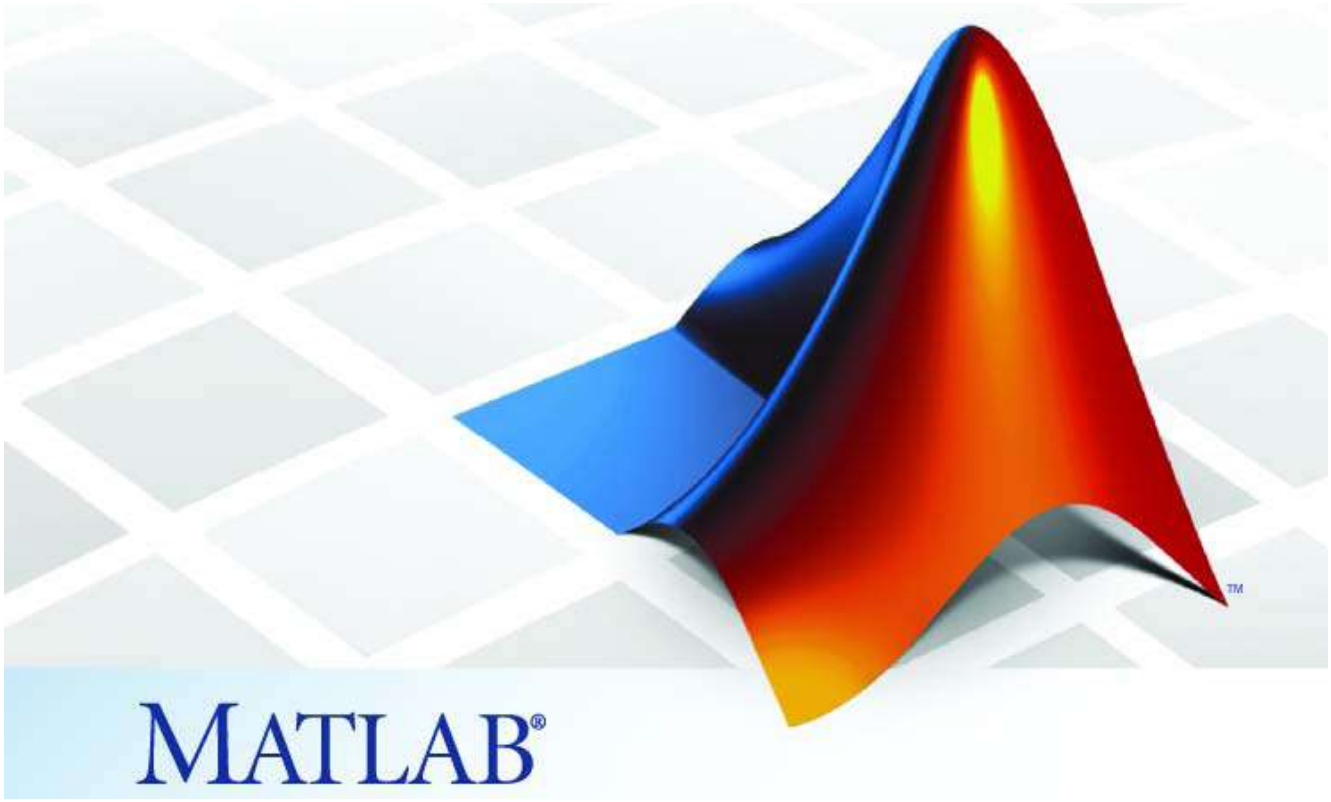
Katedra Systemów Geoinformatycznych

Pomoc do MATLAB-a

Materiały pomocnicze do wykładu:
Techniki obliczeniowe w systemach geoprzestrzennych



Gdańsk, 20015 r.

Spis treści

1 Wstęp do MATLAB-a.

1.1 Podstawowe wiadomości o programie

MATLAB jest językiem programowania wysokiego poziomu, umożliwiając jednocześnie pracę w środowisku interakcyjnym. Nazwa programu pochodzi od MATrix LABoratory. Użytkownik operuje jednym typem danych – macierzą. Nawet pojedyncza liczba reprezentowana jest przez macierz kwadratową o wymiarach 1×1. Praca w środowisku MATLAB-a polega na wprowadzaniu komend dla interpretera języka.

Podstawowe zasady języka:

- Zmienne są inicjowane automatycznie, przy pierwszym wystąpieniu, a ponieważ jest tylko jeden typ zmiennych, nie wymagają one wcześniejszej deklaracji.
- Macierze indeksowane są od 1. Stałe macierzowe zapisywane są w nawiasach kwadratowych [].
- Stałe tekstowe zapisuje się w apostrofach ".
- Listę zmiennych występujących w obszarze roboczym można zobaczyć używając komendy **who** lub **whos**.
- Usunięcie zmiennej z obszaru roboczego wykonuje się komendą **clear**.
- Nazwy rozpoczynają się od litery, litery duże i małe są rozróżniane, identyfikatorem są pierwsze 63 znaki (w zależności od wersji programu, informuje o tym funkcja **namelengthmax**). Wprowadzenie zmiennej o nazwie identycznej z nazwą istniejącej funkcji spowoduje przesłonięcie funkcji.
- Jeżeli zapiszemy wyrażenie nie posiadające lewej strony (bez operatora podstawienia), to system wygeneruje zmienną **ans**, która przyjmuje wartość ostatnio wykonanej operacji.
- Jeżeli wyrażenie nie będzie zakończone znakiem średnika, to system automatycznie uruchomi funkcję **display**, wyświetlającą wynik wykonanej operacji.
- MATLAB używa standardowego zapisu liczb z kropką jako separatorem dziesiętnym. W liczbie może wystąpić znak **e** oznaczający notację wykładniczą. Dopuszczalne jest również użycie znaków **i** lub **j** dla oznaczenia części urojonej liczby zespolonej.
- MATLAB może wyświetlać wyniki z dokładnością 16 cyfr dziesiętnych. Zakres bezwzględnej wartości liczb zmiennoprzecinkowych: 10^{-308} ... 10^{308}. Sposób wyświetlania zależy od aktualnego parametru dla funkcji **format –** może to być np. **short**, **long, bank**, **hex** i wiele innych.
- Dostępne operatory arytmetyczne (macierzowe):
  - dodawanie +
  - odejmowanie -
  - mnożenie *
  - dzielenie /
  - potęgowanie ^
- Specjalne operatory macierzowe:
  - dzielenie lewostronne \
  - transpozycja zespolona '
- Operatory tablicowe (skalarne)
  - mnożenie .*
  - dzielenie ./
  - dzielenie lewostronne .\
  - potęgowanie .^
  - transpozycja .'
- Operatory logiczne
  - równe = =
  - różne ~ =
  - mniejsze <
  - większe >
  - nie większe <=

- nie mniejsze >=
- i &
- lub |

Uwaga: operatorom odpowiadają funkcje wbudowane MATLAB-a (ich listę można znaleźć w rozdz. 6.2.8).

- Użyteczne stałe :
  - pi $\pi = 3.14159265...$
  - i, j jednostka urojona, $i = \sqrt{-1}$
  - eps dokładność mantysy liczb zmiennoprzecinkowych: $eps = 2.2204 \times 10^{-016}$
  - realmin najmniejsza dodatnia liczba zmiennoprzecinkowa: $realmin = 2.2251 \times 10^{-308}$
  - realmax największa liczba zmiennoprzecinkowa: $realmax = 1.7977 \times 10^{+308}$
  - intmin najmniejsza liczba całkowita: intmin = -2147483648
  - intmax największa liczba całkowita: intmax = 2147483647
  - inf nieskończoność (np. wynik dzielenia n/0)
  - NaN brak liczby (Not-a-Number, np. wynik dzielenia 0/0)

1.2 Podstawowe polecenia współpracy z środowiskiem

Aby uruchomić polecenie środowiska (np. *dir*) należy w wierszu poleceń w programie MATLAB użyć tego polecenia, poprzedzonego znakiem przejścia do komend systemowych **!**. Gdy chcemy dodatkowo takie polecenie uruchomić w odrębnym oknie, kończymy je znakiem **&** (np.: `!cmd&`).
MATLAB dostarcza również szeregu wbudowanych funkcji do współpracy ze środowiskiem, m.in.:

- **dir** – wylistuj zawartość (bieżącego) katalogu.
- **what** – jest odmianą polecenia **dir**, służącą do wyświetlenia zawartości bieżącego katalogu.
- **ls** – polecenie listowania zawartości katalogu w stylu systemu Unix.
- **cd** – zmiana bieżącego katalogu roboczego.
- **delete** – może służyć do usunięcia pliku z bieżącego katalogu.
- **rmdir** – zmiana nazwy katalogu.
- **mkdir** – tworzenie nowego katalogu.

1.3 Macierze i operacje na macierzach

1.3.1 Tworzenie i łączenie macierzy

Ze względu na to, że MATLAB jest środowiskiem zorientowanym macierzowo, wszystkie dane wprowadzane do programu są przechowywane w postaci macierzy (lub tablic wielowymiarowych), niezależnie od użytego typu danych. Domyślnym typem danych jest **double**:

```
>> T = 5;
>> whos T
  NameSize            Bytes      Class
  T   1×1                 8      double array
Grand total is 1 element using 8 bytes
>>
```

Najprostszym sposobem utworzenia macierzy jest zastosowanie operatora konstrukcji **[]**. Wewnątrz nawiasów kwadratowych wprowadzamy kolejno, wierszami, elementy macierzy. Jako separatory elementów w wierszach mogą być użyte znaki odstępu lub przecinki. Separatorami kolumn mogą być znaki nowego wiersza lub średniki. Przykładowo, utwórzmy macierz o wymiarach 3×3:

```
>> T = [8,1,6;3,5,7;4,9,2]
T =
    8    1    6
    3    5    7
    4    9    2
>> whos T
  NameSize            Bytes      Class
  T   3×3                72      double array
Grand total is 9 elements using 72 bytes
>>
```

(zauważmy, że brak średnika na końcu instrukcji spowodował wymuszenie wyświetlenia zawartości nowo utworzonej macierzy).
MATLAB oferuje szereg gotowych funkcji generujących pewne specjalne rodzaje macierzy np.:

- **ones** - tworzenie macierzy wypełnionej jedynkami.
- **zeros** – tworzenie macierzy wypełnionej zerami.
- **eye** – tworzenie macierzy diagonalnej, wszystkie elementy głównej przekątnej mają wartość 1.
- **diag** – tworzenie macierzy diagonalnej z wektora.
- **magic** – tworzenie "kwadratu magicznego" o zadanym wymiarze.
- **rand** – tworzenie macierzy wypełnionej liczbami losowymi o rozkładzie równomiernym w przedziale [0,1).

Rozmiary macierzy mogą być bardzo łatwo powiększane. Weźmy na przykład utworzoną poprzednio macierz T:

```
>> T(2,4)=1
T =
    8    1    6    0
    3    5    7    1
    4    9    2    0
>> whos T
  NameSize            Bytes      Class
  T   3×4                96      double array
Grand total is 12 elements using 96 bytes
>>
```

Macierz została powiększona do rozmiarów 3×4 po dodaniu nowego elementu, pozostałe (nie definiowane) elementy macierzy są inicjowane z wartością zerową.
Macierze mogą być łączone pionowo lub poziomo za pomocą konstruktora:

```
>> A = ones(2)
A =
  1    1
  1    1
>> B = rand(2)
B =
  0.9501    0.6068
  0.2311    0.4860
>> C = [A;B]
C =
  1.0000    1.0000
  1.0000    1.0000
  0.9501    0.6068
  0.2311    0.4860
>> D = [A B]
D =
  1.0000    1.0000    0.9501    0.6068
  1.0000    1.0000    0.2311    0.4860
>>
```

MATLAB oferuje również szereg funkcji ułatwiających tworzenie nowych macierzy z kombinacji macierzy istniejących, jako alternatywę dla operatora konstrukcji []:

- **cat** – łączenie macierzy wzdłuż określonego wymiaru,
- **horzcat** – łączenie macierzy poziomo (dodawanie kolumn),
- **vertcat** – dołączanie macierzy pionowo (dodawanie wierszy),
- **repmat** – wielokrotne powtórzenie macierzy pionowo i poziomo,
- **blkdiag** – konstrukcja macierzy, w której kolejne macierze dołączane są diagonalnie.

Przedstawiony powyżej przykład mógłby więc być zapisany np. w postaci:

```
>> A = ones(2); B = rand(2);
>> C = cat(1, A, B); D = horzcat(A, B);
>>
```

Puste macierze są w procesie łączenia macierzy pomijane.
Dołączenie do macierzy nowych elementów, innego typu niż elementy w niej występujące, powoduje konwersję typu danych w macierzy w taki sposób, aby wszystkie elementy macierzy wynikowej były

tego samego typu. Przykładowo liczby przy łączeniu ze znakami zamieniane są na znaki (wg kodu ASCII):

```
>> t = ['A' 'BC' 32 68 69]
t =
ABC DE
>>
```

Dane typu logicznego (**true**, **false**) konwertowane są na liczby **1** i **0** (ale nie istnieje konwersja typu logicznego na znakowy). Oczywiście w MATLAB-ie mogą występować całe macierze logiczne logiczne, które można wykorzystywać w wyrażeniach indeksowych. Przykład: mamy macierz A oraz macierz B, która będzie przyjmowała wartość **true** (w zapisie: 1), gdy spełniony jest określony warunek..

```
>> A = magic(4);
A =
  16    2    3   13
   5   11   10    8
   9    7    6   12
   4   14   15    1
>> B = (A > 8)
B =
   1    0    0    1
   0    1    1    0
   1    0    0    1
   0    1    1    0
>>
```

Macierz B może być używana np. jako maska:

```
>> C = A .* B
C =
  16    0    0   13
   0   11   10    0
   9    0    0   12
   0   14   15    0
>>
```

1.3.2 Generowanie sekwencji liczbowych

Do utworzenia sekwencji liczb używamy operatora **:** (dwukropka). Oto przykłady

```
>> a = 3:7
a =
  3    4    5    6    7
>> b = 8:-2:3
b =
  8    6    4
>> c = 1:0.2:2
c =
  1.0000    1.2000    1.4000    1.6000    1.8000    2.0000
>>
```

1.3.3 Dostęp do elementów macierzy

Wartość pojedynczego elementu macierzy możemy pobrać określając numer wiersza i kolumny elementu w macierzy:

```
>> A = rand(2)
A =
  0.9501    0.6068
  0.2311    0.4860
>> A(1,2)
ans =
  0.6068
>>
```

Każda dwuwymiarowa macierz traktowana jest jak ciąg kolumn, w którym elementy ponumerowane są kolejno, poczynając od 1 – jest to tzw. indeksowanie liniowe. W naszym przykładzie element A(1, 2) jest więc równoznaczny z elementem A(3) (ogólnie, dla tablicy o wymiarach m×n element

A(i, j) będzie miał indeks liniowy (j-1)×n+m). Dostępne są funkcje konwersji indeksu liniowego na numery wiersza i kolumny i vice-versa.

```
>> A = rand(2)
A =
  0.9501    0.6068
  0.2311    0.4860
>> n = sub2ind(size(A),1,2)
n =
  3
>> [w k] = ind2sub(size(A),3)
w =
  1
k =
  2
>>
```

Dostęp do większej ilości elementów macierzy jest możliwy przy zastosowaniu operatora **:**.
Obliczmy na przykład sumę elementów trzeciego wiersza macierzy "magicznej" o wymiarach 4×4.

```
>>A = magic(4);
>>A(3,1)+A(3,2)+A(3,3)+A(3,4)
ans =
  34
>>% To samo, ale krócej
>>sum(A(3,1:4))
ans =
  34
>>% Zastosowanie skrótowego zapisu dla całego wiersza
>>sum(A(3,:))
ans =
  34
>>
```

Suma nieparzystych elementów drugiego wiersza:

```
>>sum(A(2,1:2:end))
ans =
  15
>>
```

Wielokrotny dostęp do pojedynczego elementu macierzy można zapisać używając funkcji **ones** przy określaniu indeksu. Przykładowo utworzenie nowej macierzy B o wymiarach 2×3, wypełnionej trzecim (wg indeksowania liniowego) elementem macierzy A :

```
>>B = A(3 * ones(2,3));
>>
```

Przy pracy z macierzami przydatne są funkcje zwracające informacje o macierzy:

- **length** – największy z wymiarów macierzy,
- **ndims** – ilość wymiarów,
- **numel** – ilość elementów,
- **size** – wymiary macierzy.

1.3.4 Zmiana rozmiarów macierzy

Próba dostępu do elementu spoza macierzy wywołuje komunikat błędu. Dodanie elementu o indeksach wykraczających poza istniejące wymiary macierzy powoduje automatyczne powiększenie tych wymiarów tak, aby nowy element znalazł się wewnątrz macierzy. Powstałe przy tym nowe elementy przyjmują wartości zerowe.
Przez podstawienie zamiast **całego** istniejącego wiersza lub kolumny macierzy **pustej** ([]) można zredukować jeden z wymiarów macierzy:

```
>> A = magic(4)
A =
 16    2    3   13
  5   11   10    8
  9    7    6   12
  4   14   15    1
>> A(2, :) = []
```

```
A =
 16    2    3   13
  9    7    6   12
  4   14   15    1
>>
```

Podstawienie pustej macierzy zamiast pojedynczego elementu (lub podzbioru elementów) wywołuje komunikat o błędzie, chyba że zastosowano indeksowanie liniowe. W tym przypadku macierz zostaje **przekonwertowana** na wektor wierszowy zawierający pozostałe elementy macierzy.
Zmianę wymiarów lub kształtu macierzy mogą ułatwić wbudowane funkcje:

- **reshape** – zmiana wymiarów macierzy. W macierzy wynikowej elementy umieszczane są w kolejności indeksu liniowego macierzy wejściowej.
- **rot90** – obrót macierzy w taki sposób, że ostatnia kolumna staje się pierwszym wierszem.
- **fliplr** – obrót macierzy wokół osi pionowej.
- **flipud** – obrót macierzy wokół osi poziomej.
- **flipdim** – obrót macierzy wokół wybranej osi.
- **transpose** – obrót macierzy wokół głównej przekątnej, zamiana wektorów kolumnowych na wierszowe i vice-versa.
- **ctranspose** – obrót macierzy wokół głównej przekątnej i zastąpienie wszystkich elementów ich wartościami sprzężonymi.

Zamiast funkcji **transpose** można użyć operatora transpozycji (.'), natomiast odpowiednikiem funkcji **ctranspose** jest operator (').

1.3.5 Macierze puste, skalary i wektory

MATLAB umożliwia utworzenie macierzy pustej (która może być np. początkową macierzą w iteracji). Macierz pusta ma przynajmniej jeden wymiar równy 0. Przykłady definiowania macierzy pustych:

```
>>A = [];
>>B = zeros(0,3);
>>whos
  NameSize            Bytes      Class
  A   0×0                 0      double array
  B   0×3                 0      double array
Grand total is 0 elements using 0 bytes
>>
```

Macierzy pustych można używać przestrzegając reguł działania poszczególnych operatorów.
Szczególnym przypadkiem macierzy, jest macierz o wymiarach 1×1, nazywana skalarem lub wartością skalarną. Jest to macierzowa reprezentacja pojedynczej liczby (rzeczywistej lub zespolonej).

```
>>A = 7; ndims(A)
ans =
  2
>> size(A)
ans =
  1    1
>> isscalar(A)
ans =
  1
>>
```

Innym szczególnym przypadkiem macierzy jest wektor, czyli macierz której jeden wymiar równy jest 1, a drugi jest większy od jedności. W szczególności ciągi znaków są traktowane jak wektory , których elementami są pojedyncze znaki. W zależności od tego, który z wymiarów jest większy od jedności mamy wektory wierszowe (pierwszy wymiar jest równy 1) bądź kolumnowe.

```
>> A = 'ABCD'; size(A)
ans =
  1    4
>> isvector(A)
ans =
  1
>>
```

1.3.6 Uwagi o tablicach wielowymiarowych

W programie MATLAB macierze są dwuwymiarowe. Pozycja każdego elementu w macierzy opisana jest dwoma parametrami - numer wiersza i numer kolumny. Tablice wielowymiarowe używają większej liczby indeksów. Tablica trójwymiarowa może być utworzona przez rozszerzenie macierzy dwuwymiarowej.

```
>> A = [1 2; 3 4];
>> A(:,:,2)= [5 6; 7 8]
A(:,:,1) =
  1    2
  3    4
A(:,:,2) =
  5    6
  7    8
>>
```

Funkcje wbudowane **rand**, **randn**, **zeros**, **ones** pozwalają na tworzenie tablic wielowymiarowych. Innym sposobem jest użycie funkcji łączenia tablic w określonym wymiarze (**cat**).

```
>> A = cat(3, [1 2; 3 4], [5 6; 7 8])
A(:,:,1) =
  1    2
  3    4
A(:,:,2) =
  5    6
  7    8
>>
```

Dowolny wymiar tablicy zadeklarowanej w tych funkcjach może być równy 0, dając w efekcie tablicę pustą.

1.4 Tablice komórek

Tablica komórek (ang. *cell array*) jest klasą, która umożliwia przechowywanie różnych typów danych w ramach jednego obiektu. Każda komórka w tablicy identyfikowana jest (jak w przypadku macierzy) parą indeksów – (*wiersz, kolumna*). Podobnie jak w przypadku macierzy, można do tablicy komórek zastosować indeksowanie liniowe, w którym cała tablica traktowana jest jako zbiór połączonych kolumn. Każda komórka w tablicy może zawierać dowolne postaci danych (np. macierze liczbowe, ciągi znaków itp.).

1.4.1 Tworzenie tablicy

- Metoda indeksowania komórek

Na poszczególne pozycje tablicy wprowadzamy komórki: indeksy komórki zamykamy w zwykłych nawiasach (), a prawą stronę instrukcji podstawienia zamykamy w nawiasach {}.

```
>> A(1,1) = {[1 4 3; 0 3 8; 2 9 7]};
>> A(1,2) = {3+5i};
>> A(2,1) = {'Koniec'};
>> A(2,2) = {5};
>> whos A
  NameSize            Bytes      Class
  A   2×2               348      cell array
Grand total is 21 elements using 348 bytes
>>
```

- Metoda indeksowania zawartości

Do komórek wprowadzamy zawartość: zamykamy indeksy komórki w nawiasach {}.

```
>> A{1,1} = [1 4 3; 0 3 8; 2 9 7];
>> A{1,2} = 3+5i;
>> A{2,1} = 'Koniec';
>> A{2,2} = 5;
```

- Używanie nawiasów

Nawiasy {} są konstruktorem tablicy komórek. Można definiować zagnieżdżone tablice komórek. Wewnątrz nawiasów używamy przecinków lub odstępów do separacji komórek w jednym wierszu, a średników lub znaków nowego wiersza do separacji wierszy tablicy.

```
>> C = {[1 2], [3 4]; [5 6], [7 8; 9 10]};
C =
  [1x2 double]    [1x2 double]
  [1x2 double]    [2x2 double]
>>
```

Nawiasów kwadratowych można używać również do łączenia tablic komórek.

- Metoda prealokacji tablicy

Używamy funkcji **cell** do utworzenia pustej tablicy o odpowiednich rozmiarach, a następnie przy pomocy instrukcji podstawienia wypełniamy poszczególne komórki zawartością.

```
>> B = cell(2,2);
>> B(2,2) = {0:0.1:1};
>> B{2,1) = 'Zakres zmian';
B =
             []               []
  [1x12 char  ]    [1x11 double]
>>
```

1.4.2 Używanie zawartości tablic komórek

Dostęp do podzbioru elementów tablicy komórek można uzyskać przy pomocy zapisu:

```
B = A(zakres_numerów_wierszy, zakres_numerów_kolumn);
```

Wynik tego podstawienia jest tablicą komórek (w szczególności jednoelementową).
Dostęp do zawartości wybranej komórki tablicy A zapewnia zapis w postaci:

```
B = A{wiersz, kolumna};
```

natomiast dostęp do poszczególnych elementów komórki wymaga dołączenia następnych indeksów:

```
B = A{wiersz, kolumna}(wiersz_w_komórce, kolumna_w_komórce);
```

Jeśli wybrana komórka zawiera tablicę, to dostęp do poszczególnych elementów tej tablicy wymaga dołączenia następnych indeksów:

```
B = A(zakres_numerów_wierszy, zakres_numerów kolumn)
```

W przypadku złożonej zawartości komórek (np. kolejnych, zagnieżdżonych tablic komórek), można budować dłuższe wyrażenia indeksowe:

```
>> A = {[1 1],'1 1';[2 3],{1,'5 6'}}
A =
    [1x2 double]    '1 1'
    [1x2 double]    {1x2 cell}
>> A{2,2}{1,2}(3)
ans =
6
```

1.5 Tablice struktur

Tablica struktur (ang. *structure array*) jest tablicą wektorów (struktur o jednakowych zestawach pól), w których poszczególne pola (ang. *fields*) identyfikowane są za pomocą nazw. Poszczególne pola danej struktury a również pola o tych samych identyfikatorach w poszczególnych strukturach, mogą zawierać dowolne typy danych. Tablica struktur różni się od tablicy komórek identyfikacją komórek danych za pomocą nazw a nie indeksów liczbowych.

1.5.1 Tworzenie tablicy struktur

Są dwie metody zainicjowania skalarnej struktury:

- Metoda przypisania danych
  Strukturę (tablicę o rozmiarach 1×1) można utworzyć podstawiając wartości dla poszczególnych pól (dostęp do pola odbywa się za pomocą wyrażenia *nazwa_struktury.nazwa_pola*):

```
>> s.a1 = 2;
>> s.a2 = 'dwa';
>> s.a3 = magic(3)
s =
    a1: 2
    a2: 'dwa'
    a3: [3x3 double]
>> whos s
```

```
  Name      Size            Bytes  Class     Attributes
  s         1x1               458  struct
```

- Metoda z użyciem konstruktora
  Ten sam efekt można uzyskać za pomocą funkcji **struct**:

```
>> s = struct('a1', 2, 'a2', 'dwa')
s =
    a1: 2
    a2: 'dwa'
    a3: [3x3 double]
```

Przy podstawieniach do tablic struktur należy uwzględnić indeksy struktury w ramach tablicy:

```
>> s(1).a1 = 2;
>> s(1).a2 = 'dwa';
>> s.a3 = magic(3);
>> s(2).a2 = 'puste'
s =
1x2 struct array with fields:
    a1
    a2
    a3
>> s(1)
ans =
    a1: 2
    a2: 'dwa'
    a3: [3x3 double]
>> s(2)
ans =
    a1: []
    a2: 'puste'
    a3: []
```

Pola, którym nie nadano wartości będą zawierać element pusty ([]). Analogicznie można zainicjować taką tablicę za pomocą funkcji **struct**:

```
>> s = struct('a1', {2, []}, 'a2', {'dwa', 'puste'}, ...
   'a3', {magic(3), []});
```

Zawartości poszczególnych pól wprowadzane są jako tablice komórek o rozmiarach zgodnych z rozmiarami tablicy struktur lub jako wartości skalarne - w tym przypadku można pominąć nawiasy {}.
Uwaga: podanie pojedynczej wartości pola *a1* lub *a3* spowoduje nadanie w tablicy jednakowej wartości wszystkim polom o danym identyfikatorze!

1.5.2 Używanie zawartości tablic struktur

Dostęp do pojedynczej struktury z tablicy A zapewnia zapis w postaci:

```
A(wiersz,kolumna),
```

natomiast dostęp do poszczególnych pól wymaga dołączenia nazwy pola (i ewentualnie dalszych indeksów, jeżeli zawartość pola nie jest skalarem):

```
A(wiersz, kolumna).nazwa_pola(numer_wiersza,numer_kolumny).
```

W przypadku, gdy A nie jest skalarem, zapis

```
A.nazwa_pola
```

oznacza dostęp funkcji **disp** do zawartości określonego pola we wszystkich strukturach wchodzących w skład tablicy A, ale podstawienie

```
x = A.nazwa_pola;
```

zwróci na zmienną x jedynie wartość pola z pierwszej struktury z tablicy A.
W MATLAB-ie dopuszczalny jest również dynamiczny dostęp do pól struktury. Jeśli np. zmienna **pole** zawiera nazwę pola, to dostęp do tego pola może być realizowany przy użyciu zapisu:

```
pole = 'nazwa_pola';
x = A(wiersz,kolunma).(pole)
```

W szczególności można się odwołać do pola również poprzez:

```
A(wiersz,kolunma).('nazwa_pola') .
```

1.6 Typy danych

Dane numeryczne, w zależności od reprezentacji maszynowej, mogą być:

- całkowite (8, 16, 32 i 64 bitowe ze znakiem lub bez) – konwersja do tych typów odbywa się za pomocą odpowiedniej funkcji: **int8, int16, int32, int64, uint8, uint16, uint32 lub uint64**.
- zmiennoprzecinkowe podwójnej precyzji (64 bity, w tym 52 bity na mantysę) – konwersja za pomocą funkcji **double**. Jest to domyślna postać przechowywania liczb.
- zmiennoprzecinkowe pojedynczej precyzji (32 bity, w tym 23 bity na mantysę) – konwersja za pomocą funkcji **single.**
- Istnieje funkcja **cast**, pozwalająca na konwersję typu zmiennej do jednego z typów wbudowanych (pod warunkiem, że jest to realizowalne).

Dane logiczne mogą przybierać wartości **true** i **false**, reprezentowane odpowiednio przez **1** i **0**. Konwersja do typu logicznego odbywa się za pomocą funkcji **logical.**

Dane tekstowe mogą być przechowywane w postaci tablic znaków. Pojedynczy znak zajmuje jeden bajt. Ciąg znaków stanowi wiersz tablicy znaków. W tablicy wielowierszowej wszystkie wiersze muszą być tej samej długości. Możliwe jest również tworzenie tablic zawierających ciągi znaków różnych długości za pomocą mechanizmów tablic komórek. Przekształcanie ciągu znaków na wektor liczb odbywa się za pomocą którejś z funkcji (np. **uint8** daje 8-bitową reprezentację każdego znaku). Przekształcenie wektora liczb całkowitych na ciąg znaków odbywa się za pomocą funkcji **char.**

1.7 Skrypty i funkcje w MATLAB-ie

1.7.1 Sterowanie przebiegiem obliczeń

Innym sposobem pracy jest przygotowanie skryptu zawierającego komendy języka w pliku o rozszerzeniu **.m** i umieszczeniu tego pliku w roboczym katalogu MATLAB-a. Plik taki może zostać uruchomiony przez wywołanie jego nazwy w wierszu komendy.
W skrypcie można używać instrukcji sterujących operacjami:

- instrukcje warunkowe
  - **if** *wyrażenie_logiczne*
    instrukcje
    elseif
    instrukcje
    else
    instrukcje
    end
  - **switch** *wyrażenie*
    case *wartość_1*
    instrukcje
    case *wartość_2*
    instrukcje
    otherwise
    instrukcje
    end
- instrukcja pętli
  - **for** *zmienna = początek* : [*przyrost* :]*koniec*
    instrukcje
    end
  - **while** *wyrażenie*
    instrukcje
    end
- instrukcje **break** i **continue**.

Skrypty napisane w języku MATLAB:

- umożliwiają automatyzację obliczeń w przypadku gdy ciąg kroków programowych ma być wykonywany wielokrotnie
- nie przyjmują argumentów przy wywołaniu i nie zwracają wartości przy wyjściu

- przechowują zmienne we wspólnej przestrzeni danych programu.

Pliki skryptów w MATLAB-ie można przygotować w sposób umożliwiający ich wykorzystanie jako funkcji. Funkcje MATLAB-a:

- umożliwiają rozszerzenie standardowych możliwości języka
- mogą przyjmować argumenty wejściowe i zwracać wartości przy wyjściu
- przechowują zmienne w przestrzeni lokalnej.

1.7.2 Struktura pliku zawierającego skrypt lub funkcję

Struktura pliku o rozszerzeniu **.m,** zawierającego skrypt lub funkcję zewnętrzną programu MATLAB jest ściśle określona:

Element pliku .m	**Opis**
Wiersz definicji funkcji	Występuje tylko w przypadku funkcji. Określa nazwę funkcji oraz ilość i kolejność parametrów wyjściowych i wejściowych.
Wiersz H1	Jednowierszowy, sumaryczny opis programu (funkcji), używany przez system pomocy programu MATLAB, wyświetlany jest przy wywołaniu pomocy w odniesieniu do całego folderu zawierającego dany plik .m.
Tekst pomocy	Bardziej szczegółowy opis programu, wyświetlany wraz z wierszem H1 przy wywołaniu pomocy w odniesieniu do konkretnego pliku. Wszystkie wiersze tekstu pomocy rozpoczynają się od znaku **%**. Pierwszy wiersz rozpoczynający się od innego znaku oznacza koniec tekstu pomocy.
Ciało funkcji lub skryptu	Właściwy kod programu, wykonujący zadane obliczenia (w przypadku funkcji z wykorzystaniem wartości parametrów wejściowych) i zwracający wartości wynikowe: w przypadku skryptów poprzez wspólną przestrzeń zmiennych lub za pośrednictwem argumentów wyjściowych w przypadku funkcji
Komentarze	Teksty umieszczone wewnątrz ciała programu wyjaśniające działanie wewnętrzne programu.

Wiersz definicji funkcji ma postać:

```
function <arg_wy> = <nazwa_funkcji>(<arg_we1>, <arg_we2>,...)
```

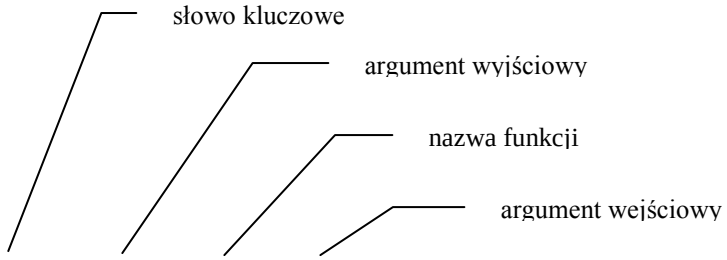
Jeżeli jest kilka argumentów wejściowych, umieszczamy je na liście rozdzielając przecinkami. Jeżeli jest kilka argumentów wyjściowych, umieszczamy je jako elementy wektora:

```
function [<arg_wy1> <arg_wy2> ...] = <nazwa_funkcji>(<arg_we>)
```

Nazwa funkcji, występująca w wierszu definicji jest pomijana, gdy różni się od nazwy pliku, w związku z tym dla uniknięcia nieporozumień dobrze jest używać tej samej nazwy.
Komentarze mogą być:

- jednowierszowe, rozpoczynające się od znaku **%**
- wielowierszowe, rozpoczynane znakami **%{** a kończone znakami **}%.** Znaki te muszą być jedynymi znakami w wierszu.
- na końcu wiersza, po znaku **%**

Przykład zawartości pliku **silnia.m** zawierającego funkcje obliczającą wartość silnia:




```
function y = silnia(x)                          wiersz definicji funkcji
% Obliczanie wartości silnia                    wiersz H1
% Funkcja silnia(n) zwraca wartość n!           tekst pomocy

% Wykorzystuje funkcję wbudowaną prod.          komentarz
y = prod(1:x);      % Body                      ciało funkcji
```

MATLAB przy pierwszym wywołaniu skryptu lub funkcji dokonuje jego kompilacji – dzięki temu każde następne użycie nie wymaga fazy interpretacji pliku. W przypadku dużych aplikacji można dokonać wstępnej kompilacji plików **.m** za pomocą funkcji **pcode** (powoduje to powstanie w bieżącym katalogu tzw. plików **.p** – *preparsed).* Taka kompilacja pozwala również na ukrycie kodu programu, ale jednocześnie ukrywa całą treść pomocy. Usunięcie skompilowanych funkcji z przestrzeni roboczej programu wykonuje się za pomocą polecenia **clear functions.**

1.7.3 Rodzaje funkcji

Z uwagi na sposób interpretacji można podzielić funkcje na następujące typy:

- funkcje wbudowane – funkcje zdefiniowane wewnętrznie w MATLAB-ie. Jeżeli istnieją odpowiadające im pliki **.m** (np. w folderach narzędziowych), to zawierają one jedynie teksty pomocy i wiersz wywołania funkcji wbudowanej,
- funkcje pierwotne (główne) – podstawowy sposób użycia funkcji tworzonych przez użytkownika. Funkcja pierwotna jest pierwszą (i najczęściej jedyną) funkcją występującą w pliku **.m**. Uruchomienie takiej funkcji odbywa się przez wprowadzenie nazwy pliku w wierszu komendy – stąd najczęściej przyjmuje się, że nazwa funkcji tożsama jest z nazwą pliku,
- podfunkcje (funkcje pomocnicze) – funkcje dodatkowo zdefiniowane wewnątrz pliku zawierającego definicję funkcji pierwotnej, wykorzystywane wewnątrz ciała funkcji pierwotnej. O ile funkcje pierwotne są dostępne z poza pliku, to do podfunkcji dostęp jest jedynie z wnętrza pliku. Każda funkcja rozpoczyna się wierszem definicji funkcji i posiada własny obszar danych. Poszczególne funkcje występują bezpośrednio jedna za drugą. Wszystkie definicje funkcji (lub żadna) zakończone są instrukcją **end**,
- funkcje zagnieżdżone, definiowane wewnątrz definicji innych funkcji. Funkcje zagnieżdżone mogą sięgać do obszaru danych funkcji nadrzędnych. Każda z definicji funkcji zagnieżdżonej wewnątrz definicji innej funkcji musi kończyć się instrukcją **end**,
- funkcje anonimowe – dają możliwość szybkiego definiowania funkcji na podstawie dowolnego wyrażenia MATLAB-a, bez tworzenia plików **.m**,
- funkcje przeciążone – używane w przypadkach, gdy istnieje potrzeba tworzenia różnych funkcji dla różnych typów argumentów wejściowych, (podobnie jak w językach zorientowanych obiektowo),
- funkcje prywatne – dają możliwość ograniczenia dostępu do funkcji (np. tylko w ramach klasy).

1.7.4 Uchwyty do funkcji

Użyteczna jest możliwość wprowadzenia zmiennej jako uchwytu do funkcji.Za przykład posłuży nam zdefiniowanie procedury wyświetlającej wykres zadanej funkcji. Załóżmy, że w pliku o nazwie **plotFHandle.m** umieszczono następujący tekst:

```
function x = plotFHandle(fhandle, data)
plot(data, fhandle(data))
```

Wywołanie tej funkcji z dwoma argumentami – uchwytem do funkcji oraz wektorem zmiennej niezależnej daje wykres tej funkcji (za pomocą funkcji wbudowanej **plot**(x,y)):

```
>> plotFHandle(@cos, -pi:.01:pi)
```

W tym przykładzie pierwszym argumentem jest uchwyt do wbudowanej funkcji cos. Uchwyt do funkcji wykorzystywany jest do tworzenia funkcji **anonimowych.** Na przykład chcąc wprowadzić definicję funkcji **m1** zwracającej pierwiastek z sumy kwadratów argumentów, użyjemy zapisu:

```
>> m1 = @(x, y) sqrt(x.^2 + y.^2);
```

Wywołanie takiej funkcji wyglądałoby następująco:

```
>> x = m1(3, 4)
x =
  5
```

Zauważmy, że ze gdyby argumentami aktualnymi były macierze, to względu na użycie skalarnych operacji potęgowania, w wyniku powstałaby macierz zawierająca wyniki działań na odpowiadających sobie elementach macierzy wejściowych x i y. Jedynym ograniczeniem jest w tym przypadku identyczność wymiarów obu macierzy.

W przypadku wywoływania funkcji bez argumentów za pomocą uchwytu do funkcji, należy użyć pustego argumentu ( ).

```
>> t1 = @ pi;          % lub t1 = @() pi;
>> t1()
ans =
    3.1416
>>
```

Użycie w definicji funkcji anonimowej nazw zmiennych nie występujących na liście argumentów formalnych, powoduje przyjęcie w tej definicji ich aktualnych wartości jako stałych. Na przykład zdefiniujemy funkcję następująco:

```
>> A = 2;
>> m1 = @(x, y) sqrt(x.^2 + y.^2)/A;
```

a następnie dwukrotnie ją wywołamy, zmieniając stałą A:

```
>> x = [3 4 5];
>> m1(x(1),x(2))
ans =
    2.5000
>> A = 5;
>> m1(x(1),x(2))
ans =
    2.5000
```

Jak widzimy, wartość stałej A w definicji funkcji nie została zmieniona.

1.7.5 Argumenty funkcji

Z punktu widzenia semantyki języka, funkcja w MATLAB-ie każdorazowo otrzymuje/przekazuje wartości argumentów (wewnętrznie MATLAB optymizuje wszelkie zbędne operacje kopiowania).

Ilość argumentów w wywołaniu funkcji nie może być większa od ilości argumentów formalnych[1]. Przy wywołaniu, do funkcji zostają przekazane odpowiednie dane z wiersza wywołania i załadowane do lokalnych zmiennych funkcji w kolejności występowania na liście argumentów wejściowych. Jeśli argumentów aktualnych jest mniej, to nadmiarowe argumenty formalne nie zostaną zainicjowane. Funkcja wbudowana **nargin** określa, ile argumentów zostało przekazane do funkcji przy jej wywołaniu.

Ilość zwracanych argumentów określona jest w wierszu definicji funkcji. Dane, które mają być zwrócone przekazane zostają ze zmiennych występujących w liście argumentów wyjściowych do zmiennych występujących kolejno w wierszu wywołania. Wywołanie funkcji z liczbą argumentów wyjściowych mniejszą niż ilość argumentów formalnych powoduje, że nadmiarowe dane nie zostaną przekazane do programu wywołującego. Za pomocą funkcji można **nargout** określić ile argumentów wynikowych przewidziano w aktualnym wywołaniu.

Funkcje **nargin** oraz **nargout** występujące w kodzie *podfunkcji* lub *funkcji zagnieżdżonych* zwracają dane dotyczące danej funkcji, a nie funkcji pierwotnej.

[1] Istnieje możliwość wprowadzania zmiennej liczby argumentów aktualnych - omówiono to w rozdziale 1.7.6.

Weźmy jako przykład funkcję, która w łańcuchu *string* poszukuje znaku ogranicznika (dowolnego znaku z łańcucha *delimiters*), a następnie zwraca pierwszą część łańcucha (do ogranicznika) jako *token*, a w przypadku wywołania z dwoma argumentami wyjściowymi również pozostałą część łańcucha jako *remainder*. Jeśli łańcuch rozpoczyna się od ogranicznika, program poszukuje w łańcuchu pierwszego znaku niebędącego ogranicznikiem i od niego rozpoczyna dalsze działanie.

```
function [token, remainder] = strtok(string, delimiters)
% Funkcja wymaga co najmniej jednego argumentu wejściowego
if nargin < 1
    error('Za mało argumentów wejściowych.');
end
token = []; remainder = [];
len = length(string);
if len == 0
    return
end
% Jeżeli jeden argument, to ogranicznikami mogą być znaki
% o kodach 9..13 i 32 (spacja)
if (nargin == 1)
    delimiters = [9:13 32]; % Znaki ograniczników
end
i = 1;
% Pozycja pierwszego znaku niebędącego ogranicznikiem
while (any(string(i) == delimiters))
    i = i + 1;
    if (i > len), return, end
end
% Pozycja pierwszego ogranicznika
start = i;
while (~any(string(i) == delimiters))
    i = i + 1;
    if (i > len), break, end
end
finish = i - 1;
token = string(start:finish);
% Przy dwóch argumentach wyjściowych podaj pozostałą
% część łańcucha (remainder)
if (nargout == 2)
    remainder = string(finish+1:end);
end
```

Oto jakie będą wyniki działania tej funkcji przy różnych wywołaniach:

```
>> s = '1234*56 78';
>> a = strtok(s)% jeden argument wejściowy
a =
1234*56
>> [a,b] = strtok(s)% j.w., dwa argumenty wyjściowe
a =
1234*56
b =
 78
>> [a,b] = strtok(s, '*')% dwa argumenty WE i dwa WY
a =
1234
b =
*56 78
```

1.7.6 Przekazywanie argumentów opcjonalnych

Jeżeli funkcja może przyjmować zmienną ilość danych opcjonalnych, to jako argumentu wejściowego można użyć tablicy komórek **varargin**, jak to pokazano w przykładach poniżej:

```
function y = mfun(varargin)
% lub gdy x1 i x2 - dane obligatoryjne, reszta – opcjonalnie:
function y = mfun(x1, x2,   varargin)
```

Dane wejściowe będą podstawione do kolejnych komórek: **varargin{1}**, **varargin{2**}, i t.d.
Podobnie, jeżeli funkcja może zwracać zmienną ilość danych opcjonalnych, to jako argumentu wyjściowego można użyć tablicy komórek **varargout** używając jednego z przykładowych wierszy definicji:

```
function varargout = mfun(x1, x2,...)
% lub gdy y1 i y2 są obligatoryjne, reszta – opcjonalnie:
function [y1 y2   varargout] = mfun(x1, x2,   )
```

Dane wyjściowe należy wówczas przekazywać za pomocą **varargout{1}**, **varargout{2}**, i t.d.
Nazwy **varargin** i **varargout** są słowami kluczowymi i muszą być pisane małymi literami.

1.7.7 Klasa inputParser

Począwszy od wersji 7.0 MATLAB-a istnieje możliwość wykorzystania klasy **inputParser** i jej metod, do przetwarzania danych wejściowych funkcji. Instancja klasy **inputParser**, posiada następujące pola:

CaseSensitive	atrybut określający, czy w nazwach parametrów mają być rozróżniane wielkie i małe litery, domyślnie **false**.
FunctionName	nazwa identyfikująca funkcję, wykorzystywana np. w przypadku obsługi błędów.
StructExpand	atrybut określający, czy można wprowadzić argumenty wejściowe w postaci struktury zamiast w postaci listy, domyślnie **true**.
KeepUnmatched	atrybut określający, czy przechowywać wartości parametrów niezgodnych ze specyfikacją funkcji, wartość domyślna **false**.
Results	struktura przechowująca wartości uzyskane w wyniku ostatniego działania metody **parse** na danych wejściowych. Struktura ta posiada pola o nazwach zgodnych z zadeklarowanymi w metodach **addRequired**, **addOptional** oraz **addParamValue**,
UsingDefaults	tablica komórek, zawierająca nazwy parametrów, które nie wystąpiły w wywołaniu funkcji, w związku z czym otrzymały wartości domyślne,
Unmatched	struktura przechowująca wartości dla parametrów przesłanych w wywołaniu funkcji w postaci par *nazwa-wartość*, których nazwy nie występują w jej specyfikacji, jeśli atrybut *KeepUnmatched* ma wartość **true**. Nazwy pól struktury zgodne są z nazwami przekazanych a niewykorzystanych parametrów.
Parameters	tablica komórek zawierająca nazwy wszystkich parametrów umieszczonych w specyfikacji funkcji przez metody **addRequired**, **addOptional** oraz **addParamValue**.

Klasa *inputParser* posiada następujące metody:

- inputParser — konstruktor klasy.
  addRequired — dodanie wymaganego argumentu wejściowego do specyfikacji funkcji. Te argumenty mogą być przy wywołaniu przekazywane albo jako *wartości* (w kolejności wprowadzania do specyfikacji) albo jako pary *nazwa - wartość* (wówczas kolejność wprowadzania jest dowolna). Argumentami metody są: nazwa argumentu i funkcja sprawdzania poprawności wartości wejściowej.
- addOptional — dodanie parametru opcjonalnego wraz z wartością domyślną do specyfikacji funkcji. Te argumenty mogą być przekazywane jako *wartości* albo jako pary *nazwa - wartość*. Argumentami metody są: nazwa argumentu i funkcja sprawdzania poprawności wartości wejściowej.
- addParamValue — dodanie opcjonalnego parametru wejściowego do specyfikacji funkcji. Te parametry muszą być wprowadzane w wierszu wywołania funkcji wyłącznie jako pary *nazwa - wartość*. Argumentami metody są: nazwa argumentu, wartość domyślna i funkcja sprawdzania poprawności wartości wejściowej.

- createCopy — utworzenie kopii instancji klasy – zwykła instrukcja podstawienia tworzy nową referencję do obiektu, a nie jego duplikat.
- parse — przetworzenie danych wejściowych i wprowadzenie ich do odpowiednich pól instancji klasy (*Results, UsingDefaults, Unmatched*). Do metody przekazywana jest lista wartości argumentów wejściowych funkcji.

Po uruchomieniu konstruktora (**inputParser**) należy za pomocą metod **addRequired**, **addOptional** oraz **addParamValue** utworzyć specyfikację argumentów wejściowych funkcji. Nazwy tych argumentów zostają umieszczone w polu *Parameters* w kolejności alfabetycznej. Następnie należy uruchomić metodę **parse**, która przypisuje kolejne parametry z wiersza wywołania parametrom wymaganym funkcji w kolejności ich wprowadzania do specyfikacji. Ewentualne pozostałe parametry z wiersza wywołania przypisywane są argumentom opcjonalnym. Argumenty, którym nie przypisano wartości z wiersza wywołania, przybierają wartości domyślne. Wyniki działania tej metody pojawiają się w polu *Results*. Nierozpoznane parametry umieszczane są w polu *Unmatched*.

1.8 Wczytywanie i zapis danych

1.8.1 Instrukcje load i save

Do eksportowania i importowania przestrzeni danych programu służą funkcje **save** i **load**. Dane są przechowywane na dysku w plikach binarnych o rozszerzeniu **.mat.** Pliki te mają specjalny binarny format podwójnej precyzji, umożliwiający przenoszenie danych między komputerami o różnych formatach z zachowanie maksymalnej możliwej precyzji.

- Funkcja **save**

save *nazwa_pliku lista_zmiennych format*

Pominięcie listy zmiennych powoduje zapis wszystkich danych. Parametr format może przybierać wartości:

-append	dodawanie do istniejących danych
-ascii	format 8-bitowy ASCII
-ascii –double	format 16-bitowy ASCII
-ascii –tabs	separator: znak tabulacji
-mat	format binarny (domyślnie)

Przy formatach innych niż –mat należy podać pełną nazwę pliku (z rozszerzeniem).
Format –ascii stosuje się do przechowywania pojedynczych macierzy w postaci tekstowej. Każdy wiersz w pliku jest zapisem pojedynczego wiersza macierzy. W przypadku danych zespolonych, część urojona nie będzie zapisana. Każdy znak w zmiennej tekstowej zostanie przed zapisaniem przekształcony na odpowiadający mu kod. W pliku nie jest przechowywana żadna informacja dotycząca typu danych.
Format –append pozwala na dodanie nowych i aktualizację macierzy zachowanych w pliku, ale w macierzach istniejących aktualizowane są jedynie zachowane elementy.

- Funkcja **load**

load *format nazwa_pliku lista_zmiennych*.

Wczytanie danych z pliku. Pominięcie nazw zmiennych powoduje wczytanie wszystkich zmiennych zachowanych w pliku. Parametr format może przybierać wartości:

-mat	format binarny (domyślny)
-ascii	format tekstowy

Jeżeli rozszerzeniem nazwy pliku jest .mat, MATLAB próbuje odczytać plik jako binarny.
W przypadku niepowodzenia traktuje go jak plik tekstowy. Wszelkie inne rozszerzenia nazwy powodują traktowanie pliku jako tekstowego. Zawartość pliku tekstowego podstawiana jest na zmienną o nazwie zgodnej z nazwą pliku. Możliwe jest też użycie tej funkcji w postaci:
zmienna = **load**('–ascii', '*nazwa_pliku*') lub po prostu *zmienna* = **load**('*nazwa_pliku*')
W tym przypadku plik zostanie odczytany jako tekstowy obraz zmiennej. Każdy wiersz w pliku traktowany jest jako wiersz w macierzy.

1.8.2 Wczytywanie danych z plików tekstowych

W przypadku bardziej złożonej struktury zapisanych danych funkcja **load** nie zapewnia sukcesu wczytania danych. MATLAB dostarcza szeregu innych funkcji, które mogą mieć zastosowanie do wczytania danych zewnętrznych.

- **dlmread** – wczytywanie danych numerycznych, w których użyto separatorów danych innych niż odstęp lub przecinek.
- **textscan** – wczytywanie danych numerycznych opatrzonych nagłówkami. Wynik umieszczony jest w pojedynczej tablicy komórek.
- **textread** – wczytywanie danych numerycznych i znakowych do zadeklarowanych zmiennych

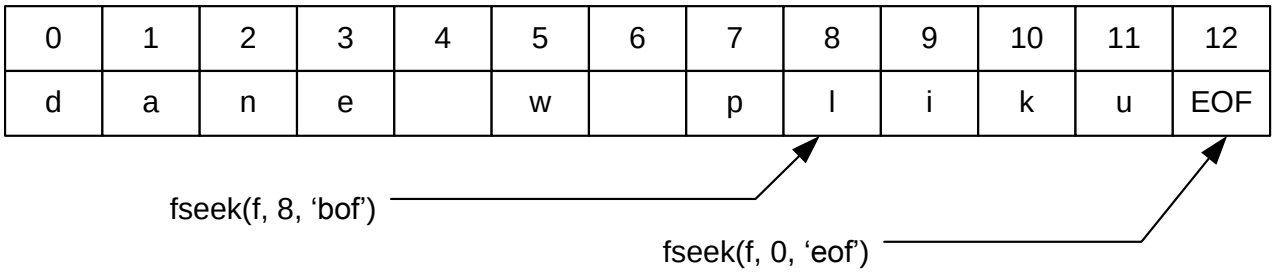
1.8.3 Używanie plikowych funkcji wejścia/wyjścia

Dane mogą być przechowywane w postaci plików binarnych lub formatowanych plików tekstowych. Operacja wczytywania bądź zapisu danych przy pomocy plikowych funkcji wejścia/wyjścia składa się z trzech etapów: otwarcia dostępu do pliku, wykonania operacji wejścia lub wyjścia oraz zamknięcia dostępu.

- Otwarcie dostępu do pliku.
  Do otwarcia dostępu do pliku używa się funkcji **fopen**(*nazwa_pliku*, *typ_dostępu*), oba argumenty są ciągami znaków. Typy dostępu mogą być następujące:
  - **r** - tylko do odczytu,
  - **w** - tylko do zapisu,
  - **a** - tylko do dołączania do pliku istniejącego,
  - **r+**- do zapisu i odczytu w istniejącym pliku.
  - **w+** - do zapisu i odczytu.
  - **a+**- zapis i odczyt, jeśli plik istnieje to dołączanie na końcu.

  Funkcja **fopen** może mieć dwa argumenty wyjściowe: identyfikator pliku i komunikat.
  W przypadku niepowodzenia zwracany identyfikator ma wartość **-1**, a komunikat informuje o charakterze błędu. Gdy operacja zakończy się sukcesem funkcja zwraca identyfikator pliku (będący liczbą całkowitą nieujemną), używany w operacjach czytania lub zapisu oraz pusty łańcuch komunikatu. Identyfikator **1** przypisany jest na stałe do wyjścia standardowego, a identyfikator **2** – do standardowej obsługi błędów. Funkcja **fopen** wywołana z jednym parametrem '**all**' zwraca wektor identyfikatorów wszystkich otwartych plików.
- Zamknięcie dostępu do pliku.
  Zamknięcie dostępu - funkcja **fclose** - może dotyczyć jednego pliku (wtedy argumentem funkcji jest identyfikator pliku) albo wszystkich plików - wtedy argument ma wartość **'all'**.
- Wykonanie operacji odczytu lub zapisu danych.
  Po otwarciu dostępu do pliku przy każdej operacji zarówno odczytu jak i zapisu, aktualizowany jest wskaźnik dostępu do pliku. Przesunięcie wskaźnika może być dokonane programowo za pomocą funkcji **fseek**. Funkcja ta ma trzy argumenty: identyfikator pliku, przesunięcie względne (bajty) i punkt odniesienia. Punkt odniesienia przyjmuje wartości:
  - 'bof' - początek pliku, liczbowa wartość parametru: -1,
  - 'cof' - pozycja bieżąca wskaźnika, liczbowa wartość parametru:0,
  - 'eof' - koniec pliku, liczbowa wartość parametru: 1.

  Bajty w pliku zawierającym n elementów ponumerowane są od 0 do n-1. Np. mamy n=12:



  Aktualne położenie wskaźnika (w bajtach) może być sprawdzane za pomocą funkcji **ftell.**

Do wczytywania danych binarnych używana jest funkcja **fread**. Argumentami tej funkcji są: identyfikator pliku, rozmiar wczytywanej macierzy (opcjonalnie, domyślnie - wektor kolumnowy wczytywany od bieżącej pozycji do końca pliku) oraz typ wczytywanych danych (opcjonalnie - domyślnie dane w postaci bajtów).
Drugi argument może przybierać formy:

- *n* - wczytanie n elementów do wektora kolumnowego
- inf - wczytywanie do końca pliku i umieszczanie w wektorze kolumnowym
- [*m,n*]- wczytanie elementów do wypełnienia macierzy m×n w kolumna po kolumnie, w przypadku niewystarczającej liczby elementów uzupełnianie zerami.

Trzeci z argumentów może przybierać postać:

- 'char' - dane znakowe (zwykle 8-bitowe),
- 'short' - liczby całkowite 16-bitowe,
- 'long' - liczby całkowite 32-bitowe,
- 'float' - liczby zmiennopozycyjne 32-bitowe,
- 'double' - liczby zmiennopozycyjne 64-bitowe.

Funkcja fread dopuszcza również inne, precyzyjnie określone typy, np. bajtowa liczba całkowita ze znakiem: 'int8'. Poniżej zamieszczono prosty przykład użycia wyżej wymienionych funkcji:

```
>> f = fopen('myfile.dat','r');
>> A = fread(f);                  % Wczytanie pliku do macierzy
                                  % (wektora kolumnowego) A
>> fseek(f,0,'bof');              % Wskaźnik odczytu na początek
>> B = fread(f,25);               % Wczytanie pierwszych 25 liczb
>> fseek(f,-16,'eof');
>> C = fread(f,[4 4]);            % Wczytanie ostatnich 16 liczb
                                  % do macierzy 4×4
>> fclose(f);
>>
```

W przypadku, gdy format zapisu do pliku był inny niż bajtowy, należy określić format w instrukcji odczytu, w przeciwnym przypadku odczyt będzie w formacie domyślnym (bajtowy bez znaku). Dane przechowywane są domyślnie jako elementy 8 bajtowe (podwójna precyzja). Typ macierzy wynikowej można zadeklarować w trzecim argumencie funkcji. W tym przypadku argument ma postać *format_czytania=>format_przechowywania*. Jeżeli oba formaty są jednakowe, można użyć skróconego zapisu **format* oznaczającego *format=>format*. W przypadku plików zawierających dane sformatowane w rekordach o stałej długości pól, możemy w funkcji **fopen** wprowadzić czwarty argument - *pomiń*, oznaczający ilość elementów z pliku, które mają być pominięte przy wczytywaniu. W tym przypadku format danych może zawierać ilość powtórzeń w postaci *N*format_czytania*. Załóżmy, że w pliku myfile.dat mamy dane w postaci 8 bitowych liczb całkowitych w grupach po 4, oddzielanych dwoma bajtami kontrolnymi. Wczytanie tych danych do macierzy A i ich konwersja na liczby 16 bitowe może się odbyć jak w poniższym przykładzie:

```
>> f = open('myfile.dat','r');
>> A = fread(f,[4 5],'4*int8=>int16',2);
>> whos A
  NameSize            Bytes      Class
  A   4×5                40      int16 array
Grand total is 20 elements using 40 bytes
>>
```

Wczytywanie wierszy tekstu z pliku może się odbywać za pomocą funkcji **fgetl** i **fgets**. Obie funkcje powodują wczytanie z pliku ciągu wartości (traktowanych jako ciąg znaków) od bieżącej pozycji do najbliższego znaku zakończenia wiersza (LF - kod 10 lub CR - kod 13). Różnica między tymi dwiema funkcjami polega na tym, że funkcja **fgetl** pomija znaki końca wiersza w wynikowym ciągu znaków:

```
>> f = fopen('myfile.dat','r');
>> A = fread(f)'
A =
  65    66    67    13    10    97    98    99    10
>> fseek(f,0,'bof');
>> B = fgetl(f)
```

```
B =
ABC
>> int8(B)
ans =
  65    66    67
>> C = fgets(f)
C =
abc
>> int8(C)
ans =
  97    98    99    10
>> fclose(f);
>>
```

Innym sposobem jest użycie funkcji [A,B,C…] = **textread(***filename,format,N***).** Funkcja ta wczytuje dane z pliku *filename* do zmiennych A, B, C i t.d. aż do osiągnięcia końca pliku (lub zadaną ilość danych). Funkcja ta jest przydatna w przypadku, gdy wczytujemy tekst z plików o znanym formacie, a dane przechowywane są w postaci wektorów. Można używać w przypadku rekordów o stałej długości lub z zadanymi delimiterami. Zadany *format* zgodny jest z konwencjami funkcji **fscanf** języka C.

MATLAB również oferuje funkcję **fscanf**, która umożliwia czytanie sformatowanych danych ASCII (działanie jej jest podobne jak w języku C, tylko wynikiem działania jest macierz). Argumentami funkcji **fscanf** są identyfikator pliku, format i rozmiar macierzy wynikowej. Format może przybierać wartości m.in.:

- '%s' - wczytaj łańcuch znaków,
- '%d' - wczytaj liczbę całkowitą zapisaną w układzie dziesiętnym,
- '%g' - wczytaj liczbę zmiennoprzecinkową podwójnej precyzji,
- '%*f' - pomiń liczbę zmiennoprzecinkową,
- '%*d' - pomiń liczbę całkowitą.

Funkcja pobiera elementy zgodnie ze wzorcem z pliku wejściowego od pozycji bieżącej aż do końca lub do osiągnięcia zadanej liczby elementów macierzy wynikowej. Zatrzymanie funkcji następuje również w wyniku znalezienia znaków niezgodnych z wzorcem. Załóżmy, że w pliku tekstowym mamy zapisane dane w postaci wierszy o długości 29 znaków (ze znakami CR i LF), zawierających interesujące nas dane liczbowe i dane tekstowe do pominięcia:

```
12345678901234567890123459789
Cena:  25.30zł ilość: 10szt
      321.15zł         2szt
```

Następujący program wczyta dane do tabeli, zamieni wiersze z kolumnami a następnie doda kolumnę zawierającą iloczyn danych z kolumny pierwszej i drugiej:

```
>> f = fopen('mydata.txt','r');
>> A = fscanf(f,'%*5c%f%*9c%f%*5c',[2 inf])';
>> A(:,3) = A(:,1)*A(:,2);
>>
```

Zapis danych do pliku binarnego odbywa się za pomocą funkcji **fwrite.** Funkcja ta pozwala na zapisanie elementów macierzy do pliku w zadanym formacie. Elementy macierzy brane są w kolejności zgodnej z indeksowaniem liniowym macierzy. Argumentami funkcji są: identyfikator pliku, dane do zapisu oraz opcjonalnie format zapisu (domyślnie - bajtowo bez znaku). Funkcja zwraca ilość wysłanych elementów.

```
>> f = fopen('MyData.dat', 'w');
>> count = fwrite(f, ones(3), 'short')
count =
  9
>> fclose(f)
ans =
  0
>>
```

Zapis sformatowanych danych tekstowych do pliku zapewnia funkcja **fprintf.** Jednym z argumentów tej funkcji jest format wyprowadzanych danych. Weźmy jako przykład zapis i odczyt tabelki tekstowej z wartościami funkcji wykładniczej:

```
>> x = 0:0.1:1;
>> y = [x;exp(x)];
>> f = fopen('MyData.txt','w');
>> fprintf(f,'Funkcja wykładnicza\n\n');
>> fprintf(f,'%6.2f %12.8f\n',y);
>> fclose(f);

>> f = fopen('Mydata.txt','r');
>> title = fgetl(f);              %pobrano wiersz nagłówka
>> [A,count] = fscanf(f,'%f %f',[2 11]);
>> A = A';                              %transpozycja
>> status = fclose(f);
>>
```

2 Grafika w programie MATLAB

MATLAB dostarcza środowiska pomagającego w tworzeniu wykresów dwu- i trójwymiarowych. Funkcje wbudowane w programie pozwalają na operowanie różnymi typami wykresów, dobieranie parametrów wykresów i ich skali, wprowadzanie opisów, legend i tekstów i ostatecznie zapisanie rysunku do pliku w formacie mapy bitowej..

2.1 Grafika dwuwymiarowa

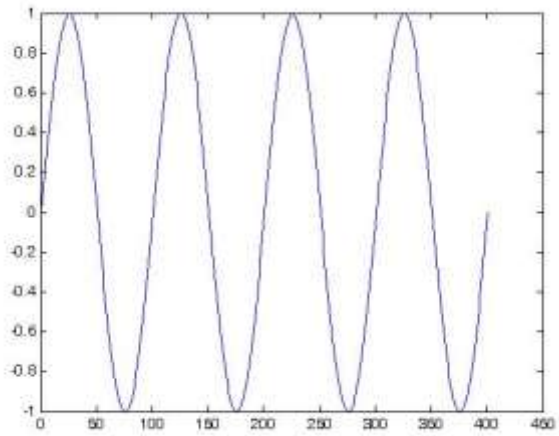
W programie MATLAB mamy szereg funkcji pozwalających na graficzne przedstawienie wektora danych jako linii, np.:

- **plot** - wykres z dwiema osiami liniowymi,
- **loglog** - wykres, w którym obie osie są logarytmiczne,
- **semilogx** - wykres z logarytmiczną skalą osi x i liniowa skalą osi y,
- **semilogy** - wykres z liniową skalą osi x i logarytmiczną osi y,
- **plotyy** - wykres z dwiema różnymi osiami y.
- **fplot** - wykres zadanej funkcji wbudowanej lub zewnętrznej (z m. pliku)

2.1.1 Wykonywanie prostych wykresów

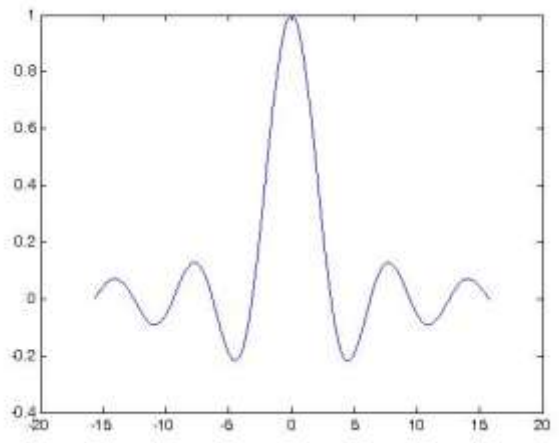
Funkcja **plot** daje różne rezultaty w zależności od argumentów wejściowych. Przykładowo, jeżeli y jest wektorem, to plot(y) utworzy wykres we współrzędnych liniowych zależności wartości kolejnych elementów wektora y od indeksu liniowego tych elementów.

```
>> t = 4*[-pi:pi/100:pi]; y = sin(t); plot(y);
>>
```



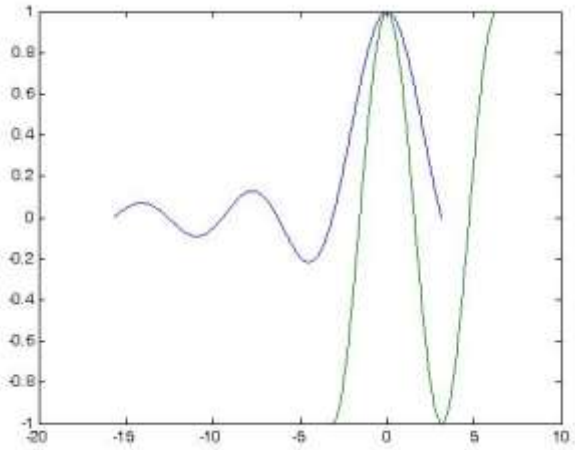
W przypadku podania jako argumentów dwóch wektorów (o tej samej liczbie elementów) otrzymamy wykres zależności jednego wektora w funkcji drugiego.

```
>> t = -5*pi:pi/100:5*pi; y = sin(t)./t; plot(t, y);
>>
```



MATLAB automatycznie dobiera dla osi odpowiednie zakresy wartości oraz podziałkę. Wykorzystując funkcję **plot** można otrzymać wykresy kilku funkcji we wspólnej skali (skala dobierana jest automatycznie stosownie do maksymalnych i minimalnych wartości wektorów na wykresie). Każdy z wykresów jest wykonany linią o innym kolorze.

```
>> t = -5*pi:pi/100:pi;
>> z = sin(t)./t;
>> x = -pi:pi/100:2*pi;
>> y = cos(x);
>> plot(t,z,x,y);
>>
```

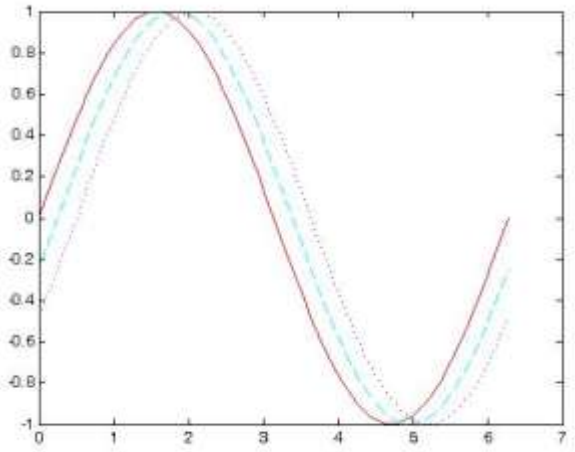


Poszczególnym liniom na wykresie można nadać różne właściwości, przesyłając odpowiednie argumenty przy wywołaniu funkcji **plot**. Argumentami tymi mogą być:

- rodzaj linii (ciągła '-', kreskowa '--', punktowa ':' i inne) - parametr 'LineStyle',
- rodzaj znaczników wartości ('+', 'o', 'x', '**s**quare', '**d**iamond', '**p**entagram', '<', '>' i t.p.) - parametr 'Marker',
- kolor linii ('r', 'g', 'b', 'w', 'c', 'm', 'y', 'k') - parametr 'LineWidth'.

Łańcuch specyfikacji może zawierać (lub nie) każdy z tych składników w dowolnej kolejności.

```
>> t = 0:pi/100:2*pi;
>> y = sin(t);
>> y1 = sin(t-0.25);
>> y2 = sin(t-0.5);
>> plot(t,y,'-r',t,y1,'c--',t,y2,':');
>>
```



Specyfikację można uzupełnić o dodatkowe parametry linii i znaczników:

- 'LineWidth' - grubość linii (w pkt),
- 'MarkerEdgeColor' - kolor krawędzi znacznika w przypadku znaczników wypełnianych,
- 'MarkerFaceColor' - kolor wypełnienia znacznika,
- 'MarkerSize' - rozmiar znacznika (w pkt).

2.1.2 Specjalne funkcje do tworzenia wykresów

MATLAB umożliwia wykorzystanie gotowych funkcji tworzących wykresy w specjalizowanych formatach:

- **fplot** - tworzenie wykresu zadanej funkcji: **fplot(**'*funkcja*',' *ograniczenia*'**)** wykonuje wykres *funkcji* zadanej jako
  - nazwy funkcji (wbudowanej lub zewnętrznej), np. w pliku cx.m:

```
function y = cx(t)
   y = sin(t)./t;
end
```

```
>> fplot('cx',[-10 10]*pi);
```

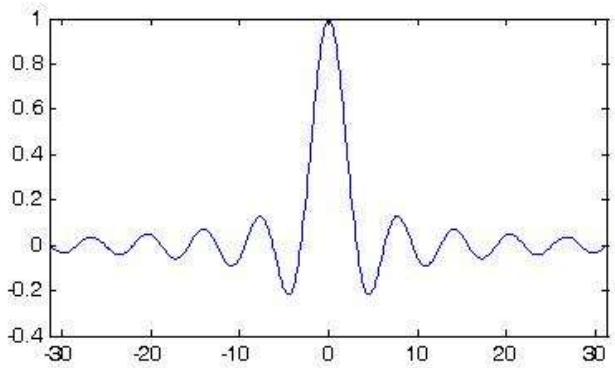
  - przepisu dla funkcji **eval:**

```
>> fplot('sin(x)./x',[-10 10]*pi);
```

  - lub uchwytu do funkcji anonimowej:

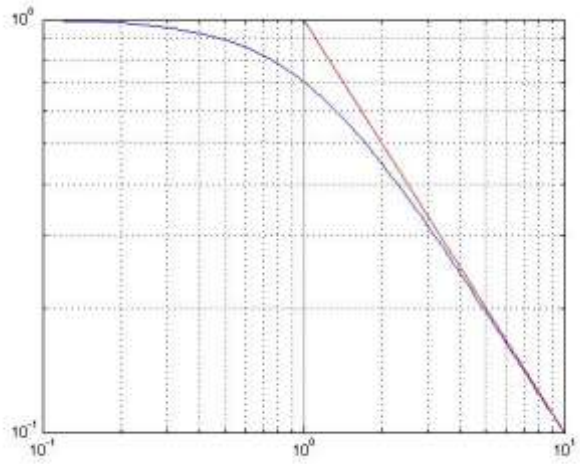
```
>> cx = @(x) sin(x)/x;
>> fplot(cx,[-10 10]*pi);
>>% lub fplot(@(x) sin(x)./x,[-10 10]*pi);
```

  Parametr *ograniczenia* jest dwuelementowym wektorem zawierającym skrajne wartości zmiennej niezależnej lub czteroelementowym, zawierającym ograniczenia obu osi wykresu.



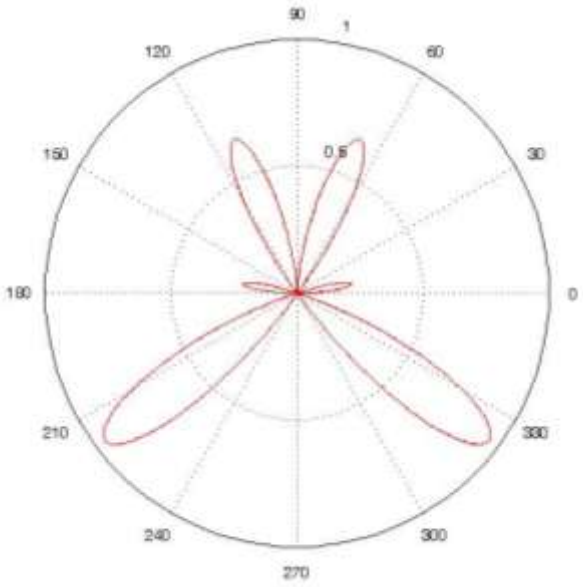
- **loglog, semilogx, semilogy** - wykonanie wykresu z obiema lub jedną z osi w skali logarytmicznej.

```
>> x = 0.1*(1:100);
>> y = 1./sqrt(1+x.^2);
>> loglog(x,y)
>> hold all
>> plot([0.1 1 10],[1 1 0.1],'r')
>> grid
>>
```



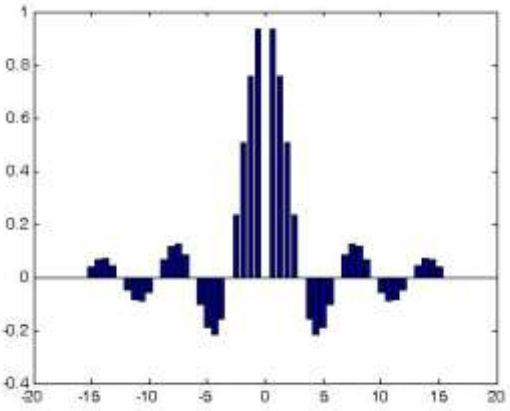
- **polar** - tworzenie wykresu w układzie biegunowym.

```
>> t = (-1:0.01:1)*pi;
>> polar(t,sin(2*t)*cos(5*t),'--r')
>>
```

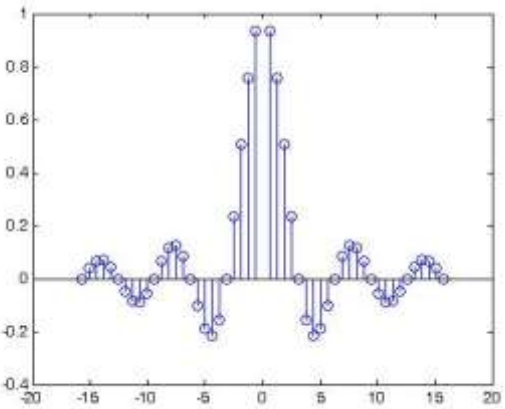


- **bar, stem, stairs** - wykresy dla danych dyskretnych.

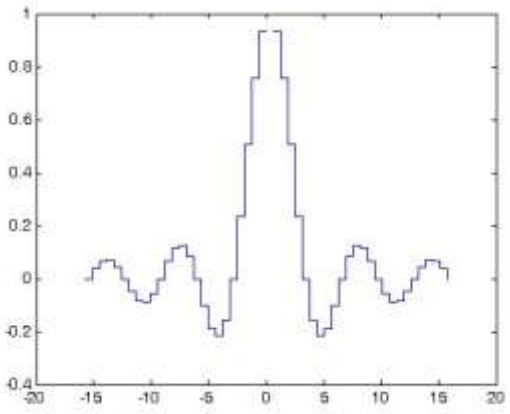
```
>> t = (-5:.2:5)*pi;
>> y = sin(t)./t;
Warning: Divide by zero
>> bar(t,y)
```




```
>> stem(t,y)
```

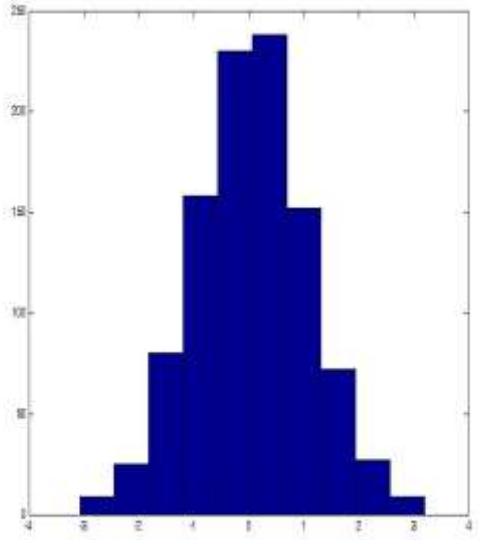


```
>> stairs(t,y)
```

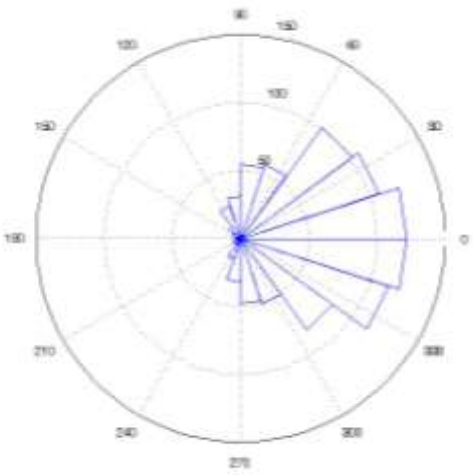


- **hist, rose** - wykonywanie histogramów (w układzie współrzędnych prostokątnych - **hist** lub biegunowych - **rose**).

```
>> y = randn(1,1000);
>> hist(y)      % -> Rys. a
```



Rys. a.



Rys. b.

```
>> y = y/max(y)*pi;
>> rose(y)       % -> Rys.b
```

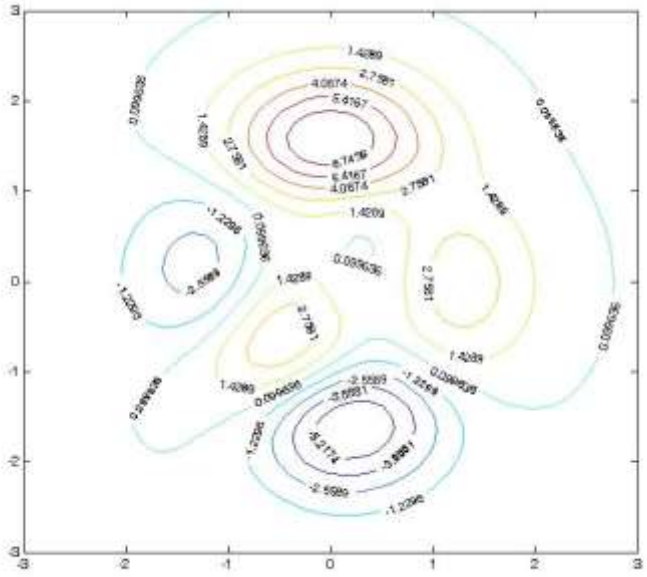
2.1.3 Wykresy izoliniowe

MATLAB daje możliwość utworzenia wykresów izoliniowych, w których argumentem wejściowym jest macierz traktowana jako wysokości punktów w stosunku do płaszczyzny. Dostępne są funkcje:

- **contour** - wykres linii łączącej punkty o jednakowej wysokości, generowany na podstawie wartości elementów macierzy Z. Można zadać ilość poziomów linii albo wektor wysokości, dla których zostaną wykonane linie,
- **clabel** - funkcja nanosząca etykiety wartości na izoliniach,
- **contourf** - wykres izolinii z wypełnieniem kolorami powierzchni między liniami.

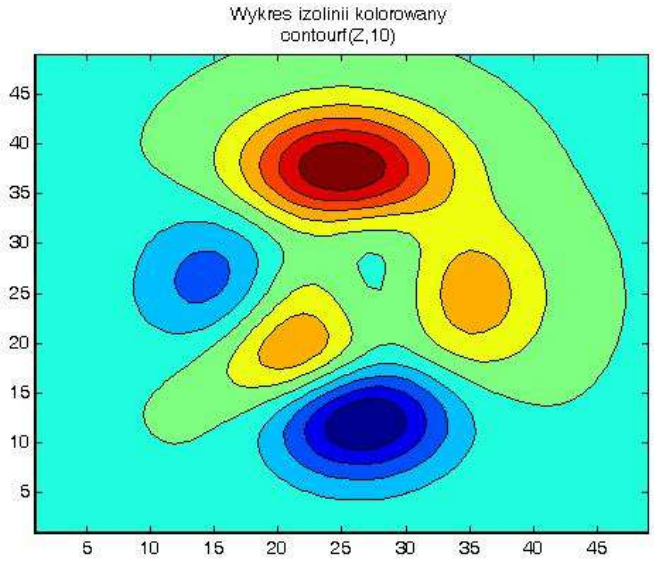
Na przykład wykonajmy wykres izoliniowy dla macierzy generowanej przez funkcję peaks:

```
>> [X,Y,Z] = peaks;
>> [C,h] = contour(X,Y,Z,10);
>> clabel(C,h,'FontSize',8)
>>
```



Ten sam wykres z kolorowaniem powierzchni (funkcja **contourf**):

```
>> Z = peaks;
>> [C,h] = contourf(Z,10);
>> title({'Wykres izolinii kolorowany','contourf(Z,10)'})
```



2.1.4 Dodatkowe operacje na wykresach

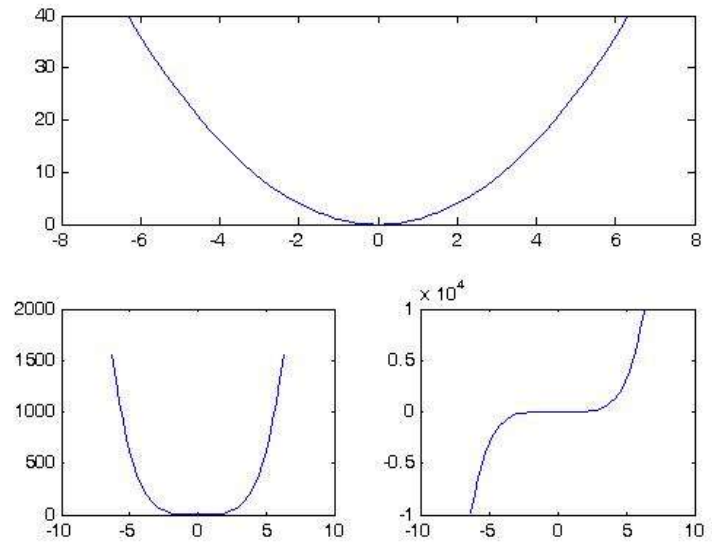
Wykres wykonany przy pomocy którejś z funkcji rysujących może zostać uzupełniony innymi elementami. Służą do tego funkcje:

- **grid** - wprowadzenie lub ukrycie na wykresie siatki prostokątnej. Funkcja może mieć argumenty:
  - **grid on** - wprowadzenie siatki głównej,
  - **grid off** - ukrycie siatki,
  - **grid minor** - wprowadzenie lub ukrycie siatki pomocniczej,
- **axis** - ustawienie lub pobranie parametrów osi bieżącego wykresu. Funkcja może mieć argumenty:
  - **axis on/off** - wyświetlenie lub ukrycie osi,
  - **axis equal** - zrównanie jednostek na obu osiach,
  - **axis square** - wykres na obszarze kwadratowym,

- **axis normal** - usunięcie działania **equal** i **square**,
- **axis auto/manual** - włączenie lub wyłączenie automatycznego doboru podziałki na osiach,
- **axis state** - zwraca informację o ustawieniach osi. Zalecane jest raczej używanie metody zwracającej parametry bieżącej instancji axis: **get(gca,…)**
- **axis([x_{min} x_{max} y_{min} y_{max}])** - określenie ręczne zakresów osi. Zastąpienie wybranych parametrów (np. x_{min} lub x_{max}) stałymi **-inf** lub **inf** daje możliwość półautomatycznego określania zakresów osi.
- **axis xy/ij** - osie w układzie kartezjańskim lub "macierzowym".

- **hold** - włączenie lub wyłączenie "zamrożenia" rysunku, do umieszczenia na nim kolejnych wykresów. Funkcja może mieć argumenty:
  - **hold on** - włączony tryb dodawania nowych wykresów,
  - **hold off** - włączony tryb zastępowania istniejącego wykresu nowym.
- **figure(***h***)** - wybór rysunku o identyfikatorze *h* jako bieżącego. Jeśli rysunek nie istnieje to zostaje zainicjowany pusty bieżący rysunek.
- **axes**(*h*) - wybór układu współrzędnych o identyfikatorze *h*. Wywołana bez argumentu - utworzenie nowego układu współrzędnych (z ewentualnym uprzednim utworzeniem nowego rysunku).
- **subplot(***m,n,i***)** - podzielenie bieżącego rysunku siatką prostokątną *m*×*n* i aktywowanie elementu nr *i* (elementy liczone są kolejno wierszami).

```
>> x = -2*pi:pi/12:2*pi;
>> y = x.^2;
>> subplot(2,2,1:2); plot(x,y)
>> y = x.^4;
>> subplot(2,2,3); plot(x,y);
>> y = x.^5;
>> subplot(2,2,4); plot(x,y)
```



2.1.5 Nanoszenie opisów i objaśnień na wykresy

MATLAB dostarcza możliwości wprowadzania opisów i objaśnień do wykresów. Opisy mogą być zgodne z formatem T_EX lub La T_EX.

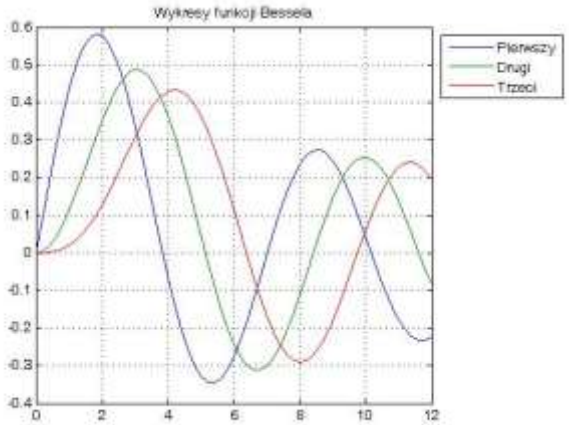
- **title(**'*string*'**)** - wprowadzenie tytułu na wykresie. Parametr 'string' może zawierać komendy sterujące formatowaniem tekstu:
  - **\bf** - czcionka wytłuszczona,
  - **\it** - czcionka pochyła,
  - **\rm** - czcionka normalna,
  - **^{…}** - indeks górny,

- _{…} - indeks dolny,
- **fontname(***fontname***)** - wybór czcionki,
- **fontsize(***fontsize***)** - określenie wielkości czcionki.

Tekst może zawierać znaki specjalne, litery greckie i symbole matematyczne zgodne z zapisem edytora T_EX, np. litera **α** - **alpha**, znak × - **times** i t.p.

- **xlabel(**'*string*'**)** - wprowadzenie opisu osi x,
- **ylabel(**'*string*'**)** - wprowadzenie opisu osi y,
- **text(***X*,*Y*,'*string*'**)** - wprowadzenie testu na wykresie w miejscu o współrzędnych (X,Y). Jeżeli X i Y są wektorami, to 'string' zostanie umieszczony we wszystkich punktach opisanych parami (x , y). W przypadku, gdy dodatkowo argument 'string' jest tablicą komórek o tej samej liczbie wierszy co wektory X i Y, to każda para (x,y) wskazuje na punkt umieszczenia odpowiedniego elementu z tej tablicy. Możliwe jest określenie sposobu wyrównania tekstu za pomocą dodatkowych parametrów:
  - **HorizontalAlignment** - 'Left', 'Center' lub 'Right',
  - **VerticalAlignment** - 'Bottom', 'Middle', 'Top', 'Cap' lub 'Baseline'.
- **gtext(**'*string*'**)** - wprowadzenie na wykresie tekstu w miejscu wskazanym myszką.
- **line(***X*,*Y***)** - wprowadzanie linii/łamanej/wielokąta. Wektor X zawiera współrzędne na osi x, a wektor Y - współrzędne na osi y kolejnych punktów na łamanej. Jeśli chcemy uzyskać linię zamkniętą należy powtórzyć współrzędne punktu początkowego.
- **fill(***X*,*Y*,*c***), patch(***X*,*Y*,*c***)** - wprowadzanie wielokątów wypełnionych zadanym (*c*) kolorem.
- **legend(**'*string1*','*string2*'...**)** - wprowadzanie pola legendy na bieżącym rysunku. Ilość łańcuchów powinna być równa ilości wykresów na rysunku, w przeciwnym przypadku nadmiarowe łańcuchy zostaną pominięte. Funkcja **plot** może być wywoływana z parametrami: **off** (usunięcie legendy), **hide/show** (ukrycie/ wyświetlenie legendy), **boxon/boxoff** (włączenie/wyłączenie obramowania legendy). Można określić dodatkowe atrybuty wyświetlania legendy, jak np. '*Location*' czy '*Orientation*'.

```
>> x = 0:.2:12;
>> plot(x,bessel(1,x),x,bessel(2,x),x,bessel(3,x));
>> legend('Pierwszy','Drugi','Trzeci','Location','NorthEastOutside')
>> grid
>> title('Wykresy funkcji Bessela')
```



2.1.6 Wykonywanie wykresów dla danych macierzowych

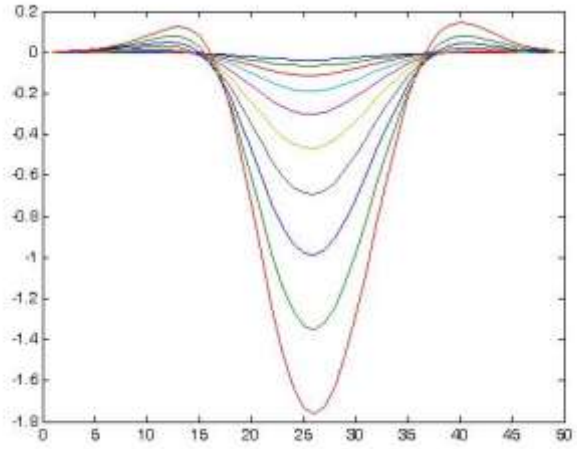
Przy najprostszym wywołaniu: **plot(A)**, gdzie A jest macierzą o wymiarach n×m, otrzymamy wykres zawierający m linii dla każdej z m kolumn macierzy. Na osi poziomej umieszczony zostanie indeks wierszy macierzy 1:n.

W ogólnym wywołaniu, gdy funkcja **plot** jest wywołana z dwoma argumentami X oraz Y, które mogą mieć więcej niż jeden wiersz lub kolumnę, mogą nastąpić przypadki:

- Jeżeli Y jest macierzą a x jest wektorem, **plot(x,Y)** tworzy wykres zależności wierszy lub kolumn Y w funkcji wektora x. Orientacja wykresu zależy od tego, czy wymiar wektora x

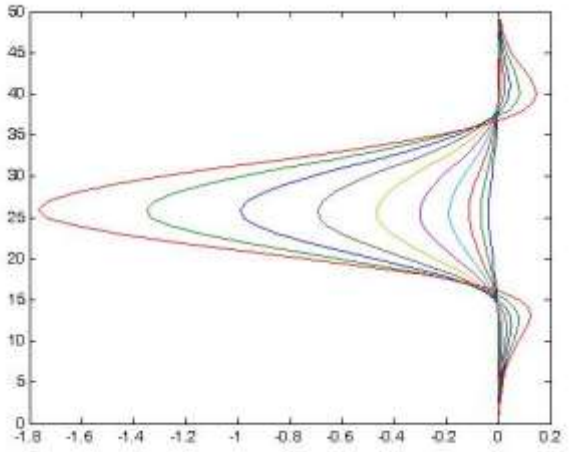
odpowiada liczbie kolumn czy liczbie wierszy macierzy Y. W przypadku kwadratowej macierzy Y wykonywany jest wykres kolumn.

```
>> Z = peaks;
>> Z = Z(:,1:10);
>> y = 1:length(peaks);
>> plot(y,Z)
```



- Jeżeli X jest macierzą a y wektorem, **plot(X,y)** tworzy wykres wierszy lub kolumn macierzy X wg wektora y. Np. w zastosowaniu do poprzedniego przykładu uzyskamy obrócenie wykresu o 90°.

```
>> Z = peaks;
>> Z = Z(:,1:10);
>> y = 1:length(peaks);
>> plot(Z,y)
```



- Jeżeli zarówno X jak i Y są macierzami, o tych samych wymiarach, to w wyniku wykonania polecenia **plot(X,Y)** tworzony jest wykres zależności kolumn macierzy X od wierszy macierzy Y.

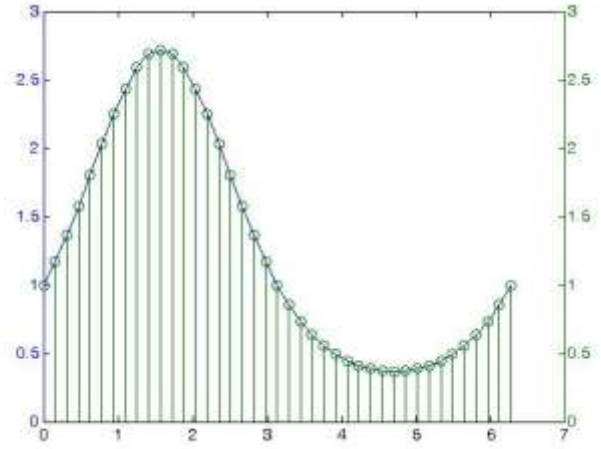
Możliwe jest umieszczenie na jednym wykresie zależności większej liczby par macierzy.
W przypadku, gdy elementy macierzy przyjmują wartości zespolone, na wykresie pomija się część urojoną. Wyjątkiem jest przypadek, gdy występuje pojedynczy argument zespolony. Wtedy zapis **plot(Z)** jest równoznaczny z **plot(real(Z),imag(Z))**. W przypadku większej ilości macierzy trzeba w sposób jawny użyć ich części rzeczywistych i urojonych.

2.1.7 Wykresy z podwójną skalą na osi y

Funkcja **plotyy(***x1*,*y1*,*x2*,*y2***)** umożliwia wykonanie na wspólnym rysunku dwóch wykresów, dla y1(x) i y2(x) z podwójną skalą na osi y, dobraną odpowiednio dla każdej z funkcji.

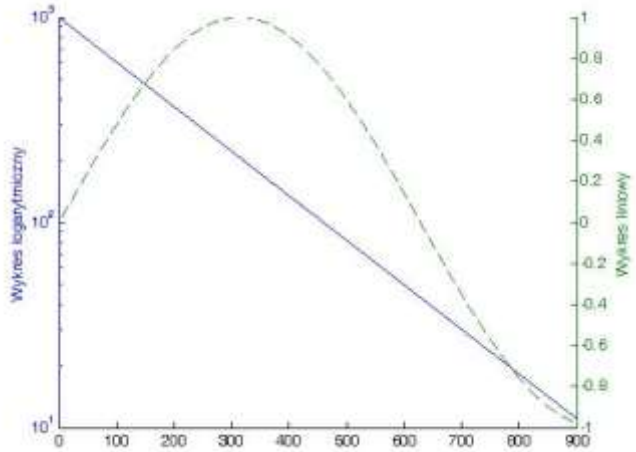
W wywołaniu funkcji **plotyy** możliwe jest podanie dodatkowo parametrów określających jakie funkcje mają być użyte do wykonania poszczególnych wykresów.

```
>> t = 0:pi/20:2*pi;
>> y = exp(sin(t));
>> plotyy(t,y,t,y,'plot',@stem)
>>
```



Za pomocą funkcji **plotyy** można również połączyć na wspólnym rysunku wykres w skali liniowej i logarytmicznej.

```
>> t = 0:900; A = 1000; alpha = 0.005; beta = 0.005;
>> z1 = A*exp(-alpha*t);
>> z2 = sin(beta*t);
>> [h l1 l2] = plotyy(t,z1,t,z2,'semilogy','plot');
>> axes(h(1))
>> ylabel('Wykres logarytmiczny')
>> axes(h(2))
>> ylabel('Wykres liniowy')
>> set(l2,'LineStyle','--')
```



MATLAB umożliwia umieszczenie we wspólnym okienku wielu wykresów obok siebie (za pomocą funkcji **subplot**) lub na wspólnym wykresie, w tym samym układzie (funkcja **hold)** lub niezależnych układach współrzędnych (funkcja **axes**).

2.2 Praca z mapami bitowymi

2.2.1 Typy obrazów i sposoby ich przechowywania

MATLAB przechowuje obrazy w postaci macierzy (tablic dwuwymiarowych), w których każdy element zawiera atrybuty wyświetlania pojedynczego piksela. Indeksy elementu odpowiadają położeniu piksela na obrazie. W pewnych przypadkach (obrazy RGB) konieczne jest użycie tablic o trzech wymiarach, odpowiadających kolejno kolorom: czerwonemu, zielonemu i niebieskiemu. Podstawowym sposobem przechowywania danych w MATLAB-ie jest double - 64-bitowy zapis zmiennopozycyjny. Jednak dla przechowywania obrazów, w celu zmniejszenia użycia pamięci stosowane są przede wszystkim klasy tablic uint8 lub uint16, wymagające 8 lub 16 bitów dla przechowania jednego elementu.
MATLAB używa trzech podstawowych typów obrazów[2]

- obrazy indeksowane
- obrazy w skali szarości (jaskrawości)
- obrazy RGB albo pełnokolorowe (true color)

Obrazy indeksowane przechowywane są w postaci macierzy danych ***X*** i palety ***map*.** Paleta jest macierzą m×3 klasy double, zawierającą wartości zmiennoprzecinkowe z przedziału [0 1]. Każdy wiersz z tej macierzy zawiera informację o składowych czerwonej, zielonej oraz niebieskiej pojedynczego koloru. Obraz indeksowany używa bezpośredniego odwołania pikseli do pozycji z palety. Obraz indeksowany można wyświetlić za pomocą instrukcji:

```
>> image(X); colormap(map)
```

W przypadku, gdy elementy macierzy ***X*** są klasy uint8 lub uint16, przy określaniu indeksu macierzy map ich wartość jest zwiększana o 1 (wartość 0 wskazuje wiersz nr 1 z macierzy map).

Obraz w skali szarości jest to macierz danych ***I***, której elementy reprezentują jaskrawość obrazu w określonej skali. Elementy mogą być klasy double, uint8 lub uint16. Obraz taki można wyświetlić za pomocą funkcji **image**, w sposób identyczny jak dla obrazów indeksowanych, ale należy zdefiniować paletę odcieni szarości za pomocą funkcji **gray** (w tej palecie wszystkie składowe barwne będą miały ten sam poziom)

```
>> image(I); colormap(gray(256))
```

Do prezentacji obrazu można również użyć funkcji **imagesc** umożliwiającej wprowadzenie tablicy zakresu wartości jaskrawości w której będzie wyświetlany obraz:

```
>> imin = 128; imax = 255;
>> imagesc(I,[imin imax]); colormap(gray)
```

Pierwszy z elementów tablicy zakresu wskazuje na pierwszy wiersz w macierzy mapy kolorów a drugi - na wiersz ostatni.
Obrazy indeksowane i w skali szarości mogą być przechowywane w plikach .bmp.

Obraz RGB (truecolor) jest przechowywany jako tablica ***RGB*** o wymiarach m×n×3, zawierająca dane definiujące składowe czerwoną, zieloną i niebieską koloru poszczególnych pikseli obrazu. Tablica może zawierać dane klasy double, uint8 lub uint16. Do wyświetlenia obrazu używamy polecenia:

```
>> image(RGB)
```

Obrazy RGB mogą być przechowywane w plikach .bmp, .jpg, .gif, .png i innych.
W przypadku plików graficznych w formacie TIFF dane przechowywane są w tablicy ***CMYK*** o wymiarach m×n×4. Funkcja **image** nie obsługuje takich tablic bezpośrednio.

[2] Biblioteka Image Processing Toolbox wyróżnia dodatkowo obrazy dwukolorowe (czarno-białe), które przechowywane są w postaci tablic klasy *logical*.

2.2.2 Wczytywanie, zapis i wyświetlanie obrazów

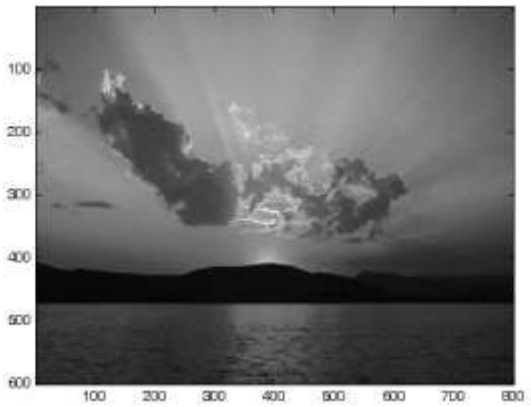
Większość plików zawierających obrazy rozpoczyna się od nagłówka, zawierającego informacje charakterystyczne dla danego formatu, a następnie dane w postaci ciągu bajtów. Z tego powodu nie da się w prosty sposób użyć standardowych funkcji we/wy - **load** i **save** - do wczytywania bądź zapisu obrazów. Matlab dostarcza szeregu specjalizowanych funkcji do pracy z obrazami

- **imread** - wczytywanie danych obrazu z pliku (BMP, TIFF, JPG, GIF, PNG itp.). W zależności od formatu pliku obraz jest przechowywany w postaci 8 bitowej lub 16 bitowej . Dla obrazów indeksowanych wczytywane są dane i paleta. Do palety zawsze stosowany jest format podwójnej precyzji (64 bitowy).
- **imwrite** - zapis danych obrazu do pliku w wybranym formacie. Domyślnym ustawieniem dla danych jest format uint8. W przypadku zapisu do plików w formacie PNG lub TIFF można użyć formatu uint16. W tym przypadku należy wprowadzić argument 'BitDepth' o wartości 16.
- **imfinfo** - wyświetlenie informacji o pliku graficznym (zależnie od typu pliku):
  - nazwa pliku
  - rozmiar pliku
  - wymiary obrazu w pikselach
  - format graficzny
  - numer wersji
  - ilość bitów na piksel
  - typ obrazu (truecolor, grayscale lub indexed)
  - ...
- **load/save** - mogą być stosowane do bezpośredniego zapisu/wczytywania macierzy danych i palety w standardowym formacie MATLAB-a.
- **image** - funkcja wyświetlająca obraz w bieżącym oknie rysunku,
- **imagesc** - funkcja wyświetlająca, przeskalowująca dane do współpracy z paletą. Stosowana najczęściej do wyświetlania obrazów monochromatycznych.
- **colormap** - funkcja zmieniająca paletę barw bieżącego rysunku. Argumentem jest nazwa m-pliku zawierającego paletę. Z reguły pliki (m-funkcje) zawierające palety dostosowują rozmiar palety do rozmiaru aktualnej palety bieżącego rysunku.

Jako przykład wczytajmy plik 'zachód słońca.jpg'. Jest to plik zawierający obraz RGB. Przekształcimy następnie ten obraz na postać indeksowaną monochromatyczną.

```
>> sun = imread('zachód słońca.jpg');
>> sun_mono = 0.2989*sun(:,:,1)+0.5870*sun(:,:,2)+0.1140*sun(:,:,3);
>> whos sun sun_mono
  Name         Size       Bytes     Class
  sun        600x800x3  1440000     uint8 array
  sun_mono   600x800     480000     uint8 array
>> figure; image(sun)                                %Rysunek a
>> figure; image(sun_mono); colormap(gray(256))  %Rysunek b
```

Rysunek a.



Rysunek b.

2.3 Grafika trójwymiarowa

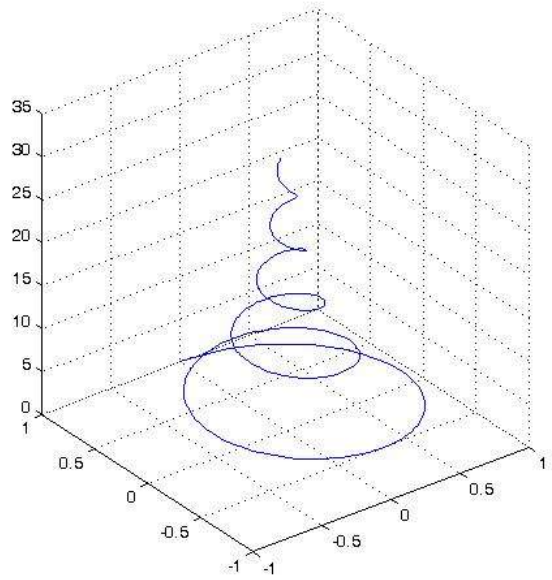
2.3.1 Wykresy liniowe danych trójwymiarowych

Do wykonywania wykresów liniowych służy funkcja **plot3**. Jeżeli x, y i z są wektorami o jednakowej długości, to w wyniku realizacji funkcji

```
>> plot3(x,y,z)
```

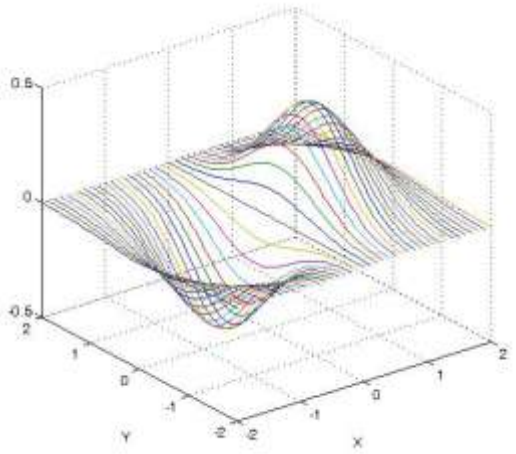
otrzymamy wykres w postaci rzutu na ekran trójwymiarowej linii łączącej kolejne punkty. Np.:

```
>> t = 0:pi/50:10*pi;
>> a = exp(-0.1*t); plot(a.*sin(t),a.*cos(t),t)
>> axis square; grid on
```

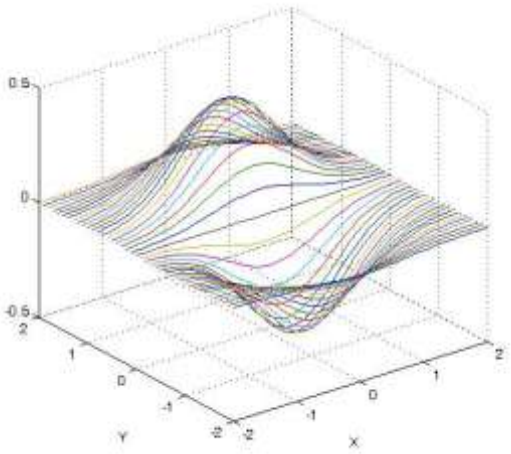


Jeżeli argumentami funkcji **plot3** są macierze o tych samych rozmiarach, to w wyniku działania funkcji otrzymamy wykresy dla kolejnych kolumn tych macierzy.

```
>> [X,Y] = meshgrid([-2:.1:2]);
>> Z = X.^exp(-X.^2-Y.^2);
>> figure; plot3(X,Y,Z); grid on; xlabel('X'); ylabel('Y') %Rysunek a
>> figure; plot3(Y,X,Z); grid on; xlabel('X'); ylabel('Y') %Rysunek b
>>
```



Rysunek a.



Rysunek b.

2.3.2 Przedstawienie macierzy jako powierzchni

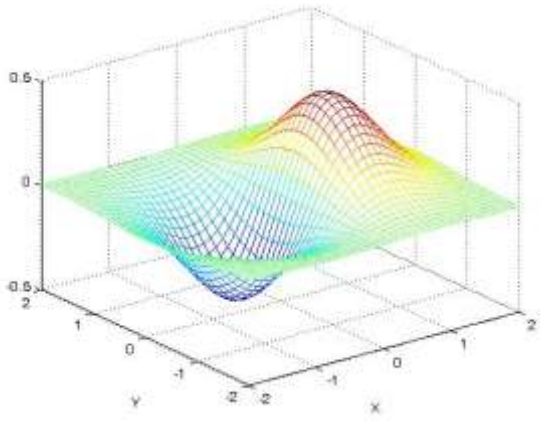
W przypadku, gdy należy przedstawić bardzo duże ilości danych, zobrazowanie tabelaryczne lub w postaci szeregu linii, obrazujących kolumny macierzy (przekroje funkcji dwóch zmiennych) jest

niewygodne. MATLAB oferuje wówczas przedstawienie wyników jako powierzchni powstałej przez połączenie sąsiadujących punktów.
MATLAB pozwala wykonywać różne rodzaje wykresów w postaci siatki (**mesh**), w której kolorowane są linie powstałe z połączenia sąsiadujących punktów lub powierzchni (**surf**), w której są kolorowane zarówno linie jak i wypełnienie między nimi. Kolory reprezentują wartości danych.

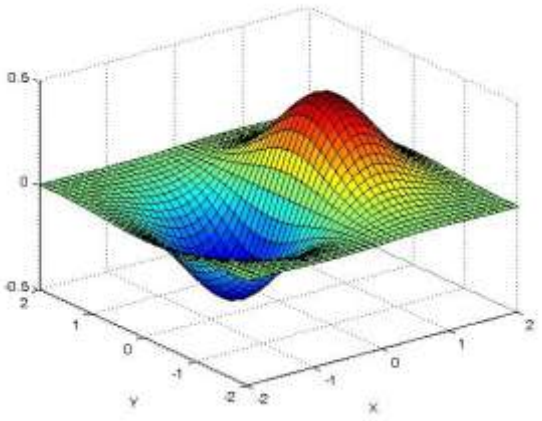
- **mesh** - zobrazowanie powierzchni w postaci kolorowanej siatki.
- **surf** - zobrazowanie powierzchni w postaci zestawu kolorowych czworokątów otoczonych ramkami siatki **mesh**. Ramki te można usunąć ustawiając wartość atrybutu EdgeColor na none lub za pomocą polecenia **shading flat.** Można również zastosować interpolację kolorów za pomocą polecenia **shading interp** (lub ustawiając wartość atrybutu 'FaceColor' na wartość 'interp') - uzyskując w ten sposób efekt gładkich przejść tonalnych kolorów.
- **meshc, surfc** - przedstawienie powierzchni wraz z wykresem warstwicowym poniżej,
- **meshz** - przedstawienie powierzchni w postaci nieprzezroczystej siatki,
- **pcolor** - przedstawienie płaskie, w którym kolory punktów reprezentują dane,
- **surfl** - zobrazowanie powierzchniowe z zewnętrznym oświetleniem. Kolorów używa się do uzyskania efektu oświetlenia powierzchni,
- **surface** - funkcja niskiego poziomu, do tworzenia obiektów graficznych (jak **line** w przypadku dwuwymiarowym).

Poniżej przedstawiono różne sposoby przedstawienia danych:

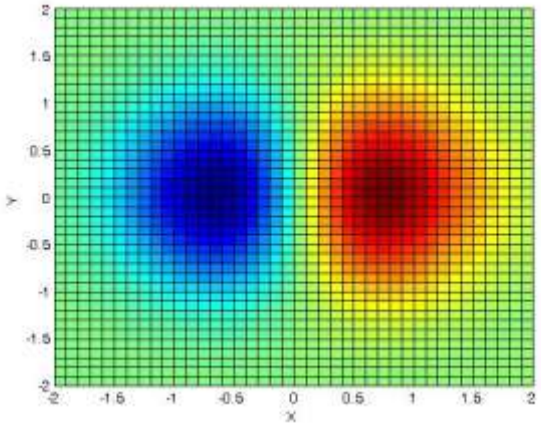
```
>> [X,Y]=meshgrid([-2:.1:2]);
>> Z=X.*exp(-X.^2-Y.^2);
>> figure; mesh(X,Y,Z); xlabel('X');ylabel('Y')    %Rysunek a
>> figure; surf(X,Y,Z); xlabel('X');ylabel('Y')    %Rysunek b
>> figure; meshz(X,Y,Z); xlabel('X');ylabel('Y')   %Rysunek c
>> figure; pcolor(X,Y,Z); xlabel('X');ylabel('Y') %Rysunek d
>>
```



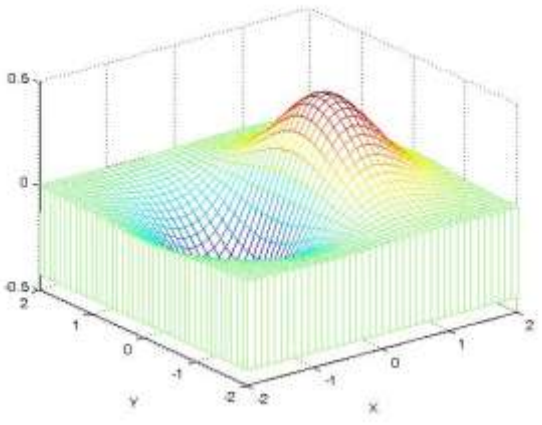
Rysunek a.



Rysunek b.



Rysunek c.



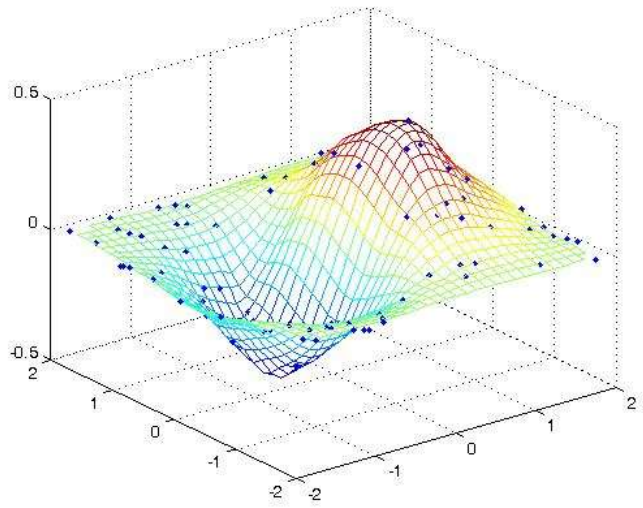
Rysunek d.

Do przedstawienia wartości funkcji dwu zmiennych przydatne jest użycie funkcji **meshgrid,** która generuje macierz wartości zmiennych x i y równomiernie rozłożonych w zadanym zakresie:

meshgrid(x_{min}:dx:x_{max}, y_{min}:dy:y_{max})

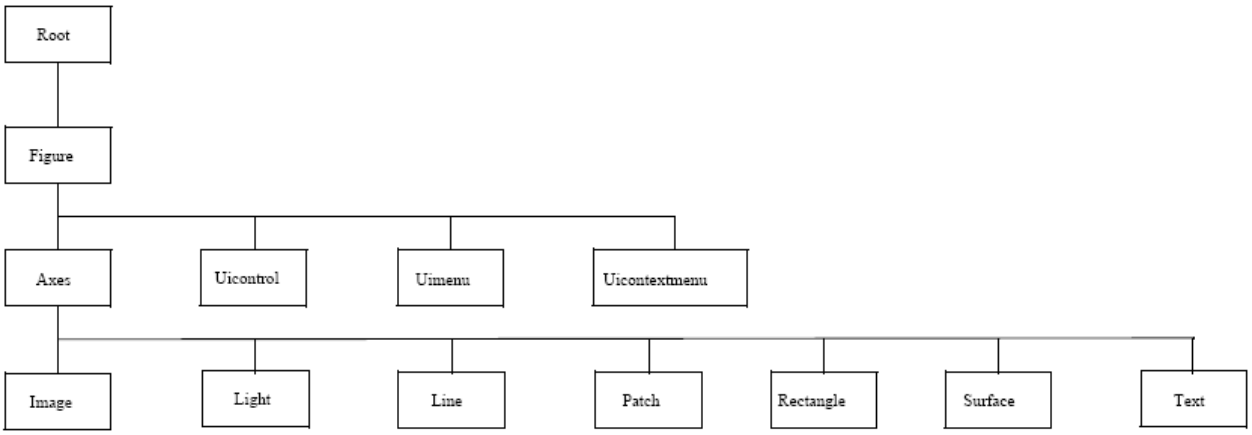
W przypadku, gdy punkty nie były rozłożone równomiernie, możliwe jest użycie funkcji **griddata** do uzyskania macierzy interpolowanej w siatce równomiernej.

```
>> x = rand(100,1)*16-8;
>> y = rand(100,1)*16-8;
>> z = x.*exp(-x.^2 - y.^2);
>> xl = linspace(min(x),max(x),33);
>> yl = linspace(min(y),max(y),33);
>> [X,Y]= meshgrid(xl,yl);
>> Z = griddata(x,y,z,X,Y,'cubic');
>> mesh(X,Y,Z)
>> axis tight; hold on
>> plot3(x,y,z,'.','MarkerSize',10)
>>
```



2.4 Hierarchia obiektów graficznych w MATLAB-ie

Przy uruchomieniu funkcji wykonującej wykres, MATLAB tworzy rysunek przy użyciu różnych obiektów graficznych, takich jak okno rysunku (**figure**), system współrzędnych (**axes**), linie obrazujące dane (**line**), napisy (**text**) i t.p.



Za każdym razem, kiedy MATLAB tworzy nowy obiekt graficzny, do obiektu przywiązywany jest uchwyt. Najważniejszymi z tych obiektów są:

- **figure** - okna zawierające paski narzędziowe, menu itp.

- **axes** - dostarcza ramy (układ współrzędnych), w których będą przedstawione dane,
- **line** - reprezentujące dane przesłane do funkcji **plot**,
- **text** - etykiety i znaczniki na osiach, tytuł i napisy na rysunku,

Dostępne są funkcje zwracające uchwyty do obiektów lub atrybuty obiektów:

- **get** - podaj wartości atrybutów obiektu, argumentem jest uchwyt do obiektu
- **gcf** - zwraca uchwyt do bieżącego okna rysunku, odpowiednik wyrażenia:

```
>> get(0,'CurrentFigure'),
```

- **gca** - zwraca uchwyt do bieżącej instancji axes, odpowiednik wyrażenia:

```
>> get(get(0,'CurrentFigure'),'CurrentAxes'),
```
czyli `get(gcf,'CurrentAxes'),`

- **gco** - zwraca uchwyt do bieżącego obiektu, odpowiednik wyrażenia:

```
>> get(get(0,'CurrentFigure'),'CurrentObject'),
```
czyli `get(gcf,'CurrentObject'),`

- **set** - zmień wartości atrybutów obiektu.

Wszystkie obiekty tworzone są z domyślnym zestawem wartości atrybutów. Wartości te można zadać w dwojaki sposób:

- specyfikując wybrane atrybuty i ich wartości przy wywołaniu konstruktora obiektu
- zmieniając funkcją **set** wartości atrybutów istniejącego obiektu.

Przy tworzeniu grafiki w plikach skryptów lub funkcji można napotkać problemy:

- funkcja zamaże dane na istniejącym bieżącym rysunku.
- bieżące okno rysunku może zachowywać się w sposób nieoczekiwany przez program.

Przydaje się wówczas używanie uchwytu do obiektu bazowego. Np. wyrażenie:

```
>> hfig = figure('Name','Wykres funkcji y(x)');
>> axes('Parent',hfig,...)
```

powoduje utworzenie obiektu **axes** w aktualnie utworzonym oknie rysunku o uchwycie *hfig*.
Przy użyciu funkcji **set** można uzyskać wykaz wszystkich możliwych atrybutów obiektu:

```
>> h = plot(x,y)
>> set(h)
ans =
                Color: {}
           EraseMode: {4x1 cell}
           LineStyle: {5x1 cell}
           LineWidth: {}
              Marker: {14x1 cell}
          MarkerSize: {}
...
```

lub wszystkich możliwych wartości wybranego atrybutu (w nazwie atrybutu nie są rozróżniane małe i wielkie litery):

```
>> set(h,'Marker')
[ + | o | * | . | x | square | diamond | v | ^ | > | < | pentagram |
hexagram | {none} ]
>> set(h,'linestyle')
[ {-} | -- | : | -. | none ]
```

Można także wyjście funkcji **set** przywiązać do zmiennej otrzymując w wyniku strukturę:

```
>> a=set(h);
>> a.EraseMode
ans =
    'normal'
    'background'
    'xor'
    'none'
```

W tym przypadku oczywiście MATLAB rozróżnia wielkie i małe litery w nazwie zmiennej.
Można znaleźć uchwyt do obiektu poszukując go na podstawie atrybutów. W tym celu należy użyć funkcji **findobj**. Np. chcemy zmienić atrybut 'HorizontalAlignment' określonego tekstu:

```
>> text(5,7,'y(5) \rightarrow')
>> h = findobj('String','y(5) \rightarrow');
>> set(h,'HorizontalAlignment','Right')
```

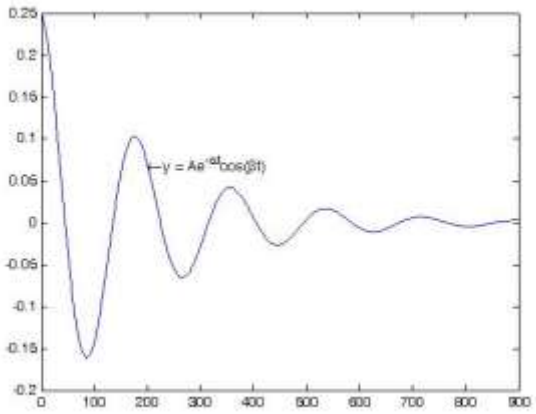
Taki zapis spowoduje przeszukiwanie całego drzewa obiektów graficznych, poczynając od **root.**
Można przyspieszyć ten proces (w przypadku, gdy jest wiele powołanych obiektów graficznych)

podając obiekt w hierarchii, od którego należy rozpocząć przeszukiwanie. W tym przypadku można rozpocząć przeszukiwanie od bieżącego układu współrzędnych **axes**:

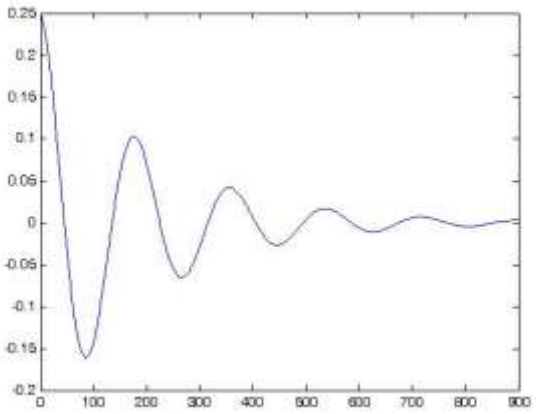
```
>> h = findobj(gca,'String','y(5) \rightarrow');
```

Funkcja **copyobj** daje możliwość utworzenia kopii obiektu (wybranego za pomocą uchwytu). Chcąc usunąć jakiś obiekt, należy ustalić jego uchwyt, a następnie użyć funkcji **delete**.

```
>> t=0:900;y=.25*exp(-.005*t).*cos(pi*t/90);
>> plot(t,y);
>> text(200,y(201),'\leftarrowy = Ae^{-\alphat}cos(\betat)')
>> % ->Rysunek a
>> fh = get(0,'Children');
>> get(fh,'Type')    %obiekty pochodne root
ans =
figure
>> ah = get(fh,'Children');
>> get(ah,'Type')    %obiekty pochodne okna
ans =
axes
>> h = get(ah,'Children'); %można było: h = get(gca,'Children')
>> get(h,'Type')     %obiekty związane z bieżącym układem wsp.
ans =
    'text'
    'line'
>> delete(h(1))      %lub delete(findobj(gca,'Type','text'))
>> % ->Rysunek b
```



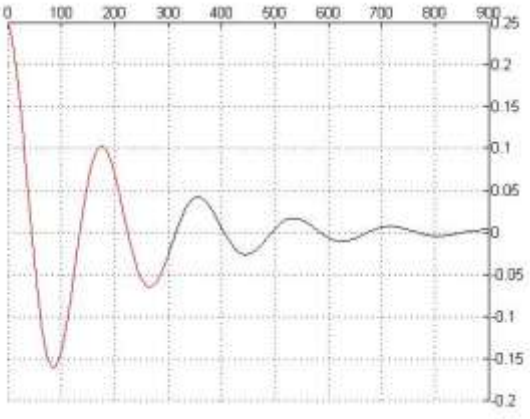
Rysunek a.



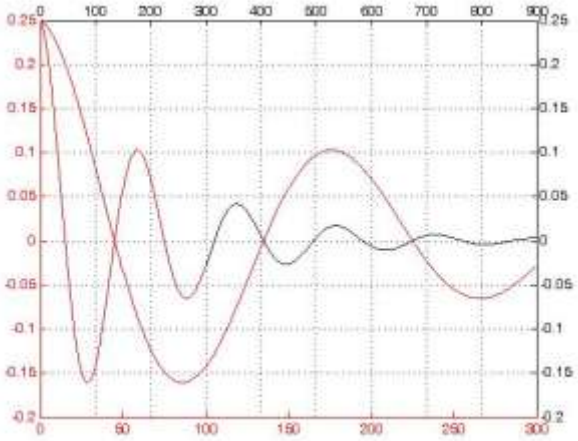
Rysunek b.

Przy użyciu uchwytów do obiektów graficznych można na jednym rysunku zdefiniować kilka różnych układów współrzędnych (**axes**) i w każdym z układów wykonać niezależny wykres (np. **line**). Na przykład spróbujmy umieścić w jednym oknie rysunku wykres funkcji czasu oraz wykres rozciągniętego jej fragmentu początkowego, używając uchwytów do obiektów.

```
>> figure; axis([0 900 -0.2 0.25]);
>> hold all; hl01=plot(t(1:300),y(1:300),'r');
>> hl02=plot(t(300:900),y(300:900),'k');hold off
>> set(gca,'XAxisLocation','top','YAxisLocation','right')
>> grid                % ->Rysunek a
>> hx2=axes('Position',get(hx1,'Position'),'Color','none',...
'XColor','r','YColor','r');
>> hl1=line(t(1:300),y(1:300),'Color','r','Parent',hx2); % ->Rysunek b
>>
```



Rysunek a.



Rysunek b.

2.5 Atrybuty podstawowych obiektów graficznych

Zestaw atrybutów obiektu graficznego można podzielić na dwie grupy. W pierwszej z nich znajdują się podstawowe atrybuty wspólne dla wszystkich obiektów (niektóre z nich dla pewnych obiektów, jak np. **root** nie mają zastosowania), a w drugiej atrybuty charakterystyczne dla danego obiektu. Obiekty pochodne dziedziczą wartości atrybutów od obiektów nadrzędnych.
Uwaga: opisane poniżej zestawy atrybutów dotyczą MATLAB-a w wersji 7.1. Atrybuty występujące w poprzednich wersjach mogą się różnić od wymienionych poniżej.

2.5.1 Atrybuty wspólne

BeingDeleted : [on | off] - wskaźnik, czy została wywołana funkcja DeleteFcn , atrybut tylko do odczytu,

BusyAction: [queue | cancel] - sposób reakcji na próbę przerwania pracy procedury przez inną procedurę. Jeżeli wartość jest *cancel*, to nowe zdarzenie nie jest obsługiwane. W przeciwnym przypadku, w zależności od atrybutu Interuptible zdarzenie może zostać umieszczone w kolejce i uruchomione we właściwej kolejności. Zdarzenia Delete, Create oraz związane z oknem zdarzenia Close i Resize dokonują przerwania niezależnie od wartości atrybutu Interutpible.

ButtonDownFcn: [łańcuch | uchwyt do funkcji | tablica komórek] - procedura (wpisana w postaci wyrażenia, nazwy m-pliku lub wskaźnika do funkcji) uruchamiana po kliknięciu myszą na obiekcie lub w jego bezpośrednim (5 pikseli) otoczeniu. W zależności od atrybutu Enable aktywny jest tylko prawy przycisk myszy (*on*) lub oba (*off*).

Children - wektor zawierający uchwyty do obiektów pochodnych (pod warunkiem, że nie są one ukryte),

Clipping: [on | off] – przycinanie obiektów pochodnych przy wyświetlaniu, istotne tylko dla obiektów **axes**,

CreateFcn: [łańcuch | uchwyt do funkcji | tablica komórek] - procedura uruchamiana przy powoływaniu instancji obiektu, już po ustaleniu jego atrybutów. W procedurze można używać uchwytu do obiektu, zwracanego przez funkcję gcbo. Istnieje możliwość zdefiniowania

domyślnej funkcji CreateFcn, wykorzystywanej przy tworzeniu nowych instancji (nie ma zastosowania do obiektu **root**),

DeleteFcn: [łańcuch | uchwyt do funkcji | tablica komórek] - procedura uruchamiana przy kasowaniu obiektu, tj. gdy uruchomione zostanie polecenie **delete** albo **close** w stosunku do okna aplikacji zawierającego dany element (nie ma zastosowania do obiektu **root**),

HandleVisibility: [on | callback | off] - poziomy dostępu do obiektu.

- Uchwyty są zawsze widoczne, gdy atrybut ma wartość *on*.
- Ustawienie wartości *callback* powoduje, że uchwyty są widziane przez procedury obsługi lub funkcje przez nie wywoływane, ale niedostępne dla funkcji wywoływanych z wiersza poleceń, co daje zabezpieczenie przed ingerencją użytkownika.
- Ustawienie wartości *off* powoduje niedostępność uchwytów w każdych okolicznościach.

Uchwyt do ukrytych obiektów można odzyskać ustawiając atrybut ShowHiddenHandles dla obiektu **root**,

HitTest: [on | off] - sprawdzenie, czy można wybrać dany obiekt przez kliknięcie,

Interruptible: [on | off] – tryb pracy procedury obsługi zdarzeń. Uwzględnia ona:

- jaki jest atrybut Interruptible obsługiwanego obiektu,
- czy w wykonywanej procedurze są użyte funkcje drawnow, figure, getframe, pause lub waitfor,
- jaki jest atrybut BusyAction obiektu oczekującego na uruchomienie procedury.

Jeśli przerywającą procedurą jest DeleteFcn, CreateFcn lub CloseRequest czy ResizeFcn okna, to przerwanie nastąpi niezależnie od wartości atrybutu Interruptible. Przerywająca procedura wystartuje przy najbliższym wyrażeniu **drawnow**, **figure**, **getframe**, **pause** lub **waitfor**.

Parent - uchwyt do obiektu bazowego. Przykładowo dla **root** jest to [], dla **figure** jest to obiekt **root** o identyfikatorze 0, dla **axes** – uchwyt do **figure** i t.d.,

Selected: [on | off] - informacja, czy obiekt jest aktualnie wybrany,

SelectionHighlight: [on | off] - czy obiekt ma wizualizować że jest wybrany,

Tag - nadana przez użytkownika etykietka identyfikująca obiekt,

Type: [root | figure | axes | uicontrol | uimenu | uicontextmenu | uipanel | uitable | image | light | line | patch | rectangle | surface | text] - typ obiektu, atrybut tylko do odczytu,

UIContextMenu - uchwyt do menu kontekstowego (pojawiającego się przy kliknięciu prawym klawiszem myszy),

UserData - dowolne dane związane z obiektem, nie wykorzystywane przez MATLAB.

Visible: [on | off] - czy dany obiekt ma być wyświetlany na ekranie. Obiekty niewidoczne w dalszym ciągu są dostępne do programowania. Wstępne ustawienie obiektów jako niewidocznych może przyspieszyć uruchomienie aplikacji.

2.5.2 Atrybuty obiektu "root"

CallbackObject - uchwyt do obiektu, którego metoda **callback** jest aktualnie wykonywana,

CommandWindowSize: [wiersze kolumny] - rozmiary okna rozkazowego,

CurrentFigure - uchwyt do ostatnio utworzonego lub ostatnio wybranego (figure(h) lub set(0,'CurrentFigure',h) okna rysunku,

Diary: [on | off] - włączenie rejestracji historii wykonywanych rozkazów,

DiaryFile - nazwa pliku historii,
Echo: [on | off] - włączenie/wyłączenie echa,
FixedWidthFontName - nazwa czcionki o stałych odstępach,
Format: [short | shortE | long | longE | bank | hex | + | rat] - format wyprowadzania liczb,
FormatSpacing: [compact | loose] - odstępy w pionie,
Language - wybór języka systemowego
MonitorPositions: [x y szerokość wysokość] - parametry ekranu monitora w pikselach,
More: [on | off | n] - aktywacja funkcji more przy wyświetlaniu okna rozkazowego,
PointerLocation: [x y] – współrzędne kursora w oknie, liczone od dolnego lewego rogu,
PointerWindow - uchwyt do okna w którym jest kursor (tylko do odczytu),
RecursionLimit - głębokość zagnieżdżenia wywołań m-funkcji,
ScreenDepth - ilość bitów do opisu koloru pojedynczego piksela,
ScreenPixelsPerInch - rozdzielczość obrazu,
ScreenSize - rozmiar ekranu (tylko do odczytu),
ShowHiddenHandles: [on | off] - wyświetlanie obiektów ukrytych,
Units: [pixels | normalized | inches | centimeters | points | characters] - jednostki miary dla atrybutów PointerLocation i ScreenSize,

2.5.3 Atrybuty obiektu "figure"

Wiele z wymienionych poniżej atrybutów okna wykorzystywanych jest wyłącznie przy tworzeniu graficznego interfejsu użytkownika.

Alphamap - wektor mający wpływ na sposób wyświetlania obiektów typu surface, image lub patch,
BackingStore: [{on} | off] - obsługa bufora przyspieszającego odświeżanie okna,
CloseRequestFcn: [łańcuch | uchwyt do funkcji | tablica komórek]- funkcja uruchamiana przy zamykaniu okna,
Color - kolor tła, podany jako wektor [R G B] lub predefiniowana nazwa koloru
Colormap - paleta RGB (macierz m×3 elementowa, m<=256),
CurrentAxes - uchwyt do bieżącego układu współrzędnych. Nowy układ współrzędnych powołuje się za pomocą instrukcji: axes(uchwyt) lub set(gcf, 'CurrentAxes', uchwyt),
CurrentCharacter - ostatnio naciśnięty klawisz,
CurrentObject - uchwyt do obiektu wybranego w oknie kliknięciem myszy. Gdy klikniemy na obiekt ukryty, zwracana jest pusta macierz,
CurrentPoint - współrzędne ostatniego kliknięcia w oknie,
DockControls: [on | off] - wskaźniki dokowania okna na pulpicie programu,
DoubleBuffer: [on | off] - bufor usuwający migotanie przy prostych animacjach w oknie,
FileName - nazwa pliku .fig zawierającego definicję okna (przy tworzeniu interfejsu graficznego),
IntegerHandle: [on | off] - uchwyty do okien w postaci liczb całkowitych (on) lub zmiennoprzecinkowych,
InvertHardcopy: [on | off] - zmiana koloru tła dla potrzeb wydruku,
KeyPressFcn: [łańcuch | uchwyt do funkcji | tablica komórek] - funkcja uruchamiana przez kliknięcie w oknie,
MenuBar: [none | figure] - włączenie paska menu wyświetlanego wraz z oknem,
MinColormap - liczba określająca minimalną ilość kolorów w palecie,
Name - nazwa okna, wyświetlana na pasku,
NextPlot: [new | add | replace | replacechildren] - sposób wyświetlania nowego elementu graficznego,

NumberTitle: [on | off] - pokazywanie identyfikatora okna w jego tytule,
PaperUnits: [inches | centimeters | normalized | points]
PaperOrientation: [portrait | landscape | rotated]
PaperPosition [lewy dół szerokość wysokość] - parametry prostokąta zawierającego rysunek na wydruku,
PaperPositionMode: [auto | manual]
PaperSize - wymiary aktualnie wybranego (PaperType) rodzaju papieru,
PaperType: [usletter | uslegal | A0 | A1 | A2 | … | B0 | … | tabloid | custom]
Pointer: [arrow | ibeam | … | circle | cross | fleur | custom | hand] - kształt kursora graficznego na rysunku,
PointerShapeCData - macierz 16×16 zawierająca mapę bitową rysunku kursora,
PointerShapeHotSpot - wektor 4 elementowy opisujący aktywne pole kursora,
Position - położenie okna na ekranie,
Renderer: [painters | zbuffer | OpenGL | None] - metoda użyta do wyświetlania grafiki,
RendererMode: [auto | manual] - automatyczny lub ręczny dobór metody wyświetlania,
Resize: [on | off] - dopuszczenie do zmiany rozmiarów okna za pomocą myszy,
ResizeFcn: [łańcuch | uchwyt do funkcji | tablica komórek] - funkcja wywoływana przy zmianie rozmiarów okna,
SelectionType: [normal | open | alt | extend] - informacja o rodzaju ostatniego kliknięcia myszy w oknie:

- normal - klawisz lewy,
- open - podwójne kliknięcie dowolnym klawiszem,
- alt - prawy klawisz lub Ctrl-lewy,
- extend - klawisz środkowy lub Shift-lewy,

ShareColors: [on | off] - nieaktualne,
ToolBar: [none | auto | figure] - określenie, czy ma być wyświetlany pasek narzędziowy,
Units: [inches | centimeters | normalized | points | pixels | characters] - jednostki używane przez atrybuty CurrentPoint i Position,
WindowButtonDownFcn: [łańcuch | uchwyt do funkcji | tablica komórek] - funkcja wywoływana przy naciśnięciu klawisza myszy,
WindowButtonMotionFcn: [łańcuch | uchwyt do funkcji | tablica komórek] - funkcja wywoływana przy poruszaniu myszą w oknie,
WindowButtonUpFcn: [łańcuch | uchwyt do funkcji | tablica komórek] - funkcja wywoływana przy zwolnieniu klawisza myszy,
WindowStyle: [normal | modal | docked] - typ okna,
WVisual: { 00 (RGB 32 GDI, Bitmap, Window) } - sposoby wyświetlania grafiki:
WVisualMode: [auto | manual] - tryb doboru sposobu wyświetlania grafiki,

2.5.4 Atrybuty obiektu "axes"

- dane ogólne

ActivePositionProperty: [position | outerposition] - określenie, czy przy zmiany stosują się do samych osi, czy też wraz z opisami i marginesami,
ALim - dwuelementowy wektor przeskalowywania grafiki (surface, patch, image) do rozmiarów okna,
ALimMode: [auto | manual] - dobór sposobu przeskalowywania,
AmbientLightColor - dodatkowe równomierne oświetlenie (tylko wraz z innymi źródłami światła),
Box: [on | off] - otoczenie układu współrzędnych ramką prostokątną (lub prostopadłościanem w przypadku 3D),
CameraPosition - parametry widoku
CameraPositionMode: [auto | manual]

CameraTarget
CameraTargetMode: [auto | manual]
CameraUpVector
CameraUpVectorMode: [auto | manual]
CameraViewAngle
CameraViewAngleMode: [auto | manual]
CLim - zakres kolorów, wektor dwuelementowy opisujący dopasowanie kolorów grafiki do palety,
CLimMode: [auto | manual] - tryb doboru zakresu kolorów,
Color - kolor tła,
ColorOrder - rotacja kolorów w wykresach wieloliniowych,
CurrentPoint - lokalizacja ostatniego kliknięcia w układzie,
DataAspectRatio - proporcje rysunku,
DataAspectRatioMode: [auto | manual] - tryb doboru proporcji rysunku,
DrawMode: [normal | fast] - tryb rysowania obiektu graficznego,
FontAngle: [normal | italic | oblique] - krój czcionki,
FontName - nazwa czcionki,
FontSize - wielkość czcionki,
FontUnits: [inches | centimeters | normalized | points | pixels] - jednostki dla FontSize,
FontWeight: [light | normal | demi | bold] - waga czcionki,
GridLineStyle: [- | -- | : | -. | none] - styl linii siatki,
Layer: [top | bottom] - położenie linii osi w stosunku do linii wykresu,
LineStyleOrder - rotacja stylów linii w wykresach wieloliniowych,
LineWidth - grubość linii osi,
MinorGridLineStyle: [- | -- | : | -. | none] - styl linii siatki pomocniczej,
NextPlot: [new | add | replace | replacechildren] - sposób wprowadzania nowego wykresu,
OuterPosition - zakres obszaru zawierającego osie i ich opisy,
PlotBoxAspectRatio - proporcje układu współrzędnych,
PlotBoxAspectRatioMode: [auto | manual] - sposób doboru proporcji układu,
Projection: [orthographic | perspective] - rzut 3D,
Position - zakres obszaru samych osi,
TickLength - długość znaczników na osiach,
TickDir: [in | out] - kierunek znaczników na osiach,
TickDirMode: [auto | manual] - tryb wyboru kierunku znaczników,
Title - tytuł,
Units: [inches | centimeters | normalized | points | pixels | characters] - jednostki dla Position,

- dane dotyczące poszczególnych osi układu współrzędnych: X, Y i Z,

XColor
XDir: [normal | reverse]
XGrid: [on | off]
XLabel
XAxisLocation: [top | bottom]
XLim
XLimMode: [auto | manual]
XMinorGrid: [on | off]
XMinorTick: [on | off]
XScale: [linear | log]
XTick
XTickLabel
XTickLabelMode: [auto | manual]

XTickMode: [auto | manual]
YColor …, ZColor ...

2.5.5 Atrybuty obiektu "line"

Color: - kolor linii,
DisplayName: - nazwa używana w legendzie wykresu,
EraseMode: [normal | none | xor | background] - tryb dodawania linii,
LineStyle: [- | -- | : | -. | none] - styl linii,
LineWidth: - grubość linii,
Marker: [+ | o | * | . | x | square | diamond | … | none] – kształt znaczników,
MarkerSize: - rozmiar znaczników,
MarkerEdgeColor: [kolor | none | auto] - kolor krawędzi znaczników,
MarkerFaceColor: [kolor | none | auto] - kolor wypełnienia znaczników,
XData: - wektory danych definiujące linię
YData:
ZData:

2.5.6 Atrybuty obiektu "text"

BackgroundColor - kolor tła (lub none gdy przezroczyste),
Color - kolor tekstu,
EdgeColor - kolor obramowania
Editing: [on | off] - możliwość interaktywnej edycji tekstu,
EraseMode: [normal | background | xor | none] – sposób nakładania tekstu na rysunek,
Extent - pozycja I wymiary okienka tekstowego,
FontAngle: [normal | italic | oblique] - krój czcionki,
FontName - nazwa czcionki,
FontSize - wielkość czcionki,
FontUnits: [inches | centimeters | normalized | points | pixels] - jednostki dla FontSize,
FontWeight: [light | normal | demi | bold] - waga czcionki,
HorizontalAlignment: [left | center | right] – wyrównanie w poziomie,
LineStyle: [- | -- | : | -. | none] - rodzaj linii,
LineWidth - grubość linii,
Margin - odstęp tekstu od obramowania,
Position - położenie początku tekstu w układzie,
Rotation - obrót tekstu,
String - tekst wyświetlany
Units - jednostki miary,
Interpreter: [latex | tex | none] – typ interpretera tekstowego znaków graficznych,
VerticalAlignment: [top | cap | middle | baseline | bottom] – wyrównanie w pionie.

2.5.7 Atrybuty obiektu "image"

AlphaData
AlphaDataMapping: [none | direct | scaled]
CData
CDataMapping: [direct | scaled]
EraseMode: [normal | background | xor | none]
XData
YData

2.5.8 Atrybuty obiektu "light"

Color - współrzędne RGB koloru,
Position - położenie źródła światła,
Style: [infinite | local] - typ źródła światła.

2.5.9 Atrybuty obiektu "surface"

AlphaData: - dane o przejrzystości obiektu,
AlphaDataMapping: [none | direct | scaled] - tryb używania danych przejrzystości,
CData - dane o kolorach poszczególnych punktów,
CDataMapping: [direct | scaled] - sposób używania palety barw,
DisplayName: - nazwa wykorzystywana w legendzie,
EdgeAlpha: [0..1 | flat | interp] - przejrzystość krawędzi,
EdgeColor: [kolor | none | flat | interp] - kolor krawędzi,
EraseMode: [normal | none | xor | background] - technika stosowana do rysowania i usuwania obiektu,
FaceAlpha: [0..1 | flat | interp | texturemap] - przejrzystość powierzchni,
FaceColor: [kolor | none | flat | interp] - kolor powierzchni,
LineStyle: [- | -- | : | -. | none] - rodzaj linii,
LineWidth: - grubość linii,
Marker: [+ | o | * | x | square | diamond | ^ | v | < | > | pentagram | hexagram | none] - znaczniki na liniach,
MarkerEdgeColor: [none | auto | flat | kolor] - kolor obramowania znaczników,
MarkerFaceColor: [none | auto | flat | kolor] - kolor wypełnienia znaczników
MarkerSize: - rozmiar znacznika,
MeshStyle: [both | row | column] - rysowanie pełnej siatki, samych wierszy lub samych kolumn,
XData: - macierze współrzędnych
YData:
ZData:
FaceLighting: [none | flat | gouraud | phong] - algorytm wyznaczania oświetlenia powierzchni
EdgeLighting: [none | flat | gouraud | phong] - algorytm wyznaczania oświetlenia krawędzi,
BackFaceLighting: [unlit | lit | reverselit] - oświetlenie tylne,
AmbientStrength: [0..1]- natężenie bezkierunkowego oświetlenia otoczenia,
DiffuseStrength: [0..1]- natężenie światła rozproszonego, padającego na powierzchnię,
SpecularStrength: [0..1]- natężenie światła pochodzącego ze źródeł oświetlenia,
SpecularExponent: - liczba >1, określająca wielkość źródła światła,
SpecularColorReflectance: [0..1] - zależność koloru światła odbitego od koloru powierzchni i koloru światła padającego,
VertexNormals: - tablica wektorów prostopadłych do powierzchni,
NormalMode: [auto | manual] - sposób generowania wektorów prostopadłych,
XDataMode: [auto | manual] - sposób wprowadzania wartości zmiennych osi,
XDataSource: - źródło wartości zmiennych osi,
YDataMode:
YDataSource:
ZDataSource:
CDataMode: [auto | manual] - sposób przypisywania kolorów wartościom danych,
CDataSource: - źródło danych.

2.5.10 Atrybuty obiektu "patch"

AlphaDataMapping: [none | direct | scaled] - tryb używania danych przejrzystości,

CData - dane o kolorach poszczególnych punktów,
CDataMapping: [direct | scaled] - sposób używania palety barw,
DisplayName: - nazwa wykorzystywana w legendzie,
EdgeAlpha: [flat | interp | 0..1] - przejrzystość krawędzi,
EdgeColor: [none | flat | interp | RGB] - kolor krawędzi,
EdgeLighting: [{none} | flat | gouraud | phong]
EraseMode: [{normal} | background | xor | none] – sposób dodawania do rysunku,
FaceAlpha: [flat | interp | 0..1] - przejrzystość obszaru,
FaceColor: [none | flat | interp | RGB] - kolor obszaru,
FaceLighting: [none | {flat} | gouraud | phong]
Faces - połączenia z innymi obszarami,
FaceVertexAlphaData
FaceVertexCData
LineStyle: [- | -- | : | -. | none] - rodzaj linii obramowania,
LineWidth: - grubość linii,
Marker: [+ | o | * | . | x | square | diamond | v | ^ | > | < | pentagram | hexagram | {none}]
MarkerEdgeColor: [none | {auto} | flat | RGB] – kolor obrysu znaczników,
MarkerFaceColor: [{none} | auto | flat | RGB] – kolor znaczników,
MarkerSize: - rozmiar znacznika,
Vertices - macierz zawierająca współrzędne krawędzi,
XData
YData
ZData
BackFaceLighting: [unlit | lit | {reverselit}]
AmbientStrength: [0..1]- natężenie bezkierunkowego oświetlenia otoczenia,
DiffuseStrength: [0..1]- natężenie światła rozproszonego, padającego na powierzchnię,
SpecularStrength: [0..1]- natężenie światła pochodzącego ze źródeł oświetlenia,
SpecularExponent: - liczba >1, określająca wielkość źródła światła,
SpecularColorReflectance: [0..1] - zależność koloru światła odbitego od koloru powierzchni i koloru światła padającego,
VertexNormals: - tablica wektorów prostopadłych do powierzchni,
NormalMode: [auto | manual] - sposób generowania wektorów prostopadłych,

3 Tworzenie graficznego interfejsu użytkownika

MATLAB daje możliwość tworzenia okien graficznego interfejsu użytkownika, umieszczania w nich standardowych elementów sterujących i oprogramowania ich reakcji na ruch bądź kliknięcia myszy. Do tworzenia bądź edycji tego typu oprogramowania służy edytor wywoływany poleceniem **guide**. Wynikiem pracy edytora jest para plików *plik.fig* oraz *plik.m* zawierające komplet informacji o zaprojektowanym oknie. Plik o rozszerzeniu .m zawiera procedury obsługi elementów sterujących i służy do uruchomienia okna interfejsu. Istnieje możliwość takiego ustawienia opcji edytora, aby wszystkie informacje zawarte były w jednym pliku z rozszerzeniem .fig - w tym przypadku uruchomienie okna odbywa się za pomocą polecenia **open**.

3.1 Edytor formularzy

Po uruchomieniu poleceniem **guide** edytora, otrzymujemy możliwość edycji istniejącego formularza lub utworzenie nowego. W drugim przypadku można utworzyć formularz od podstaw (Blank GUI) albo skorzystać z gotowych wzorów formularzy: formularz z podstawowymi elementami sterującymi (GUI with UIcontrols), formularz zawierający wykres i listę rozwijaną (GUI with Axes and Menu) lub modalne okno dialogowe (Modal Question Dialog).

3.1.1 Tworzenie formularza

Po wybraniu tej opcji z menu startowego edytora, otwiera się okno zawierające pusty obszar roboczy formularza, lub (w przypadku wyboru któregoś z formularzy wzorcowych - przykładowy formularz, który można następnie edytować). Klikając myszą w prawym dolnym rogu obszaru roboczego można zmienić jego rozmiary.
Okno edytora wyposażone jest w pasek narzędziowy, pozwalający na wybór graficznych elementów sterujących, do umieszczenia w obszarze roboczym:

- elementy pochodne klasy **uicontrol:**
  - Przycisk (Push Button) uruchamia akcję przy kliknięciu. Graficznie przycisk zmienia wygląd w momencie naciśnięcia, a jego procedura **Callback** zostaje uruchomiona po zwolnieniu przycisku.
  - Przełącznik (Toggle Button) generuje akcję zaznaczając jednocześnie swój stan - naciśnięty lub wyzwolony. Do powrotu do stanu wyjściowego potrzebne jest powtórne kliknięcie (i ponowne uruchomienie procedury **Callback**).
  - Pole wyboru (Checkbox) uruchamia akcję po kliknięciu, jednocześnie sygnalizując swoim wyglądem zmianę stanu - wybrany lub niewybrany. Stosuje się, gdy użytkownik ma dokonać szeregu niezależnych wyborów opcji.
  - Przycisk radiowy (Radio Button) jest podobny w działaniu do pola wyboru z tą różnicą, że z reguły występuje w grupach, w których dozwolony jest wybór tylko jednej opcji. Aktywacja przycisku odbywa się przez kliknięcie na obiekcie.
  - Pole tekstowe (Edit Text) pozwala na wprowadzanie i edycję łańcucha tekstu. Jego atrybut *String* zawiera tekst wprowadzony przez użytkownika. Po naciśnięciu klawisza <Enter> zostaje uruchomiona procedura **Callback**.
  - Etykieta (Static Text) zawiera tekst, który nie może być edytowany. Nie ma też możliwości uruchomienia procedury **Callback**.
  - Suwak (Slider) jest polem przyjmującym wejściowe dane numeryczne w określonym zakresie. Użytkownik przesuwa suwak za pomocą myszki wzdłuż skali. Aktualne położenie suwaka determinuje przechowywaną wartość liczbową.
  - Lista rozwijana (Popup Menu) daje możliwość wyboru jednej z pozycji. Lista wyświetlana jest po naciśnięciu klawisza ze strzałką.
  - Lista wyboru (Listbox) wyświetla na stałe listę pozycji dając użytkownikowi możliwość wyboru jednej z nich.
- elementy pochodne klasy **uipanel**

- Panel (Panel) jest to zgrupowanie różnych elementów (jak pola sterujące, inne panele czy pola wykresów) dla wizualnego ułatwienia obsługi formularza. Panel może posiadać tytuł. Elementy zawarte w panelu tworzą grupę, która przemieszcza się przy projektowaniu wraz z panelem. Pole Panel dostępne jest od wersji 7 MATLAB-a.
- Grupa przycisków (Button Group) skupia w sobie zestaw przycisków radiowych lub przełączników nawzajem się wykluczających, obsługując jednocześnie procedurę wyboru tylko jednego z nich. Jego funkcja **SelectionChangeFcn** zastępuje funkcje **Callback** elementów wchodzących w skład grupy. Ten typ pola dostępny jest od wersji 7 MATLAB-a. W poprzednich wersjach programu pojedynczy wybór musiał być obsługiwany przez procedury **Callback** poszczególnych przycisków.

- Elementy klasy **uitable** – Tablica (Table), umożliwiają przedstawienie dwuwymiarowych macierzy w postaci tabelarycznej .
- Pole wykresu (Axes) umożliwia przedstawienie grafiki (jak wykresy czy obrazy) na formularzu.
- Element ActiveX (ActiveX Control) daje możliwość użycia elementów ActiveX zarejestrowanych w systemie.

Edycja formularza polega na przeciąganiu elementów kontrolnych z paska narzędziowego do wybranych miejsc obszaru roboczego. Rozmiary elementów można zmieniać. Edytor zawiera również narzędzie do wyrównywania wzajemnego położenia elementów (menu **Tools - Align Objects**). Atrybuty obiektów graficznych można edytować za pomocą przeglądarki atrybutów (menu **View - Property Inspector**). Po wybraniu obiektu na obszarze roboczym przeglądarka wyświetla i pozwala zmieniać jego atrybuty. W procesie tworzenia nowego formularza należy na początku nadać nazwy poszczególnym elementom. Nazwę okna przechowuje atrybut formularza *Name*, nazwa obiektu panel lub grupa przycisków przechowywana jest w atrybucie *Title*, nazwy przycisków i napisów przechowują ich atrybuty *String*. W przypadku obiektów typu lista rozwijana lub lista wyboru do wprowadzenia danych wielowierszowych można użyć okienka edytora tekstu (wybieranego przyciskiem umieszczonym na lewo treści atrybutu *String*). Wygodnie jest również wprowadzić dla poszczególnych obiektów graficznych identyfikatory *Tag* (unikalne w ramach danego okna interfejsu). Aktywne elementy sterujące powinny posiadać podprogramy realizowane przy wystąpieniu typowych zdarzeń, jak kliknięcie w przycisk, wybór wartości z listy. Podprogramy te zawarte są w atrybutach **Callback**. Ponadto oprogramowane mogą być również zdarzenia związane z utworzeniem lub skasowaniem obiektu.
Przy wprowadzeniu nowego elementu sterującego, element otrzymuje identyfikator *Tag* składający się z nazwy typu elementu oraz numeru kolejnego, a jego atrybut **Callback** otrzymuje wartość **%automatic**. Po zachowaniu aktualnego projektu, atrybuty **Callback** jego aktywnych elementów zostają przekształcone do postaci odpowiadającej typowi elementu. Przy zmianie wartości atrybutu *Tag* zostają dopasowane parametry w procedurach. Przykładowo dla przycisku o identyfikatorze P1 w formularzu o nazwie Test, zostanie wygenerowana procedura:

```
Test('P1_Callback',gcbo,[],guidata(gcbo))
```

Przy pierwszym zachowaniu projektowanego formularza zostają utworzone dwa pliki o tej samej nazwie (zgodnej z atrybutem *Name*), z rozszerzeniami:

- .fig – plik (w formacie .mat), zawierający opis elementów formularza w postaci struktury, o nazwie związanej z wersją programu MATLAB (i ze sposobem kompresji danych przechowywanych w pliku). Przykładowo dla wersji 7 mamy strukturę o nazwie hgS_070000,
- .m – plik tekstowy zawierający definicję funkcji generującej formularz oraz definicje funkcji obsługi elementów sterujących formularza.

3.1.2 Zawartość pliku .fig

Zapisana w pliku *Name*.fig zmienna hgS_###### opisująca formularz posiada pola:

type	'figure' - typ obiektu ,
handle	– uchwyt do formularza
properties	– struktura, przypominająca zestaw atrybutów obiektu „figure”. Pole **properties** jest zestawem atrybutów obiektu tworzącego formularz

children – wektor struktur opisujących elementy pochodne występujące na formularzu. Struktury te mają układ identyczny ze strukturą opisującą cały formularz. Pola **type** tych struktur zawierają nazwy klas konkretnych obiektów, pola **handle** zawierają uchwyty, pozwalające na odnoszenie się do obiektów, natomiast pola **properties** - struktury opisujące dany rodzaj elementu aktywnego na formularzu.

special – wektor pusty (do przyszłego wykorzystania).

W skład struktury **properties** wchodzą m.in. pola zawierające dane, które można wykorzystywać przy oprogramowywaniu GUI. Są to

UserData - pole zawierające dane nie wykorzystywane przez MATLAB (ale np. wykorzystywane przez pakiety narzędziowe – vide Mapping Toolbox).

ApplicationData - struktura zawierająca pola z danymi aplikacji przypisanymi danemu obiektowi.

Struktura **ApplicationData** formularza zawiera pola:

GUIDEOptions opcje programu GUIDE

lastValidTag - aktualny identyfikator obiektu

W przypadku elementów formularza, w polu **ApplicationData** nie występuje pole **GUIDEOptions**. Do struktury **ApplicationData** można dołączyć dane oraz przekazywać je pomiędzy poszczególnymi formatkami GUI za pomocą funkcji **setappdata** oraz **getappdata**.

3.1.3 Zawartość pliku .m

MATLAB przechowuje podprogramy obsługujące zdarzenia związane z formularzem w m-pliku o nazwie zgodnej z nazwą formularza. Dla przykładowego formularza zapamiętanego pod nazwą ‘test’, plik **test.m** rozpoczyna się definicją funkcji kreującej, w postaci:

```
function varargout = test(varargin)
% Tu komentarz
gui_Singleton = 1;
gui_State = struct('gui_Name',   mfilename, ...
                   'gui_Singleton'    gui_Singleton, ...
                   'gui_OpeningFcn'   @test_OpeningFcn, ...
                   'gui_OutputFcn'    @test_OutputFcn, ...
                   'gui_LayoutFcn'    [], ...
                   'gui_Callback'[])
If nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
```

Funkcja ta wykorzystuje funkcje wewnętrzne **str2func** i **gui_mainfcn**. Dalej umieszczone zostają funkcje (podprogramy):

```
function test_OpeningFcn(hObject, eventdata, handles, varargin)
function varargout = test_OutputFcn(hObject, eventdata, handles)
```

oraz wygenerowane automatycznie funkcje oprogramowujące poszczególne elementy sterujące.

3.1.4 Wprowadzanie procedur obsługi zdarzeń

Pierwsze zapisanie projektu formularza powoduje powstanie m-pliku zawierającego wzorce podprogramów obsługi formularza i elementów sterujących. Do edycji tych podprogramów służy edytor wbudowany MATLAB-a. Z poziomu edytora formularzy można go wywołać za pomocą menu **View - M-file Editor**. W tym edytorze można wybrać poszukiwaną procedurę z listy uzyskanej po naciśnięciu przycisku ***f***. na pasku narzędzi edytora. Podprogramy poszczególnych elementów aktywnych mogą przekazywać sobie wartości za pomocą wygenerowanej przez MATLAB struktury o nazwie **handles**, wspólnej dla całego formularza (ponieważ jest zdefiniowana w obszarze roboczym

funkcji pierwotnej pliku). Proces przechowania w tej strukturze np. elementów macierzy X wygląda następująco:

- wybieramy nazwę (np. Dane) dla pola struktury **handles** i tworzymy to pole podstawiając jednocześnie wartość:
  ```
  handles.Dane = X;
  ```
- przechowujemy strukturę za pomocą funkcji **guidata**:
  ```
  guidata(hObject,handles)
  ```
  gdzie **hObject** jest uchwytem do elementu realizującego daną procedurę.
- Chcąc użyć tej wartości w innej procedurze, używamy instrukcji:
  ```
  Y = handles.Dane;
  ```

Dane wspólne mogą być również przechowywane w strukturach **ApplicationData** związanych z obiektami graficznymi. W ten sposób można przekazywać dane pomiędzy formularzami GUI. Na przykład załóżmy, że mamy dwa formularze: **Main** i **Sub**, które powinny ze sobą współpracować. W takim przypadku można uchwyty do tych formularzy przechować w obiekcie wspólnym, jakim jest **root**. W tym celu wprowadzamy w procedurach OpeningFcn obu formularzy odpowiednie instrukcje:

- w pliku Main.m:
  ```
  setappdata(0, 'hMain', gcf)
  ```
- w pliku Sub.m:
  ```
  setappdata(0, 'hSub', gcf)
  ```

W ten sposób w strukturze **ApplicationData** obiektu **root** pojawią się pola hMain oraz hSub, zawierające uchwyty do formularzy.
Załóżmy teraz, że należy z poziomu procedury Callback któregoś z elementów sterujących formularza Sub uruchomić funkcję fMain z formularza Main z argumentami dataMain (z formularza Main) i dataSub (z formularza Sub). W tym celu musimy

- w pliku Main.m, w procedurze Main_OpeningFcn dodać instrukcje:
  ```
  setappdata(gcf, 'dataMain', dataMain)
  setappdata(gcf, 'hfMain', @fMain)         % zakładając, że funkcja fMain
                                            % jest zdefiniowana
  ```
- w pliku Sub.m, w odpowiedniej procedurze Callback:
  ```
  hMain = getappdata(0, 'hMain');
  dataMain = getappdata(hMain, 'dataMain');
  funMain = getappdata(hMain, 'hfMain');
  feval(funMain, dataMain, dataSub);        % wywołanie funkcji o uchwycie
                                            % funMain z argumentami dataMain
                                            % i dataSub
  ```

3.2 Oprogramowanie formularza

3.2.1 Procedura OpeningFcn

Pierwszym z podprogramów jest procedura realizowana przy otwarciu formularza, zanim zostanie on wyświetlony. Np dla formularza o nazwie Test będzie to procedura :

```
function Test_OpeningFcn(hObject, eventdata, handles, varargin)
```

gdzie:

- **hObject** jest uchwytem do formularza,
- **eventdata** jest argumentem dotychczas niewykorzystywanym,
- **handles** jest strukturą, służącą do przekazywania danych między obiektami formularza,
- **varargin** zawiera dane przekazywane przy otwieraniu formularza. Taka startowa procedura może służyć do wygenerowania lub pobrania danych zewnętrznych a następnie przekazania za pomocą struktury handle do pozostałych podprogramów.

Automatycznie generowany wzorzec tej funkcji ma dodatkowo wprowadzone dwa polecenia:

```
handles.output = hObject;
guidata(hObject, handles)
```

służące do utworzenia pola **output** w strukturze **handles** i umieszczeniu w nim uchwytu do formularza. Pole to jest używane przez funkcję wyjścia OutputFcn w przypadku, gdy chcemy przekazać uchwyt np. do innego formularza.

3.2.2 Funkcja OutputFcn

Definicja tej funkcji dla formularza (niech ma on nazwę Test) ma postać:

```
function vargout = Test_OutputFcn(hObject, eventdata, handles)
vargout{1} = handles.output;
```

Standardowo w strukturze **handles** pole **output** ma wartość uchwytu do formularza. Jest to realizowane za pomocą kodu w procedurze OpeningFcn:

```
handles.output = hObject;
guidata(hObject, handles);
```

oraz w procedurze OutputFcn:

```
varargout{1} = handles.output;
```

Można w to miejsce wprowadzić inne dane, związane z konkretnym aktywnym elementem formularza, modyfikując procedury w następujący sposób:

1. w procedurze OpeningFcn wprowadzamy polecenie blokujące, powodujące że formularz będzie w interakcji z użytkownikiem:
   ```
   uwait;
   ```
2. dla elementu, który ma zmieniać wartość zwracaną przez funkcję Test, wprowadzamy w jego procedurze Callback następujące polecenia (**uiresume** pozwala na zakończenie pracy):
   ```
   handles.output = aktualna_wartość_wyjściowa;
   guidata(gcf, handles)
   uiresume;
   ```

Po otwarciu formularza poleceniem:

```
>> X = Test;
```

uruchomienie procedury Callback zdarzeniem związanym z wybranym elementem (np. naciśnięcie przycisku) spowoduje po zamknięciu formularza podstawienie *aktualnej_wartości_wyjściowej* na zmienną X.

3.2.3 Procedury Callback

W momencie uaktywnienia obiektu na formularzu zostaje uruchomiona procedura **Callback** związana z danym obiektem. Nazwa tej procedury określona jest przez atrybut *Tag* obiektu. Wiersz definicji takiej procedury, generowany automatycznie w m-pliku ma postać:

```
function Tag_Callback(hObject, eventdata, handles)
```

Argument **hObject** jest uchwytem do obiektu, a **handles** jest strukturą służącą do przekazywania danych między podprogramami formularza. Argument **eventdata** nie jest używany.

3.2.4 Sposoby wywoływania formularza

Funkcja otwierająca formularz (przyjmijmy nazwę Test) może być wywoływana na różne sposoby:

- wywołanie bez argumentów
  ```
  >> Test
  ```
  powoduje zwykłe otwarcie formularza
- wywołanie w postaci
  ```
  >> H = Test;
  ```
  powoduje otwarcie formularza i podstawienie uchwytu do niego na zmienną H,
- wywołanie
  ```
  >> Test('atrybut', wartość)
  ```
  gdzie *atrybut* jest nazwą atrybutu obiektu formularz, powoduje otwarcie formularza z jednoczesnym nadaniem wartości atrybutowi. Wywołanie może zawierać więcej par *atrybut-wartość*.
- wywołanie
  ```
  >> Test('funkcja', argument…)
  ```
  powoduje otwarcie formularza i jednoczesne uruchomienie jego wewnętrznej funkcji o zadanej nazwie z wprowadzonymi argumentami,
- wywołanie
  ```
  >> Test('nazwa', wartość…)
  ```
  gdzie *nazwa* nie jest nazwą atrybutu ani funkcji wewnętrznej formularza powoduje uruchomienie formularza i przekazanie par *nazwa* - *wartość* do funkcji OpeningFcn

formularza. W przypadku, gdy chcemy jednocześnie przekazać wartości atrybutów i wartości nie będące atrybutami formularza, należy w pierwszej kolejności umieścić atrybuty.

3.3 Oprogramowanie procedur Callback

Procedury **Callback** elementów sterujących powinny realizować zadania charakterystyczne dla rodzaju elementu. Poniżej przedstawiono kilka przykładów programowania procedur obsługi elementów sterujących.

3.3.1 Przełącznik

Procedura obsługi przełącznika ma za zadanie określenie stanu, w jakim przełącznik się znajduje. Atrybut *Max* (domyślna wartość 1) określa wartość atrybutu *Value* nadawaną przy stanie "włączonym" natomiast atrybut *Min* (wartość domyślna 0) - w stanie "spoczynkowym". Oto przykładowy fragment kodu:

```
function B1_Callback(hObject, eventdata, handles)
bs = get(hObject, 'Value');
if bs == get(hObject, 'Max')
      %przycisk w stanie włączonym
elseif bs == get(hObject, 'Min')
      %przycisk w stanie wyłączonym
end
```

Jeśli przełączniki są umieszczony wewnątrz grupy przycisków, to ich obsługą zajmuje się procedura SelectionChangeFcn tej grupy.

3.3.2 Przycisk radiowy

Użycie atrybutów *Min*, *Max* oraz *Value* jest podobne jak w przypadku przełącznika. Jeśli przyciski zostały umieszczone w obiekcie "grupa przycisków", to do ich obsługi służy procedura SelectionChangeFcn tej grupy.
Chcąc uzyskać wybór tylko jednego elementu (np. ze zbioru przycisków B1..B3), w przypadku przycisków niezgrupowanych (lub w wersjach poprzednich programu MATLAB), można wprowadzić w m-pliku nową funkcję:

```
function B_off(h)
  set(h, 'Value', 0)
```

i wywołać ją odpowiednio w procedurze **Callback** każdego z przycisków (tu dla przycisku B1):

```
B_off([handles.B2 handles.B3])
```

3.3.3 Pole wyboru

Atrybut *Value* pola wyboru może przyjmować wartości *Max* w przypadku, gdy pole jest wybrane, lub *Min*, gdy nie jest. Kolejne kliknięcia na pole wyboru zmieniają jego stan na przeciwny. W procedurze **Callback** można sprawdzić stan aktualny (po kliknięciu na pole) i wykonać odpowiedni fragment kodu programu.

3.3.4 Pole tekstowe

W przypadku pola tekstowego atrybuty *Max* oraz *Min* niosą informację o możliwości wprowadzania danych w wielu wierszach (gdy *Max* - *Min* > 1). Atrybut *String* zawiera wprowadzone dane. Chcąc interpretować zawartość pola tekstowego jak liczbę, należy użyć funkcję konwersji np. **str2double**. W przypadku niepowodzenia konwersji funkcja ta zwraca wartość nieokreśloną (NaN), co może być sprawdzone za pomocą funkcji **isnan**.

3.3.5 Suwak

Dla suwaka atrybuty *Min* oraz *Max* wyznaczają zakres wartości osiąganych przez atrybut *Value*. Dokonanie operacji poruszenia przesuwki suwaka powoduje wywołanie procedury **Callback**, w której można sprawdzić aktualną wartość atrybutu *Value*.

3.3.6 Lista wyboru

Uruchomienie może się odbyć po naciśnięciu a następnie zwolnieniu przycisku myszy na wybranym elemencie, lub po wykonaniu odpowiedniej operacji klawiaturowej:

- klawisze strzałek powodują zmianę atrybutu *Value* i uruchamiają procedurę **Callback**.
- Klawisze <Enter> oraz <Spacja> nie zmieniają atrybutu *Value*, ale wywołują procedurę **Callback.**

Przy podwójnym kliknięciu procedura **Callback** zostanie wywołana dwukrotnie. Atrybut *SelectionType* przy pierwszym kliknięciu otrzymuje wartość **normal**, a przy drugim - **open**. Atrybuty *Min* oraz *Max* informują, czy dopuszczalny jest wybór wielokrotny (*Max* - *Min* >1). Atrybut *Value* zawiera indeks (numer) wybranej pozycji.

3.3.7 Lista rozwijana

Atrybut *Value* zawiera indeks (numer) wybranej pozycji. Łańcuch odpowiadający danej pozycji może być pobrany z atrybutu *String*, który jest zmienną klasy tablica komórek. Poniżej podano fragment kodu, który umożliwia taką operację:

```
function Lista_Callback(hObject, eventdata, handles)
val = get(hObject,'Value');
string_list = get(hObject,'String');
selected_string = string_list{val};
...
```

3.3.8 Pole wykresu

Pole wykresu jest obiektem klasy **axes**. Obiekt ten nie należy do elementów kontrolnych, ale można oprogramować zdarzenie kliknięcia przez oprogramowanie funkcji **ButtonDownFcn**. W przypadku, gdy na obszarze pola wykresu zostały umieszczone jego elementy pochodne (**line**, **text** etc.), zdarzenie **ButtonDown** nie wystąpi.

3.3.9 Grupa przycisków

Grupa przycisków jest polem, którego zadaniem jest w sposób graficzny poinformować o współzależności zgrupowanych elementów, umożliwiając jednocześnie obsługę wyłącznego wyboru jednego z grupy przycisków radiowych lub przełączników. Do obsługi zdarzeń związanych z elementami grupy służy podprogram **SelectionChangeFcn**, w którym argument *hObject* jest uchwytem do aktywnego obiektu (którego identyfikator można uzyskać np. za pomocą polecenia

```
button_name = get(hObject,'Tag')
```

3.4 Atrybuty obiektów GUI

Atrybuty obiektów tworzących graficzny interfejs użytkownika mogą być dostępne (do odczytania lub zmiany) na dwa sposoby:

- za pomocą narzędzia Property Inspector, dostępnego z programu GUIDE lub uruchamianego z poziomu wiersza poleceń za pomocą funkcji **inspect**(h), gdzie h jest uchwytem (uchwytem) do obiektu. Narzędzie to pozwala na interaktywne przeglądanie i zmianę atrybutów obiektu.
- za pomocą funkcji **set**(h,...) oraz **get**(h,...), służących do zmiany wartości lub wyświetlenia informacji o atrybutach określonego obiektu.

Atrybuty obiektów klasy **figure** (okna aplikacji) oraz **axes** (układy współrzędnych) zostały omówione w rozdziale 2.5.

3.4.1 Atrybuty wspólne różnych elementów GUI

BackgroundColor	macierz 2×3 lub jeden z predefiniowanych kolorów (pełna nazwa lub pojedyncza litera: r, g, b, c, m, y, k lub w) - kolor tła komórki . Domyślnie wektor RGB (liczby w zakresie 0..1). Ewentualny 2 wiersz macierzy jest wykorzystywany gdy atrybut RowStripping ma wartość on.

Enable [on | inactive | off] – stan dostępu do elementu. Zezwolenie na aktywację obiektu.

- on - obiekt może być aktywowany,
- inactive - obiekt nie może być aktywowany,
- off - obiekt nie może być aktywowany, ma ponadto zmieniony wygląd (szary).

Kliknięcie lewym klawiszem myszy na obiekcie, którego atrybut Enable ma wartość **on** powoduje:

1. Ustawienie odpowiedniej wartości atrybutu **SelectionType** okna.
2. Uruchomienie procedury **Callback** elementu.
3. Nie zmienia atrybutu **CurrentPoint** okna i nie uruchamia procedury **ButtonDownFcn** elementu ani **WindowButtonDownFcn** okna.

Kliknięcie lewym klawiszem myszy, gdy atrybut Enable ma wartość **off** albo kliknięcie prawym klawiszem myszy na obiekcie (wartość atrybutu Enable dowolna) powoduje:

1. Ustawienie odpowiedniej wartości atrybutu **SelectionType** okna.
2. Ustawienie wartości atrybutu **CurrentPoint** okna.
3. Uruchomienie procedury **WindowButtonDownFcn** okna.

Extent – obszar, podany w postaci [0 0 szerokość wysokość] zajmowany przez dane lub tekst (atrybut String) związany z obiektem. Dla tekstów wielowierszowych zwracane są rozmiary tekstu, dla tekstów jednowierszowych zwracana jest długość i wysokość pojedynczego wiersza, nawet w przypadku, gdy tekst na elemencie będzie zawijany. (tylko do odczytu)

FontAngle [normal | italic | oblique] - pochylenie tekstu.

FontName [łańcuch] - nazwa czcionki użytej do wyświetlenia danych lub tekstu (String). Wprowadzenie wartości 'FixedWidth' pozwala na wybór jako domyślnej czcionki równoodstępowej.

FontSize [liczba] - rozmiar czcionki wyrażony w jednostkach określonych przez atrybut FontUnits.

FontUnits [points | normalized | inches | centimeters | pixels] - jednostki wielkości czcionki (1 pkt =. 1/72 cala). Wybór wartości normalized umożliwia automatyczne przeskalowywanie tekstu przy zmianie rozmiarów elementu.

FontWeight [light | normal | demi | bold] - waga czcionki.

ForegroundColor ColorSpec (jak przy BackgroundColor). Wartość domyślna – black (czarny).

KeyPressFcn [łańcuch | uchwyt_do_funkcji] - procedura wywołana przez naciśnięcie klawisza w chwili gdy element sterujący jest wybrany. Jeżeli nie jest wybrany żaden z elementów, uruchamiana jest procedura **KeyPressFcn** okna (jeśli istnieje). Jeśli wyspecyfikowaną funkcją jest nazwa m-pliku, to dane o naciśniętym klawiszu zawiera atrybut **CurrentCharacter** okna. Jeśli atrybutem jest uchwyt do funkcji, to dane o klawiszu zawiera struktura eventdata, o polach **Character**, **Modifier** i **Key**:

Pole	Opis Klawisz	**a**	=	**Shift**	**Shift/a**
Character	Interpretacja klawisza	'a'	'='	''	'A'
Modifier	Modyfikator lub pusta komórka	{1×0 cell}	{1×0 cell}	{'shift'}	{'shift'}
Key	Nazwa naciśniętego klawisza	'a'	'equal'	'shift'	'a'

Position [wektor] - pozycja i rozmiar elementu. Przechowywany w postaci wektora: [lewy dół szerokość wysokość] wyrażonego w jednostkach określonych przez atrybut Units. W przypadku suwaka większy z wymiarów {szerokość, wysokość} określa orientację elementu.

TooltipString [łańcuch] - wyskakujący opis obiektu.

Units — [pixels | normalized | inches | centimeters | points | characters] - jednostki miary stosowane w różnych atrybutach. Jednostki **normalized** obliczane są w stosunku do rozmiarów obiektu. Jednostki **characters** za podstawę mają rozmiar litery ‘x’ w domyślnej czcionce systemowej.

3.4.2 Atrybuty elementów sterujących (Uicontrol)

Callback — [łańcuch] - procedura uruchamiana przy aktywowaniu obiektu (np. kliknięciu myszą na przycisk lub przesunięciu suwaka). Procedura jest definiowana w postaci wyrażenia w języku MATLAB lub nazwy m-pliku. W przypadku pola tekstowego uruchomienie procedury nastąpi po wprowadzeniu tekstu a następnie wykonaniu jednej z czynności:

- kliknięciu na inny element,
- naciśnięciu klawisza <Enter> dla pól jednowierszowych,
- naciśnięciu kombinacji <Ctrl>+<Enter> dla pól wielowierszowych.

Dla etykiet (static text) nie ma akcji wywołujących procedurę Callback.

CData — macierz - obraz w postaci trójwymiarowej tablicy RGB o wartościach z przedziału <0,1>, wyświetlany na przycisku lub przełączniku.

HorizontalAlignment — [left | center | right] - sposób wyrównania tekstu (atrybut String) w poziomie. Dotyczy tylko pól tekstowych i etykiet.

ListboxTop — [skalar] - indeks (numer) tekstu wyświetlanego na pierwszym miejscu w liście wyboru.

Max — [skalar] - wartość maksymalna. Posiada różne znaczenie dla różnych styli elementów:

- Pole wyboru – wartość atrybutu Value w stanie wybranym.
- Pole tekstowe – jeśli Max – Min > 1 to pole może być wielowierszowe.
- Lista wyboru - jeśli Max – Min > 1 to dozwolony jest wybór wielokrotny.
- Przycisk radiowy – wartość atrybutu Value w stanie wybranym.
- Suwak – maksymalna wartość atrybutu Value. Musi być większa od Min.
- Przełącznik - wartość atrybutu Value w stanie wybranym.

Lista rozwijana, przycisk i etykieta nie używają atrybutu Max.

Min — [skalar] - wartość minimalna. Posiada różne znaczenie dla różnych styli elementów:

- Pole wyboru – wartość atrybutu Value w stanie wybranym.
- Pole tekstowe – jeśli Max – Min > 1 to pole może być wielowierszowe.
- Lista wyboru - jeśli Max – Min > 1 to dozwolony jest wybór wielokrotny.
- Przycisk radiowy – wartość atrybutu Value w stanie wybranym.
- Suwak – minimalna wartość atrybutu Value. Musi być mniejsza od Max.
- Przełącznik - wartość atrybutu Value w stanie wybranym.

Lista rozwijana, przycisk i etykieta nie używają atrybutu Min.

SliderStep — [krok_min krok_max] - krok zmiany atrybutu Value dla suwaka przy kliknięciu na strzałkę (krok_min) lub na suwak (krok_max). Wartości (z przedziału <0,1>) określają ułamek zakresu suwaka (Max - Min).

String — [łańcuch | tablica_komórek] - dla pól wyboru, pól tekstowych, przycisków, przycisków radiowych, etykiet i przełączników - tekst wyświetlany na elemencie. Dla list wyboru i list rozwijanych - zbiór wyświetlanych pozycji. Dla wielowierszowych pól tekstowych i etykiet jako znak nowego wiersza przyjmowany jest \n. Dla wielowierszowych list wyboru i list rozwijanych argument może być tablicą komórek, w

której kolejne komórki są pozycjami listy, albo łańcuchem, w którym separatorem pozycji jest znak '|'.

Style — [pushbutton | togglebutton | radiobutton | checkbox | edit | text | slider | frame | listbox | popupmenu] - typ obiektu.

Value — [skalar | wektor] - wartość aktualna. Dla pól wyboru, przycisków radiowych i przełączników - Max gdy wybrane lub Min gdy niewybrane. Dla list wyboru wektor indeksów zaznaczonych pozycji (przy wyborze wielokrotnym). Dla list rozwijanych indeks wybranej pozycji. Pola tekstowe, przyciski i etykiety nie używają tego atrybutu.

3.4.3 Atrybuty grup obiektów (Uipanel)

BorderType — [none | etchedin | etchedout | beveledin | beveledout | line] – styl obramowania obiektu. Dla obramowań 3-D (etched, beveled) używa się kolorów HighlightColor i ShadowColor.

BorderWidth — [liczba_całkowita] – szerokość obramowania (w pikselach).

HighlightColor — ColorSpec – kolor używany przy obramowaniach 3-D.

ResizeFcn — [łańcuch | uchwyt_do_funkcji] – procedura uruchamiana, gdy użytkownik zmienia rozmiary obiektu, jeżeli atrybut okna - **Resize** ma wartość **on**.

ShadowColor — ColorSpec – kolor używany przy obramowaniach 3-D.

Title — [łańcuch] – tekst wyświetlany jako nazwa panelu.

TitlePosition — [lefttop | centertop | righttop | leftbottom | centerbottom | rightbottom] – miejsce wyświetlania nazwy panelu.

3.4.4 Atrybuty grup przycisków (Uibuttongroup)

Obiekty **uibuttongroup** mają wszystkie atrybuty jak **uipanel**, a ponadto:

SelectedObject — - uchwyt do aktualnie wybranego przycisku radiowego lub przełącznika. Domyślnie jest to uchwyt do pierwszego elementu wstawionego w procesie projektowania. Jeśli nie chcemy wstępnie wybierać, wprowadzamy pusty element []. Programowa zmiana tewgo atrybutu nie wywołuje procedury **SelectionChangeFcn**.

SelectionChangeFcn — [łańcuch | uchwyt_do_funkcji] – procedura wywoływana przy zmianie stanu wybranego przycisku radiowego lub przełącznika. Procedury realizowane przy wybraniu poszczególnych przycisków muszą być umieszczone w kodzie procedury SelectionChangeFcn obiektu ‘grupa przycisków’ a nie w procedurach Callback poszczególnych przycisków. Jeżeli procedura wywoływana jest za pomocą uchwytu, obiekt **uibuttongroup** przekazuje do niej dwa argumenty:

- source – uchwyt do obiektu,
- eventdata – struktura o postaci:

Pole struktury eventdata	Opis
EventName	SelectionChanged
OldValue	Uchwyt do obiektu wybranego przed wystąpieniem zdarzenia lub [], jeśli żaden obiekt nie był wybrany.
NewValue	Uchwyt do aktualnie wybranego obiektu.

3.4.5 Atrybuty okien tabelarycznych (Uitable)

CellEditCallback — [łańcuch | uchwyt_do_funkcji | tablica_komórek] – funkcja uruchamiana przy modyfikacji zawartości komórki tabeli. Przy wywołaniu poprzez uchwyt przekazywane są dwa parametry: hObject (uchwyt do obiektu) oraz eventdata.

CellSelectionCallback – funkcja wykonywana gdy komórka tabeli zostanie wybrana.

ColumnEditable	[macierz_logiczna_1×n | wartość_logiczna | macierz_pusta] – określa, czy dane w kolumnie mogą być edytowane (true), czy też nie (false).
ColumnFormat	[char | logical | numeric | {definicja_menu} | format] – formaty wyświetlania i edycji danych w kolumnach (definicja menu jest wektorem komórek tekstowych).
ColumnName	[{wektor_nazw_1×n} | numbered | macierz_pusta] – nazwy nagłówków kolumn w tabeli.
ColumnWidth	[macierz_1×n] – szerokości poszczególnych kolumn, wyrażone w jednostkach **Units**.
Data	[macierz | tablica_komórek] – dane wyświetlane w oknie **uitable**. Macierz (lub tablica) musi być dwuwymiarowa. Modyfikacja zawartości odbywa się za pomocą metod **get** i **set**.
RearangeableColumn	[on | off] – ustalenie, czy możliwe będzie przestawianie kolumn w tabeli. Atrybut tylko do odczytu.
RowName	[{tablica_1×n} | numbered | macierz_pusta] – nazwy poszczególnych wierszy tabeli. W nazwach wielowierszowych używamy znaku '|' do wprowadzenia nowego wiersza w łańcuchu.
RowStripping	[on | off] – wyróżnianie kolejnych wierszy tabeli na przemian kolorami BackgroundColor i ForegroundColor.

4 Podstawy użytkowania Mapping Toolbox

4.1 Typy danych geograficznych

Dane geograficzne, będące podstawą do prezentacji map można podzielić na

- dane wektorowe
- dane rastrowe

4.1.1 Dane wektorowe

Dane wektorowe są to zestawy punktów, na podstawie których można wykreślić obiekty graficzne:

- punkty
- linie (łamane łączące punkty)
- obwody (łamane zamknięte - wielokąty)
- płaty (powierzchnie ograniczone wielokątami)

Istnieje wiele formatów plików, przechowujących tego typu dane. Popularny jest np. format tzw. shapefile (ESRI). Dane w tym formacie opisane są za pomocą plików:

- .shp - plik zawierający zbiór rekordów opisujących kształty geometryczne,
- .shx - plik indeksowy, zawierający opis struktury rekordów pliku .shp
- .dbf - plik (w formacie dBase III) zawierający opisy i dane statystyczne obiektów. Między rekordami w plikach .shp i .dbf istnieje relacja jeden-do-jeden.

MATLAB informacje o kształtach geometrycznych przechowuje w postaci par wektorów *lat* i *lon*, zawierających współrzędne geograficzne punktów. Wektory te mogą zawierać informacje o większej liczbie obiektów - w takim przypadku obiekty rozdzielane są elementami NaN w danych. Wektory te mogą być również elementem struktury opisującej dane geograficzne. Nowsze wersje Mapping Toolbox używają elastycznej struktury danych (Version 2). Struktura ta zawiera pola podstawowe:

- Geometry (Point, Line, Polygon)
- BoundingBox (gdy Geometry różna od Point)
- X, Y (wektory współrzędnych)

oraz zestaw pól opisujących, które mogą być dodawane zgodnie z potrzebami. Do wczytywania danych wektorowych służy funkcja **shaperead**, posiadająca rozbudowany mechanizm selekcji danych. Jednym z argumentów funkcji jest **Selector**, który jest strukturą, składającą się z uchwytu funkcji selekcji oraz nazw argumentów wejściowych tej funkcji. Funkcja może być zawarta w oddzielnym m-pliku lub też może być umieszczona w programie. Załóżmy np., że plik test.shp zawiera rekordy danych, opisujących miasta, zawierające m.in. pola 'COUNTRY' (kod kraju) oraz 'POPULATION' (ilość ludności). Chcąc przykładowo wybrać jedynie rekordy miast, dla których COUNTRY = DK, a POPULATION > 100 000 możemy utworzyć m-plik wybor.m:

```
function result = wybor(kraj,lud)
  kod = 'DK';
  minpop = 100000;
  result = strcpi(kraj,kod) && (lud>minpop);
end
```

a następnie użyć tego filtru w wywołaniu funkcji wczytywania danych

```
>> filtr = {@wybor,'COUNTRY','POPULATION'};
>> S = shaperead('test.shp','Selector',filtr);
```

albo po prostu

```
>> S = shaperead('test.shp','Selector',{@wybor,'COUNTRY','POPULATION'});
```

W MATLAB-ie wersji 7 można też użyć m-pliku wybor.m i programu:

```
>> filtr = @wybor;
>> S = shaperead('test.shp','Selector',{filtr,'COUNTRY','POPULATION'});
```

lub bez użycia m-pliku

```
>> filtr = @(kraj,lud) strcpi(kraj,'DK') && (lud>100000);
>> S = shaperead('test.shp','Selector',{filtr,'COUNTRY','POPULATION'});
```

albo

```
>> S = shaperead('test.shp','Selector',...
 {@(kraj,lud) strcpi(kraj,'DK') && (lud>100000),'COUNTRY','POPULATION'});
```

4.1.2 Dane rastrowe

Dane rastrowe zawierają informacje o poszczególnych prezentowanych punktach. Punkty te muszą być rozmieszczone w określonej siatce. W Mapping Toolbox rozróżniane są dwa rodzaje siatek:

- siatka regularna
- siatka geograficzna

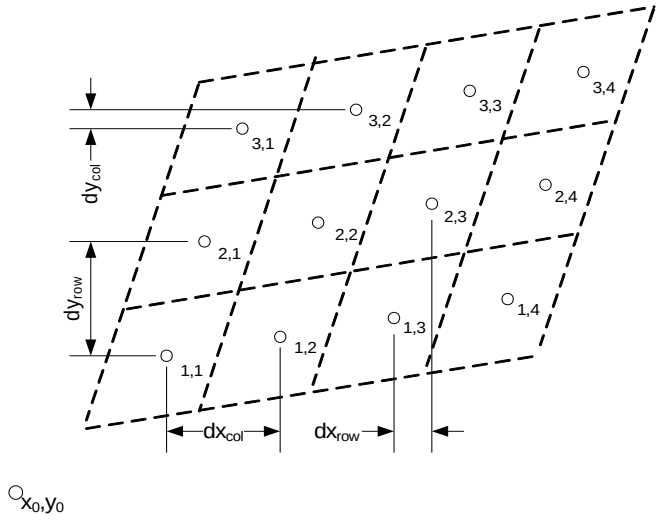
W przypadku *siatki regularnej* oprócz macierzy **map** zawierającej dane (dane mogą dotyczyć np. wartości średniej jakiegoś parametru w obrębie oczka siatki lub wartości próbki parametru w centrum tego oczka), do wyznaczenia współrzędnych geograficznych potrzebna jest również tzw. macierz referencyjna **R** (o rozmiarach 3×2).

Macierz ta pozwala na obliczenie współrzędnych *x* oraz *y* (lub *lat* i *lon*) elementu danych umieszczonego w *m*-tym wierszu i *n*-tej kolumnie tablicy:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} m & n & 1 \end{bmatrix} \times R$$

Macierz **R** w przypadku ogólnym ma postać:

$$R = \begin{bmatrix} dx_{row} & dy_{row} \\ dx_{col} & dy_{col} \\ x_0 & y_0 \end{bmatrix}$$



która pozwala na uzyskanie siatki w postaci przedstawionej na rysunku powyżej. Oczka tej siatki są równoległobokami.

W prostszym przypadku, gdy siatka jest prostokątna, elementy dx_{row} i dy_{col} są zerowe, w związku z czym pozycja w poziomie nie zależy od numeru wiersza, a pozycja w pionie nie zależy od numeru kolumny.

Najprostsza sytuacja występuje, gdy krok siatki w obu kierunkach jest jednakowy. W takiej sytuacji do jednoznacznego określenia siatki wystarcza wektor referencyjny:

$$R = \begin{bmatrix} cells\_per\_degree & north\_lat & west\_lon \end{bmatrix}$$

W tym przypadku element (1,1) zawsze położony jest w południowo-zachodnim rogu siatki.

W Mapping Toolbox można znaleźć szereg funkcji wspomagających przechodzenie ze współrzędnych geograficznych na indeksy w macierzy **map** i odwrotnie, tworzenie macierzy lub wektora referencyjnego itp.

Drugi typ siatki, *siatka geograficzna,* wymaga podania szczegółowych danych:

- macierzy wartości parametru
- macierzy szerokości geograficznych punktów
- macierzy długości geograficznych punktów, których dotyczą dane.

Mapping Toolbox stosuje trzy typy interpretacji danych siatki geograficznej:

- macierz danych posiada takie same rozmiary jak macierze współrzędnych geograficznych. W tym przypadku macierze współrzędnych o wymiarach m×n określają obszar zawierający (m-1)×(n-1) komórek i dla takiej ilości komórek dane będą wyświetlane. Wyświetlane wartości są kojarzone z lewym górnym narożnikiem komórki. Pozostałe, niewyświetlone dane są mimo wszystko przechowywane wraz z mapą.
- macierz danych ma o jeden wiersz i jedną kolumnę mniej niż macierze współrzędnych. Ilość danych odpowiada ilości komórek siatki. Wyświetlane wartości są kojarzone ze środkami komórek siatki.
- macierz danych ma rozmiary większe od macierzy współrzędnych. W tym przypadku macierze współrzędnych interpretowane są jako zgrubna siatka, na której będą rozłożone dane.

Istnieją odpowiednie funkcje pomocnicze, pozwalające na wygenerowanie macierzy współrzędnych dla danych w formacie siatki regularnej.
Dane rastrowe najczęściej zawierają informacje o wysokości względnej punktów (tzw. formaty DTM lub DEM). Typowym przykładem danych w siatce regularnej mogą być dane w systemie **gtopo30**, zawierające informacje o wysokości punktów na powierzchni Ziemi w siatce prostokątnej o boku 30" kątowych. Podstawowe dane w tym systemie zawarte są w plikach:

- DEM - dane o wysokościach (digital elevation model),
- HDR - informacje o strefie mapy zobrazowanej w pliku DEM,
- DMW - odpowiednik macierzy referencyjnej,
- STX, PRJ, SRC, SCH - pliki z danymi pomocniczymi.

Do wczytywania danych rastrowych służy cały szereg specjalizowanych funkcji, dopasowanych do poszczególnych formatów.

4.2 Tworzenie układu współrzędnych

Procedury pomocnicze, zawarte w Mapping Toolbox używają dodatkowych atrybutów klasy *układ współrzędnych* (axes). Atrybuty te są wprowadzane w postaci struktury przechowywanej jako wartość **UserData.** Dostęp do poszczególnych pól tej struktury zapewniają funkcje **getm** oraz **setm.** Powołanie nowego układu współrzędnych odbywa się przez użycie polecenia

```
>> axesm projekcja
```

które oprócz utworzenia obiektu *axes* wprowadza odpowiednie wartości do struktury **UserData.** Alternatywnie można użyć polecenia

```
>> worldmap obiekt_geograficzny
```

W wyniku działania tego polecenia powstaje układ współrzędnych z atrybutami projekcji (przechowywanymi w strukturze **UserData**) odpowiednio dobranymi dla określonego obiektu. Dodatkowe atrybuty, związane z prezentacją mapy można podzielić na cztery grupy:

- własności projekcji (map projection)
- własności pola mapy (frame)
- własności siatki geograficznej na mapie
- własności opisów siatki

4.2.1 Atrybuty określające własności projekcji

MapProjection	[łańcuch] - nazwa m-pliku właściwego dla danej projekcji. Dostępne projekcje można otrzymać za pomocą polecenia maps lub getm('MapProjection')
Zone	łańcuch - kod strefy, niezbędny dla niektórych projekcji. Dla UTM świat podzielony jest na strefy o szerokości 6◦ i wysokości 8◦. Kod strefy składa się liczby (1 do 60) oznaczającej długość geograficzną oraz litery(C do X) dla określenia szerokości geograficznej.
AngleUnits	[degrees} | radians | dms] - używane jednostki miary kąta.
Aspekt	[normal | transverse] - przy wartości normal północ (N) skierowana jest ku górze, w przypadku transverse - w prawo. Dla projekcji cylindrycznych normal oznacza układ poziomy, a transverse - układ pionowy rysunku.
FalseEasting	[skalar {0}] - Przesunięcie poziome współrzędnych do obliczeń w stosunku do układu.
FalseNorthing	[skalar {0}] - przesunięcie pionowe współrzędnych do obliczeń w stosunku do układu.
FixedOrient	[skalar {[]}] (tylko do odczytu) - dla niektórych projekcji określa, czy użytkownik może zmieniać orientację układu za pomocą trzeciego parametru atrybutu Origin.
Geoid	[wielka_półoś mimośród] - parametry elipsoidy odniesienia. Domyślnie przyjmuje się kulę ([1 0]).

MapLatLimit	[south north] | [north south] - ograniczenia dla szerokości geograficznej przedstawianej na mapie. Ograniczenia używane przez funkcje wyświetlania tekstur: meshm, surfm, surfacem i pcolrm. Jednocześnie oznacza ograniczenia dla zakresu wyświetlania południków.
MapLonLimit	[west east] - ograniczenia dla długości geograficznej przedstawianej na mapie. Ograniczenia używane przez funkcje wyświetlania tekstur: meshm, surfm, surfacem i pcolrm. Jednocześnie oznacza ograniczenia dla zakresu wyświetlania równoleżników.
MapParallels	[lat] | [lat1 lat2] - standardowe równoleżniki projekcji. Może być wektor pusty, skalar lub wektor dwuelementowy, w zależności od typu projekcji. Wartości mierzone są w jednostkach określonych przez AngleUnits. Przykładowo dla projekcji stożkowych standardowo przyjmowane są wartości 15°N i 75°N. Ustalenie dla takiej projekcji pustego wektora powoduje przeliczenie na wartości odległe o 1/6 od ograniczeń szerokości.
Parallels	0, 1, lub 2 (tylko do odczytu, zależne od projekcji) - wymagana ilość standardowych równoleżników.
Origin	[latitude longitude orientation] - ustawienie parametrów dla wszelkich przeliczeń projekcji. Wartości powinny być wyrażone w jednostkach AngleUnits. Dwa pierwsze atrybuty odnoszą się do współrzędnych odniesienia mapy. Trzeci odnosi się do kąta pochylenia lub obrotu wokół osi przechodzącej przez punkt odniesienia.
ScaleFactor	skalar {1} - współczynnik skalowania mapy. Czasami używany do minimalizacji zniekształceń skali w projekcji - przykładowo projekcja UTM używa współczynnika 0,996.
TrimLat	[south north] (tylko do odczytu, zależne od projekcji) - wartości współrzędnych, poza którymi nie będą wyświetlane elementy mapy. Np. w projekcji Mercatora jest to [86 86] w celu uniknięcia zniekształceń okolic bieguna.
TrimLon	[west east] (tylko do odczytu, zależne od projekcji) - wartości współrzędnych, poza którymi nie będą wyświetlane elementy mapy.

4.2.2 Atrybuty określające własności pola mapy

Frame	on | {off} - widoczność ramki. Ramka jest obiektem typu patch, rysowanym jako najniższa warstwa wykresu.
FFill	skalar {100} - ilość punktów (domyślnie 100) użyta do wykreślenia jednego boku ramki.
FEdgeColor	ColorSpec | {[0 0 0]} - kolor krawędzi ramki - specyfikowana nazwa lub współrzędne barwne RGB.
FFaceColor	ColorSpec | {none} - kolor tła ramki.
FLatLimit	[south north] | [north south] - współrzędne górnej i dolnej krawędzi ramki.
FLineWidth	skalar {2} - grubość linii zastosowana do rysowania krawędzi ramki.
FlonLimit	[east west] | [west east] - współrzędne lewej i prawej krawędzi ramki.

4.2.3 Atrybuty określające własności siatki

Grid	on | {off} - widoczność siatki (południków i równoleżników).
GAltitude	skalar {Inf} - wysokość warstwy, na której umieszcza się siatkę. Domyślnie jest to najwyższa warstwa mapy.
GColor	ColorSpec | {[0 0 0]} - kolor linii siatki.
GLineStyle	LineStyle {:} - styl linii siatki (domyślnie linia kropkowa).
GLineWidth	skalar {0.5} - grubość linii siatki.

MLineException	wektor {[]} - współrzędne południków, które mogą być rysowane aż do biegunów, nawet poza ograniczeniami siatki.
MLineFill	skalar {100} - ilość punktów (domyślnie 100) użyta do wykreślenia jednego południka.
MLineLimit	[north south] | [south north] {[]} - ograniczenia wyświetlania południków.
MLineLocation	skalar | wektor {30°} - skalar oznacza skok siatki w kierunku poziomym (liczony od południka 0°). Wektor oznacza wybór południków do wyświetlenia.
PLineException	wektor {[]} - współrzędne równoleżników, które mogą być rysowane nawet poza ograniczeniami siatki.
PLineFill	skalar {100} - ilość punktów (domyślnie 100) użyta do wykreślenia jednego równoleżnika.
PLineLimit	[east west] | [west east] {[]} - ograniczenia wyświetlania równoleżników.
PLineLocation	skalar | wektor {15°} - skalar oznacza skok siatki w kierunku pionowym (liczony od równika). Wektor oznacza wybór równoleżników do wyświetlenia.

4.2.4 Atrybuty określające opisy linii siatki

FontAngle	[{normal} | italic | oblique] - wybór pochylenia czcionki (domyślnie czcionka prosta).
FontColor	ColorSpec | {black} - kolor czcionki opisu - specyfikowana nazwa lub współrzędne barwne RGB.
FontName	[courier | {helvetica} | symbol | times] - nazwa czcionki opisu siatki.
FontSize	skalar {9} - wielkość czcionki wyrażona w jednostkach FontUnits.
FontUnits	[{points} | normalized | inches | centimeters | pixels] - jednostki wielkości czcionki. W przypadku normalized wartość jest interpretowana jako ułamek wysokości układu współrzędnych.
FontWeight	[bold | {normal}] - waga czcionki opisu osi.
LabelFormat	[{compass} | signed | none] - format opisów siatki. W przypadku compass do liczb dopisywane są przyrostki N lub S dla równoleżników i E lub W dla południków. Jeżeli wybrano signed wyświetlany jest znak (+ lub -) w zależności od położenia. Wartość none oznacza wyświetlanie jedynie znaku - dla południa i zachodu.
LabelRotation	[on | {off}] - określa, czy opisy wyświetlane są bez rotacji, czy też z rotacją, aby być zgodne z liniami siatki.
LabelUnits	[{degrees} | radians | dms | dm] - jednostki wyświetlania współrzędnych.
MeridianLabel	[on | {off}] - widoczność linii południków.
MLabelLocation	[skalar | wektor] - skalar oznacza skok wyświetlania opisów siatki w kierunku poziomym. Wektor oznacza wybór etykiet południków do wyświetlenia.
MlabelParallel	[{north} | south | equator | skalar] - skalar = równoleżnik wybrany do rozmieszczenia opisów południków. North - górna krawędź, south - dolna krawędź, equator - na równiku.
MlabelRound	[integer {0}] - zaokrąglenie (potęga liczby 10) wyświetlanych opisów południków.
ParallelLabel	[on | {off}] - widoczność linii równoleżników.
PlabelLocation	[skalar | wektor]

Skalar oznacza skok wyświetlania opisów siatki w kierunku pionowym. Wektor oznacza wybór etykiet równoleżników do wyświetlenia.

PlabelMeridian	[east | {west} | prime | skalar] - skalar - południk wybrany do rozmieszczenia opisów równoleżników. East - prawa krawędź, west - lewa krawędź, prime - południk Greenwich.
PlabelRound	integer {0} - zaokrąglenie (potęga liczby 10) wyświetlanych opisów równoleżników.

4.2.5 Użytkowanie układów współrzędnych

Po utworzeniu układu współrzędnych geograficznych w określonej projekcji można zmieniać atrybuty układu za pomocą polecenia **setm**(*nazwa_atrybutu, wartość_atrybutu*…). Można nawet zmieniać zastosowaną projekcję (atrybut 'MapProjection'), pamiętając że poszczególne projekcje mogą wymagać wprowadzenia/zmiany niektórych z pozostałych atrybutów. Do umieszczania w takim układzie współrzędnych obiektów geograficznych służy zestaw funkcji, dostarczonych wraz z Mapping Toolbox:

- contourm
- contour3m
- fillm
- fill3m
- framem
- gridm
- linem
- meshm
- patchm
- plotm
- plot3m
- surfm
- surfacem
- textm

Funkcje te działają podobnie jak ich odpowiedniki (bez litery **m** na końcu), ale stosują atrybuty specjalizowane, zawarte w strukturze **UserData**. Utwórzmy np. układ współrzędnych dla przedstawienia całej kuli ziemskiej w projekcji Millera:

```
>> axesm miller;
>> framem on;
>> gridm on;
>> showaxes;
>> grid off;
```

Otrzymamy w efekcie okno rysunku, zawierające układ współrzędnych geograficznych z zaznaczoną siatką równoleżników i południków. Jednocześnie na rysunku będą przedstawione osie x-y (normalnie ukrywane przy prezentacji map). Widać też, że funkcje **grid** i **gridm** działają niezależnie. Również niezależnie można uzyskać opisy osi x-y oraz latitude-longitude:

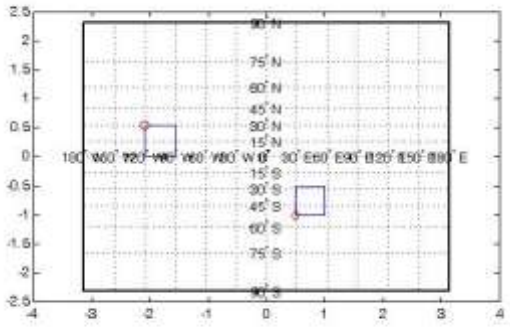
```
>> plabel on;
>> mlabel on;
>> setm(gca,'MlabelParallel','equator','PlabelMeridian',0)
```

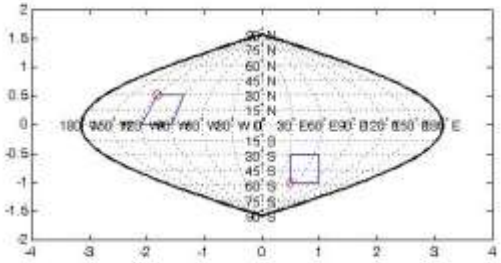
W tak przygotowanym układzie współrzędnych możemy umieszczać obiekty za pomocą standardowych funkcji MATLAB-a (we współrzędnych x-y) albo za pomocą funkcji z Mapping Toolbox (we współrzędnych geograficznych lat-lon):

```
>> h(1) = plot(.5,-1,'dr');
>> h(2) = plotm(0,-120,'or');
>> x = [.5 1 1 .5 .5 .5];
>> y = [-1 -1 -.5 -.5 -1];
>> h(3) = line(x,y);
>> lon = [-120 -90 -90 -120 -120];
>> lat = [0 0 30 30 0];
>> h(4) = linem(lat,lon);
```

Jeśli teraz zmienimy typ projekcji

```
>> setm(gca,'MapProjection','sinusoid');
>> showaxes
```





to punkty wprowadzone wg współrzędnych geograficznych przesuną się do nowych położeń, zgodnych z nową projekcją. Punkty oparte na współrzędnych x-y pozostaną na swoich miejscach. Wynika to z faktu, że funkcje w Mapping Toolbox dokonują przeliczenia wejściowych współrzędnych geograficznych na atrybuty **Xdata** oraz **Ydata** obiektów. Zauważmy, że w przypadku linii, dotyczy to wyłącznie początków i końców odcinków, bo tylko one są zapamiętane jako atrybuty. Chcąc nanieść na mapie linię, której kształt dostosuje się do zmiany projekcji, musimy każdy jej segment podzielić na większą liczbę odcinków, wprowadzając dodatkowe punkty pośrednie.

Przy użytkowaniu map często powstaje zagadnienie wyznaczenia trasy łączącej dwa punkty. Mapping Toolbox dostarcza funkcje pozwalające na wyznaczenie azymutu początkowego i odległości między punktami (**reckon**, **distance**, **azimuth**), kąta elewacji (**elevation**) a nawet wyznaczenie trasy jako zbioru współrzędnych punktów pośrednich (**scircle**, **track1**, **track2**). Wyznaczone trasy mogą przebiegać po ortodromie (w geometrii sferycznej jest to łuk koła wielkiego - *great circle*, czyli najkrótsza linia łącząca dwa punkty na powierzchni kuli) albo loksodromie (w geometrii sferycznej jest to linia przecinająca wszystkie południki pod tym samym kątem - *rhumb line*, czyli linia stałego azymutu). Wymienione funkcje używają miary kątowej do określania odległości, zatem przydatne są funkcje przeliczania między jednostkami kątowymi i jednostkami długości (**deg2km**, **km2deg** i in.).

5 Obsługa błędów

Bardzo często pożądane jest, aby podejmować określoną akcję przy wystąpieniu określonego błędu. Na przykład należy nakazać wprowadzenie większej liczby danych lub przeprowadzić ponowne obliczenia przy domyślnych wartościach. Obsługa błędów pozwala programowi sprawdzenie warunków wystąpienia błędu i wykonanie odpowiedniego kodu programu.

5.1 Polecenia try - catch

Do obsługi sytuacji w której wystąpił błąd podczas realizacji programu służy blok **try-catch.** Blok ten podzielony jest na dwie części. Pierwsza z nich rozpoczyna się poleceniem **try** a druga - **catch.** Cały blok kończy się poleceniem **end.**

- Wszystkie instrukcje w części **try** są realizowane normalnie, jak w pozostałej części programu. Jednak w przypadku wystąpienia błędu, MATLAB kończy egzekucję programu i przechodzi do drugiej części bloku.
- Część **catch** służy do obsługi błędu. W najprostszym przypadku może w niej nastąpić wyświetlenie komunikatu o błędzie. Jeżeli mogły nastąpić różne błędy, procedura obsługi może zidentyfikować błąd i odpowiednio na niego zareagować.

Blok **try-catch** może być również zagnieżdżony:

```
try
    wyrażenie1                  % Próba wykonania
catch
    try
        wyrażenie2              % Próba powrotu z błędu
    catch
        disp 'Operation failed'       % Obsługa błędu
    end
end
```

5.2 Obsługa błędów i powrót do programu

W przypadku, gdy jedyną reakcją na błąd powinna być sygnalizacja i zatrzymanie programu, można użyć funkcji **error** , jak w przykładzie poniżej:

```
x = wyrażenie
if x < 1
    error('x musi być równe 1 lub większe!')
end
następne instrukcje
```

Jeżeli obliczona wartość x jest mniejsza niż 1, program się zatrzyma, wyświetlając komunikat:

```
??? x musi być równe 1 lub większe!
```

Argument funkcji **error** są traktowane jak łańcuch formatu i argumenty dla funkcji MATLAB-a **sprintf** służącej do sformatowanego wyświetlania łańcucha, jak np. w przypadku:

```
error('Pliku %s nie znaleziono', plik)
```

W łańcuchu formatu mogą wystąpić znaki specjalne (jak %f czy \n), ale będą one interpretowane tylko w przypadku, gdy wystąpi więcej niż jeden argument funkcji **error.**

Wyświetlana wiadomość może zawierać dodatkową informację kategoryzującą:

```
error('Test:BrakPliku', 'Pliku %s nie znaleziono', plik)
```

Pierwszy z argumentów (niewyświetlany) jest traktowany jak identyfikator komunikatu i może zawierać wyłącznie znaki alfanumeryczne (bez spacji i np. polskich znaków diakrytycznych). Identyfikator ten składa się z identyfikatora programu (systemu) w którym wystąpił błąd oraz identyfikatora kategorii błędu (może być wielopoziomowy). Znakiem rozdzielającym poszczególne części jest dwukropek. Identyfikator komunikatu może być wykorzystywany przez funkcję **lasterr**, która zapamiętuje ten identyfikator oraz treść komunikatu i zwraca je na żądanie. W wersji 7.0 MATLAB-a funkcja ta zostaje zastąpiona przez **lasterror**, która zwraca strukturę

- message - treść komunikatu o błędzie
- identifier - identyfikator komunikatu
- stack - struktura zawierająca rekordy o polach:

- file - nazwa pliku
- name - nazwa funkcji
- line - numer wiersza, w którym powstał błąd

Struktura **stack** identyfikuje miejsce powstania błędu (nazwa pliku, numer wiersza) dla wszystkich poziomów hierarchii wywołania funkcji (pliku), w której powstał błąd. Wywołanie instrukcji **rethrow** z argumentem **lasterror** w części **catch** bloku **try-catch** powoduje przerwanie pracy programu i wyświetlenie komunikatu błędu, w przeciwnym przypadku wykonywane byłyby instrukcje umieszczone po tym bloku:

```
try
    instrukcje_programu
catch
    instrukcje_regeneracji
    rethrow(lasterror)       % Tu koniec po wystąpieniu błędu
end
dalsze_instrukcje
```

5.3 Ostrzeżenia

W MATLAB-ie dostępna jest funkcja **warning**, która wyświetla komunikat o wystąpieniu nieprzewidzianych okoliczności podczas wykonywania programu. Składnia wywołania tej funkcji jest podobna do funkcji **error**. Różnica w działaniu polega na tym, że po wyświetleniu komunikatu praca programu jest kontynuowana. Treść komunikatu przechowywana jest przez funkcję **lastwarn.**
MATLAB daje możliwość sterowania sposobem reagowania na wystąpienie ostrzeżenia:

- Wyświetlanie wybranych ostrzeżeń
- Ignorowanie wybranych ostrzeżeń
- Zatrzymanie debuggera przy zaistnieniu ostrzeżenia
- Wyświetlenie ścieżki wywołań funkcji po zaistnieniu ostrzeżenia.

Podobnie jak w przypadku funkcji **error** istnieje możliwość dodania identyfikatora do ostrzeżenia, o takiej samej składni. Do ustalenia sposobu reakcji na ostrzeżenie służy wyrażenie:

```
warning status identyfikator
```

Argument *status* może przyjmować wartości:

- on - zezwolenie na wyświetlenie danego ostrzeżenia
- off - zakaz wyświetlania danego ostrzeżenia
- query - wyświetlenie stanu obsługi danego ostrzeżenia.

Argument *identyfikator* może być łańcuchem identyfikującym ostrzeżenie, bądź kwantyfikatorem *all*, oznaczającym wszystkie ostrzeżenie lub *last* - zastępującym identyfikator ostatniego ostrzeżenia.
W miejsce argumentu *identyfikator* może być wprowadzony argument *tryb*, przyjmujący wartości:

- debug - zatrzymanie debuggera przy zaistnieniu ostrzeżenia
- backtrace - wyświetlenie ścieżki wywołań funkcji po zaistnieniu ostrzeżenia
- verbose - wyświetlenie dodatkowo informacji, jak zablokować ostrzeżenie.

Przykład użycia trybu **verbose**:

```
>> warning on verbose
>> A=25/0;
Warning: Divide by zero.
(Type "warning off MATLAB:divideByZero" to suppress this warning.)
>>
```

5.4 Wyjątki

Kiedy MATLAB napotka błąd w trakcie realizacji programu, powoduje wygenerowanie (wyrzucenie) wyjątku. W tym procesie MATLAB

- Przerywa działanie programu w miejscu wystąpiena błędu.
- Tworzy obiekt klasy MException.
- Zapisuje w atrybutach tego obiektu informację o błędzie.
- Wyświetla tę informację na konsoli użytkownika.
- Zatrzymuje program.

W MATLAB-ie wersji 2008 dostępna jest obsługa błędów poprzez przechwytywanie wyjątków przez funkcje realizujące blok **try-catch**.

```
>> try
    sin
catch ex
    disp(ex.message)
end

Not enough input arguments.
```

Obiekt *ex* utworzony w wyniku zaistnienia błędu w bloku **try**, jest instancją klasy MException i posiada następujące pola:

identifier - ciąg znaków identyfikujących rodzaj błędu, w tym przypadku 'MATLAB:minrhs'
message - wyświetlany komunikat 'Not enough input arguments'
cause - tablica komórek, zawierająca obiekty klasy MException związane z kaskadowym generowaniem wyjątków,
stack - struktura, zawierająca rekordy o polach:

- file - nazwa pliku, zawierającego funkcję wykrywającą błąd,
- name - nazwa funkcji, w której wykryto błąd
- line - numer wiersza (w pliku *file*), w którym powstał błąd

W strukturze *stack* może być wiele rekordów, jeżeli informacja o błędzie była przekazywana kaskadowo przez kilka funkcji.

Z klasą MException związane są następujące metody:

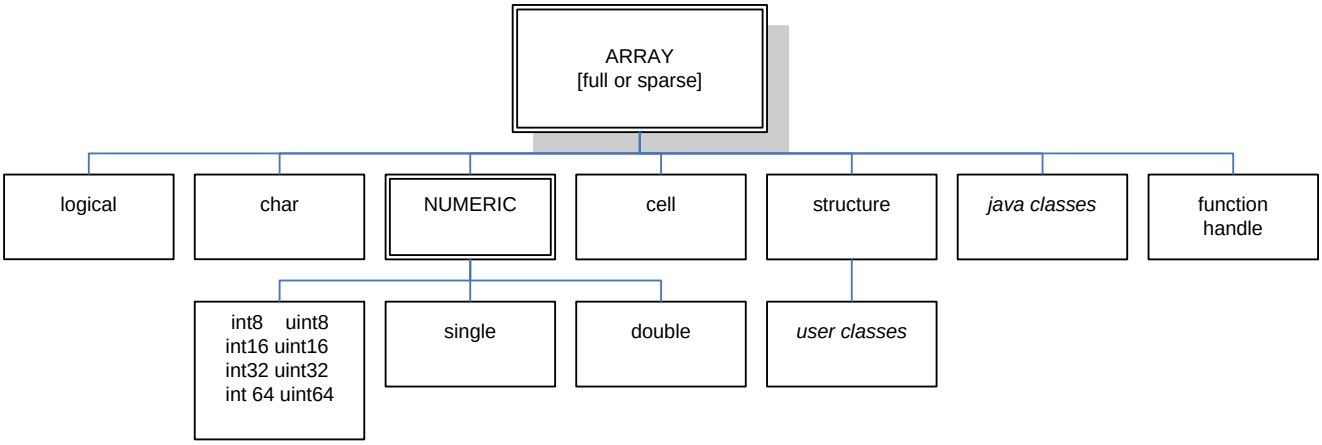
AddCause - dodaje nowy element do pola *cause*,
eq - porównanie dwóch obiektów klasy na identyczność, (przeciążenie operatora ==),
getReport - zwraca informację o błędzie, w formacie używanym przez MATLAB,
isequal - porównanie dwóch obiektów klasy na identyczność,
last - zwraca ostatnio wygenerowany obiekt klasy,
ne - porównanie dwóch obiektów klasy na nieidentyczność, (przeciążenie operatora ~=),
rethrow - uruchomienie wyjątku dla wywołania zatrzymania programu,
throw - tworzy wyjątek dla aktualnie uruchomionego m-pliku,
throwAsCaller - tworzy wyjątek dla pliku, pomijając bieżącą strukturę *stack*.

6 Klasy i obiekty

Klasy można traktować jak nowe typy danych o zdefiniowanej specyfice zachowania. Operacje określone do działania na obiektach danej klasy nazywamy **metodami** tej klasy.
Nowe obiekty uzyskujemy w wyniku powołania instancji danej klasy.
W środowisku programu MATLAB możemy dodać nową klasę przez zdefiniowanie struktury, zapewniającej przechowywanie danych związanych z instancją klasy oraz utworzenie folderu klasy, zawierającego m-pliki działające na tych danych. Te m-pliki zawierają metody klasy. Folder klasy może również zawierać funkcje definiujące w jaki sposób różne operatory MATLAB-a będą stosowane do obiektów. Przedefiniowanie sposobów działania operatorów wbudowanych nazywamy **przeciążeniem**.
Podstawowe cechy programowania zorientowanego obiektowo to:

- przeciążanie funkcji i operatorów,
- hermetyzacja danych i metod,
- dziedziczenie (proste i wielokrotne),
- agregacja obiektów wewnątrz innych obiektów.

Istniejące w MATLAB-ie typy danych są zaprojektowane w taki sposób, aby mogły funkcjonować jako klasy w programowaniu zorientowanym obiektowo. Hierarchia klas przedstawiona została na rysunku poniżej.



Hierarchia ta może być rozszerzana przez dodawanie klas użytkownika. Jak widać na rysunku, domyślną klasą bazową dla klasy użytkownika jest klasa **structure**.

6.1 Praca z obiektami

Zdefiniowanie nowej klasy użytkownika wymaga utworzenia folderu klasy (o nazwie *@nazwa_klasy*) i umieszczeniu w nim m-plików zawierających metody obsługujące klasę. Metody są m-funkcjami, które pobierają obiekt jako jeden z argumentów wejściowych. Obiekt (instancja klasy) może zostać utworzony przez uruchomienie specjalnej metody (konstruktora) i przekazanie do niej odpowiednich argumentów wejściowych. W MATLAB-ie konstruktory mają taką samą nazwę jak klasa.
Przykładowo wyrażenie:

```
p = polynom([1 0 -2 -5]);
```

powołuje obiekt *p* należący do klasy **polynom**. Po utworzeniu obiektu można działać na nim za pomocą metod zdefiniowanych dla klasy **polynom**. Przechowywane są one w folderze klasy (o nazwie **@polynom**). Folder ten jest przeszukiwany przez MATLAB w pierwszej kolejności. Składnia wywołania metody w przypadku ogólnym ma postać:

```
[out1, out2,...] = nazwa_metody(obiekt, arg1, arg2,...);
```

Niektóre metody, nazywane **prywatnymi**, mogą być wywoływane wyłącznie przez inne metody swojej klasy - nie można ich wywołać z wiersza poleceń ani z metod innych klas, włącznie z klasą bazową. Metody prywatne są umieszczane w folderze *@nazwa_metody*\private. W folderze klasy

może występować jeszcze jeden rodzaj funkcji - nie będących metodami i nie działających bezpośrednio na obiekcie, ale wspomagających działanie metod. Są to tzw. funkcje **pomocnicze.**
Po utworzeniu folderu klasy, należy zmodyfikować ścieżkę przeszukiwań MATLAB-a. Jeżeli na przykład folder @polynom zawierający metody klasy znajduje się w

```
c:\MyClasses\@polynom
```

dodajemy do ścieżki MATLAB-a folder

```
addpath c:\MyClasses
```

Obiekty przechowywane są jako struktury. Pola tych struktur i szczegóły operacji na polach są widziane tylko z wnętrza metod klasy. Odpowiednie zaprojektowanie struktury danych ma wpływ na jakość tworzonego oprogramowania.
Istnieje szereg różnic, między modelem programowania obiektowego MATLAB-a a tym oferowanym przez kompilatory C++ lub Java:

- wybór metody dokonywany jest na podstawie pierwszego z lewej argumentu
- nie ma metody - odpowiednika destruktora klasy. Należy używać funkcji **clear** do usunięcia obiektu z przestrzeni danych,
- konstruowanie typów danych występuje w fazie realizacji a nie kompilacji. Przynależność obiektu do klasy rejestruje się za pomocą funkcji **class**
- relacja dziedziczności jest tworzona w klasie pochodnej przez utworzenie obiektu klasy bazowej a następnie wywołanie funkcji **class.** Obiekt pochodny zawiera obiekt bazowy w atrybucie o nazwie klasy bazowej
- w MATLAB-ie nie ma przekazywania zmiennych przez referencje. Przy tworzeniu metody modyfikującej obiekt musimy zwrócić obiekt zmodyfikowany i użyć instrukcji podstawienia
- nie ma odpowiednika klasy abstrakcyjnej
- nie ma wirtualnego dziedziczenia ani wirtualnych klas bazowych

6.2 Projektowanie klasy użytkownika

Przy projektowaniu nowej klasy w MATLAB-ie należy dołączyć standardowy zestaw metod, pozwalający na spójne i logiczne zachowanie obiektu w środowisku. Poniżej wymieniono podstawowe metody występujące w klasach MATLAB-a:

Metoda klasy	**Opis**
konstruktor_klasy	Tworzy nowy obiekt (instancję) klasy
display	Wywoływana, gdy MATLAB wyświetla zawartość obiektu (np. w sytuacji, gdy wyrażenie nie kończy się średnikiem)
set, get	Dają dostęp do atrybutów obiektu
subsref, subsasgn	Umożliwiają dostęp indeksowy do obiektów użytkownika
end	Daje możliwość zakończenia wyrażeń indeksowych, np. A(1:end)
subsindex	Umożliwia użycie obiektu w wyrażeniach indeksowych
konwertery	Metody zamieniające obiekt na odpowiedni typ danych, np. double czy char

W zależności od potrzeb można utworzyć niektóre z wymienionych metod i dołączyć szereg nowych metod, realizujących zamierzone cele projektu. Metody dostępne dla danej klasy można wyświetlić za pomocą funkcji **methods**('*nazwa_klasy*').

6.2.1 Konstruktor klasy

Folder @ dla danej klasy musi zawierać m-plik o nazwie zgodnej z nazwa folderu (za wyjątkiem oczywiście przedrostka @), nazywany **konstruktorem** dla danej klasy. Konstruktor tworzy obiekt przez zainicjowanie struktury danych i powołanie instancji klasy. W ogólności konstruktor powinien obsługiwać trzy możliwe kombinacje argumentów wejściowych:

- **Brak argumentów**. W takiej sytuacji konstruktor powinien utworzyć obiekt o domyślnych wartościach atrybutów. Taka składnia ma zastosowanie:

- przy ładowaniu obiektu do przestrzeni danych, funkcja **load** wywołuje konstruktor klasy bez argumentów,
- przy tworzeniu tablicy obiektów, MATLAB uruchamia konstruktor klasy, aby dodać obiekt do tablicy.

- **Argumentem jest obiekt tej samej klasy**. Jeżeli pierwszym elementem z listy argumentów jest obiekt tej samej klasy, konstruktor powinien po prostu zwrócić obiekt. Do stwierdzenia, czy dany obiekt należy do danej klasy można użyć funkcji **isa**(*obiekt*, '*nazwa_klasy*'). Taka składnia ma często miejsce przy przeciążaniu operatorów.
- **Argumentem jest wektor danych**. Jeśli istnieją argumenty wejściowe i nie są obiektami danej klasy, to konstruktor tworzy obiekt przy pomocy danych wejściowych. Oczywiście pożądana jest odpowiednia kontrola danych. Typowo stosuje się argument wejściowy **varargin** i instrukcję **switch** do sterowania pracą programu. W tej części procedury przypisuje się wartości polom struktury danych obiektu, uruchamia funkcję **class** dla powołania instancji obiektu i zwraca obiekt jako argument wyjściowy. W razie potrzeby można użyć funkcji **superiorto** oraz **inferiorto** dla odpowiedniego umieszczenia obiektu w hierarchii.

Wywołana wewnątrz konstruktora funkcja **class** powoduje przypisanie obiektowi wewnętrznej etykietki klasy, dostępnej jedynie za pomocą funkcji **class** oraz **isa**. Przykładowo poniższy zapis przypisuje obiekt *p* do klasy **polynom**:

```
p = class(p, 'polynom');
```

Poza folderem klasy do identyfikacji klasy obiektu może służyć funkcja **isa**:

```
>> isa(p, ,polynom')
ans =
   true
```

Podobnie można użyć funkcji **class** (poza konstruktorem funkcja ta przyjmuje tylko jeden argument wejściowy):

```
>> class(p)
ans =
  polynom
```

Funkcja **whos** wyświetla dane o obiekcie:

```
>> whos p
  NameSize            Bytes      Class
  p   1x1               156      polynom object
>>
```

Przykład konstruktora dla klasy **polynom**:

```
function p = polynom(a)
% POLYNOM Polynomial class constructor
%   p = POLYNOM(v) creates a polynomial object from the vector v,
%   containing the coefficients of descending powers of x.
if nargin == 0
   p.c = [];
   p = class(p,'polynom');
elseif isa(a,'polynom')
   p = a;
else
   p.c = a(:).';
   p = class(p,'polynom');
end
```

6.2.2 Metoda display

MATLAB wywołuje metodę **display** za każdym razem, gdy obiekt jest wynikiem wyrażenia nie zakończonego średnikiem. W wielu klasach metoda **display** może po prostu wypisać nazwę zmiennej, a następnie użyć konwertera do typu char dla wyświetlenia zawartości lub wartości zmiennej, ponieważ MATLAB wyprowadza dane wyjściowe jako ciągi znaków. W takim przypadku musi być uprzednio zdefiniowana metoda **char**, konwersji danych obiektu na ciąg znaków.

```
function display(p)
% POLYNOM/DISPLAY Command window display of a polynom
disp(' ');
disp([inputname(1),' = '])
```

```
        disp(' ');
        disp(['   ' char(p)])
        disp(' ');
```

6.2.3 Dostęp do danych obiektu

Aby zapewnić dostęp do danych obiektu należy zdefiniować odpowiednie metody. Metody dostępu mogą stosować różne sposoby, ale każdorazowo metoda, która zmienia dane obiektu, pobiera obiekt jako argument wejściowy i zwraca nowy obiekt ze zmienionymi danymi. Funkcje mogą zmieniać wyłącznie własną, chwilową prywatną kopię obiektu. Dlatego aby zmienić istniejący obiekt należy utworzyć nowy, a następnie zastąpić nim obiekt istniejący.
Metody **get** i **set** umożliwiają dotarcie w wygodny sposób do odpowiednich danych obiektu, izolując jednocześnie inne pola struktury danych od bezpośredniego dostępu z programu. Innym sposobem jest utworzenie metod do obsługi konkretnego atrybutu obiektu. Taka metoda powinna nosić nazwę taką samą jak obsługiwany atrybut.

6.2.4 Dostęp indeksowy do obiektu

MATLAB pozwala na dostęp do instancji klas użytkownika za pośrednictwem indeksu, podobnie jak ma to miejsce w przypadku typów wbudowanych. Przykładowo, jeżeli A jest macierzą klasy double, to wyrażenie A(i) zwraca i-ty element macierzy A.
Programista projektujący klasę decyduje, jakie mają być skutki indeksowego odwołania do obiektu. Na przykład dla wspomnianej wyżej klasy **polynom** wyrażenie

```
p(3)
```

mogłoby oznaczać wyświetlenie współczynnika przy x^3, wartość wielomianu dla x = 3 lub cokolwiek innego. Definiowane jest to za pomocą metod **subsref** i **subsasgn**. Jeżeli metody te nie zostaną zaimplementowane, to dostęp indeksowany nie będzie obsługiwany.
W składni MATLAB-a możemy wyróżnić trzy sposoby dostępu - pobrania danych - za pomocą wyrażenia indeksowego *I*:

- indeks w tablicy - A(*I*),
- indeks w tablicy komórek - A{*I*},
- nazwa pola w strukturze - A.*pole*.

W wyniku każdej z wymienionych sytuacji następuje wywołanie metody **subsref** z foldera klasy. MATLAB przekazuje przy wywołaniu dwa argumenty:

```
B = subsref(A,S)
```

Pierwszym argumentem jest obiekt, do którego się odnosimy. Drugi argument, S, jest tablicą struktur zawierających dwa pola:

- **S.type** jest łańcuchem definiującym sposób dostępu.
  - '()' dla tablicy,
  - '{}' dla tablicy komórek,
  - '.' dla tablicy struktur.
- **S.subs** jest tablicą komórek lub łańcuchem zawierającym aktualne wartości indeksów. Znak dwukropka użyty jako indeks jest przekazywany w postaci łańcucha ':'.

Na przykład wyrażenie:

```
A(1:2,:)
```

powoduje wywołanie funkcji **subsref(A,S)**, gdzie S jest jednoelementową strukturą o polach:
S.type = '()'
S.subs = {1:2,':'}
Z kolei wyrażenie:

```
A(1,2).nazwa(3:4)
```

wywołuje **subsref(A,S)**, gdzie
S(1).type = '()' S(2).type = '.' S(3).type = '()'
S(1).subs = {1,2} S(2).subs = 'nazwa' S(3).subs = {3:4}
Oto fragment kodu funkcji **subsref** określający typ dostępu i interpretujący argumenty wejściowe:

```
switch S.type
case '()'
   B = A(S.subs{:});
```

```
case '{}'
   B = A(S.subs{:});      % A jest tablicą komórek
case '.'
   switch S.subs          % A jest tablicą struktur o polach pole1 i
pole2
   case 'pole1'
      B = A.pole1
   case 'pole2'
      B = A.pole2
   end
end
```

Przykładowo, dla klasy **polynom** założono, że jedyny sposób dostępu - tablica numeryczna - ma zwracać wartość wielomianu dla wprowadzonej wartości x. Metoda **subsref** realizuje to następująco:

```
function b = subsref(a,s)
%POLYNOM/SUBSREF
switch s.type
case '()'
   ind = s.subs{:};
   for k = 1:length(ind)
      b(k) = eval(strrep(char(a),'x',['(' num2str(ind(k)) ')']));
   end
   % lepiej użyć b = a.polyval(ind);
otherwise
   error('Specify value for x as p(x)')
end
```

W przypadku, gdy ma nastąpić podstawienie wartości na element wskazany indeksem (tu wystąpią te same trzy typy dostępu)

- A(*I*) = B
- A{*I*} = B
- A.*pole* = B

wywołana zostanie metoda **subsasgn(A,S,B)**. Dwa pierwsze argumenty mają to samo znaczenie co powyżej, natomiast trzeci argument, B, jest nową wartością.
Jeżeli dostęp indeksowy realizowany jest wewnątrz metody klasy, MATLAB używa swojej funkcji wbudowanej, natomiast gdy metoda sięga do danych innej klasy, zostanie wywoływana przeciążona funkcja odpowiedniej klasy.

6.2.5 Określenie końca zakresu indeksu

Kiedy w wyrażeniu indeksowym używamy słowa *end*, MATLAB wywołuje zdefiniowaną dla klasy metodę **end(A,k,n)**. Argumentami są:

- A - obiekt użytkownika,
- k - pozycja indeksu w którym użyto słowa *end*, w wyrażeniu indeksowym,
- n - całkowita ilość indeksów w wyrażeniu.

Przykładowo, dla wyrażenia:

```
A(end-1,:)
```

MATLAB wywołuje metodę **end**:

```
end(A,1,2)
```

co oznacza, że dla obiektu A użyto słowa *end* w pierwszym z dwóch indeksów.

6.2.6 Indeksowanie za pomocą innego obiektu

Kiedy MATLAB napotka obiekt jako indeks, np. mamy obiekt *a* i chcemy go użyć jako indeksu w innym obiekcie *b*:

```
c = b(a)
```

wywołuje metodę **subsindex** zdefiniowaną dla klasy obiektu *a*. W najprostszym przypadku metoda ta wykonuje konwersję do typu double (wartości zwracane mogą rozpoczynać się od 0, a nie od 1).

6.2.7 Konwertery

Konwerter jest to taka metoda klasy, której nazwa jest identyczna z nazwą innej klasy. Konwerter akceptuje obiekt jednej klasy jako argument wejściowy i zwraca obiekt innej klasy jako argument wyjściowy. Konwertery umożliwiają:

- użycie metod zdefiniowanych dla innych klas
- zapewnienie prawidłowego wykonania wyrażenia zawierającego obiekty różnych klas.

Wywołanie funkcji konwertera ma postać:

```
b = nazwa_klasy(a)
```

gdzie a jest obiektem klasy innej niż *nazwa_klasy*. W tym przypadku MATLAB poszukuje metody o nazwie *nazwa_klasy* w folderze klasy obiektu *a*. Jeżeli obiekt *a* jest klasy *nazwa_klasy*, to wywoływany jest konstruktor klasy, który w takim przypadku zwraca po prostu obiekt *a*. Przykładowo konwerter typu **polynom** do typu **double**:

```
function c = double(p)
% POLYNOM/DOUBLE Convert polynom object to coefficient vector.
%   c = DOUBLE(p) converts a polynomial object to the vector c
%   containing the coefficients of descending powers of x.
c = p.c;
```

Konwerter obiektu **polynom** do postaci znakowej zwraca łańcuch znaków przedstawiający wielomian zmiennej x. W ten sposób, dla danej wartości zmiennej x uzyskany łańcuch jest prawidłowym wyrażeniem MATLAB-a, którego wartość można wyznaczyć za pomocą funkcji wbudowanej **eval**:

```
function s = char(p)
% POLYNOM/CHAR
% CHAR(p) is the string representation of p.c
if all(p.c == 0)
   s = '0';
else
   d = length(p.c) - 1;
   s = [];
   for a = p.c;
      if a ~= 0
         if ~isempty(s)
            if a > 0
               s = [s ' + '];
            else
               s = [s ' - '];
               a = -a;
            end
         end
         if a ~=1 | d == 0
            s = [s num2str(a)];
            if d > 0
               s = [s '*'];
            end
         end
         if d >= 2
            s = [s 'x^' int2str(d)];
         elseif d == 1
            s = [s 'x'];
         end
      end
      d = d - 1;
   end
end
```

Na przykład obiekt powołany za pomocą instrukcji:

```
p = polynom([1 -2 1]);
```

może być przedstawiony w postaci wielomianu zmiennej x:

```
char (p)
ans =
   x^3 - 2*x^2 + 1
```

6.2.8 Przeciążanie operatorów i funkcji

Każdy wewnętrzny operator MATLAB-a posiada skojarzoną nazwę funkcji (np. operator + ma przypisaną funkcję **plus.m**). Aby przeciążyć operator, należy utworzyć m-plik zawierający nową funkcję, nadać mu odpowiednią nazwę i umieścić go w folderze klasy. Przykładowo, jeżeli *a* lub *b* jest obiektem klasy *nazwa_klasy*, to w przypadku wyrażenia:

```
a + b
```

wywoływana jest funkcja ***@nazwa_klasy*/plus.m**, jeżeli taka istnieje. Jeżeli oba są obiektami różnych klas, MATLAB stosuje reguły kolejności działań dla określenia, która metoda będzie użyta.
W poniższej tabeli przedstawiono nazwy funkcji odpowiadające operatorom wbudowanym:

Działanie	M-plik	Opis
`a + b`	`plus(a,b)`	Dodawanie
`a - b`	`minus(a,b)`	Odejmowanie
`-a`	`uminus(a)`	Plus jednoargumentowy
`+a`	`uplus(a)`	Minus jednoargumentowy
`a.*b`	`times(a,b)`	Mnożenie skalarne
`a*b`	`mtimes(a,b)`	Mnożenie macierzowe
`a./b`	`rdivide(a,b)`	Prawostronne dzielenie skalarne
`a.\b`	`ldivide(a,b)`	Lewostronne dzielenie skalarne
`a/b`	`mrdivide(a,b)`	Prawostronne dzielenie macierzowe
`a\b`	`mldivide(a,b)`	Lewostronne dzielenie macierzowe
`a.^b`	`power(a,b)`	Potęgowanie skalarne
`a^b`	`mpower(a,b)`	Potęgowanie macierzowe
`a < b`	`lt(a,b)`	Mniejsze
`a > b`	`gt(a,b)`	Większe
`a <= b`	`le(a,b)`	Mniejsze lub równe
`a >= b`	`ge(a,b)`	Większe lub równe
`a ~= b`	`ne(a,b)`	Nie równe
`a == b`	`eq(a,b)`	Równe
`a & b`	`and(a,b)`	Iloczyn logiczny
`a | b`	`or(a,b)`	Suma logiczna
`~a`	`not(a)`	Negacja logiczna
`a:d:b` `a:b`	`colon(a,d,b)` `colon(a,b)`	Operator dwukropek
`a'`	`ctranspose(a)`	Transpozycja zespolona
`a.'`	`transpose(a)`	Transpozycja
	`display(a)`	Wyświetlenie wartości wyrażenia
`[a b]`	`horzcat(a,b,...)`	Składanie poziome
`[a; b]`	`vertcat(a,b,...)`	Składanie pionowe
`a(s1,s2,...sn)`	`subsref(a,s)`	Odniesienie indeksowe
`a(s1,...,sn) = b`	`subsasgn(a,s,b)`	Podstawienie indeksowe
`b(a)`	`subsindex(a)`	Indeksowanie

Przykładowo, dla klasy **polynom** operacja * (mnożenie macierzowe - mtimes.m) dwóch obiektów realizowana jest jako splot wektorów ich współczynników:

```
function r = mtimes(p,q)
% POLYNOM/MTIMES Mnożenie wielomianów p * q
p = polynom(p);
q = polynom(q);
r = polynom(conv(p.c,q.c));
```

Przeciążanie funkcji odbywa się w taki sam sposób.

6.2.9 Dziedziczenie

Obiekty mogą dziedziczyć od innych obiektów właściwości i sposoby działania. Kiedy jeden obiekt (pochodny) dziedziczy od innego (bazowego), do obiektu pochodnego włączone są wszystkie pola struktury obiektu bazowego i może on używać wszystkich metod obiektu bazowego. Metody obiektu bazowego mają dostęp jedynie do pól, które obiekt pochodny dziedziczy od bazowego.
Dzięki dziedziczeniu obiekty klas pochodnych zachowują się dokładnie jak obiekt bazowy, z dodatkowymi rozszerzeniami, różnie implementowanymi w różnych klasach.
Wyróżniamy dwa rodzaje dziedziczenia:

- dziedziczenie proste, w którym obiekt pochodny dziedziczy po jednej klasie bazowej,
- dziedziczenie mnogie, kiedy występuje więcej niż jedna klasa bazowa dla obiektu.

W przypadku **dziedziczenia prostego** klasa pochodna dodaje do atrybutów klasy bazowej swoje własne atrybuty.
Konstruktor klasy w takim przypadku ma dwie specjalne cechy:

- Wywołuje on konstruktora klasy bazowej, dla utworzenia pól dziedziczonych
- Składnia jego wywołania różni się nieco, uwzględniając w argumentach obie klasy: pochodną i bazową.

Ogólny zapis utworzenia związku dziedziczenia prostego wygląda następująco:

```
obiekt_pochodny = class(obiekt_pochodny,'nazwa_klasy_pochodnej',obiekt_bazowy)
```

Proste dziedziczenie może obejmować więcej niż jeden poziom hierarchii. Jeżeli klasa bazowa sama jest klasą pochodną innej klasy, obiekt pochodny automatycznie dziedziczy również wszystkie pola tejże klasy.
Metody klasy bazowej mogą operować na obiektach klasy pochodnej (w zakresie dziedziczonych atrybutów), natomiast metody klasy pochodnej nie mogą działać na obiektach klasy bazowej. Sięganie do dziedziczonych atrybutów musi się odbywać za pośrednictwem metod klasy bazowej (np. **get** lub **subsref**). Kiedy konstruktor tworzy obiekt pochodny c wykorzystując obiekt bazowy b:

```
c = class(c,'nazwa_klasy_pochodnej',b);
```

MATLAB automatycznie tworzy pole c.*nazwa_klasy_bazowej* w strukturze obiektu, zawierające obiekt bazowy. W metodzie **display** klasy pochodnej wyświetlenie pól dziedziczonych odbywa się przez wywołanie metody **display** klasy bazowej:

```
display(c.nazwa_klasy_bazowej)
```

W przypadku **dziedziczenia mnogiego** klasa pochodna dziedziczy po większej liczbie klas. Również w tym przypadku dziedziczenie może obejmować więcej niż jeden poziom hierarchii. W konstruktorze dziedziczenie mnogie jest realizowane przez wywołanie funkcji **class** z większą ilością obiektów bazowych, reprezentujących poszczególne klasy.
Jeżeli w poszczególnych klasach bazowych występują metody o jednakowych nazwach, to przy wywołaniu uruchamiana będzie metoda z klasy bazowej pierwszej na liście argumentów wejściowych funkcji **class**. Nie ma możliwości dostępu do funkcji pozostałych klas bazowych.
Z dziedziczeniem związane są również pojęcia **agregacji** i **kompozycji**, kiedy obiekt zewnętrzny zawiera obiekt wewnętrzny jako jedno ze swoich pól. Metody dla obiektów wbudowanych mogą być wywołane jedynie w metodach obiektu zewnętrznego.

7 Nowe podejście do obiektów

7.1 Klasy zwykłe i referencyjne

W MATLAB-ie począwszy od wersji 7.6 (2008a) występują dwa rodzaje klas:

- klasy przechowujące dane (Value Classes),
- klasy przechowujące referencje do danych (Handle Classes).

Pierwszy rodzaj jest typowo stosowany np. w klasach numerycznych MATLAB-a. Rozpatrzmy następujący przykład:

```
>>a = int32(7);
>>b = a;
>>a = a^4;
>>b
    7
```

Po utworzeniu obiektu a (instancji klasy int32), podstawienie b = a; powoduje skopiowanie obiektu *a* na obiekt *b*. Następne operacje na obiekcie *a* (również jego usunięcie) nie będą wpływały na obiekt *b*.
Definicja klasy przechowującej dane składa się z bloku:

```
classdef nazwa_klasy
   ...
end
```

Przykład: pracownik **e1** (obiekt klasy *employee*), zostaje przeniesiony do innego działu, kopia danych **e2** zachowuje informację o poprzednim dziale:

```
classdef employee
   properties
      Name = ''
      Department = ''
   end
   methods
      function e = employee(name,dept)
         e.Name = name;
         e.Department = dept;
      end  %konstruktor
      function transfer(obj,newDepartment)
         obj.Department = newDepartment;
      end
   end
end

>>e1 = employee('Fred Smith','Informatics');
>>e2 = e1;
>>transfer(e1,'Technology');
>>e2.Department
ans =
Informatics
```

Inaczej zachowuje się program w przypadku wykorzystywania referencji - nowa zmienna (kopia referencji) będzie zawierać po prostu referencję do tego samego obiektu i wszelkie zmiany w obiekcie mogą być wprowadzane/obserwowane za pomocą każdej ze zmiennych, jak w poniższym przykładzie:

```
>>x = 1:10; y = sin(x);
>>h1 = line(x,y);
>>h2 = h1;
>>set(h2,'Color','red')
>>set(h1,'Color','blue')
>>delete(h2)
>>set(h1,'Color','green')
??? Error using ==> set
Invalid handle object.
```

Usunięcie obiektu funkcją *delete* za pomocą dowolnej z referencji spowoduje, że żadna z referencji już nie może na niego wskazywać (*delete* nie usuwa samej referencji!).
W przypadku klas referencyjnych, są one pochodnymi abstrakcyjnej klasy **handle**.
Definicja takiej klasy:

```
classdef nazwa_klasy < handle
   ...
end
```

Przykład: **e1** i **e2** są referencjami do obiektu klasy *employee*, po zmianie działu obie referencje wskażą pracownika w nowym dziale

```
classdef employee < handle
   properties
      Name = ''
      Department = ''
   end
   methods
      function e = employee(name,dept)
         e.Name = name;
         e.Department = dept;
      end  %konstruktor
      function transfer(obj,newDepartment)
```

```
            obj.Department = newDepartment;
        end
    end
end
>>e1 = employee('Fred Smith','Informatics');
>>e2 = e1;
>>transfer(e1,'Technology');
>>e2.Department
ans =
Technology
>
```

Przykładem zastosowania klas referencyjnych w MATLAB-ie są zmiany w strukturach obiektów graficznych. W nowszych wersjach MATLAB-a takie pola jak *Title, XLabel, YLabel* ... są obecnie referencjami do obiektów klasy *string* (agregacja).

7.2 Klasa abstrakcyjna *handle*

Klasa **handle** w MATLAB-ie służy do tworzenia klas referencyjnych. Dzięki temu klasy pochodne:

- dziedziczą szereg użytecznych metod,
- mają możliwość definiowania zdarzeń i metod nasłuchu,
- mogą posiadać pola dynamiczne.
- implementują metody *set* i *get* typu **HandleGraphics**.

Klasa **handle** nie posiada żadnych pól, natomiast udostępnia klasom pochodnym metody (które ewentualnie mogą być przeciążane):

- *eq*, *ne*, *lt*, *gt*, *le*, i *ge* - metody porównania referencji,
- *delete* - usuwa obiekt (ale pozostawia referencję),
- *isvalid* - testuje poprawność referencji,
- *findprop* - sprawdza, czy obiekt posiada pole o danej nazwie,
- *notify* - wysyła komunikaty o zdarzeniach do obiektów nasłuchujących,
- *addlistener* - tworzy obiekt nasłuchujący dla określonego zdarzenia,
- *findobj* - poszukuje obiektu o zadanej referencji.

W klasach pochodnych klasy **handle** można przeciążyć metody pobierania i modyfikacji pól, np. w celu kontrolowania zakresów wartości pola lub przechwytywania zdarzeń. W ten sposób użycie np. wyrażenia

```
nazwa_obiektu.nazwa_pola = wyrażenie;
```

spowoduje użycie metody set.*nazwa_pola* (jeśli została zdefiniowana w klasie) z argumentami *obiekt*, *wyrażenie*.

Pełna obsługa pobierania i modyfikacji pól obiektu może być zapewniona przez wywiedzenie klasy ze specjalnej klasy abstrakcyjnej **hgsetget**, będącej pochodną klasy **handle**. W takim przypadku dostęp do pól może się odbywać jak w standardowych obiektach graficznych za pomocą funkcji *set* oraz *get*. Np. podstawienie nowej wartości pola w obiekcie:

```
set(obiekt,nazwa_pola,wyrażenie);
```

a wyświetlenie listy dostępnych (publicznych) pól obiektu:

```
set(obiekt)
```

7.3 Meta-klasy

W MATLAB-ie istnieje możliwość utworzenia specjalnego obiektu - klasy *meta.class* - na podstawie definicji klasy lub jej instancji. Te meta-obiekty moą być przeglądane programowo. Do utworzenia takiego obiektu można użyć polecenia:

```
mobj = ?nazwa_klasy;
```

lub

```
mobj = metaclass(instancja_klasy);
```

Obiekt klasy *meta.class* posiada następujące pola:

- Name
- Description
- DetailedDescription

- Hidden
- Sealed
- ConstructOnLoad
- InferiorClasses
- Properties (tablica obiektów klasy *meta.property*)
- Methods (tablica obiektów klasy *meta.method*)
- Events (tablica obiektów klasy *meta.event*)
- SuperClasses
- ContainingPackage (tablica obiektów klasy *meta.package*)

Obiekt taki zawiera informację o właściwościach klasy - jej nazwie, atrybutach, polach, metodach i zdarzeniach. Informacje o polach zawarte są w tabeli obiektów klasy *meta.property* (pole *Properties*), informacje o metodach w tabeli obiektów klasy *meta.method* (pole *Methods*), natomiast informacje o zdarzeniach - w tabeli obiektów klasy *meta.event* (pole *Events*).
Klasy mogą być organizowane w pakiety (*packages*), których idea jest podobna do pakietów w języku Java. Informacja o pakietach związanych z daną klasą zawarta jest w tabeli obiektów klasy *meta.package* (pole *ContainingPackages*).

7.4 Definiowanie klasy użytkownika

W MATLAB-ie 8 wprowadzono możliwość umieszczania pełnego opisu klasy w pojedynczym pliku, o nazwie zgodnej z nazwą klasy. W takich okolicznościach nie występuje konieczność tworzenia odrębnego folderu dla klasy, w jednym folderze może występować kilka plików zawierających definicje różnych klas. Tylko w przypadku umieszczania definicji klasy w wielu m-plikach (odrębnych dla konstruktora i poszczególnych metod) należy tworzyć folder o nazwie *@nazwa_klasy*. W takim folderze może wystąpić tylko jeden plik zawierający konstruktor klasy.
Z uwagi na możliwości dziedziczenia (czyli budowanie klas na podstawie innych klas), klasy mogą być zorganizowane w hierarchie.
Nowością w MATLAB-ie 8 jest umieszczanie całej definicji klasy wewnątrz pojedynczego pliku. W takim pliku może wystąpić tylko jedna definicja klasy. Jest ona umieszczona na początku pliku, w bloku zawartym między słowami kluczowymi **classdef** i **end**. Na definicję klasy składają się bloki: pól (*properties*), metod (*methods*) oraz zdarzeń (*events*). Nazwy *properties*, *methods* oraz *events* są zastrzeżonymi słowami kluczowymi tylko w obrębie bloku definicji klasy.

7.4.1 Blok definicji klasy

Składnia bloku definicji klasy wygląda następująco:

```
classdef (atrybut_klasy = wyrażenie, ...) nazwa_klasy [< nazwy_klas_bazowych]
   bloki_definicji_pól
   bloki_definicji/deklaracji_metod
   bloki_deklaracji_zdarzeń
end
```

Atrybuty klasy służą do modyfikowania zachowania klasy. Mogą to być:

Nazwa atrybutu	Klasa	Domyślnie	Opis
Hidden	logical	false	Jeżeli ma wartość true, to nazwa nie będzie wyświetlana w listingach.
InferiorClasses	cell	{}	Tego atrybutu używa się do określenia zależności hierarchicznej między klasami
ConstructOnLoad	logical	false	Jeżeli ma wartość true, to konstruktor jest wywoływany automatycznie przy ładowaniu obiektu z pliku binarnego .mat.
Sealed	logical	false	Jezeli ma wartość true, to klasa nie może mieć klas pochodnych

7.4.2 Blok definicji pól

Składnia bloku definicji pól wygląda nastepująco:

```
properties (atrybut_pól = wyrażenie, ...)
   nazwa_pola [= wyrażenie]
   ...
end
```

Pola klasy mogą być tylko zdefiniowane, bądź zdefiniowane i zainicjowane - wtedy po nazwie pola umieszcza się wyrażenie inicjujące. Może być więcej niż jeden blok definicji pól - poszczególne bloki grupowane są wg atrybutów.
Atrybutom niewymienionym w nagłówku bloku zostaje nadana wartość domyślna (patrz tabela poniżej).
Uwaga: w odróżnieniu od "starego" modelu budowy obiektów, dostęp do odczytu i zapisu pól jest publiczny. W "starym" modelu dostęp ten był prywatny, tzn. dostęp do pól posiadały wyłącznie metody klasy.

Nazwa atrybutu	Klasa	Domyślnie	Opis
Abstract	logical	false	Jeżeli ma wartość true, to samo pole nie posiada implementacji, ale w klasach dziedziczących musi być zaimplementowane. Powinien być użyty z atrybutem klasy Sealed = false
Constant	logical	false	Jeżeli ma wartość true, to wszystkie instancje tej klasy będą miały tę samą wartość pola.
Dependent	logical	false	Jeżeli ma wartość true, to w polach obiektu przechowywane są tylko referencje do danych
GetAccess	char	public	Tryb dostępu do odczytu. Przyjmuje wartości public (bez ograniczeń), protected (tylko w metodach klasy lub jej klas pochodnych) lub private (tylko w metodach klasy).
GetObservable	logical	false	Jeżeli ma wartość true i klasa jest pochodną klasy *handle,* to mogą być konstruowane listenery. Obiekty nasłuchujące będą wywoływane przy próbie odczytu z pola.
Hidden	logical	false	Określa, czy pole powinno się pojawić na liscie.
SetAccess	char	public	Tryb dostępu do zapisu. Przyjmuje wartości public (bez ograniczeń), protected (tylko w metodach klasy lub jej klas pochodnych) lub private (tylko w metodach klasy).
SetObservable	logical	false	Jeżeli ma wartość true i klasa jest pochodną klasy *handle,* to mogą być konstruowane listenery. Obiekty nasłuchujące będą wywoływane przy próbie zapisu do pola.
Transient	logical	false	Jeżeli ma wartość true, to pola nie są zachowywane przy kopiowaniu obiektu do pliku.

7.4.3 Blok definicji metod

Definicja klasy zawiera również definicje funkcji składowych (metod). Rozróżniane są następujące typy metod:

- Metody zwykłe - funkcje działające na jednym lub wielu obiektach i zwracające nowy obiekt lub jakieś wyliczone wartości. Nie mogą modyfikować argumentów wejściowych. Wymagają istnienia obiektu klasy, na którym mogą działać. Zwykłe metody umożliwiają klasom implementację operatorów arytmetycznych i funkcji obliczeniowych.
- Konstruktory - specjalizowane metody służące do tworzenia obiektów klasy. Nazwa konstruktora musi być identyczna z nazwą klasy Typowo konstruktor inicjalizuje wartości pól na podstawie argumentów wejściowych. Konstruktor zwraca utworzony obiekt klasy.
- Destruktory - metody wywoływane automatycznie w momencie niszczenia obiektu (np. kiedy uruchamiamy funkcję `delete(obiekt)`, lub kiedy nie ma już żadnych referencji do obiektu).

- Metody obsługi dostępu do pól - umożliwiają zdefiniowanie kodu uruchamianego przy odczytywaniu lub ustawianiu wartości pola.
- Metody statyczne - funkcje związane z klasą, nie wymagające odniesienia do konkretnej instancji klasy.
- Konwertery - przeciążone metody konstruktorów innej klasy, umożliwiające przekształcenie obiektu na instancję innej klasy.
- Metody abstrakcyjne - służą do definiowania klas, które same nie służą do tworzenia obiektów, ale umożliwiają zdefiniowanie wspólnego interfejsu dla klas pochodnych. Klasy zawierające metody abstrakcyjne często nazywane są interfejsami.

Składnia bloku definicji metod wygląda następująco:

```
methods (atrybut_metod = wyrażenie, ...)
   function res = nazwa_funkcji(arg1,...)
      ciało_funkcji
   end                                %definicja metody
   res = nazwa_funkcji(arg1,...)      %deklaracja metody
   ...
end
```

Może być kilka bloków definicji metod. Metody są łączone w blokach o jednakowych atrybutach. Definicja metody jest identyczna z definicją funkcji MATLAB-a. Jeżeli metody zdefiniowane są w odrębnych m-plikach w folderze *@nazwa_klasy*, to istnieje możliwość umieszczenia deklaracji metody w bloku definicji klasy, celem nadania metodzie odpowiednich atrybutów. Deklaracja taka zawiera jedynie wiersz z argumentami o sygnaturze zgodnej z definicją funkcji w m-pliku. Np. w pliku *testdata.m* umieszczono definicję funkcji **testdata** , wywoływanej z trzema argumentami:

```
function tdata = testdata(myClass_object,argument2,argument3)
   ...
end
```

Jej deklarację, z odpowiadającą liczbą argumentów, należy umieścić w pliku definicji klasy *myClass.m*:

```
classdef myClass
...
   methods (AttributeName = value,...)
      td = testdata(obj,arg1,arg2)
...
   end % methods
...
end % classdef
```

Konstruktor musi być zdefiniowany w bloku *classdef*, zatem nie może występować w oddzielnym pliku. Dotyczy to również metod dostępu do pól: *set* oraz *get*.

Nazwa atrybutu	Klasa	Domyślnie	Opis
Abstract	logical	false	Jeżeli ma wartość true, to metody nie posiadają implementacji.
Access	char	public	Określa jaki kod może wywoływać tę metodę. Przyjmuje wartości public (bez ograniczeń), protected (dostęp tylko w metodach klasy lub jej klas pochodnych) lub private (tylko w metodach klasy).
Hidden	logical	false	Określa, czy nazwa metody ma się pojawiać w wykazach.
Sealed	logical	false	Jeśli posiada wartość true, to metoda nie może być przedefiniowywana w klasach pochodnych.
Static	logical	false	Jeżeli jest ustawiony na true, to metoda nie jest związana z żadnym obiektem klasy. Wywołanie odbywa się przy pomocy nazwy klasy: *nazwa_klasy.nazwa_metody*

Dla zapewnienia typowego dla MATLAB-a zachowania obiektów klasy, należy uwzględnić implementacje metod, przeciążających standardowe metody obiektów. Również standardowe operatory arytmetyczne lub logiczne mogą być przeciążane, dając odpowiednie zachowanie obiektów w środowisku MATLAB-a. Zagadnienie to było już omówione w poprzednim rozdziale.

7.4.4 Blok deklaracji zdarzeń

Zdarzenia mogą być definiowane w klasach pochodnych klasy **handle**. Zdarzeniami są zmiany lub akcje zachodzące wewnątrz instancji klasy, np.:

- Modyfikacja danych obiektu
- Wywołanie metody
- Odczyt lub ustawienie wartości pola
- Zniszczenie obiektu

Składnia bloku deklaracji zdarzeń wygląda następująco:

```
events (atrybut_zdarzenia = wyrażenie, ...)
   nazwa_zdarzenia
   ...
end
```

Zdarzenia umieszczane są w blokach zgrupowanych wg wartości atrybutów:

Nazwa atrybutu	Klasa	Domyślnie	Opis
Hidden	logical	false	Określa czy nazwa zdarzenia ma pojawiać się na wykazach.
ListenAccess	char	public	Określa sposób dostępu do tworzenia listenerów. Przyjmuje wartości public (bez ograniczeń), protected (tylko w metodach klasy lub jej klas pochodnych) lub private (tylko w metodach klasy).
NotifyAccess	char	public	Określa gdzie kod programu może wywołać zdarzenie. Przyjmuje wartości public (dowolny kod), protected (tylko w metodach klasy lub jej klas pochodnych) lub private (tylko w metodach klasy).

Zdarzenie definiowane jest przez zadeklarowanie jego nazwy w bloku *events*, wewnątrz definicji klasy generującej to zdarzenie (klasa powinna posiadać klasę bazową **handle**, aby dziedziczyć metody *notify* i *addlistener*):

```
classdef ToggleButton < handle
   properties
      State = false
   end
...
   events
      ToggleState
   end
end
```

Wyzwolenie zdarzenia musi być kontrolowane przez odpowiednią metodę klasy:

```
...
   methods
      function OnStateChange(obj,newState)
         if newState ~= obj.State
            notify(obj,'ToggledState');
            obj.State = newState;
         end
      end
   end
...
```

Metoda OnStateChange wywołuje funkcję *notify* w celu uruchomienia zdarzenie. Argumentami funkcji *notify* są: uchwyt do obiektu (ToggleButton), będącego właścicielem zdarzenia oraz łańcuch zawierającego nazwę zdarzenia. Przy wyzwoleniu zdarzenia MATLAB może przekazywać metodą *notify* dane (standardowo za pomocą obiektu klasy *EventData* z pakietu *event*) do obiektów nasłuchujących (*listeners*). Klasa *EventData* posiada dwa pola:

- EventName - nazwa zdarzenia opisywanego przez obiekt,
- Source - obiekt źródłowy, którego klasa opisuje zdarzenie.

Poprzez rozszerzenie klasy *EventData* można zdefiniować własną klasę obiektów przekazywanych:

```
classdef ToggleEventData < event.EventData
   properties
      NewState %pole dodane do definicji klasy EventData
   end
   methods
      function data = ToggleEventData(newState) %konstruktor
         data.NewState = newState;
      end
   end
end
```

i przekazać jej instancję za pomocą metody *notify*:

```
notify(obj,'ToggleState',ToggleEventData(newState));
```

Obiekt nasłuchujący jest instancją klasy *listener* z pakietu *event*. Klasa ta posiada pola:

- Source - tablica obiektów źródłowych,
- EventName - nazwa zdarzenia,
- Callback - funkcja uruchamiana przy wyzwoleniu zdarzenia (gdy Enabled = true), do funkcji tej przekazywane są dwa argumenty: źródło zdarzenia (obiekt wywołujący) oraz obiekt klasy event.EventData lub jej podklas,
- Enabled - zezwolenie na uruchomienie funkcji Callback (domyślnie: true), za jego pomocą można czasowo zawieszać działanie listenera,
- Recursive - jeżeli przybiera wartość true (domyślnie), listener może wywołać takie samo zdarzenie jak to, które wywołało funkcję Callback, co może prowadzić do zapętlenia.

Obiekty nasłuchujące mogą być tworzone na dwa sposoby:

- Przy użyciu metody *addlistener*, przywiązującej taki obiekt na czas życia do obiektu wywołującego zdarzenie. Metoda ta jest dziedziczona po klasie *handle*. Składnia wyrażenia generującego *listener* jest następująca:

  ```
  lh = addlistener(obiekt,nazwa_zdarzenia,funkcja_callback)
  ```

  Na przykład:

  ```
  lh = addlistener(obj,'ToggleState',@CallbackFunction)
  ```

  Argumentami są kolejno: obiekt wywołujący zdarzenie, przekazywana nazwa zdarzenia i uchwyt do metody *callback* obiektu nasłuchującego. Z odczytem lub zmianą wartości pola mogą być związane predefiniowane zdarzenia. W tym przypadku składnia wyrażenia generującego obiekt nasłuchujący jest następująca:

  ```
  lh = addlistener(obiekt,nazwa_pola,nazwa_zdarzenia,funkcja_callback)
  ```

  Dopuszczalne (predefiniowane) zdarzenia to:

  - PreSet
  - PostSet
  - PreGet
  - PostGet

  Kontrolę nad używaniem predefiniowanych zdarzeń w odniesieniu do pól mają atrybuty *SetObservable* i *GetObservable*. Jeżeli mają one wartość **true**, to zdarzenia generowane są automatycznie, nie jest potrzebne użycie funkcji *notify*.
- Przy pomocy konstruktora klasy *event.listener* - w tym przypadku obiekt nasłuchujący nie jest niszczony wraz z obiektem wywołującym zdarzenie.
  Przykładowo:

  ```
  lh = event.listener(obj,'ToggleState',@CallbackFunction)
  ```

  Argumenty są takie same jak w metodzie *addlistener*. Taki obiekt istnieje w programie tylko w granicach bloku, w którym został utworzony.

W wyniku uruchomienia metod *addlistener* lub *event.listener* generowany jest nowy obiekt klasy *event.listener*. Funkcja *callback* tego obiektu musi być zdefiniowana tak, aby przyjmować dwa argumenty:

```
function CallbackFunction(src,evnt)
   ...
end
```

7.4.5 Klasy wyliczeniowe

W MATLAB-ie 2014 wprowadzono nowy rodzaj klas – klasy wyliczeniowe (*enumeration*). Klasa wyliczeniowa powstaje przez dodanie bloku *enumeration* do definicji klasy. Przykładowo definicja klasy *WeekDays* zawierającej nazwy powszednich dni tygodnia:

```
classdef WeekDays
   enumeration
      Monday, Tuesday, Wednesday, Tuesday, Friday, Saturday
   end
end
```

Do konkretnej wartości odwołujemy się za podając nazwę klasy i nazwę elementu:

```
today = WeekDays.Monday;
```

Klasa WeekDays domyślnie będzie wyposażona w cztery metody:

- Weekdays - konstruktor
- char – konwerter do ciągu znaków
- eq – operator równości (== w wyrażeniach)
- ne – operator nierówności (~= w wyrażeniach logicznych)

Funkcja **enumeration** pozwala na wylistowanie wszystkich elementów wyliczenia:

```
>> enumeration WeekDays
Enumeration members for class 'WeekDays':
    Monday
    Tuesday
    Wedenesday
    Thursday
    Friday
```

natomiast funkcja **isenum** testuje, czy dany obiekt należy do klasy enumeration:

```
>> isenum(today)
ans =
     1
```

Przez powiązanie klasy wyliczeniowej z numeryczną klasą bazową można uzyskać numeryczne reprezentacje elementów:

```
classdef WeekDays < int8
   enumeration
    Monday     (1)
    Tuesday    (2)
    Wedenesday (3)
    Thursday   (4)
    Friday     (5)
   end
end
```

Klasa bazowa będzie akceptować obiekty klasy wyliczeniowej, umożliwiając konwersję:

```
>> today = WeekDays.Tuesday;
>> tomorrow = WeekDays(today+1)
tomorrow =
          Wednesday
>> whos
  Name          Size             Bytes  Class       Attributes
  today         1x1                105  WeekDays
  tomorrow      1x1                105  WeekDays
```

W klasie wyliczeniowej można również definiować metody:

```
classdef WeekDays < int8
    methods
        function tf = isMeetingDay(obj)
            tf = WeekDays.Monday.eq(obj);
        end
    end
    enumeration
        Monday (1), Tuesday (2), Wedenesday(3)
        Thursday (4), Friday (5)
    end
end
```

Do metody można przesyłać element wyliczenia bezpośrednio:

```
>> isMeetingDay(WeekDays.Tuesday)
ans =
     0
```

Klasa wyliczeniowa może również mieć zdefiniowane atrybuty:

```
classdef SyntaxColors
   properties
      R
      G
      B
   end
   methods
      function c = SyntaxColors(r,g,b)
         c.R = r;
         c.G = g;
         c.B = b;
      end
   end
   enumeration
      Error   (1,0,0)
      Comment (0,1,0)
      Keyword (0,0,1)
      String  (1,0,1)
   end
end
```

Zainicjowanie obiektu odbywa się z użyciem konstruktora

```
>> c = SyntaxColors.Comment;
>> c.G
ans =
     1;
```

Atrybuty są dostępne jedynie dla konstruktora, nie można ich modyfikować. Wprowadzenie klasy bazowej **handle** daje dodatkowe możliwości: dziedziczone metody (addlistener, eq, findprop, gt, le, ne, delete, findobj, ge, isvalid, lt, notify) oraz dostęp do atrybutów:

```
classdef SyntaxColors < handle
   properties
      R
      G
      B
   end
   methods
      function c = SyntaxColors(r,g,b)
         c.R = r;
         c.G = g;
         c.B = b;
      end
   end
   enumeration
      Error   (1,0,0)
      Comment (0,1,0)
      Keyword (0,0,1)
      String  (1,0,1)
   end
end
```

Zainicjowanie obiektu odbywa się z użyciem konstruktora, ale potem atrybuty dają się modyfikować

```
>> c = SyntaxColors.Comment;
>> b = SyntaxColors.Comment;
>> c.R = 0.5
c =
   Comment
>> b.R
ans =
     0.5000
```

7.5 Przykład klasy użytkownika

Jako przykład klasy użytkownika przedstawiono projekt klasy *DocPolynom*, która jest podobna do przedstawionej w rozdziale 6.2 klasy *polynom*.
Klasa *DocPolynom* posiada jedno, dostępne publicznie pole typu *double*, o nazwie **coef**, w którym przechowywane są współczynniki przy kolejnych potęgach zmiennej x (w porządku malejącym), oraz następujące metody:

- **DocPolynom** - konstruktor klasy,
```
>> p = DocPolynom([1 2 1]);
```
- **double** - konwerter obiektu klasy *DocPolynom* na typ *double* (np. zwracający wektor współczynników wielomianu),
```
>> double(p)
ans =
     1    2    1
```
- **char** - konwerter tworzący tekstowy zapis wielomianu jako sumy potęg x, używany przez metodę *disp*,
- **disp** - określa w jaki sposób MATLAB wyświetla obiekty klasy w wierszu poleceń,
```
>> p
p =
     x^2 + 2*x + 1
```
- **subsref** - umożliwia indeksowy dostęp do obiektu klasy *DocPolynom*, dostęp do jego pola *coef*, bądź dostęp do metod klasy:
```
>> p(-1)  %zwraca wartość wielomianu w punkcie x = -1
ans =
     0
>> p.coef %zwraca wartość pola coef
ans =
     1    2    1
>> p.polyval(-1) %zwraca wartość wielomianu w punkcie x = -1
                 %inaczej: polyval(p,-1)
ans =
     0
```
- **plus** - wykonuje dodawanie obiektów klasy (przeciążenie operatora +),
- **minus** - realizuje odejmowanie obiektów klasy (przeciążenie operatora -),
- **mtimes** - realizuje mnożenie obiektów klasy (przeciążenie operatora *). Uwaga: w przypadku gdy jeden z argumentów jest wektorem, metody przeciążające operatory arytmetyczne (+, -, *) dokonują wstępnie konwersji tego argumentu na typ *DocPolynom* (używając konstruktora).
```
>> p + [1 2 1]
ans =
     2*x^2 + 4*x + 2
>> [1 2] * p
ans =
     x^3 + 4*x^2 + 5*x + 2
```
- **roots** - specjalizowana metoda przeciążająca wbudowaną funkcję *roots* MATLAB-a,
```
>> roots(p)
ans =
     -1
     -1
```
- **polyval** - metoda przeciążająca dla obiektów klasy funkcję *polyval* MATLABA,
- **diff** - metoda przeciążająca dla obiektów klasy wbudowaną funkcję *diff* MATLAB-a,
- **plot** - metoda przeciążająca wbudowaną funkcję *plot*, tworząca wykresy wartości wielomianu w funkcji zmiennej x.

W klasie DocPolynom wprowadzono pewne zmiany w działaniu niektórych metod w stosunku do klasy *polynom* :

- dla konstruktora : brak możliwości wywołania konstruktora bez parametrów. Można to w prosty sposób zmodyfikować, dopisując na początku kodu konstruktora:

  ```
  if nargin == 0, c = []; end
  ```

- dla metody **subsref**: zmieniony został sposób obliczania wartości wielomianu przez wykorzystanie metody *polyval* - wywołanie:

  ```
  <nazwa_obiektu>(<wektor_x>)
  ```

  a także dodano możliwość pobrania wartości pola **coef** :

  ```
  <nazwa_obiektu>.coef
  ```

  oraz dostępu do metod klasy za pomocą notacji:

  ```
  <nazwa_obiektu>.<nazwa_metody>(<argumenty>)
  ```

W pliku DocPolynom.m, umieszczonym w folderze @DocPolynom, zawarta jest pełna definicja klasy. Składa się ona z bloku *properties* zawierającego definicję pola **coef** oraz bloku *methods*, w którym obok definicji konstruktora - funkcji **DocPolynom**, umieszczono definicje metod składowych klasy.

```
classdef DocPolynom
% file: @DocPolynom/DocPolynom.m
% Public properties
   properties
      coef
   end

   % Class methods
   methods
      function obj = DocPolynom(c)
         % Construct a DocPolynom object using the coefficients supplied
         if isa(c,'DocPolynom')
            obj.coef = c.coef;
         else
            obj.coef = c(:).';
         end
      end % DocPolynom
      function obj = set.coef(obj,val)
         ind = find(val(:).'~=0);
         if ~isempty(ind);
            obj.coef = val(ind(1):end);
         else
            obj.coef = val;
         end
      end

      function c = double(obj)
         c = obj.coef;
      end % double

      function s = char(obj)
         % Created a formated display of the polynom
         % as powers of x
         if all(obj.coef == 0)
            s = '0';
         else
            d = length(obj.coef)-1;
            s = cell(1,d);
            ind = 1;
            for a = obj.coef;
               if a ~= 0;
                  if ind ~= 1
                     if a > 0
                        s(ind) = {' + '};
                        ind = ind + 1;
                     else
```

```
                s(ind) = {' - '};
                a = -a;
                ind = ind + 1;
             end
          end
          if a ~= 1 || d == 0
             if a == -1
                s(ind) = {'-'};
                ind = ind + 1;
             else
                s(ind) = {num2str(a)};
                ind = ind + 1;
                if d > 0
                   s(ind) = {'*'};
                   ind = ind + 1;
                end
             end
          end
          if d >= 2
             s(ind) = {['x^' int2str(d)]};
             ind = ind + 1;
          elseif d == 1
             s(ind) = {'x'};
             ind = ind + 1;
          end
       end
       d = d - 1;
    end
 end
end % char

function disp(obj)
   % DISP Display object in MATLAB syntax
   c = char(obj);
   if iscell(c)
      disp(['     ' c{:}])
   else
      disp(c)
   end
end % disp

function b = subsref(a,s)
   % SUBSREF Implementing the following syntax:
   % obj([1 ...])
   % obj.coef
   % obj.plot
   % out = obj.method(args)
   % out = obj.method
   switch s(1).type
      case '()'
         ind = s.subs{:};
         b = a.polyval(ind);
      case '.'
         switch s(1).subs
            case 'coef'
               b = a.coef;
            case 'plot'
               a.plot;
            otherwise
               if length(s)>1
                  b = a.(s(1).subs)(s(2).subs{:});
               else
                  b = a.(s.subs);
               end
```

```
            end
         otherwise
            error('Specify value for x as obj(x)')
      end
   end % subsref

   function r = plus(obj1,obj2)
      % PLUS  Implement obj1 + obj2 for DocPolynom
      obj1 = DocPolynom(obj1);
      obj2 = DocPolynom(obj2);
      k = length(obj2.coef) - length(obj1.coef);
      r = DocPolynom([zeros(1,k) obj1.coef] + [zeros(1,-k) obj2.coef]);
   end % plus

   function r = minus(obj1,obj2)
      % MINUS Implement obj1 - obj2 for DocPolynoms.
      obj1 = DocPolynom(obj1);
      obj2 = DocPolynom(obj2);
      k = length(obj2.coef) - length(obj1.coef);
      r = DocPolynom([zeros(1,k) obj1.coef] - [zeros(1,-k) obj2.coef]);
   end % minus

   function r = mtimes(obj1,obj2)
      % MTIMES   Implement obj1 * obj2 for DocPolynoms.
      obj1 = DocPolynom(obj1);
      obj2 = DocPolynom(obj2);
      r = DocPolynom(conv(obj1.coef,obj2.coef));
   end % mtimes

   function r = roots(obj)
      % ROOTS.  ROOTS(obj) is a vector containing the roots of obj.
      r = roots(obj.coef);
   end % roots

   function y = polyval(obj,x)
      % POLYVAL  POLYVAL(obj,x) evaluates obj at the points x.
      y = polyval(obj.coef,x);
   end % polyval

   function q = diff(obj)
      % DIFF  DIFF(obj) is the derivative of the polynom obj.
      c = obj.coef;
      d = length(c) - 1;  % degree
      q = DocPolynom(obj.coef(1:d).*(d:-1:1));
   end % diff

   function plot(obj)
      % PLOT  PLOT(obj) plots the polynom obj
      r = max(abs(roots(obj)));
      x = (-1.1:0.01:1.1)*r;
      y = polyval(obj,x);
      plot(x,y);
      c = char(obj);
      title(['y = ' c{:}])
      xlabel('X')
      ylabel('Y','Rotation',0)
      grid on
   end % plot
end % methods
end % classdef
```

8 Techniki stosowane dla poprawy szybkości obliczeń

W programie MATLAB istnieje kilka funkcji umożliwiających pomiar czasu wykonywania programu:

- **clock** – funkcja pobierająca czas systemowy w postaci wektora T = [Rok Miesiąc Dzień Godzina Minuta Sekundy].
- **etime** – funkcja obliczająca różnicę czasu między dwiema wartościami otrzymanymi z zegara.
- **tic** – uruchomienie stopera.
- **toc** – zatrzymanie stopera i obliczenie odstępu czasowego od ostatniego wystąpienia funkcji **tic**.

Program MATLAB został opracowany i zoptymalizowany do pracy z macierzami. Operacje macierzowe są w nim wykonywane znacznie szybciej niż zewnętrzne pętle obliczeniowe. Starannie przemyślana wektoryzacja obliczeń może je znacznie przyspieszyć. Weźmy następujący przykład:

```
clear
tic
n = 0;
for t = 0:0.001:100
      n = n + 1;
      y(n) = sin (t);
end
toc
```

Dla tego programu uzyskano czas obliczeń ok. 2 min. Po zmianie kodu na następujący:

```
clear
tic
t = 0:0.001:100;
y = sin(t);
toc
```

czas obliczeń wyniósł jedynie 0.03 sek. W przykładzie tym wystąpił jeszcze jeden, ukryty czynnik przyspieszenia obliczeń – wstępna alokacja tablicy. Następny przykład pokaże, jak wielki wpływ na czas obliczeń ma dynamiczna zmiana wymiarów tablicy. Weźmy pod uwagę program:

```
clear
tic
y = zeros(1,100001);
n = 0;
for t=0:0.001:100
      n = n + 1;
      y(n) = sin(t);
end
toc
```

W tym przypadku czas obliczeń wyniósł jedynie 0.047 sek. Zaalokowanie w sposób jawny pamięci dla tablicy y dało znaczne przyspieszenie obliczeń w stosunku do pierwszego przykładu. I na koniec sprawdzimy, jaki efekt da jednoczesne jawne zarezerwowanie miejsca dla zmiennej y i zastąpienie pętli **for** podstawieniem macierzowym:

```
clear
tic
y = zeros(1,100001);
t = 0:0.001:100;
y = sin(t);
toc
```

W tym przypadku czas obliczeń wyniósł znowu 0.03 sek. Wyeliminowanie pętli **for** dało więc w tym przypadku ok. 50 % skrócenie obliczeń w stosunku do poprzedniego przykładu, jednak najistotniejsza była wstępna realokacja tablicy.