MATLAB – ELEMENTY GRAFIKI I PROGRAMOWANIA

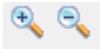
GRAFIKA

Polecenia graficzne można wydawać w Oknie Poleceń (OP) w trybie wsadowym jedno po drugim – wtedy efekty widoczne są od razu po wprowadzeniu polecenia. Jednakże w przypadku jakiegoś błędu, lub np. zamknięcia okienka graficznego wszystkie polecenia trzeba by było – jedno po drugim – wynajdywać z historii przestrzeni roboczej i wykonywać. Dlatego o wiele lepiej jest pracować w trybie edytora, zwłaszcza, iż opracowanie jednego wyniku graficznego niekoniecznie wiąże się z wydaniem tylko jednego polecenia. W takim przypadku zbieramy wszystkie linijki do jednego pliku i dopiero wtedy wykonujemy, gdy układają się już w logiczną całość. Wtedy Matlab przenosi te linijki jedna po drugiej do okna poleceń i wykonuje tak długo, aż napotka błąd, lub gdy skończy się wykonywany plik. Zanim jednak przejdziemy do organizowania programów graficznych, wykonajmy dwa przykładowe polecenia, rysujące „za jednym zamachem" wykresy funkcji.

Polecenie „fplot" rysuje wykres funkcji danej wzorem jawnym, czyli za pomocą prostej zależności $y = y(x)$. Jako pierwszy argument należy podać część prawą poprzednio podanej zależności (czyli $y(x)$) objętą w znaki ' ' (prim). Jest to nic innego, jak zmienna tekstowa, którą funkcja „fplot" zamieni na wartość liczbową przy określeniu wartości „x". Drugim koniecznym argumentem funkcji „fplot" jest przedział, w jakim funkcja ma być narysowana. Przykładowo

```
>> fplot('sin(x)',[-3*pi 2*pi])
```

spowoduje otwarcie okna graficznego, z wykresem funkcji $y = \sin(x)$ widocznym w przedziale $[-3pi \quad 2pi]$. Kolor wykresu oraz jego grubość są dobierane domyślnie, o sposobie sterowania nimi będzie mowa przy okazji omawiania bardziej ogólnej funkcji „plot".

Każde okno graficzne rządzi się swoimi prawami i można sterować jego parametrami niezależnie od wydawania poleceń w OP. Należą do nich m.in. przybliżenie, którym można sterować za pomocą lupek  . Np. uaktywnienie lupki „dodatniej"  otworzy okno przy kursorze, które pozwoli na zdefiniowanie zakresu przybliżenia. Proszę spróbować w ten sposób przybliżyć okolice jednego z maksimów („szczytów") narysowanej funkcji. Po dokonaniu powiększenia i naciśnięciu prawego klawisza myszy (PKM) możemy powrócić do początkowego widoku wybierając opcję „Reset to original view". Narzędzie „łapki"  po

prawej stronie lupek pozwala na przesuwanie wykresem, czyli zmianę współrzędnych punktu obserwacyjnego. Narzędzie obrotu 3D  („rotate 3D”) pozwoli na przestrzenne obracanie wykresem. Powrót do oryginalnego widoku płaskiego można zawsze uzyskać po wybraniu PKM opcji „Reset to original view” (lub w tym przypadku „Go to X-Y view”). O wiele więcej opcji jest dostępnych w górnym menu „Insert” oraz „Tools”, jednakże mają one tylko wpływ na aktualnie oglądany wykres. Jeżeli np. ustawimy odpowiednią skalę, położenie i widok wykresu, a następnie go zamkniemy, to po przywróceniu polecenia graficznego wykres będzie miał na powrót wygląd domyślny. Proszę np. dokonać przybliżenia oraz obrócić wykres, a następnie zamknąć okno graficzne znakiem „x” w górnym prawym rogu. Następnie proszę przywrócić ostatnio wykonane polecenie. Okno graficzne będzie miało wygląd taki sam, jak na początku, bez zmian, które zostały przez nas wprowadzone. Dlatego też, jeżeli mają mieć one charakter trwały, należy je zaprogramować w kolejnych linijkach programu. Narysujmy jeszcze jedną funkcję w OP, mianowicie o wzorze niejawnym, uwikłanym $x^2 + y^2 = 4$. Jest to wzór na okrąg o środku w punkcie (0,0) oraz promieniu 2. Ponieważ nie da się jednoznacznie z powyższego wzoru wyznaczyć relacji $y = y(x)$, nie da się zastosować jednokrotnie funkcji „fplot”. Do tego rodzaju sytuacji można wykorzystać funkcję „ezplot”, służącą do rysowania na płaszczyźnie wykresów funkcji wg relacji $f(x, y) = 0$. Przykładowo

```
>> ezplot('x^2 + y^2 - 4')
```

spowoduje narysowanie okręgu o wymienionych wyżej parametrach oraz umieszczenie wzoru na ten okrąg (w postaci tekstu) w tytule wykresu. Proszę zwrócić uwagę na skalę rysunku – nie jest ona dostosowana idealnie do wykresu, ponieważ funkcja „ezplot” domyślnie rysuje wykresy w przedziale $[-2\,pi \quad 2\,pi]$ oraz skaluje osie x i y w ten sam sposób. Proszę okno zamknąć i wywołać poprzednią linijkę wzbogacając wykonanie funkcji o parametr przedziałowy

```
>> ezplot('x^2 + y^2 - 4',[-2 2])
```

Proszę okno zamknąć. Oczywiście jako bardziej ogólna, funkcja „ezplot” nadaje się też do rysowania wykresów funkcji wg pierwszej jawnej zależności $y = y(x)$, np.

```
>> ezplot('sin(x)-y',[-3*pi 2*pi])
```

O ustawianiu zakresów osi x I y będzie mowa w dalszym ciągu opracowania. Proszę zamknąć okno graficzne.

Dalsze czynności – operacje graficzne – będą wykonywane przy pomocy edytora. Proszę z górnego menu programu Matlab wybrać opcję „New Script” (ikonka zagiętej karteczki z gwiazdką  pod menu „File”) lub nacisnąć klawisze „Ctrl+N”. Pojawi się – domyślnie

niezadokowane – okienko edytora pliku, na razie bez nazwy ("Untitled"). Plik ten ma domyślne rozszerzenie ".m". Można w nim pisać dowolną liczbę poleceń – tych samych, które pisaliśmy do tej pory w OP, ale nie będą one wykonywane do chwili, kiedy tego wyraźnie zażądamy.

Zacznijmy od stworzenia wykresów powyższych dwóch funkcji w najbardziej ogólny dostępny sposób, czyli z wykorzystaniem funkcji "plot". Działa ona na zasadzie łączenia kawałkami odcinków prostych punkty pośrednie wygenerowane w przedziale, w którym chcemy zobaczyć wykres funkcji. Jeżeli te punkty będą położone bardzo blisko siebie, to przy odpowiednio gęstym podziale powstanie wrażenie, iż oglądamy wykres funkcji ciągłej i gładkiej, a nie - jak w rzeczywistości - łamanej.

Napiszemy program rysujący wykres funkcji $y = \sin(x)$ za pomocą funkcji "plot" Rozpocznijmy pisanie programu od linijki, która wygeneruje w przedziale $[-3pi \quad 2pi]$ zbiór równomiernie rozłożonych punktów co 0.01. W tym celu wygenerujemy wektor "x" za pomocą operatora zakresu

```
x = -3*pi:0.01:2*pi;
```

Linijkę powyższą warto zakończyć znakiem średnika (";"). Wektor, który się wygeneruje po jej późniejszym wykonaniu, będzie składał się z setek elementów - nie ma potrzeby wyświetlać ich wszystkich w OP.

Następnie zbudujemy wektor wartości danej funkcji za pomocą instrukcji

```
y = sin(x);
```

Warto zwrócić uwagę, że powyższy zapis wygeneruje **wektor**, ponieważ zmienna "x" z poprzedniej linijki programu też będzie **wektorem**. Wektory "x" i "y" będą miały tyle samo elementów - będą to wektory "leżące", czyli wierszowe.

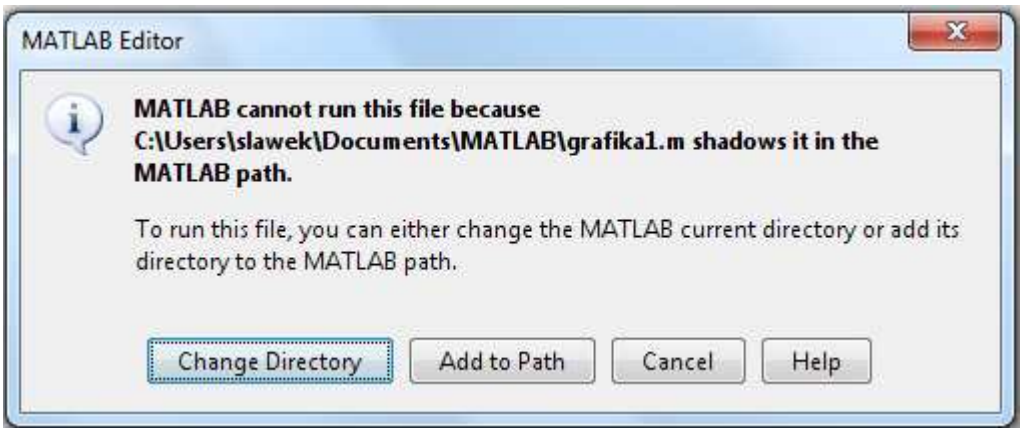
Ostatnią linijką programu będzie narysowanie łamanej przez wszystkie punkty

```
plot(x,y);
```

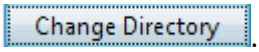
Funkcja "plot" ma bardzo prostą składnię: jako pierwszy argument podajemy wszystkie współrzędne x-owe, jako drugą - współrzędne y-owe punktów, które chcemy narysować i połączyć odcinkami. Domyślnie - bez podawania stylu rysowania tych linii i punktów, punkty będą oznaczone małymi kropeczkami ekranowymi (pikselami), a linie będą ciągłe i będą mały kolor niebieski. Cały program powinien mieć więc postać następującą:

```
x = -3*pi:0.01:2*pi;
y = sin(x);
plot(x,y);
```

Jak go uruchomić? Przede wszystkim trzeba go zapisać na dysku - w wybranej lokalizacji. W tym celu można posłużyć się opcją "File-Save" lub "File-Save As" z górnego menu edytora, lub ikonką niebieskiej dyskietki  z paska pod głównym menu edytora. Pojawi się okno zapisu i prośba o podanie nazwy pliku - oczywiście można zapisać go pod nazwą "Untitled", jaką podpowiada Matlab, ale lepiej jest nadać programowi (plikowi) taką nazwę, aby kojarzyła się z tym, co program robi. Jeżeli będziemy go szukać na dysku np. za parę miesięcy, wtedy już po nazwie będzie się można zorientować, co siedzi w środku. I tu ważna uwaga: nazewnictwo plików w Matlabie zawierających programu podlega takim samym regułom, jak nazewnictwo zmiennych (czyli nie można nazwać programu "jan kowalski.m", "1.m", "program 1.m", "plot.m", "sin.m". Wszystko to przykłady nazw **niepoprawnych** - dwie ostatnie to nazwy wbudowanych funkcji Matlaba i nazwanie tak programu zablokuje dostęp do tych funkcji. **Poprawne** nazwy to np. "jan_kowalski.m", "prog1.m", "program_1.m", "prog_plot.m", "sinus.m", ale lepiej jest nadać nazwę typu "grafika1.m", aby wyraźnie wyróżnić charakter programu oraz ustawić jego numerację. Jeżeli chodzi o lokalizację programu, może być ona dowolna. Matlab domyślnie zawsze pozwala na zapis nowego pliku w katalogu roboczym (zgodnie z ustawieniem opcji *Current Directory* w górnym pasku menu). W czasie zajęć laboratoryjnych w laboratoriach komputerowych Instytutu L-5 proszę zapisywać wszystkie pliki na partycji sieciowej H. Po zapisaniu programu można wybrać opcję z górnego mu "Debug - Run grafika1.m", lub nacisnąć klawisz "F5", lub ikonkę zielonej strzałki  z górnego menu. Jeżeli przy zapisaniu pliku dokonaliśmy zmiany katalogu zapisu (jest inny niż katalog bieżący), pojawi się formularz informacyjny z opcją wyboru o wyglądzie



Matlab informuje nas, że nie można wykonać programu, ponieważ znajduje się on w innej lokalizacji niż katalog roboczy. Aby go uruchomić, należy ustawić katalog roboczy na katalog z

uruchamianym plikiem, czy wybrać pierwszą opcję 

. W dalszym ciągu program powinien uruchomić się bez żadnych problemów, pojawi się znane już nam okienko graficzne z częścią wykresu funkcji sin (w żądanym przedziale) oraz dwie zmienne w przestrzeni roboczej (*Workspace*), o nazwie "x" i "y" - jak widzimy, są to wektory o wymiarach <1x1517>. Stąd wniosek, iż programy w Matlabie pisane w takiej formie, jak powyższy, generują zmienne w tej samej głównej przestrzeni roboczej, która używana jest w czasie pracy w trybie wsadowym w OP. Taki rodzaj programu w Matlabie nazywa się **skryptem** lub **programem skryptowym**. W dalszym ciągu będziemy modyfikować na różne sposoby powyższy prosty skrypt. Nie zamykajmy okna graficznego, natomiast w programie dokonajmy zmiany w linijce 3ciej:

```
plot(x,y,'r-');
```

Dodanie trzeciego argumentu do wywołania funkcji "plot" ustawi parametry rysowania wykresu. Podajemy je pomiędzy znakami "prim", co będzie rozumiane jako element tekstu. Mogą się tam znaleźć: parametr koloru ('r' - czerwony, 'b' - niebieski, 'k' - czarny itd.), parametr znaku ('.' - piksel, 'o' - okrąg, '*' - gwiazdka, 's' - kwadrat, '^' - trójkąt itd. - w powyższym przykładzie znaku nie ma wyspecyfikowanego) oraz styl linii ('-' - linia ciągła, ':' - linia kropkowana itd.). Komplet możliwych parametrów można zobaczyć wpisując do OP polecenie

```
>> help plot
```

Pojawi się opis polecenia oraz możliwe zestawy wywołań parametrów stylu wykresu - ułożone w trzy kolumny. Inne parametry wykresu np. grubość linii oraz kolor wnętrza znaków ustawia się na inne sposoby. Będzie to omówione w dalszej kolejności. Teraz zmodyfikujmy pierwszą

```
x = -3*pi:0.5:2*pi;
```

oraz trzecią linijkę programu

```
plot(x,y,'ks-')
```

co spowoduje: linijka 1: zwiększenie kroku w zakresie generującym wektor "x" - zostanie wygenerowanych mniej punktów pośrednich wykresu; linijka 3: wykres zostanie narysowany kolorem czarnym, a wszystkie punkty pośrednie wykresu zostaną oznaczone czarnym pustym kwadratem. Zauważmy, iż pomiędzy punktami wykresu widać wyraźnie odcinki proste - wykres stracił na gładkości, bo jego punkty są rozłożone bardzo rzadko. Jeżeli Czytelnikowi nasuwa się pytanie, dlaczego używanie funkcji "plot" (łamana) jest bardziej ogólne do rysowania wykresów, niż np. funkcji "fplot" (wykres ciągły), to odpowiedź jest następująca: zarówno funkcja "fplot" jak i "ezplot" omawiane wcześniej, rysują wykresy dokładnie tak samo, jak plot: jako łamaną. Różnica polega na tym, iż przy funkcji "plot" to użytkownik może decydować, jak wiele punktów pośrednich wykorzystuje, podczas gdy dwie pierwsze funkcje dobierają ten podział automatycznie: na podstawie przebiegu funkcji oraz

długości przedziału. Oczywiście kontrola nad tymi parametrami ze strony użytkownika też jest możliwa.
Kolejna uwaga wiąże się z rysowaniem wykresów na tym samym oknie graficznym. Zauważmy, że kolejne uruchamianie programu, np. po kolejnych zmianach, nie powoduje otwarcia nowych okienek graficznych - wszystko pojawia się cały czas w oknie o nazwie "Figure 1". Co więcej, po narysowaniu nowych wykresów, nie widzimy starych, narysowanych poprzednio. Np. po narysowaniu funkcji sin ze skokiem 0.5 nie widzimy już w tle wyniku ze skokiem 0.01. Dzieje się tak dlatego, iż domyślnie okno graficzne usuwa poprzednie obiekty graficzne (punkty, linie itd.) przed narysowaniem następnego. Wprowadźmy następujące zmiany. Dodajmy na końcu programu (nowa, czwarta linijka) polecenie "hold on"

```
hold on
```

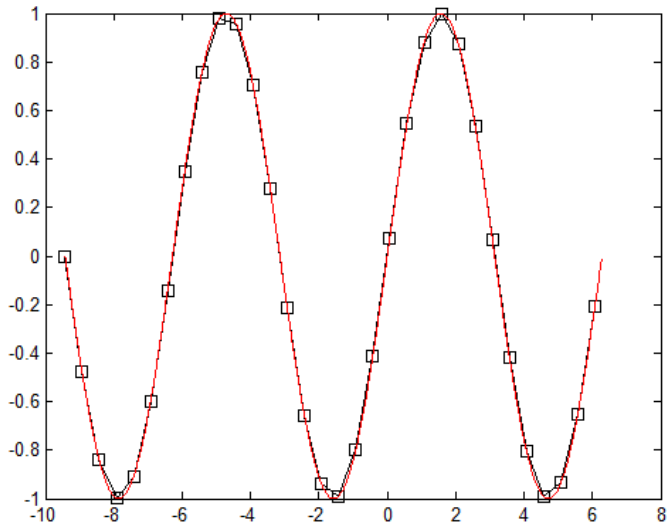
Uruchommy program. Nic się nie powinno - na razie - zmienić. Następnie nie gasząc bieżącego okna graficznego, przywróćmy wartość skoku 0.01

```
x = -3*pi:0.01:2*pi;
```

oraz zmieńmy styl rysowania wykresu

```
plot(x,y,'r-')
```

Proszę ponownie uruchomić program. Okienko graficzne powinno mieć taki wygląd, jak na poniższym rysunku

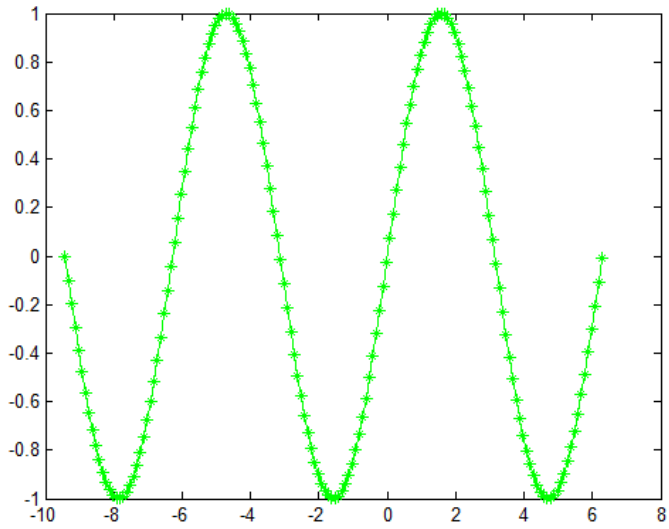


W bieżącym układzie współrzędnych są teraz dwa wykresy: "stary" czarny ze skokiem 0.5 i z oznaczeniami punktowymi oraz "nowy" czerwony ze skokiem 0.01. W większej części wykresy się pokrywają, ale przy ekstremach ("górach" i "dolinach") różnice są wyraźnie widoczne. Tak właśnie działa funkcja "hold" z argumentem "on" - oznacza to, że od tej chwili wykresy stare nie będą kasowane wraz z narysowaniem czegoś nowego. Domyślnie funkcja "hold" działa w formacie "off". Oczywiście po zamknięciu okna graficznego wszystkie

wykresy znikną i mimo polecenia "hold on" nie zostaną one przywrócone przy następnym wykonaniu programu. Zamknijmy okno graficzne i jeszcze raz uruchommy program: na wykresie powinien być już tylko wykres sinus dla skoku 0.01.
Dobrą manierą jest zamykanie okien graficznych po zakończeniu pracy z programem. Jednak nie trzeba tego robić ręcznie - jeżeli kolejne uruchomienia tego samego programu są od siebie niezależne, to można na samym jego początku umieścić polecenie "close all". Funkcja "close" pozamyka za nas wszystkie stare okna graficzne, zanim program zacznie pracować. Wprowadźmy to polecenie na początku programu i ponownie zmieńmy parametry rysowania wykresu. Po tych zmianach program powinien wyglądać następująco:

```
close all
x = -3*pi:0.1:2*pi;
y = sin(x);
plot(x,y,'g*-')
hold on
```

Zmiany nastąpiły w linijce 1 (nowe polecenie), 2 (skok 0.1) oraz 4 (parametr rysowania kolorem zielonym 'g' oraz oznaczenie punktu gwiazdką '*'). Uruchommy program (poprzednie okno graficznie jest nie zamknięte). Program sam je zamknie (funkcja "close", a potem otworzy je raz jeszcze, wraz z wykonaniem funkcji "plot"). Wynik jak na rysunku poniżej



W dalszej kolejności zostaną pokazane różne modyfikacje powyższego programu. Można je wprowadzać na różne sposoby: albo poprzez ciągłą modyfikację pliku "grafika1.m" (np. komentowanie - za pomocą znaku % - zbędnych linijek) albo poprzez tworzenie nowych plików o nazwach np. "grafika2.m", "grafika3.m" itd., albo też stosując podejście "mieszane", czyli uruchamiając nowy plik z chwilą, gdy zmiany staną się wyraźne. Wybór należy do Czytelnika. Pierwszy rozważany program znajduje się poniżej - nowe linijki w stosunku do poprzedniej wersji - zostały wyróżnione większą czcionką.

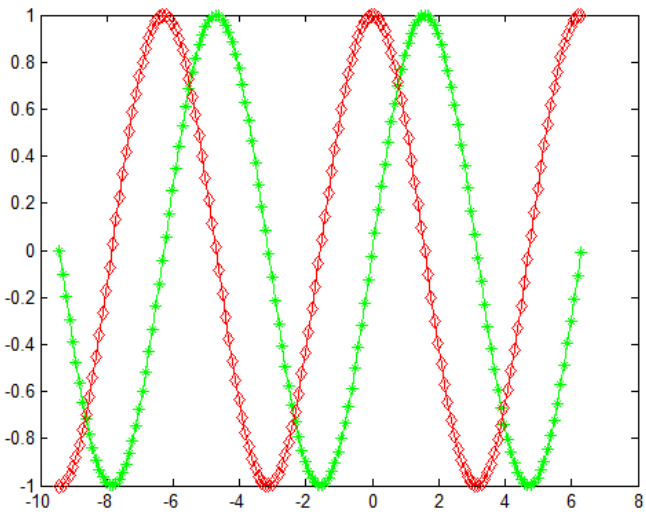
```
close all
```

```
x = -3*pi:0.1:2*pi;
y = sin(x);
plot(x,y,'g*-')
hold on
y = cos(x);
plot(x,y,'rd-')
```

Program różni się od poprzedniego dwiema ostatnimi linijkami. Mianowicie, po narysowaniu funkcji sinus i wstrzymaniu wykresów (funkcja "hold") następuje obliczenie wartości funkcji cosinus w tych samych punktach i jej rysunek - innym kolorem i oznaczeniem punktów. Wynik wg rysunku poniżej.
Wykresy funkcji sinus i cosinus można umieścić na dwóch oddzielnych oknach graficznych (Figure1 i Figure2) w następujący sposób: przed narysowaniem czegokolwiek na jednym z nich należy ustawić bieżące okno za pomocą funkcji "figure". Jej argumentem jest numer okna, na którym mają być umieszczanie poniżej wykonywane polecenia graficzne. Np.

```
close all
x = -3*pi:0.1:2*pi;
y = sin(x);
figure(1)
plot(x,y,'g*-')
hold on
y = cos(x);
figure(2)
plot(x,y,'rd-')
```

W stosunku do poprzedniego programu, powyższy różni się wprowadzeniem funkcji "figure" przed każdym z wywołań funkcji "plot".



Po wykonaniu programu powinny pojawić się dwa okna graficzne (jedno pod drugim), zawierające odpowiednio wykres sinus i cosinus. Proszę je rozsunąć i umieścić obok siebie, aby je porównać.

Jeżeli ustawimy jakąś funkcję graficzną, np. "hold" dla pierwszego z okien, nie obejmie ona swoim wpływem innego okna - dlatego też do każdego z nich należy stosować niezależne ustawienia. Np. uruchomienie programu

```
close all
x = -3*pi:0.1:2*pi;
y = sin(x);
figure(1)
plot(x,y,'g*-')
hold on
grid on
y = cos(x);
figure(2)
plot(x,y,'rd-')
```

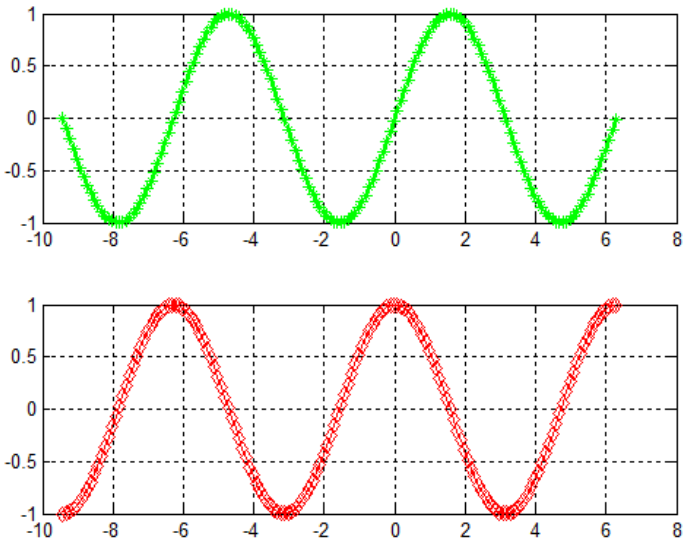
włącza widoczność siatki (funkcja "grid") na pierwszym oknie (Figure1), natomiast drugie okno nie posiada włączonej siatki - należałoby to zrobić niezależnie, tj.

```
close all
x = -3*pi:0.1:2*pi;
y = sin(x);
figure(1)
plot(x,y,'g*-')
hold on
grid on
y = cos(x);
figure(2)
plot(x,y,'rd-')
hold on
grid on
```

Polecenie "hold on" przed "grid on" przy drugim oknie (Figure2) jest konieczne, gdyż włączenie (narysowanie) siatki spowodowałoby skasowanie wykresu sinus (domyślne ustawienie funkcji "hold" w formacie "off").
Innym sposobem wizualizacji wykresów dwóch różnych funkcji jest narysowanie ich w jednym oknie graficznym, ale na dwóch osobnych układach współrzędnych. Do podziału jednego okna graficznego na "podwykresy" służy funkcja "subplot", która przyjmuje trzy argumenty: liczba wykresów w poziomie, liczba wykresów w pionie, oraz numer aktualnego wykresu, na którym mają być umieszczane obiekty graficzne, w kolejności poziomej. Proszę zamienić w powyższym programie linijki z "figure(1)" oraz "figure(2)" na:

```
close all
x = -3*pi:0.1:2*pi;
y = sin(x);
subplot(2,1,1);
plot(x,y,'g*-')
hold on
grid on
y = cos(x);
subplot(2,1,2);
plot(x,y,'rd-')
hold on
grid on
```

Poprawny graficzny rezultat działania tego programu znajduje się na rysunku poniżej. Pierwsze wywołanie funkcji "subplot" przygotowuje podział na dwa wykresy w poziomie i jeden w pionie, oraz ustawia numer aktualnego wykresu na "1" (czyli ten, który znajduje się powyżej). Drugie wywołanie odnosi się do tego samego podziału, ale ustawia aktualny numer wykresu na "2".



Skala wszystkich narysowanych dotąd wykresów (w tym dwóch powyższych) jest dobierana automatycznie przez program. Skala czyli zakres osi „x” oraz „y”. Oczywiście można ją dowolnie modyfikować za pomocą własnych ustawień. Przyjrzyjmy się przykładowo powyższym wykresom. Skala osi „x” jest nieco za duża – widać po lewej i po prawej puste miejsca, gdzie już punkty wykresów nie zostały wygenerowane. Z kolei skala osi „y” mogłaby być większa – punkty wykresów dotykają linii końcowych układów współrzędnych. Do tego typu ustawień służy funkcja „axis”. Można za jej pomocą odczytać bieżące zakresy obydwu osi, lub też ustawić swoje własne. Dla przykładu dopiszmy do powyższego programu jedną linijkę na końcu

```
close all
x = -3*pi:0.1:2*pi;
y = sin(x);
subplot(2,1,1);
plot(x,y,'g*-')
hold on
grid on
y = cos(x);
subplot(2,1,2);
plot(x,y,'rd-')
hold on
grid on
axis
```

Na wykresach nic się nie zmieniło, ale w OP pojawi się rezultat działania funkcji „axis” dla drugiego z wykresów (funkcji cosinus) – wektor czteroelementowy, którego elementy to w kolejności $x_{\min}$ $x_{\max}$ $y_{\min}$ $y_{\max}$. Proszę porównać te wartości z zakresami osi „x” oraz „y” widocznymi na drugim podwykresie. Aby zmienić te zakresy – dla obydwu wykresów, należy ponownie użyć funkcji „axis”, ale tym razem podać jako jej argument wektor czteroelementowy, w którym zapisane będą powyższe cztery wartości, osobno dla każdego z wykresów

```
close all
x = -3*pi:0.1:2*pi;
y = sin(x);
subplot(2,1,1);
plot(x,y,'g*-')
hold on
grid on
axis([-3*pi 2*pi –1.1 1.1])
y = cos(x);
subplot(2,1,2);
plot(x,y,'rd-')
hold on
grid on
axis
axis([-3*pi 2*pi –1.1 1.1])
```

Proszę uruchomić I sprawdzić, czy nastąpiło właściwe przeskalowanie. Funkcja „axis” może też mieć inny format wywołania, np. „axis equal” ustawia równe „przyrosty” zakresów obydwu osi, a „axis square” – równe „długości”. Proszę spróbować dodawać kolejno poniższe formaty jako ostatnie linijki programu, i po dodaniu kolejnej program uruchamiać i sprawdzać, jak się wykres drugi (z cosinus) zachowuje

- Wyrównanie przyrostów

```
axis equal
```

- Wyrównanie długości

```
axis square
```

- Wyłączenie układu współrzędnych

```
axis off
```

- Optymalne ustawienie zakresów osi w odniesieniu do narysowanych obiektów

```
axis tight
```

- Powrót do ustawień domyślnych

```
axis auto
```

Powyższe formaty oraz inne możliwe są do podglądnięcia po wpisaniu do OP polecenia

>> help axis

Wygląd powyższych wykresów oraz parametry okienek graficznych można dodatkowo modyfikować za pomocą różnych ich ustawień. Np. poprzez dodanie funkcji opisujących legendę („legend”), osie („xlabel”, „ylabel”) oraz tytuł wykresu („title”), a także modyfikację pól obiektów graficznych (np. funkcji „plot”). Proszę wprowadzić następujące zmiany w ostatnim programie

```
close all
x = -3*pi:0.1:2*pi;
y = sin(x);
subplot(2,1,1);
plot(x,y,'g*-')
hold on
grid on
axis([-3*pi 2*pi -1.1 1.1])
legend('y=sin(x)');
xlabel('x');
ylabel('y');
title('sinus');
y = cos(x);
subplot(2,1,2);
p = plot(x,y,'rd-')
get(p)
hold on
grid on
axis
axis([-3*pi 2*pi -1.1 1.1])
legend('y=cos(x)');
xlabel('x');
ylabel('y');
title('cosinus');
```

W powyższym programie, oprócz wspomnianych powyżej funkcji “kosmetycznych”, pojawia się także “uwartościowienie” funkcji plot rysującej wykres funkcji sinus (nadanie jej wartości przekazanej do zmiennej “p”). W takim wypadku do zmiennej „p” trafi tzw. adres zmiennej obiektowej (w żargonie informatycznym – „uchwyt”), czyli adres miejsca w pamięci komputera, od którego zaczyna się zapisywanie własności tego obiektu – w tym przypadku zestawu kresek i punktów tworzących wykres funkcji „sinus”. Funkcją „get” możemy wyświetlić w OP listę tych własności, wraz z przypisanymi im aktualnymi wartościami. Po uruchomieniu programu zobaczmy do OP – spośród wielu własności, najciekawsze są na początku – m.in. grubość linii („LineWidth” z wartością 0.5), rozmiar znaku („MarkerSize” z wartością 6) czy też kolor wnętrza znaku („MarkerFaceColor”, z wartością ‘none’, co oznacza puste – niezamalowane wnętrze, jak na bieżącym rysunku. Oczywiście każdą z tych własności można modyfikować – można zrobić to dwojako. Bardziej ogólna metoda wymaga zastosowania funkcji „set” – podajemy w jej wywołaniu trzy argumenty: nazwę uchwytu

obiektu graficznego (u nas to zmienna „p"), nazwę własności (można pisać całość małymi literami, ale musi się ona znaleźć pomiędzy znakami „prim" - ' ') oraz wartość, jaką chcemy danej własności nadać. Drugi sposób – szczególny dla funkcji „plot" to wzbogacenie jej wywołania o kolejne argumenty według schematu: nazwa własności (w „primach") oraz wartość tej własności. Poniższa modyfikacja programu stosuje pierwszy sposób dla funkcji cosinus oraz drugi sposób dla funkcji sinus:

```
close all
x = -3*pi:0.1:2*pi;
y = sin(x);
subplot(2,1,1);
plot(x,y,'g*-','linewidth',3,'markersize',3)
hold on
grid on
axis([-3*pi 2*pi -1.1 1.1])
legend('y=sin(x)');
xlabel('x');
ylabel('y');
title('sinus');
y = cos(x);
subplot(2,1,2);
p = plot(x,y,'rd-')
get(p)
set(p,'linewidth',2);
set(p,'markersize',8);
set(p,'markerfacecolor','y');
hold on
grid on
axis
axis([-3*pi 2*pi -1.1 1.1])
legend('y=cos(x)');
xlabel('x');
ylabel('y');
title('cosinus');
```

Funkcją "set" można ustawić kilka własności na raz, w powyższym przykładzie zostało to podzielone na trzy linijki, aby uniknąć pisania jednego długiego polecenia w dwóch linijkach. Jest to oczywiście możliwe (za pomocą znaku kontynuacji „..."), ale byłoby słabo czytelne.
Zamiast tych trzech linijek można też napisać jedną -

```
set(p,'linewidth',2,'markersize',8,'markerfacecolor','y');
```

Zmiany dla wykresu funkcji sinus są następujące: linia jest grubsza 6 razy w stosunku do grubości pierwotnej, ale znak ('*') mniejszy dwa razy. Z kolei dla funkcji cosinus: linia jest grubsza 4 razy, znak większy o 33% wielkości poprzedniej, a jego wnętrze jest zamalowane na żółto. Proszę program uruchomić.
W podobny sposób ustawienia można zmieniać dla innych funkcji graficznych, np. funkcji „title" czy „ylabel". Bieżący program po wprowadzeniu poniższych zmian zmieni litery na większe w tytule wykresów, oraz zastosuje pogrubioną czcionkę (sinus) lub pochyloną (cosinus). Dodatkowo opis osi y (tekst „y") zostanie obrócony do poziomu. Oczywiście o nazwach powyższych własności dla danego obiektu można się dowiedzieć zwracając uchwyt danego obiektu oraz wyświetlając jego cechy za pomocą funkcji „get"

```
close all
```

```
x = -3*pi:0.1:2*pi;
y = sin(x);
subplot(2,1,1);
plot(x,y,'g*-','linewidth',3,'markersize',3)
hold on
grid on
axis([-3*pi 2*pi -1.1 1.1])
legend('y=sin(x)');
xlabel('x');
ylab = ylabel('y');
set(ylab,'Rotation',0);
t = title('sinus');
set(t,'fontsize',15,'fontweight','bold');
y = cos(x);
subplot(2,1,2);
p = plot(x,y,'rd-')
get(p)
set(p,'linewidth',2);
set(p,'markersize',8);
set(p,'markerfacecolor','y');
hold on
grid on
axis
axis([-3*pi 2*pi -1.1 1.1])
legend('y=cos(x)');
xlabel('x');
ylab = ylabel('y');
get(ylab)
set(ylab,'Rotation',0);
t = title('cosinus');
get(t)
set(t,'fontsize',15,'fontangle','italic');
```

Program powyższy, mimo iż poprawnie napisany i działający, rysuje wykresy funkcji sinus i cosinus tylko w przedziale $[-3\pi \quad 2\pi]$. Aby nadać mu większy stopień ogólności, należałoby wstawić w miejsce początku i końca przedziału dwie zmienne (np. "a" i "b") i cały program zmodyfikować tak, aby zamiast stałych, zafiksowanych końców przedziału, używać wartości określone przez te zmienne. Oczywiście w czasie wykonania programu zmienne "a" i "b" muszą mieć jakieś konkretne wartości, które zazwyczaj podaje użytkownik - w części, która nazywa się preprocesorem programu (czyli przygotowaniem danych). W Matlabie są różne możliwości budowania graficznych interfejsów użytkownika - np. za pomocą funkcji "menu". Na obecnym etapie przedstawione zostanie działanie funkcji "input" pozwalającej na wprowadzenie "interaktywne" w OP - w czasie działania programu - wartości zmiennych z klawiatury przez użytkownika. Zmodyfikowany program poniżej:

```
a = input('Podaj początek przedziału a = ')
b = input('Podaj koniec przedziału b = ')
close all
x = a:0.1:2*b;
y = sin(x);
subplot(2,1,1);
```

```
plot(x,y,'g*-','linewidth',3,'markersize',3)
hold on
grid on
axis([a b -1.1 1.1])
legend('y=sin(x)');
xlabel('x');
ylab = ylabel('y');
set(ylab,'Rotation',0);
t = title('sinus');
set(t,'fontsize',15,'fontweight','bold');
y = cos(x);
subplot(2,1,2);
p = plot(x,y,'rd-')
get(p)
set(p,'linewidth',2);
set(p,'markersize',8);
set(p,'markerfacecolor','y');
hold on
grid on
axis
axis([a b -1.1 1.1])
legend('y=cos(x)');
xlabel('x');
ylab = ylabel('y');
get(ylab)
set(ylab,'Rotation',0);
t = title('cosinus');
get(t)
set(t,'fontsize',15,'fontangle','italic');
```

W powyższym programie miejsce stałych wartości "-3*pi" oraz "2*pi" zajęły odpowiednio zmienne "a" i "b". Za początku programu funkcja "input" wyświetli w OP tekst zachęty podany jako jej argument i wartość wprowadzoną z klawiatury zapisze kolejno do zmiennych "a" i "b". Po wykonaniu programu proszę przejść do OP, gdzie nastąpi zatrzymanie programu i oczekiwanie na wprowadzenie przez użytkownika tych wartości (tzn. wprowadzamy pierwszą - początek przedziału - naciskamy "Enter", wprowadzamy drugą - koniec przedziału - naciskamy "Enter" - pojawia się rysunek, program się kończy). Proszę wykonać program dla kilku wartości "a" i "b", np. -4 i 6; -20 i 40; 0 i 3*pi oraz -12*pi i 7*pi.
Jednakże dalsze uruchamianie tego programu i wszystkich następnych byłoby dosyć uciążliwie, gdy za każdym razem należałoby podawać wartości tych dwóch zmiennych. Zwłaszcza w fazie testowania poprawności działania nowego programu ciągłe wpisywanie na początku jego działania danych z klawiatury jest nieefektywne. Dlatego też w dalszych przykładach interfejs zastąpimy za pomocą kilku pierwszych linijek programu, w których to będą określone konkretne wartości zmiennych stanowiących dane do programu - zastąpią one w ten sposób pytanie się o te wartości użytkownika. Oczywiście program musi być tak napisany w dalszym ciągu, aby używał on zmiennych, a nie konkretnych wartości - wtedy będzie działa poprawnie dla dowolnych wartości, jakie określimy na początku. To jedna z podstawowych zasad programowania: należy nadać programowi jak największą skalę ogólności, tak, aby nie trzeba było "grzebać" w jego kodzie w celu przestawienia go na inne parametry. Poniżej przykład definiowania danych na początku programu:

```
%a = input('Podaj początek przedziału a = ')
%b = input('Podaj koniec przedziału b = ')
```

```
a = -8;
b = 6;
close all
x = a:0.1:2*b;
y = sin(x);
subplot(2,1,1);
plot(x,y,'g*-','linewidth',3,'markersize',3)
hold on
grid on
axis([a b -1.1 1.1])
legend('y=sin(x)');
xlabel('x');
ylab = ylabel('y');
set(ylab,'Rotation',0);
t = title('sinus');
set(t,'fontsize',15,'fontweight','bold');
y = cos(x);
subplot(2,1,2);
p = plot(x,y,'rd-')
get(p)
set(p,'linewidth',2);
set(p,'markersize',8);
set(p,'markerfacecolor','y');
set(p,'linewidth',2,'markersize',8,'markerfacecolor','y');
hold on
grid on
axis
axis([a b -1.1 1.1])
legend('y=cos(x)');
xlabel('x');
ylab = ylabel('y');
get(ylab)
set(ylab,'Rotation',0);
t = title('cosinus');
get(t)
set(t,'fontsize',15,'fontangle','italic');
```

Przy okazji pokazane zostanie, jak wyprowadzać na ekran graficzny lub do OP teksty informujące o wynikach działania programu. Funkcjami, które takie teksty mogą umieszczać w oknie graficznym są np. poznane już funkcje "title" oraz "legend", ale także ogólna funkcja "text". Funkcja, która służy do wyświetlania prostych i złożonych zmiennych tekstowych w OP, nazywa się "disp". Jako argument przyjmuje ciąg znaków ("łańcuch"), na który mogą składać się stałe znakowe (czyli litery, znaki interpunkcyjne, cyfry itd.) oraz wartości liczbowe przekonwertowane ("przerobione") na tekst (np. za pomocą funkcji "num2str"). Proszę dopisać do powyższego programu na samym jego końcu linijkę

```
disp('Ten program rysuje wykresy funkcji sinus i cosinus');
```

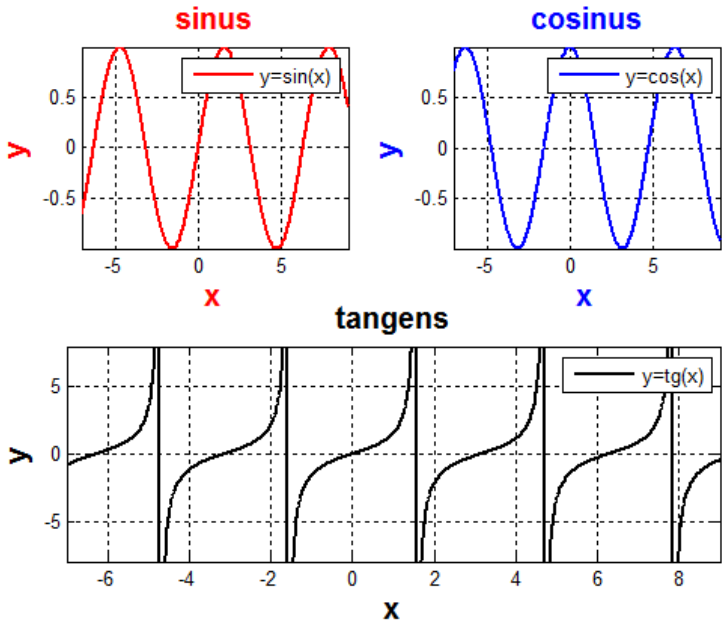
i go wykonać. W OP zobaczymy wyświetlony (niezależnie od zastosowanego znaku średnik na końcu linii) tekst. Jeżeli natomiast chcielibyśmy ten tekst rozszerzyć np. o informację, w jakim przedziale są rysowane te wykresy, należałoby obok stałej części tekstu - pojawiającej się w takiej formie przy każdym wykonaniu programu, wyświetlić także aktualne wartości zmiennych "a" i "b", z jakimi program został uruchomiony. Nie można tego zrobić w takim sposób - `disp('Ten program rysuje wykresy funkcji sinus i cosinus w przedziale a i b');` - bo wtedy pokaże się tekst zawierający litery, a nie wartości liczbowe. Należy zatem

połączyć tekst stały i wartości liczbowe w wektor wierszowy w podobny sposób, jak w przypadku samych wartości liczbowych - oddzielając poszczególne jego wyrazy za pomocą spacji lub przecinka. Proszę ostatnią linijkę programu przerobić w następujący sposób:

```
disp(['Ten program rysuje wykresy funkcji sinus i cosinus w przedziale ['...
    ,num2str(a) ';' num2str(b) ']']);
```

a następnie program uruchomić. Proszę zobaczyć na postać tego, co wypisane zostało w OP. Powyższa linijka wypisuje za pomocą funkcji "disp" tekst (argumentem tej funkcji jest wektor znakowy, z wyrazami ujętymi w znaki "prim" ' ') składający się z trzech elementów stałych (pierwszy, trzeci i piąty), oraz dwóch elementów będących wartościami liczbowymi zmiennych "a" i "b" przekonwertowanymi na tekst za pomocą funkcji "num2str". Trzy kropki, które znajdują się tuż za pierwszym składnikiem tekstu to znak pozwalający na pisanie tego samego polecenia w nowej linii edytora Matlabu.

Ćwiczenie nr 1 do samodzielnego wykonania:
Narysować wykresy (za pomocą funkcji "plot") trzech podstawowych funkcji trygonometrycznych (sinus, cosinus, tangens) w trzech odrębnych układach współrzędnych, ale w jednym oknie graficznym. Wykresy funkcji sinus i cosinus powinny się znaleźć w górnym wierszu, a w dolnym wierszu – sam wykres funkcji tangens - wartości tej funkcji mają być z przedziału $[-8 \quad 8]$. Wykresy funkcji mają być widoczne w przedziale $[a \quad b]$, i mają wynikać z podziału tego przedziału na n równych części. Wykresy mają być rysowane różnymi kolorami, , kreską o grubości = 2 punkty wykresów mają być nieoznakowane. Osie oraz wykresy mają być podpisane (pogrubiona czcionka TimesNewRoman, wielkość czcionki = 15, kolor czcionki jak kolor wykresu). Na wykresie ma być ustawiona legenda oraz siatka.
Poniżej pokazano przykładową postać okienka graficznego - rezultatu działania tego programu dla `a = -7; b = 9; n = 1000;`



Wskazówki do wykonania programu

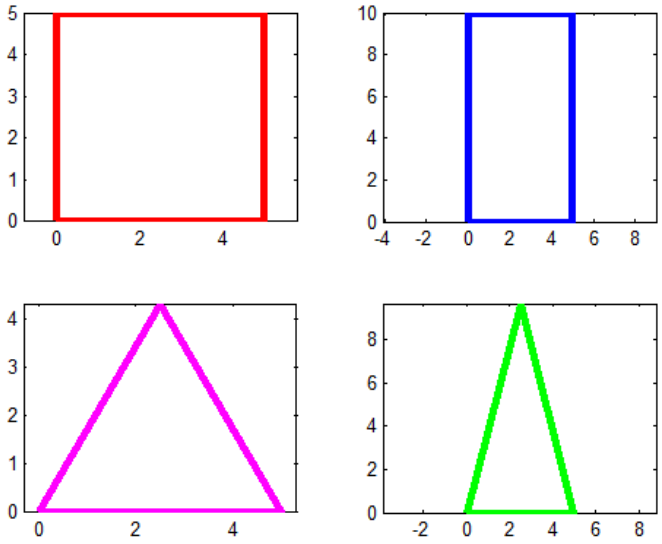
- proszę otworzyć nowy plik edytora Matlabu,
- proszę pozamykać poprzednie wykresy,
- proszę zdefiniować trzy wielkości dane, w postaci zmiennych a, b oraz n,
- proszę obliczyć wartość skoku potrzebnego do generacji wektora argumentów "x" wynikającą z podziału przedziału $[a \quad b]$ na n równych części,
- proszę wygenerować wektor "x" za pomocą operatora zakresu, oraz trzy wektory "y1", "y2" oraz "y3", odpowiadające zestawom wartości trzech funkcji trygonometrycznych, proszę utworzyć pierwszy podwykres do narysowania funkcji sinus, narysować wykres sinus za pomocą funkcji plot, ustawić parametry graficzne wykresu, zatrzymać wykres funkcją "hold" oraz narysować siatkę funkcją "grid", a następnie dopasować osie układu do wykresu za pomocą funkcji "axis", zatytułować wykres, podpisać osie, sformatować teksty tytułu i opis osi do żądanej wielkości, czcionki i koloru, oraz umieścić legendę; formatowanie tekstu tytułu oraz opisów dwóch osi x i y może odbyć się za pomocą jednego wywołania funkcji "set" - wystarczy podać jako pierwszy jej argument podać wektor uchwytów do trzech obiektów graficznych, np. `set([t1 t2 t3], …`
- proszę utworzyć dwa następne podwykresy i powtórzyć czynności wymienione w poprzednim podpunkcie,
- proszę zapisać i uruchomić program, następnie skontrolować poprawność jego wykonania i poprawić ewentualne błędy składniowe w pliku np. kierując się informacją o błędzie i miejscem jego wystąpienia w pliku wyświetlonymi w OP.

Ćwiczenie nr 2 do samodzielnego wykonania:
Narysować za pomocą funkcji "plot" okrąg o wzorze $x^2 + y^2 = r^2$. Wartość promienia r stanowi daną do programu.
Wskazówka: należy wygenerować wektor "x" w odpowiednim przedziale wynikającym z położenia okręgu, a następnie wektor "y" z przekształconego powyższego wzoru. Będzie to zbiór wartości np. górnej części okręgu (nad osią "y"). Przy obliczeniach należy pamiętać o odpowiednim zapisie działań wektorowych (np. notacja tablicowa "z kropką"). Druga część okręgu znajdująca się pod osią "y" powinna być dorysowana za pomocą ujemnych wartości wektora "y". Zakresy osi należy wyrównać za pomocą funkcji "axis", tak, by wykres przedstawiał okrąg, a nie elipsę.

Ćwiczenie nr 3 do samodzielnego wykonania:
Uogólnić program z ćwiczenia nr 2 dla rysowania okręgu o środku w punkcie (x_0, y_0) (wzór: $(x - x_0)^2 + (y - y_0)^2 = r^2$) Dane do programu: x0, y0 oraz r.
Wskazówka: należy stworzyć dwa niezależne zestawy wartości funkcji obrazujących górną i dolną część okręgu, wynikających z odpowiedniego przekształcenia powyższego wzoru.

Ćwiczenie nr 4 do samodzielnego wykonania:
Rozwiązać poprzednie dwa ćwiczenia stosując współrzędne biegunowe $\begin{cases} x = x_0 + r\cos(\alpha) \\ y = y_0 + r\sin(\alpha) \end{cases}$.

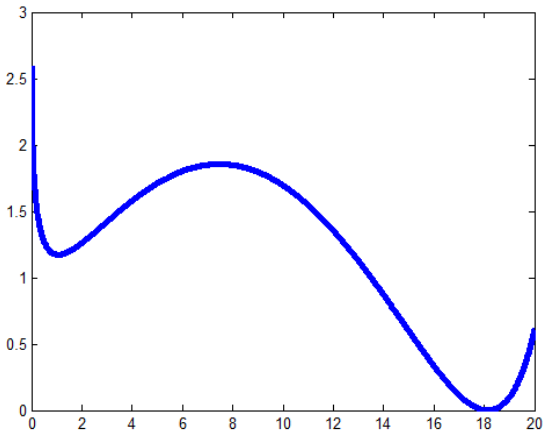
Ćwiczenie nr 5 do samodzielnego wykonania:
Narysować za pomocą funkcji "plot" w czterech podwykresach jednego okienka graficznego kwadrat o boku długości a, prostokąt o długościach boków a i b, trójkąt równoboczny o długości boku "a" oraz trójkąt równoramienny o długościach boków a i b. Dane do programu: a i b.
Wskazówka: figury o konturach zamkniętych należy rysować za pomocą funkcji "plot" podając jako pierwszy argument wektor zawierający współrzędne x-owe wszystkich punktów, a jako drugi argument - wektor zawierający współrzędne y-owe wszystkich punktów. Jako ostatni element tych wektorów należy powtórzyć współrzędne pierwszego z punktów, aby zamknąć figurę - czyli połączyć odcinkiem punkt pierwszy z ostatnim. Np. polecenie `plot([x0 x1 x2 x0],[y0 y1 y2 y0],'r-');` narysuje trójkąt o wierzchołkach w punktach (x0,y0), (x1,y1) oraz (x2,y2). Poniżej na rysunku pokazane jest przykładowe działanie programu dla a = 5 i b =10.



Ćwiczenie nr 6 do samodzielnego wykonania:

Narysować za pomocą funkcji "plot" wykres funkcji uwikłanej o wzorze $\frac{e^{\sqrt{\frac{x}{x+1}}+\sqrt{y}}}{1+\sin\frac{x}{5}}=5$ w przedziale $[0 \quad 20]$.

Poprawny wynik działania programu:



Ćwiczenie nr 7 do samodzielnego wykonania:

Narysować za pomocą funkcji "plot" wykres funkcji danej wzorem $\frac{1}{\sqrt{1-x^2}} = \frac{x+y}{x+7}$.

Wyznaczyć dziedzinę funkcji i na tej podstawie określić przedział, w którym można narysować jej wykres.

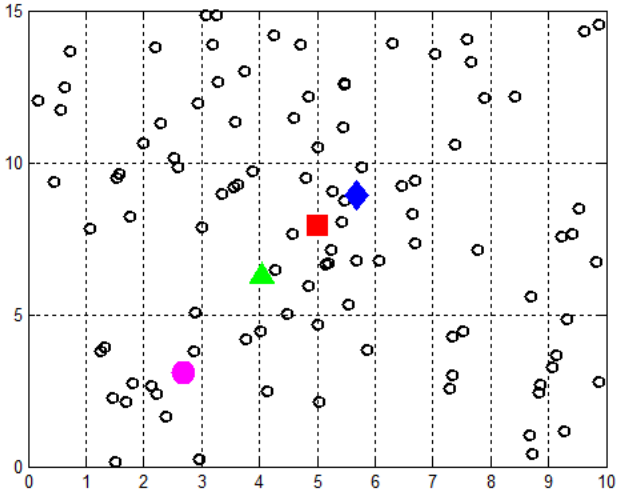
Ćwiczenie nr 8 do samodzielnego wykonania:

Wygenerować losowo n punktów o współrzędnych x z przedziału od 0 do a i współrzędnych y z przedziału od 0 do b. Narysować te punkty, a następnie obliczyć cztery rodzaje średnich współrzędnych punktów tego zbioru:

- średniej arytmetycznej $S_a = \left(\frac{1}{n}\sum_{i=1}^{n} x_i \quad \frac{1}{n}\sum_{i=1}^{n} y_i \right)$
- średniej kwadratowej $S_k = \left(\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2} \quad \sqrt{\frac{1}{n}\sum_{i=1}^{n} y_i^2} \right)$
- średniej geometrycznej $S_g = \left(\sqrt[n]{\prod_{i=1}^{n} x_i} \quad \sqrt[n]{\prod_{i=1}^{n} y_i} \right)$
- średniej harmonicznej $S_h = \left(\frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}} \quad \frac{n}{\sum_{i=1}^{n} \frac{1}{y_i}} \right)$

Narysować punkty o współrzędnych określonych przez te średnie.

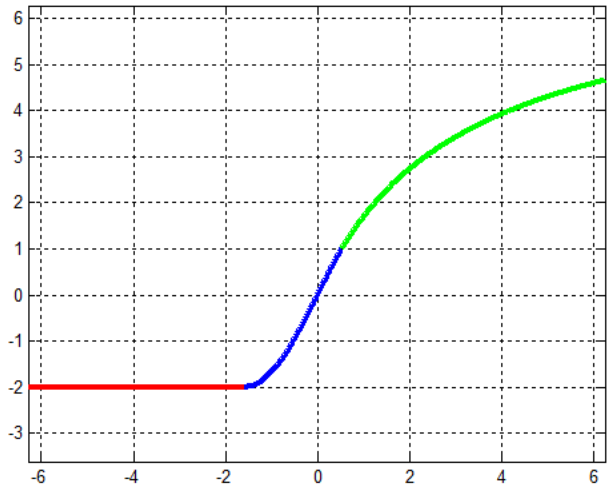
Wskazówka: zastosować wbudowane funkcje Matlaba: "rand" (generacja losowa elementów macierzy), "plot", "sum" (suma elementów wektora), "prod" (iloczyn elementów wektora) oraz "sqrt". Przykładowy wynik działania programu dla a = 10, b = 15 i n = 100 pokazuje poniższy rysunek (czerwony kwadrat - średnia arytmetyczna, niebieski romb - średnia kwadratowa, zielony trójkąt - średnia geometryczna, różowe koło - średnia harmoniczna).



Ćwiczenie nr 9 do samodzielnego wykonania:
Narysować za pomocą funkcji "plot" wykres funkcji danej zależnością

$$y(x)=\begin{cases} -2 \quad , & x \le -\dfrac{\pi}{2} \\ 2\sin(x) \quad , & -\dfrac{\pi}{2} < x \le \dfrac{\pi}{6} \\ \dfrac{7x}{x+\pi} \quad , & x > \dfrac{\pi}{6} \end{cases}$$

w przedziale $\left[-2\pi \quad 2\pi\right]$.
Poprawne działanie programu ilustruje poniższy rysunek.



Oczywiście funkcja "plot" nie jest jedyną dostępną funkcją rysującą w Matlabie. Inne jej odmiany to np.

- funkcja "loglog" która stosuje skalę logarytmiczną (logarytm naturalny) na obydwu osiach układu,
- funkcja "semilogx", która stosuje skalę logarytmiczną na osi x,
- funkcja "semilogy", która stosuje skalę logarytmiczną na osi y,
- funkcja "fill", która wypełnia danym kolorem obszar zamknięty,
- funkcja "area", która wypełnia danym kolorem obszar pod wykresem.

Proszę uruchomić następujący program:

```
close all
subplot(2,1,1);
x = -4:0.01:7;
area(x,sin(x)+cos(x),'facecolor','y');
subplot(2,1,2);
x = [-3 -1 2 3 -4 -3];
y = [-4 -1 -4 5 3 -4];
fill(x,y,'g');
```

Niezwykle ciekawymi funkcjami są

- funkcja "delaunay" dokonująca podziału na trójkąty (triangularyzacja Delaunay) zbioru punktów,
- funkcja "triplot" rysująca trójkąty Delaunay na płaszczyźnie,
- funkcja "voronoi" dokonująca podziału na wielokąty Voronoi zbioru punktów oraz rysująca je na płaszczyźnie.

Pomysły te, zaczerpnięte z kartografii pozwalają na zbudowanie siatki elementów na zbiorze punktów dowolnie rozłożonych - trójkąty Delaunay budowane są pomiędzy punktami, a wielokąty Voronoi - dookoła punktów (przypisywane są punktom). Funkcje te są szczególnie przydatne w algorytmach numerycznych mechaniki obliczeniowej, w których potrzebny jest podział obszaru zadania na proste figury geometryczne (np. w metodzie elementów skończonych lub też w niektórych metodach bezsiatkowych).
Proszę uruchomić następujący program:

```
close all
X = [0 0.5 1 0.5 1 0 1];
Y = [0 0 0 0.5 0.5 1 1];
plot(X,Y,'bo','linewidth',2);
hold on
T = delaunay(X,Y)
triplot(T,X,Y,'y','linewidth',2);
v=voronoi(X,Y,'r-.');
set(v,'linewidth',2);
```

Wektory "X" oraz "Y" zawierają współrzędne 7 punktów, tworzących kwadrat. Funkcja "delaunay" tworzy trójkąty - jej rezultat powinien być widoczny w OP - wyświetlona została lista 6 trójkątów, z numerami węzłów tworzących ich wierzchołki. Funkcja "triplot" rysuje te trójkąty żółtą kreską. Następnie funkcja "voronoi" rysuje wielokąty przypisane węzłom czerwoną linią o stylu "kreska-kropka".

Wektor na płaszczyźnie w Matlabie można narysować za pomocą funkcji "quiver". Jako argumenty podajemy współrzędne wektora (ale nie jego punktów!), domyślnie zaczepiony jest on w punkcie o współrzędnych (1,1). Jeżeli punkt zaczepienia ma być inny, podajemy jego współrzędne jako dwa pierwsze argumenty.
Poniższy przykładowy program rysuje osie układu w prostokącie o bokach a x b.

```
close all
a = 5;
b = 6;
q = quiver(-a/3,0,a,0);
hold on
grid on
set(q,'linewidth',3,'color','k');
q = quiver(0,-b/3,0,b);
set(q,'linewidth',3,'color','k');
```

Grafikę trójwymiarową (3D) tworzy się w Matlabie bardzo podobnie - w analogiczny sposób do grafiki na płaszczyźnie. Aby narysować wykres dowolnej funkcji, należy stworzyć zestaw bardzo gęsto położonych punktów na płaszczyźnie (x,y) - tzw. siatki rysunkowej, a następnie obliczyć w tych punktach wartości danej funkcji. Do generacji tych punktów służy np. funkcja "meshgrid", a do ich rysowania można użyć jednej z wielu funkcji w zależności od rodzaju powierzchni, którą chcemy zobaczyć (wykres przestrzenny, konturowy, siatkowy itd.). Poniżej znajduje się lista najważniejszych funkcji do tworzenia grafiki 3D wraz z opisem ich działania, a także przykładowy program rysujący wykres jednej funkcji na cztery sposoby.

-	"meshgrid"	generowanie siatki rysunkowej
-	"surf"	rysowanie powierzchni
-	"plot3"	rysowanie łamanej 3D
-	"fill3", "patch"	rysowanie i wypełnianie wielokątów
-	"contour", "contour3"	wykres konturowy: płaski i przestrzenny
-	"mesh"	wykres siatkowy
-	"bar3"	rysuje prostopadłościan
-	"sphere", "cylinder"	rysowanie sfery, walca
-	"trimesh", "trisurf"	wykresy siatkowe i powierzchniowe na siatce trójkątów
-	"zlabel"	opis osi "z"
-	"view"	punkt i kąt obserwacji
-	"colorbar"	mapa kolorów
-	"clabel"	ustawienie opisu mapy konturowej

```
close all
[xs,ys] = meshgrid(0:0.05:1,0:0.05:1);
zs = -xs.^3 - ys.^3 + exp( -( (xs-0.5)/0.2).^2 - ( (ys-0.5)/0.2).^2 );
subplot(2,2,1);
s=surf(xs,ys,zs,zs);
set(s,'edgecolor','interp');
subplot(2,2,2);
mesh(xs,ys,zs,zs);
subplot(2,2,3);
contour3(xs,ys,zs,zs);
view(2);
subplot(2,2,4);
```

```
X = [0 1 1 0 0];
Y = [0 0 1 1 0];
h = 3;
plot3(X,Y,[0 0 0 0 0],'b-','linewidth',2);
hold on
plot3(X,Y,[h h h h h],'b-','linewidth',2);
plot3([X(1) X(1)],[Y(1) Y(1)],[0 h],'b-','linewidth',2);
plot3([X(2) X(2)],[Y(2) Y(2)],[0 h],'b-','linewidth',2);
plot3([X(3) X(3)],[Y(3) Y(3)],[0 h],'b-','linewidth',2);
plot3([X(4) X(4)],[Y(4) Y(4)],[0 h],'b-','linewidth',2);
X = X+3;
fill3(X,Y,[0 0 0 0 0],'g');
fill3(X,Y,ones(1,5)*h,'g');
fill3([X(1) X(2) X(2) X(1) X(1)],zeros(1,5),[0 0 h h 0],'g');
fill3([X(1) X(2) X(2) X(1) X(1)],ones(1,5),[0 0 h h 0],'g');
view(3);
axis equal
```

Powyższy program rysuje wykres funkcji danej wzorem $z(x,y) = -x^3 - y^3 + e^{-\left(\frac{x-0.5}{0.2}\right)^2 - \left(\frac{y-0.5}{0.2}\right)^2}$. Poleceniem "meshgrid" generowana jest siatka rysunkowa w kwadracie $\{(x,y), 0 \le x \le 1, 0 \le y \le 1\}$ ze skokiem 0.05. Współrzędne (x,y) siatki są przechowywane w macierzach "xs" i "ys". W tym przypadku wystarczyłoby napisać tylko jeden jej argument - jeżeli zakresy obydwu osi są takie same. W dalszej kolejności obliczana jest macierz "zs" wartości funkcji w punktach siatki. Na kolejnych trzech podwykresach rysowany jest wykres tej samej funkcji na trzy różne sposoby. Pierwszy wykres powierzchniowy na ustawioną własność "edgecolor" (czyli kolor brzegu) na parametr "interp", co oznacza, iż kolor brzegów "blaszek" wykresów dobierany jest na podstawie uśredniania (interpolacji) z wierzchołków. Dodatkowo na czwartym podwykresie narysowane są dwa prostopadłościany o podstawie kwadratu o boku "1" oraz wysokości "h" - za pomocą zbioru krawędzi oraz za pomocą zbioru wypełnionych płaszczyzn z otwartym przelotem.

PROGRAMOWANIE

Naukę **programowania** w Matlabie rozpoczniemy od wprowadzenia pojęcia **funkcji**. Mianowicie do tej pory pisane programy - graficzne z poprzedniego rozdziału - były tzw. **skryptami**, czyli programami, które stanowiły powiązany tematycznie zestaw poleceń. Polecenia te równie dobrze mogłyby być wydawane w Oknie Poleceń (OP) - wszystkie zmienne bowiem programów skryptowych są zmiennymi globalnymi, czyli zapisywane są w tej samej przestrzeni zmiennych (globalnej), w której zmienne tworzone podczas pracy wsadowej. Proszę jeszcze raz uruchomić jakikolwiek program graficzny pisany w poprzednim

rozdziale - po jego zakończeniu w przestrzeni roboczej zostają wszystkie zmienne, które w czasie pracy programu zostały stworzone bądź uległy modyfikacji. W OP można wyświetlić ich wartość oraz posługiwać się nimi w dalszych obliczeniach - prowadzonych w OP lub poprzez uruchomienie innych skryptów. Dla przykładu po wykonaniu ostatniego programu z poprzedniego rozdziału dotyczącego grafiki 3D w przestrzeni roboczej pozostaną zmienne "X", "Y", "h", "xs", "ys" itd. Proszę wyświetlić ich wartości w OP.
Zupełnie inną koncepcję prezentują tzw. **funkcje**, lub też **programy funkcyjne**. Tworzy się je dokładnie tak samo - w tym samym edytorze Matlaba posługując się tymi samymi zestawami poleceń. Podstawowa różnica polega na rozpoczęciu takiego programu nagłówkiem zawierającym słowo kluczowe "function". Jego ogólna składnia jest następująca:

function [wartosc1 wartosc2 wartosc3] = nazwa_funkcji(argument1 argument2)

Patrząc na powyższe można stwierdzić, iż funkcja Matlaba bardzo przypomina funkcję matematyczną, która też składa się z wartości, nazwy i argumentu, a zapis ma podobny. Np. w zapisie $y = \sin(x)$ "y" to wartość, "sin" to nazwa, a "x" to argument. To samo wyrażenie zapisane w języku Matlab wyglądałoby następująco

function [y] = sinus(x)

lub

function y = sinus(x)

Kilka słów komentarza: nazwy "sin" nie powinno się używać dla swojej własnej funkcji, bo tak samo nazywa się już istniejąca w Matlabie funkcja matematyczna. Jeżeli funkcja jest jednowartościowa, nazwę wartości (zmienna "y") nie trzeba pisać w nawiasach kwadratowych. Funkcja Matlaba stanowi uogólnienie funkcji matematycznej, ponieważ może być funkcją wielowartościową i wieloargumentową, ale może też nie mieć żadnych wartości, ani żadnych argumentów. Argumenty i wartości z kolei mogą stanowić zbiór zmiennych różnych typów (np. zmienne liczbowe - rzeczywiste i zmienne znakowe - teksty).
Poniżej znajduje się przykład niezwykle prostego programu funkcyjnego, obliczającego… sumę dwóch liczb i wyświetlającego wynik w OP:

```
function suma_liczb
a = 1;
b = 3;
c = a + b;
disp(['suma liczb wynosi = ' num2str(c)])
```

Przed uruchomieniem programu należy go zapisać. Proszę zwrócić uwagę, że edytor sam podpowiada nazwę pliku, taką samą jak nazwa zdefiniowanej już funkcji, czyli w tym przypadku "suma_liczb" z rozszerzeniem "m". Należy tego nie zmieniać, gdyż program funkcyjny musi nazywać się tak samo, jak w plik na dysku, w którym znajduje się jego kod. Jeżeli piszemy funkcję w pliku już nazwanym, lub zmieniamy istniejący program skryptowy na

funkcję, należy zadbać o to, aby nazwa pliku i nazwa funkcji były identyczne. Program możemy zapisać i uruchomić za pomocą ikonki "Save and run", czyli  .
Po uruchomieniu programu w OP pojawi się oczywisty wypis podający sumę zmiennych "a" i "b", czyli "4" dla aktualnych ich wartości. Jednakże, gdy teraz zechcemy wyświetlić ich wartości w OP pisząc

```
>> a
>> b
```

to albo ukaże się błąd wynikający z nieistnienia tych zmiennych, lub też wyświetlą się zupełnie inne wartości pochodzące z innych skryptów. Dlaczego? Dlatego, iż każda funkcja, z chwilą uruchomienia, dostaje swój własny przydział pamięci zwanej stosem, na którym są zapisywane wszystkie zmienne, które powstaną na jej terenie (tzw. zmienne lokalne). Pamięć ta, czyli stos, jest czyszczona automatycznie z chwilą, gdy funkcja zakończy działanie i nie ma możliwości dostępu do niej i do jej zmiennych z terenu OP, skryptów czy innych funkcji. To najprostsze z możliwych działanie funkcji, która nie ma argumentów i wartości - jedyną więc widzialną zmianą w stosunku do skryptów jest traktowanie zmiennych. Do dalszych celów usuńmy więc wszystkie zmienne globalne

```
>> clear all
```

Jednakże nie tylko do tego służą funkcje. Przede wszystkim pozwalają traktować swój kod na zasadzie "czarnej skrzynki", tzn. podprogramu dokonującego dowolną ilość obliczeń, ale ograniczającego komunikację tekstową i graficzną z użytkownikiem do niezbędnego minimum. Użytkownik na terenie funkcji nie podaje już żadnych danych, ani też funkcja niczego, co stanowi rezultat jej działania, nie wyświetla w OP. Komunikację tę przejmują argumenty i wartości funkcji. Argumenty formalne funkcji (czyli zmienne na liście argumentów, z jakimi projektowania jest funkcja) przyjmą podczas wywołania funkcji konkretne wartości, jakie wprowadzi "z zewnątrz" użytkownik. "Z zewnątrz" oznacza: z innej funkcji, skryptu, lub z OP. Np. w przypadku powyższego programu argumenty formalne to zmienne, które mają za zadanie dostarczyć dane potrzebne do działania programu - czyli zmienne "a" i "b". Z kolei "c" może stanowić wartość - rezultat działania programu - sumę zmiennych - argumentów formalnych. To właśnie wartość wyprowadzana jest na zewnątrz - czyli tam, gdzie funkcja jest wywoływana, gdy posługujemy się jej nazwą. Poniżej zostały pokazane modyfikacje tego samego programu:

- funkcja wartościowa, ale bez argumentów

```
function c = suma_liczb
a = 1;
b = 3;
c = a + b;
%disp(['suma liczb wynosi = ' num2str(c)])
```

Na początku funkcji zdefiniowane są dane. Poza tym funkcja ta jest "niema" - coś liczy, ale nic nie wypisuje. Jednakże jej rezultat (suma a i b, czyli zmienna "c") jest przekazywany na zewnątrz - wewnątrz funkcji przez zmienną "c", na zewnątrz poprzez jej nazwę "suma_liczb". Program taki możemy uruchomić naciskając  lub pisząc w OP nazwę funkcji

```
>> suma_liczb
```

Za każdym razem w OP wyświetli się rezultat działania programu - stanowiący praktyczne wykonanie nagłówka funkcji. Musimy pamiętać, że na zewnątrz nie ma zmiennej "c", tak więc, o ile nie przypiszemy działania funkcji do jakieś zmiennej w OP, jej wartość trafi do "ans". Dlatego też można napisać jawne podstawienie

```
>> wynik = suma_liczb
```

Wtedy do zmiennej globalnej "wynik" trafi rezultat działania funkcji.
Uwaga! Jeżeli funkcja jest wartościowa - w jej wnętrzu **musi** znaleźć się przypisanie do jej wartości czegokolwiek (w domyśle: wyników działania funkcji). W innym przypadku program nie wykona się).

- funkcja argumentowa, ale bez wartości

```
function suma_liczb(a,b)
%a = 1;
%b = 3;
c = a + b;
disp(['suma liczb wynosi = ' num2str(c)])
```

W tym przypadku dane do wykonania programu (zmienne a i b) dostarczane SA jako argumenty funkcji - już nie musimy się przejmować, aby na terenie funkcji je zdefiniować - wręcz nie wolno tego robić, bo wtedy podawane przez użytkownika w czasie wykonywania funkcji argumentu aktualne nie będą do niczego potrzebne. Funkcja znów nie ma wartości - wynik jej działania jest wypisany w postaci tekstu w OP. Jak taką funkcję uruchomić? Jeżeli funkcja ma co najmniej jeden argument i jest on do czegoś używany (a powinien być), to nie można już jej uruchomić za pomocą  , klawisza "F5" czy też napisania w OP polecenia

```
>> suma_liczb
```

Dlaczego? Wyobraźmy sobie sytuację, iż ktoś każe nam obliczyć sinus… O co wtedy zapytamy? Z jakiej liczby ten sinus obliczyć. Bo przecież sinus to funkcja matematyczna jedno argumentowa, i jeżeli chcemy obliczyć jej wartość, to musimy znać argument. Dokładnie o to samo upomina się Matlab wyświetlając na czerwono w OP komunikat o błędzie przy próbach standardowego uruchomienia programu. Jak więc uruchomić go poprawnie? Jedyna możliwość to napisanie w OP

```
>> suma_liczb(4,5)
```

Wtedy funkcja rozpocznie działanie - do zmiennych formalnych "a" i "b" na jej terenie zostaną podstawione wartości "4" i "5". Oczywiście w OP można zdefiniować sobie wcześniej jakieś zmienne i posługując się nimi uruchomić funkcję

```
>> liczba1 = 10;
>> liczba2 = -14;
>> suma_liczb(liczba1,liczba2)
```

Jak widać po powyższym przykładzie, zmienne globalne do wywoływania funkcji nie muszą nazywać się tak samo, jak jej zmienne lokalne - ale oczywiście mogą, co pokaże kolejny przykład

```
>> a = -13;
>> b = 16;
>> suma_liczb(a,b)
```

Zmienne "a" i "b" będą istniały dalej w przestrzeni roboczej

```
>> a
>> b
```

lecz są to zmienne zdefiniowane jako globalne przed działaniem funkcji. Dlaczego funkcja ich nie wyczyściła? Funkcja wyczyściła "swoje" zmienne, które też nazywały się "a" i "b", ale poza nazwą nie miały nic wspólnego z zmiennymi globalnymi "a" i "b". Podobnie jest z ludźmi: w naszym kraju żyje wiele Janów Kowalskich, a odróżnia ich od siebie z urzędniczego punktu widzenia np. adres zamieszkania. I właśnie adres jest tym, co również odróżnia od siebie zmienne, np. tak samo nazywające. Adres, czyli miejsce w pamięci, w którym są one zapisywane. A jak pamiętamy, zmienne lokalne i globalne są zapisywane w zupełnie innych miejscach.

- funkcja argumentowa i wartościowa

```
function c = suma_liczb(a,b)
%a = 1;
%b = 3;
c = a + b;
%disp(['suma liczb wynosi = ' num2str(c)])
```

Odrzucając komentarze można stwierdzić, iż funkcja składa się z nagłówka i jednej linijki - i faktycznie, dane do programu stanowią argumenty określane na zewnątrz, a rezultat jest zwracany do zmiennej "c" (wewnątrz funkcji) poprzez jej nazwę (zewnętrzna powłoka wywołania funkcji, np. OP). Przykładowe prawidłowe wywołania tej funkcji w OP:

```
>> suma_liczb(-6,8)
>> suma_liczb(a,b)
```

```
>> suma = suma_liczb(liczba1, -45);
>> suma
```

W trzecim wywołaniu funkcji na końcu linijki znajduje się średnik, a więc rezultat działania polecenia nie będzie wyświetlony. Jednakże wartość została wpisana do zmiennej globalnej "suma", którą możemy się dalej posługiwać według uznania.
Zarówno w skryptach, jak i funkcjach mogą występować typowe instrukcje programistyczne: **instrukcje pętli** (o określonej i nieokreślonej liczbie przebiegów), **instrukcje warunkowe** (pozwalające na rozgałęzienie programu) oraz **instrukcja wyboru**.
Instrukcja pętli o **określonej liczbie przebiegów** (pętla "for") wygląda następująco: składa się ona z nagłówka, zakresu pętli (czyli zestawu operacji, które będą w niej wykonywane) oraz słówka kończącego jej działanie "end":

```
for parametr_sterujacy = wart_poczatkowa:skok:wart_koncowa
        ...
        …
        …
end
```

Zmienna "parametr_sterujacy" służy do sterowania liczbą przebiegów pętli - to zwyczajna zmienna (choć często o specyficznej nazwie typu "i", "j", "k" itd.), której nadajemy wartość początkową, definiujemy skok (czyli o ile będzie się ona zmieniała wraz z obrotem pętli), oraz definiujemy wartość końcową. Poniższy program stanowi przykład pętli wypisującej 10 razy ten sam tekst w OP. Proszę umieścić kod tego programu w pliku skryptowym edytora Matlaba:

```
clc
for i=1:10
    disp('pętla się kręci...')
end
```

i uruchomić zapisując program pod dowolną (ale poprawną!) nazwą. Co program robi? Polecenie "clc" czyści zawartość OP. Następnie program 10 razy wypisuje ten sam tekst w OP - 10 razy, choć tekst został fizycznie tylko raz napisany w kodzie programu - to włożenie go do odpowiedniej pętli zapewniło jego "rozmnożenie". Przyjrzyjmy się nagłówkowi pętli. Gdyby nie było słowa kluczowego "for" zapis "i=1:10" wygenerowałby wektor wierszowy o nazwie "i" złożony z 10 liczb od 1 do 10 ([1 2 3 4 5 6 7 8 9 10]). Jednakże poprzez dodanie na początku słowa "for", zmienna "i" będzie skalarem (liczbą) zmieniającą się od 1 do 10 co 1 po każdym obrocie pętli. Najpierw pętla się wykonuje (czyli wykonują się polecenia między linijką z "for", a "end") dla "i" = 1, potem "i" zostaje zwiększone (automatycznie!) o "1", i wynosi 2 dla drugiego obiegu pętli, itd., aż dla 10tego obiegu będzie miało wartość "10". Po obiegu 10tym nie zostanie już zwiększone o "1" do "11", bo tym samym przekroczyłoby wartość końcową "10". Spowoduje to zatrzymanie działania pętli. Możemy wyświetlić wartość "i" z OP:

```
>> i
```

Posiada ono wartość końcową "10", po której pętla się zatrzymała. Zróbmy małą modyfikację:

```
clc
for i=1:2:10
    disp('pętla się kręci...')
end
```

Program wypisze tekst tylko 5 razy, bo skok (zwiększanie "i" po każdym obrocie pętli) wynosi teraz "2".
Tak, jak przy generowaniu wektorów za pomocą zakresu (':"), wartość końcowa parametru sterującego pętlą może być mniejsza niż wartość początkowa - ale wtedy skok musi być ujemny, inaczej pętla nie wykona się ani razu:

```
clc
for i=10:-1:1
    disp('pętla się kręci...')
end
```

Parametr sterujący pętlą służy nie tylko do numeracji liczby jej przebiegów. Poprawne i sensowne działanie pętli zapewnione jest wtedy, gdy instrukcje w niej zawarte wykorzystują w jakiś określony sposób wartość jej parametru. Np. do pracy z elementami wektorów lub macierzy pętle "for" są wręcz stworzone - wtedy parametr sterujący "przechodzi" przez wszystkie wiersze i kolumny macierzy i zapewnia tym samym dostęp do pojedynczych je elementów.
Poniższy program pokazuje zastosowanie parametru sterującego w celu wyświetlenia jego wartości w OP podczas kolejnych obiegów pętli:

```
clc
for i=1:10
    disp(['pętla się kręci, i jest równe ' num2str(i)])
end
```

Tak naprawdę to w Matlabie powinno się unikać programowania z wykorzystaniem pętli "for". Dlaczego? Dlatego, iż Matlab jest językiem zorientowanym na obliczenia macierzowe i tablicowe, które pozwalają na wykonywanie działań na elementach macierzy bez uruchamiania pętli - jedynie przy użyciu samych nazw macierzy. Wykonywane są one w takim przypadku najszybciej. Zapisanie tych samych działań za pomocą pętli znacznie opóźni działanie programu. Ale oczywiście są mniej lub bardziej nietypowe sytuacje, gdy użycie i działanie pętli jest niezbędne. W obecnym opracowaniu pierwsze ćwiczenia z pętli "for" będą dotyczyły oczywiście prostych zagadnień, które można by zrealizować bez jej użycia. Jednakże w celu lepszego jej opanowania, właśnie takie przykłady będą rozważane, a alternatywny zapis Matlaba (macierzowy i tablicowy, lub przy wykorzystaniu jego funkcji wbudowanych) pozwoli na kontrolę rezultatów.
Jednymi z najczęstszych zagadnień, które są organizowane w pętlach, są sumowania wszystkich bądź wybranych elementów macierzy/wektora. Dla przykładu wróćmy do jednego z zadań z poprzedniej części skryptu dotyczącej operacji graficznych - do ćwiczenia nr 8. Zadanie wiązało się z obliczeniem kilku średnich współrzędnych zbioru punktów

generowanych losowo - wszystkie wymienione średnie należało policzyć za pomocą funkcji wbudowanych Matlaba. Teraz postaramy się policzyć je za pomocą instrukcji pętli i odpowiednich funkcji matematycznych.
Zadanie jest następujące: dany jest wektor "X" złożony z "n" liczb rzeczywistych dodatnich - wynikiem działania funkcji powinny być cztery rodzaje średnich elementów tego wektora. Poniżej podano odpowiednie wzory dla: średniej arytmetycznej $S_a = \frac{1}{n}\sum_{i=1}^{n} x_i$, średniej kwadratowej $S_k = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}$, średniej geometrycznej $S_g = \sqrt[n]{\prod_{i=1}^{n} x_i}$ oraz średniej harmonicznej $S_h = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$.

Ponieważ program ma być funkcją, należy zacząć od stworzenia jego nagłówka. Powinna to być funkcja czterowartościowa (cztery różne średnie), oraz jedno lub dwuwartościowa. Skąd ta rozbieżność? Mianowicie jako argument można przesłać sam wektor "X", lub też dodatkowo jego rozmiar "n". Jednakże ponieważ w Matlabie łatwo możemy sprawdzić, jaki jest jego rozmiar na terenie funkcji (np. funkcjami "size" lub "length"), nie ma potrzeby kłopotać tym użytkownika. Nagłówek może wyglądać następująco:

```
function [Sa,Sk,Sg,Sh] = srednie(X)
```

Następnie należy obliczyć "n" - liczbę elementów wektora "X", przesłanego przez argument funkcji:

```
n = length(X);
```

Kolejnym krokiem musi być obliczenie trzech sum i jednego iloczynu, potrzebnych do obliczenia średnich - patrz wzory. W średniej arytmetycznej występuje suma elementów, w średniej kwadratowej - suma kwadratów elementów, w średniej geometrycznej - iloczyn elementów, a w średniej harmonicznej - suma odwrotności elementów. Ponieważ suma elementów to "n" (czyli w zasadzie dowolna liczba naturalna), to sumowania i "iloczynowanie" muszą odbyć się w pętli - nie da się bowiem zapisać sumy i iloczynu "n" liczb w jednej linijce (oczywiście nie używając funkcji "sum" czy "prod"). Dlatego też przed otwarciem pętli należy przygotować wartości startowe dla trzech sum i jednego iloczynu. Jakie to wartości? Elementy neutralne dla tego typu działań - od których może pętla zaczynać obliczenia. Dla sumy to wartość "0", dla iloczynu - wartość "1":

```
Sa = 0;
Sk = 0;
Sg = 1;
Sh = 0;
```

Następnie należy uruchomić pętlę. Ale "pętlę" czy "pętle"? Skoro należy policzyć trzy sumy i jeden iloczyn, to może trzeba stworzyć cztery niezależne pętle? Oczywiście można tak zrobić,

ale nie ma takiej potrzeby. Proszę zwrócić uwagę, iż wszystkie trzy sumy i iloczyn w powyższych wzorach mają ten sam przebieg - tzn. zaczynają się dla pierwszego elementu wektora (czyli dla "i = 1"), "skaczą" co "1", a kończą się na elemencie ostatnim (czyli "i=n"). Dlatego też obliczenia można zorganizować w jednej pętli "for":

```
for i=1:n
    Sa = Sa + X(i);
    Sk = Sk + X(i)^2;
    Sg = Sg*X(i);
    Sh = Sh + 1/X(i);
end
```

Wraz z otwarciem pętli - czyli napisaniem jej nagłówka, edytor sam zaproponował wcięcie - to dobra maniera, widać wtedy bardzo wyraźnie, co jest pod zakresem pętli (między "for", a "end"). Najpierw przypatrzmy się tworzeniu sum - przy każdym obrocie pętli wartość sumy "Sa" jest zwiększana o kolejny i-ty element wektora "X". Działa to podstawienie na zasadzie: do "starej" wartości sumy "Sa" dodajemy kolejny nowy element wektora i przypisujemy to działanie do tej samej zmiennej "Sa" aktualizując w ten sposób jej wartość. Np. gdybyśmy liczyli sumę arytmetyczną dla wektora X = [3 4 6 7], to pętla przebiegłaby 4 razy (bo X ma cztery elementy, czyli "n=4") i przy pierwszym obiegu pętli (dla "i=1") do zmiennej "Sa" o wartości startowej "0" podstawione zostanie "Sa = Sa + X(1)", czyli "Sa = 0 + 3", czyli "Sa = 3" po pierwszym kroku. Kolejny obieg pętli dla "i=2" dokona podstawienia "Sa = Sa + X(2)", czyli "Sa = 3 + 4", czyli "Sa = 7". Kolejny obieg ("i=3") podstawi "Sa = Sa + X(3)", czyli "Sa = 7 + 6", czyli " Sa = 13". Ostatni obieg dla "i=4" podstawi " Sa = Sa + X(4)", czyli "Sa = 13 + 7", czyli "Sa = 20" i będzie to wartość końcowa, jaką osiągnie "Sa" po zakończeniu działania pętli. Podobnie zadziała zwiększanie sum "Sk" oraz "Sh", z tą różnicą, że dla "Sk" dodawane będą kwadraty X(i)^2, a dla "Sh", odwrotności 1/X(i). W przypadku iloczynu "Sg" iloczyn będzie zwiększany X(i) razy, co każdy obrót pętli. Poniższa tabelka pokazuje działanie poszczególnych obrotów pętli dla danego wektora [3 4 6 7] - czyli wartości czterech zmiennych "Sa", "Sk", "Sg" oraz "Sh" po każdym obrocie pętli:

i	X(i)	X(i)^2	1/X(i)	Sa	Sk	Sg	Sh
przed pętlą	-	-	-	0	0	1	0
1	3	9	0.3333	3	9	3	0.3333
2	4	16	0.25	7	25	12	0.5833
3	6	36	0.1667	13	61	72	0.7500
4	7	49	0.1429	20	110	504	0.8929

Wartości tych czterech zmiennych "Sa", "Sk", "Sg" oraz "Sh" po zakończeniu działania pętli "for" to oczywiście jeszcze nie ostateczne wartości średnich, tylko trzy sumy i jeden iloczyn. Należy jeszcze przeprowadzić na nich ostateczne operacje wg wzorów:

```
Sa = Sa/n;
Sk = sqrt(Sk/n);
Sg = Sg^(1/n);
Sh = n/Sh;
```

co kończy działanie programu. I tak w przypadku podanego wektora, końcowe wartości powinny wynosić:

- Sa = Sa/n, czyli Sa = 20/4 = 5;
- Sk = sqrt(Sk/n), czyli Sk = sqrt(110/4) = 5.2440;
- Sg = Sg^(1/n), czyli Sg = (504)^(1/4) = 4.7381;
- Sh = n/Sh, czyli Sh = 4/0.8929 = 4.4800.

Proszę uruchomić program, który w całości wygląda następująco:

```
function [Sa,Sk,Sg,Sh] = srednie(X)
n = length(X);
Sa = 0;
Sk = 0;
Sg = 1;
Sh = 0;

for i=1:n
    Sa = Sa + X(i);
    Sk = Sk + X(i)^2;
    Sg = Sg*X(i);
    Sh = Sh + 1/X(i);
end

Sa = Sa/n;
Sk = sqrt(Sk/n);
Sg = Sg^(1/n);
Sh = n/Sh;
```

Najpierw należy go zapisać, oczywiście pod zaproponowaną przez edytor nazwą "srednie.m". Ponieważ funkcja ma argumenty, nie należy programu uruchamiać spod edytora, tylko spod OP, pisząc jego nazwę i podając argument aktualny:

```
>> srednie([3 4 6 7])
```

Jaki jest efekt? Ukazało się rozwiązanie, podstawione do "ans", ale tylko jedno… ("5"). Dlaczego skoro funkcja jest czterowartościowa? Dlatego, iż przez taki rodzaj zapisu wywołania funkcji (czyli bez podstawienia do zmiennej lub zmiennych), wyświetlana jest tylko jedna wartość - pierwsza. Aby zobaczyć w OP wszystkie, należy napisać pełne wywołanie, zgodne z definicją (nagłówkiem) funkcji:

```
>> [s1,s2,s3,s4] = srednie([3 4 6 7])
```

W powyższym wywołaniu funkcji program Matlab tworzy cztery zmienne globalne ("s1", "s2", "s3", "s4"), do których zapisuje cztery wartości wynikające z działania funkcji. Ponieważ linijka nie jest zakończona średnikiem, rezultat (wartości zmiennych globalnych) jest wypisany w OP.
Aby lepiej pamiętać, o tym co robi funkcja i jakie ma wywołanie (składnię), dobrze jest umieszczać "ku pamięci" krótki wpis (komentarz) na początku pliku pod nagłówkiem funkcji:

```
function [Sa,Sk,Sg,Sh] = srednie(X)
```

```
%Funkcja oblicza cztery średnie: arytmetyczną,
%kwadratową, geometryczną
%oraz harmoniczną elementów wektora X.
%Składnia:
%   [Sa,Sk,Sg,Sh] = srednie(X)
%   wersja: 12.11.2011
n = length(X);
Sa = 0;
Sk = 0;
Sg = 1;
Sh = 0;

for i=1:n
    Sa = Sa + X(i);
    Sk = Sk + X(i)^2;
    Sg = Sg*X(i);
    Sh = Sh + 1/X(i);
end

Sa = Sa/n;
Sk = sqrt(Sk/n);
Sg = Sg^(1/n);
Sh = n/Sh;
```

Pamiętajmy, że po tej zmianie należy program **zapisać**! Jeżeli tego nie zrobimy, wykona się jego poprzednia wersja. Edytor informuje nas o niezapisanych zmianach na dysku w edytowanym pliku umieszczając gwiazdkę ("*") po nazwie pliku w górnym pasku edytora. Zapisać można klikając na ikonkę dyskietki () lub… ikonkę uruchomienia programu (). Ta ostatnia wprawdzie funkcji nie uruchomi (pojawi się komunikat o błędzie w OP), ale dokona zapisu programu. Po zapisaniu pliku program będzie działał tak samo, tylko że możliwe będzie uzyskanie pomocy o jego działaniu i składni w OP poprzez polecenie:

```
>> help srednie
```

Istnieje jeszcze jedna możliwość zwracania wartości przez nazwę funkcji. Mianowicie zamiast pisać funkcję czterowartościową, można stworzyć funkcję jednowartościową, której wartością będzie… wektor czteroelementowy (ale traktowany jako jedna zmienna). Naszym zadaniem będzie pod koniec działania funkcji zebrać wszystkie rezultaty jej działania (czyli cztery średnie) do tego wektora. Przykład: proszę otworzyć nowy plik, wkleić do niego zawartość starej funkcji "srednie", zmienić jej nazwę np. na "srednie_wektor" i dokonać innych zmian wskazanych poniżej:

```
function Srednie = srednie_wektor(X)
%Funkcja oblicza cztery średnie: arytmetyczną, kwadratową, geometryczną
%oraz harmoniczną elementów wektora X.
%Składnia:
%  srednie = srednie_wektor(X)
%   srednie - wektor [1x4]
%   X - wektor [nx1] lub [1xn]
```

```
%   wersja: 12.11.2011
n = length(X);
Sa = 0;
Sk = 0;
Sg = 1;
Sh = 0;

for i=1:n
    Sa = Sa + X(i);
    Sk = Sk + X(i)^2;
    Sg = Sg*X(i);
    Sh = Sh + 1/X(i);
end

Sa = Sa/n;
Sk = sqrt(Sk/n);
Sg = Sg^(1/n);
Sh = n/Sh;

Srednie = [Sa,Sk,Sg,Sh];
```

Po dokonaniu obliczeń program zapisze wszystkie cztery zmienne (średnie) do jednego wektora "Srednie" i ten wektor właśnie zostanie zwrócony przez nazwę funkcji na zewnątrz do zmiennej globalnej w OP. Po zapisaniu programu (pod nazwą "srednie_wektor.m") i wywołaniu funkcji w OP otrzymamy rezultat

```
>> srednie_wektor([3 4 6 7])
```

Spróbujmy jeszcze obliczyć średnie korzystając z obydwu funkcji dla większego wektora złożonego z 12 dodatnich liczb całkowitych z przedziału od 1 do 10:

```
>> x = round(rand(12,1)*9)+1
>> [s1,s2,s3,s4] = srednie(x)
>> s = srednie_wektor(x)
```

Ćwiczenie nr 1 do samodzielnego wykonania
Napisać funkcję "suma_iloczyn", która dla danego wektora "X" obliczy sumę i iloczyn jego elementów. Argumenty funkcji: "X", wartości funkcji: "suma", "iloczyn" lub jeden wektor zbierający te dwie wartości. Program proszę napisać z wykorzystaniem pętli "for".

Ćwiczenie nr 2 do samodzielnego wykonania
Napisać funkcję "wariancja", która dla danego wektora "X" obliczy wariancję jego elementów. Wariancja zdefiniowana jest w sposób następujący: $w = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})$, gdzie $\overline{x}$ oznacza wartość średnią elementów wektora "X". Dane do programu: X. Program proszę napisać z wykorzystaniem pętli "for".

Ćwiczenie nr 3 do samodzielnego wykonania

Napisać funkcję obliczającą sumę n wyrazów szeregu $S = \sum_{i=0}^{n} \frac{1}{1+i}$. Dane do programu: n. Program proszę napisać z wykorzystaniem pętli "for".

Ćwiczenie nr 4 do samodzielnego wykonania
Napisać funkcję obliczającą iloczyn skalarny dwóch wektorów X i Y o tej samej liczbie elementów. Dane do programu: X, Y. Program proszę napisać z wykorzystaniem pętli "for".

W dalszym ciągu wykorzystamy pętlę "for" do pracy z macierzami - jako iż mają one dwa wymiary - wierszowy i kolumnowy, niezbędne stanie się wykorzystanie dwóch pętli "for" - jedna w drugiej - czyli tzw. **pętli zagnieżdżonej**. Od razu przeanalizujmy przykład: program dokonujący mnożenia macierzy przez wektor. Jak wiemy, jest to możliwe wtedy, gdy liczba kolumn macierzy jest równa liczbie elementów wektora. Dane do programu stanowią: macierz "A" i wektor "b", wynikiem powinien być wektor "c", stanowiący iloczyn "A" i "b". Poniżej przypomniano wzór na obliczanie elementów wektora "c":

$$c_i = \sum_{j=1}^{m} A_{i,j} b_j \quad , \quad i = 1,\ldots,n$$

Poniżej odpowiedni program funkcyjny realizujący to zadanie:

```
function c = mnozenie_macierz_wektor(A,b)
[n m] = size(A);

c = zeros(n,1);
for i=1:n
    s = 0;
    for j=1:m
        s = s + A(i,j)*b(j);
    end
    c(i) = s;
end
```

Pierwszą instrukcją programu (pod nagłówkiem) jest podstawienie do zmiennych "n" i "m" wymiarów macierzy "A" za pomocą funkcji "size". Sprawdzenie wymiarów wektora "b" nie będzie w tej chwili dokonywane, gdyż zakładamy, iż ma on "m" elementów. Jednakże sprawdzenie tego faktu oczywiście jest potrzebne, by program nie zakończył działania błędem, gdy jego rozmiar będzie mniejszy niż "m". Ale takie zabezpieczenie wymagałoby znajomości instrukcji warunkowej, co będzie tematem następnego podrozdziału. Zanim uruchomimy pętlę "for", należy przygotować wektor "c", zerując go w właściwych wymiarach - a z teorii mnożenia macierzy przez wektor wiadomo, iż będzie to wektor o "n" elementach. Po co to przygotowanie? W pętli "for" kolejne elementy wektora - patrz wzór - będą tworzone jeden po drugim. Najpierw c(1), potem c(2), itd. Program nie wiedząc z góry, jakie wymiary ma wektor, będzie zwiększał jego rozmiary z każdym obiegiem pętli - nie jest to efektywne, ze względu na ciągłą potrzebę lokowania w pamięci nowego wektora. Dlatego też

warto "powiedzieć" programowi, jakie wymiary będzie miał wektor, generując go np. z samych zer, a potem tylko wypełniać jego elementy.
Pierwszą pętlą "for" (zewnętrzną) jest pętla po "i" - zgodnie ze wzorem. Pętla ta służy obliczeniu i-tego elementu wektora "c". Składa się na niego suma składników będących iloczynami elementu macierzy "A" przez element wektora "b". Ponieważ suma zależy od "m", należy zaprogramować ją w kolejnej pętli "for". Najpierw jednak zerujemy jej wartość ("s=0"), potem uruchamiamy drugą pętlę wewnętrzną (po "j") i w tej pętli zwiększamy sumę ("s") o kolejny składnik, zgodnie z wzorem. Gotową sumę wpisujemy do i-tego elementu wektora "c". Pamiętajmy, że pod daną pętlą powinno się znaleźć tylko to, co wyraźnie zależy od jej parametru sterującego. Ale np. zerowanie zmiennej "s" musi się odbywać dla każdego i-tego elementu wektora "c", dlatego też linijka "s=0" musiała się znaleźć wewnątrz pętli po "i". Zapiszmy i uruchommy program. W tym celu stwórzmy w OP macierz "A" i wektor "b" złożone z liczb losowych całkowitych.

```
>> A = round(rand(5,3)*10 - 5)
>> b = round(rand(3,1)*10 - 5)
>> mnozenie_macierz_wektor(A,b)
```

a następnie sprawdźmy poprawność jego działania wykonując "matlabowskie" mnożenie "A" przez "b":

```
>> A*b
```

Wyniki oczywiście powinny być identyczne.

Ćwiczenie nr 5 do samodzielnego wykonania
Zmodyfikować powyższy program tak, aby dokonywał mnożenia macierzy "A" przez macierz "B". Mnożenie powinno odbywać się według zależności:

$$C_{i,j} = \sum_{k=1}^{p} A_{i,k} B_{k,j} \quad , \quad \begin{matrix} i = 1,...,n \\ j = 1,...,m \end{matrix}$$

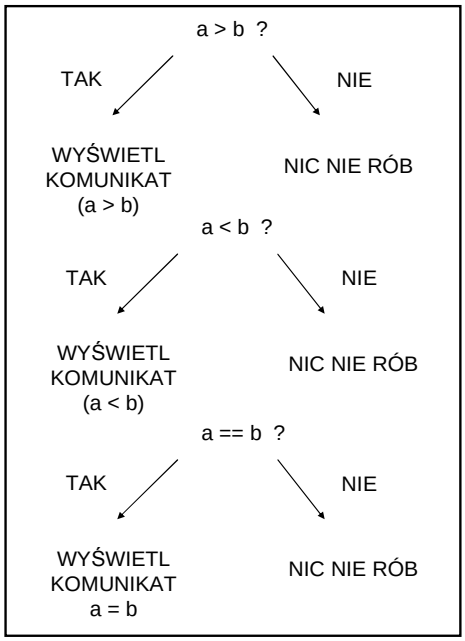
Macierz "A" ma wymiary [n x p], macierz "B" ma wymiary [p x m]. Program napisać z wykorzystaniem pętli "for". Wyniki sprawdzić dokonując mnożenia macierzowego A*B.

W dalszej części opracowania zajmiemy się wspomnianymi już wcześniej instrukcjami warunkowymi. Nie mają one nic wspólnego z pętlami, są narzędziem do rozgałęziania programu, jeżeli następuje taka potrzeba. Na przykład: jeżeli od wartości jakieś zmiennej lub wyrażenia logicznego - równości "==", różności ("~="), mniejszości ("<" lub "<="), większości (">" lub ">="). zależy droga postępowania, wtedy należy użyć instrukcji warunkowej. Zaczynamy ją zawsze od słówka kluczowego "if", po którym musi nastąpić wyrażenie logiczne, a kończymy słowem "end". W ramach jednej instrukcji można tworzyć wiele gałęzi za pomocą słów "else" lub "elseif". Wszystko to zostanie wyjaśnione na przykładach.

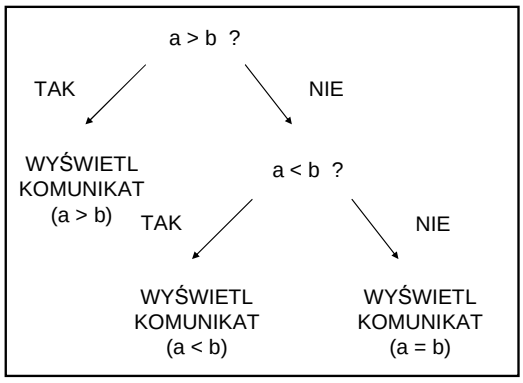
Napiszmy sobie prosty skrypt mający na celu porównanie ze sobą dwóch liczb i wyświetlenie informacji o tym, która z nich jest większa, lub o tym, że są równe:

```
clc
a = 1;
b = 6;

if a>b
    disp([num2str(a) ' jest większe od ' num2str(b)]);
end

if a<b
    disp([num2str(a) ' jest mniejsze od ' num2str(b)]);
end

disp('ta linijka wykona się zawsze');

if a==b
    disp([num2str(a) ' i ' num2str(b) ' są sobie równe']);
end
```

Definicja zmiennych "a" i "b" następuje bezpośrednio w skrypcie, na jego początku. Potem program sprawdza, czy większość a>b jest prawdziwa, i jeżeli tak, to wykonuje to, co znajduje się w zakresie pierwszej instrukcji "if" - czyli wyświetla odpowiedni komunikat. Potem program biegnie dalej i napotyka następną instrukcję "if" - tym razem sprawdzamy mniejszość a<b, jeżeli to prawda, to wykonywana jest instrukcja "disp" w zakresie drugiej instrukcji "if". Potem znów wszystko wraca do normy - aby to uwypuklić, umieszczono w programie linijkę, która zawsze się wykona, niezależnie od wartości "a" i "b". Ostatnia z kolei instrukcja "if" sprawdza równość "a" i "b" i jeżeli ona zachodzi, umieszcza odpowiedni komentarz. Proszę uruchomić program dla wartości "1" i "6", a potem dla innych zestawów: np. "6" i "1", a także "1" i "1". Za każdym razem proszę sprawdzać, który wpis pojawia się na ekranie w OP.
W powyższym przykładzie trzy instrukcje warunkowe, mimo iż powiązane tematycznie (pracujące na tych samych zmiennych "a" i "b") są napisane oddzielnie. Ich działanie można zobrazować w następujący sposób:



Nie jest to jednak dobra praktyka. W takich przypadkach powinno się je wiązać ze sobą przechodząc na wyrażenia logiczne przeciwne za pomocą poleceń "else" lub "elseif", co można przedstawić schematycznie w następujący sposób:



Najpierw algorytm sprawdza, czy a > b. Jeżeli tak - do OP trafia odpowiedni komunikat i po sprawie… Jeżeli nie, to mamy dwa wyjścia, albo liczby te są równe, albo a<b. No to sprawdźmy, czy a<b? Jeżeli tak, wypisujemy komunikat i koniec, jeżeli - to już nie ma innych możliwości - muszą być równe sobie. Tak więc równości nie musimy już sprawdzać, wystarczy wypisać komunikat o równości a i b, jeżeli algorytm nam odpowiedział dwa razy "nie" na zadane wcześniej pytania o większość i mniejszość. W Matlabie modelem informatycznym gałęzi przeciwnej z następnym pytaniem jest instrukcja "elseif", a modelem gałęzi przeciwnej bez specyfikacji logicznej (bez pytania, czy ma być wykonana) - instrukcja "else".
Poniżej pokazano odpowiednią modyfikację tego programu z uwagi na powyższy algorytm:

```
clc
a = 1;
b = 6;
```

```
if a>b
    disp([num2str(a) ' jest większe od ' num2str(b)]);
elseif a<b
    disp([num2str(a) ' jest mniejsze od ' num2str(b)]);
else
    disp([num2str(a) ' i ' num2str(b) ' są sobie równe']);
end
```

Proszę przetestować powyższy program dla różnych par wartości zmiennych "a" i "b", tak, aby przećwiczyć wszystkie możliwe warianty działania programu.

Instrukcje warunkowe często działają w zespół z pętlami "for". Np. rozpatrzmy następujący przykład: napisać funkcję, która będzie sprawdzała, ile elementów macierzy "A" jest dodatnich, a ile ujemnych, oraz podawała procentowy udział jednych i drugich w stosunku do wszystkich elementów macierzy. Funkcja, obok zwracania wartości, ma również wyświetlać wynik w OP. Poniżej pokazano rozwiązanie:

```
function rezultat = el_dod_uj(A)
[n m] = size(A);
clc
il_dod = 0;
il_uj = 0;
for i=1:n
    for j=1:m
        if A(i,j)>0
            il_dod = il_dod + 1;
        end
        if A(i,j)<0
            il_uj = il_uj + 1;
        end
    end
end
ud_dod = il_dod*100/(n*m);
ud_uj = il_uj*100/(n*m);
disp(['Liczba elementów dodatnich = ' num2str(il_dod)]);
disp(['Procentowy udział el.dod = ' num2str(ud_dod,3) '%']);
disp(['Liczba elementów ujemnych = ' num2str(il_uj)]);
disp(['Procentowy udział el.uj = ' num2str(ud_uj,3) '%']);
rezultat = round([il_dod ud_dod il_uj ud_uj]);
```

Najważniejsze elementy programu: zmienne "il_dod" i "il_uj" odpowiadają za liczbę elementów dodatnich oraz ujemnych w macierzy A. Przed wejściem do macierzy za pomocą pętli są one zerowane: stajemy przed "taśmą", na której będą się przesuwać elementy macierzy - mam zliczyć elementy dodatnie i ujemne - zanim taśmę uruchomimy, wpisujemy stan zerowy jednych i drugich. Taśmę uruchamiamy - pętla zaczyna działać. Elementy macierzy są ułożone w wierszach i kolumnach, dlatego też potrzebna jest osobna pętla "w poziomie" (po kolumnach) i "w pionie" (po wierszach). Będąc wewnątrz pętli podwójnie zagnieżdżonej, sprawdzamy, jaki znak ma element A(i,j) - jeżeli jest dodatni, zwiększamy zmienną "il_dod" o jeden. Dlaczego o jeden, a nie o A(i,j)? Pamiętajmy, że nie mamy liczyć sumy elementów dodatnich, tylko mamy policzyć, ile ich jest. W takim razie, ilekroć natkniemy się na "taśmie" na element dodatni, stawiamy kreseczkę w naszym "formularzu" - kolejny element dodatni zliczony. Za to właśnie odpowiada zwiększenie "il_dod" o 1.

Dokładnie tak samo wygląda praca ze zmienną "il_uj". Po zakończeniu pętli liczone są procentowe udziały liczby elementów dodatnich i ujemnych w całej macierzy A - czyli np. liczbę elementów dodatnich "il_dod" dzielimy przez liczbę wszystkich elementów macierzy A (czyli "m*n"), a następnie mnożymy przez 100 (%). Stąd wynika wartość zmiennej "ud_dod". Podobnie wygląda obliczenie wartości zmiennej "ud_uj". Wszystkie cztery zmienne są wypisywane na ekran (procentowe udziały z wykorzystaniem trzech miejsc znaczących - stąd drugi argument funkcji konwertującej "num2str"), a potem zaokrąglone wprowadzane są do wektora "rezultat" stanowiącego wartość funkcji.
Proszę program zapisać i uruchomić w OP:

```
>> A = round(rand(6,4)*12-5)
>> el_dod_uj(A)
```

Uwaga! Pętle programu można zapisać w nieco inny sposób, wykorzystując fakt, iż do elementów macierzy w Matlabie można odnosić się w następujący sposób: A(i), co oznacza kolejne elementy macierzy wzdłuż kolumn. Dlatego też zamiast dwóch pętli po "i" od "1" do "n" i po "j" od "1" do "m", można zapisać jedną pętlę po "i" od "1" do "m*n", czyli liczby elementów macierzy:

```
...
for i=1:n*m
    if A(i)>0
        il_dod = il_dod + 1;
    end
    if A(i)<0
        il_uj = il_uj + 1;
    end
end
...
```

Często przy konstruowaniu wyrażeń logicznych potrzebne jest łączenie dwóch prostych wyrażeń za pomocą relacji koniunkcji (iloczynu logicznego), alternatywy (sumy logicznej) lub negacji (zaprzeczenie logiczne). W Matlabie oznacza się te relacje za pomocą odpowiednich znaków: "&&" - koniunkcja, "||" - alternatywa,"~" - negacja. W starszych wersjach Matlaba (np. wersja 6) można było używać tylko pojedynczych symboli ("&" oraz "|"). W wersji przejściowej R2008 można używać obydwu notacji. W wersjach nowszych i następnych będzie można używać już tylko zapisu podwójnego (podobnie jak w języku C++). Poniżej program pokazujący zastosowanie relacji logicznych. Program sprawdza, ile elementów wektora "X" mieści się w przedziale zamkniętym [3 6] (zmienna "ile1"), ile leży w jego dopełnieniu do zbioru liczb rzeczywistych (zmienna "ile2"), oraz ile jest nieparzystych (zmienna "ile3"). Program mógłby być napisany w nieco inny sposób, ale ma on na celu pokazanie działania relacji logicznych.

```
function [ile1,ile2,ile3] = przedzial(X)
n = length(X);

ile1 = 0;
ile2 = 0;
ile3 = 0;
```

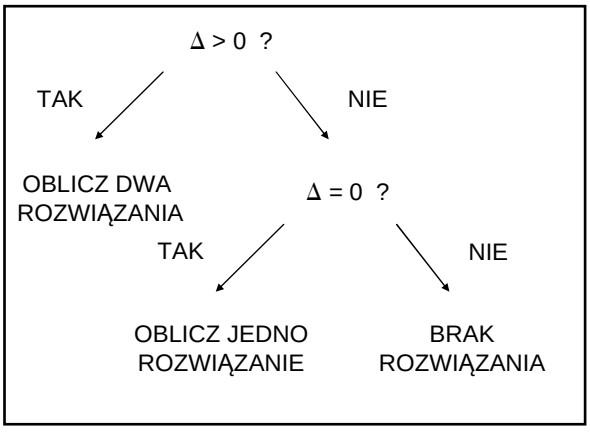
```
for i=1:n
    if X(i)>=3 && X(i)<=6
        ile1 = ile1 + 1;
    end
    if X(i)<3 || X(i)>6
        ile2 = ile2 + 1;
    end
    if ~(round(X(i)/2) == X(i)/2)
        ile3 = ile3 + 1;
    end
end
```

Sprawdzenie nieparzystości liczby polega na zbudowaniu zaprzeczenia parzystości. Parzystość z kolei sprawdzana jest jako równość wyniku dzielenia elementu przez 2 z zaokrągleniem tego wyniku - równość (brak części ułamkowej) wskazuje na liczbę parzystą (podzielną przez 2 bez reszty). Zaprzeczenie parzystości wskazuje na liczbę nieparzystą.

Typowym zadaniem, na którym pokazuje się działanie pętli, jest rozwiązanie równania kwadratowego. I właśnie to zagadnienie zostanie rozwiązane w Matlabie w dalszej kolejności. Dyskusja liczby rozwiązań równania kwadratowego postaci $ax^2+bx+c=0$, $a\neq 0$ rozpoczyna się od obliczenia wyróżnika równania $\Delta = b^2-4ac$. Następnie w zależności od jego znaku mogą wystąpić trzy sytuacje:

- $\Delta > 0$ - dwa rozwiązania $x_1 = \frac{-b-\sqrt{\Delta}}{2a}$, $x_2 = \frac{-b+\sqrt{\Delta}}{2a}$,
- $\Delta = 0$ - jedno rozwiązanie $x_1 = \frac{-b}{2a}$,
- $\Delta < 0$ - brak rozwiązania.

Konstrukcja algorytmu warunkowego powyższych wariantów jest bardzo podobna do tej, która już była omawiana przy okazji zadania porównywania dwóch liczb:



Poniżej pokazano przykładowy program funkcyjny:

```
function X = row_kwad(a,b,c)
```

```
delta = b^2 - 4*a*c;
if delta > 0
    X(1) = (-b-sqrt(delta))/(2*a);
    X(2) = (-b+sqrt(delta))/(2*a);
elseif delta == 0
    X(1) = -b/(2*a);
else
    X = [];
end
```

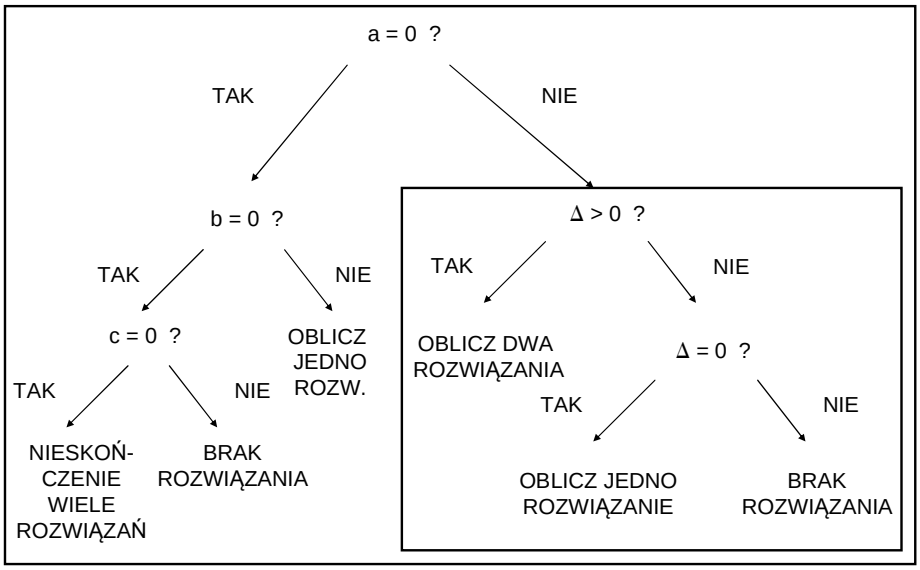
Program powyższy działa na zasadzie typowej "czarnej skrzynki" - nic nie wypisuje w OP, zwraca za to w wektorze "X" rozwiązania - dwa, jedno lub żadnego - wtedy wektor "X" ma przypisany symboliczny wektor pusty ("[ ]").
Proszę zapisać program, a następnie uruchomić go w OP, testując jego działanie dla różnych zestawów współczynników "a", "b" i "c", np.:

```
>> row_kwad(1,3,-10)
>> row_kwad(1,3,10)
>> row_kwad(1,-2,1)
>> row_kwad(1,0,16)
>> row_kwad(1,0,-16)
>> row_kwad(2,4,0)
```

Za każdym razem powinniśmy zobaczyć poprawne rozwiązanie. Co jednak, gdy współczynnik "a" będzie równy "0"? Spróbujmy wykonać:

```
>> row_kwad(0,4,2)
```

Pojawiły się symboliczne wartości "inf" (czyli nieskończoność) oraz "Nan" (czyli symbol nieoznaczony), co było spowodowane podzieleniem przez "0". A tymczasem dla "a" równego zero, równanie ma postać liniową $bx+c=0$, które przecież ma rozwiązanie, o ile $b\neq 0$. Jednakże algorytm rozwiązujący równanie kwadratowe nie potrafi tego uwzględnić. Należy więc go nieco rozbudować. Przede wszystkim wszystko to, co się wykonuje teraz, może się wykonywać pod warunkiem, że "a" jest różne od zera - w innym przypadku mamy do czynienia z równaniem liniowym, które może mieć jedno rozwiązanie (dla $b\neq 0$, $x_1=-\frac{c}{b}$), nieskończenie wiele rozwiązań (dla $b=0$ i $c=0$), lub może nie mieć rozwiązania (dla $b=0$ i $c\neq 0$). To wszystko rozszerza algorytm warunkowy do następującej postaci:



Natomiast program można skonstruować w sposób następujący:

```
function X = row_kwad(a,b,c)

if a==0
    if b==0
        if c==0
            X = Inf;
        else
            X = [];
        end
    else
        X(1) = -c/b;
    end
else
   delta = b^2 - 4*a*c;
   if delta > 0
       X(1) = (-b-sqrt(delta))/(2*a);
       X(2) = (-b+sqrt(delta))/(2*a);
   elseif delta == 0
       X(1) = -b/(2*a);
   else
       X = [];
   end
end
```

Nieskończenie wiele rozwiązań zostało symbolicznie oznaczone pseudo-wartością "Inf" dla wektora "X". Po zapisaniu programu można wrócić do OP i spróbować wykonać poprzednie polecenie:

```
>> row_kwad(0,4,2)
```

a także inne wywołania testujące pozostałe nowe warianty wykonania programu:

```
>> row_kwad(0,0,1)
>> row_kwad(0,0,0)
>> row_kwad(0,5,0)
```

Oczywiście poprzednie wywołania, dla równania kwadratowego muszą działać tak samo, jak poprzednio, np.

```
>> row_kwad(1,-3,-10)
>> row_kwad(1,1,1)
>> row_kwad(1,-2,1)
```

Ćwiczenie nr 6 do samodzielnego wykonania

Napisać program rozwiązujący układ równań 2x2 za pomocą metody wyznaczników Cramera. Danymi do programu są: macierz współczynników A oraz wektor b. Program powinien: dokonać analizy liczby rozwiązań, narysować dwie proste tworzące układ równań oraz zaznaczyć rozwiązanie (o ile istnieje). Wyznaczniki proszę obliczać bez użycia funkcji "det", metodą krzyżową, tzn. np. wyznacznik $\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{21}a_{12}$.

Ćwiczenie nr 7 do samodzielnego wykonania

Dane są współrzędne trzech punktów: (x0,y0), (x1,y1) oraz (x2,y2). Napisać program, który: narysuje trójkąt o wierzchołkach w tych trzech punktach, obliczy długości jego boków, obwód, długości wysokości, pole, miary kątów wewnętrznych (w stopniach), sprawdzi, jaki to rodzaj trójkąta (równoboczny, równoramienny, prostokątny, ostrokątny, rozwartokątny).
Wskazówki:

- długości boków trójkąta obliczyć jako długości wektorów tworzących boki tego trójkąta
- pole trójkąta obliczyć ze wzoru Herona: $P_{\Delta} = \sqrt{p(p-a)(p-b)(p-c)}$, gdzie a,b,c oznaczają długości boków, a p - połowę długości jego obwodu,
- miary kątów można wyznaczyć korzystając z twierdzenia cosinusów, które pozwala zapisać następujące równanie dla każdej kombinacji długości boków

$$a^2 = b^2 + c^2 - 2bc\cos(b,c)$$

W powyższym równaniu obliczany jest cosinus kąta pomiędzy bokami "b" i "c". Z powyższego związku można zatem wyznaczyć cosinus tego kąta, oraz cosinusy pozostałych kątów z analogicznych zależności. Same kąty (w radianach) można wyznaczyć korzystając z funkcji cyklometrycznej "acos". Następnie należy zamienić je na miarę stopniową.

- do sprawdzenia rodzaju trójkąta będą potrzebne porównania długości boków ora z kątów. Pamiętajmy, iż nie powinno się porównywać równości dwóch zmiennych typu rzeczywistego, których wartości są obliczane - mogą być one obliczane z błędem obcięcia na poziomie dokładności maszynowej i dlatego też mogą nie być idealnie równe sobie. Jeżeli zatem chcemy sprawdzić, czy dwie liczby a i b są równe sobie, zamiast pisać

```
a == b
```

należałoby napisać np.

```
abs(a-b)<10^-12
```

czyli $|a-b|<10^{-12}$. Jeżeli ta różnica okaże się mniejsza niż bardzo mała przecież liczba 10^{-12}, to uznajemy te liczby za równe sobie, mimo iż idealnie wcale nie muszą takie być.

Ćwiczenie nr 8 do samodzielnego wykonania

Rozbudować poprzedni program w celu sprawdzenia, czy punkty, na których w dalszej kolejności budowany jest trójkąt, nie są współliniowe. Jeżeli takie są, trójkąta zbudować na nich nie można.

Wskazówka: należy ułożyć równanie prostej przechodzącej przez dwa dowolne punkty (z trzech), a następnie sprawdzić, czy trzeci punkt na tej prostej leży - jeżeli tak, to są one współliniowe i nie wolno wykonywać następnych obliczeń dla trójkąta.

Ćwiczenie nr 9 do samodzielnego wykonania

Dany jest okrąg o promieniu "r" i środku w punkcie (0,0). Dokonać losowania "n" punktów leżących wewnątrz kwadratu opisanego na tym okręgu. Policzyć, ile punktów leży wewnątrz okręgu, odnieść liczbę takich punktów do wszystkich wylosowanych. Zadanie zilustrować rysunkiem: narysować okrąg, oraz wszystkie wylosowane punkty, które znajdą się w jego wnętrzu. Dane do programu: "r" oraz "n".

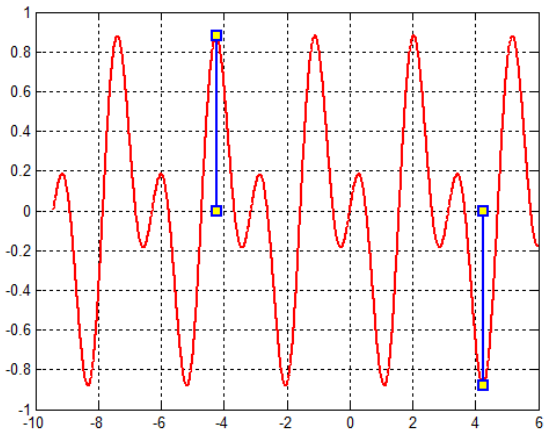
Ćwiczenie nr 10 do samodzielnego wykonania

Narysować zestaw ("pęk") n wektorów o długościach jednostkowych wychodzących promieniście z jednego punktu o współrzędnych (x0,y0). Kolory tych wektorów dobierać losowo z zestawu 'r', 'b', 'k', 'g', 'k', 'm' ,'c' oraz 'y'.

Ćwiczenie nr 11 do samodzielnego wykonania

Dana jest funkcja $\sin(x)\cos(kx)$ w przedziale $[a \quad b]$. Narysować jej wykres, oraz wykorzystać wygenerowany wektor wartości funkcji w punktach pośrednich wykresu do znalezienia ekstremalnych (max i min) wartości (y) funkcji w danym przedziale oraz ich położeń (x). Znalezione punkty narysować na wykresie. Program wykonać w dwóch wariantach: z wykorzystaniem pętli "for" oraz instrukcji "if", oraz jedynie za pomocą wbudowanych funkcji Matlaba (m.in. "sortrows"). Dane do programu: "k", "a", "b".

Przykładowy efekt graficzny działania programu dla "k=3", "a=-3*pi" i "b=6".



Ćwiczenie nr 12*[1] do samodzielnego wykonania
Zmodyfikować tak program z zadania nr 11, aby na wykresie zaznaczane były wszystkie maksima i minima o "tych samych" wartościach (czyli w ocenie programu różniących się o bardzo niewiele) funkcji.

Ćwiczenie nr 13[2] do samodzielnego wykonania**
Zmodyfikować tak program z zadania 12, aby program zaznaczał na wykresie także wszystkie maksima i minima lokalne.

Jako ostatni przykład działania instrukcji warunkowej "if" zaprezentowany zostanie program obliczający silnię liczby naturalnej (n!). Silnia może być obliczana dwojako. Pierwszy sposób, wynikający z tzw. definicji iteracyjnej silni, to obliczenie iloczynu kolejnych liczb naturalnych od 2 do n. Z zaprogramowaniem tego sposobu Czytelnik nie powinien mieć żadnych problemów. Ciekawszy jednak jest tzw. sposób rekurencyjny, czyli

$$n! = \begin{cases} 1 & dla \quad n \le 1 \\ n(n-1)! & dla \quad n > 1 \end{cases}$$

Rekurencja (wzajemne wywoływanie) polega na tym, iż silnia liczona jest przy wykorzystaniu… silni. Jednakże jest to zawsze silnia z liczby o 1 mniejszej - dlatego też rekurencja trwa tak długo, aż nie dojdzie do silni z 1 - wtedy wynik jest znany (1). Definicję powyższą można przenieść w skali 1:1 do Matlaba dlatego, iż w Matlabie dopuszczalne jest wywoływanie rekurencyjne funkcji - jednakże po pięćsetnym wykorzystaniu rekurencji tej

[1] zadanie o podwyższonym stopniu trudności

[2] zadanie o znacznie podwyższonym stopniu trudności

samej funkcji wystąpi błąd i przerwanie działania funkcji.. To pozwala obliczyć silnię maksymalnie z liczby 499. Zaprogramujmy więc powyższe:

```
function s = silnia(n)
if n <= 1
    s = 1;
else
    s = n*silnia(n-1);
end
```

Prawda, że proste? Teraz nie pozostało nic innego do zrobienia, jak zapisać program i przetestować go w OP - wyniki możemy porównać z działaniem wbudowanej funkcji Matlaba - "factorial":

```
>> silnia(0)
>> silnia(1)
>> silnia(2)
>> silnia(3)
>> silnia(4)
>> silnia(5)
>> silnia(10)
>> factorial(10)
```

W dalszym ciągu opracowania zapoznamy się z drugim rodzajem pętli: o **nieokreślonej liczbie przebiegów**. Pętla ta potrzeba jest wtedy, gdy nie jesteśmy w stanie określić w czasie pisania programu, ile razy będą się musiały wykonać instrukcje w niej zawarte. Dlatego też zawsze w przypadku takich pętli to właśnie od efektów działania tych instrukcji zależy, czy pętla ma się zatrzymać, czy dalej "kręcić". Warunek **działania** (działania, a nie przerwania!) definiowany jest na jej początku, po słowie kluczowym "while" (które tłumaczymy "podczas gdy" - dlatego też jest to warunek działania pętli). Instrukcje zawarte w pętli należy umieścić pomiędzy nagłówkiem pętli (while), a instrukcją "end":

```
while warunek
    …
    …
end
```

Ponieważ wiele razy stosowana była już pętla "for", spróbujmy odtworzyć jej działanie, ale z wykorzystaniem pętli "while". Proszę uruchomić następujący skrypt:

```
clc
i = 1;
while i<=10
    disp('pętla się kręci...')
    i = i + 1;
end

i = 1;
while i<=10
```

```
    disp(['pętla się kręci, i jest równe ' num2str(i)])
    i = i + 1;
end
i
```

Zawiera on dwie pętle "while". Każda z nich "jest liczona" za pomocą parametru "i" - nie jest on jednak związany z pętlą (tak, jak to było w przypadku pętli "for"), i dlatego też sami musimy zadbać o to, aby ustawić jego wartość startową ("i=1"), napisać warunek działania ("i<=10") oraz zwiększyć o "1" po każdym obiegu pętli. Po uruchomieniu obydwie pętle zadziałają tak, jak odpowiedniki w postaci pętli "for", wcześniej już uruchamiane. Czy wszystko jest takie same? Na koniec programu wypisywana jest wartość zmiennej "i". W przypadku pętli "for" miałaby ona wartość "10", tutaj jest o "1" większa. Dlaczego? Dlatego, iż program musiał ją fizycznie zwiększyć, aby przekonać się, że "i=11" nie spełnia warunku działania pętli. W przypadku pętli "for", pętla sama sprawdzała, czy wartość "i" już dosięgła wartości końcowej parametru sterującego, zadeklarowanej w jej nagłówku.
Oczywiście pisanie pętli "while" zamiast pętli "for" nie ma dużego sensu. Jakie więc może ona mieć zastosowanie? Rozpatrzymy następujący przykład: napisać program, który będzie losował liczby całkowite z przedziału od "a" do "b" tak długo, aż ich suma będzie większa niż lub równa 1000. W przypadku takiego zagadnienia nie wiemy z góry, ile takich losowań będzie potrzebnych, bo to zależy od tego, jak wielki jest przedział i jak duże liczby będą z niego losowane. Dlatego też tego zadania **nie da** się - przynajmniej w sposób racjonalny - zrealizować za pomocą pętli "for". Pozostaje pętla "while". Proszę uruchomić następujący skrypt:

```
clc
a = 2;
b = 10;

suma = 0;
ile = 0;
while suma < 1000
    suma = suma + round(rand(1,1)*(b-a) + a);
    ile = ile + 1;
end

disp(['Suma wynosi = ' num2str(suma)]);
disp(['Wylosowano liczb = ' num2str(ile)]);
```

Proszę zwrócić uwagę, iż nagłówek pętli zawiera wyrażenie logiczne będące przeciwieństwem sformułowania, które padło w tekście zadania - to dlatego, iż - jeszcze raz to podkreślam - pętla musi mieć zdefiniowany warunek działania, a nie przerwania, jak to się intuicyjnie nasuwa. A pętla ma działać wtedy, gdy suma będzie mniejsza niż 1000. Z kolei ma się przerwać, gdy suma będzie większa lub równa 1000 - jedno oczywiście wynika z drugiego. Proszę zapisać program i uruchomić go kilka razy dla różnych zestawów wartości zmiennych "a" i "b", wprowadzanych w kodzie programu. Ale proszę pamiętać: chociaż "b" musi być dodatnie!
A teraz zróbmy mały eksperyment: proszę w skrypcie zdefiniować "a" i "b" ujemne, np. "a=-10" i "b= -1". Proszę uruchomić program - i zajrzeć do OP - nic się nie pokazało, program wciąż pracuje. Co się stało? Skoro przedział zawiera tylko liczby ujemne, to tylko liczby

ujemne są losowane - w takim wypadku nie ma najmniejszych szans stworzyć sumy, która przekroczy wartość 1000 - suma będzie też ujemna. Co zatem zrobić, aby zapętlony program odblokować? Zamknąć Matlab? Nie ma takiej potrzeby - proszę przejść do Matlaba i nacisnąć kombinację klawiszy "Ctrl+Pause/Break". Program powinien się przerwać z informacją o błędzie - przerwaniu manualnym. Stąd wniosek: przy pisaniu pętli "while" trzeba pamiętać o niebezpieczeństwie zapętlenia. Dobrze jest zastosować jakąś "boczną furtkę" ucieczki z pętli, gdyby główny warunek działania pętli był zawsze spełniony. Np. w naszym przypadku może to być maksymalna liczba losowań, na jaką może sobie pozwolić program:

```
clc
a = -10;
b = -1;

ile_max = 10000;

suma = 0;
ile = 0;
while suma < 1000 && ile<=ile_max
    suma = suma + round(rand(1,1)*(b-a) + a);
    ile = ile + 1;
end

disp(['Suma wynosi = ' num2str(suma)]);
disp(['Wylosowano liczb = ' num2str(ile)]);
```

W powyższym przykładzie, maksymalna liczba losowań wynosi dziesięć tysięcy - i właśnie dla powyższych ujemnych wartości "a" i "b" program zadziała prawidłowo, po 10000 losowań zakańczając działanie. Proszę sprawdzić, jak działa dla prawidłowych danych, np. "a=1" i "b=10". Jeszcze jeden komentarz: pętla ma się zakończyć, gdy suma będzie równa lub większa od 1000 **lub** przekroczona zostanie maksymalna liczba losowań. Ale warunek działania pętli jest odwrotny: pętla ma działać, gdy suma będzie mniejsza od 1000 **i** (&&) liczba losowań nie przekroczy wartości maksymalnej.

Ćwiczenie nr 14 do samodzielnego wykonania

Napisać funkcję obliczają wartość jednej z funkcji analitycznych (sinus, cosinus, funkcja ekspotencjalna itd.) korzystając z odpowiedniego rozwinięcia w szereg potęgowy Taylora. Dane do programu stanowią: "x" - argument, dla którego ma być obliczana wartość, "edop" - dopuszczalny błąd obliczeń, wyrażony w procentach, "nmax" - maksymalna liczba wyrazów rozwinięcia, jakie może wykorzystać program. Kontroli przerwania pętli dokonać za pomocą kryterium opartego na bezwymiarowym błędzie względnym.
Rozwinięcia podstawowych funkcji są dostępne np. pod adresem:

http://pl.wikipedia.org/wiki/Wz%C3%B3r_Taylora#Rozwini.C4.99cia_niekt.C3.B3rych_funkcji_w_szereg_Maclaurina

Proszę wybrać sobie jedno z nich, np. proste rozwinięcie funkcji ekspotencjalnej

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

napisać program funkcyjny oparty o to rozwinięcie, a następnie sprawdzić wyniki używając wbudowanej funkcji Matlaba, "exp".

Ćwiczenie nr 15* do samodzielnego wykonania
Rozszerzyć poprzednie zadanie o ilustrację graficzną w postaci wykresu zbieżności rozwiązania do wartości ścisłej, oraz zbieżności względnego (bezwymiarowego) błędu obliczeń w skali pół-logarytmicznej.

Na koniec opracowania przedstawiona zostanie **instrukcja wyboru**. Jeżeli mamy do zaprogramowania instrukcję wyboru z wieloma wariantami, to zamiast pisać rozgałęzionej instrukcji warunkowej "if", można wszystko zorganizować za pomocą instrukcji wyboru "switch". Ma ona następującą składnię:

```
switch zmienna_wyboru
        case wartosc1
                …
                …
        case wartosc2
                …
                …
        …
        otherwise
                …
                …
end
```

Bardzo często jest używana przy projektowaniu graficznego menu, w którym użytkownik ma szansę wyboru odpowiedniej opcji poprzez kliknięcie na nią. Wtedy odpowiednia akcja będzie podejmowana w oparciu o instrukcję "switch". Wróćmy do zagadnienia rozwiązywania równania kwadratowego - efektem pracy była funkcja "X = row_kwad(a,b,c)", którą należało uruchomić z OP podając jej trzy argumenty aktualne. Teraz napiszemy program, w którym to użytkownik wybierając odpowiednią opcję będzie mógł: zdefiniować współczynniki równania, uruchomić obliczenia, zilustrować zadanie grafiką, wyjść z programu. Do zorganizowania tych opcji w całość służy funkcja "menu". Podajemy jego nazwę oraz nazwy kolejnych pól wyboru. Funkcja ta zwraca numer pola, w które kliknął użytkownik. Następnie instrukcją "switch" wybieramy działania dla poszczególnych pól. Całość programu działa w obrębie pętli "while", jako iż program po wykonaniu jednej z czynności, np. zdefiniowania współczynnika "a" powinien cały czas działać i pozwalać na wybór innych pól.
Poniżej przedstawiono przykładowy program realizujący założone czynności:

```
clc
close all
clear all
a = [];
b = [];
c = [];

opcja = 1;

while opcja ~= 6
    opcja = menu('PROGRAM - RÓWNANIE KWADRATOWE','WSPÓŁCZYNNIK a',...
        'WSPÓŁCZYNNIK b','WSPÓŁCZYNNIK c','ROZWIĄZANIE','WYKRES','WYJŚCIE');
    switch opcja
        case 1
            a = input('Podaj współczynnik a = ');
        case 2
            b = input('Podaj współczynnik b = ');
        case 3
            c = input('Podaj współczynnik c = ');
        case 4
            if isempty(a)==0 && isempty(b)==0 && isempty(c)==0
                disp('Współczynniki równania');
                disp([a b c]);
                x = row_kwad(a,b,c);
                disp('Rozwiązania równania');
                disp(x);
            else
                disp('Nie wszystkie współczynniki zostały zdefiniowane!');
            end
        case 5
            % przyszła grafika
    end
end
```

Domyślne wartości współczynników stanowią puste obiekty ("[ ]"). Całość programu działa w pętli "while", która kręci się tak długo, aż użytkownik wybierze pole nr 6 - czyli 'WYJŚCIE'. W każdym innym przypadku pętla działa. Funkcja "menu" rysuje menu i po uruchomieniu pozwala na wybór jednego z 6 pól. Użytkownik klikając przypisuje zmiennej "opcja" jedną z wartości 1,2,3,4,5 lub 6. Każda odpowiada za coś innego - instrukcja "switch" rozdziela każdej zadania: 1,2,3 zapytują się o wartości odpowiednio "a", "b" lub "c". Pole 4 odpowiada za rozwiązanie zadania - o ile wszystkie współczynniki są zdefiniowane. Jeżeli tak nie jest (co sprawdza funkcja "isempty", która jest równa zeru, jeżeli jej argument jest obiektem pustym), pojawia się w OP odpowiedni komentarz. Pole 5 jest na razie puste - Czytelnik proszony jest o jego wypełnienie procedurami graficznymi rysującymi wykres funkcji równania oraz zaznaczającymi na nim rozwiązanie(a), o ile istnieje(ą). Prawdopodobnie pomocna będzie funkcja "isinf", sprawdzająca, czy liczba (argument) jest określona symbolem nieskończoności ("inf"). Proszę zwrócić uwagę, że i to pole może się wykonać tylko wtedy, gdy znane są współczynniki i rozwiązanie (np. gdy rozwiązania nie ma, to może narysować się sam wykres, a w OP może być wyświetlona informacja, iż ewentualne rozwiązanie zostanie naniesione na wykres, jeżeli zadanie zostanie rozwiązane).