznych, przydatna niekiedy jest również funkcja *collect* w postaciach:

collect(*wyrażenie_symboliczne*)

collect(*wyrażenie_symboliczne, zmienna*)

która wykonuje grupowanie według potęg zmiennej domyślnej lub wyszczególnionej jako argument, wykonując uprzednio odpowiednie działania, na przykład przemnożenie wielomianów. Poniżej przykład wyjaśniający działanie tej funkcji.

clc,clear syms x y f = (x-4)*(y-4)*(x-2)+y^2 % grupowanie względem x (domyślnie) f1 = collect(f) % grupowanie względem y f2 = collect(f,y)	f = y^2 + (x – 2)*(x – 4)*(y – 4) f1 = (y – 4)*x^2 + (24 – 6*y)*x + y^2 + 8*y – 32 f2 = y^2 + (x – 2)*(x – 4)*y – 4*(x – 2)*(x – 4)

Dla rozwiązywania układu równań liniowych MATLAB dysponuje użyteczną funkcją *equationsToMatrix.* Po zdefiniowaniu równań liniowych w postaci tożsamości o dowolnie "pomieszanych" zmiennych (uwaga na opreator ==), funkcja *equationsToMatrix* przekształca je w odpowiednie macierze: macierz współczynników przy niewiadomych i macierz wyrazów wolnych. Następnie utworzone macierze można wykorzystać do rozwiązania układu równań przy pomocy poznanej w rozdz. 6.3.9 funkcji *linsolve*.

Poniżej przykład obliczeń:

Kod	Wynik
clc, clear syms x y z r1= 3*x - 3*y + z == -1 ; r2 = 4 == -x + y ; r3 = y == 5 - 2*z ; [A,B]=equationsToMatrix([r1,r2,r3]); Q=linsolve(A,B)	A = [3, -3, 1] [1, -1, 0] [0, 1, 2] B = -1 -4 5 Q = -21 -17 11

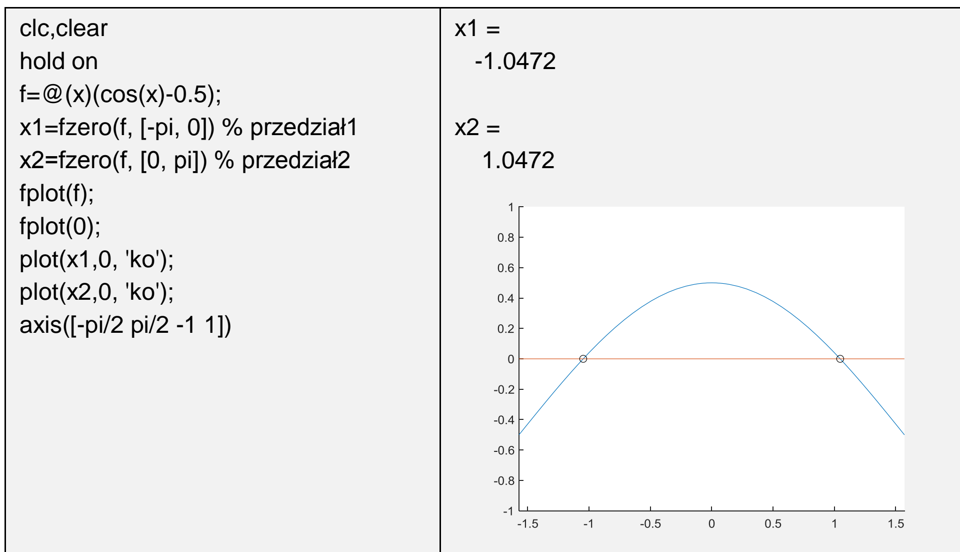
Należy zaznaczyć, że *MATLAB* może nie poradzić sobie z rozwiązaniem bardziej złożonych równań nieliniowych. Wówczas, alternaywnie do funkcji *solve* można wykorzystać funkcję *vpasolve*, która umożliwia również znalezienie miejsc zerowych (punktów przecięcia dwóch krzywych) w wybranym przedziale zmiennej niezależnej.

Poniżej przykład ilustrujący zastosowanie funkcji *vpasolve* dla obliczenia współrzędnych punktów przecięcia dwóch krzywychw założonych przedziałach:

```
clc,clear
syms x
hold on
f=sin(x); g=cos(x);
x=double(vpasolve(f==g, x, [0 2]));
y= double (subs(f));
x1=x
y1=y
syms x
x= double (vpasolve(f==g, x, [3 5]));
y= double (subs(f));
x2=x,
y2=y
fplot(f);
fplot(g);
plot(x1,y1,'ko',x2,y2,'ko');
axis([0 5 -2 2]);
```

```
x1 =
   0.7854
y1 =
   0.7071
x2 =
   3.9270
y2 =
  -0.7071
```


Jeżeli nie posiadamy pakietu Sybbolic Toolbox, do rozwiązywania równań można wykorzystywać funkcję *fzero*, która również daje możliwość określenia przedziału dla szukanego miejsca zerowego (uwaga: wartości funkcji w obydwu granicach przedziału muszą mieć różny znak):



```
clc,clear
hold on
f=@(x)(cos(x)-0.5);
x1=fzero(f, [-pi, 0]) % przedział1
x2=fzero(f, [0, pi]) % przedział2
fplot(f);
fplot(0);
plot(x1,0, 'ko');
plot(x2,0, 'ko');
axis([-pi/2 pi/2 -1 1])
```

```
x1 =
  -1.0472

x2 =
   1.0472
```


12.7. Obliczanie pochodnych

Pakiet *Symbolic Toolbox* umożliwia wyznaczanie **pochodnych** funkcji opisanych symbolicznie. Wykonuje to funkcja pakietu *Symbolic Toolbox* o nazwie *diff* (ang. *differential*), w ogólnej postaci:

diff (*funkcja*, *rząd_pochodnej*, *zmienna*)

Jeżeli pominięty zostanie drugi argument funkcji:

diff (*funkcja*, *zmienna*)

obliczana jest pochodna pierwszego rzędu.

Zmienną można pominąć jeżeli wyrażenie jest funkcją jednej zmiennej Gdy funkcja ma wiele zmiennych, pochodna liczona jest względem domyślnej zmiennej *x*.

diff (*funkcja*, *rząd_pochodnej*)

Przykłady obliczenia pochodnych 2-go i 3-go rzędu:

```
clc, clear
syms  x
f =  sin(x)
%pochodna 1-go rzędu
df = diff(f)
%pochodna 2-go rzędu
df2 = diff(f,2)
%pochodna 3-go rzędu
df3 = diff(f,3)
```

```
f =
  sin(x)
df =
   cos(x)
df2 =
   -sin(x)
df3 =
   -cos(x)
```

Poniżej przykłady obliczeń kilku pochodnych dla innych funkcji elementarnych:

clc, clear syms x g = x^3 dg = diff(g) h = 1/x dh = diff(h) m = log10(x) dm = diff(m)	g = x^3 dg = 3*x^2 h = 1/x dh = -1/x^2 m = log(x)/log(10) dm = 1/(x*log(10))

Jak widać *Symbolic Toolbox* potrafi wyrazić logarytm dziesiętny przy pomocy logarytmu naturalnego.

Można też liczyć pochodne bardziej złożonych funkcji, przykładowo:

clc, clear syms x % iloczyn funkcji f = sqrt(x)*sin(x) df = diff(f) % dla funkcji zagnieżdżanych g = sin(sqrt(x)) dg = diff(g)	f = x^(1/2)*sin(x) df = x^(1/2)*cos(x) + sin(x)/(2*x^(1/2)) g = sin(x^(1/2)) dg = cos(x^(1/2))/(2*x^(1/2))

Dla funkcji wielu zmiennych istnieje możliwość wyznaczenia pochodnych cząstkowych, według każdej zmiennej:

clc, clear syms x y f = sin(x)*cos(y) %pochodna względem x dfx = diff(f, x) %pochodna względem y dfy = diff(f, y)	f = cos(y)*sin(x) dfx = cos(x)*cos(y) dfy = -sin(x)*sin(y)

12.8. Obliczanie całek nieoznaczonych i oznaczonych

Całkę **nieoznaczoną** funkcji oblicza funkcja *int* (ang. *integral*) w poniższych postaciach:

int(*funkcja*, *zmienna*) - całka względem zmiennej *zmienna*

int(*funkcja*) - jeżeli funkcja jednej zmiennej lub całka względem *x*

Przykład:

clc, clear syms x f = sin(x)^2 c = int(f)	f = sin(x)^2 c = x/2 – sin(2*x)/4

Rezultaty nie uwzględniają stałych całkowania.

Całki oznaczone liczone są również przy pomocy funkcji *int*, stosując zapisy:

int(*funkcja*, *zmienna*, *granica_dolna*, *granica_górna*)

to całka oznaczona, względem zmiennej *zmienna*, natomiast:

int(*funkcja*, *granica_dolna*, *granica_górna*)

całka, jeżeli funkcja jest jednej zmiennej lub całka względem *x*.

Poniżej przykład:

clc, clear syms x f = sin(x) c = int(f, 0, pi)	f = sin(x) c = 2

12.9. Granice ciągów i funkcji

Dla obliczania **granic** ciągów i funkcji z pomocą przychodzi funkcja *limit* w poniższych wersjach:

limit (*funkcja*)	- obliczenie granicy w punkcie 0, funkcja jednej zmiennej
limit (*funkcja*, *zmienna*)	- obliczenie granicy przy zmiennej dążącej do 0
limit (*funkcja*, *zmienna*,**inf**)	- obliczenie granicy przy zmiennej dążącej do ∞
limit (*funkcja*, *zmienna*, *p*,**'right'**)	- obliczenie granicy prawostronnej w punkcie *p*
limit (*funkcja*, *zmienna*,*p*,**'left** ')	- obliczenie granicy lewostronnej w punkcie *p*

Poniżej przykład obliczenia kilku granic funkcji (ciągów):

clc, clear syms n x f = 2*n-sqrt(4*n^2-n) granica = limit(f, n, inf) gran_l = limit(tan(x), x, pi/2, 'left') gran_p = limit(tan(x), x, pi/2, 'right')	f = 2*n – (4*n^2 – n)^(1/2) granica = 1/4 gran_l = Inf gran_p = -Inf

Funkcja *tan(x)* jest nieciągła w dla kąta *π/2*, granice lewo- i prawostronna się różnią.

12.10. Wykresy funkcji symbolicznych

Wykresy funkcji symbolicznych wykonuje funkcja *ezplot*:

ezplot (*funkcja*)	- wykres dla zmiennej niezależnej w przedziale (-2π, 2π)

ezplot(*funkcja*, [*xmin*, *xmax*]) - wykres dla zmiennej niezależnej w przedziale (*xmin*, *xmax*)

ezplot(*funkcja*, [*xmin*,*xmax*,*ymin*,*ymax*]) - dodatkowo limity osi y

Można również stosować poznaną uprzednio funkcję *fplot*:

fplot(*funkcja*) - wykres dla zmiennej niezależnej w przedziale (-5, 5),

fplot(*funkcja*, [*xmin*, *xmax*]) - wykres dla zmiennej niezależnej w przedziale (*xmin*, *xmax*),

fplot(*wektor funkcji*, [*xmin*, *xmax*]) - wykres dla zmiennej niezależnej w przedziale (*xmin*, *xmax*),

Jak w przypadku wykresów dla danych liczbowych, można ustalać zakresy osi przy pomocy funkcji *axis*:

axis([*xmin*, *xmax*, *ymin*, *ymax*])

- tworzyć siatkę na wykresie:

grid

- dodawać opisy osi:

xlabel(*'tekst1'*)

ylabel(*'tekst2'*)

- tworzyć wiele krzywych w jednym układzie współrzędnych przez uprzednie dołączenie polecenia:

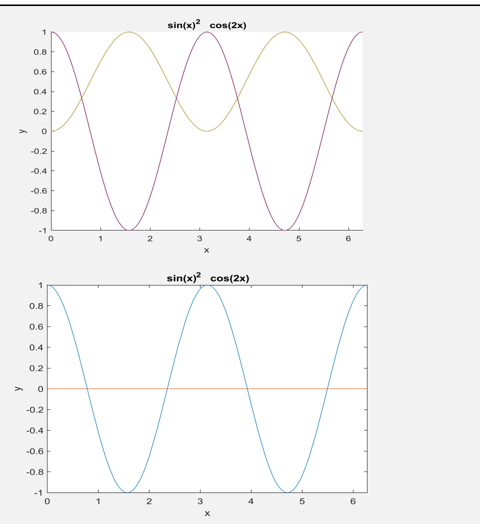
hold on

- a także dokonywać podziału okna *Figure* na wiele układów współrzędnych::

subplot (*m*, *n*, *k*)

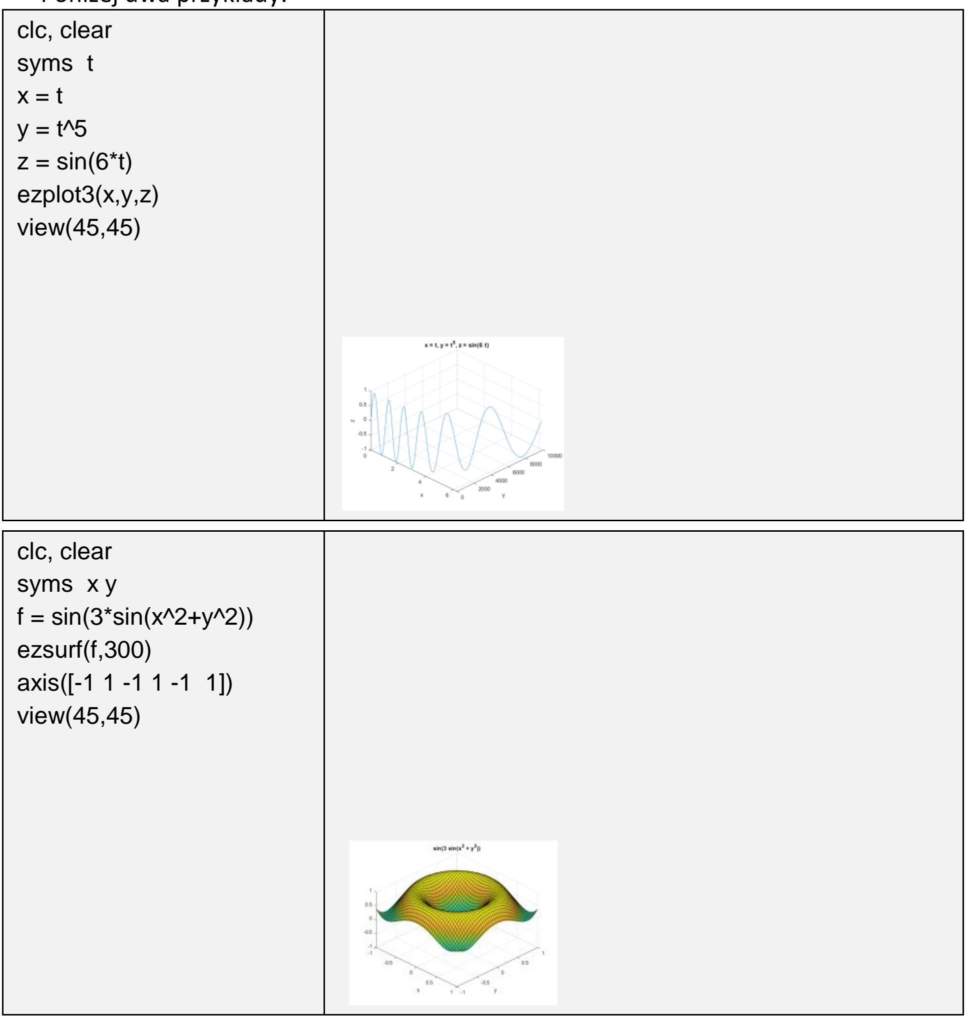
Przykład wykresu funkcji symbolicznych:

```
clc, clear
hold on %będzie wiele krzywych
syms  x
f = sin(x)^2;
g = cos(2*x);
% skorzystamy z ezplot
ezplot(f, [0   2*pi])
% a teraz wypróbujmy fplot
figure(2)
fplot([g 0], [0   2*pi]) %z osią x

axis([0  2*pi  -1    1])
title('sin(x)^2   cos(2x)')
xlabel('x')
ylabel('y')
```



Wykresy krzywych i powierzchni trójwymiarowych wykonuje się korzystając z funkcji: *ezplot3*, *ezmesh* i *ezsurf*, w podobny sposób jak opisane wcześniej funkcje *plot3*, *mesh* i *surf*.

Poniżej dwa przykłady:



```
clc, clear
syms  t
x = t
y = t^5
z = sin(6*t)
ezplot3(x,y,z)
view(45,45)
```

```
clc, clear
syms  x y
f = sin(3*sin(x^2+y^2))
ezsurf(f,300)
axis([-1 1 -1 1 -1  1])
view(45,45)
```

12.11. Równania różniczkowe

Równania różniczkowe zawierają złożony zapis funkcji i jej pochodnych, wymagają znalezienia takiej funkcji, która je spełnia. Rozwiązanie równania różniczkowego wykonuje funkcja *dsolve* w postaci ogólnej:

rozwiązanie = **dsolve**('*równanie*', '*warunki_początkowe*', '*zmienna_niezależna*')

Zasady korzystania z funkcji *dsolve*:

- domyślnie zmienną niezależną jest *t*, wówczas może być pominięty argument '*zmienna_niezależna*',
- '*równanie*', '*warunki_ początkowe*' i '*zmienna_niezależna*' powinny być otoczone apostrofami (' '),

- w równaniu używany jest identyfikator szukanej funkcji i może być użyty identyfikator zmiennej niezależnej,
- pochodną pierwszego rzędu zapisuje się, pisząc przed identyfikatorem funkcji dużą literę **D**, pochodną drugiego rzędu oznaczamy **D2**, itd.,
- dla eliminacji stałych całkowania tworzony jest zapis warunków początkowych określający współrzędne punktu, przez który ma przechodzić wykres funkcji, jej pochodnej itd.- liczba warunków początkowych powinna być równa rzędowi równania różniczkowego.

Sposób rozwiązania równania różniczkowego pierwszego stopnia, opisanego równaniem:

$$\frac{df}{dt} = 1 + f^2$$

przedstawia krótki skrypt:

clc, clear z = dsolve('Dy=1+y^2') %sprawdzenie rozwiązania %przez podstawienie spr = diff(z)-1- z.^2 %uwaga .^ bo to jest wektor!	z = tan(C1 + t) 1i -1i spr = 0 0 0

Rozwiązanie rzeczywiste zawiera stałą całkowania.

Teraz można zmienić domyślny identyfikator zmiennej niezależnej oraz dołączyć warunek początkowy *y(0)=1*:

clc, clear z = dsolve('Dy=1+y^2','y(0)=1', 'x') %sprawdzenie rozwiązania lewa = diff(z) prawa = 1+z^2 %sprawdzenie warunku początkowego x = 0 r0 = subs(z) %podstawienie	z = tan(x + pi/4) lewa = tan(x + pi/4)^2 + 1 prawa = tan(x + pi/4)^2 + 1 x = 0 r0 = 1

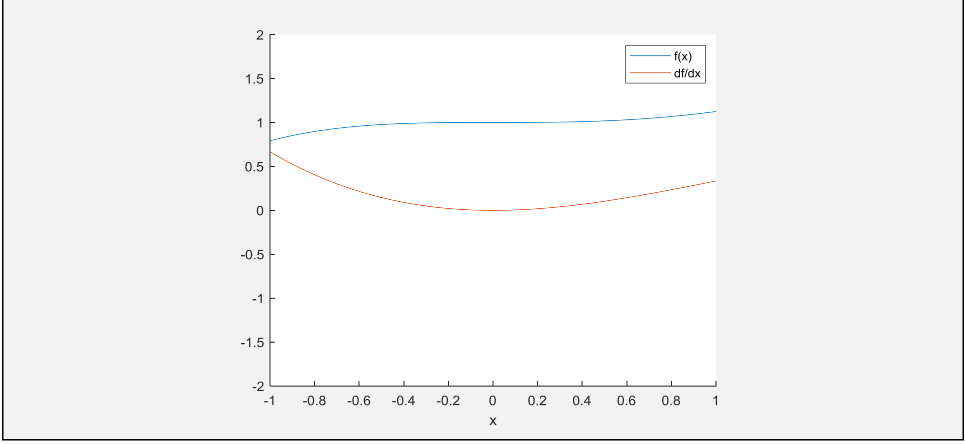
Dodatkowo sprawdzono poprawność rozwiązania, porównując lewą i prawą stronę równania oraz wstawiono do rozwiązania wartość *x = 0* (wykorzystując funkcję *subs*), w celu sprawdzenia poprawności warunku początkowego.

W przypadku równania różniczkowego zwyczajnego drugiego stopnia (występowanie w równaniu różniczkowym pochodnej drugiego rzędu), funkcja *dsolve* wymaga podania dwóch warunków początkowych:

```
clc,clear, close
syms x
hold on
y = dsolve('D2f+Df=sin(x)','f(0)=1',...
                        'Df(0)=0','x')
pretty(y)
ezplot(y)
ezplot(diff(y))
axis([-1 1 -2 2])
legend('f(x)', 'df/dx')
```

```
y =
  2 – (2^(1/2)*cos(x – pi/4))/2 – exp(-x)/2

                 /      pi \
   sqrt(2) cos| x -   --   |
                 \      4  /       exp(-x)
2 -  ------------------------   -  -------
               2                      2
```

oraz wykres funkcji i jej pochodnej:



Zademonstrowano działanie funkcji *pretty*:

pretty(*wyrażenie*)

która modeluje tekstowo wyrażenie symboliczne, przybliżając je do postaci matematycznej. Dodano też legendę do wykresu.

Funkcja *dsolve* prowadzi także do rozwiązania układu równań różniczkowych zwyczajnych kilku zmiennych, z warunkami początkowymi lub bez.

Wykonany będzie przykład rozwiązania dwóch równań różniczkowych, opisanych równaniami:

$$\frac{df}{dt} = 3f(t) + 4g(t)$$

$$\frac{dg}{dt} = -4f(t) + 3g(t)$$

Poszukiwana jest postać funkcji *f(t)* i *g(t)*.

Utworzono *M-plik*:

clc, clear S = dsolve('Df = 3*f+4*g','Dg=-4*f+3*g') F = S.f G = S.g	S = struct with fields: g: [1×1 sym] f: [1×1 sym] F = C1*cos(4*t)*exp(3*t) + C2*sin(4*t)*exp(3*t) G = C2*cos(4*t)*exp(3*t) – C1*sin(4*t)*exp(3*t)

Pojawiła się informacja, że rozwiązania znajdują się w strukturze ***S*** o polach ***f*** i ***g***. Zapis kwalifikowany pozwala na dostęp do pól składowych.

Rozwiązania są zależne od dwóch stałych całkowania *C1* i *C2*. Eliminację tych stałych wykonuje się sposobem jak wyżej, przez wstawienie warunków początkowych:

clc, clear S = dsolve('Df=3*f+4*g','Dg=-4*f+3*g',... 'f(0)=0', 'g(1)=1'); F = S.f G = S.g	F = (sin(4*t)*exp(3*t)*exp(-3))/cos(4) G = (cos(4*t)*exp(3*t)*exp(-3))/cos(4)

Nowsze wersje MATLAB-a umożliwiają alternatywną notację argumentów funkcji *dsolve*. Porównanie obu metod ilustruje przykład:

clc
% 1 sposób
r1='Df+f=1';
w1='f(0)=0';
f=dsolve(r1,w1)
 	
% 2 sposób
syms t g(t)
r2=diff(g)+g==1;
w2=g(0)==0;
g=dsolve(r2,w2)
	f =
	
	1 - exp(-t)
	
	g =
	
	1 - exp(-t)

Problemy rozwiązywania równań różniczkowych są na tyle trudne matematycznie, że z wieloma funkcja *dsolve* sobie nie poradzi. Pojawia się wówczas powiadomienie o tym fakcie w oknie *Command Window*.

Rozwiązywanie numeryczne równań różniczkowych wspomaga też rodzina funkcji *ode* (np. *ode45*). Bardziej zaawansowane działania wymagają pakietu *Optimization Toolbox*.

12.12. Przykładowe zastosowania działań symbolicznych

12.12.1. Badanie przebiegu funkcji

Miejsca zerowe dla przebiegu funkcji symbolicznej znajduje funkcja *solve*. Ekstrema funkcji wyznaczane są przez rozwiązanie równania pochodnej funkcji, ponieważ miejsca zerowe pochodnej to punkty, w których następuje zmiana przedziału monotoniczności funkcji, z malejącej funkcja staje się rosnąca lub odwrotnie.

Zadanie: Zbadać parabolę 3-go stopnia, podaną równaniem:

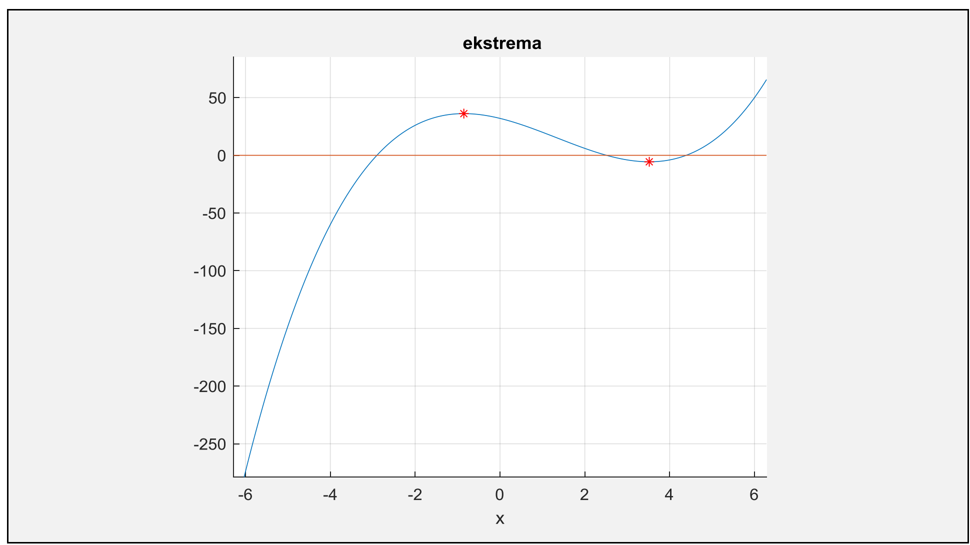
$$x^3 - 4x^2 - 9x + 36 = 0$$

Należy znaleźć jej miejsca zerowe i punkty ekstremum.

Skrypt realizujący to zadanie:

```
clc, clear, close
syms  x
hold on
f = x^3-4*x^2-9*x+32
m_zerowe = double(solve(f))
df = diff(f)
% ekstrema
ekstrema = double(solve(df))
ezplot(f);
ezplot(0*x);  %oś x
% zaznaczenie ekstremów gwiazdką
x = ekstrema(1);        % x liczbowe
y = subs(f);            %podstawienie
plot(x, y, 'r*')        % rysuj gwiazdkę
x = ekstrema(2);        % jak wyżej
y = subs(f);
plot(x, y, 'r*'); grid; title(' ekstrema');
```

```
f =
  x^3 – 4*x^2 – 9*x + 32

m_zerowe =
    2.5122
   -2.9018
    4.3896

df =
   3*x^2 – 8*x – 9

ekstrema =
   -0.8525
    3.5191
```



Funkcja ma trzy miejsca zerowe i dwa punkty ekstremum. Wykorzystano tu funkcję konwersji ułamków symbolicznych do typu *double*. Dorysowano funkcją *plot* czerwone gwiazdki w punktach ekstremów. Można stwierdzić zgodność obliczonych wartości miejsc zerowych i ekstremów z wartościami odczytanymi z wykresu.

Badanie czy w punkcie ekstremum występuje maksimum czy minimum wymaga zastosowania zasady matematycznej, że pochodna drugiego rzędu ma wartość dodatnią w punkcie minimum, a ujemną w punkcie maksimum.

Po modyfikacji skrypt przedstawia realizację badania rodzaju ekstremów:

```
clc, clear, close
syms  x
f = x^3-4*x^2-9*x+32
df = diff(f) %pochodna 1 rzędu
d2f = diff(f,2) %pochodna 2 rzędu
ekstrema = double(solve(df))
% do x podstawiamy ekstrema
x = ekstrema;
%ekstrema podstawiamy do drugiej pochodnej
b = subs(d2f);
for  k = 1:length(ekstrema)
 if   b(k)>0
   fprintf('Dla x=%f jest minimum \n',ekstrema(k))
 else
   fprintf('Dla x=%f jest maksimum \n',ekstrema(k))
  end
end
```

```
f =
  x^3 – 4*x^2 – 9*x + 32
df =
  3*x^2 – 8*x – 9
ekstrema =
   -0.8525
    3.5191
d2f =
  6*x – 8
Dla x=-0.852480 jest maksimum
Dla x=3.519146 jest minimum
```

Przy pomocy poznanej funkcji *subs* podstawiono punkty ekstremów do obliczonej pochodnej drugiego rzędu. Liczbę elementów w wektorze ekstremów wyznaczono funk-

cją *length* dla wektora ekstremów. W pętli sprawdzano znak kolejnych wartości pochodnej drugiego rzędu w punktach ekstremów.

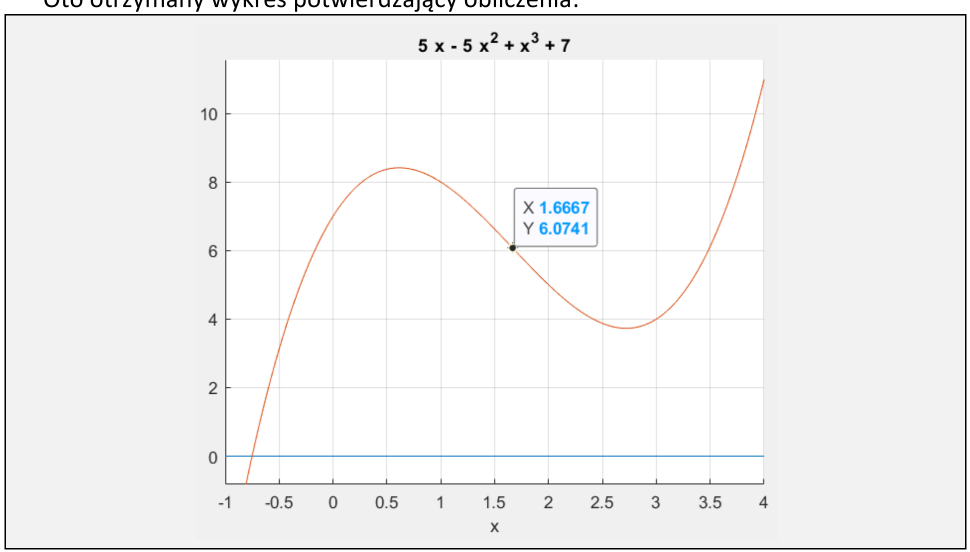
Skrypt może być użyteczny dla badania ekstremów funkcji wielomianowych o wyższych potęgach.

Można również zbadać punkty przegięcia funkcji, rozwiązując równanie drugiej pochodnej funkcji:

```
clc, clear, close
syms x
hold on
f = x^3-5*x^2+5*x+7

%wykres
ezplot(0*x, [-1,4])  %oś x
ezplot(f, [-1,4]);grid;

%pochodna 2 stopnia
d2f = diff(f,2);
ppx = double(solve(d2f));      %punkt przegięcia

%podstawienie x do funkcji
x = ppx; %x teraz jest typu double
ppy = double(subs(f));
fprintf('Punkt przegięcia:%0.3f,  %0.3f\n', ppx,ppy)
```

```
f =
  x^3 – 5*x^2 + 5*x + 7
Punkt przegięcia:1.667,  6.074
```

Oto otrzymany wykres potwierdzający obliczenia:



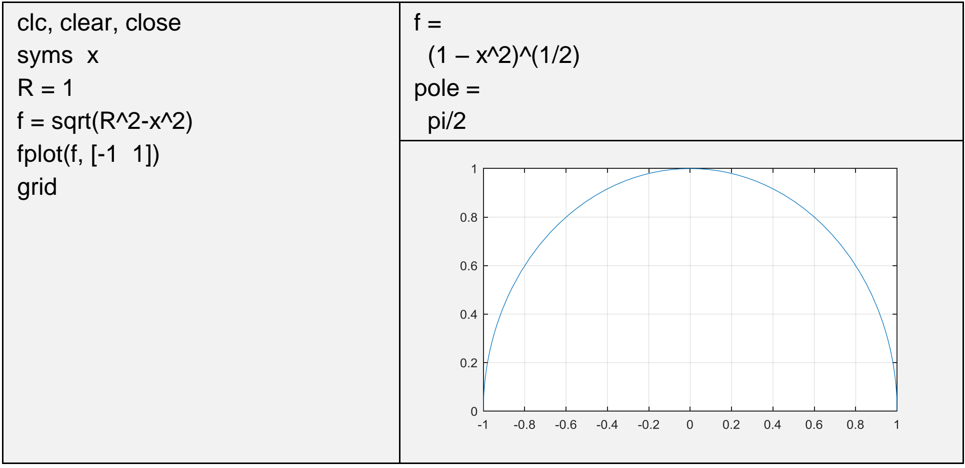
12.12.2. Obliczanie pól powierzchni

W celu wyznaczania pól powierzchni wykorzystuje się całkę oznaczoną, która określa wartość powierzchni między krzywą opisaną daną funkcją a osią *x,* w granicach całkowania.

Zostanie to sprawdzone dla półokręgu, który jest opisany równaniem:

$$y = \sqrt{R^2 - x^2}$$

Jeżeli zostanie założona długość promienia *R*=1:



```
clc, clear, close
syms  x
R = 1
f = sqrt(R^2-x^2)
fplot(f, [-1  1])
grid
```

```
f =
  (1 – x^2)^(1/2)
pole =
  pi/2
```


Otrzymano wynik zgodny z oczekiwaniem.

Rozwiązanie zadania, polegającego na obliczeniu pola powierzchni między dwiema krzywymi opisanymi funkcjami f_1 i f_2 , wymaga kolejnych kroków działania:

- znalezienia granic całkowania, czyli punktów przecięcia obydwu krzywych, a więc rozwiązania równania:

  $$f_1 = f_2$$

  czyli

  $$f_1 - f_2 = 0$$

  Wykonuje się tę operację wykorzystując funkcję *solve*,

- obliczenia całki oznaczonej z różnicy obydwu funkcji (lub różnicy całek dla każdej funkcji z osobna).

  Wykorzystuje się w tym celu funkcję *int* dla całek oznaczonych, granicami całkowania będą wyznaczone wcześniej współrzędne ***x*** punktów przecięcia.

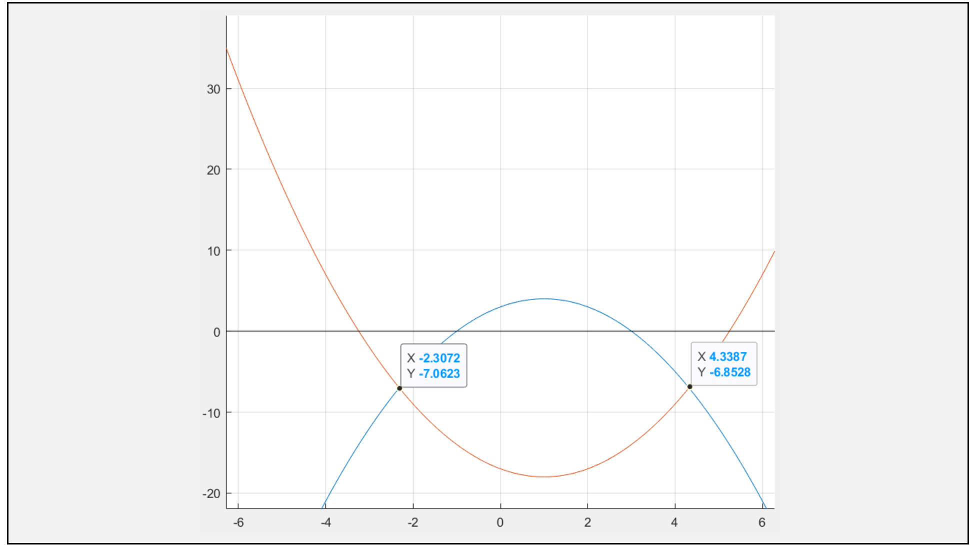
Czasem krzywe przecinają się w trzech lub więcej punktach, należy dokonać właściwego wyboru granic całkowania.

Poniżej przykład skryptu ilustrującego obliczenie pola powierzchni pomiędzy dwiema parabolami:

Skrypt	Wynik

clc, clear , close
syms x
hold on
% dwie funkcje
f1 = -x^2+2*x+3
f2 = x^2-2*x-17
%wykresy
fplot(f1);
fplot(f2);
grid
wykres = fplot(0*x); %przypisanie wykresu do zmiennej
%kolor czarny osi x
wykres.Color = [0 0 0];
%punkty przecięcia krzywych
p_przec = double(solve(f1-f2))
% pole powierzchni: całka oznaczona z f1-f2
pole1 = double(int(f1-f2, p_przec(1), p_przec(2)))
% sprawdzamy też różnicę całek
pole2 = int(f1, p_przec(1),p_przec(2)) ...
- int(f2, p_przec(1),p_przec(2));
%liczba dziesiętna
pole2 = double(pole2)
	f1 =
	- x^2 + 2*x + 3
	
	f2 =
	x^2 – 2*x – 17
	
	p_przec =
	-2.3166
	4.3166
	
	pole1 =
	97.2877
	
	pole2 =
	97.2877

Wykres wykonany przez skrypt:



Można też zastosować wyrażenie:

```
abs(f1-f2)
```

aby uniezależnić obliczenia od wzajemnego położenia krzywych i zawsze otrzymywać dodatnie wartości pola powierzchni.

Dodatkowo pokazano tu sposób ustalenia koloru krzywej. Obiektowi *wykres*, reprezentującemu funkcję *fplot*, nadaje się właściwość *Color* Kolor ma wszystkie składowe *RGB* równe 0 (czarny).

12.12.3. Obliczanie długości krzywych

Matematyka określa sposób obliczenia długości krzywej płaskiej według wzoru:

$$d = \int_{x1}^{x2} \sqrt{1 + \left(\frac{df}{dx}\right)^2}\, dx$$

Wykonany zostanie prosty przykład obliczenia długości paraboli o równaniu:

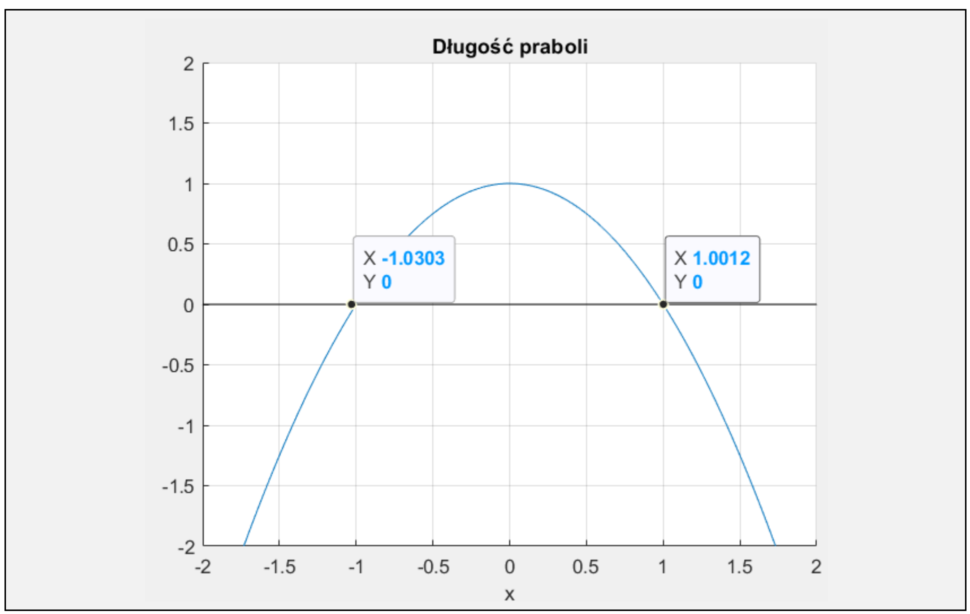
$$f(x) = -x^2 + 1$$

w granicach określonych miejscami zerowymi (współrzędnymi punktów przecięcia z osią *x*).

Oto kod oraz wykres ilustrujący zadanie:

```
clc,clear
syms x
hold on
% funkcja
f = -x^2+1

%miejsca zerowe
mz = solve(f)

%wykres
ezplot(f)
%oś x
ezplot(0*x);
title('Długość praboli')
axis([-2  2 -2  2])

%całka oznaczona
d = int(sqrt(1+diff(f)^2), mz(1), mz(2));
disp('całkowita długość:')
dl = double(d)
```

```
f =
  1 – x^2
mz =
  -1
   1
całkowita długość:
dl =
    2.9579
```



12.12.4. Obliczenia objętości

Zadanie: Na prostokątnym dachu o bokach długości ***a*** i ***b*** jest zaspa śniegowa wysokości ***h***. Należy obliczyć jej objętość.

Kształt zaspy w przybliżony sposób opisuje równanie paraboloidy:

$$z = f(x, y) = \frac{h}{a^2 b^2}(a^2 - 4x^2)(b^2 - 4y^2)$$

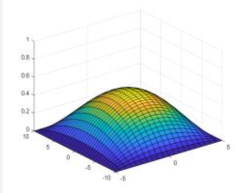
Objętość oblicza całka oznaczona podwójna:

$$\mathrm{v} = \int_{-\frac{a}{2}}^{\frac{a}{2}}\int_{-\frac{b}{2}}^{\frac{b}{2}} f(x, y)\, dx\, dy$$

Poniżej skrypt, w którym wykonano wykres powierzchni, a następnie, wykorzystując działania symboliczne, przy pomocy podwójnej całki oznaczonej, obliczono objętość zaspy.

```
clc, clear
syms y x h a b
zsym =  h*(a^2-4*x^2)*(b^2-4*y^2)/b^2/a^2;

%całkowanie
p = int(int(zsym,x,-a/2,a/2),y,-b/2,b/2)
%dane
a = 10; b = 20;  h = 0.5;

%podstawienie danych
v = double(subs(p))

%wykres
[x, y] = meshgrid(-a/2:0.5:a/2, -b/2:0.5:b/2);
z = h*(a^2-4*x.^2).*(b^2-4*y.^2)/b^2/a^2;
surf(x,y,z)     %wykres 3D
axis([-5 5 -10 10 0 1])
```

```
p =
 (4*a*b*h)/9
v =
  44.4444
```



13. Zastosowania *MATLAB-a* w statystyce

13.1. Zestawienie użytecznych funkcji

Zestaw podstawowych funkcji *MATLAB-*, przydatnych w analizach statystycznych przedstawia tabela 13.1.

Tabela 13.1. Funkcje przydatne w statystyce

sum(*A*)	suma elementów liczbowych wektora
length(*A*)	liczba elementów wektora
sort(*A*)	sortowanie elementów wektora w porządku rosnącym
sort(*A*,'descend')	sortowanie elementów wektora w porządku malejącym
mean(*A*)	średnia arytmetyczna
median(*A*)	mediana
std(*A*) **std**(*A*,0)	odchylenie standardowe dla próby
std(*A*,1)	odchylenie standardowe dla populacji
var(*A*) **var**(*A*,0)	wariancja – dla próby
var(*A*,1)	wariancja – dla populacji
cov(*A*)	kowariancja
corrcoef(*A*) **corrcoef**(*A*,*B*)	współczynnik korelacji
histogram(*A*)	rysowanie histogramu
rand	generator losowy liczb z przedziału (0,1) – rozkład równomierny
randn	generator losowy liczb z przedziału (0,1) – rozkład normalny

Populacja to wszystkie elementy zbioru, **próba** to podzbiór populacji, przybliżający populację.

W dalszej części zamieszczono kilka przykładów obliczeniowych.

13.2. Suma i średnia arytmetyczna

Poniżej skrypt realizujący wyznaczenie **sumy** i **średniej arytmetycznej** elementów wektora oraz porównanie wyników z własnym algorytmem:

```
clc, clear
W = [2  4  1 -3  5  8]
suma = sum(W)
srednia = mean(W)
%obliczenia własne
%ile elementów ma zbiór
N = length(W);
s = 0; %wyzerowanie sumy
for k = 1:N
   s = s+W(k);
end
fprintf('Nasza suma = %d\n',s);
fprintf('Nasza średnia = %0.3f\n',s/N);
```

```
W =
   2    4    1   -3    5    8

suma=
   17
srednia=
    2.833

Nasza suma=17
Nasza średnia=2.833
```

13.3. Sortowanie zbioru

Sortowanie elementów wektora przy pomocy wbudowanej funkcji *sort* w porządku rosnącym i malejącym:

```
clc, clear
M = [2  4  1  2  0  -9]
Mrosn = sort(M)
Mmal = sort(M,'descend')
```

```
M =
   2    4    1    2    0   -9
Mrosn =
  -9    0    1    2    2    4
Mmal =
   4    2    2    1    0   -9
```

Istnieje wiele algorytmów sortowania, np. sortowanie bąbelkowe, metody klasy "dziel i zwyciężaj" (*quicksort*, *mergesort*) i inne. Efektywność i szybkość sortowania jest szczególnie istotna w dużych systemach baz danych.

Poniżej zostanie utworzony skrypt, implementujący **sortowanie bąbelkowe**. Jego algorytm polega na kolejnym porównywaniu sąsiadujących elementów i zamiany wartości miejscami, jeżeli ich kolejność jest nieprawidłowa dla wymaganego porządku sortowania. Wykonuje się przebieg *N*-1 porównań (gdzie *N* jest liczebnością zbioru) – pierwszego elementu z drugim, drugiego z trzecim itd., i ewentualnych zamian miejscami przy niewłaściwej kolejności. Po pełnym przebiegu największy (lub najmniejszy przy sortowaniu w porządku malejącym) element znajdzie się na końcu zbioru. Przebieg jest ponawiamy, tym razem dla *N*-2 porównań (porównanie przedostatniego z ostatnim elementem już nie jest potrzebne), z każdym kolejnym przebiegiem liczba porównań jest zmniejszana o 1. Po *N*-1 przebiegach zbiór zostanie posortowany.

Oto skrypt dla ilustracji sortowania metodą bąbelkową dla przykładowego wektora:

```
clear, clc
disp('Wektor wyjściowy')
M = [ 1  9  12  5   7  2  0]
N = length(M)
for  m = 1:N-1            % N-1 przebiegów
  for  k=1:N-m            % N-m porównań
     if   M(k)>M(k+1)
        pom = M(k);    % 3 kroki zamiany
        M(k) = M(k+1);
        M(k+1) = pom;
     end;
   end;
end
disp('Porządek rosnący')
disp(M)
```

```
M =
    1    9   12    5    7    2    0

N =
    7

Porządek rosnący
    0    1    2    5    7    9   12
```

Wykorzystano tu dwie zagnieżdżane pętle, zewnętrzna wykonuje *N*-1 przebiegów, wewnętrzna kolejne porównania kolejnych par elementów, porównań jest *N–m*, gdzie ***m*** jest numerem przebiegu. Porównanie *k*-tego elementu z następnym (*k*+1) wykonuje instrukcja warunkowa.

Zamiana zawartości elementów (przy niewłaściwej ich kolejności) dokonuje się z wykorzystaniem zmiennej pomocniczej w trzech kolejnych krokach:

- przypisanie wartości *k*-tego elementu do zmiennej pomocniczej *pom*,
- przypisanie wartości elementu *k*+1 do elementu *k*-tego,
- przypisanie wartości zmiennej pomocniczej *pom* do elementu *k*+1.

13.4. Mediana

Mediana to wartość środkowa zbioru. Wartość mediany ilustruje, że połowa z ogólnej liczby elementów zbioru ma wartość poniżej wartości mediany, a druga połowa ma wartość powyżej wartości mediany.

Algorytm obliczenia mediany polega na posortowaniu zbioru i wyznaczeniu:

- elementu środkowego – w przypadku nieparzystej liczby elementów,
- średniej arytmetycznej dwóch elementów środkowych – w przypadku parzystej liczby elementów.

Badana jest liczba elementów *N* w wektorze reprezentującym zbiór:

- jeżeli liczba jest nieparzysta, to środkowy element wektora ma indeks: $\frac{N+1}{2}$,
- jeżeli liczba jest parzysta, to obliczana jest średnia arytmetyczna dwóch środkowych elementów wektora, o indeksach:

$$\frac{N}{2} \qquad \text{oraz} \qquad \frac{N}{2}+1$$

Realizacja powyższego algorytmu, wraz z porównaniem wyniku z rezultatem wbudowanej funkcji *median*:

clc, clear
M = [1.1 5.2 9.5 6.2 -7.5 -4.0 6.2 2.0];
M = sort(M) %sortujemy rosnąco
N = length(M)
if rem(N,2)==1
  med = M((N+1)/2)
else
  med = (M(N/2)+M(N/2+1)) /2
end
%sprawdzenie wyniku funkcji median
med_s = median(M)
	M =
	1.10 5.20 9.50 6.20 -7.50 -4.00 6.20 2.00
	M =
	  -7.50 -4.00 1.10 2.00 5.20 6.20 6.20 9.50
	N =
	    8
	med =
	   3.6000
	med_s =
	   3.6000

Parzystość lub nieparzystość wyznacza się badaniem reszty z dzielenia liczby elementów wektora przez 2, z użyciem funkcji *rem*.

13.5. Wariancja i odchylenie standardowe

Wariancja i **odchylenie standardowe** określają wielkość zróżnicowania wyników w zbiorze – pokazują czy różnice pomiędzy średnią arytmetyczną a poszczególnymi elementami zbioru są duże czy też niewielkie.

Poniżej zamieszczono wzory matematyczne dla obliczenia wariancji dla populacji i próby:

$$\mathrm{w_{populacji}} = \frac{\sum(\mathrm{x_i} - \mu)^2}{\mathrm{N}}$$

$$\mathrm{w_{próby}} = \frac{\sum(\mathrm{x_i} - \mu)^2}{\mathrm{N} - 1}$$

gdzie μ - średnia arytmetyczna elementów zbioru.

Odchylenie standardowe obliczane jest w obydwu przypadkach według zależności:

$$\sigma_{populacji} = \sqrt{w_{populacji}}$$

$$\sigma_{próby} = \sqrt{w_{próby}}$$

Dla dużych zbiorów częściej liczona jest wariancja i odchylenie standardowe dla próby, które dają przybliżenie (estymację) wartości tych parametrów dla pełnego zbioru.

Można zatem napisać *M-plik* dla obliczenia (według powyższego wzoru) wariancji i odchylenia standardowego próby dla losowego zbioru i sprawdzenia wyniku z rezultatami funkcji standardowych *var* i *std*.

clear,clc
N = 100; %liczebność próby
M = rand(1,N);
 	
%średnia próby
sr_pr = sum(M)/N;
s_pr = 0;
for k = 1:N
    s_pr = s_pr+(M(k)-sr_pr)^2;
end;
war_pr = s_pr/(N-1);
fprintf('Wariancja próby = %f\n',war_pr)
 	
disp('Sprawdzenie z wynikiem funkcji var')
fprintf('var(M) = %f\n',var(M))
if war_pr-var(M) == 0
    disp('Wariancja próby poprawna ')
end
odch_pr = std(M)
odch_pr2 = sqrt(war_pr)
	Wariancja próby = 0.075572
	Sprawdzenie z wynikiem funkcji var
	var(M) = 0.075572
	Wariancja próby poprawna
	
	odch_pr =
	    0.2749
	
	odch_pr2 =
	    0.2749

13.6. Krzywa *Gaussa*

Krzywa Gaussa (tak zwana "krzywa dzwonowa") reprezentuje rozkład normalny, który określa prawdopodobieństwo wylosowania elementów blisko średniej jako najwyższe. Im dalej od średniej w stronę wartości niskich lub wysokich, tym bardziej zmniejsza się prawdopodobieństwo wystąpienia danych.

Krzywą Gaussa można przybliżyć funkcją matematyczną:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

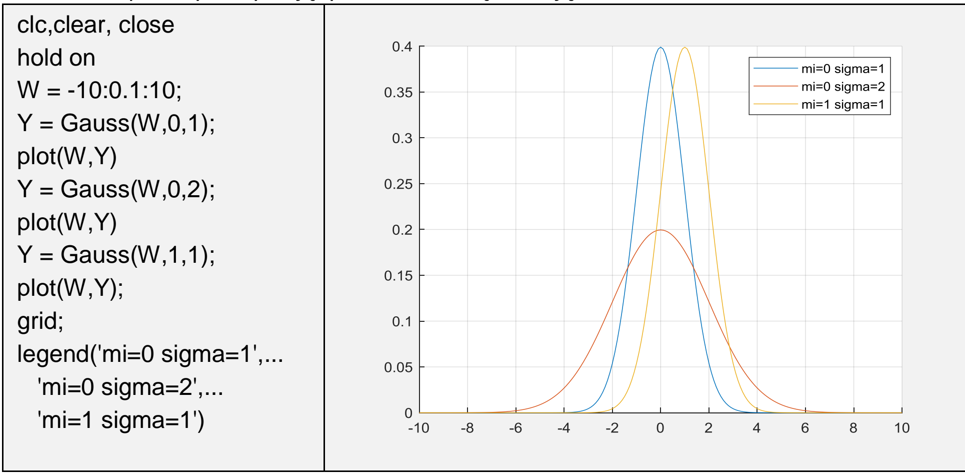
gdzie: μ - średnia arytmetyczna elementów zbioru,

σ - odchylenie standardowe.

Utworzono w tym celu definicję funkcji w pliku *Gauss.m*:

```
function [G] = Gauss(X, mi, sigma)
%Uwaga – X jest wektorem
G = 1/sigma/sqrt(2*pi)*exp(-(X-mi).^2/2/sigma^2)
```

oraz *M-plik* wykorzystujący zdefiniowaną funkcję:

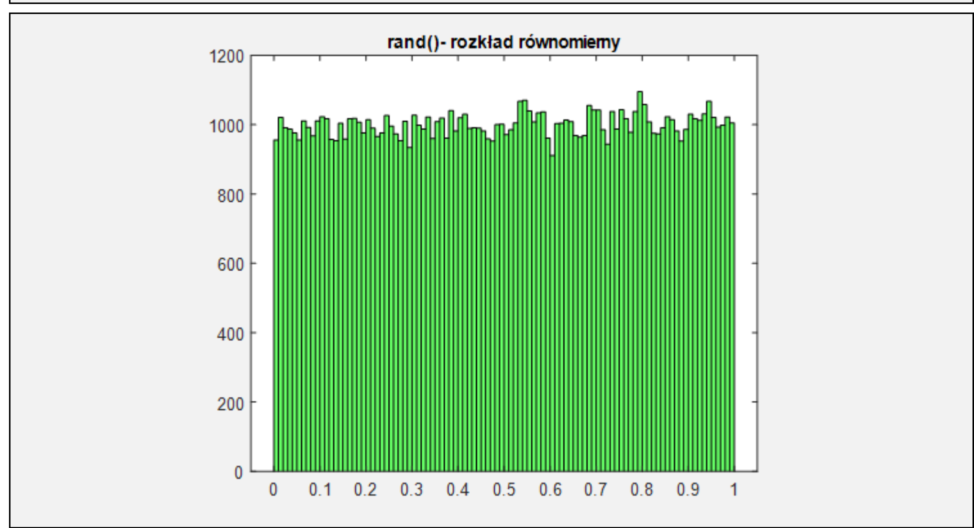


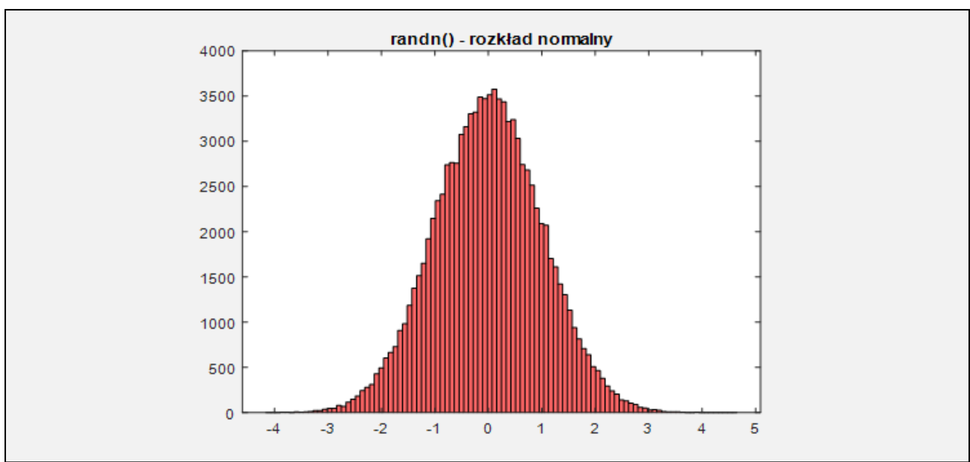
```
clc,clear, close
hold on
W = -10:0.1:10;
Y = Gauss(W,0,1);
plot(W,Y)
Y = Gauss(W,0,2);
plot(W,Y)
Y = Gauss(W,1,1);
plot(W,Y);
grid;
legend('mi=0 sigma=1',...
  'mi=0 sigma=2',...
  'mi=1 sigma=1')
```


Wykresy ilustrują wpływ parametrów μ i σ na kształt krzywej.

Ilustracją różnicy pomiędzy rozkładem równomiernym a normalnym może być poniższy przykład:

```
N = 100000 ;
figure(1)
A = rand(N,1);
histogram(A,100 ,'FaceColor', [0 1 0]);
title('rand()- rozkład równomierny')
figure(2)
B = randn(N,1);
histogram(B,100 ,'FaceColor',[1 0 0]);
title('randn() – rozkład normalny')
```





Równocześnie zaprezentowano działanie funkcji *histogram*, rysującej histogram.

13.7. Operacje na zbiorach

W analizie zbiorów danych przydatne mogą być funkcje operacji na zbiorach. Jeżeli istnieją dwa wektory liczbowe *A* i *B* dowolnej liczebności, to:

union(*A*, *B*)	suma zbiorów czyli wektor elementów wspólnych wektorów *A* i *B* (bez powtórzeń)
setdiff(*A*, *B*)	różnica zbiorów czyli wektor, w którym z wektora *A* usunięto elementy wektora *B*
intersect(*A*, *B*)	iloczyn zbiorów czyli wektor elementów wspólnych dla wektorów *A* i *B*
ismember(*x*, *A*)	wartość logiczna, sprawdza czy element *x* znajduje się w zbiorze *A*

Poniżej przykład ilustrujący działanie funkcji operujących na zbiorach:

```
clc,clear
A = [1  2  3  4  5]
B = [1  2  6]

%suma zbiorów
suma = union(A,B)

%różnica zbiorów
roznica = setdiff(A,B)

%iloczyn zbiorów
iloczyn = intersect(A,B)

%czy element jest w zbiorze
pytanie = ismember(7,A)
```

```
A =
    1    2    3    4    5
B =
    1    2    6

suma =
    1    2    3    4    5    6
roznica =
    3    4    5
iloczyn =
    M    2
pytanie =
  logical
   0
```

14. Wybrane problemy obliczeniowe

14.1. Oscylator harmoniczny z tłumieniem

Równanie różniczkowe opisujące oscylator harmoniczny (np. wahadło lub sprężynę):

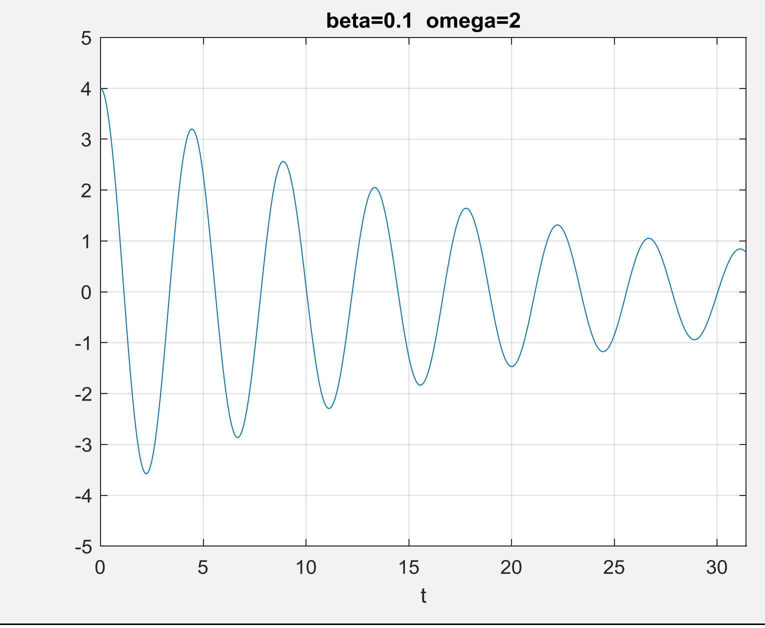
$$\frac{d^2y}{dt^2} + 2\beta\frac{dy}{dt} + {\omega_0}^2 y = 0$$

gdzie: y(t) – funkcja opisująca położenie w czasie,
β – współczynnik tłumienia,
ω_0 - częstość drgań.

Założono początkową amplitudę równą 4 i początkową prędkość (pierwsza pochodna położenia) równą 0, co opisują warunki początkowe. Poniżej realizacja obliczeń w skryp-cie oraz wykres położenia y(t):

```
clc,clear
syms t
%założono beta= 0.05,   om0=sqrt(2);
y = dsolve('D2y+0.1*Dy+2*y=0','y(0)=4','Dy(0)=0');
ezplot(y, [0, 10*pi, -5  5])
title('beta=0.1  omega=2');
```

```
y =
 4*exp(-t/20)*cos((799^(1/2)*t)/20) + (4*799^(1/2)*exp(-t/20)*sin((799^(1/2)*t)/20))/799
```



Jak wynika z wykresu, warunek początkowy wychylenia y(0)=4, jest spełniony

14.2.Analiza rzutu ukośnego

Trajektorię rzutu ukośnego pod kątem α_0 z prędkością początkową v_0 (opór powietrza pominięty) opisują równania różniczkowe, oddzielnie dla składowych x i y :

$$a_x(t) = \frac{d^2x(t)}{dt^2} = 0$$

$$a_y(t) = \frac{d^2y(t)}{dt^2} = -g$$

gdzie g to przyspieszenie ziemskie.

Założono następujące warunki początkowe:

$$v_x(0) = x'(0) = \ v_0 cos\ \alpha_0$$
$$v_y(0) = y'(0) = \ v_0 sin\ \alpha_0$$
$$x(0) = 0$$
$$x(0) = 0$$

Zostanie utworzony *M-plik*, którego kolejne etapy prowadzą do otrzymania przebiegów położenia, drogi, prędkości oraz innych parametrów, takich jak:

- zasięg rzutu,
- maksymalna wysokość,
- czas do uzyskania najwyższej wysokości,
- czas do uderzenia w ziemię,

a także innych wielkości.

Wykonano wykresy przebiegów:

- składowych (poziomej i pionowej) położenia i prędkości w funkcji czasu,
- położenia *y(x),*
- wypadkowej prędkości i drogi w funkcji czasu.

Poszczególne etapy obliczeń opatrzono odpowiednimi wyjaśnieniami w formie komentarzy.

Poniżej kod skryptu:

```
clear,clc
% Rzut ukośny  -  prędkość początkowa v0, kąt wyrzutu alfa0
syms a t0 t v0 v g alfa0
fprintf('Rzut ukośny\n');

%dane
alfa0 = pi/4; v0 = 60; g = 9.81;
% rozwiązanie równań różniczkowych
x = dsolve('D2x = 0','Dx(0) = v0*cos(alfa0)','x(0)=0','t');
y = dsolve('D2y = -g','Dy(0) = v0*sin(alfa0)','y(0)=0','t');

% wstawienie danych do wzorów x(t) i y(t)
xu = subs(x);  yu = subs(y);
```

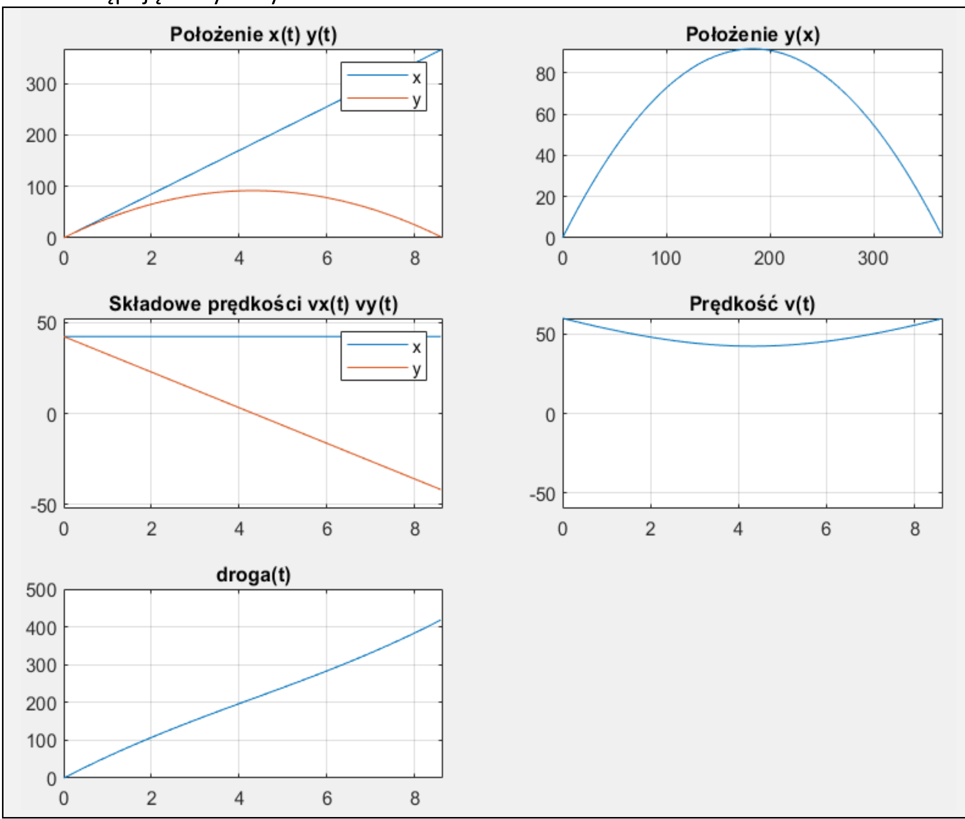
```
% obliczenie funkcji składowych prędkości vx(t) vy(t)
vx = double(diff(xu, t));                     % konwersja do stałej
vy = diff(yu, t);

%obliczenie czasu osiągnięcia ziemi
t_ziemia = double(solve(yu));                 % yu(t)=0   - dwa rozwiązania!
fprintf('Czas osiągnięcia ziemi: %f\n',t_ziemia(2));
%bo oczywiście t_ziemia(1) = 0

% wyznaczenie funkcji s(t)
st = int(sqrt(diff(x, t)^2+diff(y, t)^2),t ,0, t);    % ze wzoru na długość krzywej
s = subs(st);                                 % wstawienie danych alfa0 i v0 do wzoru

% obliczenie drogi całkowitej
t = t_ziemia(2);
sc = double(subs(s));                    % wstawienie czasu t_ziemia(2) do funkcji s(t)
fprintf('Całkowita droga: %f\n',sc);

% Obliczenie zasięgu x rzutu
t = t_ziemia(2);
zasieg = double(subs(xu));
fprintf('Zasięg rzutu: %f\n',zasieg);

% Obliczenie czasu osiągnięcia wysokości ymax
tmax = double(solve(vy));
fprintf('Czas osiągnięcia wysokości max: %f\n',tmax);

% Obliczenie wysokości ymax
t = tmax;
ymax = double(subs(yu));
fprintf ('Y max: %f\n',ymax);

% Obliczenie składowej y prędkości początkowej
t = 0;
vy0 = double(subs(vy));
fprintf('Składowa  y prędkości początkowej:  %f\n',vy0);

%Obliczenie składowej  y prędkości przy uderzeniu w ziemię
t = t_ziemia(2);
vyk = double(subs(vy));
fprintf('Składowa y prędkości przy uderzeniu w ziemię: %f\n',vyk);
```

```
%WYKRESY
t = 0:0.1:t_ziemia(2);                    % oś czasu

% wykres x(t) i y(t)
xp = subs(xu);                            % wstawienie wektora czasu
yp = subs(yu);                            % wstawienie wektora czasu
subplot(3,2,1)
plot(t,xp,t, yp);grid;
axis([0 ,t_ziemia(2), 0 , zasieg])
title('Położenie x(t) y(t)')
legend('x', 'y')

% wykres y(x)
subplot(3,2,2)
plot(xp, yp);grid;
axis([0 ,zasieg, 0, double(ymax)])
title('Położenie y(x)')

%wykres vx(t) i vy(t)
vx = subs(vx);                           % wektor dla vx=const  w punktach czasu
vx = ones(1, length(t))*vx;
vy = subs(vy);                           % wektor vy dla punktów czasu
subplot(3,2,3)
plot(t,vx,t,vy);grid;
axis([0 ,t_ziemia(2), -double(vy0)-10 double(vy0)+10])
title('Składowe prędkości vx(t) vy(t)')
legend('x', 'y')

%wykres v(t)
v = sqrt(vx.^2+vy.^2);                   % sumowanie składowych vx i vy
subplot(3,2,4)
plot(t ,v);grid;
axis([0 ,t_ziemia(2), -v0 v0])
title('Prędkość v(t)')

%wykres s(t)
st = subs(s);                            % wstawienie wektora czasu
subplot(3,2,5)
plot(t,st);grid;
axis([0 ,t_ziemia(2), 0 500])
title('droga(t)')
```

W *Command Window* otrzymano poniższe wyniki liczbowe:

```
Rzut ukośny
Czas osiągnięcia ziemi: 8.649624
Całkowita droga: 421.208651
Zasięg rzutu: 366.972477
Czas osiągnięcia wysokości max: 4.324812
Y max: 91.743119
Składowa  y  prędkości początkowej: 42.426407
Składowa  y  prędkości przy uderzeniu w ziemię: -42.426407
```

oraz następujące wykresy:



14.3. Zagadnienia kinematyki

Przy znajomości odpowiednich wzorów, stosunkowo łatwe są rozwiązania problemów kinematyki metodami arytmetycznymi. Można zrealizować też rozwiązania symboliczne z zastosowaniem wzorów ogólnych:

$$a(t) = \frac{d^2x(t)}{dt^2} = \frac{dv(t)}{dt}$$

$$v(t) = \frac{dx(t)}{dt}$$

lub

$$v = v_0 + \int_{t0}^{t} a(t)dt$$

$$s = s_0 + \int_{t0}^{t} v(t)dt = s_0 + v_0(\mathrm{t} - t_0) + \iint_{t0}^{\mathrm{t}} a(t)\, dt\, dt$$

Poniżej zamieszczono skrypt, który zastosuje powyższe zależności dla ruchu jednostajnie przyspieszonego oraz, zakładając zerowe wartości początkowego czasu i drogi, sporządzi odpowiednie wykresy.

Sprawdzone zostaną wzory znane z kinematyki:

$$v = v_0 + \mathrm{a}(\mathrm{t} - t_0)$$

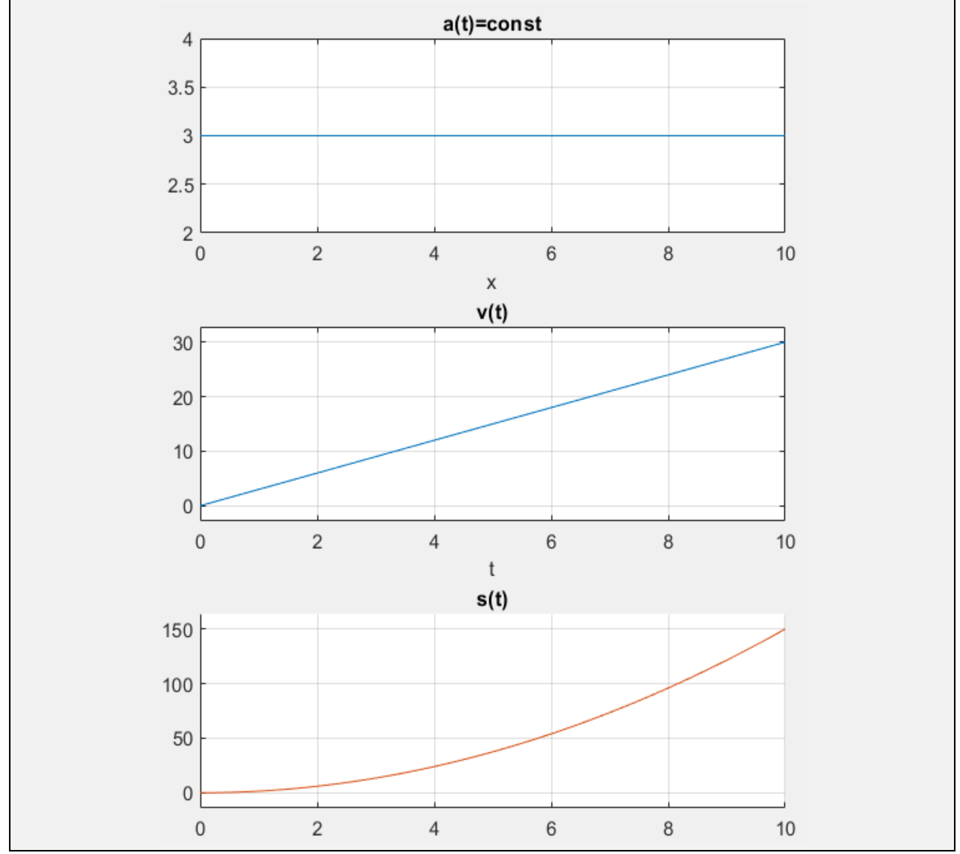
$$s = s_0 + v_0(\mathrm{t} - t_0) + \frac{\mathrm{a}(\mathrm{t} - \mathrm{t}_0)^2}{2}$$

```
clc,clear, close
syms a  t  s0  v0  t0  a
v = v0+int(a,t,t0,t)
s1 = s0+int(v,t,t0,t)                              % całkowanie
s2 = s0 +v0*(t-t0)+int( int(a, t, t0, t), t, t0, t)    % lub alternatywnie
%czy wzory na drogę takie same?
s1_minus_s2 = simplify(s1-s2)                      % uproszczenie
%dane
a = 3; t0 = 0; s0 = 0;v0 = 0;                      % jednostka [m/s2]
v = subs(v)                                        % podstawiamy dane
s1 = subs(s1)
s2 = subs(s2)
%wykresy
subplot(3,1,1)                                     % wykres a(t)
ezplot(0*t+a,[t0 t0+10]);grid;
title('a(t) = const')
subplot(3,1,2)                                     % wykres v(t)
ezplot(v,[t0 t0+10]); ;grid;
title('v(t)')
subplot(3,1,3)
hold on                                            % będą dwie krzywe
ezplot(s1,[t0  t0+10]);                            % s1(t)
ezplot(s2,[t0  t0+10]); ;grid;                     % s2(t)
title('s(t)')
%obliczamy drogę po 10 sek.
t = t0+10;
s10 = subs(s1);
fprintf('Przez 10 s pojazd przejechał %0.2f m\n',s10);
```

Wyniki:

```
v =
   v0 + a*(t – t0)
s1=
   s0 + ((t – t0)*(2*v0 + a*t – a*t0))/2
s2 =
   s0 + v0*(t – t0) + (a*(t – t0)^2)/2
s1_minus_s2 =
   0
v =
  3*t
s1 =
   (3*t^2)/2
s2 =
   (3*t^2)/2
Przez 10 s pojazd przejechał 150.00 m
```

oraz otrzymane wykresy:

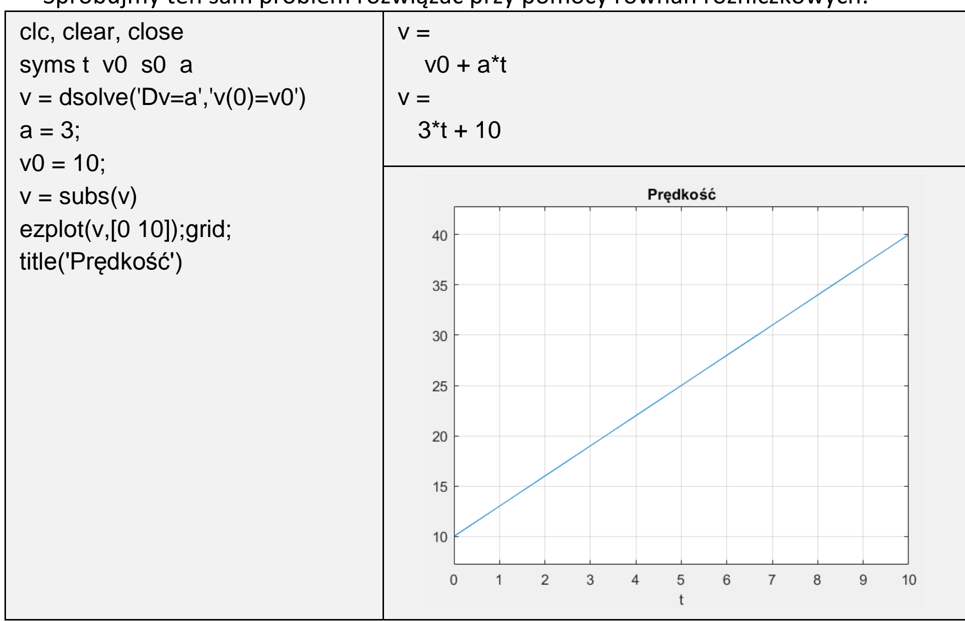


Równania $s_1(t)$ i $s_2(t)$ są identyczne i krzywe te pokrywają się na wykresie.

Jeżeli w powyższym skrypcie zostaną założone niezerowe wartości *s0*, *v0*, otrzymuje się następujące wyniki i wykresy:

... a = 3; t0 = 0; s0 = 10; v0 = 10; ...	v = v0 + a*(t – t0) s = s0 + ((t – t0)*(2*v0 + a*t – a*t0))/2 v = 3*t + 10 s = (t*(3*t + 20))/2 + 10 Po 10s droga = 260.00 m

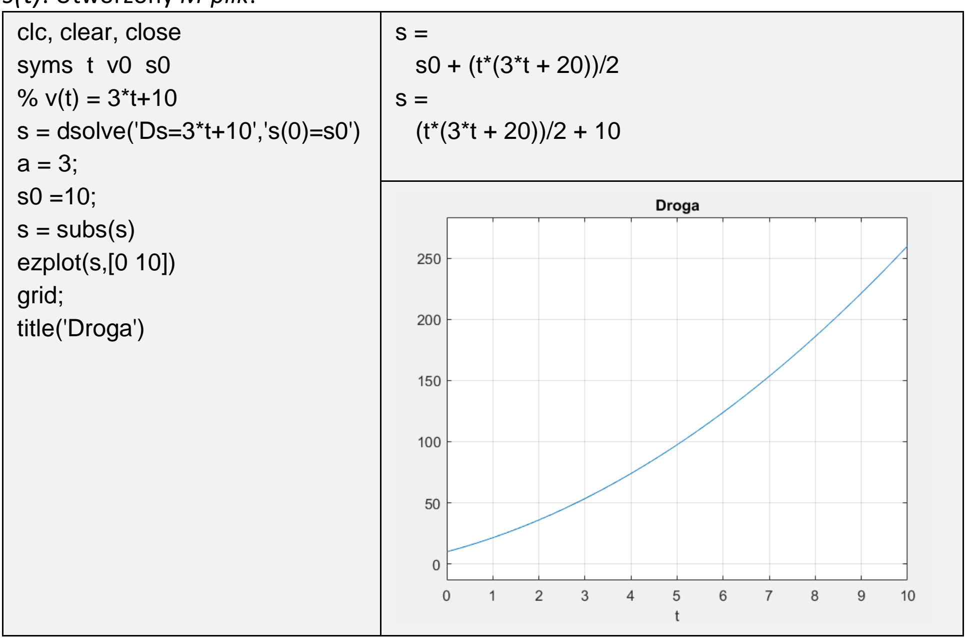
Spróbujmy ten sam problem rozwiązać przy pomocy równań różniczkowych:



```
clc, clear, close
syms t  v0  s0  a
v = dsolve('Dv=a','v(0)=v0')
a = 3;
v0 = 10;
v = subs(v)
ezplot(v,[0 10]);grid;
title('Prędkość')
```

```
v =
   v0 + a*t
v =
  3*t + 10
```


Dysponując wzorem na prędkość *v(t)*, można w podobny sposób wyznaczyć funkcję *s(t)*. Utworzony *M-plik*:



```
clc, clear, close
syms  t  v0  s0
% v(t) = 3*t+10
s = dsolve('Ds=3*t+10','s(0)=s0')
a = 3;
s0 =10;
s = subs(s)
ezplot(s,[0 10])
grid;
title('Droga')
```

```
s =
  s0 + (t*(3*t + 20))/2
s =
  (t*(3*t + 20))/2 + 10
```


Jak widzimy, wyniki otrzymane tą metodą są zgodne z poprzednimi.

Jeżeli ruch jest niejednostajny można zastosować alternatywnie: albo funkcje całkujące *int*, albo rozwiązanie równań różniczkowych funkcją *dsolve*.

Jeżeli przyspieszenie dane jest funkcją, np.:

$$a(t) = Ae^{-Bt}$$

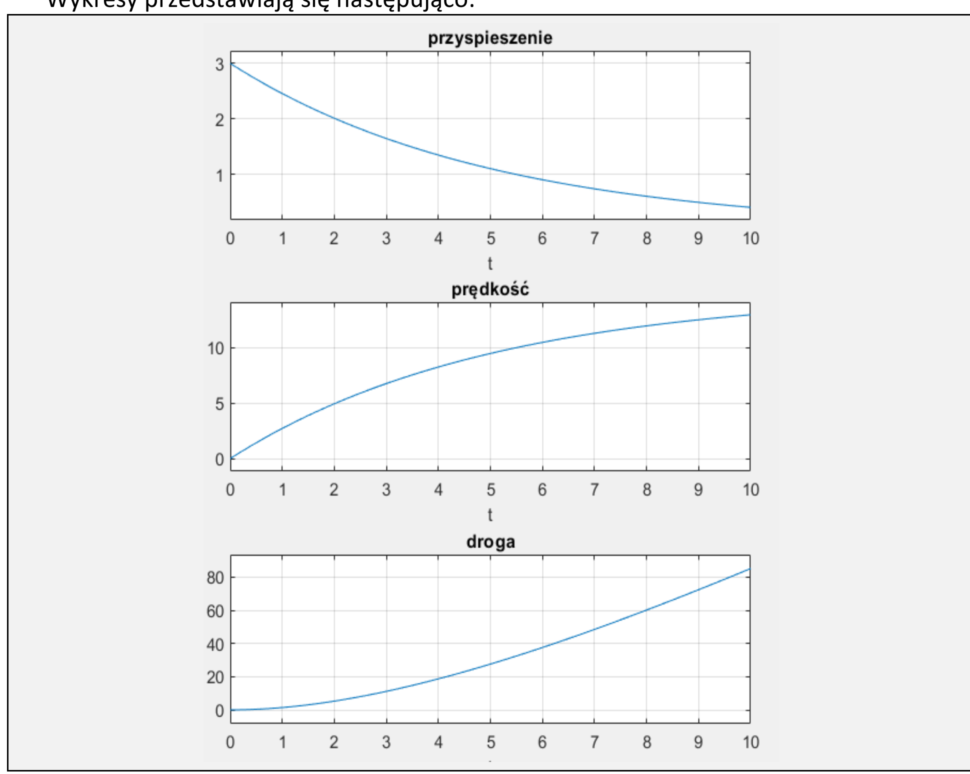
można napisać poniższy skrypt:

```
clc,clear
syms A B v0  s0  t
a = A*exp(-B*t)
v = v0+int(a, t, 0, t)
s = s0+int(v, t, 0, t)
A = 3;  B = 0.2;  v0 = 0;  s0 = 0;
a = subs(a)

%WYKRESY
subplot(3,1,1);ezplot(a,[0 10]); grid;
title('przyspieszenie');
v = subs(v);
s = subs(s);
subplot(3,1,2); ezplot(v,[0 10]);grid;
title('prędkość');
subplot(3,1,3);ezplot(s,[0 10]); grid;
title('droga');

t = 10;
s10 = subs(s); % droga po 10 sekundach
fprintf('Przez 10s pojazd przejedzie %0.2f m\n',s10);
```

który daje następujące rezultaty:

```
a =
   A*exp(-B*t)
v =
   -(A*(exp(-B*t) – 1))/B
s =
   (A*(exp(-B*t) + B*t – 1))/B^2
A =
     3
B =
    0.2000
a =
   3*exp(-t/5)
Przez 10s pojazd przejedzie 85.15 m
```

Wykresy przedstawiają się następująco:



Poniżej przykład zastosowania funkcji *dsolve* w ruchu niejednostajnym, gdy przyspieszenie *a(t)* dane jest funkcją czasu.

Przykładowo, przyspieszenie jest opisane funkcją:

$$a(t) = -0.2t$$

a prędkość początkowa auta: v_0=100 m/s.

Po jakim czasie auto się zatrzyma?

Należy wyznaczyć dodatnie miejsce zerowe dla obliczonej funkcji prędkości ***v(t)***. Funkcję tę otrzymamy rozwiązując równanie różniczkowe.

Poniżej rozwiązanie:

```
clc, clear,close
syms t
hold on
% równanie różniczkowe
v = dsolve('Dv=-0.2*t','v(0)=100')

% szukamy miejsca zerowe dla v(t)
tk = double(solve(v));

%drukujemy tk(2) bo tk(1) ujemny
fprintf(' Czas do zatrzymania = %0.4f\n',tk(2))
ezplot(0*t,[0 tk(2)]);  % oś x
ezplot(v, [0 tk(2)]);
grid;
title('v(t)')
```

```
v =
  100 – t^2/10
Czas do zatrzymania= 31.6228
```


Znając funkcję prędkości można obliczyć przebytą drogę, rozwiązując równanie różniczkowe:

```
s = dsolve('Ds=100-t^2/10','s(0)=0')
```

a następnie do otrzymanej funkcji ***s(t)*** podstawić obliczony wcześniej czas jazdy $t_k(2)$.

Podobną metodykę można stosować w zadaniach, w których znamy funkcję ***s(t)***, a celem jest znalezienie funkcji ***v(t) i a(t)***.

Jeżeli przyspieszenie jest dyskretną funkcją czasu (podlega skokowym zmianom wartości), problem rozwiązuje się przedziałami.

Zadanie: Pojazd przez 10 sekund jedzie z przyspieszeniem:

$$a_1 = 4\ m/s^2$$

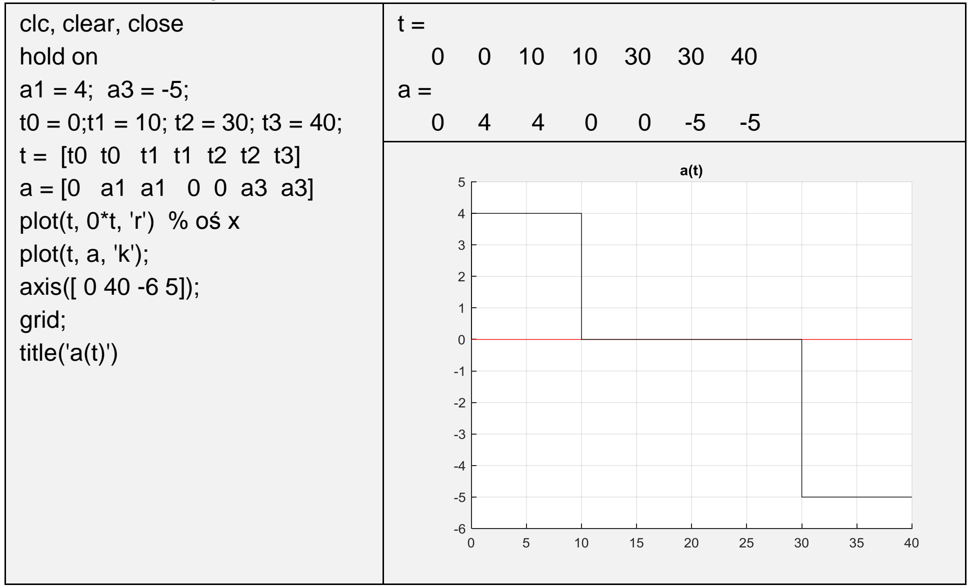
następnie przez 20*s* ze stałą prędkością (a_2=0), a kolejny przedział czasu z przyspieszeniem: $a_3 = -5m/s^2$. Po jakim czasie pojazd się zatrzyma i jaką drogę przebędzie?

Na wstępie zostanie utworzony wykres przyspieszenia *a(t)*. Konieczne jest tu zbudowanie dwóch wektorów, z podwojonymi wartościami czasu dla punktów skokowej zmiany przyspieszenia:

t = [t_0 t_0 t_1 t_1 t_2 t_2 t_3]

a = [0 a_1 a_1 0 0 a_3 a_3]

Końcowy czas t_3 przyjęto z "zapasem". Oto skrypt i wykres *a(t)*:



```
clc, clear, close
hold on
a1 = 4;  a3 = -5;
t0 = 0;t1 = 10; t2 = 30; t3 = 40;
t =  [t0  t0   t1  t1  t2  t2  t3]
a = [0   a1  a1   0  0  a3  a3]
plot(t, 0*t, 'r')  % oś x
plot(t, a, 'k');
axis([ 0 40 -6 5]);
grid;
title('a(t)')
```

```
t =
    0    0   10   10   30   30   40
a =
    0    4    4    0    0   -5   -5
```


Wykres *v(t)* dla ruchu jednostajnie przyspieszonego wyznacza zależność:

$$v(t) = v_0 + a(t - t_0)$$

Jeżeli założymy $t_0 = 0$, to dla kolejnych przedziałów:

$$v_1(t) = a_1 \mathrm{t}$$

$v_2(t) = v_{1t_1} = const$ (wartość w punkcie t_1 i także w punkcie t_2)

$$v_3(t) = v_{1t_1} + a_3(t - t_2)$$

Czas zatrzymania pojazdu będzie można odczytać z wykresu, lub z prostego przekształcenia równania:

$$v_{1t_1} + a_3(t_{zatrz} - t_2) = 0$$

$$t_{zatrz} = -\frac{v_{1t_1}}{a_3} + t_2$$

```
clc, clear
hold on
%dane
a1 = 4;  a2 = -5;
t0 = 0; t1 = 10; t2 = 30; t3 = 40;
T1 = t0: 0.1: t1;
%przedziały
V1 = a1*T1;
T2 = [t1  t2];
V2 = [V1(end)  V1(end)]; % = const
T3 = t2:0.1:t3;
V3 = V1(length(V1))+a2*(T3-t2 );
%scalenie przedziałów T i V
T = [T1 T2  T3];
V = [V1 V2 V3];
plot(T, V);
plot(T, 0*T); % oś x
grid;
title('v(t)')
tk = -V1(end)/a2+t2;
fprintf('Czas zatrz.:%0.2f s\n',tk);
```

```
Czas zatrz.:  38.00 s
```

Nieco trudniej zapisać wyrażenia definiujące wektory *s(t)* w każdym przedziale, trzeba uwzględniać przesunięcia czasowe:

$$s_1(t) = a_1\frac{t^2}{2}$$

$$s_2(t) = s_{1t_1} + v_{1t_1}(t - t_1)$$ (ponieważ a_2=0)

$$s_3(t) = s_{2t_2} + v_{2t_2}(t - t_2) + a_3\frac{(t - t_2)^2}{2}$$

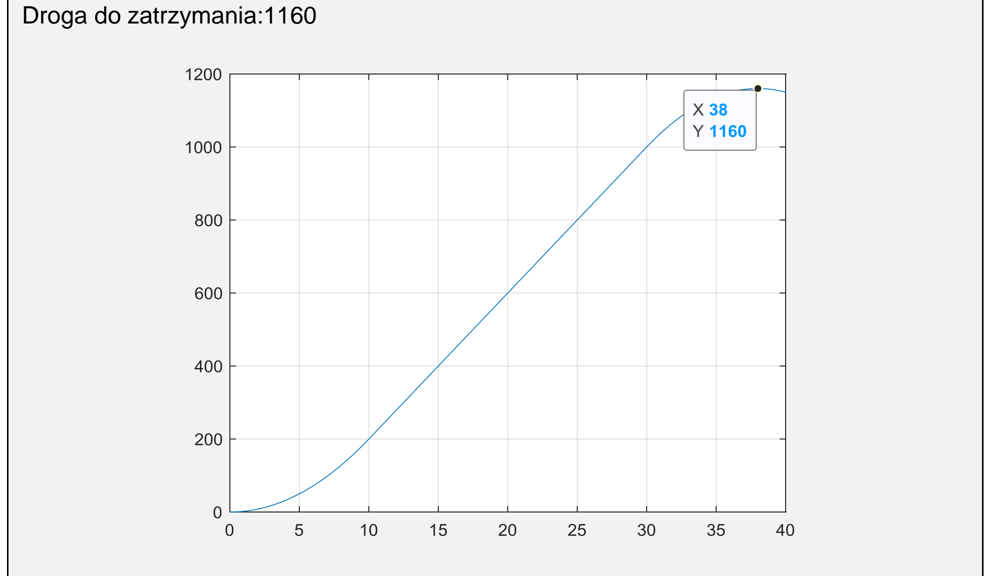
W celu obliczenia drogi przebytej przez pojazd należy wyznaczyć wartość wyrażenia:

$$s_{zatrz} = s_{2t_2} + v_{2t_2}(t_{zatrz} - t_2) + a_3\frac{(\mathrm{t_{zatrz}} - \mathrm{t_2})^2}{2}$$

Kontynuacja poprzedniego skryptu:

```
% ....kontynuacja
figure(2)  %okno dla drugiego  wykresu
S1 = a1*T1.^2/2;
T2 = t1:0.1:t2;
S2 = S1(end)+V1(end)*(T2-t1);
S3 = S2(end)+V2(2)*(T3-t2)+a2*(T3-t2).^2/2;
T = [T1 T2  T3];
S = [S1 S2  S3];
plot(T, S); grid;
%Obliczenie drogi do zatrzymania
Sk = S2(end)+V2(2)*(tk-t2)+a2*(tk-t2)^2/2;
fprintf('Droga do zatrzymania:%0.0f\n', Sk)
```

```
Droga do zatrzymania:1160
```



Można również znaleźć w wektorze *V3* element najbliższy wartości zero i wyszukać w wektorze *S3* element o tym samym indeksie.

W skrypcie uwzględniono, że wartość ostatniego elementu wektorów *V* i *S* danego przedziału jest jednocześnie wartością początkową następnego przedziału. Wartość ostatniego elementu w wektorze można zapisywać jako:

W(end)

choć można też użyć postaci:

W(length(*W*))

czyli element o indeksie równym długości wektora.

Należy pamiętać o operatorze tablicowego podnoszenia elementów wektora do kwadratu (**.**^). Dla operacji mnożenia wektora przez skalar stosuje się operator macierzowy (*), choć można też zastosować operator tablicowy (**.***).

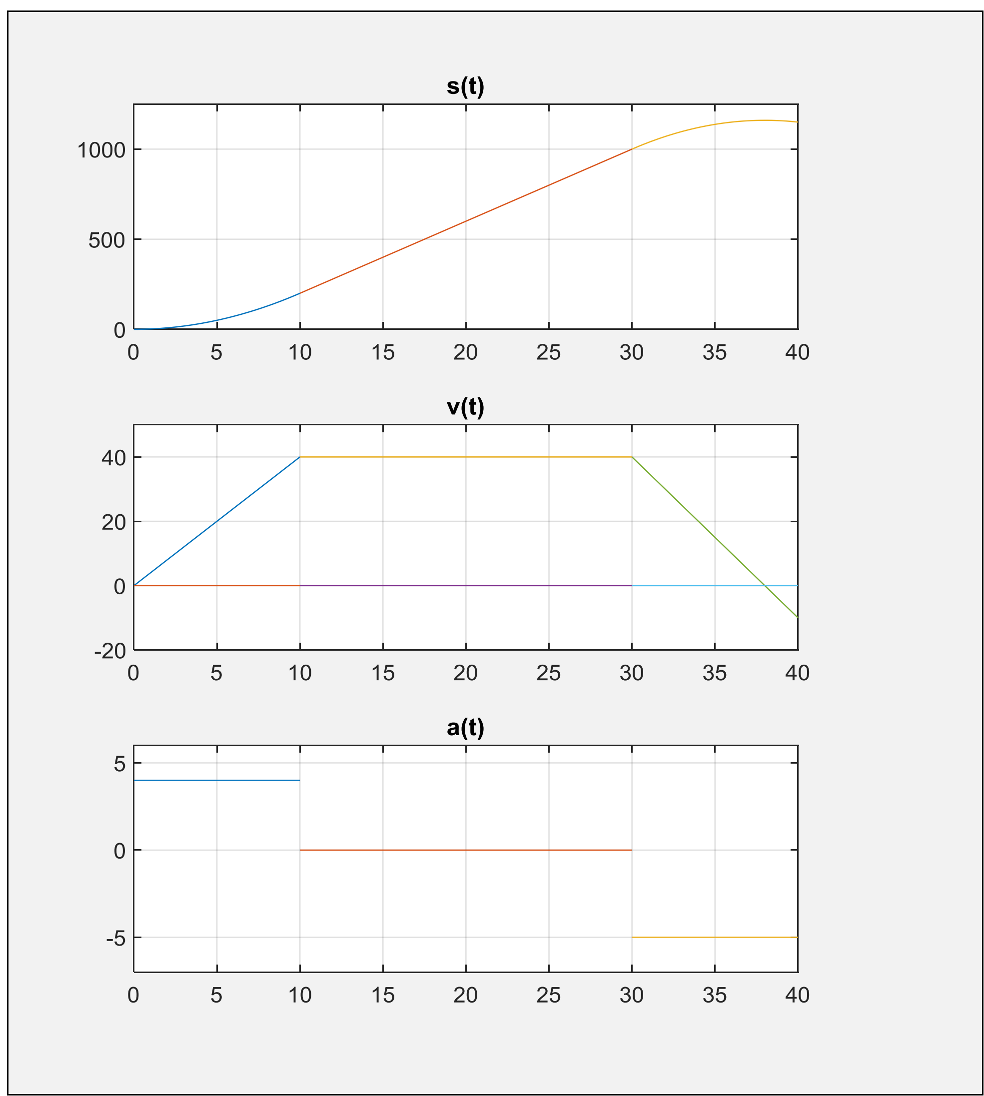
Dla rozwiązania powyższego zadania można wykorzystać również funkcję *dsolve* lub całkowanie symboliczne. Poniżej skrypt ilustrujący te metody:

```
clc, clear, close
syms t tp sp vp a
%dane
T = [0 10 30 40]
A = [4 0 -5]
sp0 = 0;                                    %droga początkowa
vp0 = 0;                                    %prędkość początkowa
N = length(T)-1;                            %liczba przedziałów

%równanie różniczkowe
s = dsolve('D2s=a','s(tp)=sp','Ds(tp)=vp')
%całkowanie
s2 = sp +vp*(t-tp)+int( int(a, t, tp, t), t, tp, t)
sprawdzenie = simplify(s-s2)

%Obliczenie S(t) w przedziałach
for  k = 1:N
   tp = T(k);
   a = A(k);
   if k==1
      sp = sp0;
      vp = vp0;
   end
   S(k) = subs(s);
   V(k) = diff(S(k));
   t = T(k+1);                      % t = czas początkowy kolejnego przedziału
 %obliczenie warunków początkowych dla następnego przedziału
   sp = subs(S(k));
   vp = subs(V(k));
   syms t                           %zmienna t z powrotem typu symbolicznego
end
% wykres s(t) w przedziałach
subplot(3,1,1)
for k = 1:N
   fplot(S(k),[T(k) T(k+1)])
   hold on
   t = T(k+1);                      %końcowy  czas przedziału
   z(k) = subs(S(k));               %podstaw do wzoru
end
axis([T(1) T(end) 0 double(z(end))+100])
title('s(t)'); grid;
```

```
%wykres prędkości
syms t
subplot(3,1,2)
for k = 1:N
   v(k) = diff(S(k));
   fplot([v(k) 0],[T(k)   T(k+1)]);
   hold on
end
axis([T(1)   T(end)   -20   50]);
title('v(t)'); grid;

%wykres przyspieszeń
subplot(3,1,3)
syms a
for  k = 1:N
   a(k) = diff(v(k));
   fplot(a(k),[T(k)  T(k+1)]);
   hold on
end
axis([T(1)   T(end)   min(A)-2   max(A)+2]);
title('a(t)'); grid;

%Obliczenie czasu zmiany kierunku prędkości
t = double(solve(diff(S(3))));
%Obliczenie zasięgu max jazdy
zasieg = subs(S(end));
fprintf('Auto uzyskało zasięg: %d m\n', zasieg);
%Obliczenie końcowego dystansu
t = T(end);
dystans = subs(S(end));
fprintf('Dystans: %d m\n', dystans);
```

Poniżej rezultaty obliczeniowe I graficzne wykonania skryptu:

```
T =
    0    10    30    40
A =
    4     0    -5
s =
  sp - tp*vp + (a*t^2)/2 + (a*tp^2)/2 + t*(vp - a*tp)
s2 =
 sp + vp*(t - tp) + (a*(t - tp)^2)/2
sprawdzenie =
  0
Auto uzyskało zasięg: 1160 m
Dystans: 1150 m
```



Jak widać rezultaty są zgodne z obliczeniami wykonanymi poprzednio metodą algebraiczną. Obliczono zasięg maksymalny jazdy i realnie pokonany dystans, z analizy można wnioskować, że pojazd w ostatnich sekundach zmienił zwrot prędkości, a zatem rozpoczął cofanie.

W skrypcie porównano wzór uzyskany z rozwiązania równania różniczkowego i całkowania symbolicznego, uzyskano zgodny wynik.

Należy zwrócić uwagę, że w iteracji zmienna *t* była naprzemiennie typu symbolicznego (w celu wykorzystywania wzorów ogólnych) oraz typu liczbowego (w celu podstawień do wzorów, aby uzyskać nowe warunki początkowe kolejnego przedziału). Stąd stosowanie deklaracji *syms* dla tej zmiennej wewnątrz pętli.

Dokładniejszą analizę proponowanego rozwiązania oraz dostosowanie powyższego skryptu do innych danych oraz innej liczebności przedziałów, pozostawia się czytelnikowi.

14.4. System ze sprzężeniem zwrotnym

System ze sprzężeniem zwrotnym to taki system, w którym zachodzi oddziaływanie danych stanu końcowego (wyjściowego) na jego sygnały wejściowe.

Zadanie: Pojazd rozpoczyna jazdę od prędkości v=0 ze stałym przyspieszeniem **a**.

W pojeździe działa urządzenie zwane *tempomatem*, o następującej zasadzie:

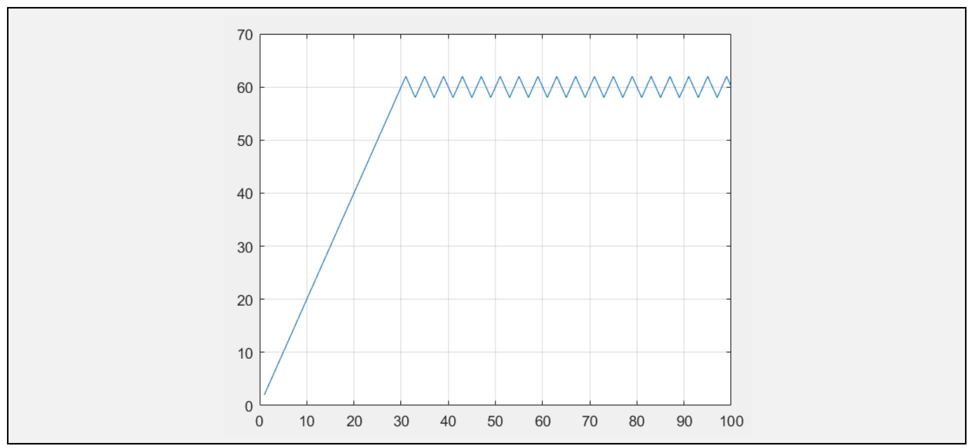
- po osiągnięciu prędkości $\mathbf{v}_{max}$ przyspieszenie pojazdu wynosi **–a**,
- po osiągnięciu prędkości $\mathbf{v}_{\mathbf{min}}$ przyspieszenie pojazdu wynosi **a**.

Realizacja w *MATLAB-ie* z pakietem *Symbolic Toolbox*, z wykorzystaniem funkcji całkującej *int*:

```
clc,clear,close
syms a t t0 v0
v = v0+int(a,t,t0,t)  %całka
ax = 2; v0 = 0;  t0 = 0;  vmin = 59;   vmax = 61;    %dane
a = ax;                          %początkowe przyspieszenie
for t = 1:100                    %pętla po jednostkach czasu
    % podstawienie a,vo,t0, t do v
    V(t) = subs(v);
    T(t) = t;                    %element wektora czasu
    if  V(t) > vmax              %sprawdzenie vmin, vmax
        a = -ax;                 %ujemne przyspieszenie
    elseif V(t) < vmin
        a = ax;                  %dodatnie przyspieszenie
   end
   v0 = V(t);                    %nowy warunek początkowy
   t0 = T(t);                    %nowy warunek początkowy
end
plot(T,V);grid                   %wykres
```

Rezultat obliczeniowy i graficzny:

```
v =
  v0 + a*(t –t0)
```



Do wyznaczenia funkcji *v(t)*, zamiast funkcji całkującej *int*, można skorzystać z funkcji *dsolve*:

```
%równanie różniczkowe
v = dsolve('Dv=a','v(t0)=v0')
```

co da identyczny rezultat.

14.5. Metoda *Monte Carlo*

Metoda *Monte Carlo* jest stosowana do modelowania matematycznego procesów trudnych do rozwiązania analitycznego. Podstawę metody stanowi losowanie wielkości charakteryzujących proces. Metoda ma szczególne zastosowanie w obliczeniach inżynierskich, gdzie ważna jest szybkość, a mniej ważna precyzja.

14.5.1. Wyznaczanie przybliżonej wartości liczby π

Pierwszy prosty przykład pokazuje metodę znalezienia przybliżonej wartości liczby $\boldsymbol{\pi}$.

Losuje się *N* par liczb z przedziału (-$\frac{A}{2}$, $\frac{A}{2}$) jako współrzędne punktów wewnątrz kwadratu o boku A, czyli "strzela" w kwadratową tarczę o boku równym A. Następnie zlicza się liczbę N_t trafień bliższych niż $\frac{A}{2}$, licząc od środka tarczy, czyli trafień w okrąg o promieniu R=$\frac{A}{2}$.

Liczba N_t w stosunku do wszystkich *N* strzałów ma się jak powierzchnia okręgu do powierzchni kwadratu:

$$\frac{N_t}{N} = \frac{\pi\left(\frac{A}{2}\right)^2}{A^2}$$

stąd:

$$\pi = \frac{4\,N_t}{N}$$

Powyższy ogólny opis algorytmu można opisać następującym pseudokodem:

1. Wypełnia się zerami macierz *W* o wymiarach 2×*N*,
2. Inicjowana jest zmienna *Nt*, reprezentująca licznik trafień w okrąg, na wstępie jest równa zeru,
3. W pętli o *N* powtórzeniach dokonuje się losowania dwóch wartości *x* i *y* jako współrzędnych punktu (przy założeniu A=1 losuje się dwie liczby z przedziału (-0.5, 0.5)); w każdym kroku pętli badana jest odległość od środka tarczy według wzoru: d=$\sqrt{x^2 + y^2}$
   - Jeżeli odległość ta jest mniejsza lub równa promieniowi okręgu R=0.5, współrzędne wpisywane są do wektora (do komórek o indeksach *(1,n)* i *(2,n)*, gdzie *n* jest aktualną wartością licznika pętli) i zwiększana jest wartość *Nt* o 1,
   - Jeżeli odległość ta jest większa od promienia okręgu (brak trafienia w okrąg przechodzimy do następnego losowania,
4. Po zakończeniu pętli obliczane jest wyrażenie *4*Nt/N*, co stanowi przybliżoną wartość liczby π,
5. Macierz W służy do narysowania wykresu punktów trafień. Pozostałe pary punktów w macierzy *W* pozostają zerami, też są rysowane na wykresie, lecz w samym centrum "tarczy" – w punkcie o współrzędnych (0, 0).

Poniżej treść skryptu i uzyskany wykres:

```
clc,clear
N = 100000;
W = zeros(2,N); Nt = 0;
for n = 1:N
    x = rand-0.5;
    y = rand-0.5;
    if  x^2+y^2 <= 0.25
        W(1,n) = x;
        W(2,n) = y;
        Nt = Nt + 1;
    end
end
plot(W(1,:),W(2,:), 'g')
format long
naszePI = 4*Nt/N
format short
```

```
naszePI =
   3.14320000000000
```

14.5.2. Wyznaczanie przybliżonej wartości całki oznaczonej

Drugi przykład dotyczy przybliżonego obliczania całek oznaczonych. Metoda *Monte Carlo* polega tu na wylosowaniu *N* liczb c_k z przedziału (0, 1), następnie obliczeniu dla każdej z nich wyrażenia:

$$x_k = a + (b - a)c_k \qquad \textit{dla } k=0,1,2,\ldots,N$$

gdzie a i b są granicami całkowania.

Szukaną wartość całki oznaczonej opisuje zależność:

$$\int_a^b f(x)dx \approx \frac{b-a}{N}\sum_{k=1}^{N} f(x_k)$$

W zamieszczonym poniżej skrypcie obliczono wartość całki oznaczonej:

$$\int_{\frac{\pi}{4}}^{\frac{3\pi}{4}} sin(x)\, dx$$

z wykorzystaniem funkcji *int* pakietu *Symbolic Toolbox* i porównano z rezultatem metody *Monte Carlo* według przepisu przytoczonego powyżej.

```
clc,clear
syms x a b

%metoda symbolicznej całki oznaczonej
f = sin(x);
p1 = int(f,x,a,b);
a = pi/4; b = 3*pi/4;
p1 = double(subs(p1));   %podstaw i przelicz
fprintf('Wynik całkowania  int( ) :%f\n',p1);

%metoda Monte Carlo
N = 1000; s = 0;
for  n = 1:N
   los = rand;                %losuj liczbę
   x = a+(b-a)*los;
   f1 = double(subs(f)); %podstaw i przelicz
   s = s+f1;                  %dodaj do sumy
end
p2 = (b-a)/N*s;
fprintf('Wynik metody Monte Carlo :%f\n',p2);
```

```
Wynik całkowania int( ) :1.414214
Wynik metody Monte Carlo:1.416027
```

Obydwie metody dają zbliżone rezultaty.

14.6. Transformacje geometryczne

Jeżeli kontur zamknięty jest opisany macierzą:

$$F = \begin{bmatrix} x_1 & x_2 & x_3 \cdots \\ y_1 & y_2 & y_3 \cdots \\ 1 & 1 & 1 \cdots \end{bmatrix}$$

i zdefiniuje się macierze transformacji figur płaskich:

macierz przesunięcia: $M_{prz} = \begin{bmatrix} 1 & 0 & p_x \\ 0 & 1 & p_y \\ 0 & 0 & 1 \end{bmatrix}$

macierz skalowania: $$M_{ska} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

macierz obrotu: $$M_{obr} = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

to macierz wyznaczająca nowy kontur po przekształceniu opisuje równanie:

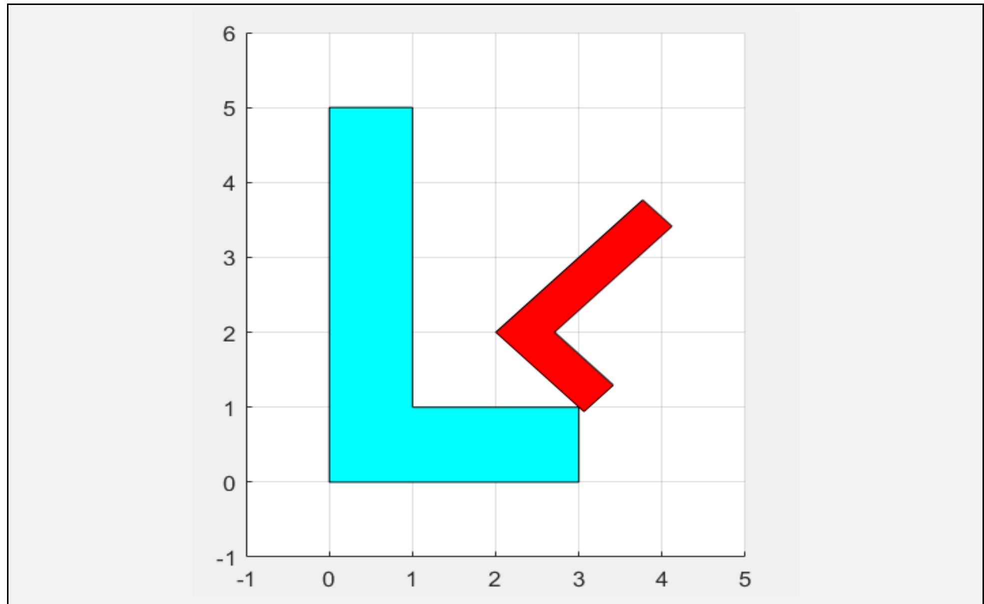
$$F_{tr} = M_{prz} * M_{ska} * M_{obr} * F$$

Skrypt, w którym przekształcenia macierzy wykonane będą symbolicznie, przy założonych współczynnikach transformacji:

$$p_x = p_y = 2, \quad s_x = s_y = 0.5, \quad \alpha = \frac{\pi}{4}.$$

przedstawia się następująco:

```
clc,clear ,close
hold on
%macierz konturu wyjściowego
F = [0 0 1 1 3 3 0
     0 5 5 1 1 0 0
     1 1 1 1 1 1 1];
fill(F(1,:), F(2,:), 'c');
axis([-1  6  -1 6]);
axis equal                        %równe jednostki osi x i y
syms px py sx sy kat

% macierze transformacji
Prz = [1 0 px;  0 1 py;  0 0 1];
Ska = [sx 0 0; 0 sy 0; 0  0 1 ];
Obr = [cos(kat)  sin(kat)  0; -sin(kat) cos(kat)  0; 0 0 1];

% macierz nowego konturu po transformacji
Ft = Prz*Ska*Obr*F;
% dane
px = 2; py = 2; sx = 0.5; sy = 0.5; kat = pi/4;
Ft = subs(Ft);
fill(Ft(1,:),  Ft(2,:), 'r')
grid
axis([-1  5  -1  6])
```



Wykorzystano funkcję *fill* do utworzenia wykresu konturu zamkniętego, przed i po transformacji.

15. Aproksymacja i interpolacja

15.1. Aproksymacja

Metoda **aproksymacji** polega na znalezieniu funkcji *f(x)*, której wykres przechodzi w pobliżu zbioru zadanych punktów, zaś sama funkcja minimalizuje wartość pewnego kryterium dopasowania (np. aby suma kwadratów odchyłek była jak najmniejsza).

Przykładowo, problem pojawia się przy posiadaniu wielu dyskretnych punktów pomiarowych i celem jest znalezienie funkcji aproksymującej.

Funkcja aproksymująca nie musi wcale przyjmować identycznych wartości jak funkcja aproksymowana.

MATLAB dostarcza przydatną funkcję *polyfit*, której zadaniem jest znalezienie współczynników wielomianu *n*-tego stopnia, jako funkcji aproksymującej.

Postać funkcji *polyfit*:

p = **polyfit**(*X*, *Y*, *n*)

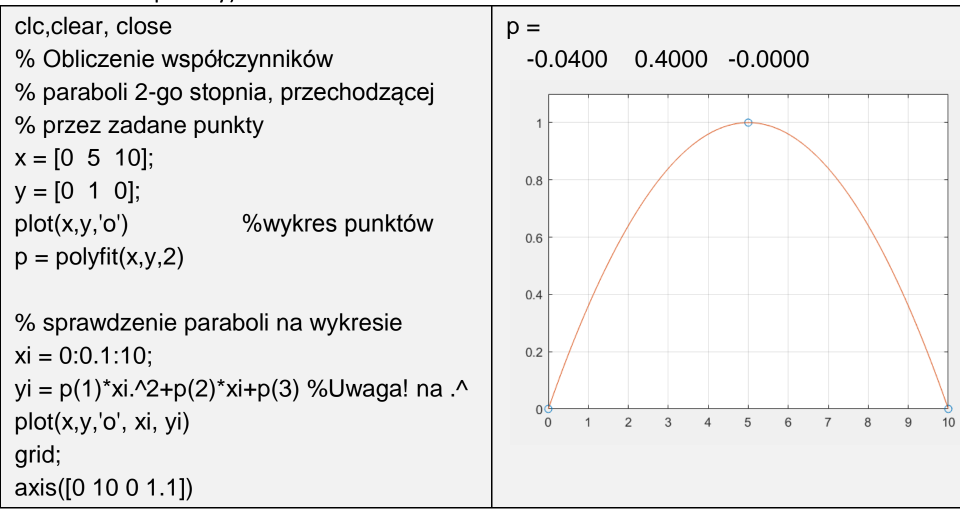
gdzie:

X, Y – wektory współrzędnych dla zadanych punktów,

n – rząd wielomianu.

p – wektor współczynników wielomianu rzędu *n*,

Oto prosty skrypt, ilustrujący wykorzystanie powyższej funkcji ("kółeczkami" oznaczono zadane punkty):



```
clc,clear, close
% Obliczenie współczynników
% paraboli 2-go stopnia, przechodzącej
% przez zadane punkty
x = [0  5  10];
y = [0  1  0];
plot(x,y,'o')              %wykres punktów
p = polyfit(x,y,2)

% sprawdzenie paraboli na wykresie
xi = 0:0.1:10;
yi = p(1)*xi.^2+p(2)*xi+p(3) %Uwaga! na .^
plot(x,y,'o', xi, yi)
grid;
axis([0 10 0 1.1])
```

```
p =
  -0.0400    0.4000   -0.0000
```

Rząd wielomianu powinien być mniejszy od liczby zadanych punktów, inaczej wielomian nie będzie jednoznaczny, co skutkuje ostrzeżeniem w *Command Window*:

Warning: Polynomial is not unique; degree >= number of data points

Ostrzeżenie: wielomian nie jest unikalny;
stopień> = liczba punktów danych.

Przy wyższych rzędach wielomianu wygodne jest stosowanie funkcji *polyval*, wyznaczającej współrzędne *y* dla zadanego wektora wartości *x* i wektora współczynników wielomianu:

y = **polyval**(p, x)

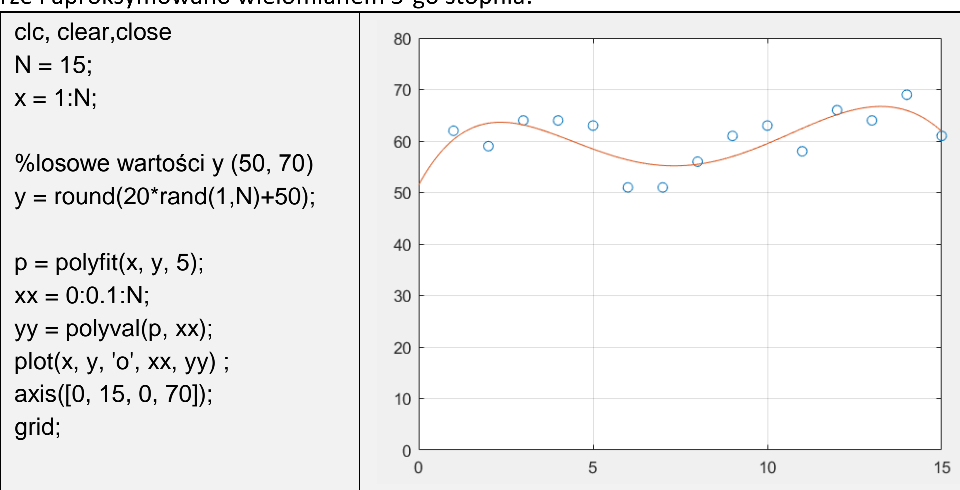
gdzie:

x – wektor wartości zmiennej niezależnej,

p – wektor współczynników wielomianu (w porządku malejącym) wielomianu rzędu: length(p)-1,

y – szukane wartości zmiennej zależnej w punktach wektora *x*.

Poniżej przykład zastosowania w skrypcie, w którym wylosowano 15 liczb w wektorze i aproksymowano wielomianem 5-go stopnia:



```
clc, clear,close
N = 15;
x = 1:N;

%losowe wartości y (50, 70)
y = round(20*rand(1,N)+50);

p = polyfit(x, y, 5);
xx = 0:0.1:N;
yy = polyval(p, xx);
plot(x, y, 'o', xx, yy) ;
axis([0, 15, 0, 70]);
grid;
```


15.2. Interpolacja

Jeżeli znane są dokładne wartości funkcji w punktach węzłowych przedziałów, interpolacja przybliża przebieg w każdym przedziale przy pomocy funkcji z pewnej klasy (np. wielomianami, funkcjami trygonometrycznymi, funkcjami sklejanymi itp.). Zadaniem interpolacji jest zatem, przy założonej funkcji interpolującej, znalezienie przybliżonych wartości funkcji w punktach nie będących węzłami. W tym celu należy znaleźć w każdym przedziale pomiędzy węzłami, parametry funkcji interpolującej *W(x)*, która w węzłach interpolacji przyjmuje założone wartości.

Interpolacja jest zadaniem odwrotnym do tablicowania funkcji. Przy tablicowaniu mając analityczną postać funkcji tworzona jest tablica określonych wartości, natomiast interpolacja, na podstawie tablicy wartości funkcji, określa jej postać analityczną i z niej uzyskuje się współrzędne punktów pośrednich.

W *MATLAB-ie* mamy dostępną funkcję *interp1*:

y_i = **interp1**(X, Y, x_i, *metoda*)

gdzie:

X, Y – wektory współrzędnych dla punktów węzłowych,

x_i - wektor wartości *x*, dla których szukamy wektora współrzędnych *y*,

metoda jest jedną z:

'linear' - liniowa,

'pchip' - sześcienna,

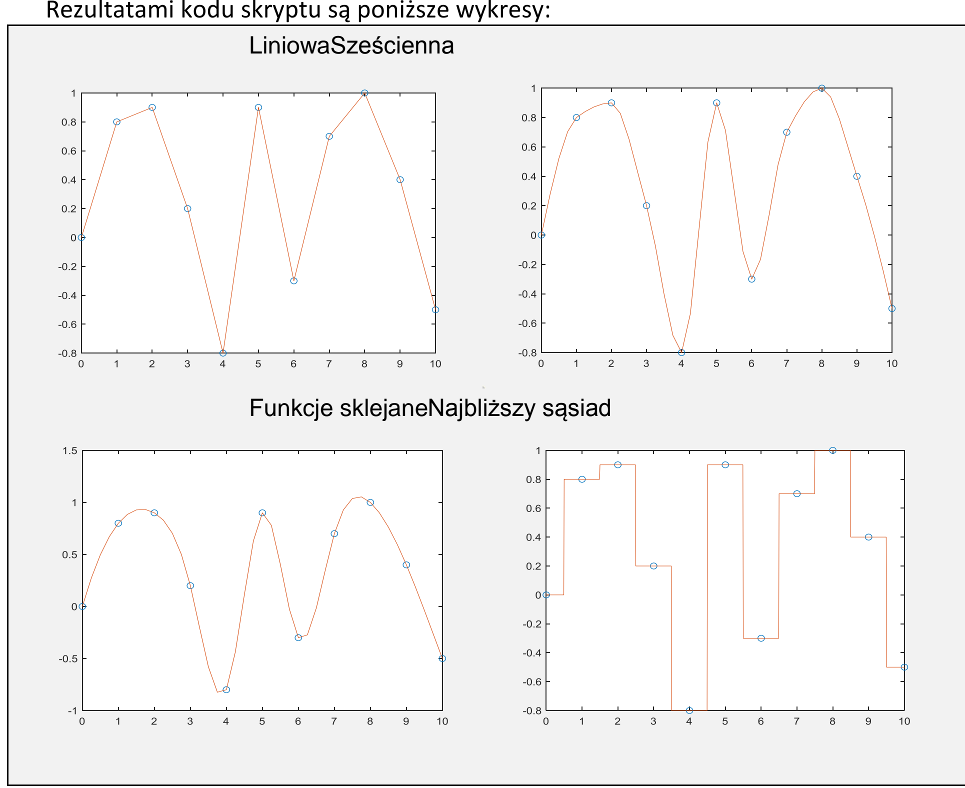
'spline' - przy pomocy funkcji sklejanych,

'nearest' - interpolacja "najbliższy sąsiad"

Zostanie utworzony skrypt, ilustrujący na wykresach poszczególne metody interpolacji. Na każdym wykresie umieszczono "kółeczka" w węzłach interpolacji:

```
clc, clear
x = 0:10;                  %współrzędne x węzłów
y = [0  0.8 0.9 0.2 -0.8 0.9 -0.3 0.7 1 0.4  -0.5]
xi = 0: 0.01:10;           % dla tych xi poszukiwane będą yi

%metoda liniowa
figure(1)
yi = interp1(x, y, xi, 'linear');
plot(x, y,'o', xi, yi)

%metoda sześcienna
figure(2)
yi = interp1(x, y, xi, 'pchip');
plot(x, y,'o', xi, yi)

% funkcje sklejane
figure(3)
yi = interp1(x, y, xi, 'spline');
plot(x, y, 'o', xi, yi)

%metoda "najbliższy sąsiad"
figure(4)
yi = interp1(x, y, xi, 'nearest');
plot(x,y,'o',xi,yi)
```

Rezultatami kodu skryptu są poniższe wykresy:

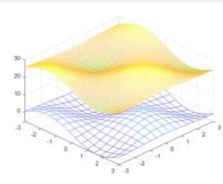


Można stosować interpolację dla przebiegu funkcji dwóch zmiennych, wykorzystując funkcję *interp2*:

z_i= interp2 (*X*, *Y*, *Z*, x_i, y_i, *metoda*)

Poniżej przykład, w którym utworzono funkcję *Z(x,y)* powierzchni z "rzadkimi" punktami, interpolacja określiła punkty pośrednie funkcji *ZI(x,y)*, według metody liniowej. Przesunięto powierzchnię $Z_i(x_i,y_i)$ w górę na wykresie dla lepszego oglądu.

```
clc, clear
hold on
%siatka "rzadka"
[X,Y] = meshgrid(-3: 0.4: 3);
Z = 5*sin(X).*cos(Y);

%siatka "gęsta"
[XI,YI] = meshgrid(-3: 0.1: 3);
ZI = interp2(X, Y, Z, XI, YI, 'linear');

mesh(X,Y,Z)
mesh(XI, YI, ZI+25) %przesunięta o 25 w górę
axis([-3 3 -3 3 -5 30])
view(45, 45)
```



16. Podstawowe zasady korzystania z pakietu *Simulink*

Simulink to zestaw narzędzi (*toolbox*) *MATLAB-a*, który służy do przeprowadzenia symulacji układów (mechaniki, automatyki, elektroniki itp.). Pakiet pozwala budować modele symulacyjne w postaci graficznej, z wykorzystaniem specjalnych bloków o zróżnicowanym działaniu. Przy pomocy narzędzia *Simulink* można przeprowadzać symulacje dyskretne i ciągłe, modelować układy liniowe i nieliniowe.

Simulink wyposażony jest w interfejs graficzny, pozwalający budować modele układów jako schematów blokowych.

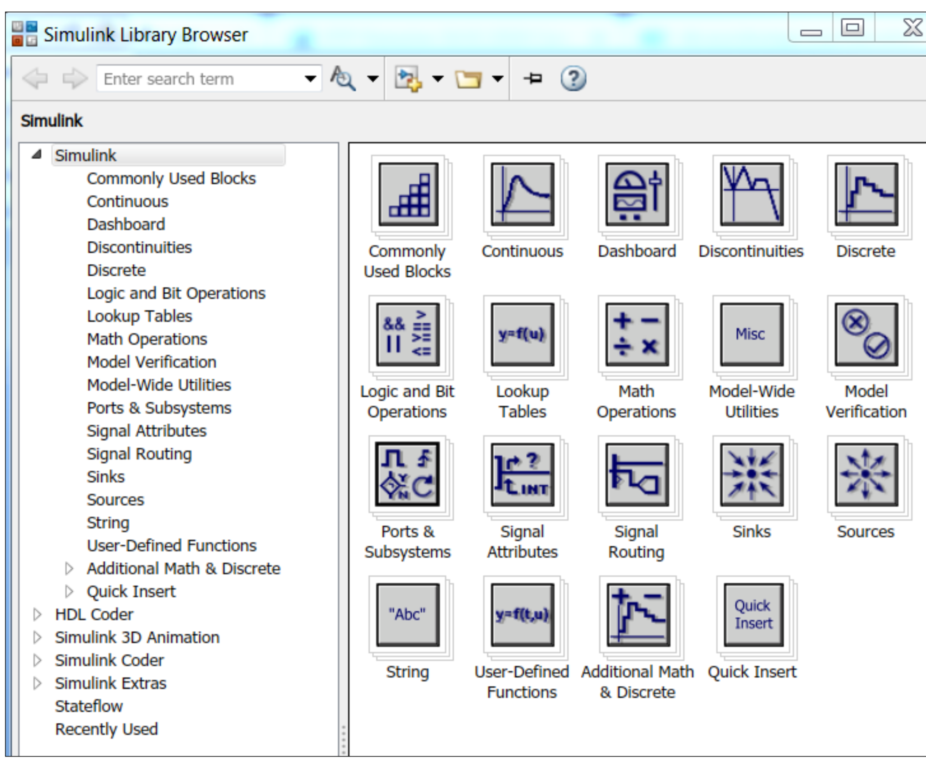
Po wpisaniu w wierszu poleceń *Command Window* polecenia:

```
>>simulink
```

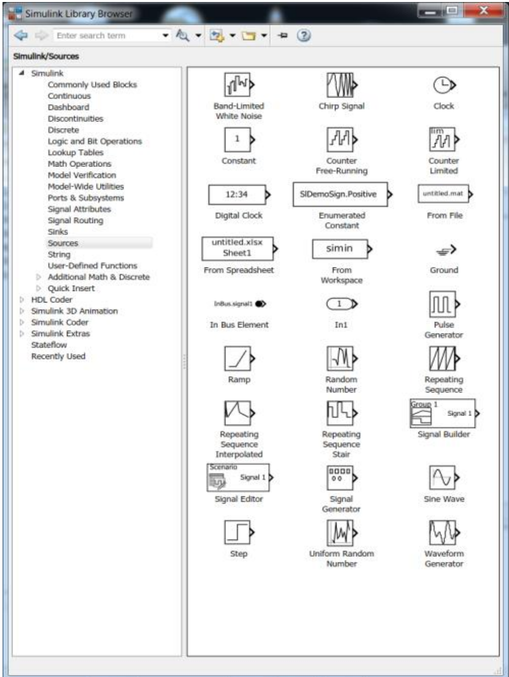
ukazuje się okno wyboru projektów i możliwość otwarcia zapisanego i utworzenia nowego modelu. Podobną inicjację działania *Simulink-a* można wykonać z menu *Home/Simulink*.

W oknie modelu można korzystać (poprzez menu *Tools/Library browser*) z bibliotek bloków: źródeł i rejestratorów sygnału, elementów liniowych i nieliniowych

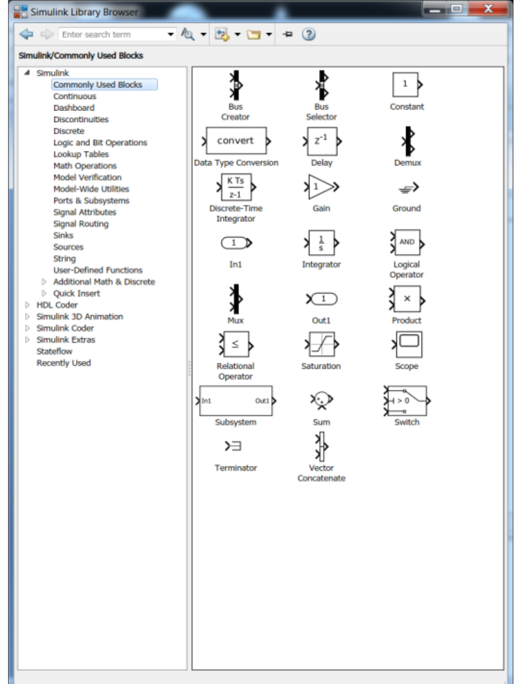
Modele elementów są pogrupowane hierarchicznie (rys.16.1).



Rys.16.1. Okienko listy bibliotek bloków



Rys.16.2. Okienko biblioteki: *Commonly used blocks*



Rys.16.3. Okienko biblioteki *Sources*

Najważniejsze biblioteki to:

- *Commonly used blocks* – najczęściej wykorzystywane bloki,
- *Continous* - bloki funkcji ciągłych, takie jak pochodna i integrator,
- *Discontinouities* – nieciągłe bloki funkcyjne,
- *Discrete* – bloki dyskretne,
- *Logic and Bit Operations* – logika lub bloki operacji bitowych, takie jak operator logiczny i operator relacyjny,
- *Math Operations* – operacje matematyczne, bloki funkcyjne, takie jak wzmacniacz (*gain*), iloczyn, sumator itp.,
- *Sinks* – rejestratory sygnałów, mierniki, oscyloskop,
- *Sources* (źródła) – generowanie lub importowanie danych sygnałowych.

Wykorzystując oscyloskopy można obserwować przebiegi czasowe sygnałów układu.

Poniższy przykład zawiera analizę modelu oscylatora harmonicznego z tłumieniem, który analizowany był w rozdziale (15.1) z wykorzystaniem funkcji *dsolve*, rozwiązującej równania różniczkowe modelu:

$$\frac{\mathrm{d}^2\mathrm{y}}{\mathrm{dt}^2} + 2\beta\frac{\mathrm{dy}}{\mathrm{dt}} + {\omega_0}^2\mathrm{y(t)} = 0$$

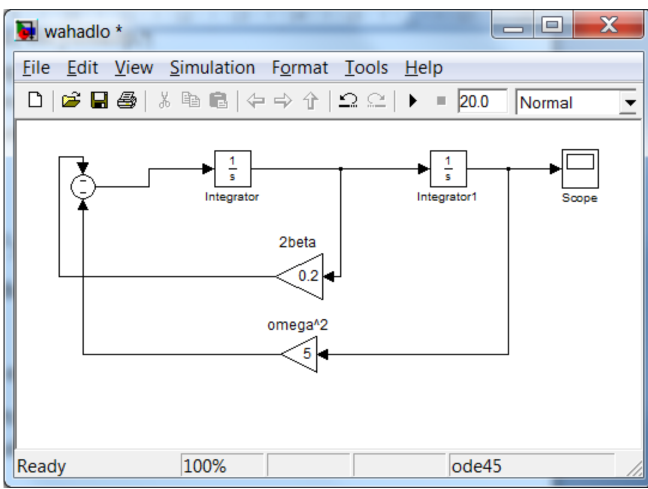
gdzie: ω = częstość drgań, β– współczynnik tłumienia.

Po przekształceniu równania do postaci:

$$\frac{d^2y}{dt^2} = -2\beta\frac{dy}{dt} - {\omega_0}^2 y(t)$$

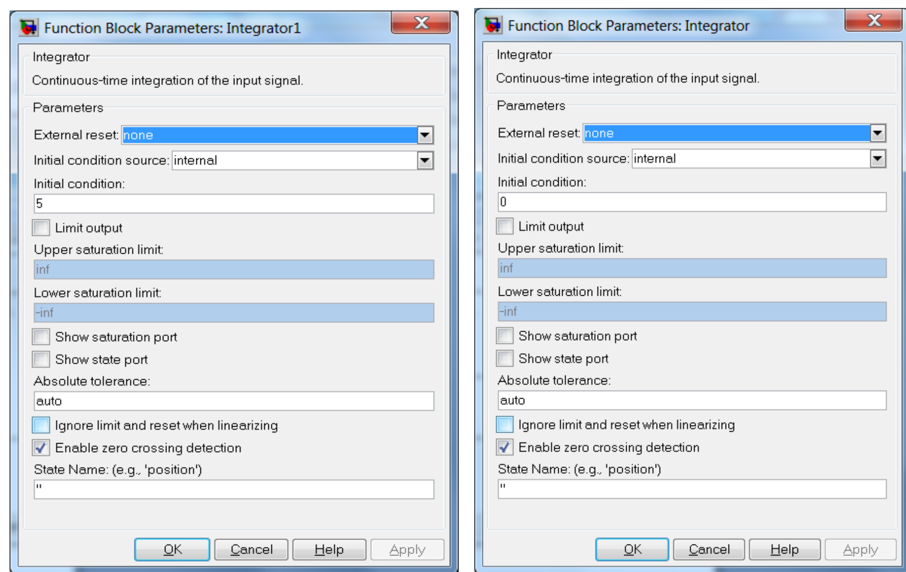
na jego podstawie zostanie utworzony nowy model (*File/New/Model*).

Odpowiednie bloki przeciąga się myszką z biblioteki do okna modelu. Wykorzystano tu blok całkujący (*integrator*), wzmacniacz (*gain*) i sumator (*sum*). Bloki są łączone strzałkami kierunkowymi – również przy pomocy myszki (rys.16.4).



Rys.16.4. Schemat blokowy układu

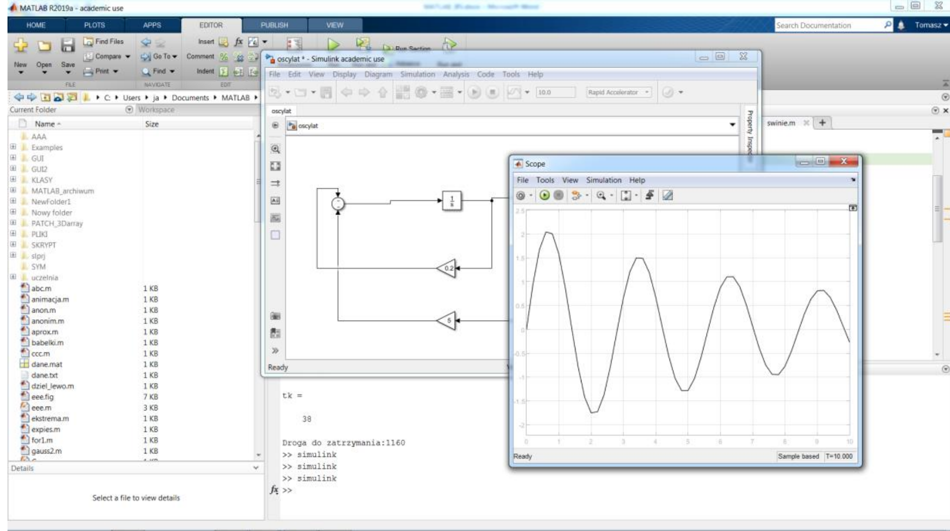
Istotne jest ustawienie odpowiednich warunków początkowych dla równania różniczkowego. W parametrach integratorów (okienka wywoływane z menu kontekstowego bloku) można wpisać wartość wychylenia początkowego równego 5 i prędkości początkowej równej *0* (rys.16.5):



Rys.16.5. Okna parametrów integratorów

Czas symulacji ustawiono na 10 sekund. Podwójne kliknięcie w rejestrator (*Scope*) otwiera okno do obserwacji sygnału (szukanej funkcji wychylenia *y(t)*). Uruchomienie symulacji odbywa się przyciskiem ***Run*** w oknie modelu.

Rozwiązanie graficzne problemu (obraz z rejestratora) przedstawia rys.16.6.



Rys.16.6. Przebieg funkcji *y(t)* dla oscylatora harmonicznego na rejestratorze.

17. Programowanie obiektowe

17.1. Zasady ogólne

Programowanie zorientowane obiektowo (OOP – *Object Oriented Programming*) umożliwia przedstawienie problemu w postaci logicznie powiązanych ze sobą struktur danych zwanych obiektami, wymieniających informacje między sobą.

„Obiektowość" opiera się na koncepcyjnym klasyfikowaniu rzeczywistości. Na świat składają się obiekty i procesy zachodzące z ich udziałem lub bez. System reaguje na zdarzenia („siły sprawcze"), których efektem jest zainicjowanie:

- funkcji przetwarzania parametrów obiektów,
- funkcji przesyłu informacji między obiektami,
- funkcji oddziaływania jednych obiektów na inne.

Klasa (lub typ obiektowy) to definicja złożonej struktury danych o określonej liczbie atrybutów.

Atrybuty klasy dzieli się na:

- **pola** (ang. *properties, fields*) – właściwości opisane wartościami dowolnych typów (liczby, teksty, także typy strukturalne),
- **metody** (ang. *methods*) – funkcje wykonywane na polach; metoda jest czynnością wykonywaną na obiekcie, podprogramem (najczęściej funkcją); metoda obiektu operuje na polach (danych) obiektu-właściciela metody lub innych obiektów, przy pomocy metod możliwy jest dostęp do pól i wpływ na ich wartości.

Dla celów bardziej złożonych aplikacji tworzone są zbiory klas, ułożonych zwykle w hierarchii. Tworzone są klasy nadrzędne o podstawowych cechach i zachowaniach, na ich podstawie są konstruowane, wywodzące się z nich, klasy potomne. Klasy potomne posiadają wszystkie atrybuty "przodka" (niekiedy je przedefiniowujące) i mogą być wyposażone w atrybuty dodatkowe. Mechanizm ten nosi nazwę **dziedziczenia** (ang. *inheritance*).

Przykładowo: klasa *Człowiek* ma pewne właściwości łatwe do opisania (pola: *nazwisko*, *wzrost* i inne, metody: *śpij*, *pracuj* itd.), klasa *Student* posiada wszystkie atrybuty klasy *Czlowiek* oraz dodatkowe właściwości, np. *numer_indeksu*, *rok_studiów*, oraz metody, np. *zdawaj_ezamin, zalicz semestr* i inne.

17.2. Definicja klasy

Definicja klasy (typu obiektowego) wymaga osobnego *M-pliku* o nazwie:

nazwa_klasy.m

Plik definicyjny klasy zawiera nagłówek z nazwą klasy. Treść pliku dzieli się na dwie części:

- opis pól (***properties*** – właściwości),
- opis metod (***methods*** – funkcji).

Postać ogólna definicji klasy:

```
classdef  nazwa_klasy                 %nagłówek klasy

  properties                          % POLA (WŁAŚCIWOŚCI)
     nazwa1 = wartość
     nazwa2 = wartość
     nazwa3 = wartość
     nazwa4            %niektóre właściwości nie muszą mieć wartości
     ...itd
  end

  methods                             % METODY
      function obj = nazwa_klasy(argumenty)      %metoda konstruktor
         treść % działania inicjacyjne
      end

     function nazwa_f1(argumenty)
         treść
      end

     function nazwa_f2(argumenty)
         treść
      end
      ...itd

     function delete(obj)                        %metoda destruktor
         treść
      end
  end
end
```

Zwykle pierwszą funkcją jest tzw. **konstruktor** (funkcja o nazwie tożsamej z nazwą klasy), czyli metoda opisująca działania potrzebne do utworzenia obiektu danej klasy (np. nadanie imienia, ustalenie pewnych wstępnych właściwości itp.).

Usuwanie utworzonego egzemplarza (instancji) obiektu wykonuje **destruktor**, metoda o nawie *delete*.

Utworzenie szkieletu klasy ułatwia pozycja menu *Home/New/Class*.

Programy wykorzystujące definicję klasy mogą tworzyć wiele obiektów danego typu i zarządzać nimi, wykorzystując tylko te właściwości i metody, które posiada klasa.

17.3. Przykładowe programy zarządzania obiektami

Przykład 1

Przeanalizujmy plik *Pies.m*, który definiuje klasę o nazwie *Pies*:

```
classdef   Pies <handle
    properties              % właściwości
        imie                % na razie bez wartości
        nogi = 4;           % 4 nogi
        ileHau = 0;         % początkowo zero szczeknięć
    end
  methods                            % metody
        function obj = Pies( )       % metoda konstruktor
          % działania inicjacyjne
        end
        function nadajImie(obj, x)   % metoda nadawania imienia x
            obj.imie = x;
            fprintf( 'Nadano imię: %s\n', x);
        end
        function Hau(obj)            % funkcja szczeknięcia
            disp('hauu');
            obj.ileHau = obj.ileHau + 1;
            fprintf( ' %d raz\n' ,obj.ileHau);
        end
        function ileNog(obj)         %wypisz liczbę nóg
            fprintf('Ma nóg: %d\n',obj.nogi);
        end
        function delete(obj)         %metoda destruktor
        end
  end
end
```

Zdefiniowana klasa zawiera trzy właściwości:

- *imie* - wstępnie nie posiada wartości,
- *nogi* – każdy pies ma cztery nogi,
- *ileHau* – liczba szczeknięć, początkowo zero.

oraz pięć metod – funkcji konstruowanych według zasad, które przedstawiono w rozdz. 11):

- *Pies* - funkcja konstruktora, w tym przypadku jest bezargumentowa, jedynym jej rezultatem jest utworzenie egzemplarza (tzw. instancji) obiektu,
- *nadajImie* – funkcja, której argumentami są: *obj* (obiekt tej klasy) i *x* (zmienna typu tekstowego, imię psa),
- *Hau* – wykonanie tej funkcji dla danego obiektu spowoduje wypisanie tekstu szczeknięcia, zwiększenie licznika szczeknięć (na początku licznik był wyzerowany) i wypisanie, ile razy szczeknął dotychczas dany egzemplarz psa,

- *ileNog* - funkcja wypisująca ile nóg ma pies,
- *delete* - destruktor obiektu klasy.

Funkcje można przydzielić do pewnych grup. Przede wszystkim funkcje działające na polach obiektu:

– typu ***set*** (nadaj wartość) – czyli nadające lub modyfikujące wartości pól; w przykładzie jest to funkcja *Hau*, zwiększająca wartość pola *ileHau* o 1,

– typu ***get*** (pobierz wartość) – czyli pobierające dane z pól; w przykładzie funkcja *ileNog*, wypisująca wartość właściwości *nogi*.

Posiadając definicję klasy, można napisać program, który tworzy obiekty (zmienne o różnych nazwach, egzemplarze obiektów), instancje w klasie, uruchamia metody obiektów.

Zapis właściwości lub metody obiektu jest kwalifikowany, to znaczy wskazuje na obiekt, którego dotyczy i na nazwę właściwości lub metody (sposób jak w strukturach):

nazwa_obiektu.nazwa_pola

oraz

nazwa_obiektu.nazwa_metody(*argumenty_aktualne*)

Można zatem utworzyć aplikację, która będzie:

- tworzyć (ewentualnie usuwać) obiekty,
- pobierać, nadawać i zmieniać ich właściwości (zwykle z wykorzystaniem metod typu *get* i *set*),
- uruchamiać inne funkcje obiektów (także oddziaływań na inne obiekty).

Oto przykładowe instrukcje programu użytkowego:

```
pies1 = Pies()                % utworzy obiekt o nazwie pies1, należący do klasy Pies,
pies1.nadajImie('Burek')      % wykona funkcję typu set, nadającą imię danemu psu,
pies1.ileNog(pies1)           % wykona funkcję typu get, pobierającą liczbę nóg.
```

Po wykonaniu metody usuwającej obiekt:

```
pies1.delete( )
```

Próba wykonania metody:

```
pies1.ileNog( )
```

zwróci błąd:

```
Invalid or deleted object.
```

Niewłaściwy lub usunięty obiekt

Przykładowy skrypt zarządzający obiektami klasy *Pies*:

```
 clc,clear
pies1 = Pies( );          %utworzenie obiektu pies1
pies1.nadajImie('Burek');%nadanie psu imienia
pies1.Hau();              % wykonanie metody: szczeknięcie tego psa
pies1.Hau();              % drugie szczeknięcie tego psa
fprintf('Ma nóg:%d\n',pies1.nogi);%wypisanie wartości właściwości(1)
pies2 = Pies();           %utworzenie innego obiektu pies2
pies2.nadajImie('Ciapek'); %nadanie psu imienia
for   k = 1:3             % 3 szczeknięcia psa1 w pętli
      pies2.Hau( );       %wykonanie metody
end
for i = 1:3
      Hau(pies2);         %tak też można wykonać metodę Hau
end
fprintf('%s szczeknął %d razy\n',pies1.imie,pies1.ileHau);   %wypisz informacje(2)
fprintf('%s szczeknął %d razy\n',pies2.imie,pies2.ileHau);   %wypisz informacje(3)
```

Wynik wykonania skryptu:

```
Nadano imię: Burek
hauu 1 raz
hauu 2 raz
Ma nóg:4
Nadano imię: Ciapek
hauuuuuuuuuu 1 raz
hauu 2 raz
hauu 3 raz
hauu 4 raz
hauu 5 raz
hauu 6 raz
Burek szczeknął 2 razy
Ciapek szczeknął 6 razy
```

W powyższym przykładzie, w instrukcjach (1), (2) i (3) wykorzystano bezpośredni dostęp do pól (właściwości).

W większości realizacji aplikacji obiektowych, pola klasy mają charakter prywatny, to znaczy dostęp do nich (pobranie wartości, zmiana wartości) może się odbywać jedynie przy pomocy odpowiednich metod (publicznych). Zasada ta nosi nazwę **hermetyzacji**.

Jeżeli w definicji klasy nastąpi poniższa korekta, ustalająca prywatny dostęp do wszystkich pól:

```
classdef Pies<handle
   properties (Access = private)  %wszystkie właściwości prywatne
      imie
      nogi = 4;
      ileHau = 0;
   end
  %...reszta jak poprzednio
```

to przy wykonaniu instrukcji programu:

```
fprintf('Ma nóg:%d\n',pies1.nogi);
```

pojawi się błąd:

```
No public property 'nogi' for class 'Pies'.
```

Nie istnieje publiczna właściwość 'nogi' dla klasy 'Pies'

Dostęp do właściwości *nogi* jest możliwy tylko z wykorzystaniem publicznej metody *ileNog( )*:

```
pies1.ileNog();
```

Przykład 2

Przykład definicji klasy *Czlowiek* w pliku *Czlowiek.m*:

```
%definicja klasy
classdef   Czlowiek<handle
   properties (Access = private)
      waga;
   end
   methods % metody
      function obj = Czlowiek(x)
         obj.waga = x;
      end
      function jedz(obj, posilek)
           obj.waga = obj.waga+posilek/20;
     end
     function wypiszWage(obj)
           fprintf('Waga:%f\n', obj.waga);
      end
   end
end
```

Konstruktor ma argument *x*, reprezentujący wstępną wagę. Metoda *jedz()* zawiera kod, który określa, jak zmienna *posilek*, będąca argumentem metody, wpływa na właściwość *waga* (tu określono przykładowo przyrost wagi ciała o 1/20 wagi posiłku).

Skrypt wykonawczy:

```
clc,clear
osoba1 = Czlowiek( 90 ); %utworzenie obiektu o wadze 90
osoba1.wypiszWage( );
osoba1.jedz(10)
osoba1.wypiszWage( );
osoba1.jedz(10)
osoba1.wypiszWage( );
```

wyświetli poniższy rezultat:

```
Waga:90.000000
Waga:90.500000
Waga:91.000000
```

Dwukrotne zjedzenie posiłku o wartości 10 skutkuje wzrostem wagi z 90 do 91.

Przykład 3

Poniższy przykład podano bez komentarzy, jego analizę pozostawiono czytelnikowi.
Definicja klasy *Faktura* w pliku *Faktura.m*:

```
classdef   Faktura<handle
    properties
        nr
        liczba_pozycji = 0
        pozycja = struct('lp', 0 , 'towar', '', 'cena', 0)%struktura dla pozycji faktury
        spis = struct('lp', 0 , 'towar', '', 'cena', 0)  %tablica 1-elementowa
    end
    methods
        function obj = Faktura(x)
           % konstruktor: nadaje numer faktury
           obj.nr = x ;
        end
        function wpiszPozycje(obj,a,b,c)
          obj.pozycja.lp = a;
          obj.pozycja.towar = b;
          obj.pozycja.cena = c;
          obj.liczba_pozycji = obj.liczba_pozycji+1;%zwiększ liczbę pozycji
          obj.spis(obj.liczba_pozycji) = obj.pozycja;%dodajemy pozycję do spisu
        end
    end
end
```

Plik wykonawczy, wykorzystujący definicję klasy *Faktura*:

```
clc
clear
f1 = Faktura(1);
fprintf('Numer faktury:%d\n',f1.nr);
f1.wpiszPozycje(1,'Cukier',2.2);
f1.wpiszPozycje(2,'Herbata',3.5);
f1.wpiszPozycje(3,'Sól',1.5);
%wydruk spisu
for  k = 1:length(f1.spis)
   fprintf('Poz:%2d  Towar:%10s  Cena:%0.2f\n',  ...
                  f1.spis(k).lp,f1.spis(k).towar,f1.spis(k).cena);
end

fprintf('    Liczba pozycji:%d\n',  f1.liczba_pozycji)
```

oraz wyniki jego działania:

```
Numer faktury:1
Poz: 1  Towar:    Cukier  Cena:2.20
Poz: 2  Towar:   Herbata  Cena:3.50
Poz: 3  Towar:       Sól  Cena:1.50
    Liczba pozycji:3
```

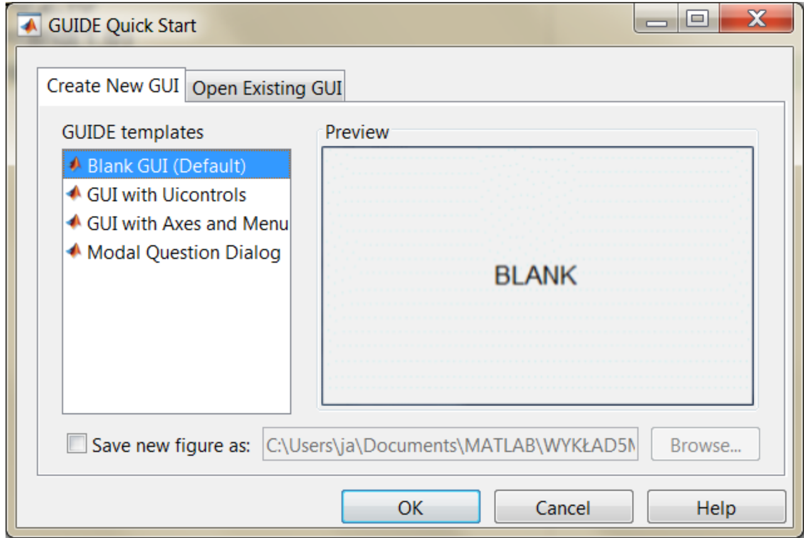
18.Tworzenie interfejsu graficznego *GUI*

18.1. Inicjacja narzędzia *GUI* (*Graphic User Interface*)

Uruchomienie narzędzi interfejsu graficznego odbywa się przez wpisanie w *Command Window* polecenia:

```
>>guide
```

Pojawia się okienko dialogowe z możliwością tworzenia nowego formularza (*Create New GUI*) lub edycji istniejącego (*Open Existing GUI*) jak na rys.18.1.

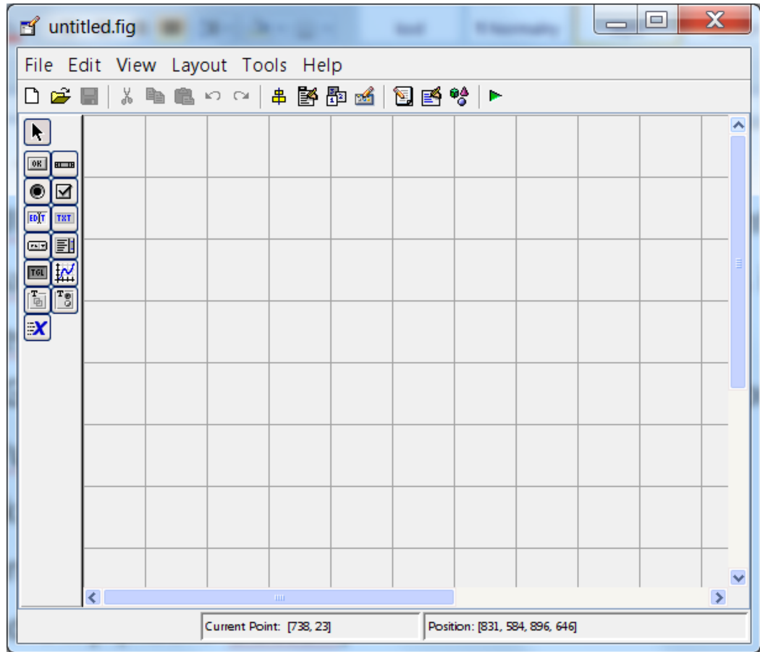


Rys.18.1. Okno inicjujące projektowanie aplikacji *GUI*

Przy tworzeniu nowego formularza *GUI* można dokonać wyboru:

- *Blank GUI* - pusty formularz,
- *GUI with UIcontrols* – formularz z podstawowymi elementami sterującymi,
- *GUI with Axes and Menu* – formularz zawierający wykres i listę rozwijaną,
- *Modal Question Dialog* - modalne okno dialogowe.

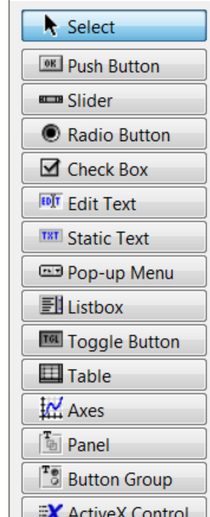
Rezultatem wybrania *Blank GUI* jest puste okno projektu aplikacji (rys 18.2). Okno zawiera pasek narzędziowy z dostępną paletą komponentów, możliwych do umieszczenia na formularzu (rys.18.3).



Rys.18.2 Okno projektu z pustym formularzem

Paleta, poniżej narzędzia *Select*, służącego do wybierania elementów, zawiera ikony następujących komponentów:

- Przycisk (*Push Button*)
- Suwak (*Slider*)
- Przycisk radiowy (*Radio Button*)
- Pole wyboru (*Checkbox*)
- Pole tekstowe (*Edit Text*)
- Etykieta (*Static Text*)
- Lista rozwijana (*Popup Menu*)
- Lista wyboru (*Listbox*)
- Przełącznik (*Toggle Button*)
- Wykres (*Axes*)
- Panel
- Grupa przycisków (*Button Group*)
- Kontrolka *ActiveX*



Rys.18.3. Paleta komponentów

Uwagi:

- zestaw komponentów może się różnić, w zależności od wersji *MATLAB-a*,
- można uwidocznić nazwy komponentów w palecie, ustawiając w menu *File/Preferences/GUIDE* projektu opcję: *Show names in component palette.*

Metodyka projektowania i oprogramowania aplikacji *GUI* zostanie pokazana na prostych przykładach.

18.2. Projektowanie aplikacji

Tworzymy nową aplikację – *Blank GUI*. Po wybraniu myszką potrzebnego komponentu, umieszczamy go w obszarze okna aplikacji. Komponent można łatwo przemieszczać i wymiarować.

Po dwukrotnym kliknięciu w dowolne miejsce obszaru okna aplikacji (lub w dowolny utworzony komponent) pojawia się okno ***Property Inspector*** – lista właściwości danego komponentu z wartościami domyślnymi.

Inne możliwości uruchomienia *Inspektora* to:

- kliknięcie w komponent prawym przyciskiem myszki i z rozwijanego menu wybranie opcji *Property Inspector*,
- z menu *View* wybranie *Property Inspector*.

W oknie *Property Inspector* można ustawić podstawowe informacje o danym komponencie, wartości możemy modyfikować. Wybór komponentu do ustalania właściwości ułatwia też widok okienka *Object Browser*, uruchamianego z menu *View*.

Ustalmy przykładowo wielkość okna aplikacji w inspektorze właściwości. Po dwukrotnym kliknięciu w obszar okna aplikacji *GUI* znajdujemy odpowiednie właściwości:

- *Units* – wybieramy: *centimeters*,
- *Position* – naciskamy znak „+" obok właściwości i ustalamy w rozwinięciu:

  width : 15 (szerokość – w centymetrach)

  height : 10 (wysokość – w centymetrach).

Napisy na przyciskach określa właściwość tekstowa *String*, dowolny tekst (może być wielowyrazowy, polskie znaki są dozwolone).

Bardzo istotna jest właściwość tekstowa *Tag* – nazwa komponentu, pod którą jest on identyfikowany w kodzie. Możemy zaakceptować nazwy domyślne (np. *pushbutton1*, *pushbutton2* itd,) lub nadawać własne nazwy. Tworzenie nazwy dla właściwości *Tag* podlega tym samym zasadom, co tworzenie identyfikatorów (początkowy znak to litera, dalej litery i cyfry, zakaz spacji i polskich znaków diakrytycznych, istotne duże i małe litery).

18.3. Dodanie funkcjonalności do aplikacji

Po utworzeniu projektu formularza i ustawieniu odpowiednich właściwości w *Inspektorze,* należy zapisać projekt w pliku. Powstaje plik *nazwa.fig* z projektem raz *M-plik* o identycznej nazwie, z rozszerzeniem *.m*, który jest otwierany w edytorze *MATLAB-a*. Zawiera on wiele funkcji dotyczących projektu i jego komponentów, wiele też anglojęzycznych komentarzy, które można przeczytać albo usunąć.

Dodanie funkcjonalności dla aplikacji wymaga ingerowania w kod *M-pliku*.

Istotne jest odnalezienie w *M-pliku* bloku funkcji *Callback* dla danego komponentu umieszczonego w projekcie. W treści tej funkcji można napisać instrukcje, które zostaną wykonane po zdarzeniu obsługi danego komponentu, np. kliknięcie przycisku, przesuwanie suwaka, kliknięcie w pole opcji itp.

M-plik, utworzony przez *MATLAB-a* dla aplikacji, jest zazwyczaj obszerny, znalezienie odpowiedniej funkcji *Callback* dla danego komponentu ułatwić może jego menu kontekstowe komponentu (np. przycisku *Push Button*), z którego należy wybrać opcję *View Callbacks/ Callback*.

Zostaniemy przekierowani do bloku odpowiedniej funkcji w *M-pliku*:

function *pushbutton1*_Callback(hObject, eventdata, handles)

Nazwa *pushbutton1* może być inna, jeżeli zmieniliśmy *Tag* przycisku.

W obszarze funkcji będziemy tworzyli kod obsługi przycisku, czyli jakie akcje mają być wykonane po kliknięciu.

18.4. Pierwsza aplikacja *GUI*

Wykonamy projekt prostej aplikację, która w reakcji na kliknięcie przycisku rysuje wykres. Projektujemy zawartość okna:

- wstawiamy do formularza przycisk *Push Button* oraz komponent *Axes:* element do prezentowania wykresu graficznego,
- nadajemy w inspektorze dla przycisku *Push Button*:

właściwość *String* – wpisujemy: *OK*,

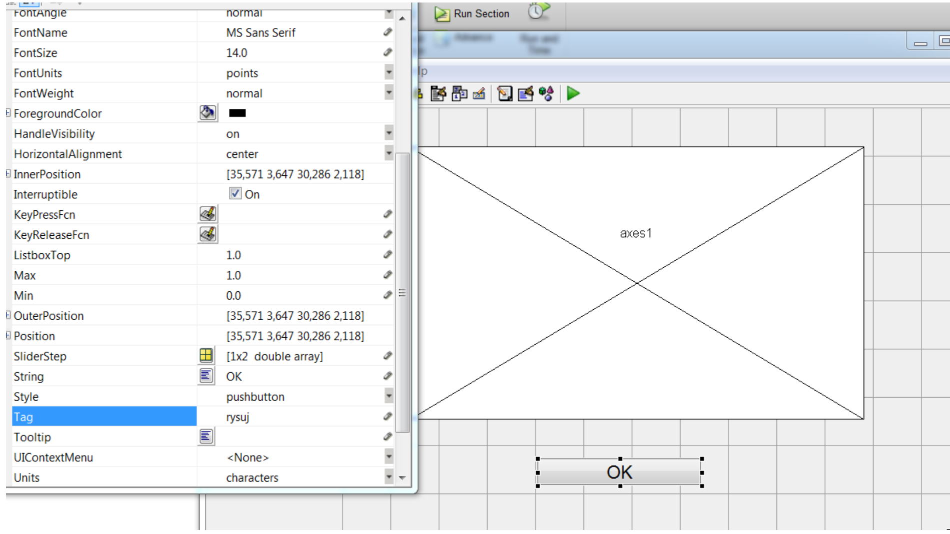
właściwość *Tag* – wpisujemy: *rysuj*,

właściwość *FontSize* – wpisujemy: 14,

właściwość *Position* (rozwijana przyciskiem +) posiada składowe:

- *x*, *y*: współrzędne lewego, dolnego narożnika,
- *width*, *height*: szerokość i wysokość.

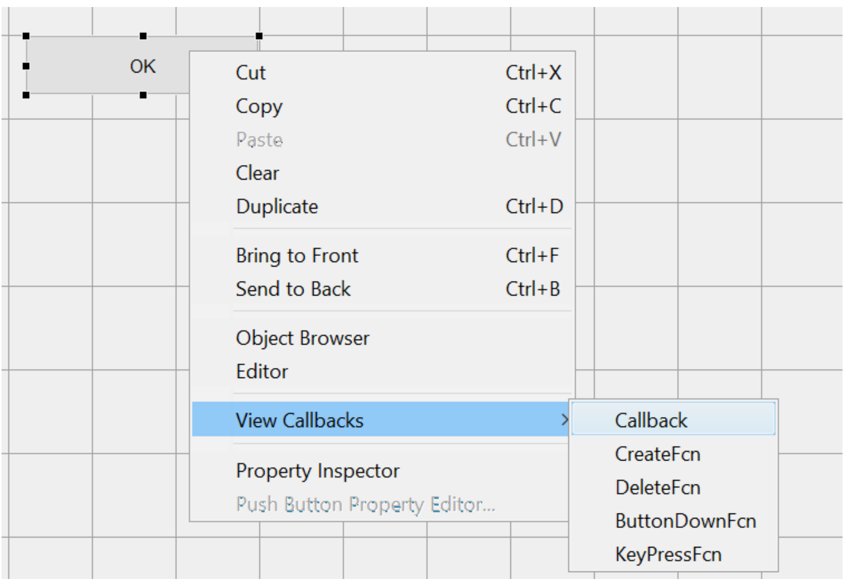
Zapisujemy projekt. Otrzymujemy okno aplikacji jak na rys.18.4.



Rys.18.4. Projekt prostego okna aplikacji oraz *Property Inspector*

Zapisujemy projekt w plikach *wykres.fig* i *wykres.m*.

Dodajmy teraz funkcjonalność przyciskowi. Z menu kontekstowego przycisku *Push Button* o nazwie (*Tag-u*): *rysuj,* z tekstem (*String*): *OK*, wybieramy *View Callbacks/Callback* (rys.18.5).



Rys.18.5. Przeniesienie widoku do funkcji *Callback* w kodzie *M-pliku.*

Następuje przekierowanie do funkcji *rysuj_Callback* w *M-pliku* aplikacji. Funkcja *Callback* będzie wykonywana po kliknięciu tego przycisku w uruchomionej aplikacji.

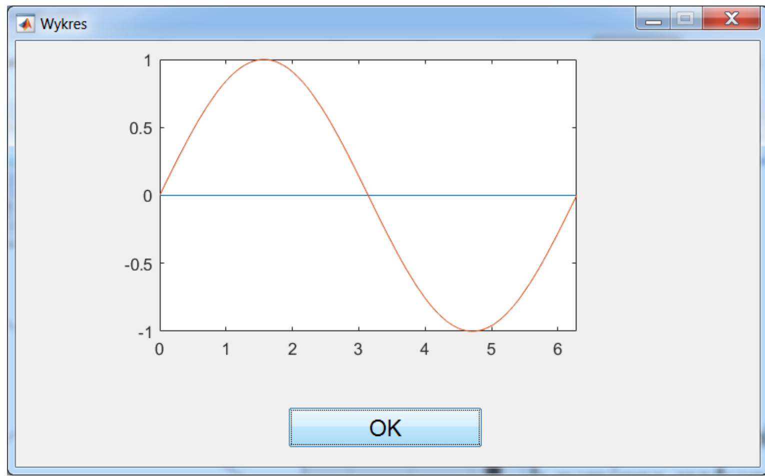
Poniżej nagłówka funkcji *Callback* przycisku *Push Button* o nadanej wartości *Tag*: *rysuj*, dopisujemy instrukcje rysowania wykresu:

```
function rysuj_Callback(hObject, eventdata, handles)
  x = 0: 0.1 :2*pi;
  y = sin(x);
  plot(x, y, x, 0*x)
```

Działanie programu jest inicjowane sposobami:

- przyciskiem *Run* w oknie *M-pliku*,
- przyciskiem *Run* w oknie formatki aplikacji,
- *F5* na klawiaturze, z aktywnego okna *Editor*.

W oknie działającego programu klikamy przycisk, którego funkcja *Callback* powinna wykonać wykres sinusoidy wraz osią ***x***. Tytuł w nagłówku okienka jest zgodny z nazwą pliku (rys.18.6).



Rys.18.6. Okno uruchomionej aplikacji.

18.5. Dostęp do właściwości komponentów w kodzie aplikacji

Po uruchomieniu aplikacji wszystkie komponenty umieszczane są w zmiennej (strukturze) o nazwie ***handles****.* Dostęp do obiektów składowych struktury *handles* odbywa się przy pomocy ich identyfikatora (*Tag*) w sposób kwalifikowany (kropkowy), według zasady:

handles.*tag_obiektu*

a dostęp do wybranej właściwości tego obiektu:

handles.*tag_obiektu.nazwa_właściwości*

Jeżeli w kodzie funkcji *Callback* przycisku *Push Button* umieścimy instrukcję:

handles

a następnie po uruchomieniu aplikacji klikniemy przycisk, w oknie *Command Window* pojawi się informacja o składzie struktury *handles*, czyli lista komponentów naszego projektu.

Przykładowy rezultat:

```
handles =
  struct with fields:
       figure1: [1×1 Figure]
       pushbutton1: [1×1 UIControl]
       axes1: [1×1 Axes]
       output: [1×1 Figure]
```

Jeżeli mamy zamiar ustalić w kodzie funkcji *Callback* wartość wybranej właściwości komponentu (nawet nie będącego "właścicielem" tej funkcji *Callback*), robimy odpowiednie przypisanie w funkcji *Callback*, jak w poniższych przykładach:

```
handles.edit1.String = 'Politechnika';                  % ustal  tekst w edit1
handles.slider1.Value = 10 ;                            % ustal pozycję suwaka slider1
handles.edit1.String = handles.edit2.String ;           % przepisz  tekst z edit2 do edit1
```

Należy tu uważać na wielkość liter w nazwach komponentów (*Tag*) i właściwości oraz na typ danych właściwości (liczby, teksty, czasem tablice). Przykładowo, właściwość *Color* to wektor trzech liczb nasycenia barw *RGB* (*red*, *green*, *blue*), każda liczba w przedziale [0, 1]:

```
handles.figure1.Color = [1  0  0]
```

co spowoduje nadanie koloru czerwonego dla tła okienka aplikacji (*Tag* dla okna aplikacji to domyślnie *figure1*, chyba że to zmienimy w inspektorze).

Zamiar pobrania (do zmiennej) lub zmiany właściwości dowolnego komponentu może być realizowany z wykorzystaniem dwóch funkcji:

get – pobranie właściwości do zmiennej:

zmienna = **get** (handles.*tag_obiektu*, '*nazwa_właściwości* ')

set – nadanie wartości wybranej właściwości:

set (handles.*tag_obiektu*, '*nazwa_właściwości* ', *wartość_właściwości*)

Przykładowo, pobranie wartości łańcucha znaków *String* z komponentu *edit1* do zmiennej ***x***:

```
x = get(handles.edit1, 'String')
```

a ustalenie wielkości czcionki dla komponentu *edit2*:

```
set(handles.edit2,'FontSize',14)
```

Pamiętajmy, że nazwy właściwości piszemy w apostrofach.

Wykorzystanie funkcji ***set*** i ***get*** jest z reguły zamienne z bezpośrednim dostępem do składowych struktury *handles*, jednak niekiedy, w przypadku właściwości prywatnych (o czym wspomniano w rozdziale 18.3) dostęp do nich jest możliwy jedynie przy pomocy odpowiednich metod.

18.6. Umieszczanie własnych zmiennych w strukturze *handles*

Często mamy potrzebę utworzenia pomocniczych zmiennych w naszej aplikacji. W ramach dowolnej funkcji *Callback* zmienne mają charakter lokalny, czyli po zakończeniu działania tej funkcji przestają istnieć. Jeżeli mamy zamiar przekazywać między funkcjami dodatkowe dane różnych typów, powinniśmy naszą zmienną dołączyć do struktury *handles* i koniecznie zaktualizować strukturę funkcją *guidata( )* przy pomocy funkcji guidata:

guidata(hObject, handles)

Przykładowy kod funkcji *Callback* akcji przycisku *Push Button* może zawierać instrukcje:

```
zmienna = 20+10i;               %przykładowa zmienna z wartością zespoloną
handles.dana = zmienna;         %przypisanie zmiennej do pola handles.dana
guidata(hObject, handles);      %aktualizacja struktury handles
```

Teraz można w funkcji *Callback* innego przycisku korzystać z pola *dana* struktury *handles* w formie:

```
handles.dana
```

18.7. Przykłady aplikacji *GUI*

Poniżej zamieszczono kilka przykładów prostych aplikacji z interfejsem graficznym.

Przykład 1

W nowej aplikacji wstawiamy komponent *Edit Text*. Znajdujemy w kodzie *M-pliku* funkcję o nazwie *naszplik*_OpeningFcn.

Funkcja Opening_Fcn jest zawsze wykonywana automatycznie po uruchomieniu aplikacji. Poniżej istniejącej instrukcji:

```
guidata(hObject, handles);
```

wpisujemy:

```
set(handles.edit1, 'String', 'Tekst po starcie');
```

Instrukcja powoduje nadanie (*set*) ustalonej wartości tekstowej właściwości *String* elementu *Edit1*.

Wstawiamy przycisk *Push Button.* W kodzie jego funkcji *Callback* wpisujemy:

```
set(handles.edit1, 'String', 'Tekst po kliknięciu');
```

Zapisujemy projekt i sprawdzamy działanie aplikacji.

Przykład 2

Utworzymy aplikację z dwoma przyciskami *Push Button* i dwoma komponentami *Axes*. Kliknięcie przycisków powoduje tworzenie wykresów funkcji *sin(x)* i *cos(x)* we wspólnym układzie współrzędnych.

Uwaga: W przypadku umieszczenia w oknie kilku komponentów *Axes*, wybór w którym komponencie *Axes* ma być rysowany wykres uzyskujemy wykonując instrukcję:

```
axes(handles.tag_odpowiedniego_axes)
```

przed instrukcją tworzącą wykres.

Wykonajmy aplikację, w której odbędzie się sterowanie parametrem wykresu przy pomocy suwaka. Wstawiamy w okienku aplikacji suwak (*Slider*) i ustalamy w *Inspektorze* jego właściwości:

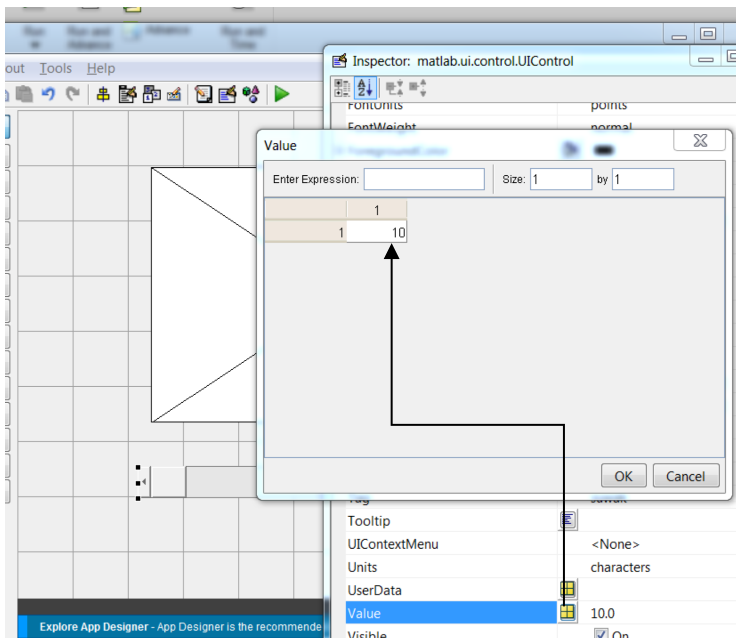
Tag : suwak

Min:10

Max:50

a także, w osobnym okienku, ustalamy wartość początkową suwaka (rys.18.7):

Value: 10



Rys.18.7. Ustawienie początkowej wartości suwaka

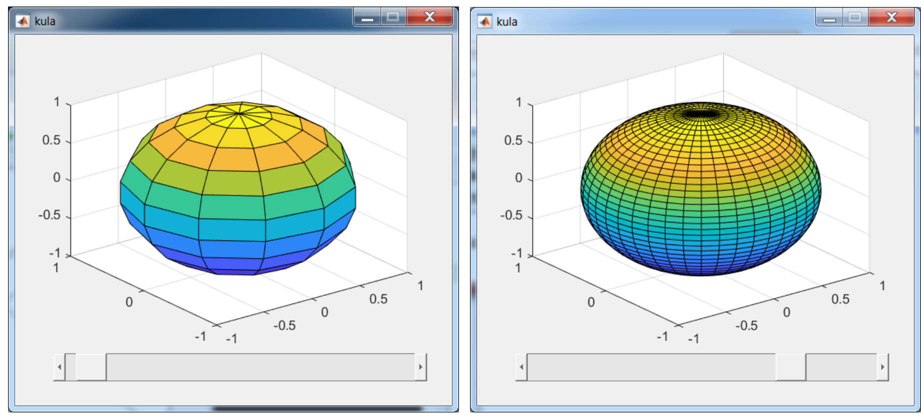
Pamiętajmy, żeby zapisywać projekt okna (plik *fig*) po każdej zmianie projektu lub właściwości komponentu.

Dodajemy w kodzie funkcji *Callback* suwaka (akcja będzie wykonywana przy jego przesuwaniu):

```
function suwak_Callback(hObject, eventdata, handles)
  pozycja = hObject.Value; %pobranie Value suwaka do zmiennej pozycja
                           %   hObject to obiekt tej funkcji Callback, czyli suwak
                           %   zamiast hObject można pisać handles.suwak
  z = round(pozycja);  %zaokrąglenie do liczby całkowitej
  sphere(z);           %rysowanie wykresu kuli z argumentem
```

Funkcja *sphere* rysuje wykres kuli, posiada argument określający dokładność odwzorowania jej powierzchni.

Przykładowe wykresy po uruchomieniu aplikacji i przesuwaniu suwaka pokazuje rys.18.8.



Rys.18.8. Okienko aplikacji sterowania parametrem funkcji przy pomocy suwaka

Po pobraniu pozycji suwaka i zaokrągleniu zmiennej ***z*** do liczby całkowitej, można też wykorzystać parametr do utworzenia wykresu przy pomocy funkcji symbolicznej *ezplot*. Można do tego celu wykorzystać drugi suwak o kodzie *Callback*:

```
z = hObject.Value;
syms x
%rysowanie symbolicznej funkcji sin(ax) z różnymi częstotliwościami
ezplot(subs(sin(z*x)),[0 pi]);
```

Pozycja suwaka, będąca jego właściwością *Value* zostaje podstawiona (*subs*) do funkcji symbolicznej. Tak utworzona aplikacja, po zapisaniu i uruchomieniu, po przesunięciu suwaka ustali (zgodnie z wartością pozycji suwaka) wartość parametru (częstotliwości funkcji *sinus*) wykresu dla funkcji.

W ramach funkcji *plik_OpeningFcn* można dopisać w projekcie rysowania wykresu kuli:

```
sphere(10);
```

aby zaraz po uruchomieniu pojawiał się żądany wykres kuli z założoną początkową wartością parametru.

Przykład 3

Do poprzedniego przykładu dodamy funkcjonalność, aby po przesunięciu suwaka wartość jego pozycji była wyświetlana w komponencie *Edit Text*.

Dodajemy do aplikacji komponent *Edit Text* (*Tag*: *edit1*). Ustalamy jego właściwości manualnie (położenie, rozmiar) oraz w oknie *Inspektora.* Zapisujemy projekt.

W kodzie *Callback* suwaka dopisujemy po uprzednich instrukcjach:

```
% wypisywanie wartości suwaka w edit1
handles.edit1.String = num2str(pozycja);
   % albo zamiennie
   % handles.edit1.String = num2str(handles.suwak.Value);
```

Uwaga: zastosowano tu funkcję *num2str* wykonującą **konwersję typu danych** z liczbowego na tekstowy, bo ten typ jest wymagany dla właściwości *String*:

zmienna_tekstowa = **num2str**(*liczba*) - konwersja liczby na tekst

Konwersja odwrotna, tekstu na liczbę wykonywana jest funkcją *str2double*:

zmienna_liczbowa = **str2double**(*tekst*) - konwersja tekstu na liczbę

Funkcja ta będzie użyteczna do pobierania danych tekstowych, reprezentujących liczby, wpisane znakowo do komponentów *Edit Text*.

Przykład 4

Celem aplikacji będzie narysowanie wykresu funkcji *sin(x)* w reakcji na kliknięcie przycisku. Tym razem pozycja suwaka ma określić wartość końcową przedziału kąta dla funkcji, czyli od 0 do wartości ustalanej pozycją suwaka.

Umieszczamy w oknie przycisk *Push Button* i zmieniamy *String* dla przycisku na *Kliknij* i *Tag* na *rysuj*. Ustalamy dla suwaka odpowiednie wartości właściwości *Min* i Max oraz wartość początkową *Value*.

Wpisujemy w kodzie funkcji *rysuj_Callback*:

```
function rysuj_Callback(hObject, eventdata, handles)
z = get(handles.suwak,'Value');     %pobranie wartości Value suwaka
fplot('sin(x)',[0,  pi*round(z)]);      %rysowanie wykresu z ustalonym argumentem
```

Przykład 5. Obliczenia arytmetyczne

Wykonamy prostą aplikację dla obliczania sumy dwóch liczb dziesiętnych. Umieszczamy na formularzu trzy komponenty *Edit Text* i jeden przycisk *Push Button.*

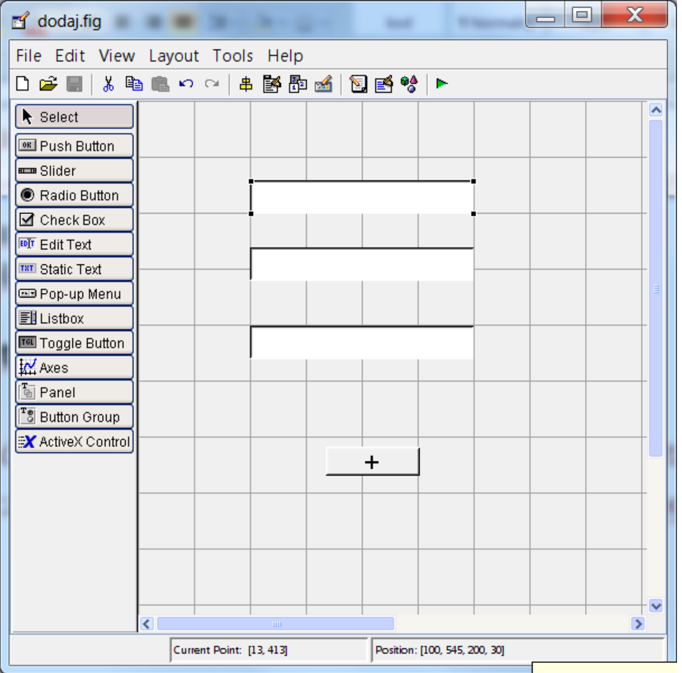
Korzystając z *Inspektora* – polom edycyjnym zmieniamy właściwości:

- *Tag-i*: *e1*, *e2* i *e3*,
- usuwamy domyślne teksty we właściwościach *String*.

Dla przycisku ustalamy właściwości:

- *String* zmieniamy na znak: '+' , reprezentujący sumowanie,
- *Tag* zmieniamy na: *dodaj.*

Okienko projektu aplikacji przedstawia rys.18.9.



Rys.18.9. Okno projektu aplikacji dla obliczenia sumy dwóch liczb.

Zapisujemy projekt. W kodzie funkcji *Callback* przycisku *dodaj* piszemy:

```
function dodaj_Callback(hObject, eventdata, handles)
  s1 = str2double(get(handles.e1,'String'));  %pobierz napis z e1 i konwertuj na liczbę
  s2 = str2double(get(handles.e2,'String'));  %pobierz napis z e2 i konwertuj na liczbę
  suma = s1+s2;                               %dodawanie arytmetyczne
  set(handles.e3,'String',num2str(suma));    %konwertuj sumę na tekst i wyślij do e3
  guidata(hObject, handles);
```

Funkcja *Callback* pobiera teksty z *edit1* i *edit2*, konwertuje je na typ liczbowy, wykonuje dodawanie, rezultat (po konwersji na typ tekstowy) wysyła do *edit3*.

Uruchamiamy aplikację i sprawdzamy jej działanie. Trzeba uważać, aby poprawnie wpisywać tekst reprezentujący sumowane liczby, by funkcja konwersji mogła je poprawnie zinterpretować jako dane typu liczbowego.

Do uprzednio utworzonej aplikacji dodajemy dwa nowe przyciski, których zadaniem będzie:

- skasowanie treści wszystkich pól edycyjnych – przypisanie im pustych ciągów znaków: ' ' (dwa apostrofy).
- wykonanie operacji odejmowania dwóch liczb,

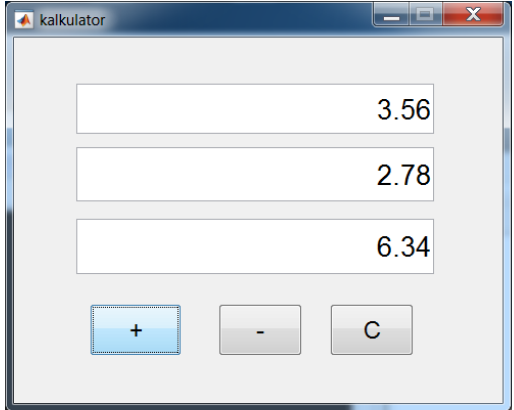
Funkcja *Callback* przycisku kasowania pola *Edit Text* powinna zawierać kod:

```
handles.e1.String = ''
```

Dwa apostrofy ' ' – to pusty ciąg znaków.

Dodanie kodu w funkcji *Callback* innych operacji (odejmowania, mnożenia, dzielenia) dla dodatkowych przycisków jest już proste.

Okienko naszej aplikacji komponentowej:



Rys.18.10. Aplikacja prostego kalkulatora

Typ wyrównania tekstów (*left, center, right*) w komponentach *Edit Text* określa właściwość *HorizontalAlignment*.

Dla ułatwienia korzystania z aplikacji możemy umieścić obok komponentów etykiety z tekstami opisującymi ich cel (*Static Text* z odpowiednim tekstem we właściwości *String*).

Rozbudowa kalkulatora może polegać na dodaniu przycisków do obliczania wartości wybranych funkcji matematycznych dla podawanego w komponencie edycyjnym argumentu.

Przykład 6. Operacje symboliczne w *GUI*

Istnieje możliwość pobrania zapisu funkcji symbolicznej z komponentu *Edit Text*, wymagana jest jednak konwersja danych **typu tekstowego na zapis symboliczny**. Wykonuje to funkcja *str2sym*:

zmienna_symboliczna = **str2sym**(*zmienna_tekstowa*)

Po obliczeniach symbolicznych konwertujemy postać wyrażenia **typu symbolicznego na tekstowy** przy pomocy funkcji *char*:

wektor_znaków = **char**(*zmienna_ symboliczna*)

Wykonajmy aplikację *GUI* z dwoma kontrolkami *Edit Text* – jedną do tworzenia zapisu funkcji symbolicznej, drugą do wyświetlenia obliczonej pochodnej tej funkcji - oraz z przyciskiem *Push Button*.

Funkcja *Callback* dla przycisku otrzymuje treść:

```
syms x
tekst = get(handles.edit1,'String'); %pobranie tekstu z pola edit1
funkcja = str2sym(tekst);            %konwersja tekstu do typu symbolicznego
df = diff( funkcja );                %obliczenie pochodnej
wynik = char(df);                    %konwersja wyniku do wektora znaków
set(handles.edit2, 'String', wynik); %wysłanie wyniku do edit2
```

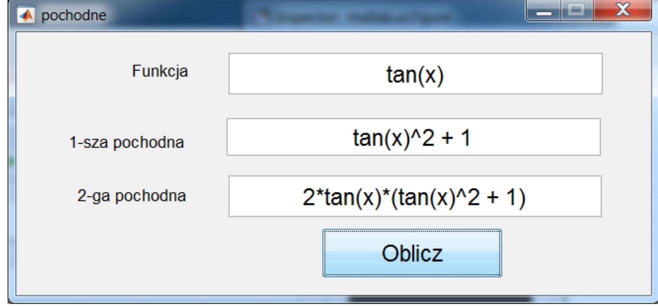
Powstanie prosta aplikacja okienkowa, pozwalająca na wygodne testowanie pochodnych pierwszego rzędu dla dowolnych funkcji.

Znając powyższą zasadę łatwo można rozbudować program o:

- obliczanie pochodnych wyższych rzędów,
- wyznaczanie całek nieoznaczonych,
- rysowanie wykresów funkcji symbolicznych,

tworząc dodatkowe okienka edycyjne i dopisując odpowiednie instrukcje.

Przykładowo aplikacja może wyglądać jak na rys.18.11. Dodano tu element edycyjny do prezentacji pochodnej funkcji drugiego rzędu, utworzenie kodu pozostawiono czytelnikowi do samodzielnego wykonania według opisanych zasad.



Rys.18.11. Okienko aplikacji *GUI* dla wyznaczania pochodnych funkcji

19. Przykład prostej aplikacji bazy danych

Aplikacja zilustruje:

- obsługę wektora struktur,
- tworzenie i zarządzanie kilkoma utworzonymi funkcjami,
- zapis i odczyt z pliku.

Dane gromadzimy w wektorze o nazwie *studenci*, którego elementy będą strukturami o nazwie *student*, o następujących składowych:

- *Lp* - liczba naturalna, liczba porządkowa studenta,
- *nazwisko*- dana tekstowa, nazwisko studenta,
- *numer_indeksu*- dana liczbowa, reprezentująca numer indeksu studenta.

Celem aplikacji będzie wyświetlanie listy studentów, dopisywanie nowego studenta i usuwanie wybranej pozycji.

Na wstępie tworzymy cztery pliki – definicje naszych funkcji.

Funkcja ***menu.m***, wyświetlająca menu dostępnych operacji, argumentem wyjściowym jest numer operacji wybranej przez użytkownika.

Treść funkcji *menu.m*:

```
function [x] = menu()
clc
disp ('Wybierz opcję:');
disp('1 – dopisz,  2 – wypisz, 3 – usuń,  4 – KONIEC'  )
x = input('Co wybierasz?:');
```

Funkcja ***czytaj.m***, listująca zawartość pliku czyli listę wszystkich studentów:

```
function [ ] = czytaj(plik)
load(plik);
disp('Oto cała lista');
if length(studenci)==0
   disp('PUSTA LISTA');
else
  for   k = 1:length(studenci)
   fprintf('Lp %d Nazwisko: %s Indeks: %d\n', ...
         studenci(k).Lp, studenci(k).nazwisko, studenci(k).nr_indeksu);
  end
end
```

Funkcja ***usun.m***, realizująca usuwanie wybranej struktury (rekordu) z wektora struktur:

```
function [ ] = usun(plik)
disp('USUWANIE');
czytaj(plik);
numer = input('Podaj liczbę porządkową studenta do usunięcia:');
load(plik);

%trzeba sprawdzić czy jest taka liczba porządkowa
%utworzymy wektor z zerami i jedynką w pozycji znalezionej
indeksy = [studenci.Lp]==numer;
n = find(indeksy==1);            %szukaj indeksu jedynki – jeżeli nie ma, to pusty wektor
if   ~isempty(n)                  %jeżeli niepusty wektor
   for  k = n:length(studenci)-1
       studenci(k) = studenci(k+1);  %przepisz następny element
       studenci(k).Lp = k;           %nadaj nowy numer
   end;
   studenci(length(studenci)) = [ ];  %usuń ostatni  (pusty element)
   save(plik,'studenci');            %zapisz do pliku
   clc                               %wyczyść ekran
   czytaj(plik);                     %wypisz nową listę studentów
else
   disp('Nie ma takiego studenta');
end
```

Funkcja ***dopisz.m***, realizująca dopisywanie nowego rekordu, danych dla danych nowego studenta:

```
function [ ] = dopisz(plik,student)
disp('Dopisywanie rekordu');
a = input('Podaj nazwisko:','s');
b = input('Podaj numer:');
load(plik);                                %otwórz plik
student.Lp = length(studenci)+1;           %nowy numer porządkowy
student.nazwisko = a;
student.nr_indeksu = b;
studenci(length(studenci)+1) = student;   %dopisz studenta na końcu wektora
save(plik,'studenci');
clc
czytaj(plik)
```

Teraz tworzymy nasz plik wykonawczy ***baza.m***, sterujący wykonywaniem poszczególnych funkcji:

```
clc,clear
plik = 'grupa.mat';
%jeżeli nie ma pliku – tworzymy plik 'grupa.mat' z dwiema strukturami
if isfile(plik)
     disp('Jest plik bazy')
else
    student = struct('Lp',1,'nazwisko','Nowak','nr_indeksu',12345);
    studenci(1) = student;       %wpisanie do wektora
    %  i  druga struktura
    student = struct('Lp',2,'nazwisko','Kowalski','nr_indeksu',12346);
    studenci(2) = student;       %wpisanie do drugiej komórki wektora
    save(plik,'studenci');
    disp('Utworzono w nowym pliku 2 testowe rekordy')
end
opcja = menu( );                 %wyświetl menu operacji
while opcja<4                    %pętla – dopóki wybieramy działania
   clc
   switch opcja                  %wykonaj wybraną opcję
     case 1
         dopisz(plik);
     case 2
         czytaj(plik);
     case 3
         usun(plik);
   end
   disp('Dowolny klawisz – MENU');
   pause
   clc
   opcja = menu( );
end
disp('KONIEC');
```

W pliku wykonawczym sprawdzamy czy istnieje plik, w przypadku jego braku tworzony jest wektor dwóch przykładowych struktur.

Zwróćmy uwagę na fakt, że funkcje mogą wywoływać inne funkcje, na przykład funkcje *dopisz i usun* uruchamiają funkcję *czytaj*.

Trudniejszy problem stanowi sortowanie wektora struktur według wybranego pola. Oto metoda posortowania wektora struktur alfabetycznie według nazwisk:

```
clc,clear

%Tworzymy trzy przykładowe struktury w tablicy studenci
student = struct('Lp',1,'nazwisko','Cabacki','nr_indeksu',11111);
studenci(1) = student;   %wpisz do wektora w pozycji 1
student = struct('Lp',2,'nazwisko','Babacki','nr_indeksu',33333);
studenci(2) = student;   %wpisz do wektora w pozycji 2
student = struct('Lp',3,'nazwisko','Abacki','nr_indeksu',77777);
studenci(3) = student;    %wpisz do wektora w pozycji 3

%wypisanie zawartości wektora
for  k = 1:length(studenci)
      fprintf('Lp %d Nazwisko: %s Indeks: %d\n', ...
         studenci(k).Lp,  studenci(k).nazwisko,  studenci(k).nr_indeksu);
end

%Struktura samych nazwisk
nazwiska = {studenci.nazwisko};

%uzyskanie nowych indeksów po posortowaniu
[A,indeksy] = sort(nazwiska);

%kolejność w tablicy studenci według nowych indeksów
studenci = studenci(indeksy);

%wypisanie po posortowaniu
disp('----------------')
for   k = 1:length(studenci)
    studenci(k).Lp = k; %zmiana numeracji
    fprintf('Lp %d Nazwisko: %s Indeks: %d\n', ...
          studenci(k).Lp, studenci(k).nazwisko, studenci(k).nr_indeksu);
end
```

Oto rezultat powyższego skryptu:

```
Lp 1 Nazwisko: Cabacki Indeks: 11111
Lp 2 Nazwisko: Babacki Indeks: 33333
Lp 3 Nazwisko: Abacki Indeks: 77777
------------------------------------
Lp 1 Nazwisko: Abacki Indeks: 77777
Lp 2 Nazwisko: Babacki Indeks: 33333
Lp 3 Nazwisko: Cabacki Indeks: 11111
```

Cele poszczególnych operacji wyjaśniają komentarze w treści skryptu.

Powyższy przykład aplikacji bazodanowej można rozbudować o inne funkcjonalności, na przykład:

- poszerzyć skład struktury *student* o nowe pola, np. datę urodzenia (macierz o trzech elementach):

```
student = struct('Lp',1,'nazwisko','Nowak','nr_indeksu',12345,'data_ur',[1,10,2002])
```

- dodać funkcję wyszukiwania studenta, według wybranego klucza, np. według numeru indeksu
- wprowadzić sortowanie wektora studenci według numerów indeksu

i inne funkcjonalności.

Istnieje również możliwość opracowania podobnej aplikacji stosując typ danych *table*, poznany w rozdz.7.4 .

20. Zadania

Wyrażenia obliczeniowe

1. Obliczyć wartość wyrażenia:

$$\frac{(cos\,3x-4\,sin^2x)\sqrt[4]{x-1}}{|e^{-3x}\,ln\,3x-3|}$$ dla x = $\pi/2$.

2. Obliczyć wartość wyrażenia:

$$\frac{|6\,sin\,6x-5|}{\sqrt{y^3+3}}$$ dla x = 15^0 oraz y = 2

3. Wykonać *M-plik*, w którym użytkownik podaje wartość promienia okręgu. Wyprowadzić wzory na pole kwadratu opisanego na tym okręgu i pole kwadratu wpisanego w okrąg. Udowodnić, że stosunek tych pól wynosi 2, dla kilku podawanych przez użytkownika promieni okręgu.

4. Stożek ma promień podstawy R = 4.5 i wysokość H = 34.6. Utworzyć ciąg instrukcji wprowadzania danych, obliczenia i formatowanego wypisania wartości objętości i pola powierzchni tego stożka.

5. Przy pomocy skryptu Matlaba obliczyć dla wyprowadzonych wzorów na pole powierzchni trójkąta równobocznego wpisanaego w okrąg o promieniu R i trójkata równobocznego opisanego na tym okręgu (w obu przypadkach środek okręgu dzieli wysokość trójkąta w stosunku 1:2). Sprawdzić stosunek tych pól dla kilku promieni okręgu.

Operacje macierzowe i tablicowe

1. Napisać *M-plik*, w którym przy pomocy operacji macierzowych rozwiązywany jest układ równań:

$$-3x + 5{,}2y + 3\,z - 12v = 0$$

$$2x - 4\,y + 3{,}3\,z - 11v = 2$$

$$2y + 2x - 2z - 12{,}1v = -3$$

$$-x + y + 3z = -1$$

Sprawdzić rozwiązania w jednym z równań.

2. Wyznaczyć pierwiastki równania:

$$x^5 - 4x^4 + 14x^2 - 5 = 0$$

Następnie sporządzić wykres funkcji ilustrujący obliczone miejsca zerowe (porównać na wykresie współrzędne punktów przecięcia z osią x z obliczonymi).

3. Wykonać *M-plik*, w którym:
   - wypełniana jest macierz:

$$\begin{bmatrix} -14.2 & -13 & 1.2 & -3 \\ -2.5 & \sqrt{5} & -1 & 2{,}61 \\ -3 & -0.3 & \pi & 4.23 \\ 1.22 & -3 & 4 & 6 \end{bmatrix}$$

   - obliczany jest wyznacznik macierzy i macierz odwrotna,
   - obliczana jest suma elementów dodatnich macierzy,
   - obliczana jest suma elementów dwóch pierwszych kolumn (z zastosowaniem pętli),
   - wyzerowane są elementy ostatniej kolumny,
   - do nowej macierzy przepisane są elementy dwóch ostatnich wierszy.
4. Wykonać *M-plik*, w którym:
   - wypełniane są macierze:

$$\begin{bmatrix} -4.2 & -\frac{1}{7} & \frac{1}{\sqrt[3]{1{,}5}} & -3 \\ -2.5 & \log_{10}3.5 & 2 & 2{,}61 \\ -13 & 0.3 & 2\pi & \sqrt{3} \\ 1.3 & -3 & 4 & 6 \end{bmatrix} \begin{bmatrix} 4.5 & -\frac{1}{7} & e^{-\frac{1}{\pi}} & \frac{1}{13} \\ 0 & 18 & -1 & 2 \\ -3.1 & \sin 0.2\pi & 7 & -3 \\ -1{,}1 & 3 & 4 & -6 \end{bmatrix}$$

   - obliczane są iloczyny obydwu macierzy – macierzowy i tablicowy,
   - obliczany jest iloczyn macierzowy pierwszej macierzy przez transponowaną drugą macierz,
   - obliczana jest suma obydwu macierzy,
   - wszystkie macierze będą zapisane w pliku tekstowym.
5. Napisać *M-plik*, którego zadaniem będzie:
   - zdefiniowanie macierzy: $\begin{bmatrix} -1.1 & -3 & 2 & -3 \\ 0 & \text{ctg}35^{o} & 1 & \sqrt[3]{2} \\ 3.1 & 16 & 4 & -11 \\ -1.3 & 3 & 4{,}5 & 2 \end{bmatrix}$
   - wypisanie na ekranie wszystkich elementów dodatnich (zastosować zagnieżdżane pętle *for* oraz instrukcję warunkową).
6. Napisać *M-plik*, którego zadaniem będzie:
   - zdefiniowanie macierzy: $\begin{bmatrix} -4.2 & -1.3 & 12 & -3 \\ -2.5 & 1.8 & -1 & 2{,}61 \\ -13 & 0.3 & \pi & -\sqrt{11} \\ 1.3 & -3 & 4 & -1.23 \end{bmatrix}$
   - obliczenie sumy elementów na przekątnej głównej.
   - wypisanie na ekranie wszystkich elementów należących do przedziału (-3, 3) i ich sumy.
   - policzenie, ile elementów należy do przedziału (-3, 3),
   - utworzenie wektora zawierającego indeksy elementów dodatnich.

7. Napisać *M-plik*, w którym generowana jest macierz o rozmiarach 3×4 o elementach dziesiętnych należących do przedziału (-100, 100).
8. Napisać *M-plik*, w którym generowany jest wektor o rozmiarze 100, o elementach całkowitych należących do przedziału (100, 200). Obliczyć ile elementów tego wektora się powtarza (wykorzystać funkcję *unique*).
9. W *M-pliku* wygenerować macierz o rozmiarze 3x4 z losowymi wartościami liczb dziesiętnych z przedziału 0 do 20. Wykorzystując pętlę *for* policzyć ile elementów tej macierzy zawiera się w przedziale (*A*, *B*), gdzie wartości *A* i *B* podaje użytkownik interakcyjnie w trakcie wykonywania *M-pliku*.

   Dodatkowo: Zabezpieczyć program na wypadek, gdy podawane wartości *A* lub *B* będą poza przedziałem losowania.
10. Wygenerować losowo macierz o rozmiarach 5x5. Przepisać z tej macierzy do nowej macierzy elementy drugiej i trzeciej kolumny oraz obliczyć ich sumę.
11. Umieścić w wektorze (np. o rozmiarze 100) kolejne wyrazy ciągu danego zależnością:

$$b_n = (1 + \frac{1}{n})^n$$

   Przy pomocy pętli *for* znaleźć sumę kolejnych wyrazów od 30-go do 50-go.
12. Umieścić w 100-elementowym wektorze kolejne liczby ciągu *Fibonacciego* o następującej zasadzie: pierwsze dwa elementy to 0 i 1, każdy następny element od 3-go jest sumą dwóch poprzednich. Obliczyć sumę wyrazów od 20-go do 30-go.

Wykresy

1. Wykonać wykres funkcji:

$$y = e^{2\sqrt{x}} + sin20x$$

   dla x ∈ (0, 1).
2. Utworzyć wykres dwóch krzywych:

$$y = 2cos^2 2x$$

$$z = sin\, x$$

   w jednym układzie współrzędnych, w przedziale x∈(0, 4π)
3. Matematyka twierdzi, że zachodzi tożsamość:

$$sin\, 4x = 8\, cos^3 x\, sin\, x - 4\, cos\, x\, sin\, x$$

   Sporządzić wykresy w przedziale (0, 4π), które prezentują tożsamość obydwu funkcji (lewej i prawej strony równania) w wybranym przedziale zmienności kąta *x* (wykresy powinny się pokrywać).

4. Matematyka twierdzi, że zachodzi tożsamość:

$$\frac{1 + cos\,x}{sin\,x} = \frac{sin\,x}{1 - cos\,x}$$

Sporządzić wykresy, które prezentują tożsamość obydwu funkcji (lewej i prawej strony równania) w przedziale (0, 360°).

5. Dwie proste o równaniach:

$$y = 3x+5 \quad \text{i} \quad y = -2x+7$$

przecinają się. Wyznaczyć z wykresu punkt przecięcia tych krzywych.

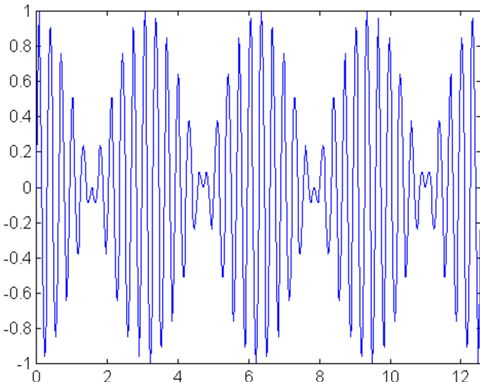
6. Napisać *M-plik*, którego zadaniem będzie wykonanie wykresu funkcji:

$$f(x) = |1 - ln\,x|\frac{sin^2 x}{x} \quad \text{dla } x \in (0\,,10)$$

7. Wykonać wykres krzywej o równaniu:

$$f(x) = cos(kx)sin(x)$$

dla x∈(0, 4π). Metodą prób i błędów tak dobrać wartość parametru *k*, aby otrzymać wykres jak najbliższy poniższemu:



8. Dana jest funkcja dwóch zmiennych:

$$f(x,y) = \left|(x - y)\,e^{-x^4 - y^4}\right|$$

Narysować wykres powierzchniowy tej funkcji, wykorzystując funkcję *surf.*

9. Dana jest funkcja dwóch zmiennych:

$$f(x,y) = (x^2 - y^2)^2\,e^{x^2 - y^2}$$

Narysować wykres powierzchniowy tej funkcji wykorzystując funkcję *mesh.*

10. Powierzchnia opisana równaniem:

$$f(x,y) = x^2 - y^2$$

to tak zwana powierzchnia "siodłowa". Udowodnić jej kształt na wykresie trójwymiarowym (*surf* lub *mesh*).

11. W układzie współrzędnych kartezjańskich narysować trójkąt o współrzędnych narożników:

(1, 1), (3, 1) i (2, 4).

Korzystając z macierzy transformacji (rozdz.14.6) dokonać obrotu tego trójkąta o kąt 30° przeciwnie do ruchu wskazówek zegara oraz dorysować ten obrócony trójkąt do poprzedniego wykresu.

12. Samochód w chwili t = 0 jedzie z prędkością 100 km/h i rozpoczyna hamowanie z opóźnieniem opisanym funkcją a = $\frac{5}{t+1}$. Obliczyć po ilu sekundach auto się zatrzyma i jaka długa będzie droga hamowania. Utworzyć odpowiednie wykresy prędkości, przyspieszenia i drogi w funkcji czasu.

13. Wykonać symulację (przy wykorzystaniu iteracji *for* z krokiem co 1 km) jazdy samochodem, wykorzystując następujące dane:
    - samochód posiada bak pojemności 50 litrów, pełny na starcie,
    - zużycie paliwa wynosi 8 l/100km,
    - jeżeli w baku jest mniej niż 1 litr benzyny samochód jedzie na stację benzynową i tankuje do pełna,
    - całkowity dystans przejechany to 10 000 km.

    Wykonać wykres stanu benzyny w funkcji przejeżdżanych kilometrów, obliczyć ile będzie procesów tankowania i ile benzyny zostanie w baku po przejechaniu całego dystansu.

14. Rano beczka o pojemności 10 litrów jest pełna wody. W dzień wycieka z beczki 3% aktualnej zawartości, a w nocy przybywa 3% aktualnej zawartości. Wykonać w *MATLAB-ie* symulację zawartości naczynia w czasie, z wykorzystaniem pętli wypełniać macierz gromadzącą kolejne dane oraz sporządzić wykres zawartości beczki w czasie.

15. Wykonać animację rysowania krzywej o równaniu:

$f = -0.2x^2 + x$ w przedziale (0, 5)

Kolejne punkty wykresu pojawiają się co 0,1 sekundy (wykorzystać pętlę i funkcję *pause*).

Dodatkowo: punkt maksimum krzywej ma być oznaczony zielonym kółeczkiem.

16. Przy pomocy pętli *for* narysować wykres 30 małych kwadratów, których pozycje na wykresie będą losowe. Utworzyć wektory dla kwadratu, w pętli losować dwie liczby z wybranego przedziału, w funkcji plot stosować wyrażenia *x+los1* i *y+los2*. Dobrać odpowiednie zakresy osi.

Obliczenia symboliczne

1. Napisać *M-plik*, którego zadaniem będzie:
   - symboliczne obliczenie pochodnej pierwszego rzędu i całki nieoznaczonej dla funkcji:

$$f(x) = \frac{sin^2 x}{1 - ln\, x}$$

   - wykonanie wykresów funkcji f(x) i jej pochodnej.
2. Dana jest funkcja dwóch zmiennych *x* i *y*:

$$\mathrm{f}(xy) = 3\ \mathrm{tg}\ (xy) - \frac{1}{x-y}$$

Obliczyć dla tej funkcji pochodne cząstkowe pierwszego i drugiego rzędu względem każdej zmiennej.

3. Zbadać funkcję wielomianową:

$$y = -x^3 + 12x^2 - 13x - 5$$

Znaleźć dokładne wartości miejsc zerowych, współrzędne minimów i maksimów oraz punktów przegięcia. Zilustrować wyniki na wykresie.

4. Zbadać funkcję:

$$y = -3x^4 + 1.5x^2 - 13x - 5$$

Znaleźć dokładne wartości miejsc zerowych, współrzędne minimów i maksimów oraz punktów przegięcia. Zilustrować wyniki na wykresie.

5. Dla paraboli danej równaniem:

$$\mathrm{y} = -\mathrm{x}^2 + 1$$

znaleźć równanie linii prostej, stycznej do tej paraboli w punkcie o współrzędnej x = –3. Zilustrować rozwiazanie na wykresie.

6. Funkcja momentu dla silnika indukcyjnego jest dana równaniem:

$$M(s) = \frac{2M_{max}}{\frac{s_0}{s} + \frac{s}{s_0}}$$

Narysować w jednym układzie współrzędnych dla $s \in (-5, 5)$ wykresy funkcji *M(s)* dla założonych wartości M_{max} = 1 oraz wariantów parametrów s_0 = 1, 2 i 3. Znaleźć symboliczne rozwiązanie ogólne, określające dla jakiej wartości zmiennej *s* moment ma wartość maksymalną.

7. Znaleźć najbliższe początkowi układu współrzędnych miejsca zerowe funkcji:

$$y = \sin x + \cos 2x$$

Zilustrować funkcję i miejsca zerowe na wykresie. Zbadać ekstrema tej funkcji.

8. Znaleźć długość krzywej paraboli danej równaniem:

$$y = 2x^2 - 3x + 1$$

w przedziale zmiennej $x \in (0, 10)$, jeżeli znamy wzór:

$$d = \int_a^b \sqrt{1 + \left(\frac{dy}{dx}\right)^2}\, dx$$

9. Obliczyć pole powierzchni wspólne dla dwóch półokręgów o równaniach:

$$y_1 = \sqrt{1 - x^2}$$

oraz

$$y_2 = -\sqrt{1 - x^2} + 0.3$$

(Pole powierzchni obliczyć w granicach punktów przecięcia obydwu półokręgów)

10. Dana jest funkcja dwóch zmiennych:

$$f(x, y) = 3\, ctg(x - y) - \frac{e^{-3x}}{x - y}$$

Obliczyć pochodne cząstkowe pierwszego i drugiego rzędu dla tej funkcji.

11. Znaleźć dokładne współrzędne ekstremum funkcji:

$$y = \ln x - x^2$$

Zilustrować funkcję i punkt ekstremum na wykresie.

12. Parabole o równaniach:

$$y_1 = 3x^2 + 2$$

$$y_2 = -3x^2 + 6$$

przecinają się w dwóch punktach. Znaleźć współrzędne punktów wspólnych obydwu krzywych metodą graficzną (odczytać z wykresów) oraz obliczeniową i obliczyć pole powierzchni pomiędzy parabolami w przedziale między punktami przecięcia.

13. Samochód w chwili t = 0 jedzie z prędkością 10 km/h. Rozpoczyna przyspieszanie, gdzie przyspieszenie a(t) jest opisane funkcją:

    $$a(t) = 4+0.02t$$

    Obliczyć prędkość auta po czasie t = 20. Jaką drogę pokona auto. Narysować odpowiednie wykresy prędkości, przyspieszenia i drogi.

14. Napisać program w *M-pliku*, który znajduje dokładne współrzędne punktów przecięcia półokręgu opisanego funkcją:

    $$y = \sqrt{1 - x^2}$$

    z parabolą według funkcji:

    $$z = 5x^2 - 3$$

    Zilustrować rozwiązanie na wykresie.

15. Znaleźć pole powierzchni pomiędzy parabolą o równaniu:

    $$y_1 = -6x^2+2x+1$$

    a prostą poziomą o równaniu:

    $$y_2 = -3$$

16. Znaleźć pole powierzchni pomiędzy parabolą o równaniu:

    $$y_1(x) = x^2-10$$

    a prostą o równaniu:

    $$y_2(x) = 3x+2$$

    w granicach pomiędzy ich punktami przecięcia.

17. Znaleźć metodą symboliczną granicę ciągu:

    $$b_n = (1 + \frac{1}{n})^n \qquad \text{dla } n \to \infty.$$

Równania różniczkowe

1. Rozwiązać równanie różniczkowe:

    $$y'' + y = tg\, x$$

    z warunkami początkowymi: y(0) = 3, y'(0) = 5

    Narysować wykres funkcji w przedziale (0, π). Sprawdzić rozwiązanie przez podstawienie do równania.

2. Rozwiązać układ równań różniczkowych:

$$\frac{du}{dx} = -u + v$$

$$\frac{dv}{dx} = 3v + u$$

z warunkami początkowymi: u(0) = 3, v(0) = 0. Narysować wykresy obydwu funkcji w przedziale (0, 1). Sprawdzić rozwiązania przez podstawienie do równań.

3. Rozwiązać równanie różniczkowe:

$$3y' - 8\,cos^3\mathrm{x} = 0$$

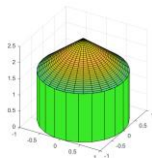
z warunkiem początkowym: y(0) = 3, narysować wykres tej funkcji oraz wyznaczyć pole powierzchni pomiędzy osią x a tą krzywą dla x w przedziale (0, 10).

Definiowanie funkcji użytkownika

1. Dla kwadratu o boku długości *a* napisać funkcję anonimową, która wyznacza pole okręgu opisanego i okręgu wpisanego w ten kwadrat. Zastosować funkcję w *M-pliku*, dla wartości boku *A* podawanej przez użytkownika.
2. Napisać funkcję anonimową (albo funkcję w osobnym *M-pliku*), której zadaniem będzie obliczenie różnicy pól pomiędzy okręgiem o zadanym promieniu *R*, a kwadratem wpisanym w ten okrąg.
3. Napisać funkcję anonimową (albo funkcję w osobnym *M-pliku*), której zadaniem będzie obliczanie przeciwprostokątnej trójkąta prostokątnego przy zadawanych wartościach przyprostokątnej *a* i kąta *alfa* między tą przyprostokątną a przeciwprostokątną.
4. Utworzyć własną funkcję w osobnym pliku, której zadaniem będzie rysowanie wykresu paraboli – argumentami tej funkcji będą współczynniki *a*, *b*, *c* tej paraboli. Wykorzystać 3-krotnie tę funkcję do narysowania wykresów trzech dowolnych paraboli w jednym układzie współrzędnych.
5. Utworzyć własną funkcję w osobnym pliku, której zadaniem będzie losowanie 100 liczb całkowitych z przedziału (a, b) i obliczanie ile z nich należy do przedziału ($a + \frac{b-a}{3}$, $b - \frac{b-a}{3}$). Niech *a* i *b* będą argumentami funkcji.
6. Utworzyć w osobnym pliku własną definicję funkcji, której zadaniem jest obliczanie pola powierzchni równoległoboku o podstawie *A*, wysokości *H* i kącie między podstawą i bokiem *ALFA*. Wykorzystać funkcję we własnym *M-pliku* dla wybranego zestawu danych.

   Dodatkowo: Wyposażyć naszą funkcję w rysowanie wykresu tego równoległoboku.

7. Napisać własną funkcję obliczającą objętość bryły jak na rysunku, przy zadanych wysokościach walca i stożka oraz promieniu podstawy.



8. Utworzyć własną definicję funkcji w osobnym *M-pliku*, której zadaniem jest obliczanie pola powierzchni wspólnego dla dwóch paraboli o równaniach:

$$y_1 = A\,x^2 - B$$

oraz

$$y_2 = -Cx^2 + D \qquad \text{gdzie } A, B, C, D > 0$$

Pole obliczamy w granicach punktów przecięcia obydwu paraboli.

Wykorzystać zdefiniowaną funkcję do obliczeń tej powierzchni dla założonych dowolnych wartości A, B, C, D. Zilustrować równania na wykresie obydwu paraboli.

9. Utworzyć w *M-pliku* własną funkcję, której zadaniem będzie skalowanie dowolnego konturu zamkniętego (macierz transformacji – rozdz.15.6) zdefiniowanego przez współrzędne opisane w dwóch wektorach. Argumentem funkcji będzie współczynnik skali. Wykorzystać zdefiniowaną funkcję na wykresie przedstawiającym obraz oryginalny i skalowany.

Funkcje statystyki

1. Wyniki badania cen pewnego towaru w 10 sklepach (próba) przedstawia wektor:

   ceny =[11.90 11.30 10.98 12.10 9.99 10.50 10.90 11.87 10.20 11.45]

   Obliczyć wartość mediany dla tych cen.

2. W *M-pliku* wygenerować macierz o rozmiarze 3x30 z losowymi wartościami elementów z przedziału 100 do 200. Traktując każdy wiersz macierzy jako próbę, znaleźć wariancje i odchylenia standardowe dla każdego z wierszy macierzy.

Aproksymacja i interpolacja

1. Znaleźć parabolę 3 stopnia aproksymującą zestaw punktów określonych współrzędnymi na płaszczyźnie:

   (0, 1) (2, 2) (4, -5) (6,0)

   Zilustrować zadanie przy pomocy wykresu.
2. Znaleźć współczynniki paraboli 4-go stopnia, przechodzącej przez punkty:

   (-1, -1), (2, 1), (3, 0) (4, 2.5)

   Zilustrować rozwiązanie wykresem. Znaleźć wszystkie rzeczywiste miejsca zerowe paraboli oraz wartości zmiennej niezależnej dla jej ekstremów.
3. Korzystając z funkcji *polyfit* znaleźć parabolę 3-go stopnia, posiadającą następujące miejsca zerowe:

   -2.3, 1.0, 3.56

   Zilustrować rozwiązanie na wykresie. Znaleźć wartości zmiennej niezależnej dla jej ekstremów. Znaleźć współrzędne punktów przegięcia.
4. Znaleźć parabolę przechodzącą przez zestaw punktów określonych współrzędnymi na płaszczyźnie:

   (0, 1) (2, 2) (4, -5) (6,0) (8, 2)

   Dobrać stopień paraboli. Zilustrować zadanie przy pomocy wykresu.
5. Utworzyć dwa wektory: x = 0:2:10 i y o tej samej długości, z dowolnymi liczbami. Wyznaczyć funkcję interpolującą metodą "*spline*" i znaleźć interpolowane wartości y w punktach dla x = 1, 3 i 5.

Klasy i obiekty

1. Utworzyć definicję klasy *Student* o polach: *nazwisko*, *imię*, *rok_studiow*, *przedmioty* (struktura nazw przedmiotów z ocenami) oraz metodach: *nadaj_ocene* i *wyświetl_wszystkie_oceny*. W *M-pliku* utworzyć dwa obiekty klasy *Student* i wykorzystać metody dla tych obiektów, korzystając z definicji klasy.
2. Utworzyć definicję klasy *Macierz*, wyposażoną we właściwość M = [ ] (pusta macierz). Konstruktor niech tworzy macierz losową o rozmiarze n×n, gdzie rozmiar jest argumentem wejściowym. Zdefiniować dwie metody: *macierz_odwrotna* i *wyznacznik*. W *M-pliku* utworzyć obiekt klasy *Macierz* i wykorzystać obydwie metody.

Aplikacja komponentowa *GUI*

1. Utworzyć aplikację wizualną *GUI*, w której:
   - dwa suwaki ustalają zakres zmienności liczb od -4 do 4 każdy,
   - po ustawieniu pozycji suwaków kliknięty przycisk rysuje wykres paraboli o równaniu:

   $$f(x) = ax^2 + b$$

   gdzie *a* i *b* to pobrane pozycje suwaków.

2. Utworzyć aplikację wizualną *GUI*, w której:
   - trzy komponenty edycyjne będą służyły do wpisania trzech liczb,
   - czwarty komponent edycyjny po kliknięciu przycisku będzie wyświetlał średnią arytmetyczną tych trzech liczb.
3. Utworzyć aplikację wizualną *GUI*, w której:
   - umieszczone są dwa pola edycyjne i trzy przyciski,
   - kąt *x*, w stopniach wpisujemy w pierwszym polu edycyjnym,
   - po kliknięciu odpowiedniego przycisku obliczane są wartości trzech funkcji: *sin(x)*, *cos(x)* i *tg(x)*,
   - wynik obliczenia (wybranego przyciskiem) pokazuje się w drugim polu edycyjnym,
   - dodatkowo zabezpieczyć program przed brakiem wpisanej liczby (puste pole edycyjne kąta).

Bibliografia

1. *MATLAB, The Language of Technical Computing. Desktop Tools and Development Environment*, MathWorks Inc.
2. *MATLAB, The Language of Technical Computing. Mathematics*, MathWorks Inc.
3. *MATLAB, The Language of Technical Computing. Using MATLAB Graphics,* MathWorks Inc.
4. Mrozek B., Mrozek Z.:*Matlab 5.x – poradnik użytkownika*, Wyd. PLJ, Warszawa, 1998.
5. Mrozek B., Mrozek Z.:*MATLAB i Simulink. Poradnik użytkownika*, Wyd.IV, Helion. 2017.
6. Pratap R.: *Matlab 7 dla naukowców i inżynierów*, Wyd. PWN, 2007.
7. Kamińska A., Pańczyk B.: *Ćwiczenia z Matlab. Przykłady i zadania*, Wyd. Mikom, 2002.
8. Drozdowski P.: *Wprowadzenie do Matlab-a*, skrypt, Politechnika Krakowska, 1996.
9. https://www.mathworks.com/products/matlab.html

Alfabetyczny skorowidz podstawowych funkcji w *MATLAB-ie*

abs	wartość bezwzględna liczby, także zespolonej
alfa	ustawienie przezroczystości
and	koniunkcja (iloczyn logiczny)
any	sprawdzenie czy w macierzy są wartości niezerowe
angle	kąt atan(b/a) dla liczby zespolonej
axis	ustalenie granic dla osi wykresu
bar	wykres słupkowy
break	wymuszenie zakończenia pętli *for* lub *while*
cd	zmiana katalogu bieżącego
ceil	zaokrąglenie do całkowitej w kierunku $+\infty$
char	konwersja do macierzy znaków
class	informacja o typie zmiennej lub wyrażenia
clc	wyczyszczenie okna *Command Win*dow
clear	usuwanie zmiennych z przestrzeni roboczej *Workspace*
close	zamknięcie okna wykresu
clf	usuwanie krzywych z wykresu
collect	grupowanie wyrażenia symbolicznego według potęg określonej zmiennej
colormap	ustalanie mapy kolorów (np. dla wykresów)
cos	cosinus kąta podanego w radianach
corrcoef	współczynnik korelacji
cosd	cosinus kąta podanego w stopniach
cot	cotangens kąta podanego w radianach
cotd	cotangens kąta podanego w stopniach
cov	kowariancja
datetime	definiowanie daty
delete	usuwanie pliku
det	wyznacznik macierzy kwadratowej
diag	wektor przekątnej głównej macierzy
diary	obsługa pliku dziennika sesji
diff	wyznaczenie pochodnej dla funkcji symbolicznej
dir	spis elementów katalogu
disp	wypisanie tekstu lub wartości zmiennej
double	konwersja na typ dziesiętny (np. liczb symbolicznych), kod ASCII znaku
dsolve	rozwiązywanie równania lub układu równań różniczkowych
equationsToMatrix	konwersja układu symbolicznych równań liniowych do macierzy współczynników

eig	wyznaczenie wartości własnych macierzy
exit	zakończenie programu *MATLAB*
exp	funkcja wykładnicza
eye	macierz jednostkowa (jedynki na przekątnej)
ezplot	wykres funkcji symbolicznej dwuwymiarowej
ezplot3	wykres krzywej symbolicznej w przestrzeni
ezsurf	wykres powierzchni symbolicznej w przestrzeni
ezmesh	wykres powierzchni symbolicznej w przestrzeni
figure	inicjacja okna wykresu, obrazu
fill	wykres konturu zamkniętego
find	wyszukiwanie w macierzy
findstr	znajduje fragment tekstu wektorze znaków
fix	zaokrąglenie do liczby całkowitej w kierunku 0
floor	zaokrąglenie do liczby całkowitej w kierunku $-\infty$
format	ustalenie formatu liczb
fplot	szybki wykres dwuwymiarowy
fprintf	wydruk formatowany
fzero	szukanie miejsc zerowych funkcji anonimowej
grid	siatka wykresu
guide	inicjacja tworzenia aplikacji okienkowych
help	pomoc *MATLAB-a*
histogram	rysowanie histogramu
hold	zablokowanie (w celu dorysowania nowych krzywych) lub odblokowanie okna wykresu
horzcat	łączenie poziome macierzy
imag	część urojona liczby zespolonej
image	wyświetlanie obrazu
importdata	ładowanie obrazu z pliku
input	wprowadzanie interakcyjnej wartości do zmiennej
inputdlg	okienko do wprowadzania danych
int	całkowanie symboliczne
interp1	interpolacja dwuwymiarowa
interp2	interpolacja trójwymiarowa
intersect	iloczyn zbiorów
inv	macierz odwrotna
ismember	badanie czy element jest w macierzy
legend	legenda wykresu
length	długość wektora, większy z rozmiarów macierzy
limit	granica ciągu lub funkcji symbolicznej
linsolve	rozwiązywanie układu równań liniowych

linspace	tworzenie wektora o stałym przyroście wartości elementów
load	odczyt z pliku
log	logarytm naturalny
log10	logarytm dziesiętny
lower	zamienia duże litery na małe
magic	macierz kwadratu magicznego
max	największy element wektora, każdej kolumny macierzy
median	mediana
mesh	wykres powierzchniowy
meshgrid	siatka dla wykresu powierzchniowego
min	najmniejszy element wektora, każdej kolumny macierzy
mkdir	tworzenie katalogu
not	negacja
nthroot	pierwiastek n-tego stopnia
num2str	konwersja liczby na tekst
numel	liczba elementów macierzy
ones	tworzenie macierzy wypełnionej jedynkami
or	alternatywa (suma logiczna)
patch	rysowanie wielokątów
path	ścieżka do katalogów przeszukiwań
pause	zatrzymanie biegu programu
plot	wykres krzywej lub krzywych w układzie kartezjańskim
plot3	wykres krzywej w przestrzeni
polarplot	wykres biegunowy
polyfit	aproksymacja wielomianem
polyval	wstawianie liczb do wielomianu
power	potęga
pretty	tekstowy model matematyczny wyrażenia
prod	iloczyn elementów wektora
rand	liczba pseudolosowa z przedziału (0, 1) – rozkład równomierny
randn	liczba pseudolosowa z przedziału (0, 1) – rozkład normalny
randi	liczba pseudolosowa całkowita
readtable	funkcja odczytu tabeli z pliku
real	część rzeczywista liczby zespolonej
rem	reszta z dzielenia
reshape	rekonfiguracja macierzy
return	wymuszenie zakończenia działania funkcji
rmdir	usuwanie katalogu
roots	pierwiastki wielomianu

rot90	obrócenie macierzy o 90 stopni
round	zaokrąglenie do najbliższej liczby całkowitej
save	zapis do pliku
set	nadawanie wartości właściwościom obiektów
setdiff	różnica zbiorów
simplify	uproszczenie wyrażenia symbolicznego
simulink	uruchomienie pakietu *Simulink*
sin	sinus kąta podanego w radianach
sind	sinus kąta podanego w radianach
size	rozmiary macierzy
solve	symboliczne rozwiązywanie równań nieliniowych
sort	sortowanie elementów macierzy
sphere	rysowanie wykresu kuli
sqrt	pierwiastek kwadratowy
std	odchylenie standardowe
str2num	konwersja tekstu na liczbę
strcat	łączenie tekstów
str2double	konwersja tekstu na liczbę, także tekstów będących elementami tablicy komórkowej (*cell array*)
str2sym	konwersja tekstu na postać symboliczną wyrażenia
strcmp	porównywanie tekstów
struct	definicja struktury
subplot	wykres składowy w oknie *Figure*
subs	podstawianie liczb do wyrażenia symbolicznego
sum	suma elementów wektora, każdej kolumny macierzy
surf	funkcja tworząca wykres powierzchniowy
syms	definiowanie zmiennych symbolicznych
table	funkcja tworzenia tabeli
tan	tangens kąta podanego w radianach
tand	tangens kąta podanego w stopniach
title	tytuł wykresu
trace	suma elementów przekątnej macierzy
transpose	transpozycja macierzy
tril	macierz trójkątna dolna
triu	macierz trójkątna górna
union	suma zbiorów
unique	zwraca wektor unikalnych wartości macierzy
var	wariancja
upper	zamienia małe litery na duże
writetable	funkcja zapisu tabeli do pliku

vertcat	łączenie pionowe macierzy
view	perspektywa patrzenia na wykres 3D
vpasolve	rozwiązywanie rownań w wybranym przedziale
whos	lista zmiennych *Workspace* i ich typy
xlabel	etykieta osi x wykresu
ylabel	etykieta osi y wykresu
zeros	macierz z elementami zerowymi