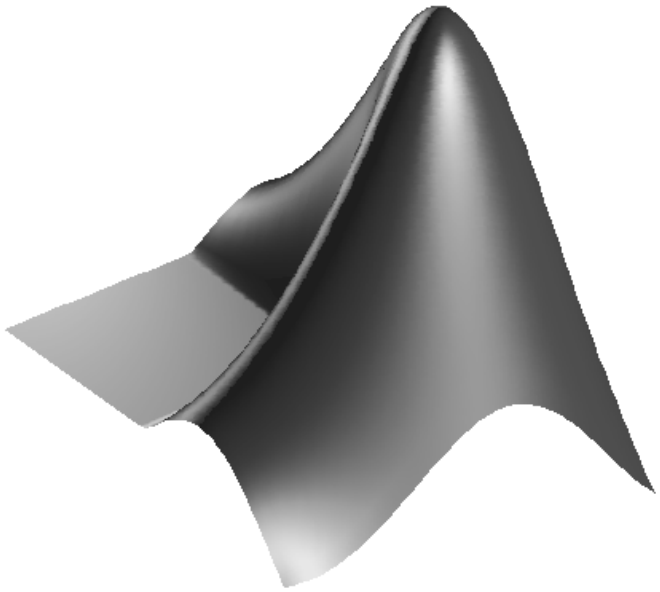

KURS MATLAB II

Rok 2005/2006, semestr zimowy

Uniwersytet Warszawski
Wydział Fizyki

Ryszard Buczyński, Rafał Kasztelanic

Spis Treści

KURS MATLAB II

Rok 2005/2006 semestr zimowy, wymiar 15h

Prowadzący:

dr Ryszard Buczyński, ryszard.buczynski@mimuw.edu.pl,
dr Rafał Kasztelanic, kasztel@mimuw.edu.pl

Zakład Optyki Informacyjnej, Instytut Geofizyki, Wydz. Fizyki UW
Zajęcia odbywają się w Instytucie Geofizyki, ul. Pasteura 7, V piętro, pok. 508. (lub pok. 106)

Oprogramowanie:

Matlab, *The MathWorks, Inc.;* wersja 7.0; platforma UNIX/LINUX.

Krótki opis kursu:

Głównym celem kursu jest nauka posługiwania się Matlabem jako narzędziem programistycznym na poziomie średnio zaawansowanym. Kurs MATLAB II jest kontynuacją kursu MATLAB I skoncentrowany na zagadnieniach związanych z programowaniem. W programie kursu przewidziane są również tematy wymagające korzystania z dodatkowych modułów pakietu Matlab oraz dodatkowych narzędzi programistycznych.
Kurs składa się z 15 godzin zajęć komputerowych. Każdy student pracuje indywidualnie przy osobnym terminalu. Wszystkie spotkania rozpoczynają się od wprowadzenia merytorycznego do poruszanych zagadnień. Dalsza część zajęć poświęcona jest na indywidualną pracę z komputerem nad serią zadań.
Kurs odbywać się będzie zarówno w języku polskim jak i angielskim.

Forma zaliczenia:

Do zaliczenia Kursu na ocenę dostateczną lub zal. wymagane jest zaliczenie wszystkich ćwiczeń-laboratoriów. Ocena końcowa wystawiana jest przez prowadzącego na podstawie osiągniętej sprawności i postępów w posługiwaniu się MATLABem, oraz kreatywności studenta.
Obecność na wszystkich zajęciach jest obowiązkowa, dopuszczamy jedna nieobecność, przy większej ilości wymagane jest zwolnienie lekarskie. Nieobecność nie zwalnia studenta z zaliczenia poszczególnych zadań.
Ostatnia godzina 15-ta jest przeznaczona na wystawianie ocen i ewentualne poprawki.

Wymagania:

MATLAB I, znajomość algebry, analizy matematycznej i programowania.

Zagadnienia poruszane w ramach kursu MATLAB II:

Kurs MATLAB II bazuje na wiedzy zdobytej w trakcie kursu MATLAB I. DO głównych zagadnień poruszanych w trakcie trwania kursu należą:

1. Fourierowska analiza danych
2. Przetwarzanie obrazów – *toolbox Image Processing*
3. Całkowanie numeryczne
4. Numeryczne różniczkowanie funkcji
5. Równania różniczkowe i całkowe
6. Obsługa błędów
7. Grafika uchwytów
8. Graficzny interfejs użytkownika
9. Złożone struktury danych
10. Programowanie zorientowane obiektowo
11. Obliczenia symboliczne w Matlabie
12. Współpraca Matlaba z innymi środowiskami programistycznymi

Literatura:

1. A. Zalewski, R. Cegieła, Matlab - obliczenia numeryczne i ich zastosowania.
2. B. Mrozek, Z. Mrozek, Matlab 6 - poradnik użytkownika.
3. D. Higham, N. Higham: Matlab guide.
4. The MathWorks, Inc: Numerical Computing with MATLAB.
5. http://www.mathworks.com/

TEMATY

FOURIEROWSKA ANALIZA DANYCH

Analiza widma funkcji otrzymywanej za pomocą transformaty Fouriera stanowi podstawię większości algorytmów przetwarzania sygnałów.
Matlab dostarcza wygodnych narzędzi do takiej analizy.

Temat 1
Jedno wymiarowa transformata Fouriera

Definicja ***fft(x)*** gdzie x jest wektorem próbek sygnału. ***ifft(x)*** – oznacza odwrotną transformatę Fouriera.
Wszystkie pochodne funkcji transformaty Fouriera znajdują się w grupie *datafun*.

UWAGA: Funkcja *fft* i jej pochodne najszybciej wykonywana jest dla danych o rozmiarze będącym potęgą dwójki, np. 16, 32, 64, itd. Trochę wolniej trwają obliczenia dla danych o rozmiarze będącym liczbą pierwszą. Obliczenia dla danych o innych rozmiarach są zdecydowanie wolniejsze.

Temat 2
Inne pomocne funkcje

fftshift	przesunięcie odpowiednich elementów składowych sygnału
real	część rzeczywista liczby zespolonej
imag	część urojona liczby zespolonej
conj	sprzężenie zespolone
angle	część fazowa sygnału
abs	moduł sygnału
conv	splot dwóch wektorów
deconv	dekonvolucja dwóch wektorów
cov	kowariancja
decimate	przetworzenie wektora – rzadsze próbkowanie
interp	przetworzenie wektora – gęstsze próbkowanie
unwrap	Uciąglenie fazy

UWAGA: Aby obliczyć wartość natężenia sygnału należy skorzystać z następującego przekształcenia:

```
>> nat = abs(sygnal).^2;
```

Temat 3
Dwu wymiarowa transformata Fouriera Transformata

Z przypadkiem dwu wymiarowej transformaty Fouriera najczęściej mamy do czynienia przy przetwarzaniu zdjęć.
Dwu wymiarowa transformata Fouriera – *fft2*.
Funkcje odwrotne: ifft2, ifftshift.

Przykład:

```
>> s = rand(128); % stworzenie 2 wymiarowego zbioru danych
>> fs = fftshift(fft2(s)); % 2 wymiarowa transformata Fouriera wraz z przesunięciem danych
```

Temat 4
Wielowymiarowa transformata Fouriera Transformata

Matlab oferuje możliwość obliczenia wielowymiarowej transformaty Fouriera przy wykorzystaniu funkcji *fftn*.

Przykład:

```
>> s4 = rand(32,32,32,32); % stworzenie 4 wymiarowego zbioru danych
>> fs4 = fftshift(fftn(s4));
```

Temat 5
Filtry analogowe i cyfrowe

Signal Processing Toolbox udostępnia wiele funkcji związanych z przetwarzaniem sygnałów za pomocą filtrów. Filtr analogowy, odpowiedź częstotliwościowa, realizowany jest przy wykorzystaniu funkcji *freqs*(*a, b, w*). Gdzie *a* i *b* są współczynnikami wielomianów a *w* jest wektorem z częstościami, dla których prowadzimy analizę.
Filtrację cyfrową, FIR (skończona odpowiedź impulsowe) lub IIR (nieskończona odpowiedź impulsowa), wykonuje się za pomocą funkcji *filter*(*b, a, x*). Gdzie *b* i *a* są współczynnikami odpowiednich wielomianów a *w* jest wektorem dyskretnych wartości sygnału na wejściu filtra.

Przykład:

```
>> data = [1:0.2:4]';
>> filter(ones(1,5)/5,1,data); % filtr cyfrowy tyu FIR (uśredniający)

>> a = [1 0.4 1]; b = [0.2 0.3 1]; w = logspace(-1,1);
>> freqs(b,a,w) % filtr analogowy
```

PRZETWARZANIE DŹWIĘKU

Przykładem sygnału 1 wymiarowego jest sygnał dźwiękowy.

Temat 6
Wczytanie pliku dźwiękowego – funkcja *wavread*

Do wczytania pliku dźwiękowego w formacie .wav wykorzystuje się funkcję *wavread*().

Przykład:

```
>> y = wavread('plik.wav'); % wczytanie pliku dźwiękowego
>> [y,Fs,bits] = wavread('plik.wav'); % zwraca plik dźwiękowy y, częstość próbkowania Fs
                                        i liczbę bitów próbkowania
```

UWAGA: Do wczytania ciągu danych niebędących plikiem dźwiękowym służy funkcja *load*.

Temat 7
Odtworzenie pliku dźwiękowego – funkcja *wavplay*

Funkcją *wavplay* można odtworzyć jako dźwięk dowolną zmienną będącą wektorem liczb rzeczywistych.

Przykład:

```
>> wavplay(y); % odtworzenie pliku dźwiękowego y

>> d = rand(1,20000);
>> wavplay(d); % odtworzenie wektora losowego

>> t = (0:0.001:1)';
>> y = sin(2*pi*50*t) + 2*sin(2*pi*120*t); sygnał o 2 składowych o częstościach 50 i 120 Hz
>> wavplay(y) % odtworzenie dźwięku y
```

Temat 8
Zarejestrowanie dźwięku – funkcja *wavrecord*

Do zarejestrowania dźwięku wykorzystuje się funkcję *wavrecord*. Funkcja ta może być wywoływane z kilkoma parametrami:
waverecord(liczba próbek, częstość próbkowania, próbkowanie);

Przykład:

```
>> Fs = 11025;
>> y = wavrecord(5*Fs,Fs,'int16'); % Nagranie 5 sekund z częstością 11025 Hz i
                                     próbkowaniem 16 bitowym
```

Temat 9
Zapisanie pliku dźwiękowego – funkcja *wavwrite*

Do zapisania zarejestrowanego lub zmienionego pliku dźwiękowego wykorzystuje się funkcję wavwrite(zmienna z danymi, częstość próbkowania, 'nazwa pliku')

Przykład:

```
>> wavwrite(y,11025,'muzyka.wav'); % zapisanie zmiennej y z próbkowaniem 11025 Hz w
                                    pliku muzyka.wav
```

Temat 10
Generowanie przebiegów czasowych

sawtooth	przebieg trójkątny (periodyczny)
square	przebieg prostokątny (periodyczny)
gauspuls	przebieg sinusoidalny modyfikowany Gaussem (nieperiodyczny)
chirp	Przebieg cosinusoidalny o zmiennym okresie

Temat 11
Fourierowska analiza próbek dźwiękowych – funkcja *specgram*
Funkcja *specgram* udostępniona jest w ramach *Signal Processing Toolbox*.
Zadaniem tej funkcji jest przeprowadzenie analizy częstotliwościowej sygnału zmiennego w czasie. Wynikiem działania funkcji jest obraz, na którego osi poziomej mamy informacje o czasie a na osi pionowej informacje o częstotliwościach.

Przykład:

```
>> specgram(y,512,2); % wyświetla
```

PRZETWARZANIE OBRAZÓW

Przy przetwarzaniu obrazów wygodnie jest korzystać z *Image Processing Toolbox*.

Temat 12
Wczytanie obrazka – funkcja *imread*

Przy wykorzystaniu funkcji *imread* można wczytać dowolny, rozpoznawalny przez Matlaba (cur, gif, ico, tif, png, hdf), plik graficzny.

Przykład:

```
>> obr = imread('pout.tif');
```

UWAGA: Wczytany obrazek jest pamiętany jako macierz typu uint8 lub uint16. Czasami w celu wykonania różnych operacji zachodzi potrzeba zmiany typu macierzy na double:

```
>> obr = double(imread('pout.tif'));
```

Temat 13
Wyświetlenie obrazka na ekran

```
>> imshow(obr) % wyświetlenie obrazka
>> image(obr)  % wyświetla macierz danych jako obraz
```

Temat 14
Zapisywanie obrazka – funkcja *imwrite*

Podstawowe wywołanie funkcji imwrite składa się z 2 parametrów określających macierz z danymi oraz nazwę pliku. Rozszerzenie nazwy pliku determinuje w jakim formacie zapisany jest obraz. W wywołaniu funkcji może się znaleźć też więcej parametrów odpowiedzialnych za takie parametry jak palety barw, ilość kolorów, przezroczystość, kompresje itp.

Przykład:

```
>> imwrite(obr,'obraz.png'); % nagranie macierzy z danymi obr w pliku obrazowym typu png
```

Temat 15
Podstawowe operacje na obrazach

imabsdiff	różnica między dwoma obrazami
imadd	suma 2 obrazów, plus stała
imdivide	średnia arytmetyczna z 2 obrazów (piksel po pikselu)
imlincomb	kombinacja liniowa między serią obrazów
immultiply	iloczyn 2 obrazów
imsubtract	odejmowanie dwóch obrazów, minus stała

UWAGA: W przypadku plików typu uint8 lub uint16, dane po każdej cząstkowej operacji są ucinane do określonego zakresu. Może to powodować niezamierzone działania.

Temat 16
Metody poprawiania obrazów

imadjust	poprawienie kontrastu obrazka
fspecial	różnego rodzaju filtry poprawiające obraz
imfill	wypełnianie dziur w obrazie
histeq	poprawienie kontrastu obrazu na podstawie histogramu

Przykład:

```
>> filtr = fspecial('unsharp'); % generacja filtry wyostrzającego kontury
>> fobr = imfilter(obr,filtr); % zastosowanie zdefiniowanego filtru do obrazu Obr
```

Temat 17
Przetwarzanie obrazów

imcontour	znajdowanie konturów obiektów w obrazie
edge	znajdowanie krawędzi w obrazie
imfilter	splot obrazu z filtrem
deconv*	szereg funkcji przeprowadzających dekonvolucję
imregionalmax	szukanie lokalnych maksimów w obrazie
imnoise	dodanie szumu do obrazu

Przykład:

```
>> filtr = ones(3)/9; % generacja filtru uśredniającego 3x3
>> fobr = imfilter(obr,filtr); % zastosowanie filtru do obrazu
```

Temat 18
Morfologia matematyczna w przetwarzaniu obrazów

imclose	operacja zamknięcia morfologicznego
imopen	operacja otwarcia morfologicznego
imdilate	operacja delacji morfologicznej
imerode	operacja erozji morfologicznej
strel	tworzenie elementu strukturyzującego
imbothat	operacja morfologiczna dolny-kapelusz
imtophat	operacja morfologiczna górny-kapelusz
bwmorph	Operacje morfologiczne na obrazie czarno-białym

Przykład:

```
>> obr = imread('circles.png'); % wczytanie obrazka czarno-białego
>> elemstr = strel('disk',5); % definicja elementu strukturyzującego , okrąg o promieniu 5
>> erodeobr = imerode(obr,elemstr); % erozja obrazu elementem strukturyzującym
>> obr = bwmorph(BW,'remove')
```

Temat 19
Inne przydatne funkcje do pracy z obrazami

imhist	histogram obrazu
improfile	przekrój przez wiersz lub kolumnę obrazu
impixel	kolor lub wartość piksela
bwarea	liczenie powierzchni obiektu (obraz czarno-biały)
imrotate	obrót obrazu o dowolny kąt
imresize	zmiana rozmiaru obrazka
im2bw	Zamiana obrazka szaroodcieniowego na obraz binarny

Temat 20
Filmy w Matlabie

aviread	wczytanie pliku filmowego typu avi
movie	odtworzenie zmiennej typu avi
avifile	stworzenie nowego obiektu typu avi
getframe	pobranie wnętrza okna graficznego jako klatki filmu
addframe	dodanie ramki/obrazka do zmiennej typu avi

Przykład:

```
>> x=1:10; % oś X
>> for j=1:10,
       plot(x,j*x,'r'), % w pętli rysuje kolejne obrazki
       axis([0 100 0 10]); % ograniczenie zmienności wyświetlania osi wykresu
       F(j)=getframe; % pobieranie kolejnych klatek
   end
>> movie(F,10) % 10 krotne wyświetlenie animacji
```

Temat 21
Korelacyjne metody rozpoznawania obrazów

Korelacja jest miarą podobieństwa między 2 obiektami. Można ją wykorzystać do rozpoznawania obrazów.

Przykład:

```
function [w]=rozp1

% scena do filtru
a=double(imread('A.bmp')); % obrazki o rozmiarze 64x64
b=double(imread('B.bmp'));
c=double(imread('C.bmp'));
d=double(imread('D.bmp'));
filtr=zeros(256); % filtr będzie większy 256x256
filtr(128-64-32:128-64+31,128-64-32:128-64+31)=a; % poszczególne litery w 4 ćwiartkach
filtr(128+64-32:128+64+31,128-64-32:128-64+31)=b;
filtr(128-64-32:128-64+31,128+64-32:128+64+31)=c;
filtr(128+64-32:128+64+31,128+64-32:128+64+31)=d;
figure, subplot(2,2,1), imshow(filtr) % rysuje co zapamiętane na filtrze

% liczenie filtru
f=fft2(filtr);
filtr=conj(f)./abs(f);

% scena wejsciowa
we=zeros(256);
we(128-32:128+31,128-32:128+31)=a; % litera w środku obrazu 256x256
subplot(2,2,2), imshow(we) % rysuje scenę wejściową

% rozpoznanie – korelacja między obrazem wejściowym a sceną na filtrze
p1=fft2(we);
p2=p1.*filtr;
p3=fftshift(ifft2(p2));
wy=abs(p3).^2; % natężenie w płaszczyźnie wyjściowej

subplot(2,2,3), imshow(wy/max(max(wy))) % płaszczyzna korelacji 2D
subplot(2,2,4), mesh(wy) % płaszczyzna korelacji 3D
```

CAŁKOWANIE NUMERYCZNE

Temat 22
Całkowanie numeryczne

Problem polega na znalezieniu całki oznaczonej funkcji f(x) na przedziale <a,b>. Jest to łatwe, gdy znana jest funkcja pierwotna F(x) taka, że F(x)'=f(x) nie zawsze jest to jednak możliwe. Metody całkowania numerycznego (kwadratury) polegają na przybliżeniu funkcji podcałkowej f na danym przedziale <a,b> lub jego podprzedziałach przy pomocy innej funkcji, dla której wartość całki jest określona analitycznie. Matlab stosuje kwadratury Newtona-Cotesa - *quad* (interpolacja wielomianem drugiego stopnia) i Simpsona - *quadl* (interpolacja wielomianowa – dobierany stopień wielomianu).
Q=quad(f,a,b,tol,trace);
Q=quadl(f,a,b,tol,trace);
f – łańcuch zawierający nazwę funkcji, funkcja musi być umieszczona w odpowiednim skrypcie, musi zwracać wektor wartości a jej argumentem jest wektor elementów.
a,b – przedział całkowania,
tol – wymagana tolerancja względna, domyślnie 10^(-3)
trace – parametr opcjonalny, pozwala na rysowanie wykresu z węzłami kwadratury.

Przykład 1

```
function [y] = funkcja_calkowana(x)
%%  funkcja
y=sin(x.*x);
%% Koniec
```

```
%%Wywołanie funkcji całkowania
>> q1 = quad('funkcja_calkowana',0,pi,1e-5,1);
>> ql = quadl('funkcja_calkowana',0,pi,1e-5,1);
```

Temat 23
Uchwyt do funkcji – @

Istnieje kilka sposobów wywoływania funkcji całkowania. Jedną z nich, wykorzystywaną także w innych przypadkach, jest metoda oparta na uchwycie funkcji. Uchwyt @ można wykorzystać do definiowania własnych funkcji.

Przykład:

```
>> srednia = @(x,y) (x+y)/2; % definicja funkcji
>> srednia(5,3) % wywołanie z linii poleceń Malaba
```

Wykorzystanie uchwytu @ do całkowania funkcji przedstawiono poniżej.

Przykład:

```
>> fs = @(x) sin(x);
>> ql = quad1(fs,0,pi,1e-5,1);
```

Temat 24
Inne funkcje związane z całkowaniem numerycznym funkcji

dblquad	obliczanie całek podwójnych
trapz	całkowanie metodą trapezów
triplequad	obliczanie całek potrójnych

NUMERYCZNE RÓŻNICZKOWANIE FUNKCJI

Temat 25
Numeryczne różniczkowanie funkcji

Przybliżoną wartość pochodnej można obliczyć przez obliczenie różnic pomiędzy wartościami tych samych współrzędnych sąsiadujących punktów funkcji zadanej numerycznie. Korzysta się tu z definicji pochodnej funkcji:

$$\frac{d\,f(x)}{dx} = \frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

Do wykonania tej operacji wykorzystuje się funkcję *diff*(), która oblicza różnice pomiędzy sąsiadującymi elementami wektora.

Przykład

```
>> dxdy = diff(y)./diff(x) % y jest wektorem z wartościami funkcji w punktach x
```

Temat 26
Inne funkcje związane z numerycznym obliczaniem pochodnej.

gradient	Numeryczne obliczenie gradientu funkcji
del2	Laplasian funkcji

RÓWNANIA RÓŻNICZKOWE

Równania różniczkowe, które mamy możliwość rozwiązywania w Matlabie można podzielić na trzy grupy:

- równania różniczkowe zwyczajne gdzie szukamy rozwiązania równania różniczkowego dla zadanego warunku początkowego.
- równania różniczkowe zwyczajne gdzie szukamy rozwiązania równania różniczkowego dla zadanych warunków granicznych
- równania różniczkowe cząstkowe

Temat 27
Równania różniczkowe zwyczajne pierwszego rzędu

W rozwiązaniach wielu problemów fizycznych, ekonomicznych wymagana jest znajomość funkcji *y=y(t)* przy znajomości funkcji *y'=f(t, y)* oraz warunków początkowych *y(a)=y0*, gdzie *a* oraz *y0* są liczbami rzeczywistymi a *f* funkcją w postaci jawnej. W takim przypadku mamy do czynienia z równaniem różniczkowym zwyczajnym pierwszego rzędu (ordinary differential equations - ODEs).

Funkcja	Opis metody
ode23	Rozwiązuje zagadnienie początkowe dla równań różniczkowych zwyczajnych metodą Runge-Kutta rzędu 2 i 3
ode45	Rozwiązuje zagadnienie początkowe dla równań różniczkowych zwyczajnych metodą Runge-Kutta rzędu 4 i 5
ode113	Rozwiązuje zagadnienie początkowe dla równań różniczkowych zwyczajnych metodą Adams-Bashforth-Moulton
ode15s	Metoda oparta na formule numerycznego różniczkowania
ode23s	Metoda oparta na zmodyfikowanej formule Rosenbrocka 2 rzędu

Podstawowa formuła rozwiazywania ODE w Matlabie jest nastepujaca:
[t, y] = *ode23*(fun, tspan, y0,options]
gdzie:
fun – zmienna łańcuchową będąca nazwę funkcji zawierającą rozwiązywane równanie różniczkowe
tspan – wartościami czasu, dla którego poszukiwane jest rozwiązanie, y0 jest wektorem, w którym przechowywane są wartości rozwiązania układu w chwili początkowej.
Jeśli zmienna tspan zawiera więcej niż dwa elementy, to metoda zwraca obliczone wartości y dla tych elementów. Parametry wyjściowe t i y zawierają wektory wartości do obliczeń t i obliczone dla nich wartości y.
Wartości options są ustawiane za pomocą funkcji odeset i pozwalają ingerować w parametry rozwiązywania równania
Analogicznie rozwiązuje się równania różniczkowe dla metody ode45.

Przykład:

Szukamy rozwiązań numerycznych **y = y(t) dla wartości t = 0, .25, .5, .75, 1** dla równania różniczkowego $\mathbf{y' = -2ty^2}$, przy warunku początkowym **y(0) = 1**. Zastosowane zostaną dwie metody **ode23** i **ode45**.
Powyższy problem ma rozwiązanie analityczne **y(t) = 1/(1 + t2)** co pozwoli porównać otrzymane wyniki numeryczne z wynikami analitycznymi.

```
>> %% definicja rozwiązywanego równania różniczkowego y' = -2ty^2
function dy = eq1(t,y)
dy = -2*t.*y(1).^2;

%% rozwiązanie równania różniczkowego zwyczajnego
>> format long
>> tspan = [0 .25 .5 .75 1]; y0 = 1; % wartości dla których liczymy i warunki początkowe
>> [t1 y1] = ode23('eq1', tspan, y0); % metoda ode23
>> [t2 y2] = ode45('eq1', tspan, y0); % metoda ode45
>> [t1 y1 y2] % porównanie wyników osiągniętych obiema metodami
```

Przykład:

W poniższym przykładzie szukamy rozwiązań numerycznych układu równań różniczkowych pierwszego stopnia:

$$\begin{cases} y_1'(t) = y_1(t) - 4y_2(t) \\ y_2'(t) = -y_1(t) + y_2(t) \end{cases}$$

przy warunkach początkowych:

$y_1(0) = 1, y_2(0) = 0$.

Zastosowana zostanie metoda **ode23**.
Powyższy problem ma rozwiązanie analityczne

$$y_1 = \frac{1}{2}\left[\exp(-t) + \exp(3t)\right]$$

$$y_2 = \frac{1}{4}\left[-\exp(3t) + \exp(-t)\right]$$

co pozwoli porównać otrzymane wyniki numeryczne z wynikami analitycznymi.

```
%% definicja rozwiązywanego układu równań różniczkowych za pomocą funkcji inline
>> dy = inline('[1 –4;-1 1]*y', 't', 'y')
>> tspan = [0 1]; % przedział dla którego szukamy rozwiązań
>>y0 = [1 0]; % warunki początkowe
>>[t,y] = ode23(dy, tspan, y0); % metoda ode23
>>plot(t,y(:,1),t,y(:,2),'--'), legend('y1','y2'), xlabel('t'), ...
ylabel('y(t)'), title('Numerical solutions y_1(t) and y_2(t)') % zobrazowanie wyników
```

Przykład:

W poniższym przykładzie szukamy rozwiązania numerycznego układu równań różniczkowych dla równania Eulera dla bryły sztywnej bez sił zewnetrzych:

$$\begin{cases} y_1' = y_2 y_3 \\ y_2' = -y_1 y_3 \\ y_3' = -0,51 y_1 y_2 \end{cases}$$

```
function dydt = f(t,y) %% zapisujemy równia Eulera
dydt = [y(2)*y(3); -y(1)*y(3); -0.51*y(1)*y(2)];

>> tspan = [0 12]; % przedział, dla którego szukamy rozwiązań
>> y0 = [0; 1; 1]; warunki początkowe
>> ode45(@f,tspan,y0); % metoda ode45
```

Temat 28
Równania różniczkowe zwyczajne wyższego rzędu

Matlab pozwala również na rozwiązywanie równań różniczkowych wyższego rzędu. W tym przypadku trzeba jednak zapisać równanie w postaci układu równań różniczkowego pierwszego rzędu.

$y' = f(t, y)$

Dowolne równanie różniczkowe wyższego rzędu

$y^{(n)} = f\left(t, y, y', ..., y^{(n-1)}\right)$

można zapisać jako układ równań pierwszego rzędu stosując podstawienie

$y_1 = y,\ y_2 = y', ..., y_n = y^{(n-1)}$

Wtedy dostaniemy układ n równani różniczkowych pierwszego rzędu:

$$\begin{cases} y_1' = y_2 \\ y_2' = y_3 \\ \quad \vdots \\ y_n' = f\left(t, y_1, y_2, ..., y_n\right) \end{cases}$$

Przykład:

W poniższym przykładzie szukamy rozwiązania numerycznego równania różniczkowego drugiego stopnia van der Pola

$$y_1'' - \mu\left(1 - y_1^2\right) y_1' + y_1 = 0$$

Przyjmując podstawienie

$$y_1' = y_2$$

otrzymujemy układ równań pierwszego stopnia w postaci:

$$\begin{cases} y_1' = y_2 \\ y_2' - \mu\left(1 - y_1^2\right) y_2 - y_1 \end{cases}$$

```
function dydt = f(t,y,mu) % funkcja w której zapisaliśmy równanie różniczkowe
dydt = [y(2); mu*(1-y(1)^2)*y(2)-y(1) ];

>> Mu=100; % przyjmujemy parametr μ
>> tspan = [0; max(20,3*MU)]; % przedział, dla którego szukamy rozwiązań
>> y0 = [2; 0]; %warunki początkowe
>> [t,y] = ode15s(@f,tspan,y0); % metoda ode15s
>> plot(t,y(:,1)); % zobrazowanie wyników
>> title(['Solution of van der Pol Equation, \mu = ' num2str(MU)]);
>> xlabel('time t');
>> ylabel('solution y_1');
>> axis([tspan(1) tspan(end) -2.5 2.5]);
```

OBSŁUGA BŁĘDÓW

Temat 29
Komunikat błędu – funkcja *error*

Funkcja *error* wypisuje komunikat błędu: error('to jest błąd')
W celu wyprowadzenia na ekran komunikatu o błędzie w postaci okienka należy użyć funkcji *errordlg*(*'errorstrin','dlgname'*).

Przykład:

```
>> error('Cos jest nie tak');
>> errordlg('Cos jest nie tak','Komunikat o błędzie');
```

Temat 30
Komunikat ostrzeżenia – funkcja *warning*

Wypisuje komunikat ostrzeżenia.

Przykład:

```
>> warning('to jest ostrzeżenie');
```

Temat 31
Zmienne standardowe: *nargin*, *nargout*

Liczba parametrów w definicji funkcji może się różnić od liczby parametrów, jaką podaje użytkownik. Zmienna standardowa *nargin* określa dla danego wywołania funkcji z iloma parametrami funkcja ta została wywołana. Odpowiednio, zmienna *nargout* określa ile parametrów wyjściowych zostanie pobranych przez użytkownika.

Przykład:

```
function [y1, y2, y3, y4]=funkcja_x(x1, x2, x3, x4)
%%  [y1, y2, y3, y4]=funkcja_x(x1, x2, x3, x4)
% y1 ...y4 parametry wyjściowe funkcji
% x1....x4 parametry wejściowe funkcji

if (nargin<1), % nie ma wprowadzonych parametrów – wprowadzamy wartości domyślne
   x1=0; x2=0; x3=0; x4=0; % parametry wejściowe funkcji
elseif(nargin<2) % wprowadzono tylko jeden parametr
..... % jakaś akcja, np. komunikat błędu
elseif(nargin<3)
..... % jakaś akcja
end
% wszystkie warunki wprowadzania zostały sprawdzone

% testowanie ile parametrów wyjściowych chce otrzymać użytkownik, jeżeli użytkownik nie
chce dodatkowych parametrów wyjściowych szkoda czasu na ich obliczanie
if(nargout>1)
y2 = ....
y3= .....
y4= .....
% koniec funkcji
```

Zmienne *nargin*, *nargout* mogą być wykorzystywane do sprawdzania poprawności liczby wprowadzonych lub wyprowadzanych zmiennych z funkcji.

Przykład:

```
function funk(x,y)
if nargin ~= 2
    error('Wrong number of input arguments')
end
```

UWAGA: Do wprowadzania i wyprowadzania dowolnej liczby dowolnych elementów do i z funkcji wykorzystuje się odpowiednio funkcję *varargin* oraz funkcję *varargout*.

Temat 32
Reagowanie na nieprawidłowy argument funkcji

Do sprawdzenia czy dana zmienna ma oczekiwaną przez nas postać wykorzystujemy operatory typu is*.

ischar	Sprawdza czy dana wejściowa jest zmienną typu char
isempty	Sprawdza czy dana wejściowa jest zmienną pustą
isequal	Sprawdza czy dane 2 macierze są sobie równe
isglobal	Sprawdza czy dana wejściowa jest zmienną globalną
isinteger	Sprawdza czy wszystkie elementy ziennej są typu integer
islogical	Sprawdza czy dana wejściowa jest zmienną logiczną
isnan	Znajduje elementy nieskończone (NaN)
isnumeric	Sprawdza czy dana wejściowa jest zmienną numeryczną
isprime	Znajduje elementy będące liczbami pierwszymi
isreal	Sprawdza czy wszystkie elementy zmiennej są typu real
isscalar	Sprawdza czy dana wejściowa jest skalarem
issorted	Sprawdza czy dana wejściowa jest posortowana
issparse	Sprawdza czy dana wejściowa jest macierzą rzadką
isvector	Sprawdza czy dana wejściowa jest wektorem

Przykład:

```
if ~isreal(arg2),
   error('Zmienna musi być liczbą rzeczywistą')
end
```

```
>> a = 1:20; % zmienna wejściowa
>> p = isprime(a); % wektor z 1 tam gdzie dana liczba jest pierwsza
```

GRAFIKA UCHWYTÓW

Temat 33
Uchwyt

Każdy element graficzny wyświetlony w oknie wykresu ma swój uchwyt. Wykorzystując uchwyty możemy mieć dostęp do wszystkich elementów okna graficznego.
W celu uzyskania głównych uchwytów stosujemy funkcję *findobj*.

Przykład:

```
>> plot(1:10,'o-') % otworzenie okna graficznego i wyrysowanie linii prostej
>> h = findobj; % szukam głównych uchwytów
```

```
% wartości zwrócone przez Matlaba
h = 0
1.0000
73.0011
1.0050
```

Aby uzyskać te dane w bardziej zrozumiałej formie stosujemy następującą składnię:

```
>> get(h,'type')

ans = 'root'
'figure'
'axes'
'line'
```

W tej chwili wywołanie np. h(4) daje nam uchwyt do obiektu Line.

Temat 34
Parametry obiektów graficznych

W celu wyświetlenia listy dostępnych parametrów danego elementu wykresu korzystamy z funkcji *set*.

```
>> set(h(4))
```

Aby uzyskać opcje danego składnika pojedynczego elementu graficznego korzystamy z następującej składni:

```
>> set(h(4),'Marker') % słówko Marker określa interesujący nas składnik
```

Lista parametrów dostępna jest w opisie danego elementu graficznego.

Temat 35
Ustawianie parametrów graficznych – funkcja *set*

Mając uchwyt do danego elementu graficznego możemy zmienić jego parametry.
W tym celu korzystamy z funkcji *set(uchwyt, parametr,'wartość)*

Przykład:

```
>> set(h(4), 'Marker', 's', 'MarkerSize', 16) % kwadratowy marker ('s') o wielkości 16
```

Temat 36
Pobieranie bieżących parametrów obiektów graficznych – funkcja *get*

Mając uchwyt do danego elementu graficznego możemy sprawdzić jego parametry.
W tym celu korzystamy z funkcji *get(uchwyt,parametr*).

```
>> get(h(4),'Marker');
```

Temat 37
Inne funkcje związane z grafiką uchwytów

reset	Przywraca domyślne parametry obiektu
delete	Usuwa obiekt
drawnow	Wykonuje zaległe operacje graficzne
findobj	Szuka obiektu o zadanych parametrach
copyobj	Tworzy kopię obiektu

Temat 38
Bieżące okno graficzne

W przypadku aktywnego okna graficznego Matlab udostępnia 3 predefiniowane uchwyty:

gca	Uchwyt do osi współrzędnych
gcf	Uchwyt do okna graficznego
gco	Uchwyt do bieżącego obiektu

Temat 39
Inne funkcje związane z obsługą okien graficznych

figure	Utworzenie nowego okna graficznego
clf	Czyszczenie okna graficznego
shg	Pokazuje ostatnie okno graficzne
close	Zamyka bieżące okno graficzne
refresh	Odświeża zawartość bieżącego okna graficznego
cla	Czyści zawartość układu współrzędnych

GRAFICZNY INTERFEJS UŻYTKOWNIKA

Temat 40
Tworzenie graficznego interfejsu użytkownika – funkcja *guide*

Pisząc aplikacje na własny użytek często zapominamy o przyjaznym interfejsie. Jednak ze względu na funkcjonalność, łatwość obsługi i w przypadku, gdy z naszego programu mają korzystać także inni użytkownicy dobrze jest dodać do aplikacji graficzny interfejs użytkownika.
Aby rozpocząć tworzenia takiego interfejsu należy wywołać funkcję *guide*.
Po otworzeniu się okna dialogowego należy wybrać rodzaj tworzonego przez nas okna oraz katalog w którym ma być ono zapisane.

UWAGA: Matlab automatycznie wygeneruje 2 pliki. W jednym *.fig trzymana jest informacja o wyglądzie interfejsu a w drugim *.m szablon funkcji, którą będzie wykonywał interfejs.

Temat 41
Okno edytora graficznego interfejsu użytkownika.

Do głównych elementów okna edytora graficznego należą: menu, pasek narzędzi, elementy interfejsu oraz obszar roboczy.

Do najważniejszych elementów paska narzędzi należą:
ikona M-file editor – dostęp do kodu tworzonego interfejsu
run – uruchamianie tworzonego interfejsu

Temat 42
Praca nad interfejsem użytkownika.

Prace nad interfejsem użytkownika można podzielić na kilka etapów. Do najważniejszych należą: projektowanie, ustawianie elementów, właściwości elementów, programowanie, testowanie.

Temat 43
Dostępne elementy interfejsu użytkownika.

Matlab oferuje kilka elementów graficznych, które mogą się znaleźć w oknie tworzonego interfejsu użytkownika. Należą do nich:

push button	przycisk działający po naciśnięciu go myszką
slider	suwak
radio button	wybór 1 opcji z kilku
checkbox	pole wyboru. Możliwy wybór kilku opcji jednocześnie
edit text	pole edycji
static text	tekst bez możliwości jego edycji
popup menu	menu podręczne aktywowane po naciśnięciu prawym przyciskiem myszki
list box	rozwijana lista wyboru
toggle button	przycisk, który można wcisnąć na stałe
axes	układ współrzędnych
panel	obszar do ustawiania pogrupowanych innych elementów
button group	obszar do ustawiania grupy przycisków tego samego typu
activeX control	kontrolka activeX

Temat 44
Ustawianie elementów i ich właściwości.

W celu umieszczenia danego elementu interfejsu użytkownika na obszarze pola roboczego należy wskazać ten element a następnie nacisnąć (lub przeciągnąć) w obrębie pola roboczego. Elementy można przesuwać i wyrównywać z innymi. Dostęp do parametrów elementu możliwy jest po dwukrotnym kliknięciu lub w menu podręcznym danego elementu.

Temat 45
Właściwość obiektu *Tag*

Jednym z najważniejszych własności każdego elementu interfejsu użytkownika jest pole *Tag*. Wartość tekstowa wpisana w to pole daje dostęp do danego elementu oraz pozwala na reagowanie na wszelkie akcje związane z tym elementem (funkcja *callback*).

Temat 46
Zapamiętanie interfejsu użytkownika

Po etapie projektowania interfejsu użytkownika wygodnie jest zapamiętać rezultaty naszej pracy. Matlab automatycznie wygeneruje 2 pliki. W jednym *.fig trzymana jest informacja o wyglądzie interfejsu a w drugim *.m szablon funkcji, którą będzie wykonywał interfejs.

Temat 47
Reagowanie na akcje użytkownika – funkcja *callback*

Wygenerowane w czasie zapisywania pliki zawierają obsługę wszystkich zdarzeń związanych z wywoływaniem naszego interfejsu i wykonywaniem go w środowisku Matlaba. Pozostaje teraz zdefiniować wszystkie funkcje związane z interakcją użytkownika z projektowanym interfejsem i programem.
W tym celu z częścią elementów występującą w oknie interfejsu np. przyciskiem należy skojarzyć jakąś funkcję, która będzie wykonywana po jego naciśnięciu. W tym celu należy dla danego elementu zdefiniować funkcję *callback*.
Można to wykonać albo w oknie *Property Inspektor* danego elementu lub po naciśnięciu danego elementu prawym przyciskiem myszki.
Po stworzeniu funkcji *callback* w kodzie programu pojawi się odpowiadający jej nagłówek funkcji. Rolą projektanta interfejsu jest wpisanie w dalszej części programu instrukcji jakie mają zostać wykonane.

Przykład:

Załóżmy, że w interfejsie użytkownika mamy zdefiniowane dwa elementy: *PushButton* (Tag = Przycisk) oraz *StaticText* (Tag = Tekst). Zadaniem elementu tekstowego jest wyświetlanie informacji o liczbie naciśnięć przycisku.

```
function Przycisk_Callback(hObject, eventdata, handles)
% hObject    handle to Przycisk (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
persistent ile % definicja zmiennej w której trzymamy liczbę naciśnięć przycisku
if isempty(ile) % jeśli nie ma żadnej wartości to przypisuję wartość 0
   ile=0;
end
ile = ile + 1; % reakcja na naciśnięcie. Zwiększam licznik o 1
str=['Liczba nacisniec: ', Int2Str(ile)]; % tekst do wyświetlenia
set(handles.tekst,'String',str); % przypisanie właściwości do pola String obiektowi tekst
```

Temat 48
Zmienne w programie

Dobrą praktyką jest definiowanie tych zmiennych, które mają ulegać modyfikacji w trakcie działania programu w ramach uchwytów. Funkcje *callback* mają pełny dostęp do wszystkich elementów programu zdefiniowanych w uchwycie.

Przykład:

```
% Choose default command line output for untitled1
handles.output = hObject;

% DODANE
handles = guihandles(hObject); % pobranie uchwytu do całego okna interfejsu
handles.ile = 0; % dodanie nowego elementu do uchwytu i nadanie mu wartości
% KONIEC DODANEGO

% Update handles structure
guidata(hObject, handles); % zapamiętanie uchwytu
%

function Przycisk_Callback(hObject, eventdata, handles)
% hObject    handle to Przycisk (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.ile = handles.ile + 1; % zwiększenie licznika o jeden
guidata(hObject, handles); % zapamiętanie nowej wartości uchwytu
str=['Liczba nacisniec: ', Int2Str(handles.ile)]; % tekst do wyświetlenia
set(handles.tekst,'String',str); % przypisanie właściwości do pola String obiektowi tekst
```

Temat 49
Wykonanie programu ze zdefiniowanym interfejsem użytkownika

W celu wykonania programu z interfejsem użytkownika należy w oknie komend wpisać nazwę funkcji z interfejsem.

Temat 50
Interfejs graficzny – *EditBox*

Przykład:

```
function edit1_Callback(hObject, eventdata, handles)

str = get(handles.edit1, 'String'); % pobranie tekstu z elementu Edit1
set(handles.edit1, 'String', str); % ustawienie tekstu w obiekcie Edit1
```

Temat 51
Interfejs graficzny – *ToggleButton*

Przykład:

```
function togglebutton1_Callback(hObject, eventdata, handles)

stan = get(handles.togglebutton1,'Value'); % pobranie stanu przycisku togglebutton1
if stan==0,
  … % klawicz wyłączony
else
  … % klawisz włączony
end;
```

Temat 52
Interfejs graficzny – *RadioButton*

W Matlabie pola *RadioButton* nie są ze sobą powiązane. Oznacza to, że obsługę tych elementów polegającą na włączaniu lub wyłączaniu odpowiednich pól *RadioButton* musi przeprowadzić użytkownik.

Przykład:

```
function radiobutton1_Callback(hObject, eventdata, handles)

set(handles.radiobutton1,'Value',1); % Włączenie zaznaczenia pola RadioButton1
set(handles.radiobutton2,'Value',0); % Wyłączenie zaznaczenia pola RadioButton2
```

Temat 53
Interfejs graficzny – *ListBox*

Wszystkie opcje dostępne do wyboru w obiekcie *ListBox* są dostępne we właściwości 'String'.

Przykład:

```
function listbox1_Callback(hObject, eventdata, handles)

opcja = get(handles.listbox1, 'Value'); % pobranie numeru wybranej linii
str = ['Opcja', num2str(opcja)]; % zmienna tekstowa z informacją o wybranej opcji

tresc = get(handles.listbox1, 'String'); % pobranie wszystkich linii tekstu z listbox1
set(handles.text1, 'String', str(opcja)); % wyświetlenie w obiekcie text1 (StaticText) tekstu z
                                          % wybranej linii obiektu listbox1
```

UWAGA: Z podobną obsługą mamy do czynienia w przypadku Pop-up Menu.

Temat 54
Interfejs graficzny – *Slide*

Położenie suwaka określone jest liczbą z przedziału od 0.0 do 1.0.

Przykład:

```
function slider1_Callback(hObject, eventdata, handles)

polozenie = get(handles.slider1, 'Value'); % pobranie informacji o położeniu suwaka
```

Temat 55
Interfejs graficzny – *Axes*

Załóżmy, że w oknie interfejsu graficznego mamy 2 elementu. Przycisk (PushButton) i Wykres (Axes). Naciśnięcie przycisku powoduje wyświetlenie na wykresie okręgu.

Przykład:

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

t=0:0.01:2*pi; x=cos(t); y=sin(t); % dane do wykresu
axes(handles.wykres) % pobranie uchwytu do wykresu
plot(x,y,'r') % wyrysowanie koła
```

Temat 56
Okna do wprowadzania danych – funkcja *inputdlg*

Prostą metodą zrobienia prostego interfejsu do wprowadzania danych jest skorzystanie z funkcji *inputdlg*.

Przykład:

```
pyt{1} = 'Zmienna X:'; % Tekst przy polu do wprowadzenia zmiennej 1
pyt{2} = 'Zmienna Y:'; % Tekst przy polu do wprowadzenia zmiennej 2
tytul = 'Moje Okno'; % nazwa okna
odp = {'5','7'}; % opcjonalne wartości domyślne
wynik = inputdlg(pyt, tytul, 1, odp); % wywołanie okna dialogowego
```

ZŁOŻONE STRUKTURY DANYCH

Temat 57
Struktury

Struktury w Matlabie służą do przechowywania danych różnych typów i różnych wielkości w obrębie jednej macierzy. Jest to zbiór takich samych rekordów.
Przykład definiowania struktury danych typu student:

```
>> student(1).nazwisko = ‘Abacki’;
>> student(1).wiek = ‘20’;
>> student(1).dane = rand(3);
>> student(2).nazwisko = ‘Babacki’;
>> student(2).wiek = ‘22’;
>> student(2).dane = randn(3);
```

Temat 58
Definiowania struktury przy wykorzystaniu funkcji *struct*

Do definiowania danych zorganizowanych w strukturę wykorzystujemy funkcję *struct*.

Przykład:

```
>> student = struct(‘nazwisko’,{‘Abacki’,’Babcki’},…
                   ‘wiek’,{20,22},…
                   ‘dane’,{rand(3),randn(3)});
```

Takie definiowanie struktury daje możliwość wcześniejszej alokacji pamięci co wpływa na szybkość wykonywania operacji:

```
>> student = repmat(struct('nazwisko',{''},'wiek',{''},'dane',{''}),10,1);
% alokacja pamięci dla 10 studentów
```

UWAGA: Funkcja *repmat* służy do powielania elementów.

Temat 59
Wyświetlanie danych ze struktury

Ze względu na fakt, że w pojedynczej strukturze mogą być zapisane elementy o różnym stopniu zagłębienia, do wyświetlania poszczególnych danych wykorzystuje się różną składnię.

Przykład:

```
>> student  % Nazwa struktury. Wyświetlenie nazw pól
>> student(1) % Wyświetlenie danych dla danego rekordu
>> student(2).wiek % Wyświetlenie konkretnej danej
>> celldisp(student) % Wyświetlenie całości danych dla struktury student
```

Temat 60
Macierz komórek – cell array

Główna różnica między strukturą a macierzą komórek jest fakt, że w macierzy komórek odwołujemy się do poszczególnych elementów za pomocą adresu a nie za pomocą nazwy elementu.
Macierz komórek można rozumieć jako macierz, gdzie każdy element macierzy może być inną macierzą, zmienną lub strukturą.

Przykład:

```
>> A = {1:5, pi; rand(3), 'tekst'}
```

```
A =
   [1x5 double]    [3.1416]
    [3x3 double]    'tekst'
```

Temat 61
Definiowania macierzy komórek przy wykorzystaniu funkcji *cell*.

Przykład:

```
>> a = cell(2,2); % definicja macierzy komórek o rozmiarze 2x2 z pustymi komórkami
```

Takie definiowanie macierzy komórek daje możliwość wcześniejszej alokacji pamięci, co wpływa na szybkość wykonywania operacji.

Temat 62
Wyświetlanie danych z macierzy komórek.

Wyświetlanie danych z macierzy komórek jest podobne jak przy wyświetlaniu elementów zwykłej macierzy z tą różnicą, że teraz korzystamy z nawiasów { }.

Przykład:

```
>> a{1,1} % Wyświetlenie pojedynczego elementu
>> a{2,:} % Wyświetlenie kolumny elementów
>> celldisp(a) % Wyświetlenie całości danych
>> cellplot(a) % Graficzne przedstawienie danych
```

Temat 63
Konwersja typów między strukturą a macierzą komórek

struct2cell	zamiana strukturę na macierz komórek
cell2struct	zamiana macierzą komórek na strukturę

Przykład:

```
>> a = {1:5, pi; 1:10, 4*3}; % Zdefiniowanie macierzy komórek
>> pola = {'wektor','liczba'}; % zdefiniowanie nazwy pól
>> cell2struct(a,pola,2); % Zamiana macierzy komórek na strukturę
```

Temat 64
Definiowanie klas - konstruktor

Konstruktor danej klasy musi być umieszczony w funkcji o nazwie *nazwa_klasy.m*. Musi się on znajdować w osobnym folderze o nazwie *@nazwa_klasy*. W folderze tym muszą też znajdować się wszystkie metody (funkcje i operatory) działające na elementach danej klasy. Dostęp do klasy mają tylko te funkcje.
Zadaniem konstruktora jest: sprawdzenie czy przy wywołaniu konstruktora dane są wartości pól obiektu i jeśli nie to stworzenie obiektu pustego, sprawdzenie czy dane są obiektem danej klasy, lub wstawienie domyślnych parametrów do pól tworzonego obiektu.

Przykład:

```
function c=czlowiek(dane)
% konstruktor człowiek.m dla klasy obiektów 'czlowiek'
% zapisane w katalogu @czlowiek
if nargin==0 % brak parametrów wywołania
   c.imie = [ ];
   c.plec = [ ];
   c.wiek = [ ];
   c = class(c,'czlowiek'); % utworzenie pustego obiektu 'człowiek'
elseif isa(dane,'czlowiek') % funkcja isa sprawdza czy dana należy do klasy czlowiek
   c = dane; % dane wejściowe są obiektem typu 'czlowiek'
else
   c.imie = dane(1);
   c.plec = dane(2);
   c.wiek = dane(3);
   c = class(c,'czlowiek'); % tworzony obiekt 'czlowiek' z polami wypełnionymi danymi 'dane'
end
```

Temat 65
Tworzenie obiektu

```
>> Jacek = człowiek({'Jacek','m',26});
```

Temat 66
Hermetyzowanie danych

Dostęp do danych w polach obiektu i możliwości operowania nimi mają tylko wyłącznie metody danej klasy.
Do wyświetlenia nazwy pól obiektu wykorzystuje się funkcję *fieldnames*.

Przykład:

```
>> fieldnames(Jacek) % wyświetli nazwy pól obiektu Jacek
>> Jacek.plec % BŁĄD, brak dostępu do danych
```

Temat 67
Metody obsługi obiektów

Zadaniem metody *emeryt* jest sprawdzenie czy dany człowiek jest w wieku emerytalnym. Metoda musi być zapisana w tym samym katalogu, co konstruktor czyli @czlowiek.

```
function emeryt(czlowiek)
p = czlowiek.plec;
w = czlowiek.wiek;
if p=='m'
   if w>=65
     display('EMERYT')
   else
      display('NIE EMERYT')
else
   if w>=60
     display('EMERYT')
   else
      display('NIE EMERYT')
end
```

```
>> Anna = człowiek({'Anna','k',54}); % utworzenie obiektu
>> emeryt(Anna) % wywołania funkcji emeryt
```

Temat 68
Przeciążanie funkcji i operatorów

Warunkiem przeciążenia operatora lub funkcji jest dostęp do folderów klas, dla których je definiujemy. Każdy operator obsługiwany jest przez odpowiednią funkcję. Np. dodawanie + wykonywane jest przez funkcję *plus*. Przeciążenie operatora polega na zdefiniowaniu jak dany operator ma działać dla obiektów, które zdefiniowaliśmy.

Przykład:

```
function w=gt(a,b)
% przeciążenie operatora gt(a,b) odpowiednik a>b
% porównuje wiek 2 obiektów klasy 'czlowiek'
p = a.wiek - b.wiek;
if p >= 0
   w = 1;
else
   w = 0;
end;
```

Temat 69
Dziedziczenie klas obiektów

W programowaniu obiektowym można tworzyć hierarchię klas. Na podstawie jednych klas tworzyć nowe klasy o np. bardziej zawężonym charakterze.

Przykład:

```
function s=student(dane)
% klasa ‘student’ jest potomkiem klasy ‘czlowiek’
if nargin==0 % brak parametrów wywołania
   c=człowiek;
   o.ocena = [ ];
   s = class(o,’student’,c); % utworzenie pustego obiektu ‘student’
elseif isa(dane,’student’) % sprawdzenie czy zmienna dana należy do klasy ‘student’
   s = dane; % dane wejściowe są obiektem typu ‘student’
else
   c = czlowiek({dane(1), dane(2), dane(3)});
   o.ocena = dane(4);
   s = class(o,’student’,c); % tworzony obiekt ‘student’ z polami wypełnionymi danymi ‘dane’
end
```

OBLICZENIA SYMBOLICZNE W MATLABIE

Temat 70
Definiowanie danych symbolicznych – funkcja *sym*.

Do definiowania zmiennych lub stałych symbolicznych służą funkcje *sym* i syms.

Przykłady:

```
>> x = sym('x'); % definicja zmiennej x
>> a = sym('5'); % definicja stałej a=5
>> rho = sym('(1+sqrt(5))/2'); % definicja zmiennej symbolicznej określającej złoty środek
>> syms y z % definicja dwu zmiennych y, z
```

Uwaga: Wszystkie zmienne, wykorzystywane w obliczeniach symbolicznych muszą być wcześniej zdefiniowane.

Temat 71
Symboliczne rozwiązywanie równań i układów równań – funkcja *solve*

Symboliczne rozwiązywanie równań możliwe jest dzięki wykorzystaniu funkcji *solve()*. Dla przykładu rozwiązania równania kwadratowego szukamy w następujący sposób:

```
>> x = sym('x'); % zdefiniowanie zmiennej symbolicznej
>> y = solve('a*x^2+b*x+c=0'); % x jest domyślną zmienną
```

Funkcję *solve* można również wywoływać w inny sposób:

```
>> y = solve('a*x^2+b*x+c',x); % równoważne poprzedniemu wywołaniu, podana zmienna x
>> y = solve('a*x^2+b*x+c'); % domyślnie przyjęte, że lewa strona równania równa się 0
```

Podobnie postępujemy przy układach równań gdzie kolejne równania wpisujemy rozdzielone przecinkami.

Przykład:

```
>> syms x y % zdefiniowanie 2 zmiennych symbolicznych
>> [x,y] = solve('x^2+y^2=1','x+2*y=2'); % rozwiązanie. Wynik w postaci 2 wektorów
>> s = solve('x^2+y^2=1','x+2*y=2'); % rozwiązanie. Wynik w postaci struktury z 2 wektorami
```

Temat 72
Upraszczanie wyrażeń symbolicznych – funkcja *simplify()*

Często wynik operacji symbolicznych jest trudny do interpretacji. Uproszczenie wyniku uzyskujemy dzięki zastosowaniu funkcji *simplify*.

Przykład:

```
>> simplify(exp(x)*exp(y)) % =exp(x+y)
```

Temat 73
Zmiana postaci wyrażenia symbolicznego

W zależności od zastosowań wiele wyrażeń matematycznych można przedstawić w różnych formach. Matlab oferuje kilka funkcji, których zadaniem jest takie przedstawienie wyniku, aby spełniało nasze kryteria:

collect	postać wielomianowa, grupowanie zmiennych
expand	rozwija mnożenia i sumowania
horner	postać zagłębionych iloczynów
factor	wielomiany w postaci iloczynowej

Temat 74
Inne możliwe przedstawienia wyniku – funkcja *simple()*

W przypadku, gdy nie możemy się zdecydować, które przedstawienie jest najodpowiedniejsze do naszych celów wygodnym może być wykorzystanie funkcji *simple()*, która dane wyrażenie matematyczne wypisuje we wszystkich możliwych formach.

Temat 75
Wartości liczbowe w obliczeniach symbolicznych – funkcja *subs()*

Gdy znamy wartości liczbowe poszczególnych zmiennych wynik liczbowy obliczeń symbolicznych uzyskujemy dzięki funkcji *subs*.

Przykład:

```
>> syms a b c x % definicja 4 zmiennych symbolicznych
>> y = solve(a*x^2+b*x+c); % rozwiązanie równania względem zmiennej x
>> a=3; b=2; c=1; % Przypisanie wartości liczbowych
>> w = subs(y) % Wartość liczbowa y
```

Uwaga: Jeśli wynik obliczeń jest już liczbą ale przedstawioną w mało zrozumiałej formie w celu uzyskania jej wartości liczbowej wystarczy napisać: double(x).

Funkcja *subs()* wykorzystywana jest też do zamiany jednego wyrażenia na inne.

Przykład:

```
>> syms a b x % definicja 3 zmiennych symbolicznych
>> subs(a+b,{a,b},{sin(x),exp(x)}) % = sin(x)+exp(x)
```

Temat 76
Zmiana wyglądu wyświetlanych wyników – funkcja *pretty()*

Wyniki uzyskiwane w Matlabie, nawet po uproszczeniach dzięki różnym funkcjom często są trudne w odbiorze. Aby wyświetlić bardziej skomplikowane wyrażenia w formie zbliżonej do tej jak zapisuje to człowiek Matlab oferuje funkcję *pretty*.

Przykład:

```
>> a = sym('sin(x)/(2*cos(y))'); % definicja wyrażenia symbolicznego
>> pretty(a) % wyświetlenie w przyjaznej formie
```

UWAGA: Funkcja ta ma swoje zastosowanie zarówno do wyrażeń jak i do macierzy.

Temat 77
Różniczkowanie symboliczne – funkcja *diff*

Do różniczkowania symbolicznego wykorzystujemy funkcję diff(funkcja, rząd), przy, czym rząd pochodnej domyślnie równa się 1.

Przykład:

```
>> syms x n % definicja 2 zmiennych symbolicznych
>> diff(x^n) % różniczkowanie symboliczne
```

Temat 78
Pochodne cząstkowe.

W przypadku funkcji wielu zmiennych możliwe jest liczenie pochodnych cząstkowych. W tym celu także korzystamy z funkcji *diff()* z wywołaniem: diff(funkcja, zmienna)

Przykład:

```
>> syms s t % definicja 2 zmiennych symbolicznych
>> f = sin(s*t); % definicja funkcji 2 zmiennych
>> diff(f,t) % pochodna funkcji po zmiennej t
```

Temat 79
Całkowanie symboliczne – funkcja *int()*

W Matlabie można całkować zarówno całki oznaczone jak i nieoznaczone. W przypadku całki nieoznaczonej stosujemy wywołanie funkcji w postaci: *int*(funkcja)
W przypadku całki oznaczonej wywołanie ma następującą postać: *int*(funkcja, dolne ograniczenie, górne ograniczenie).

Przykład:

```
>> int('x'); % całko nieoznaczona
>> int('sqrt(tan(x))',0,1); % całka oznaczona
```

Temat 80
Symboliczne rozwiązywanie równań różniczkowych – funkcja *dsolve()*

Równania różniczkowe dowolnego rzędu rozwiązujemy przy wykorzystaniu funkcji *dsolve*, której użycie jest podobne do użycia funkcji solve. Szczególne znaczenie w symbolicznym rozwiązywaniu równań różniczkowych ma zmienna D (duże D), która określa różniczkę pierwszego stopnia, i podobnie D2 to różniczka stopnia drugiego, itd.

Przykład:

```
>> dsolve('Dy = a*x'); % różniczka pierwszego stopnia
```

W przypadku układu równań, kolejne równania oddzielamy przecinkami. Podobnie warunki brzegowe, podajemy po wszystkich równaniach, także oddzielone przecinkami:
dsolve(równanie 1, równanie 2, …, warunek 1, warunek 2, …)

Przykład:

```
>> dsolve('D2y = -a^2*y', 'y(0) = 1', 'Dy(pi/a) = 0'); % różniczka stopnia drugiego z warunkami
                                                        brzegowymi
```

Temat 81
Inne funkcje Matlaba wykorzystywane do obliczeń symbolicznych

limit	granica
taylor	szeregiTtaylora
jacobian	macierz Jacobiego
symsum	sumowanie szeregów

Temat 82
Symboliczne odpowiedniki funkcji Matlaba

Część funkcji Matlaba wykorzystywana w obliczeniach numerycznych ma swe odpowiedniki symboliczne. Do najważniejszych należą:

diag	macierz diagonalna
inv	macierz odwrotna
det	wyznacznik macierzy
eig	wartości własne
expm	potęgowanie macierzy

Temat 83
Obliczenia o zmiennej precyzji – funkcja *vpa*

W przypadku gdy interesuje nas wartość liczbowa z dokładnością większą niż oferuje zmienna double możemy podać ile cyfr znaczących danej zmiennej ma być wyświetlanych:
vpa(zmienna, ilość cyfr);

Przykład:

```
>> vpa(pi,50) % wypisuje 50 cyfr znaczących liczby pi
```

WSPÓŁPRACA MATLABA Z INNYMI ŚRODOWISKAMI PROGRAMISTYCZNYMI

Temat 84
Kompilator

Zadaniem kompilatora Matlaba jest zamiana funkcji zapisanej w pliku typu M-file na wersję wykonywalną, bibliotekę MEX plik lub kod programu w C czy C++.
Kompilator generuje kilka rodzajów plików:
- kod w języku C do budowania plików typu MEX
- kod w języku C lub C++
- wykonywalną aplikacją
- biblioteki dynamiczne i statyczne

Dzięki wykorzystaniu kompilatora uzyskujemy przyspieszenie pracy aplikacji, ukrycie kodu przed innymi użytkownikami oraz możliwość budowy niezależnych aplikacji.

Temat 85
Wywołanie kompilatora – funkcja *mcc*

Składnia wywołania funkcji:
mcc [-opcja] funkcja [funkcja2 …] [mex1 … mexN] [mlib1 … mlibN]
Gdzie:
W nawiasach kwadratowych umieszczone parametry opcjonalne,
opcja – litery oznaczające różne opcje kompilatora
funkcja – nazwa funkcji poddawana kompilacji,
max – nazwa pliku typu MEX
mlib – nazwa biblioteki

Najważniejsze opcje:

-c	Generuje kod C
-p	Generuje kod C++ oraz działającą aplikację
-x	Generuje plik typu MEX
-t	Generuje kod w C lub w C++
-T <opcje>	W zależności od opcji generuje różne pliki: codegen compile:mex compile:exe compile:lib link:mex link:exe link:lib

Przykład:

```
>> mcc –x funkcja % generuje plik typu MEX na podstawie pliku funkcja.m
>> mcc –m funkcja % generuje wykonywalną aplikację na podstawie pliku funkcja.m
>> mcc –p funkcja % generuje kod w C++ i wykonywalną aplikację
>> mcc –t –L C % generuje kod w C na podstawie pliku funkcja.m
>> mcc –t –L Cpp % generuje kod w C++ na podstawie pliku funkcja.m
>> mcc –m funkcja1 funkcja2 % generuje wykonywalne aplikacje na podstawie plików
     funkcja1.m oraz funkcja2.m
>> mcc –T compile:exe funkcja % generuje wykonywalną aplikację
>> mcc –t W lib:a –T link:lib a0 a1 % generuje bibliotekę dynamiczną „a” na podstawie plików
    a0.m oraz a1.m
```

Temat 86
Ograniczenia użycia funkcji kompilatora

Kompilator pracuje tylko na funkcjach zapisanych w plikach typu M. Nie kompiluje skryptów.
Kompilacji nie podlegają funkcje wykorzystujące obiekty.
Kompilacji nie podlegają funkcje korzystające z poleceń *input*, *eval* jeśli te działają na zmiennych z *workspace* oraz operujące na zmiennych globalnych.

Uwaga: Kompilacji nie podlegają funkcje matlabowe, które nie mają swojego pliku typu M.

Temat 87
Współprace Matlaba z Excelem
Do wczytania danych z programu Microsoft Excel wygodnie jest wykorzystać okno importu danych dostępne z menu Matlaba.
Do zapisania zmiennych z Matlaba w formie łatwej do otworzenia w Excelu należy skorzystać z funkcji xlswrite.

Przykład:

```
>> x=rand(5); % generacja przykładowej macierzy
>> xlswrite('dane',x) % zapisuje w pliku Excelowym dane.xls zmienną x
```