Podstawowe operacje, wykonywane w środowisku obliczeniowym Matlab
oraz
Wstęp do tworzenia funkcji m-skryptów uruchomieniowych

Środowisko obliczeniowe Matlab obecnie znajduje szeroki krąg użytkowników (strona firmowa Mathworks™ http://www.mathworks.com). Pomimo istnienia we wcześniejszym okresie narzędzi obliczeniowych o charakterze czysto akademickim (na przykład programu Maple™ - patrz: strona domowa *Waterloo University* w Kanadzie), służących w rachunku symbolicznym, po części analitycznym, środowisko Matlab przeważyło w liczbie użytkowników pragnących wykorzystywać pewne narzędzie analiz, symulacji, obliczeń analitycznych, symbolicznych, jak i złożonych narzędzi modelowania obiektów.

Na obecnych zajęciach przedstawione zostaną przede wszystkim podstawy rachunku macierzowego, po części wektorowego. Bowiem środowisko obliczeniowe Matlab (skrót od dwóch pojęć: ***Mat****hematical* ***Lab****oratory*) jest w szczególności dobrze wyposażone w narzędzia do manipulacji danymi macierzowymi.

Natomiast istotną wadą tego środowiska jest *stosunkowo wysoka cena* licencji, zarówno jednostanowiskowej, jak i wielostanowiskowej, zawężająca w praktyce możliwość stosowania tego pakietu obliczeniowego wyłącznie do sal komputerowych ośrodków akademickich. Dla rzeszy studentów, w praktyce jest to narzędzie o cenie zaporowej, nieosiągalnej z punktu widzenia budżetu finansowego przeciętnej osoby podejmującej studia na kierunkach matematyczno-przyrodniczych.

Z pomocą przychodzą programy – klony, imitujące składnię i funkcjonowanie składni poleceń środowiska Matlab. Dostępne są one bezpłatnie, a o nazwach podprojektów – kolejnych wersji mikro-środowisk obliczeniowych – zgrupowanych wokół projektu ***Octavia***. Ten z kolei projekt rozwijany jest od blisko dziesięciu lat (strona domowa projektu tego przedsięwzięcia: http://www.octavia.org). W tym wypadku można oczekiwać skromnej w rozmiarach instalacji programu projektu ***Octavia***, równie skromnego interfejsu, pozbawionego graficznego wsparcia w podstawowych oknach nawigacji tego środowiska: oknie historii poleceń, oknie poleceń, oknie zmiennych środowiskowych (są to podokna tekstowe!). Chociaż jednocześnie należy oczekiwać wykonania w oknach graficznych podstawowych poleceń plottingu, czy przedstawienia danych obrazów 2D, analogicznie do wykonań tych poleceń w środowisku Matlab.

Stąd, można oczekiwać istnienia dość dużej zgodności ze składnią, sensem i przesłaniem operacji podstawowych, zwłaszcza dotyczących poleceń z zakresu algebry liniowej, rachunku macierzowego, organizowania danych w przestrzeni roboczej Matlab, operacji zapisu i odczytu z dysku twardego danych roboczych, odpowiednio: z i do przestrzeni roboczej tego środowiska.

Poniżej przedstawiono podstawowe operacje wejścia/wyjścia, zarządzania/organizacji zmiennych oraz ponadto manipulacji (tj. sklejanie, wycinanie, filtrowanie, obracanie, wyliczanie) zarówno na zbiorze nazw zmiennych, jak i zawartości zmiennych środowiskowych. W następnej kolejności przedstawiono reguły tworzenia skryptów – funkcji uruchomieniowych w realizacji podstawowych, elementarnych obliczeń (na przykładzie różnych sposobów realizacji zadania obliczania silni i nie tylko).

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

Po uruchomieniu środowiska obliczeniowego Matlab (w wersji na przykład 7.6), po wydaniu polecenia ***version***:

```
>>version
```

otrzymujemy, odpowiedź, zgłaszającą numer wersji:

```
ans =
7.10.0.499 (R2010a)
```

Natomiast po wydaniu polecenia ***version***, otrzymujemy opis (nazwy i numeru wersji) wszystkich skrzynek narzędziowych – zestawów specjalistycznych funkcji (*toolbox'ów*). Poniżej przedstawiono częściowy listing wyników tego polecenia (wnętrze listingu, tj. wewnętrzne linijki zostały wykropkowane 6 kropkami w trzech kolejnych liniach i opatrzone komentarzem **na czerwono**!) :

```
>>ver
-----------------------------------------------------------------------------
MATLAB Version 7.10.0.499 (R2010a)
MATLAB License Number: 161051
Operating System: Microsoft Windows 7 Version 6.1 (Build 7600)
Java VM Version: Java 1.6.0_12-b04 with Sun Microsystems Inc. Java HotSpot(TM) 64-Bit
Server VM mixed mode
-----------------------------------------------------------------------------
MATLAB                                                  Version 7.10       (R2010a)
Simulink                                                Version 7.5        (R2010a)
Aerospace Blockset                                      Version 3.5        (R2010a)
Aerospace Toolbox                                       Version 2.5        (R2010a)
Bioinformatics Toolbox                                  Version 3.5        (R2010a)
Communications Blockset                                 Version 4.4        (R2010a)
. . .
. .
.%poniżej występuje cząstkowe przytoczenie listingu%
Symbolic Math Toolbox                                   Version 5.4        (R2010a)
System Identification Toolbox                           Version 7.4        (R2010a)
```

```
SystemTest                                          Version 2.5        (R2010a)
Video and Image Processing Blockset                 Version 3.0        (R2010a)
Wavelet Toolbox                                     Version 4.5        (R2010a)
```

Uwaga: We wszystkich wydanych poleceniach w wydrukach rezultatów obrano wydruk ciemnoniebieską czcionką (jednakże o zmiennej, dostosowanej do potrzeb edycyjnych tego dokumentu wielkości).

Polecenie ***clc*** służy do czyszczenia pozostałości wydruków głównego okna wydawania poleceń:

```
>>clc
```

Inicjacja zmiennych jest dość intuicyjna, to znaczy z automatycznie obieranym typem zmiennej, w zależności typu przypisanej wartości:

```
>>a=2;
>>b=3.0;
>>c='A';
>>s='to jest łańcuch znakowy';
```

W celu wylistowania liczby, typu i wielkości pamięci obszaru roboczej, zajmowanej przez powyższe cztery zmienne, zainicjowane przez wartości o różnym typie, należy wykorzystać polecenie skrótowych wydruków ***who*** (o zmiennych środ.):

```
>>who
Your variables are:
a    ans  b    c    s
```

, lub polecenie szczegółowego wydruku ***włos*** (obejmującego informację o typie i wielkości zajętości pamięci przez zmienne)*:*

```
whos
  Name        Size              Bytes  Class     Attributes
  a           1x1                   8  double
  ans         1x19                 38  char
  b           1x1                   8  double
  c           1x1                   2  char
  s           1x23                 46  char
```

Zmienne liczbowe o wartościach inicjujących, równych 2 lub 3.0 są domyślnie tworzonymi zmiennymi o typie ***double*** (ang. ***double float point –*** format zmiennoprzecinkowy podwójnej precyzji). Celem zmiany typu zmiennej z formatu zmiennoprzecinkowego na typ całkowitoliczbowy bez znaku należy zastosować operator rzutowania do typu ***uint8*** (ang. ***unsigned integer with 8 bits*** – ***uint8***):

```
>>a=int8(2);
```

Jednak zakres zmienności liczby całkowito-liczbowej dla typu ***uint8*** wynosi tylko 256 (od 0 do 255 włącznie). Tymczasem zapis liczby całkowito-liczbowej może wymagać większego zakresu zmienności. Stąd można zastosować operator rzutowania do typu uint16 (ang. ***unsigned integer with 16bits*** – ***uint16***):

```
>>b=int16(3);
```

Zakres zmienności wartości możliwych do przypisania liczbie 16-bitowej reprezentacji typu *całkowito-liczbowego* wynosi $[0\ 2^{16-1}]$, to jest (bagatela) od 0 do 65635 włącznie. Niemniej sposób i format zapisu liczby *zmiennoprzecinkowej* określa znacznie większy (bezwzględny) zakres zapisu i przechowywania wartości, zarówno liczb niezwykle dużych, jak i liczb niezwykle małych. W formacie liczby zmiennoprzecinkowej wyróżnia się obszar zapisu mantysy (przedrostka ciągu cyfr znaczących) oraz obszar zapisu eksponentu liczby. Stąd, chociaż liczby całkowito-liczbowe charakteryzują się *bezwzględną* dokładnością zapisu (bezwzględną i jednoznaczną, bo o wartościach dopuszczalnych – jedynie *całkowitych*), to liczby zmiennoprzecinkowe osiągają wartości o eksponencie równym **+-308**. Ponadto obszar pamięci wymagany w przechowywaniu liczby zmiennoprzecinkowej (***float***) jest czterokrotnie większy od obszaru pamięci wymaganego przez format liczby ***uint16*** oraz aż 8-krotnie większy od obszaru pamięci wymaganego przez format liczby ***uint8***.

```
>>whos
  Name        Size              Bytes  Class     Attributes
  a           1x1                   1  int8
  ans         1x19                 38  char
  b           1x1                   2  int16
  c           1x1                   2  char
  s           1x23                 46  char
```

Samo polecenie ***who*** można wykorzystywać w kontekście znanym z przetwarzania potokowego (filtracji) nazw plików systemowych (UNIX/DOS):

```
>>b_wartosc_dodatnia = abs(-4)
>>a_absolute_value = abs(-3)
>>c_zmienna_bis = 'C'
>>s_lancuch_I = 'to s¹ znaki'
>>s_lancuch_II = 'to jest inny ci¹g znakow'
```

Poniżej przykład wyświetlania szczegółowego zmiennych z wystąpieniem gdziekolwiek w ich nazwach, co najmniej jednej litery ***a***:

```
>>whos *a*
  Name                    Size            Bytes  Class     Attributes
  a                       1x1                 1  int8
  a_absolute_value        1x1                 8  double
  ans                     1x19               38  char
  b_wartosc_dodatnia      1x1                 8  double
  c_zmienna_bis           1x1                 2  char
  s_lancuch_I             1x11               22  char
  s_lancuch_II            1x24               48  char
```

Analogicznie, przykład wyświetlania zmiennych z wystąpieniem gdziekolwiek w ich nazwach, co najmniej jednej litery ***b***:

```
>>whos *b*
  Name                    Size            Bytes  Class     Attributes
  a_absolute_value        1x1                 8  double
  b                       1x1                 2  int16
  b_wartosc_dodatnia      1x1                 8  double
  c_zmienna_bis           1x1                 2  char
```

Podobnie, poniżej przykład z wystąpieniem jednej oraz dwóch rzymskich jedynek (**I** i **II**):

```
>>whos *I*
  Name              Size            Bytes  Class    Attributes
  s_lancuch_I       1x11               22  char
  s_lancuch_II      1x24               48  char

>>whos *II*
  Name              Size            Bytes  Class    Attributes
  s_lancuch_II      1x24               48  char
```

Ostatecznie przytoczono przykład z filtrowaniem określającym odpowiednio wystąpienie litery ***a*** na początku i końcu nazwy:

```
>>whos a*
  Name                  Size            Bytes  Class     Attributes
  a                     1x1                 1  int8
  a_absolute_value      1x1                 8  double
  ans                   1x19               38  char

>>whos *a
  Name                    Size            Bytes  Class     Attributes
  a                       1x1                 1  int8
  b_wartosc_dodatnia      1x1                 8  double
```

Zmienne środowiska Matlab, pozostawione same sobie są o charakterze ulotnym, (zamiast permanentne, które zachowują się przy wyjściu ze środowiska Matlab). Stąd zachodzi konieczność zapisu zmiennych roboczych do pliku systemowego (pojedynczego) z użyciem polecenia ***save***. Natomiast polecenie ***load*** umożliwia załadowanie zmiennych roboczych ze wskazanego pliku systemowego:

```
>>save zmienne.mat
>>clear
>>load zmienne.mat
>>who
Your variables are:
a                   b                   c_zmienna_bis       s_lancuch_II
a_absolute_value    b_wartosc_dodatnia  s
ans                 c                   s_lancuch_I
```

W powyższym przytoczeniu 4 linijek poleceń, należy rozróżnić polecenie ***clc*** (ang. ***clear console***) od polecenia ***clear***, które (nieodwołalnie) usuwa z przestrzeni roboczej Matlab zmienne środowiskowe!

Środowisko Matlab dysponuje stosunkowo dobrze rozbudowanym zestawem poleceń, operujących na danych formatu daty oraz czasu (również na danych bieżącej daty i czasu). W przeciwieństwie do systemu UNIX (gdzie następuje pomiar w sekundach, jako jednostce wewnętrznej czasu i odcinków czasu naliczanych w systemie od 1 stycznia 1970r), jak i również w przeciwieństwie do arkusza kalkulacyjnego Excel (tam czas jest naliczany od 1 stycznia 1980r), w środowisku Matlab czas naliczany jest w dniach, jako wewnętrznej jednostce przeliczania czasu od umownego punktu będącego początkiem naszej ery!

Niemniej, polecenia typu ***clock*** zwracają wprawdzie bieżący czas i bieżącą datę, lecz w postaci sześcioelementowego wektora wyjściowego danych:

```
>>clock
ans =
  1.0e+003 *
    2.0110    0.0110    0.0060    0.0200    0.0250    0.0575
```

Zmienna uniwersalna ***ans*** przechowuje wynik ostatnio wywoływanego polecenia (w tym przypadku polecenia ***clock***).

Z pomocą w rozwikłaniu wyniku polecenia ***clock***, którego wynikiem są liczby 6-elementowego wektora o znaczeniu:

[rok miesiąc dzień godzina minuty sekundy],

przychodzi polecenie ***datestr***:

```
>> datestr(ans)
ans =
06-Nov-2011 20:25:57
```

Polecenie ***datestr*** można również wykorzystać w zagnieżdżeniu polecenia ***clock***, otrzymując czytelną postać bieżącego czasu oraz daty:

```
>> datestr(clock)
ans =
06-Nov-2011 20:38:52
```

Analogicznie, można wykorzystywać polecenie ***now***, chociaż zwrócona wartość seryjna czasu, równie dobrze '*tłumaczona*' na czytelny format danych:

```
>> ntime=now
ntime =
  7.3481e+005
>> datestr(ntime)
ans =
06-Nov-2011 20:42:52,
```

przy czym liczba przechowywana w zmiennej ***time*** równa 734813, wskazuje liczbę dni, które upłynęły od początku ery. Stąd wstawienie wyrażenia:

```
>> datestr(734813.0)
```

skutkuje wskazaniem jedynie bieżącej daty:

```
ans =
06-Nov-2011
```

Innymi słowy, jest to w skutkach wykonania (powyższe) polecenie dokładnie takie samo, co polecenie ***date***:

```
>> date
ans =
06-Nov-2011
```

Natomiast wydanie polecenia ***datenum*** umożliwia określenie liczby dni, (czyli określenie w formacie wewnętrznym przeliczania czasu w Matlab liczby jednostek czasu), które upłynęły od początku naszej ery do ustalonego przez wskazaną datę, (na przykład daty – początku naliczania sekund we wspomnianym systemie UNIX):

```
>> datenum('01-Jan-1970')
ans =
      719529
```

Okno szczegółowej pomocy ***help*** może nieco więcej rzucić światła na składnię poleceń. Poniżej przytoczono przykładowe wywołanie opcji szczegółowej pomocy dla polecenia ***datenum***:

```
>>help datenum
 DATENUM Serial date number.
      N = DATENUM(V) converts one or more date vectors V into serial date
      numbers N. Input V can be an M-by-6 or M-by-3 matrix containing M full
      or partial date vectors respectively.  DATENUM returns a column vector
      of M date numbers.
. . .
. .
.
% poniżej występuje cząstkowo przytoczony listing opcji szczegółowej pomocy polecenia datenum %
      Examples:
            n = datenum('19-May-2000') returns n = 730625.
            n = datenum(2001,12,19) returns n = 731204.
            n = datenum(2001,12,19,18,0,0) returns n = 731204.75.
            n = datenum('19.05.2000','dd.mm.yyyy') returns n = 730625.
```

Ostatecznie, by od założonej daty na przykład 18 Listopada 2011 roku '*cofnąć się*' o powiedzmy 20 lat, należy wydać następujące polecenie:

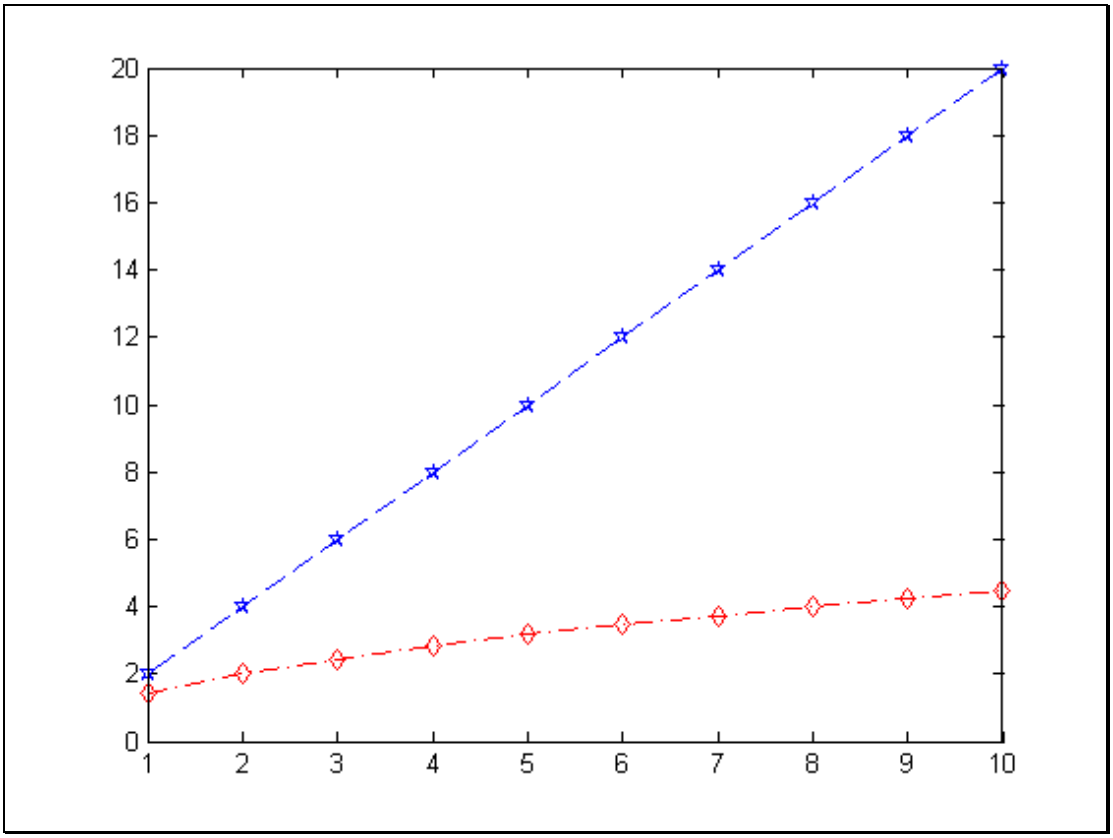
```
>> datestr(datenum('18-Nov-2011')-20*365.25)
ans =
18-Nov-1991
```

W środowisku Matlab, istnieje kilka udogodnień – dotyczących możliwości rozwijania szeregów liczb rzeczywistych (w szczególności liczb całkowitych). Oto przykład rozwinięć dwóch szeregów:

```
a=1:10 %szereg liczb naturalnych

a =
     1     2     3     4     5     6     7     8     9    10

b=2:2:20 % szereg liczb naturalnych parzystych
b =
     2     4     6     8    10    12    14    16    18    20
```

Poniżej przedstawiono wywołanie dwukrotne polecenia ***plot*** wykreślania zależności (funkcji tabelarycznie przedstawionej jako: ***a*** – szereg dziedziny, ***b*** - szereg przeciwdziedziny):

```
plot(a,b)
hold on;
plot(a,b.^(1/2),'rd');
plot(a,b,'bp--');
```



*Rys. 1 Wykres zależności y = 2x, to znaczy zależności w postaci szeregu **b** liczb parzystych od szeregu **a** liczb naturalnych (przedstawionej na niebiesko) oraz zależności w postaci szeregu pierwiastków kwadratowych z wartości szeregu **a** (przedstawionej na czerwono).*

W szczególności, opcja szczegółowej pomocy, wywołana dla polecenia ***plot***, przedstawia liczne możliwe wartości atrybutu oznaczenia *koloru* (czarny - ***black*** – 'k', niebieski - ***blue*** – 'b', czerwony - ***red*** – 'r', żółty - ***yellow***' – y, cyjanowy - ***cyjan*** – 'c', magenta - ***magenta*** – 'm', zielony - ***green*** – 'g', itp.). Występuje on w trzecim argumencie na miejscu **pierwszej** litery.

Ponadto na miejscu **drugiej** litery trzeciego argumentu w opcji szczegółowej pomocy przedstawione są liczne możliwe wartości atrybutu oznaczenia *symbolicznego* ('^' lub '>' lub '<' – symbole trójkątów, 'p' – ***pentagram*** – gwiazdka pięciokątna, 'h' – ***hexagon*** – gwiazdka sześciokątna, 'd' – ***diamond*** – romb, '*' – gwiazdka właściwa, 'o' – kółko, '+' – krzyżyk, itp.).

Ostatecznie, na **trzecim** miejscu, lub na **drugim** miejscu ciągu literowego, (jeśli nie występują powyższe oznaczenia symboliczne), może wyjątkowo pojawić się *oznaczenie linii* ('-' – linii ciągłej, ' - - ' – linii przerywanej, ': ' – linii kropkowanej, '-.' – linii na przemian kropkowanej i ciągłej, itp.).

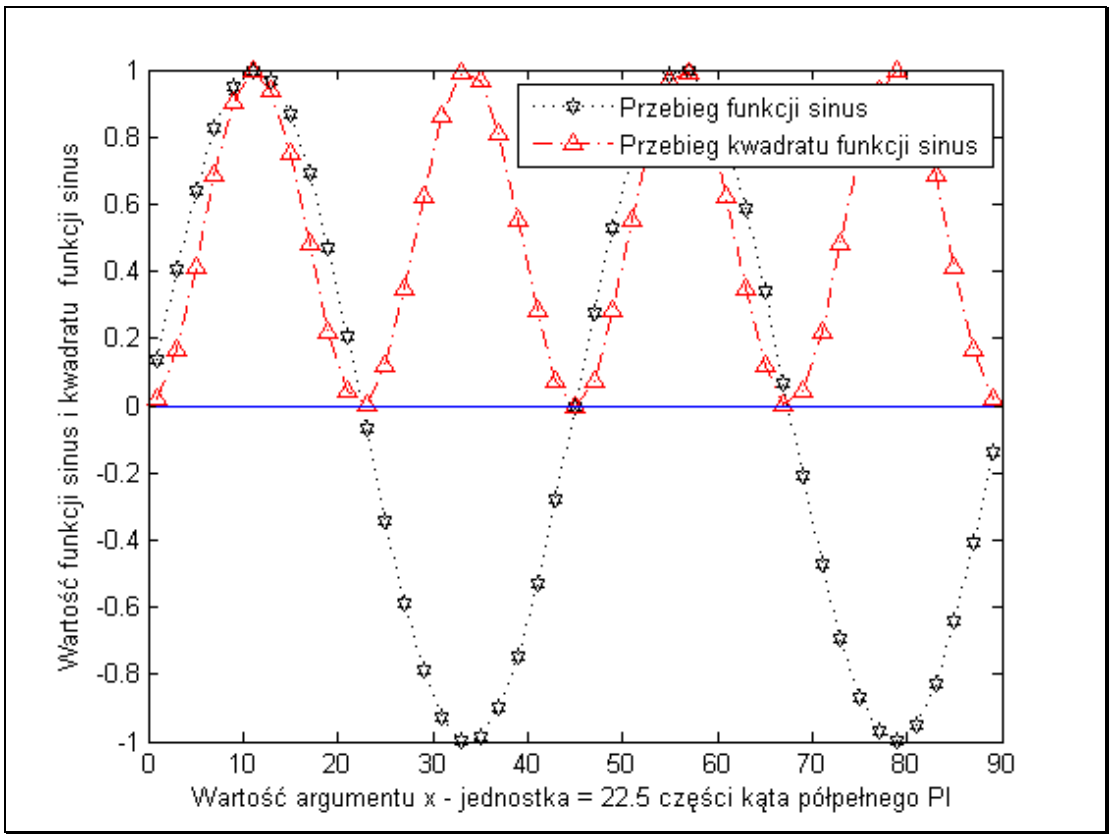
```
>> help plot
...
..
.% poniżej przytoczono cząstkowy listing opcji szczegółowej pomocy polecenia plot%
          b     blue          .     point              -     solid
          g     green         o     circle             :     dotted
          r     red           x     x-mark             -.    dashdot
          c     cyan          +     plus               --    dashed
          m     magenta       *     star           (none)    no line
          y     yellow        s     square
          k     black         d     diamond
```

```
w     white          v     triangle (down)
                     ^     triangle (up)
                     <     triangle (left)
                     >     triangle (right)
                     p     pentagram
                     h     hexagram
```

Opcje na pozycji litery *drugiej* i *trzeciej* mogą *jednocześnie* wystąpić wówczas, gdy oprócz przeprowadzenia linii przez punkty pomiarowe, tj. w sposób sugerujący ciągłość funkcji pomiędzy punktami pomiarowymi, występuje również konieczność zaznaczenia *dyskretnego* charakteru danych.

A oto inny przykład przedstawienia danych, wraz z opisem osi ***X*** i ***Y*** (polecenia: ***labelX*** oraz ***labelY***) oraz opisem w postaci legendy (polecenie - **legend**):

```
c = 1:2:90;
d = sin(c.*2*pi/45);
plot(c,d,'kh:');
hold on;
plot(c,d.^2,'r^-.');
legend('Przebieg funkcji sinus','Przebieg kwadratu funkcji sinus');
xlabel('Wartość argumentu x - 22.5 części kąta półpełnego PI');
xlabel('Wartość argumentu x - jednostka = 22.5 części kąta półpełnego PI');
ylabel('Wartość funkcji sinus i kwadratu  funkcji sinus');
e = zeros(1,45);
plot(c,e,'b-');
hold off;
```



*Rys. 2 Wydruk **funkcji sinus** na czarno (linią czarną kropkowaną oraz symbolami pięciokątnej gwiazdki) oraz **kwadratu funkcji sinus** (linią czerwoną przerywaną i symbolami trójkąta równoramiennego skierowanego w górę), ·jak również linii zerowej, tj. zgodnej z przebiegiem osi X (wykreślonej linią ciągłą niebieską)*

Omawiane szeregi danych, ujęte na wykresach nie służą jedynie ułatwieniom w wizualizacji. Oto definicje szeregów ***a*** i ***b***:

```
a =
     1     2     3     4     5     6     7     8     9    10
b =
     2     4     6     8    10    12    14    16    18    20,
```

a iloczyn skalarny wiersza danych ***a*** przez wiersz danych ***b***, wymaga transponowania danej ***b*** do wektora danych:

```
wynik_iloczynu_skalarnego = a*b'
wynik_iloczynu_skalarnego =
   770
```

Jednakże, transponowanie pierwszego wiersza danych ***a***, po czym realizacja mnożenia jego zawartości przez wektor ***b***, skutkuje powstaniem macierzy (w operacji mnożenia, nadal według reguł rachunku macierzowego):

```
wynik_iloczynu_dwoch_wektorow_A=a'*b

wynik_iloczynu_dwoch_wektorow_A =

     2     4     6     8    10    12    14    16    18    20
     4     8    12    16    20    24    28    32    36    40
     6    12    18    24    30    36    42    48    54    60
     8    16    24    32    40    48    56    64    72    80
    10    20    30    40    50    60    70    80    90   100
    12    24    36    48    60    72    84    96   108   120
    14    28    42    56    70    84    98   112   126   140
    16    32    48    64    80    96   112   128   144   160
    18    36    54    72    90   108   126   144   162   180
    20    40    60    80   100   120   140   160   180   200
```

W powyższej macierzy każdy z wierszy jest w istocie zawartością wiersza danych ***b*** przemnożonego przez odpowiedni element z wiersza danych ***a***.

W środowisku Matlab, istnieje możliwość wywoływania poleceń tworzenia blokowego danych – kolumn, wierszy, macierzy 2D oraz macierzy wielowymiarowych. Spośród elementarnych poleceń należy wymienić tworzenie macierzy zer, macierzy jedynek (macierzy kwadratowych domyślnie, jeśli tylko podany został jeden argument, tj. jeden wymiar takiej macierzy):

```
A = zeros(3)
A =
     0     0     0
     0     0     0
     0     0     0

B = ones(3)
B =
     1     1     1
     1     1     1
     1     1     1

C = ones(3,6)
C =
     1     1     1     1     1     1
     1     1     1     1     1     1
     1     1     1     1     1     1

D = eye(3)
D =
     1     0     0
     0     1     0
     0     0     1
```

Poniżej przedstawiono dodawanie macierzy (realizowane *de facto* tak łatwo i naturalnie, jak dodawanie skalarnych wartości):

```
E = B + D
E =
     2     1     1
     1     2     1
     1     1     2
```

Następnie przedstawiono odwracanie macierzy (wg rachunku macierzowego algebry liniowej) oraz odwracanie **po-elementowe** macierzy (każdy z elementów macierzy jest odwracany z osobna) (są to przykłady z zajęć laboratoryjnych):

```
inv(E) %1/E - składnia postrzegana jako poprawna do niższych wersjach Matlab %
ans =
    0.7500   -0.2500   -0.2500
   -0.2500    0.7500   -0.2500
   -0.2500   -0.2500    0.7500

1./E
ans =
    0.5000    1.0000    1.0000
    1.0000    0.5000    1.0000
    1.0000    1.0000    0.5000
```

Ostatecznie przedstawiono (dość użyteczną) opcję tworzenia macierzy z niezerowymi elementami na diagonalnej tej macierzy. Elementy diagonalnej, jak i rozmiar tworzonej kwadratowej macierzy są określane na podstawie przekazywanego w argumencie wejściowym wektora danych (tutaj przykładowo jest to wektor: [1 2 3]):

```
F = diag([1 2 3])
F =
     1     0     0
     0     2     0
     0     0     3
```

W szczególności przedstawiono macierz ***G*** utworzoną, jako sumę macierzy jedynek oraz macierzy o elementach niezerowych na głównej diagonalnej, utworzonych z wektora [1 2 3]:

```
G = ones(3) + diag([1 2 3])
G =
     2     1     1
     1     3     1
     1     1     4
```

Dokonano potęgowania ***po-elementowego*** macierzy **G**(elementy podniesione do potęgi drugiej, każdy z osobna)**:**

```
G.^2
ans =
     4     1     1
     1     9     1
     1     1    16
```

Ponadto, dokonano potęgowania ***po-elementowego*** macierzy **G** (elementy podniesione do potęgi trzeciej, każdy z osobna)**:**

```
G.^3

ans =

     8     1     1
     1    27     1
     1     1    64
```

Ostatecznie dokonano potęgowania macierzy ***G*** **(**według reguł rachunku macierzowego) (są to przykłady zajęciowe):

```
G^2
ans =
     6     6     7
     6    11     8
     7     8    18
```

Ponadto, spośród predefiniowanych funkcji blokowego tworzenia danych, przytoczono jeszcze wywołanie ***magic*** – tworzenia magicznego kwadratu Chińskiego, kwadratu liczb całkowitych, których suma na kierunku poziomym, pionowym, jak i przekątnych jest zawsze stała:

```
M=magic(5)
M =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

Dane macierzy ***M*** posłużyły do zademonstrowania możliwości blokowego filtrowania danych oraz zastosowań operatora wyliczania, oznaczonego symbolem dwukropka: '**:**'. Między innymi dysponując danymi blokowymi (macierzami 2D lub macierzami wielowymiarowymi) możliwe jest arbitralne filtrowanie danych.

Na przykład: wyliczenie danych macierzy ***M*** i przekazanie do zmiennej kolumnowej (wektorowej) ***MM*** tylko tych elementów, które są większe od 10, realizowane jest w następujący sposób:

```
MM = M(M>10)
MM =
    17
    23
    11
    24
    12
    18
    13
    19
    25
    14
    20
    21
    15
    16
    22
```

Natomiast wydzielenie danych pierwszej kolumny wymaga wykorzystania operatora dwukropka, w miejsce pierwszego wymiaru, w realizacji operacji wyliczenia wszystkich wierszy, przy zadanej (ustalonej) pierwszej kolumnie:

```
MK1 = M(:,1)

MK1 =

    17
    23
     4
    10
    11
```

Analogicznie, wydzielenie danych pierwszego wiersza wymaga wykorzystania operatora dwukropka w miejsce drugiego wymiaru, w operacji wyliczenia wszystkich kolumn przy pierwszym zadanym (ustalonym) wierszu:

```
>> Mw1 = M(1,:)

Mw1 =

    17    24     1     8    15
```

Operator wyliczenia można również używać równolegle, na przykład: w ograniczonym zakresie (z ustaloną dolną i górną granicą pierwszego wymiaru) względem pierwszego wymiaru oznaczającego liczbę wierszy oraz ponadto w całkowitym wyliczeniu wszystkich elementów w drugim wymiarze. Stąd poniżej wyliczono elementy drugiego i trzeciego wiersza:

```
>> Mw23 = M(2:3,:)

Mw23 =

    23     5     7    14    16
     4     6    13    20    22
```

W szczególności wykorzystanie uzupełniającego operatora ***end*** , razem z operatorem dwukropka '**:** ', umożliwia skopiowanie wnętrza macierzy ***M,*** tj. z pominięciem obrzeży:

```
M2x2 = M(2:end-1,2:end-1)

M2x2 =

     5     7    14
     6    13    20
    12    19    21
```

Ostatecznie operator wyliczenia '**:**', można stosować do całkowitego wyliczenia wektora (danych jedno-wymiarowych), jak i do macierzy dowolnego wymiaru (np.: macierzy 2D lub macierzy 3D), w postaci kolumny danych. Przy czym dane w przypadku macierzy 2D są wyliczane kolejno kolumnowo z lewa na prawo:

```
M(:)

ans =

    17
    23
     4
    10
    11
    24
     5
     6
    12
    18
     1
     7
    13
    19
    25
     8
    14
    20
    21
     2
    15
    16
    22
     3
     9
```

Oprócz, operatorów wyliczania '**:**', oraz operatora końcowego zakresu '***end***', w manipulacjach na danych macierzy 2D istotna jest również umiejętność posługiwania się operatorem '**[ ]**' , tj. operatorem konkatenacji (ang. *concatenation*), to znaczy operatorem sklejania, lub inaczej mówiąc operatorem łączenia pojedynczych liczb, wektorów, wierszy, lub podbloków macierzowych danych w większe agregaty danych.

Przypuśćmy, że mamy do dyspozycji zdefiniowaną macierz ***A*** zer oraz ponadto zdefiniowaną macierz ***B*** jedynek:

```
A =
```

```
     0     0     0
     0     0     0
     0     0     0

B

B =

     1     1     1
     1     1     1
     1     1     1
```

Przykładowe złożenie (sklejenie) tych dwóch macierzy poziomo, w kolejności: najpierw macierz ***A***, a następnie macierz ***B,*** oraz na odwrót wygląda następująco:

```
AB = [A B]

AB =

     0     0     0     1     1     1
     0     0     0     1     1     1
     0     0     0     1     1     1

BA = [B A]

BA =

     1     1     1     0     0     0
     1     1     1     0     0     0
     1     1     1     0     0     0
```

Jednakże sklejenie macierzy ***AB*** oraz ***BA*** pionowo (celem uzyskania macierzy 6 na 6 elementów) wymaga dodatkowo wykorzystania symbolu (tutaj występującego w roli operatora) średnika '**;**'(!). Jest to wówczas operator łamania ciągu danych o jednakowym wymiarze drugim (tj. liczbie elementów w wierszach danych) pionowo(tj. składania pionowego):

```
ABBA = [AB; BA]

ABBA =

     0     0     0     1     1     1
     0     0     0     1     1     1
     0     0     0     1     1     1
     1     1     1     0     0     0
     1     1     1     0     0     0
     1     1     1     0     0     0
```

W powstałej macierzy ***ABBA*** występują elementy niezerowe na… przeciw-diagonalnej, jak również w niektórych okolicach około przeciw-diagolnalnych. Celem uzyskania położenia tych elementów w okolicach głównej diagonalnej, stworzoną macierz ***ABBA*** należy obrócić przeciwnie do ruchu wskazówek zegara (tj. zgodnie z dodatnim kierunkiem kąta skierowanego na płaszczyźnie XY):

```
DIAG = rot90(ABBA)

DIAG =

     1     1     1     0     0     0
     1     1     1     0     0     0
     1     1     1     0     0     0
     0     0     0     1     1     1
     0     0     0     1     1     1
     0     0     0     1     1     1
```

Oprócz obrotu macierzy (dodajmy, macierzy dowolnej 2D, tj. prostokątnej, a w szczególności kwadratowej), istnieje możliwość wykorzystania poleceń dokonujących odbicia w symetrii względem osi poziomej (ang. *flip-up-down*):

```
DIAG_SYMETRIA_OSI_POZIOMEJ = flipud(DIAG)

DIAG_SYMETRIA_OSI_POZIOMEJ =

     0     0     0     1     1     1
     0     0     0     1     1     1
     0     0     0     1     1     1
     1     1     1     0     0     0
     1     1     1     0     0     0
     1     1     1     0     0     0
```

Ponadto istnieje możliwość wykorzystania analogicznego polecenia, realizującego odbicie macierzy w symetrii względem osi pionowej (polecenie ***fliplr*** – ang. *flip-left-right*).

W nawigacji, po zestawie plików i katalogów podrzędnych w bieżącej lokalizacji (nawiasem mówiąc, bieżąca ścieżka katalogu bieżącego wskazywana jest przez polecenie ***pwd***) użyteczne może być wywołanie polecenia z konsoli emulatora środowiska DOS. Są to polecenia zawsze poprzedzane znakiem wykrzyknika '***!***', a formatowane w kolorze czcionki, dla całej linijki polecenia, **na jasno-brązowo**.

Poniżej przytoczono DOS-owe polecenie (***dir /ad***) wyliczania tylko i wyłącznie podkatalogów:

```
!dir /ad
 Wolumin w stacji C to Cs
 Numer seryjny woluminu: C612-D617

 Katalog: c:\MatlabV7p6\work

2011-11-06  18:53    <DIR>          .
2011-11-06  18:53    <DIR>          ..
2011-11-03  18:40    <DIR>          AISO3XI11
2011-11-06  16:25    <DIR>          AiSOss
               0 plik(˙w)                   0 bajt˙w
               4 katalog(˙w)   2˙531˙069˙952 bajt˙w wolnych
```

Natomiast, poniżej przytoczono polecenie DOS-owe (***dir /os***) listujące pliki o rozszerzeniu ****.m***, w kolejności wzrastającej rozmiaru tych plików systemowych:

```
!dir /os *.m
 Wolumin w stacji C to Cs
 Numer seryjny woluminu: C612-D617

 Katalog: c:\MatlabV7p6\work

2011-11-02  21:30               161 silnia_for.m
2011-11-02  21:32               204 silnia_while.m
2011-11-02  21:30               232 silnia_for_wstecz.m
2011-11-02  21:36               273 silnia_while_wstecz.m
2011-11-02  21:41               292 silnia_for_rek.m
2011-11-02  22:12               333 silnia_rek.m
2011-11-02  21:44               339 silnia_while_rek.m
2011-07-19  19:45               889 Smatrix2angles.m
2011-07-25  08:07             1˙272 imwrite_gcf.m
2011-07-06  22:04             1˙455 surflgray.m
2011-07-19  19:47             1˙594 surlm.m
2011-07-06  19:28             2˙308 grad_bsd_sphere.m
2011-11-02  22:25             2˙804 wydawaj_sume_pieniedzy.m
2011-07-10  11:37             8˙172 loader.m
              15 plik(˙w)           122˙483 bajt˙w
               0 katalog(˙w)   1˙728˙153˙134 bajt˙w wolnych
```

Ostatecznie, bazując na wcześniej zdefiniowanej macierzy ***M*** o rozmiarze 5 na 5 elementów (zdefiniowanej z pomocą polecenia-funkcji ***magic***), warto przytoczyć operację blokową realizacji operacji przypisania dowolnej wartości (na przykład: wartości zerowej) elementom większym w tej macierzy ***M*** od wartości przykładowo równej 10:

```
M=magic(5)

M =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

M_mniejsze_niz_10 = M

M_mniejsze_niz_10(M_mniejsze_niz_10>10)=0

M_mniejsze_niz_10 =

     0     0     1     8     0
     0     5     7     0     0
     4     6     0     0     0
    10     0     0     0     3
     0     0     0     2     9
```

W tym celu skopiowano macierz ***M*** do macierzy ***M_mniejsze_niz_10***, a następnie wykorzystując operator ‘**( )’,** blokowo porównano każdy w elementów (jakby w blokowej operacji, jednak każdy z elementów z osobna!) z wartością 10. W przypadku stwierdzenia prawdziwości warunku: ***M_mniejsze_niz_10 > 10*** w miejsce elementu o wartości większej niż 10 wpisywane jest zero.

Podsumowując przegląd składni i poleceń (dość pobieżny przegląd, i o tyleż przykładowy, co rozwlekły, choć być może wystarczająco czytelny, w przeciwieństwie do zestawienia tabelarycznego składni i poleceń środowiska Matlab) należy wspomnieć o możliwości przytoczenia opcji szczegółowej pomocy, względem na przykład operatora ‘*****’:

```
>> help *
  Operators and special characters.
```

```
  Arithmetic operators.
    plus       - Plus                                +    % dodawanie dwuargumentowe
    uplus      - Unary plus                          +    % dodawanie jednoargumentowe
    minus      - Minus                               -    % odejmowanie dwuargumentowe
    uminus     - Unary minus                         -    % odejmowanie jednoargumentowe
    mtimes     - Matrix multiply                     *    % mnożenie macierzowe(wg.algebry liniowej)
    times      - Array multiply                     .*    % mnożenie po-elementowe
    mpower     - Matrix power                        ^    % potęgowanie macierzowe(wg. algebry lin.)
    power      - Array power                        .^    % potęgowanie po-elementowe
    mldivide   - Backslash or left matrix divide     \    % dzielenie lewostronne (mac. i skalarne)
    mrdivide   - Slash or right matrix divide        /    % dzielenie prawostronne (mac. i skalarne)
    ldivide    - Left array divide                  .\    % dzielenie lewostronne po-elementowe
    rdivide    - Right array divide                 ./    % dzielenie prawostronne po-elementowe
    idivide    - Integer division with rounding option.  % dzielenie całkowito-liczbowe
    kron       - Kronecker tensor product                % operator Kronecker'a

  Relational operators.
    eq         - Equal                              ==     % operator relacji równości(porównania)
    ne         - Not equal                          ~=     % operator różności (braku równości)
    lt         - Less than                           <     % operator relacji mniejszości
    gt         - Greater than                        >     % operator relacji większości
    le         - Less than or equal                 <=     % operator nieostrej relacji mniejszości
    ge         - Greater than or equal              >=     % operator nieostrej relacji większości

. . .
. .
. % tutaj następuje cząstkowe przytaczanie listingu

 Special characters.                                      % znaki-operatory specjalnego zastosowania
    colon      - Colon                               :    % operator wyliczenia (np.wnętrza macierzy)
    paren      - Parentheses and subscripting       ( )   % operator indeksowania elementów tablicy
    paren      - Brackets                           [ ]   % domyślnie: operator poziomego łączenia
    paren      - Braces and subscripting            { }
    punct      - Function handle creation            @    % uchwyt funkcji
    punct      - Decimal point                       .    % kropka - operator dostępu do pól structur
    punct      - Structure field access              .    % kropka - rówież: numeryczna kropka
    punct      - Parent directory                    ..   % wskazanie (w poleceniu) bieżącego katalg.
    punct      - Continuation                        ...  % kontynuacja linii polecenia w nast. linii
    punct      - Separator                           ,    % ogólnie - znak separatora elementów
    punct      - Semicolon                           ;    % średnik
    punct      - Comment                             %    % symbol łańcucha komentarza tekstowego
    punct      - Invoke operating system command     !    % tutaj: znak poprzedzajacy wyw. Pol. Emu. DOS
    punct      - Assignment                          =     % operator przypisania
    punct      - Quote                               '     % symbol inicjujący/kończący cytowanie
    transpose  - Transpose                          .'     % również: operator transpozycji macierzy
    ctranspose - Complex conjugate transpose         '     % również: op. transpoz. mac. lczb. zesp.
    horzcat    - Horizontal concatenation           [,]    % operator poziomego łączenia danych
    vertcat    - Vertical concatenation             [;]    % operator pionowego łączenia danych
    subsasgn   - Subscripted assignment          ( ),{ },
    subsref    - Subscripted reference           ( ),{ },
    subsindex  - Subscript index
    metaclass  - Metaclass for MATLAB class          ?

  Bitwise operators.                                      % bitowe operatory logiczne..
    bitand     - Bit-wise AND.
    bitcmp     - Complement bits.
    bitor      - Bit-wise OR.
    bitmax     - Maximum floating point integer.
    bitxor     - Bit-wise XOR.
    bitset     - Set bit.
    bitget     - Get bit.
    bitshift   - Bit-wise shift.

. . .
. .
. % poniżej następuje cząstkowe przytaczanie listingu

  See also arith, relop, slash, function_handle.
```

W powyższym zestawieniu operatorów i ich znaczeń, listing opcji szczegółowej pomocy został uzupełniony o komentarze w j. polskim, o czcionce sformatowanej **kolorem ciemno-zielonym**. Pośród nich skomentowano głównie operatory wyliczeń, konkatenacji (łączenia danych) oraz operatory rachunku macierzowego i operatory działań po-elementowych, omówione pokrótce powyżej w tym dokumencie.

Dalsze omówienia poleceń, składni, wywołań predefiniowanych funkcji, jak na przykład:

-rozwinięcia szeregów liczb z użyciem funkcji typu ***linspace*** lub ***logspace***,

-generacja siatki 2D współrzędnych z wykorzystaniem funkcji ***meshgrid***,

- wykreślanie danych (o opisie parametrycznym) z wykorzystaniem funkcji ***plot3***,

- definicja palet kolorów z wykorzystaniem funkcji ***colormap***, itp…

wymagają z pewnością odrębnego dokumentu omówień w bardziej szczegółowym temacie zastosowań.

Poniżej przedstawione będą czynności, składnia definicyjna, jak również przykładowe struktury bloków decyzyjnych, w tworzeniu funkcji – definicji ***m-skryptów***.

Definiowanie złożonego ciągu operacji logicznych i arytmetycznych zazwyczaj wymaga wykorzystania edytora m-skryptów. W edycji takiego m-skryptu, uruchamianej poleceniem:

```
edit funkcja_uzytkownika
```

tworzony jest nowy plik o nazwie ***funkcja_uzytkownika.m*** (o domyślnym rozszerzeniu *.m).

Jeśli decydujemy się na definicję odrębnego bloku instrukcji i poleceń, z listą argumentów wejściowych i wyjściowych oraz reprezentatywną nazwą wywołania (w tym wypadku nazwą: ***funkcja_użytkownika***) wymagany jest nagłówek definicyjny funkcji. Przy czym lista argumentów wyjściowych podawana jest w obejmujących nawiasach kwadratowych po słowie kluczowym ***function***. Natomiast po znaku równości podawana jest nazwa formalna funkcji (najlepiej zgodna z nazwą fizycznie istniejącego na dysku pliku systemowego definicji, lecz bez rozszerzenia *.*m*).

Ostatecznie za nazwą formalną funkcji, w nawiasach okrągłych podawana jest lista argumentów wejściowych. Ciąg argumentów zarówno: listy wejściowej, jak i listy wyjściowej argumentów, jest przedzielany przecinkiem, (jeśli argumentów jest więcej niż jeden). Przy czym w definicji, jak i w wywołaniu, jeśli lista argumentów wyjściowych jest jedno-elementowa, nie ma potrzeby przytaczać ujmujących ją nawiasów kwadratowych.

```
function [sn_for,tab_sn_for] = silnia_for(n)
```

Ponadto, w jednym fizycznie istniejącym pliku systemowym na dysku, służącym, jako definicja m-skryptu, może występować wiele nagłówków definicyjnych funkcji – wraz z ich definicjami. Wówczas, każda następna definicja (następująca po pierwszej definicji zasadniczej w pliku ****.m***) ma charakter wyłącznie lokalny. Mogą być wykorzystywane te definicje, jako wywołania funkcji podrzędnych, względem funkcji nadrzędnej. Jednak plusem (czy też czasem wadą) takiego rozwiązania jest fakt, że te następne nagłówki definicyjne przesłaniają definicje tak samo brzmiących funkcji, a zdefiniowanych w oddzielnych plikach systemowych w tym samym katalogu roboczym.

Poniżej przedstawiono podstawowy zarys funkcji naliczania wartości silni, z wykorzystaniem pętli ***for***:

```
function [sn_for,tab_sn_for] = silnia_for(n)
sn_for = 1;
tab_sn_for = [];
for i = 1:1:n
    sn_for = sn_for * i;
    tab_sn_for = [tab_sn_for sn_for];
end;
```

Zmienna ***sn_for*** służy do przechowywania cząstkowego wyniku naliczanej silni. Rzecz jasna z uwagi na iloczynowy charakter operacji naliczania silni, elementem obojętnym w tej operacji jest wartość 1. Stąd, zmienna ***sn_for***, przed pierwszym wejściem przebiegu funkcji (wejściem inicjującym licznik pętli powtarzającej ***for***, tj. zmienną ***i***) jest ustawiona na wartość 1.

```
sn_for = 1;
```

Analogicznie, celem horyzontalnego przechowywania wyników cząstkowych w naliczaniu silni wartością pustą zainicjowano zmienną ***tab_sn_for*** (tabelaryczna zmienna przechowywania wyników cząstkowych).

```
tab_sn_for = [];
```

W pętli ***for,*** tj. w jej wnętrzu definicyjnym następuje kolejne przemnażanie poprzedniej wartości ***sn_tab*** przez zwiększaną z iteracji na iterację wartość licznika iteracji ***i***:

```
    sn_for = sn_for * i;
```

Natomiast, do poprzedniej wartości tabeli wyników cząstkowych w zmiennej ***tab_sn_for***, dołączane są z iteracji na iterację kolejne wyniki cząstkowe z użyciem operatora poziomej konkatenacji:

```
    tab_sn_for = [tab_sn_for sn_for];
```

Powyższa wersja funkcji może być alternatywnie zrealizowana z wykorzystaniem pętli powtarzającej ***while***:

```
function [sn_while,tab_sn_while] = silnia_while(n)
i = 1;
sn_while = 1;
tab_sn_while = [];
while i <= n
    sn_while = sn_while * i;
    tab_sn_while = [tab_sn_while sn_while];
    i = i + 1;
end;
```

W powyższej definicji funkcji naliczania silni, dodatkowo jest wymagane jest inicjowanie licznika ***i*** kolejnych iteracji pętli ***while***:

```
i = 1;
```

Ponadto we wnętrzu definicyjnym pętli ***while*** wymagane jest zwiększanie wartości zmiennej iteracyjnej ***i***:

```
    i = i + 1;
```

Z kolei pętla ***for*** może być zastosowana we wstecznym naliczaniu silni. Jedynie pośrednie wyniki cząstkowe naliczanej silni znacząco się różnią od wyników cząstkowych naliczanych z prostym zwiększaniem licznika pętli ***for***:

```
function [sn_for_wstecz,tab_sn_for_wstecz] = silnia_for_wstecz(n)
sn_for_wstecz = 1;
tab_sn_for_wstecz = [];
for i = n:-1:1
    sn_for_wstecz = sn_for_wstecz * i;
    tab_sn_for_wstecz = [tab_sn_for_wstecz sn_for_wstecz];
end;
```

W nagłówku definicyjnym pętli ***for*** występuje zmienna iteracyjna inicjowana szeregiem wartości naliczanych wstecznie od wartości zmiennej ***n*** do 1 włącznie z krokiem -1:

```
for i = n:-1:1
```

Poniżej przytoczono wyniki uruchomień funkcji ***silnia_for*** oraz funkcji ***silnia_for_wstecz***:

```
>> [sn,sn_tab]=silnia_for(5)
sn =
   120
sn_tab =
     1     2     6    24   120

>> [sn,sn_tab]=silnia_for_wstecz(5)
sn =
   120
sn_tab =
     5    20    60   120   120
```

Zarówno w przypadku definicji: ***silnia_for*** oraz ***silnia_while***, jak i definicji: ***silnia_for_wstecz*** oraz ***silnia_while_wstecz***, wartość zmiennej przechowującej cząstkowy wynik naliczanej silni ustawiana była na wartość 1. Natomiast, pierwsze przejście iteracyjne (iteracji inicjującej automatycznie zmienną ***i*** w pętli ***for***) w funkcji ***silnia_for_wstecz*** od razu ustawiało wartość tej zmiennej (nazywanej ***sn_for_wstecz*** w tej funkcji naliczania silni) na wartość najwyższą, tj. równej argumentowi naliczanej silni. Stąd wyniki cząstkowe są w dwóch powyższej przytaczanych wywołaniach funkcji: ***silnia_for*** oraz ***silnia_for_wstecz*** , diametralnie różne.

Analogicznie poniżej przedstawiono definicję funkcji ***silnia_while_wstecz***, z licznikiem iteracji o wartościach coraz mniejszych z iteracji z na iteracji.

```
function [sn_while_wstecz,tab_sn_while_wstecz] = silnia_while_wstecz(n)
i = n;
sn_while_wstecz = 1;
tab_sn_while_wstecz = [];
```

```
while i > 0
    sn_while_wstecz = sn_while_wstecz * i;
    tab_sn_while_wstecz = [tab_sn_while_wstecz sn_while_wstecz];
    i = i - 1;
end;
```

Przy czym wartość licznika iteracji jest tutaj ustawiana na wartość najwyższą, równą argumentowi silni:

```
i = n;
```

Natomiast, sam licznik ulega de-inkrementacji, tj. zmniejszania od jeden z iteracji na iteracje:

```
    i = i - 1;
```

Najciekawszą koncepcją budowy funkcji naliczania silni, jest wykorzystanie struktury rekurencyjnej, to znaczy pierwotnego zagłębiania się w kolejne pod-wywołania funkcji o tej samej treści definicyjnej, jednak z ostatecznym naliczaniem silni (w cząstkowych wartościach iloczynu) przy powrocie z dna rekurencyjnego wywołań tych funkcji.

Poniżej przedstawiono funkcję ***silnia_rek.m*** o rekurencyjnej strukturze definicyjnej:

```
function [sn_rek, tab_sn_rek] = silnia_rek(n)
%sn_rek = 1;
tab_sn_rek = [];
    if n == 1
     sn_rek = 1;
     tab_sn_rek = [1];
     disp(['Rekurencja najniższa nr: 1' ]);
     disp(['Wartość cząstkowa silni: 1, ustalona na samym dnie wywołań rekurencji pośrednich']);
     return;
    else
     disp(['Rekurencja pośrednia nr: ' num2str(n)]);
     [sni, tabi] = silnia_rek(n - 1);
     sn_rek = n * sni;
     disp(['Wartość cząstkowa silni: ' num2str(sn_rek) ' naliczana nawrotnie w iteracji: ' num2str(n)]);
     tab_sn_rek = [tabi sn_rek];
     disp(['Tabela wyników cząstkowych: [' num2str(tab_sn_rek) '] naliczana nawrotnie w iteracji: ' num2str(n)]);
    end;
```

W powyższej definicji o strukturze rekurencyjnej uderza po pierwsze brak pętli powtarzającej.

Nieco więcej światła na przebieg wywołania funkcji naliczania silni o strukturze definicyjnej może rzucić analiza wydruku – listingu wywołania funkcji, dla powiedzmy argumentu równego 5 (**na ciemno-zielono** oraz **ciemno-czerwono** zaznaczono komentarze własne, uzupełniające do poniższego listingu):

```
>> [sn,sntab]=silnia_rek(5)
Rekurencja pośrednia nr: 5 % pierwsze rekurencyjne wywołanie funkcji w funkcji
Rekurencja pośrednia nr: 4 % drugie rekurencyjne wywołanie funkcji w funkcji
Rekurencja pośrednia nr: 3 % trzecie rekurencyjne wywołanie funkcji w funkcji
Rekurencja pośrednia nr: 2 % czwarte rekurencyjne wywołanie funkcji w funkcji
Rekurencja najniższa nr: 1 % piąte rekurencyjne wywołanie, sięgające 'dno' rekurencji
Wartość cząstkowa silni: 1, ustalona na samym dnie wywołań rekurencji pośrednich
Wartość cząstkowa silni: 2 naliczana nawrotnie w iteracji: 2
Tabela wyników cząstkowych: [1  2] naliczana nawrotnie w iteracji: 2
Wartość cząstkowa silni: 6 naliczana nawrotnie w iteracji: 3
Tabela wyników cząstkowych: [1  2  6] naliczana nawrotnie w iteracji: 3
Wartość cząstkowa silni: 24 naliczana nawrotnie w iteracji: 4
Tabela wyników cząstkowych: [1   2   6  24] naliczana nawrotnie w iteracji: 4
Wartość cząstkowa silni: 120 naliczana nawrotnie w iteracji: 5
Tabela wyników cząstkowych: [1    2    6   24  120] naliczana nawrotnie w iteracji: 5
sn =
   120
sntab =
     1     2     6    24   120
```

Wynik naliczania silni, a nawet wyniki cząstkowe naliczania silni są identyczne, jak w przypadku wywołań funkcji naliczania silni (***silnia_for***, ***silnia_while***) z prostą (to jest wprzód) zmianą wartości zmiennej licznika iteracji.

Jednakże, pierwsze pięć przebiegów rekurencyjnych funkcji (rekurencyjnych, lecz nie iteracyjnych!), oparte jest o pięciokrotne zagłębianie, rekurencyjne wywoływanie funkcji w funkcji w obrębie bloku alternatywnego wykonania instrukcji warunkowej ***if***:

```
    else
```

```
     disp(['Rekurencja pośrednia nr: ' num2str(n)]);
     [sni, tabi] = silnia_rek(n - 1);
     sn_rek = n * sni;
     disp(['Wartość cząstkowa silni: ' num2str(sn_rek) ' naliczana nawrotnie w iteracji: ' num2str(n)]);
     tab_sn_rek = [tabi sn_rek];
    disp(['Tabela wyników cząstkowych: [' num2str(tab_sn_rek) '] naliczana nawrotnie w iteracji: ' num2str(n)]);
   end;
```

Konkretnie, rekurencyjne pięciokrotne wywoływanie funkcji w funkcji następuje w oparciu o linijkę definicyjną:

```
     [sni, tabi] = silnia_rek(n - 1);
```

Z kolei, '*na dnie*' wszystkich pięciu poziomów wywołań rekurencyjnych następuje istotny zwrot algorytmiczny. Mianowicie następuje unikatowe, (bo jednokrotne) wykonanie bloku zasadniczego (zamiast alternatywnego) instrukcji warunkowej ***if***:

```
    if n == 1
     sn_rek = 1;
     tab_sn_rek = [1];
. . .
. .% tutaj  następuje cząstkowy listing definicji funkcji silnia_rek.m
     return;
    else
```

W rzeczy samej następuje '*odbicie się*' rekurencyjnego przebiegu algorytmu od samego dna (od takiego niby parteru cztero-piętrowego budynku – parter jest najniższym, piątym poziomem kondygnacji). W tym momencie w obrębie powyższego bloku wykonania zasadniczego, instrukcji warunkowej ***if*** następuje przypisanie zmiennej cząstkowego naliczania silni wartości 1:

```
     sn_rek = 1;
     tab_sn_rek = [1];
```

Jak widać powyżej, oprócz tego nadano również wartość początkową zmiennej, tablicowego gromadzenia wyników cząstkowych ***tab_sn_rek***.

Od tego momentu następuje etap nawrotny, tj. powrotu z zagłębionych wywołań funkcji w funkcji, wraz z jednoczesnym rozpoczęciem właściwego etapu naliczania cząstkowych wartości iloczynów, skutkujących na końcu wyznaczeniem silni, przy zadanej wartości argumentu wejściowego – zmiennej ***n***. *De facto* właściwe naliczanie silni występuje w dwóch pod-linijkach bloku wykonań alternatywnych instrukcji warunkowej ***if***:

```
     [sni, tabi] = silnia_rek(n - 1);
     sn_rek = n * sni;
. . .%tutaj następuje cząstkowy listing definicji funkcji silnia_rek.m
     tab_sn_rek = [tabi sn_rek];
```

Reasumując: realizacja zadania naliczania silni w oparciu o strukturę rekurencyjnych wywołań (w miejsce struktury iteracyjnej opartej o pętle powtarzające: ***for*** lub ***while***) wydaje się z pozoru niezmiernie skomplikowana. Jednakże w realizacji zadań obliczeniowych znacznie bardziej złożonych zaznacza się znaczna oszczędność kodu (definicyjnego funkcji), względem standardowej definicji (w oparciu o iteracyjną definicję funkcji). Ponadto, przy zadaniach niezmiernie złożonych, zauważa się konieczność stosowania pewnego kompromisu pomiędzy: ilością pamięci operacyjnej oraz często: innych zasobów obliczeniowych (wymaganych w *rekurencyjnym* definiowaniu zadania), a objętością samego kodu definicyjnego (wymaganego w iteracyjnym *definiowaniu* zadania).

Bibliografia (w niektórych wypadkach jest możliwe znalezienie publikacji ***edycji CRC*** w darmowych zasobach Internetu):

[1] Wojciech Tarnowski: „*Symulacja i optymalizacja w Matlab'ie*", ISBN 83-87438-81-2, Wyd. i Druk: „Intergraf" S. C. Sopot, 2001 (uwaga – tylko rozdział wstępny – środowisko Matlab)

[2] Matlab v. 6.5 << *Manual* >> - podręcznik ***pdf*** na stronie http://www.mathworks.com

[3] Matlab v. 6.5 << *Image Processing Toolbox* >> - podręcznik ***pdf*** na stronie http://www.mathworks.com

[4] Patrick Marchand, O. Thomas Holland: "*Graphics and GUIs with Matlab*" 3rd Edition, Chapman & Hall, ***CRC Press Company***, 2003

[5] Andy H. Register: "*A Guide to Matlab Object-Oriented Programming*", ***Chapman & Hall CRC***, 2007

[6] Misza Kalechman: "*Practical Matlab Applications for Engineers*", ***CRC Press***, Taylor & Francis Group, 2009

[7] Peter J. Acklam: "*Matlab array manipulation tips and tricks*", http://home.online.no/~pjacklan, 18nth Oct. 2003

Przykłady zadań obliczeniowych zrealizowanych
w postaci funkcji – m-skryptów
zdefiniowane iteracyjnie oraz rekurencyjnie

Na użytek zajęć laboratoryjnych z przedmiotu ***AiSO*** (*Algorytmy i Systemy Obliczeniowe*) wytyczono kilka najbardziej przekrojowych i ilustracyjnych zadań algorytmicznych. Ponadto przedstawiono niektóre spośród rozwiązań tych zadań, w postaci zrealizowanych funkcji – m-skryptów w środowisku Matlab.

Po pierwsze: celem uzupełnienia przeglądu składni instrukcji warunkowego wyboru i przetwarzania przytoczono dwie krótkie funkcje: ***losuj_zgaduj.m*** (losuj kulę w urnie i zgadnij jej numer) oraz ***losuj_wybieraj.m*** (losuj kilka kul z urny, a następnie wyświetlaj numery kolejno wylosowanych kul).

```
function [jedna_kula] = losuj_zgaduj(n)
jedna_kula = randi(n,1,1);
ktora_kula = -1;
while ktora_kula ~= jedna_kula
  ktora_kula = input(['Zgadnij numer wylosowanej kuli z urny w zakresie od ' num2str(1) ' do ' num2str(n)]);
end;
    switch ktora_kula
        case 1
            disp(['Wylosowano kule numer 1']);
        case 2
            disp(['Wylosowano kule numer 2']);
        case 3
            disp(['Wylosowano kule numer 3']);
        case 4
            disp(['Wylosowano kule numer 4']);
        case 5
            disp(['Wylosowano kule numer 5']);
        case 6
            disp(['Wylosowano kule numer 6']);
        case 7
            disp(['Wylosowano kule numer 7']);
        case 8
            disp(['Wylosowano kule numer 8']);
        case 9
            disp(['Wylosowano kule numer 9']);
        case 10
            disp(['Wylosowano kule numer 10']);
        otherwise
            disp(['Wylosowano kule spoza zakresu <1-10>']);
    end;
```

Wylosowany numer kuli przechowywany w zmiennej ***jedna_kula*** (z zakresu obranego w argumencie powyższej funkcji), jest porównywany w relacji różności z wartością zmiennej ***ktora_kula,*** w warunku logicznym pętli powtarzającej ***while***.

```
while ktora_kula ~= jedna_kula
  ktora_kula = input(['Zgadnij numer wylosowanej kuli z urny w zakresie od ' num2str(1) ' do ' num2str(n)]);
end;
```

Jednak, gdy wartość zmiennej ***ktora_kula*** wprowadzona z klawiatury (poleceniem ***input***) zrówna się z wartością zmiennej ***jedna_kula***, następuje blok warunkowego wyboru, zaczynający się słowem kluczowym: ***switch***.

```
    switch ktora_kula
        case 1
            disp(['Wylosowano kule numer 1']);
        case 2
            disp(['Wylosowano kule numer 2']);
        case 3
. . .
. .
. % poniżej częściowo przytoczono listingu kodu funkcji
        case 10
            disp(['Wylosowano kule numer 10']);
        otherwise
            disp(['Wylosowano kule spoza zakresu <1-10>']);
    end;
```

Instrukcja warunkowego wielo-wyboru: ***switch …case…end***, (zaimplementowana zresztą w analogiczny sposób w większości języków programowania wyższego poziomu), oferuje sprawny wybór podbloku instrukcji przeznaczonych do wykonania, w oparciu o wartość zmiennej, następującej po słowie kluczowym ***switch***. Natomiast podbloki instrukcji przeznaczone do warunkowego wywołania, każdy z nich jest poprzedzony słowem kluczowym ***case*** oraz jedną z wartości, przyjmowanych przez zmienną, występującą właśnie po słowie kluczowym ***switch***.

Jednak w środowisku Matlab, opcja wielo-wyboru umożliwia bardziej elastyczne definiowanie warunków wyboru instrukcji. Na przykład: możliwe jest wprowadzanie wielu alternatywnych wartości, adekwatnych do kryterium wywołania **pojedynczego** podbloku instrukcji ***case*** (w tym również wartości numerycznych, znakowych, jak i łańcuchów znakowych).

Funkcja druga ***losuj_wybieraj.m*** dokonuje realizacji nieco innego zadania oraz w nieco bardziej bezpośredni sposób:

```
function [urna] = losuj_wybieraj(n)
urna = randi(10,n,1);
for i=1:n
    switch urna(i)
        case 1
            disp(['Wybrano kule numer 1']);
        case 2
            disp(['Wybrano kule numer 2']);
        case 3
            disp(['Wybrano kule numer 3']);
        case 4
            disp(['Wybrano kule numer 4']);
        case 5
            disp(['Wybrano kule numer 5']);
        case 6
            disp(['Wybrano kule numer 6']);
        case 7
            disp(['Wybrano kule numer 7']);
        case 8
            disp(['Wybrano kule numer 8']);
        case 9
            disp(['Wybrano kule numer 9']);
        case 10
            disp(['Wybrano kule numer 10']);
        otherwise
            disp(['Wybrano kule spoza zakresu <1-10>']);
    end;
end;
```

Mianowicie, po wylosowaniu wektora dziesięciu losowych liczb całkowitych (tutaj: liczb naturalnych), w pętli iteracyjnej ***for*** następuje wykonanie warunkowe bloku: ***switch…case ….case …end*** – wielo-wyboru, skutkujące w wielokrotnym wyświetleniu 10 numerów kul, wylosowanych z urny zawierającej ***n*** kul, ponumerowanych wg. szeregu liczb naturalnych.

Po drugie: za pomocą funkcji ***wydawaj_sume_pieniedzy.m***, zrealizowano wirtualną maszynę – automat (a może: automat-bankomat) do zakupu produktu, o pewnej określonej cenie (o wysokości ceny produktu, zadawanej w złotówkach, jako drugi argument tej funkcji). Celem poniższej funkcji jest wydanie reszty z zakupu tego produktu, przy określonej sumie pieniędzy (pieniędzy będących na koncie automatu-bankomatu, a może pieniędzy wpłaconych do tego automatu-bankomatu) o wysokości tej sumy pieniędzy, przekazanej w pierwszym argumencie wejściowym.

Przy czym poniższy '*wirtualny*' automat-bankomat powinien wypłacać z dokładnością do 1 grosza, wykorzystując nie tylko *pięć* dostępnych na rynku polskim nominałów banknotów (nominałów papierowego środka płatniczego), lecz również *aż dziewięć* dostępnych na rynku polskim nominałów bilonów (nominałów monetarnego środka płatniczego).

W definicji funkcji ***wydawaj_sume_pieniedzy.m*** zdefiniowano strukturę ***str***, o 5 polach papierowego środka płatniczego oraz ponadto o 9 polach monetarnego środka płatniczego (kropka jest rozdzielnikiem pola struktury o samej struktury, jako całości).

```
str.banknot200zl = ' 200 zł ';
str.banknot100zl = ' 100 zł ';
str.banknot050zl = '  50 zł ';
str.banknot020zl = '  20 zł ';
str.banknot010zl = '  10 zł ';
%--------
str.bilon5zl     = '   5 zł ';
str.bilon2zl     = '   2 zł ';
str.bilon1zl     = '   1 zł ';
str.bilon50gr    = '  50 gr ';
str.bilon20gr    = '  20 gr ';
str.bilon10gr    = '  10 gr ';
str.bilon05gr    = '   5 gr ';
str.bilon02gr    = '   2 gr ';
str.bilon01gr    = '   1 gr ';
```

W definicji funkcji zastosowano globalną pętlę powtarzającą warunkowo ***while***, oraz ponadto pięć pętli podrzędnych i dziewięć pętli podrzędnych, wszystkie one zdefiniowane, jako pętle powtarzające warunkowo ***while***, odnoszące się odpowiednio do wydawania reszty z banknotach (wg 5 nominałów) oraz do wydawania reszty w bilonie (wg 9 dostępnych nominałów).

Zastosowanie w miejsce 14 podrzędnych pętli powtarzających ***while***, instrukcji warunkowych ***if*** (a jest to pierwsza myśl – inicjatywa w realizacji konstrukcji tej funkcji) powodowałaby, że w funkcji ***wydawaj_sume_pieniedzy.m***, istniałaby tylko możliwość, co najwyżej jedno-razowego '*odmierzania-wydawania z reszty*' każdego z nominałów, w jednokrotnym przebiegu pętli globalnej ***while***. Dodatkowo algorytm wydawania mógłby się przypadkowo '*zakleszczać*' na dość niestandardowo określonych pośrednich wynikach reszty, skutkując dość niespójnym i nieuzasadnionym działaniem.

```
function [reszty] =
wydawaj_sume_pieniedzy(suma,cena);
reszta = suma - cena;
disp(['Zapłacono: ' num2str(cena) ' zł,…
 z sumy pieniędzy: ' num2str(suma) ' zł']);
disp(['Do wydania pozostało: ' …
num2str(reszta) ' zł reszty']);
disp('-----------------------------------------');
%---------------------
str.banknot200zl = ' 200 zł ';
str.banknot100zl = ' 100 zł ';
str.banknot050zl = '  50 zł ';
str.banknot020zl = '  20 zł ';
str.banknot010zl = '  10 zł ';
%--------
str.bilon5zl     = '   5 zł ';
str.bilon2zl     = '   2 zł ';
str.bilon1zl     = '   1 zł ';
str.bilon50gr    = '  50 gr ';
str.bilon20gr    = '  20 gr ';
str.bilon10gr    = '  10 gr ';
str.bilon05gr    = '   5 gr ';
str.bilon02gr    = '   2 gr ';
str.bilon01gr    = '   1 gr ';
%---------------------
while (reszta > 0.01)
    %-----
    %banknoty
    %-----
    while (reszta - 200) >= 0
        reszta = reszta - 200;
        disp( [ 'Wydano: ' str.banknot200zl ] );
    end;
    %-----
    %-----
    while (reszta - 100) >= 0
        reszta = reszta - 100;
        disp( [ 'Wydano: ' str.banknot100zl ] );
    end;
    %-----
    %-----
    while (reszta - 50) >= 0
        reszta = reszta - 50;
        disp( [ 'Wydano: ' str.banknot050zl ] );
    end;
    %-----
    %-----
    while (reszta - 20) >= 0
        reszta = reszta - 20;
        disp( [ 'Wydano: ' str.banknot020zl ] );
    end;
    %-----
    %-----
    while (reszta - 10) >= 0
        reszta = reszta - 10;
        disp( [ 'Wydano: ' str.banknot010zl ] );
    end;
    %-----
    %bilony
    %-----
    while (reszta - 5) >= 0
        reszta = reszta - 5;
        disp( [ 'Wydano: ' str.bilon5zl ] );
    end;
    %-----
    %-----
    while (reszta - 2) >= 0
        reszta = reszta - 2;
        disp( [ 'Wydano: ' str.bilon2zl ] );
    end;
    %-----
        %-----
    while (reszta - 1) >= 0
        reszta = reszta - 1;
        disp( [ 'Wydano: ' str.bilon1zl ] );
    end;
    %-----
    %-----
    while (reszta - 0.5) >= 0
        reszta = reszta - 0.5;
        disp( [ 'Wydano: ' str.bilon50gr ] );
    end;
    %-----
    %-----
    while (reszta - 0.2) >= 0
        reszta = reszta - 0.2;
        disp( [ 'Wydano: ' str.bilon20gr ] );
    end;
    %-----
    %-----
    while (reszta - 0.1) >= 0
        reszta = reszta - 0.1;
        disp( [ 'Wydano: ' str.bilon10gr ] );
    end;
    %-----
    %-----
    while (reszta - 0.05) >= 0
        reszta = reszta - 0.05;
        disp( [ 'Wydano: ' str.bilon05gr ] );
    end;
    %-----
    %-----
    while (reszta - 0.02) >= 0
        reszta = reszta - 0.02;
        disp( [ 'Wydano: ' str.bilon02gr ] );
    end;
    %-----
    %-----
    while (reszta - 0.01) >= 0
        reszta = reszta - 0.01;
        disp( [ 'Wydano: ' str.bilon01gr ] );
    end;
    %-----
end;
%---------------------------
disp('-----------------------------------------------');
reszty = reszta;
disp( [ 'Do wydania zostało: ' …
     num2str(round(100*reszty)) ' groszy reszty'] );
%---------------------------
```

Powyżej przytoczono całość funkcji wydawania reszty, sformatowany dwukolumnowo. Poniżej przedstawiono wynik wywołania powyższej funkcji, przy zadanej sumie pieniędzy (200 zł), przeznaczonych do uiszczenia opłaty za produkt o cenie wynoszącej 131 zł i 78 groszy:

```
>> [grosik ] = wydawaj_sume_pieniedzy(200, 131.78);
Zapłacono: 131.78 zł, z sumy pieniędzy: 200 zł
Do wydania pozostało: 68.22 zł reszty
-----------------------------------------------
Wydano:   50 zł
Wydano:   10 zł
Wydano:    5 zł
Wydano:    2 zł
Wydano:    1 zł
Wydano:   20 gr
Wydano:    1 gr
-----------------------------------------------
Do wydania zostało: 1 groszy reszty
```

Po trzecie: ostatni przykład wykorzystania pętli powtarzającej ***while***, zaprezentowano dla zadania znajdywania pojedynczego pierwiastka funkcji y = f(x) (funkcji ciągłej, a najlepiej lokalnie monotonicznej, lub przynajmniej funkcji ciągłej o wartościach na krańcach zadanego wstępnie przedziału [a, b] o różnych znakach). Metoda bisekcji, lub metoda prostej dwusiecznej ma dość prostą regułę postępowania. Jest to metoda prostej siecznej, dzielącej przedział [a, b] na dwa podprzedziały, a następnie wybierającej ten podprzedział, w którym wartość funkcji na jej krańcach znowu będą miały różne znaki:

```
function [rts] = pierwiastek_metody_prostej_dwudzielnej(feval_str,intvls_ab,steps,tolr);
%---------------------
a = intvls_ab(1);   % pobierz z wiersza 2giego argumentu położenie punktu a na osi X%
b = intvls_ab(2);   % pobierz z wiersza 2giego argumentu położenie punktu b na osi Y%
%------------%rozwiń szereg argumentu X oraz linii zerowej (wzdłuż osi X) %
X = linspace(a,b,steps);% inaczej: X = a : (b-a)/(steps-1) : b;%%!%%
Z = 0.*X;
%---------------------
Y = feval(feval_str,X); % wyznacz funkcję o nazwie określonej w 1wszym argumencie %
figure,
plot(X,Y,'b--');        % wykreśl funkcję badaną na wystąpienie pierwistka%
hold on;                % wykres do dalszego uzupełnienia, - jego uchwyt wstrzymany%
plot(X,Z,'k-');         % wykreśl linię zerową (wzdłuż osi X)%
title(['Przebieg funkcji: ' feval_str ', w znajdywaniu pierw. metodą bisekcji']);
xlabel(['przebieg funkcji badanej - linia niebieska przerywana, linia bisekcji - linia czerw. przeryw.' ...
        ', bieżąca zmiana jednej z granic przedziału [a,b] - czarne romby, wart. funk. na krańcach bież.'...
        'przedz. [a,b] - czarne krzyżyki']);
ylabel(['Wartości funkcji: ' feval_str]);
%---------------------
errs = 2;              % ustaw błąd na przypadkową oraz znaczną wartość (przed wejściem do pętli)%
licznik = 0;           % ustaw licznik przebiegu pętli while na zero - wart. pocz. %
% początek pętli głównej - kryterium wyjścia- oceną marginesu błędu znajdywania pierwiastka funkcji %
while (errs >= tolr)    % dopóki błąd nie spadnie poniżej wymaganej toleracji, kontynuuj pętlę while %
    Ya = feval(feval_str,a); % oceń wartość funkcji na końcach przedziału [a,b] %
    disp(['Wartość początkowa a przedziału [a,b]: ' num2str(a) ' z wartością funkcji Y(a): ' num2str(Ya)]);
    Yb = feval(feval_str,b);
    disp(['Wartość początkowa b przedziału [a,b]: ' num2str(b) ' z wartością funkcji Y(b): ' num2str(Yb)]);
    %---
    plot(a,Ya,'rd');plot(b,Yb,'rd'); % oznacz wartości funkcji na krańcach przed. pocz. czerwn. rombami%
    ab  = linspace(a,b,steps);       % wyznacz szereg dziedziny w przedziale [a,b]%
    Yab = linspace(Ya,Yb,steps);     % wyznacz liniowy szereg przeciwdzieny w przedziale [a,b]%
    plot(ab,Yab,'r-.');              % wykreśl prostą - sieczną funkcji w przedziale [a,b]%
    %-
    rts = Ya*(b - a)/(Ya - Yb) + a; % z podobieństwa trójkątów znajdź przybliż. pierwiast. - przecięcie osi X %
    Yrts = feval(feval_str,rts);    % znajdź wartość funkcji w miejscu tego przybliż. pierwiast. %
    errs = abs(Yrts);               % jego wartość bezwględna, jest jednocześnie błędem znalezienia pierw. %
    plot(rts,0,'kd');            % wykreśl gwiazdką przybliżoną lokalizację pierwiastka (na osi X na czarno) %
    plot(rts,Yrts,'bd');         % wykreśl rombem przybliżoną lokalizację pierw. w przeciwdziedzinie %
    LV = linspace(0,Yrts,steps); % rozwin przeciwdziedzinę-współrzędne wys. linii pionowej
    plot(rts*ones(1,steps),LV,'k--'); % wykr. ln. pion., od punktu na osi X do wart.funk. w. punkt. na osi X
        %---
    if (Yrts > 0)                % jeśli w przeciwdziedzinie jest to wartość dodatnia %
      a = rts;
     Ya = Yrts;                  % zawężaj przedział [a,b] z lewej strony %
     disp(['Przedzial [a,b] zawężony od lewej strony (od strony punktu a)']);
     %---
    elseif (Yrts < 0);           % jeśli w przeciwdziedizenie jest to wartość ujemna %
    %---
      b  = rts;
     Yb = Yrts;                  % zawężaj przedział [a,b] w prawej strony %
     disp(['Przedzial [a,b] zawężony od prawej strony (od strony punktu  b)']);
    end;
    %---
     plot(a,Ya,'k+');plot(b,Yb,'k+');
    %---
    licznik = licznik + 1        % zwiększ licznik przebiegu  pętli o jeden%
end;
%-----
hold off;
```

Znając treść powyższego kodu, warto przeanalizować wyniki jednego w wywołań funkcji znajdywania pierwiastka funkcji o następującej poniższej definicji:

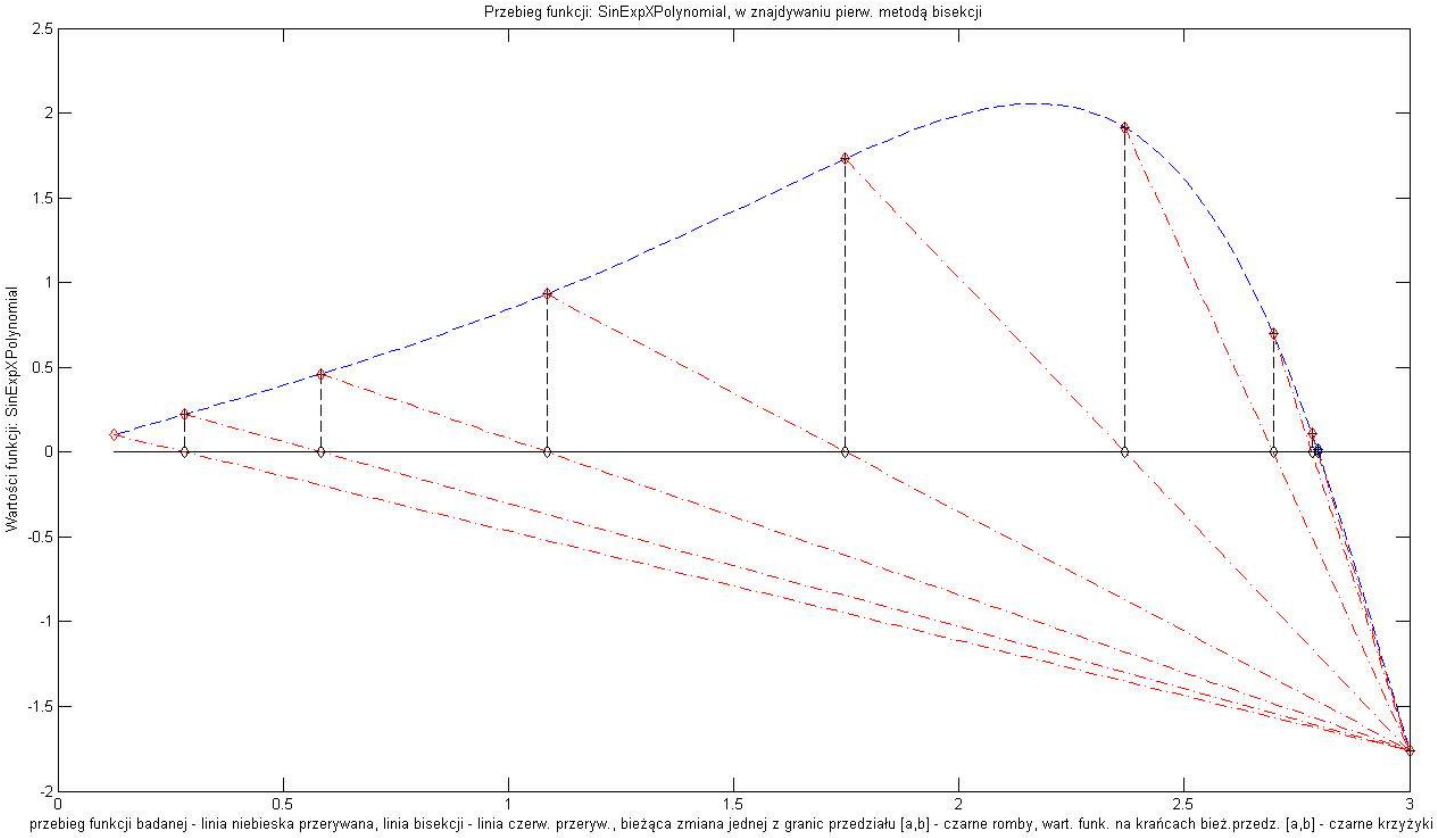
```
function [Yrow] = SinExpXPolynomial(X)
Yrow = X.*sin(exp(X.^(4/3)-X));
```

Poniżej przedstawiono rezultat wywołania funkcji znajdywania pierwiastka dla zależności, jak powyżej, w przedziale: [1/8 3.0], w paru zadanych krokach wizualizacji funkcji w bieżącym przedziale oraz bezwzględnej wartości tolerancji wynoszącej 0.05:

```
>>  [rts] = pierwiastek_metody_prostej_dwudzielnej('SinExpXPolynomial',[0.125 3.000],100,0.05);

Wartość początkowa a przedziału [a,b]: 0.125 z wartością funkcji Y(a): 0.1009
```

```
Wartość początkowa b przedziału [a,b]: 3 z wartością funkcji Y(b): -1.7606
Przedzial [a,b] zawężony od lewej strony (od strony punktu a)

licznik =

     1

Wartość początkowa a przedziału [a,b]: 0.28084 z wartością funkcji Y(a): 0.22131
Wartość początkowa b przedziału [a,b]: 3 z wartością funkcji Y(b): -1.7606
Przedzial [a,b] zawężony od lewej strony (od strony punktu a)

licznik =

     2

Wartość początkowa a przedziału [a,b]: 0.58448 z wartością funkcji Y(a): 0.46096
Wartość początkowa b przedziału [a,b]: 3 z wartością funkcji Y(b): -1.7606
Przedzial [a,b] zawężony od lewej strony (od strony punktu a)

licznik =

     3

Wartość początkowa a przedziału [a,b]: 1.0857 z wartością funkcji Y(a): 0.93111
Wartość początkowa b przedziału [a,b]: 3 z wartością funkcji Y(b): -1.7606
Przedzial [a,b] zawężony od lewej strony (od strony punktu a)

licznik =

     4

Wartość początkowa a przedziału [a,b]: 1.7479 z wartością funkcji Y(a): 1.7306
Wartość początkowa b przedziału [a,b]: 3 z wartością funkcji Y(b): -1.7606
Przedzial [a,b] zawężony od lewej strony (od strony punktu a)

licznik =

     5

Wartość początkowa a przedziału [a,b]: 2.3686 z wartością funkcji Y(a): 1.9142
Wartość początkowa b przedziału [a,b]: 3 z wartością funkcji Y(b): -1.7606
Przedzial [a,b] zawężony od lewej strony (od strony punktu a)

licznik =

     6

Wartość początkowa a przedziału [a,b]: 2.6975 z wartością funkcji Y(a): 0.6996
Wartość początkowa b przedziału [a,b]: 3 z wartością funkcji Y(b): -1.7606
Przedzial [a,b] zawężony od lewej strony (od strony punktu a)

licznik =

     7

Wartość początkowa a przedziału [a,b]: 2.7835 z wartością funkcji Y(a): 0.11046
Wartość początkowa b przedziału [a,b]: 3 z wartością funkcji Y(b): -1.7606
Przedzial [a,b] zawężony od lewej strony (od strony punktu a)

licznik =

     8
```

Na podstawie powyższego wydruku, stwierdzono wykonanie 8 przebiegów globalnej pętli powtarzającej warunkowo ***while***. Natomiast wartość funkcji badanej w punkcie, określonym w tej pętli powtarzającej, jako przybliżone położenie pierwiastka funkcji badanej, wynosi aż 0.11046 i wskazuje na sukcesywne przybliżanie się do tego punktu lewego krańca przedziału domkniętego [a, b]. Zmniejszenie wartości progu tolerancji, powiedzmy 10-krotne (z wartości 0.05 do 0.005) powoduje nieznaczne tylko wydłużenie wykonania procedury znajdywania przybliżonego położenia pierwiastka (z 8 do 9 przebiegów pętli ***while***) z uwagi na znaczne nachylenie funkcji oraz jej monotoniczny charakter zmian w otoczeniu położenia pierwiastka badanej funkcji, przy wartości funkcji w znalezionym punkcie wynoszącym 0.013223.



Rys.3 Przebieg funkcji badanej (oznaczoną linią przerywaną niebieską), przebieg kolejnych prostych – siecznych przedziału [a, b] (oznaczone linią przerywaną czerwoną) oraz miejsca wyznaczania nowego położenia lewego przedziału a (oznaczonych symbolem czarnego rombu).

Wgląd w naturę przebiegu funkcji (która jest niemonotonicznie zmienna) oraz zasada działania metody znajdywania jednego z wielu pierwiastków badanej funkcji, wskazuje początkowo na niekorzystny wzrost wartości badanej funkcji na lewym bieżącym krańcu przedziału [a, b] (z przebiegu na przebieg pętli ***while***). Z kolei powyżej 8 przebiegu pętli powtarzającej ***while*** w funkcji następuje stosunkowo duży monotoniczny spadek tej funkcji, skutkujący stosunkowo znacznym tempem zbieżności algorytmu do rzeczywistego położenia szukanego pierwiastka na osi X.

Po czwarte: postanowiono przytoczyć anty-przykład zastosowania *rekurencyjnego* obliczania wartości elementów ciągu Fibonacciego. Poczynając od dwóch pierwszych elementów równych odpowiednio: 1 i 1, wartości każdego z następnych elementów wyznacza się, jako sumę elementu *ostatniego* i *przedostatniego*. Nawiasem mówiąc, prócz zastosowań ściśle matematycznych (oraz w rozwinięciu niektórych szeregów – również o zastosowaniu ściśle matematycznym), kolejne elementy ciągu Fibonacciego wyznaczają pewne proporcje.

Proporcje te, sugerowane, jako dogodne w estetycznym (wizualnym) odbiorze na przykład: podstawowych wymiarów wznoszonych budynków, portali, detali reliefów, również geometrycznych ram zawierających różnego rodzaju dzieła sztuki itp., leżały przez znaczący okres antycznego okresu Europy Południowej u podstaw geometrycznego planowania przestrzennego.

Pomimo prostoty w wyznaczaniu *iteracyjnym* elementów tego ciągu, tzn. o wartościach pierwszych dziesięciu elementów równych: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, *paradoksalnie* wykorzystanie *rekurencyjnej* definicji obliczania wybranego elementu ciągu Fibonacciego, w praktyce powyżej około 11-12 staje się prawdziwą studnią algorytmiczno-obliczeniową.

```
function [lfb] = fb(n);
if n == 1 %gdy rekurencyjnie osiągnięto dno,wykonuj blok podrzędny pod: 'if n == 1'%
    disp('Rekurencja najniższa nr: 1');
    lfb = 1;
    return;
elseif n == 2 % gdy rekurencyjnie osiągnięto przedostatni poziom powyżej 'dna', to: 'elseif n == 2'%
    disp('Rekurencja przedostatnia względem najniższej nr: 2');
    lfb = fb(n - 1) + fb(n - 1);
    disp(['Wartość czastkowa ciągu Fibonaciego: ' num2str(lfb) ', w nawrocie rekurencji nr: ' num2str(n)]);
else
    if n > 2 % gdy rekurencyjnie osiągnięto poziom wyższy niż 2, to: 'else ... if n > 2 ...end; ...end;'%
       disp(['Rekurencja pośrednia nr: ' num2str(n)]);
       lfb = fb(n - 1) + fb(n - 2);
       disp(['Wartość cząstkowa ciągu Fibonacciego: ' num2str(lfb) ', w nawrocie rekurencji nr: ' num2str(n)]);
       return;
    end;
end;
```

W rekurencyjnej definicji (przytoczonej powyżej) dla wartości elementu ciągu Fibonacciego, wyznaczanego rekurencyjnie poprzez wywoływanie *funkcji w funkcji*, powyżej drugiego elementu, następująca linijka wywołań:

```
    lfb = fb(n - 1) + fb(n - 1);
```

czyni z powyższego zadania, zadanie o dość ścisłej odpowiedniości względem zadania *przeszukiwania pełnego drzewa binarnego* (drzewa posiadającego w węźle rodzicielskim zawsze dwóch potomków z poziomu niższego).

Na marginesie, dla pełnego drzewa binarnego o 10 poziomach, na najniższym poziomie znajduje się aż 1024 liści – potomków (liści – węzłów, pozbawionych jeszcze niższych – zależnych od nich potomków). Przeszukiwanie takiego – pełnego drzewa binarnego – wraz z nawrotami przypomina – w czasie wykonania – algorytm o wielomianowym czasie wykonania (lub nawet algorytm o wykładniczym czasie wykonania).

Poniżej przedstawiono listing uruchomienia rekurencyjnej funkcji wyznaczania n-tego elementu szeregu Fibonacciego:

```
>> [lfb] = fb(5);
Rekurencja pośrednia nr: 5
Rekurencja pośrednia nr: 4
Rekurencja pośrednia nr: 3
Rekurencja przedostatnia względem najniższej nr: 2
Rekurencja najniższa nr: 1
Rekurencja najniższa nr: 1
Wartość czastkowa ciągu Fibonaciego: 2, w nawrocie rekurencji nr: 2
Rekurencja najniższa nr: 1
Wartość cząstkowa ciągu Fibonacciego: 3, w nawrocie rekurencji nr: 3
Rekurencja przedostatnia względem najniższej nr: 2
Rekurencja najniższa nr: 1
Rekurencja najniższa nr: 1
Wartość czastkowa ciągu Fibonaciego: 2, w nawrocie rekurencji nr: 2
Wartość cząstkowa ciągu Fibonacciego: 5, w nawrocie rekurencji nr: 4
Rekurencja pośrednia nr: 3
Rekurencja przedostatnia względem najniższej nr: 2
Rekurencja najniższa nr: 1
Rekurencja najniższa nr: 1
Wartość czastkowa ciągu Fibonaciego: 2, w nawrocie rekurencji nr: 2
Rekurencja najniższa nr: 1
Wartość cząstkowa ciągu Fibonacciego: 3, w nawrocie rekurencji nr: 3
Wartość cząstkowa ciągu Fibonacciego: 8, w nawrocie rekurencji nr: 5
```

Jak wynika z powyższego listingu, wartości cząstkowe (wyniki pośrednie ciągu Fibonacciego) w wielokrotnych nawrotach na różnych poziomach przeszukiwania (przypomnijmy: analogicznym do nawrotów w pełnym drzewie binarnym na różnych jego poziomach) są nadmiarowo – wielokrotnie powtarzane w obliczeniach.

Natomiast dla wartości argumentu równego 7 (jedynie od 2 pozycje wyżej), uruchomienie rekurencyjnego wywołania funkcji fb, skutkuje już znacząco większym listingiem wyników:

```
>> [lfb] = fb(7);
Rekurencja pośrednia nr: 7
Rekurencja pośrednia nr: 6
Rekurencja pośrednia nr: 5
Rekurencja pośrednia nr: 4
Rekurencja pośrednia nr: 3
Rekurencja przedostatnia względem najniższej nr: 2
Rekurencja najniższa nr: 1
Rekurencja najniższa nr: 1
Wartość czastkowa ciągu Fibonaciego: 2, w nawrocie rekurencji nr: 2
Rekurencja najniższa nr: 1
Wartość cząstkowa ciągu Fibonacciego: 3, w nawrocie rekurencji nr: 3
Rekurencja przedostatnia względem najniższej nr: 2
Rekurencja najniższa nr: 1
Rekurencja najniższa nr: 1
Wartość czastkowa ciągu Fibonaciego: 2, w nawrocie rekurencji nr: 2
```

```
Wartość cząstkowa ciągu Fibonacciego: 5, w nawrocie rekurencji nr: 4
Rekurencja pośrednia nr: 3
Rekurencja przedostatnia względem najniższej nr: 2
Rekurencja najniższa nr: 1
Rekurencja najniższa nr: 1
Wartość czastkowa ciągu Fibonaciego: 2, w nawrocie rekurencji nr: 2
Rekurencja najniższa nr: 1
Wartość cząstkowa ciągu Fibonacciego: 3, w nawrocie rekurencji nr: 3
Wartość cząstkowa ciągu Fibonacciego: 8, w nawrocie rekurencji nr: 5
Rekurencja pośrednia nr: 4
Rekurencja pośrednia nr: 3
Rekurencja przedostatnia względem najniższej nr: 2
Rekurencja najniższa nr: 1
Rekurencja najniższa nr: 1
Wartość czastkowa ciągu Fibonaciego: 2, w nawrocie rekurencji nr: 2
Rekurencja najniższa nr: 1
Wartość cząstkowa ciągu Fibonacciego: 3, w nawrocie rekurencji nr: 3
Rekurencja przedostatnia względem najniższej nr: 2
Rekurencja najniższa nr: 1
Rekurencja najniższa nr: 1
Wartość czastkowa ciągu Fibonaciego: 2, w nawrocie rekurencji nr: 2
Wartość cząstkowa ciągu Fibonacciego: 5, w nawrocie rekurencji nr: 4
Wartość cząstkowa ciągu Fibonacciego: 13, w nawrocie rekurencji nr: 6
Rekurencja pośrednia nr: 5
Rekurencja pośrednia nr: 4
Rekurencja pośrednia nr: 3
Rekurencja przedostatnia względem najniższej nr: 2
Rekurencja najniższa nr: 1
Rekurencja najniższa nr: 1
Wartość czastkowa ciągu Fibonaciego: 2, w nawrocie rekurencji nr: 2
Rekurencja najniższa nr: 1
Wartość cząstkowa ciągu Fibonacciego: 3, w nawrocie rekurencji nr: 3
Rekurencja przedostatnia względem najniższej nr: 2
Rekurencja najniższa nr: 1
Rekurencja najniższa nr: 1
Wartość czastkowa ciągu Fibonaciego: 2, w nawrocie rekurencji nr: 2
Wartość cząstkowa ciągu Fibonacciego: 5, w nawrocie rekurencji nr: 4
Rekurencja pośrednia nr: 3
Rekurencja przedostatnia względem najniższej nr: 2
Rekurencja najniższa nr: 1
Rekurencja najniższa nr: 1
Wartość czastkowa ciągu Fibonaciego: 2, w nawrocie rekurencji nr: 2
Rekurencja najniższa nr: 1
Wartość cząstkowa ciągu Fibonacciego: 3, w nawrocie rekurencji nr: 3
Wartość cząstkowa ciągu Fibonacciego: 8, w nawrocie rekurencji nr: 5
Wartość cząstkowa ciągu Fibonacciego: 21, w nawrocie rekurencji nr: 7
```

Wyznaczenie wartości 22-go elementu ciągu Fibonacciego w środowisku obliczeniowym Matlab w wersji 7.6, według wywołania powyższej definicji rekurencyjnej funkcji skutkuje listingiem, o objętości rzędu parędziesięciu kilobajtów oraz wymaga, co najmniej 10 sekund pracy procesora (procesora 1.6GHz, na platformie 64-bitowej systemu Windows 2007).

Ostatecznie, wyznaczenie wartości przykładowo 27 elementu ciągu Fibonacciego, według wywołania powyższej definicji rekurencyjnej funkcji skutkuje listingiem liczonym najprawdopodobniej w megabajtach oraz czasem wykonania wynoszącym, co najmniej 105 sekund!

Natomiast, wykonanie wywołania tej funkcji z argumentem rzędu 40 lub 50 może okazać się niemożliwym, przy umiarkowanych zasobach obliczeniowych. Prostszym rozwiązaniem jest wykonanie obliczeń elementów ciągu Fibonacciego wprost, iteracyjnie.