Cyprian T. Lachowicz

"Informatyczne podstawy projektowania"
Podstawy Scilaba

Rozdział 1

Typy danych, skalary, wektory,macierze

1.1. Skalary

1.1.1. Tworzenie i operacje na skalarach

Wyrażenia skalarne są to z reguły liczby rzeczywiste lub zespolne dla których zdefiniowano podstawowe operatory działań arytmetycznych, dysponujemy więc możliwością wykorzystania 5 operatorów +,- ,⋆,/ i ^ (potęgowanie). Wartości liczbowe mogą być przypisane do zmiennych dzięki operatorowi przypisania, przykładowo:

```
a=5
 a  =

    5.
```

Wartości zmiennych zespolonych przypisujemy w sposób identyczny pamiętając o konieczności użycia symbolu liczby urojonej $\%i$

```
b=4-3*%i;

a*b
 ans  =

    26. - 7.i
```

Należy zwrócić uwagę, że Scilab wykonuje polecenia natychmiast po wciśnięciu Return. Wynik jest wyświetlany jeśli polecenie tekstowe nie było zakończone średnikiem.

1.1.2. Tworzenie macierzy

Podstawowym typem danych w Scilabie jest macierz[1]. Możemy definiować macierze zarówno o elementach rzeczywistych jak i zespolonych. Najprostszym sposobem definiowania macierzy jest wprowadzenie z klawiatury listy jej elementów, oczywiście należy wykonać to w określony sposób.

- Elementy macierzy w każdej linii są separowane od siebie albo spacją albo przecinkiem.

[1] Liczba jest więc interpretowana jako macierz o wymiarze zerowym, również wektor to oczywiście przykład macierzy

- Lista elementów musi być ograniczona nawiasami kwadratowymi.
- Każdy wiersz macierzy za wyjątkiem ostatniego musi być zakończony średnikiem.

Przykładowo polecenie generujące macierz $(3 * 3)$ ma postać

```
A=[1 2 3;5 4 3;1 1 1]
```

```
A  = !   1.    2.    3. !
     !   5.    4.    3. !
     !   1.    1.    1. !
```

Wektor (lub ogólnie macierz) jest przechowywany w pamięci komputera i w każdej chwili może być przywołana, poprzez odwołanie się do jej nazwy. Prześledźmy efekty wykonania poleceń.

```
 -->b=[10 56 23 1];
```

Wektor b nie jest wyświetlany na ekranie, ponieważ polecenie zakończyliśmy średnikiem. Aby zobaczyć wektor b piszemy b :

```
b

 b  =

!   10.    56.    23.    1. !
```

Użycie polecenia ' umożliwia przekształcenie wektora wierszowego w wektor kolumnowy.

```
b'
```

Polecenia zbyt długie, aby zmieściły się w jednej linii, mogą być zapisywane w kilku liniach, przy wykorzystaniu znaku kontynuacji składającego się z trzech kropek.

```
T=[1 1 1 1 1 ;...
   2 2 2 2 2 ;...
   3 3 3 3 3;...
   4 4 4 4 4]

 T  =
!   1.    1.    1.    1.    1. !
!   2.    2.    2.    2.    2. !
!   3.    3.    3.    3.    3. !
!   4.    4.    4.    4.    4. !
```

Aby wprowadzić liczby zespolone piszemy:

```
c=1 + 2*%i
c  =
    1. + 2.i
```

natomiast macierz o argumentach zespolonych

```
Y=[1+%i, 1-%i;1,%i]
Y  = !   1. + i      1. - i   !
     !   1.             i     !
```

1.1.3. Różne rodzaje macierzy

Macierz diagonalna jednostkowa

Argumentami funkcji generującej macierz są liczba wierszy `n` i liczba kolumn `m`. Aby otrzymać macierz diagonalna jednostkową wymiarach 4*4 piszemy:

```
I=eye(4,4)
 I  =
!   1.    0.    0.    0. !
!   0.    1.    0.    0. !
!   0.    0.    1.    0. !
!   0.    0.    0.    1. !
```

Polecenie diag() pozwala wygenerować macierz z niezerową przekątna główną, której elementami są poszczególne elementy wcześniej zdefiniowanego wektora b.

```
B=diag(b)
 B  =
!   10.    0.     0.     0. !
!   0.     56.    0.     0. !
!   0.     0.     23.    0. !
!   0.     0.     0.      1.!
```

Zastosowanie polecenia diag() do macierzy B umożliwia wydobycie w formie wektora kolumnowego elementów na głównej przekątnej.

```
b=diag(B)
 b  =
!   10. !
!   56. !
!   23. !
!   1.  !
```

Macierze zerowe i macierz jednostkowe

Polecenie `zeros()` i `ones()` umożliwia tworzenie macierzy zerowej oraz macierzy jednostkowej. Podobnie jak dla polecenia `eye()`, jej argumentami są liczba wierszy n i kolumn m.

```
C=ones(3,4)
 C  =
!   1.    1.    1.    1. !
!   1.    1.    1.    1. !
!   1.    1.    1.    1. !
```

Jako argumentu można użyć również nazwy innej macierzy, której wymiary będą określały wymiary nowej macierzy.

```
O=zeros(C)
 O  =
!   0.    0.    0.    0. !
!   0.    0.    0.    0. !
!   0.    0.    0.    0. !
```

Macierz trójkątne

Polecenia `triu()` i `tril()` umożliwiają zbudowanie na bazie danej macierzy nowej macierzy trójkątnej górnej „u" lub dolnej „l".

```
U=triu(C)
 U  =
!   1.    1.    1.    1. !
!   0.    1.    1.    1. !
!   0.    0.    1.    1. !
```

Macierze losowe

Polecenie `rand()` umożliwia zbudowanie macierzy zawierające liczby pseudolosowe z przedziału [0,1]. Jej argumentami są liczba wierszy n i kolumn m.

```
M=rand(2,4)
 M  =
!   0.7263507    0.5442573    0.2312237    0.8833888 !
!   0.1985144    0.2320748    0.2164633    0.6525135 !
```

Wektory

Wektor wierszowy o elementach równomiernie, rozłożonych uzyskujemy poleceniem `linspace()`.

```
x=linspace(0,0.8,9)
 x  =
!   0.    0.1    0.2    0.3    0.4    0.5    0.6    0.7    0.8 !
```

Podobny efekt uzyskujemy wykorzystując konstrukcję

```
y= wartość początkowa :  przyrost :  wartość ostateczna


y=0: 0.3: 1
 y  =
!   0.    0.3    0.6    0.9 !
```

Jeśli opuścimy `przyrost`, to domyślnie jest przyjmowany jako 1.

```
ind=1: 5
 ind  =
!   1.    2.    3.    4.    5. !
```

Odmianą polecenia `linspace()` jest polecenie `logspace()` o takiej samej składni, rozmieszczające elementy rozłożone w sposób logarytmiczny.

1.2. Operatory działań macierzowych

Wszystkie operatory do działań na macierzach są dostępne dla macierzy o właściwych dla danej operacji kombinacji wymiarów.

1.2.1. Dodawanie i odejmowanie

Dodawanie i odejmowanie macierzy polega na dodawaniu element do elementu. Przykładowo do macierzy A dodajemy macierz jedynkową o tym samym wymiarze.

```
A=[1 2 3; 4 5 6; 7 8 9]
 A  =

!   1.    2.    3. !
!   4.    5.    6. !
!   7.    8.    9. !

D=A+ones(A)
 D  =

!   2.    3.    4.  !
!   5.    6.    7.  !
!   8.    9.    10. !
```

Oczywiście dodanie macierzy o różnych wymiarach generuje błąd.

```
A+M
    !--error     8
inconsistent addition
```

1.2.2. Mnożenie, potęgowanie

Mnożenie dwu macierzy jest możliwe tylko wtedy, jeśli ich wymiary są odpowiednio dostosowane do siebie. Wynikiem mnożenia dwu kompatybilnych macierzy o wymiarach (3,3)*(3,4) jest

```
A*C
 ans  =

!   6.     6.     6.     6.  !
!   15.    15.    15.    15. !
!   24.    24.    24.    24. !
```

Iloczyn niezgodnych macierzy generuje błąd.

```
C*A
    !--error    10
inconsistent multiplication
```

Podnoszenie do potęgi, to wielokrotne mnożenie macierzy.

```
A^3
 ans  =
!   468.     576.     684.  !
!   1062.    1305.    1548. !
!   1656.    2034.    2412. !
 ans  =
```

1.2.3. Transpozycja

Polecenie transpozycji macierzy uzyskujemy poleceniem '. Przykładowo obliczmy macierz transponowaną `At`.

```
At=A'
 At  =
!   1.    4.    7. !
!   2.    5.    8. !
!   3.    6.    9. !
```

W przypadek macierzy zespolonych transpozycja macierzy generuje macierze sprzężone.

```
Ac=A+%i*eye(3,3)
 Ac  =

!   1. + i      2.          3.        !
!   4.          5. + i      6.        !
!   7.          8.          9. + i    !

Ac_adj=Ac'
 Ac_adj  =

!   1. - i      4.          7.        !
!   2.          5. - i      8.        !
!   3.          6.          9. - i    !
```

1.2.4. Produkt skalarny wektorów

Aby otrzymać produkt skalarny dwu wektorów, mnożymy dwa wektory: wierszowy i kolumnowy.

```
x=linspace(0,1,5)'
 x  =
```

```
!   0.   !
!   0.25 !
!   0.5  !
!   0.75 !
!   1.   !

-->y=(1:5)'
 y  =

!   1. !
!   2. !
!   3. !
!   4. !
!   5. !

-->p=y'*x
 p  =

    10.
```

Jeśli mnożenie przeprowadzimy w sposób odwrotny to jest wektor kolumnowy (5,1) przez wektor wierszowy (1,5)), otrzymujemy macierz (5,5).

```
Pext=y*x'
 Pext  =

!   0.    0.25    0.5    0.75    1. !
!   0.    0.5     1.     1.5     2. !
!   0.    0.75    1.5    2.25    3. !
!   0.    1.      2.     3.      4. !
!   0.    1.25    2.5    3.75    5. !
```

1.2.5. Operatory działań tablicowych

Bardzo użyteczną klasą operatorów działań arytmetycznych są operatory tablicowe. Aby pomnożyć dwie macierze A i B o tych samych wymiarach, element po elemencie, wykorzystujemy operatory działań tablicowych '.∗', './'. Wrażenie A.*B jest macierzą $[a_{ij}b_{ij}]$ i wyrażenie A./B jest macierzą $[a_{ij}/b_{ij}]$.

```
A./A
 ans  =

!   1.    1.    1. !
!   1.    1.    1. !
!   1.    1.    1. !

A=[1 2 3; 4 5 6; 7 8 9]
```

```
!   1.    2.    3. !
!   4.    5.    6. !
!   7.    8.    9. !

B=[2 2 2; 2 2 2; 2 2 2]
 B  =

!   2.    2.    2. !
!   2.    2.    2. !
!   2.    2.    2. !

C=A*B
 C  =

!   12.    12.    12. !
!   30.    30.    30. !
!   48.    48.    48. !


D=A.*B
 D  =

!   2.     4.     6.  !
!   8.     10.    12. !
!   14.    16.    18. !

E=A./B
 E  =

!   0.5    1.     1.5 !
!   2.     2.5    3.  !
!   3.5    4.     4.5 !
```

Podnoszenie do potęgi wymaga użycia operatora `.^`

```
-->A.^2
 ans  =

!   1.     4.     9.  !
!   16.    25.    36. !
!   49.    64.    81. !

-->A^2
 ans  =
```

```
!   30.     36.     42.  !
!   66.     81.     96.  !
!   102.    126.    150. !
```

1.2.6. Specyficzne aspekty algebry macierzowej

Zarówno mnożenie jak i dzielenie macierzy przez wielkość skalarną jest realizowane według schematu każdy element macierzy mnożymy (dzielimy) przez skalar. Tylko nieco mniej oczywiste jest dodawanie (odejmowanie) skalaru do macierzy.

1. Wyrażenie `M+s` gdzie M jest macierzą a s jest skalarem, wykonywane jest właściwie poleceniem `M+s*ones(M)`, co sprowadza się do dodania do macierzy M odpowiednio zbudowanej macierzy `s*ones(M)`.
2. Jak wiemy wyrażenie `A./B` odpowiada dzieleniu tablicowemu, jeśli więc A jest skalarem wyrażenie `A./B` jest obliczane właściwie jako A*ones(B)./B którego rezultatem jest macierz.

Sumowanie elementów i inne „sztuczki"

Dla obliczenia sumy elementów od 1 do 6 piszemy:

```
sum(1:6)
 ans  =

    21.
```

Polecenie `sum()` jest bardzo elastyczne i może być wykorzystywane na różne sposoby. Przykładowo obliczamy sumę wierszy i kolumn macierzy

```
B=[1 2 3 ; 4 5 6]
 B  =

!   1.    2.    3. !
!   4.    5.    6. !

sum(B,"row")
 ans  =

!   5.    7.    9. !

sum(B,"col")
 ans  =

!   6.  !
!   15. !
```

Praktycznym obiektem jest macierz pusta, definiowana jak pokazano niżej.

```
C=[]
```

```
 C  =

     []
```

Macierz pusta może być wykorzystywana jak każda macierz. Przykładowo: `A=[ ]+ A` i `A=[ ]*A`. Jeśli zastosujemy do tej macierzy polecenie `sum()`, otrzymamy:

```
sum([])
 ans  =

    0.
```

Produkt - iloczyn elementów

Chcąc obliczyć iloczyn kolejnych liczb wykorzystujemy polecenie `prod()`

```
prod(1:5)
 ans  =

    120.
```

Polecenie generuje wektor wierszowy, którego argumentami są iloczyny elementów wszystkich kolumn.

```
-->prod(B,"row")
 ans  =

!   4.    10.    18. !
```

Natomiast polecenie generuje wektor kolumnowy, którego argumentami są iloczyny elementów wszystkich wierszy.

```
prod(B,"col")
 ans  =

!   6.   !
!   120. !

-->prod(B)
 ans  =

    720.

prod([])
 ans  =

    1.
```

Polecenie `matrix()` pozwala przebudować daną macierz w nową o innych wymiarach, ale o tej samej liczbie elementów.

```
B_new=matrix(B,3,2)
 B_new  =

!   1.    5. !
!   4.    3. !
!   2.    6. !
```

Utworzona macierz powstała z macierzy B, według zasady, że trzy kolejne (licząc wierszami) elementy tworzą pierwszą kolumnę trzy następne drugą kolumnę.

1.2.7. Polecenie `size()` i `length()`

`Size()` umożliwia uzyskanie informacji o liczbie kolumn i wierszy danej nieznanej macierzy lub wektora.

```
[nl,nc]=size(B)  // nl liczba wierszy,  nc - liczba kolumn
 nc  =

    3.
 nl  =

    2


x=[5 4 3 2 1 ]
 x  =

!   5.    4.    3.    2.    1. !

size(x)
 ans  =

!   1.    5. !
```

Pierwsza cyfra (w ostatnim wyrażeniu) oznacza liczbę linii (tu równą 1), druga liczbę kolumn (tu równą 5). Polecenie `length()` pozwala określić liczbę elementów macierzy (rzeczywistej czy zespolonej).

```
length(x)
 ans  =

    5.

length(B)
 ans  =

    6.
```

Wersja polecenia `size()`: `size(A,'r')` i `size(A,'c')` pozwala obliczyć liczbę wierszy lub liczbe kolumn macierzy `A`.

1.2.8. Odwoływanie się ..., wydobywanie z ..., zawężanie macierzy i wektorów , sklejanie macierzy

Do poszczególnych elementów macierzy można się odwołać dzięki poleceniu `NAZWAMACIERZY(numer_wiersza, numer_kolumny)`. Dla macierzy `A`:

```
A  =

!   1.    2.    3. !
!   5.    4.    3. !
!   1.    1.    1. !
```

odwołujemy się do elementu leżącego w trzecim wierszu i trzeciej kolumnie poleceniem:

```
A33=A(3,3)
 A33  =

    1.
```

W przypadku, kiedy mamy do czynienia z wektorami, wystarczy podać tylko pozycję danego elementu.

Poważną zaletą języka typu Scilab jest możliwość wydobywania z macierzy jej submacierzy. Przykład poniżej pokazuje jak wydobyć drugą kolumnę

```
A(:,2)
 ans  =

!   2. !
!   4. !
!   1. !
```

bądź trzeci wiersz.

```
-->A(3,:)
 ans  =

!   1.    1.    1. !
```

W kolejnym przykładzie widzimy jak wydobyć submacierz, składającą się z części wspólnej od 1 do 2 wiersza oraz od 1 do 2 kolumny.

```
-->A(1:2,1:2)
 ans  =

!   1.    2. !
```

```
!   5.    4. !
```

Jeden z możliwych wariantów tworzenia submacierzy na bazie macierzy wyjściowej, polega na następującej idei: jeśli A jest macierzą o wymiarach (n,m) i jeśli mamy dwa wektory V1 V2 wskazujące które wiersze i kolumny macierzy A nas interesują, to macierz A(V1,V2) jest macierzą o wymiarach (p,q) utworzoną przez części wspólne wybranych linii i kolumn zapisanych w wektorach V1 i V2.

```
A([1 3],[2 3])
 ans  =

!   2.    3. !
!   1.    1. !
```

lub

```
A([3 1],[2 1])
 ans  =

!   1.    1. !
!   2.    1. !
```

W codziennej praktyce wydobywanie poszczególnych wierszy lub kolumn jest stosunkowo proste, o czym można się przekonać analizując poniższe polecenie - umożliwiające wydobycie z macierzy A trzech kolejnych kolumn.

```
A(1:2:3,:) // lub A([1 3],:)
 ans  =

!   1.    2.    3. !
!   1.    1.    1. !
```

Szczególnie przydatne jest polecenie $ umożliwiające wskazanie ostatniej lini lub kolumny w macierz której wymiarów nieznamy.

```
A($,$-1)
 ans  =

    1.
```

Przejdźmy teraz do polecenia umożliwiającego składanie z różnych macierzy większej macierzy wynikowej. Zdefiniujmy kilka mniejszych macierzy

```
A11=1;

A12=[2 3 4];
A21=[1;1;1];
A22=[4,9,16;
    8,27,64;
  16,81,256];
```

Wykorzystując polecenie łączenia otrzymujemy macierz A zbudowaną z czterech macierzy składowych:

```
A=[A11,A12;A21,A22]
 A  =

!   1.    2.     3.     4.   !
!   1.    4.     9.     16.  !
!   1.    8.     27.    64.  !
!   1.    16.    81.    256. !
```

1.3. Ćwiczenia w konstruowaniu macierzy

1. Utwórz macierz 5x5 wykorzystując jeden wektor oraz jego transpozycję
   ```
   A  =

           !   1.    2.     3.     4.     5.   !
           !   2.    4.     6.     8.     10.  !
           !   3.    6.     9.     12.    15.  !
           !   4.    8.     12.    16.    20.  !
           !   5.    10.    15.    20.    25.  !
   ```
2. Wyciągnij z macierzy A pierwszą i ostatnią linię. Oblicz produkt skalarny tak określonych wektorów. Wyjmij z macierzy A elementy leżące na przekątnej głównej. Zbuduj macierze trójkątne górne i dolne.
3. Oblicz macierz transponowaną do macierzy A
4. Utwórz macierz z elementami losowymi. Wykonaj operacje sumowania,dzielenia z macierzą A
5. Wydobądź element macierzy A leżący na przecięciu trzeciej linii i czwartej kolumny.
6. Zdefiniuj macierz o konstrukcji podanej poniżej

$$C = \begin{pmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & 0 \\ \dots & \dots & \dots & \dots & -1 \\ 0 & \dots & 0 & -1 & 2 \end{pmatrix}$$

Rozdział 2

Inne typy danych

2.1. Wielomiany

Scilab posiada bardzo rozbudowaną bibliotekę funkcji umożliwiających operacje na wielomianach algebraicznych dowolnego stopnia postaci:

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0 x = b$$

2.1.1. Tworzenie wielomianów

Istnieją dwa sposoby tworzenia wielomianów. Poniżej pokazano pierwszy z nich.

```
p=poly([1 2 3],'z','coeff')
 p  =

                2
    1 + 2z + 3z
```

p jest wielomianem zmiennej `z` posiadającym trzy współczynniki 1,2,3. Istnieje możliwość napisania samego 'c' w miejsce 'coeff'. Dodatkowo jeśli pominiemy trzeci argument w wywołaniu funkcji będzie, to oznaczać że generowany wielomian ma mieć pierwiastki podane jako pierwszy argument tu [1 2 3].

```
p=poly([1 2 3],'z')
 p  =

                2   3
  - 6 + 11z - 6z + z
```

Powyższa konstrukcja powinna być stosowana tylko dla kolejno rosnących wykładników. Aby wygenerować dowolny wielomian wykorzystuje się konstrukcję:

```
s=poly(0,'s') \\ tworzymy wielomian ze zmienną s
 s  =

    s

p=1+s^2
```

```
 p  =

         2
    1 + s

p=1+4*s^3
 p  =

          3
    1 + 4s
```

Aby otrzymać pierwiastki tego wielomianu wykorzystuje się polecenie `roots()`.

```
-->roots(p)
 ans  =

!   0.3149803 + 0.5455618i !
!   0.3149803 - 0.5455618i !
!  -0.6299605              !
```

2.1.2. Podstawowe operacje na wielomianach

Wykonajmy kolejne polecenia :

```
-->q=1+2*s
 q  =

    1 + 2s

-->p+q
 ans  =

                3
    2 + 2s + 4s

 -->p-q
 ans  =

           3
  - 2s + 4s

-->p*q
 ans  =

                3    4
    1 + 2s + 4s + 8s
```

```
-->p/q
 ans  =

          3
    1 + 4s
    ------
    1 + 2s
```

Podsumowując wielomiany mogą być dodawane mnożone, odejmowane i dzielone.

Polecenie `derivat()` umożliwia łatwe obliczenie pochodnej wielomianu:

```
-->derivat(p)
 ans  =

       2
    12s
```

Polecenie `horner()` pozwala na wyliczenie wartości wielomianu w wybranym punkcie.

```
-->horner(p,3)
 ans  =

    109.
```

2.1.3. Macierze wielomianów

Wielomiany podobnie jak liczby rzeczywiste, zespolone mogą być wykorzystane jako elementy macierzy.

Należy zauważyć, że instrukcja `poly()` pozwala łatwo wyznaczyć wielomian charakterystyczny danej macierzy.

```
-->poly([1 2;3 4],'s')
 ans  =

              2
  - 2 - 5s + s
```

Budowa macierzy której elementami są wielomiany jest bardzo prosta:

```
A=[1 s;s 1+2*s^2]
 A  =

!   1     s       !
!                 !
!               2 !
!   s     1 + 2s  !
```

lub inaczej:

```
B=[1/s 1/(1+s);1/(1+s) 1/s^2]
 B  =

!   1           1    !
!   -         -----  !
!   s         1 + s  !
!                    !
!     1       1      !
!   -----     -      !
!               2    !
!   1 + s     s      !
```

Argument *num* i *den* umożliwia wydobycie z danej macierzy macierzy zbudowanej z liczników i macierzy zbudowanej z mianowników:

```
B('num')
 ans  =

!   1     1  !
!            !
!   1     1  !

B('den')
 ans  =

!   s         1 + s  !
!                    !
!               2    !
!   1 + s     s      !
```

```
B('den')(2,2)\\ wydobycie mianownika elementu z drugiej linii i drugiej kolumny

 ans  =

     2
    s
```

2.2. Listy

Lista jest strukturą zawierającą dane w postaci łańcuchów znakowych. Istnieją trzy rodzaje list: lista zwykła - `list`, lista o strukturze wektorowej - `tlist`, lista o strukturze macierzowej `mlist`.

2.2.1. Lista zwykła `list()`

Lista zwykła jest zbiorem obiektów dostępnych w Scilabie. Podobnie jak w Matlabie obiekty nie muszą być tego samego rodzaju. Mogą więc być skalarami, wektorami, macierzami, łańcuchami znakowymi, funkcjami i wszystkimi możliwymi ich kombinacjami. Przykład listy pokazano poniżej. Poleceniem kreującym listę jest `list()`.

```
lista=list('test',[1,2;3 4],...

['To jest przykład'; 'roznych danych'])
 lista  =


       lista(1)

 test

       lista(2)

!   1.    2. ! !   3.    4. !

       lista(3)

!To jest przykład  !
!                  !
!roznych danych    !
```

Dostęp do poszczególnych elementów listy uzyskujemy poprzez indeksy.

```
lista(1)
ans=
 test
```

Dostęp do dowolnego elementu macierzy na przykład z pierwszego wiersza i drugiej kolumny, która jest elementem tej listy, uzyskujemy bardzo prosto- wręcz intuicyjnie.

```
lista(1)(1,2)
ans=
 2
```

Chcąc dopisać coś na początek listy, wykorzystujemy indeks zerowy. Oczywiście wszystkie obiekty przesuwają się o jedna pozycję do przodu.

```
lista(0)=%eps;
lista(2)
ans
 test
```

Dopisanie można zrobić też w nieco inny sposób, zachowanie kolejności nowych elementów nie jest konieczne,brakujące pozostaną niezdefiniowane.

```
lista(8)='koniec'
 lista  =
       lista(1)
    2.220D-16
       lista(2)
 test
       lista(3)
!   1.    2. !
!   3.    4. !
       lista(4)
!To jest przykład  !
!                  !
!roznych danych    !
       lista(5)
    Undefined
[More (y or n ) ?]
       lista(6)
    Undefined
       lista(7)
    Undefined
       lista(8)
 koniec
```

Usuwanie z listy może odbywać się po jednym elemencie, wpisaniem na odpowiednią pozycję `null()`.

```
a=list(1,2,3,4,5)
 a  =


       a(1)

    1.

       a(2)

    2.

       a(3)

    3.

       a(4)

    4.
```

```
       a(5)

    5.

a(3)=null();  // usuniecie elementu 3

a             // wydruk nowej listy
 a  =


       a(1)

    1.

       a(2)

    2.

       a(3)

    4.

       a(4)

    5.
```

Jak widać brakuje 3, lista oczywiście przenumerowała automatycznie indeksy. Stosunkowo łatwo jest wydobyć pewne elementy listy i przypisać je pod nowe zmienne.

```
[u,v]=a(2:3) v  =

    4.
 u  =

    2.
```

Istnieją funkcje do pracy na listach, ich nazwy mówią dużo o przeznaczeniu: size(), length().

2.2.2. Lista typów `tlist()`

Lista typów inicjowana poleceniem `tlist()` jest jej specjalnym rodzajem listy. Pierwszy elementem tej listy jest zawsze łańcuch lub wektor łańcuchów.

```
Tlista=tlist(['przyklad','pierwszy','drugi'],1.23,[1,2],' pozostałe')
Tlista  =
       Tlista(1)
!przyklad  pierwszy  drugi  !
```

```
       Tlista(2)
    1.23
       Tlista(3)
!   1.    2. !
       Tlista(4)
  pozostałe
```

Dostęp do elementów listy uzyskujemy w zwykły sposób poprzez jej indeksy, ale jest również możliwy inny sposób, poprzez nazwy zawarte w pierwszym obiekcie listy (za wyjątkiem pierwszego łańcucha). Oczywiście dostęp będzie możliwy tylko do dwu pierwszych elementów listy.

```
Tlista.pierwszy
 ans  =
    1.23
Tlista.drugi
 ans  =
!   1.    2. !
Tlista.przykład
         !--error     4
```

2.2.3. Lista macierzowa

Ostatnim rodzajem listy jest lista macierzowa inicjowana poleceniem `mlist()`. Konstrukcja tej listy jest zbliżona do listy typów, ale nie jest tu możliwy dostęp poprzez indeksy, jedynym sposobem jest dostęp poprzez nazwę wymienioną w pierwszym łańcuchu tworzonej listy.

```
M=mlist(['V','name','value'],['a','b','c'],[1 2 3])
// odwołanie się elementu
M.name
 ans  =
!a  b  c  !

//odwolanie się do elementu
 M('value')
 ans  =
!   1.    2.    3. !
```

2.3. Ćwiczenia

1. Utwórz wielomiany $u = 1 + z^2$ i $v = 1 + 3 * z^3$ wykonaj dzielenie u/v.
2. Oblicz $u * v$, wyznacz pierwiastki nowo utworzonego wielomianu.
3. Wyznacz pochodną wielomianu $u * v$
4. Oblicz wartość wielomianu dla z=4
5. Utwórz macierz wielomianów (u,v;u/v,u*v).

Rozdział 3

Funkcje użytkownika

3.1. Tworzenie funkcji i sposoby ich wykorzystywania

Konstrukcją bardzo rozszerzającą możliwości każdego języka programowania jest funkcja. W Scilabie są dwa różne sposoby definiowania funkcji.

Pierwszy, bardzo prosty, ale ograniczony do prostych wyrażeń matematycznych, polega na bezpośrednim włączeniu kodu funkcji w program interpretowany lub umieszczeniu go w jego skrypcie. Służy do tego polecenie `deff()`, jego składnia jest następująca

```
deff('sposób_wywołania','co_oblicza')
```

Jej wykorzystanie jest bardzo proste, co pokazują przykłady.
Obliczanie iloczynu *z=sin(x)*cos(x)*

```
deff('z=sincos(x)','z=cos(x)*sin(x)')
```

jej wywołanie może nastąpić z linii komend lub z wnętrza skryptu.

```
sincos(12.0)
ans=0.4527892
```

Dwa kolejne przykłady nie wymagają komentarza.

```
deff('[x]=mojeplus(y,z)','x=y+z')
deff('[x]=mojemacro(y,z)',['a=3*y+1'; 'x=a*z+y'])
```

Drugi sposób definiowania funkcji i sposób jej wykorzystania jest znacznie bardziej rozbudowany ale możliwości tak zdefiniowanych funkcji sa ogromne.
Ogólnie funkcje definiuje się w odrębnym pliku dyskowym, dla Scilaba powinien mieć rozszerzenie `*.sci`. Poprawnie skonstruowana funkcja powinna mieć następującą strukturę

```
function[out1,out2,...]=nazwafunkcji(in1,in2,...)
ciąg_instrukcji
endfunction
```

Jako przykład przedstawiono funkcję obliczającą iloczyn liczb naturalnych od 1 do n.

```
function[a]=iloczyn(n)
 a=1
 for x=1:n do
 a=a*x
 end
endfunction
```

Funkcja iloczyn będzie wywoływana z pliku (w którym ja zapisaliśmy) poleceniem

```
getf('ścieżka\nazwa_funkcji').
```

Przykład wykorzystania w trybie interpretacji lub z poziomu skryptu jest następujący

```
getf('c:'\katalog\iloczyn.sce')
iloczyn(6)
ans=720
```

Przekazywanie argumentów do funkcji odbywać się może dwutorowo[1]

— poprzez zmienne wyszczególnione na liście argumentów wejściowych (*in1,in2,...*),
— poprzez zmienne globalne.

Odbieranie argumentów odbywa się poprzez

— zmienne wyszczególnione na liście argumentów wyjściowych (*out1,out2,...*),
— zmienne globalne,
— argumenty wyjściowe po wykonaniu polecenia `return`.

Przykład funkcji obliczającej pole i obwód koła o promieniu r.

```
function[Pole,Obwod]=kolo(promien)
  Pole=%pi*promien^2
  Obwod=2*%pi*promien
  endfunction
```

Wywołanie i wykorzystanie tak zdefiniowanej funkcji może być następujące.

```
getf('scieżka do pliku funkcji\kolo.rozszerzenie')
[P O]=kolo(6)
```

[1] Scilab ma jeszcze inne możliwości, patrz dokumentacja tworców

Funkcje użytkownika mogą być zdefiniowane w formie rekurencyjnej, klasycznym przykładem jest tu funkcja obliczająca silnię.

$$dla \begin{cases} n = 0 & silnia(0) = 1 \\ n \neq 0 & silnia(n) = n * silnia(n-1) \end{cases}$$

```
function[y]=silnia(n)
if n==0 then y=1
else
y=n*silnia(n-1)
end
endfunction
```

Ciekawym sposobem wykorzystania funkcji definiowanej bezpośrednio poleceniem `deff()` jest „łatwe" znajdywanie miejsc zerowych tak zdefiniowanych funkcji. Jest to możliwe dzięki wykorzystaniu polecenia `fsolve(): fsolve (wartość początkowa, funkcja)`.

```
deff('[y]=myplus(x)','y=x+3')      // definicja funkcji inline
fsolve(1,myplus)                   // szukanie miejsc zerowych
 ans  =

  - 3.
```

3.2. Uwagi o niektórych poleceniach specyficznych dla funkcji

W obu systemach istnieją pewne polecenia charakterystyczne wyłącznie dla funkcji. Jednym z nich jest instrukcja `return` umożliwiająca przerwanie pracy funkcji i powrót do programu nadrzędnego z zachowaniem aktualnych wartości argumentów wyjściowych funkcji. Prezentowana poniżej funkcja `dziel` wykonuje dzielenie liczby *1* przez dowolną liczbę (również *0*). Ponieważ dzielnie przez *0* wygeneruje błąd, działanie funkcji musi być w takim przypadku przerwane, do tego celu wykorzystuje się polecenie `return`. W programie pokazano również sposób wykorzystania stałej oznaczającej nieskończoność (%inf).

```
function z=dziel(x)
if x==0 then
 z=%inf                    w Matlabie z=inf
return
end
z=1/x
endfunction
```

3.3. Przykład

Nietrywialnym przykładem użycia funkcji jest obliczanie obliczanie współczynników wielomianu Czebyszewa. Jego definicja ma charakter rekurencyjny

$$T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x), n = 2, 3, \ldots, T_0(x) = 1, T_1(x) = x.$$

Implementacja tej definicji w postaci funkcji jest względnie łatwa.

```
function T=ChebT(n)

 t0=1;
 t1=[1 0];
 if n== 0
 T=t0;
  elseif n==1;
     T=t1;
  else
     for k=2:n
        T=[2*t1 0] -[0 0 t0];
        t0=t1;
        t1=T;
     end
end
```

Współczynniki wielomianu obliczamy poleceniem.

```
 wspol=ChebT(3)
 wspol = 4 0 -3 0
```

Wielomian przyjmuje postać $T_3(x) = 4x^3 - 3x$.

3.4. Skrypty

Skrypty to komendy, funkcje, dane zapisane w odrębnym pliku tekstowym.

3.5. Cwiczenia

1. Napisz funkcję obliczającą objętość i pole powierzchni kuli.
2. Wykorzystując rekurencyjną definicje liczb Fibonacciego napisz funkcję obliczającą te liczby.

$$\begin{cases} n = 0 & fib(0) = 0, \\ n = 1 & fib(1) = 1, \\ n \neq 0, 1 & fib(n) = fib(n-1) + fib(n-2). \end{cases}$$

3. Znajdź miejsca zerowe funkcji $y = ax^2+bx+c$ wykorzystując polecenie `fsolve()`. Obliczenia wykonaj dla kilku zestawu stałych a, b, c.

Rozdział 4

Grafika

4.1. Okna graficzne

Wykorzystanie takich instrukcji graficznych jak `plot(), plot2d(), plot3d()` nie jest zalecane, jeśli nie jest utworzone jedno lub nawet kilka okienek graficznych. Istnieje wiele bardzo użytecznych poleceń ułatwiających generowanie okienek graficznych oraz pracę w ich obrębie. W tablicy (tab. 4.1) zestawiono kilkoro spośród nich.

Tablica 4.1. Podstawowe polecenia do pracy z oknami graficznymi

`xset("window",num)`	Ustawia bieżące okienko graficzne, jeśli okienko nie istnieje to zostanie utworzone
`xselect()`	Powoduje że dane okienko graficzne staje się aktywne, jeśli okienko nie istnieje to zostanie utwożone
`xbasc([num])`	Czyści dane okno graficzne
`xdel([num])`	Usuwa dane okienko graficzne

Ogólnie jeśli wybierzemy okienko graficzne bieżące, na przykład poleceniem `xset ("window", num)`[1], rodzina instrukcji `xset(ńom",a1,a2,...)` umożliwia ustawienie wszystkich parametrów rządzących sposobem wyświetlania informacji graficznych. Począwszy od kroju czcionki, poprzez grubości i kolor linii a skończywszy na opisie w postaci legendy. Zespół parametrów opisujących ustawienia w okienku graficznych nazywa się konstekstem graficznym.

Szczegółowe informacje dotyczące kontekstu, jego ustawiania i kontroli zawarte są w helpie, rozdział Graphic Library. Część z nich jest również dostępna z poziomu rozwijanego menu okna graficznego. Grupa instrukcji [a1,a2,...]=xget('nom') umożliwia kontrolę różnych parametrów danego okna graficznego. Scilab umożliwia również skalowanie rysunków kreślonych w obrębie okna graficznego, istnieje kilka wariantów tego polecenia:

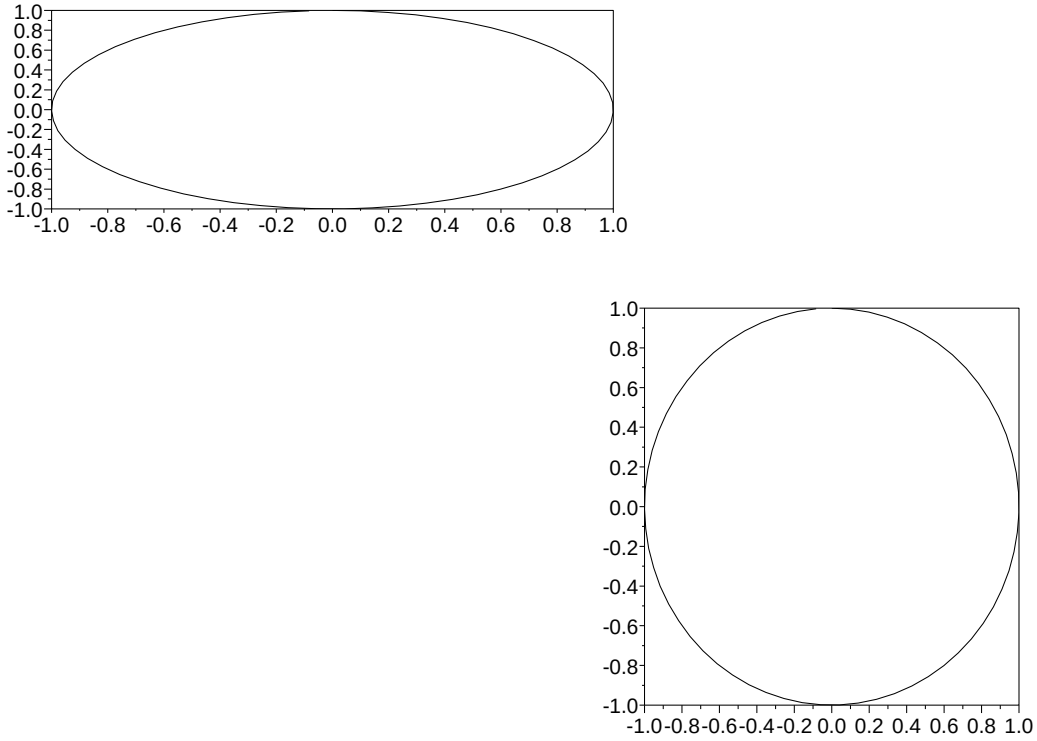
— `isoview()` - umożliwia utworzenie prostokątnego okna graficznego
— `square()` - umożliwia zmianę skali izometrycznego rysunku
— `xsetech()` - definiuje rodzajsub okna w obrębie danego okna graficznego

Specyfikę poleceń najprościej dostrzec na przykładach, tu pokazano przykład użycia polecenia `xsetech()`.

```
t=(0:0.1:2*%pi)';
```

[1] Więcej w helpie Scilaba

```
xsetech([0.,0.,0.6,0.3],[-1,1,-1,1]);

plot2d(sin(t),cos(t));

xsetech([0.5,0.3,0.4,0.6],[-1,1,-1,1]);

plot2d(sin(t),cos(t));
```



Rysunek 4.1. Efekt użycia instrukcji `xsetch()`

Jak widać na rys. 4.1 pierwszy raz okrąg został wykonany jako elipsa a drugi raz jako okrąg.

4.2. Kreślenie prostej grafiki

Załóżmy że chcemy wykreślić funkcje $y = e^x sin(4x)$ w przedziale $x \in [0, 2\pi]$. Na początek poleceniem `linspace()` definiujemy gęstość podziału na wartości dyskretne przedziału zmienności zmiennej niezależnej x.

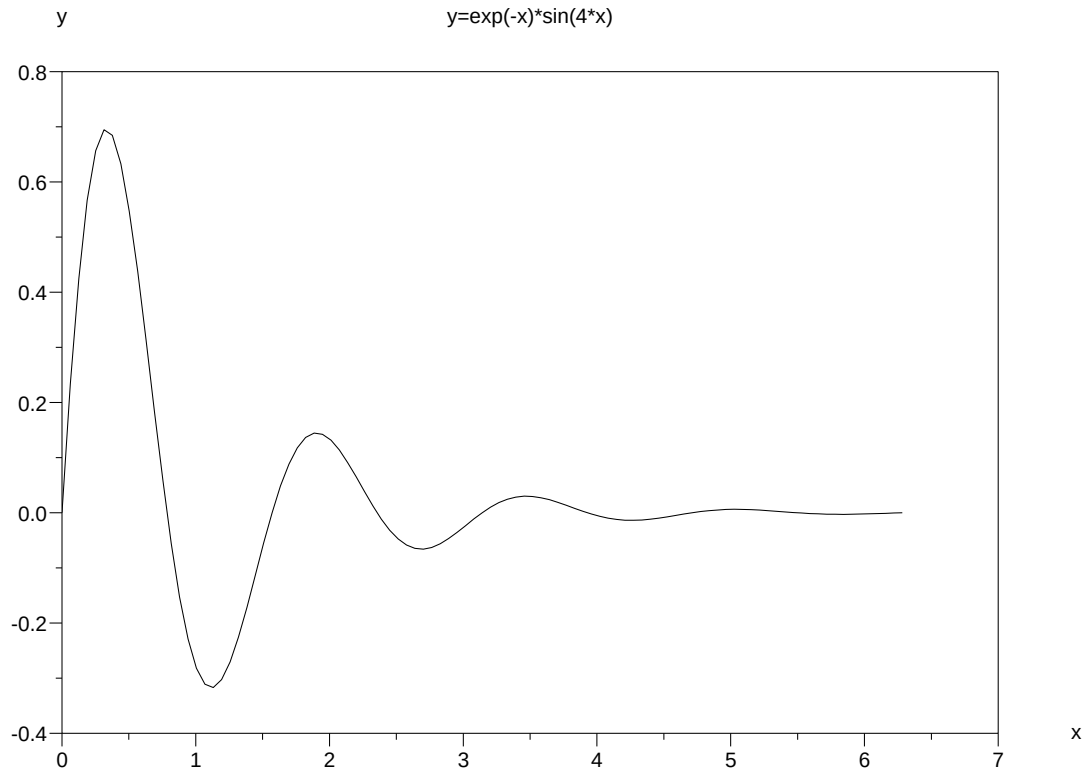
```
x=linspace(0,2*%pi,101);
```

Następnie obliczamy wartości naszego wyrażenia dla wektora zmiennej x [2].

```
y=exp(-x).*sin(4*x);
```

W końcu kreślimy grafike.

```
plot(x,y,'x','y','y=exp(-x)*sin(4*x)')
```



Rysunek 4.2. Wykres funkcji $y = e^{-x}\sin(4x)$

4.3. Polecenie plot2d()

Podstawowym poleceniem graficznym w obszarze rysunków dwuwymiarowych jest polecenie `plot2d()`.

4.3.1. Przykład wprowadzający

Prześledźmy działanie polecenia `plot2d()` na bazie prostego skryptu, który będziemy stopniowo uzupełniać o kolejne polecenia:

[2] zwróćmy uwagę, że nie ma potrzeby wykorzystywania instrukcji for ... jako że wykorzystujemy mnożenie tablicowe.

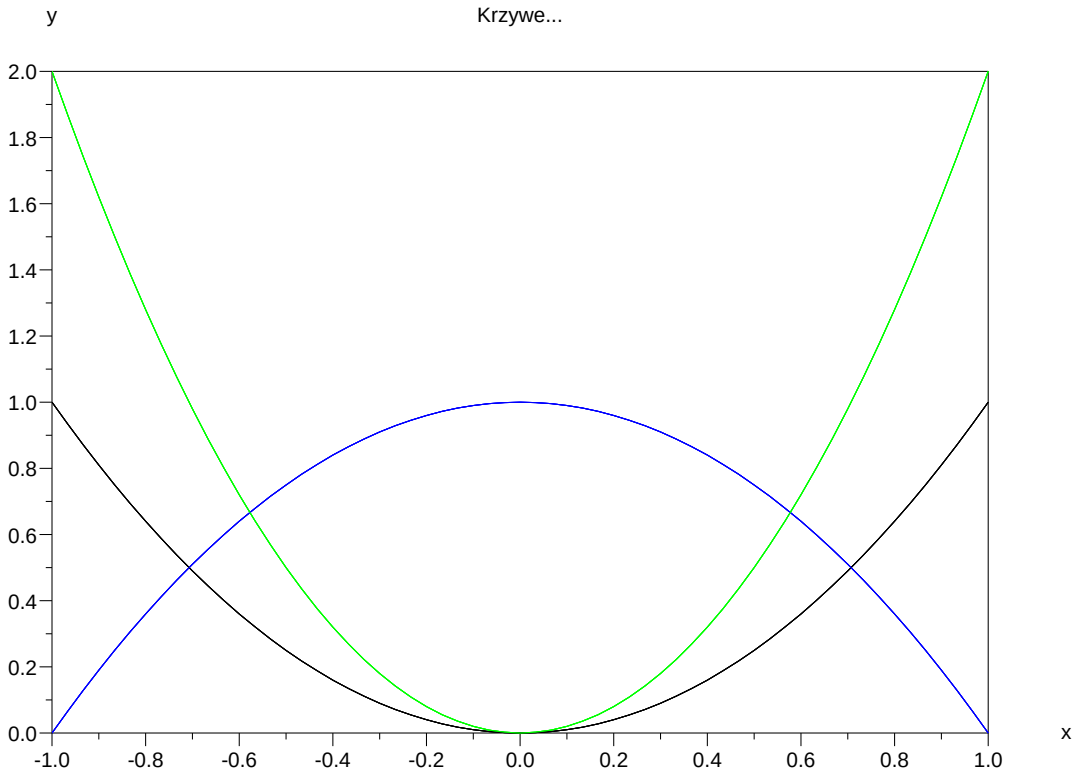
```
x=linspace(-1,1,61)'; //definiowanie argumentów

y=x.^2;   //obliczanie rzędnych ( potęgowanie tablicowe)

plot2d(x,y); // kreślenie
```

Należy pamiętać że zarówno zmienna 'x' jak 'y' są wektorami kolumnowymi. Dopiszmy do naszego skryptu kolejną funkcje:

```
ybis=1-x.^2;

plot2d(x,ybis)
```

W tym przypadku wartości rzędnych są obliczane dla tych samych argumentów co poprzednio. Dorzućmy kolejną trzecią funkcję:

```
yter=2*y;

plot2d(x,yter)
```



Rysunek 4.3. Użycie polecenia `plot2d() do kreślenia wykresów funkcji` x^2, $1-x^2$, $2x^2$

Jak można dostrzec na rysunku 4.3, Scilab tak dopasował zakres zmienności rzędnych funkcji, aby wszystkie trzy zmieściły się na rysunku.
Jeśli chcemy aby rysunek był do wykorzystania w innych programach dobrze jest go uzupełnić o tytuł.

```
-->xtitle("Krzywe...")
```

Pewnym uproszczeniem pozwalającym kreślić więcej niż jedną funkcję przy użyciu jednego polecenia plot2d() jest konstrukcja:

```
xbasc()

plot2d([x x x],[y ybis yter])

xtitle("Krzywe...","x","y")
```

Zwróćmy uwagę, że możliwe jest wykorzystanie polecenia `plot2d()` w postaci `plot2d(Mx,My)` gdzie `Mx` i `My` to dwie macierze o tych samych wymiarach. Liczba krzywych `nc` jest równa liczbie kolumn, krzywe są otrzymywane jako wektory kolumnowe odpowiednich macierzy `Mx(:,i)` (odcięte) i `My(:,i)` (rzędne). Na koniec tego podpunktu „nieco" bardziej zaawansowany przykład.

```
deff('[st]=pole(r)','st=4*%pi*r^2');
r=linspace(0,10,30)';
plot2d(r,pole(r)');
```

Przy jego konstruowaniu wykorzystano zdefiniowaną funkcję typu `inline`.

4.3.2. Składnia polecenia plot2d

Ogólna składnia polecenia plot2d() jest następująca:

```
plot2di(Mx,My,[style,strf,leg,rect,nax])
```

Index może być zastąpiony na pięć różnych sposobów

„i" opuszczamy skalowanie liniowe
i=1 tak jak wyżej z możliwościami użycia skali logarytmicznej
i=2 kreślenie metodą schodkową
i=3 pionowe słupki

Przykładowy skrypt poniżej generuje rysunek (4.4) który wszystko objaśni.

```
t=(1:0.1:8)';

xset("font",2,3);

xsetech([0.,0.,0.5,0.5],[-1,1,-1,1]);

plot2d([t,t],[1.5+0.2*sin(t) 2+cos(t)]);
```

```
xtitle('Kreslenie odcinkami liniowymi - Plot2d');

xsetech([0.5,0.,0.5,0.5],[-1,1,-1,1]);

plot2d1('oll',t,[1.5+0.2*sin(t) 2+cos(t)]);


xtitle('Skalowanie logarytmiczne - Plot2d1')

xsetech([0.,0.5,0.5,0.5],[-1,1,-1,1]);

plot2d2('onn',t,[1.5+0.2*sin(t) 2+cos(t)]);


xtitle(' Kreslenie metodą schodkow - Plot2d3')

xsetech([0.5,0.5,0.5,0.5],[-1,1,-1,1]);

plot2d3('onn',t,[1.5+0.2*sin(t) 2+cos(t)]);


xtitle('Slupki pionowe Plot2d3')
```

4.3.3. Różne warianty grafiki dwu wymiarowej

Prosty przykład grafiki 2D z siatką pomocniczą

```
t=(0:0.1:6*%pi);
plot2d(t',sin(t)');
xtitle('plot2d and xgrid ','t','sin(t)');
xgrid();
```

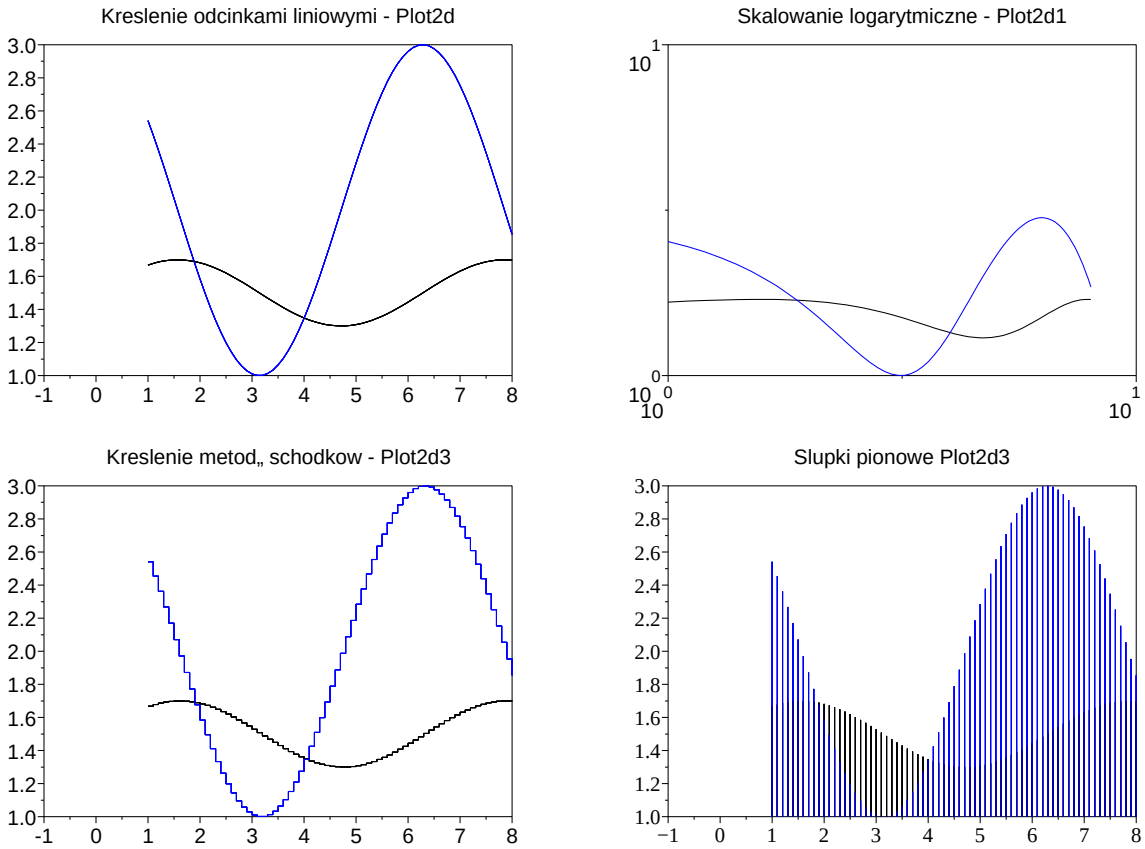
Wykres słupkowy

Poniższy przykład pokazuje sposób kreślenia wykresu słupkowego na przykładzie danych uzyskanych po wykonaniu transformaty Fouriera na funkcji `u`.

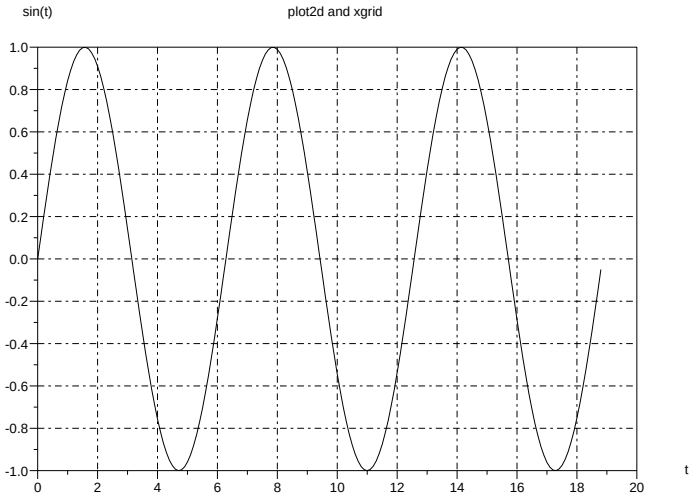
```
n=32-1;t=(0:n)/n;
u=sin(80*%pi*t)+sin(100*%pi*t);
plot2d3('enn',1,abs(fft(u,-1))'/n);
xtitle('plot2d3 (vbarplot)','t','f(t)');
```

Histogram

Przykład histogramu wartości losowych o rozkładzie normalnym zapisanych w wektorze `d` i podzielonych na 20 klas.

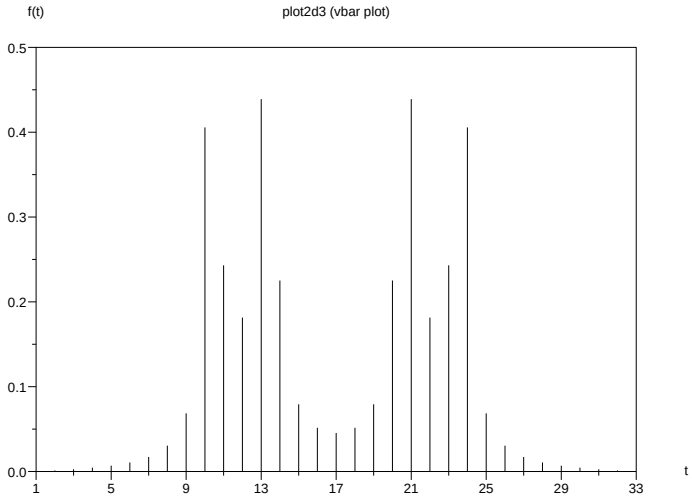


Rysunek 4.4. Różne warianty polecenia `plot2d()`



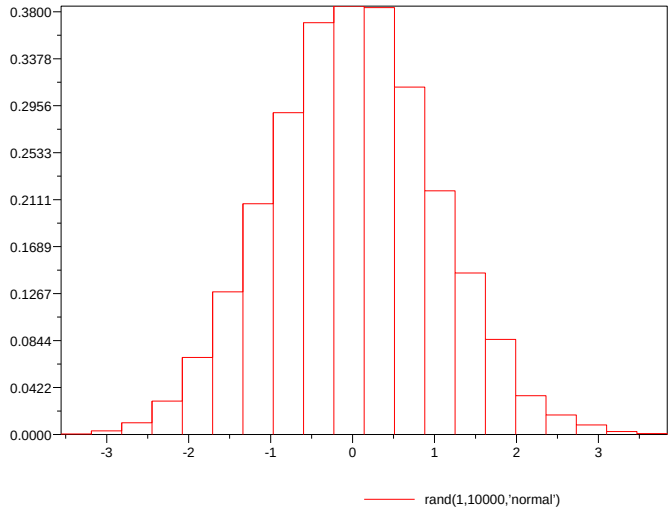
Rysunek 4.5. Prosty przykład grafiki 2D z siatką pomocniczą

```
d=rand(1,10000,'normal');
```



Rysunek 4.6. Wykres słupkowy

```
//clf();histplot(20,d)
//clf();histplot(20,d,normalization=%f)
clf();histplot(20,d,leg='rand(1,10000,''normal'')',style=5)
```



Rysunek 4.7. Histogram dwuwymiarowy

4.4. Polecenie `plot3d()`

Podobnie jak to miało miejsce w odniesieniu do grafiki dwu wymiarowej grafika trójwymiarowa może być wykonana na wiele różnych sposobów. Głównym poleceniem jest polecenie `plot3d()`.

4.4.1. Przykład wprowadzający

Jeśli nasza funkcja jest zadana równaniem postaci `z=f(x,y)`, jest szczególnie łatwo przedstawić ja w postaci graficznej w prostokątnym obszarze. Przykładowo weźmy funkcję `f(x,y)=cos(x)*sin(y)` - odpowiedni zestaw poleceni Scilaba do jej wizualizacji jest następujący.

```
x=linspace(0,2*%pi,31);

z=cos(x)'*cos(x);

plot3d(x,x,z)
```

Proszę zwrócić uwagę na konieczność wykonania transpozycji pierwszej funkcji $cos(x)$. Wiąże się to z faktem, że budujemy macierz z.

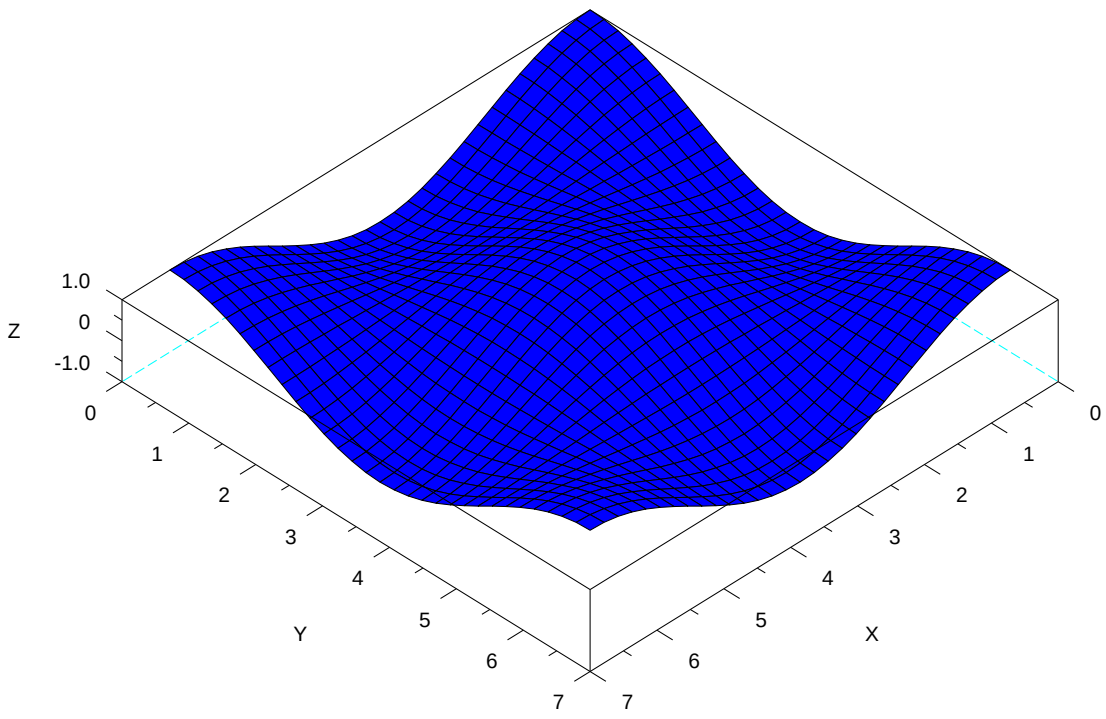
4.4.2. Składnia polecenia `plot3d()`

Poprzedni przykład pokazał, że wykorzystanie polecenia `plot3d()` jest bardzo proste. Tym bardziej trzeba mieć świadomość, że polecenie to ma bardzo duże możliwości kontrolowanych przez grupę parametrów opcjonalnych.

```
plot3d(x,y,z [,theta,alpha,leg [,flag,ebox]])
```

gdzie:
- `x` i `y` – reprezentują dwa wektory wierszowe `((1,nx) i (1,ny))` odpowiadające zmiennym `x` i `y`.
- `z` – jest macierzą o wymiarach `(nx,ny)` odpowiadającą wartościom funkcji
- `theta` i `alpha` – to wartości kąta określającego punkt widzenia kreślonej powierzchni.
- `leg` – jest łańcuchem pozwalającym otrzymać legendę dla każdej z osi przykładowo $leg =$ "$x@y@z$"
- flag – jest wektorem o trzech składowych `flag=[mode type box]`
  - — Parametr mode -
  - — Parametr type -
  - — Parametr box -

Ze względu na bardzo rozbudowane możliwości polecenia `plot3d()`
nie omawiamy szczegółowo jego możliwości, polecając lekturę helpu programu Scilab. Tytułem przykładu pokazujemy natomiast skrypt pokazujący pełnię możliwości



Rysunek 4.8. Różne warianty polecenia `plot3d()`

polecenia `plot3d()`. Jego skopiowanie i uruchomienie w systemie Scilab da pogląd o możliwościach graficznych danych przez polecenie `plot3d()`.

```
// prosty wykres z=f(x,y)
xset("window",1);
t=[0:0.3:2*%pi]';
z=sin(t)*cos(t');
plot3d(t,t,z)


// ten sam co poprzenia z same plot using facets computed by genfac3d
xset("window",2);
[xx,yy,zz]=genfac3d(t,t,z);
plot3d(xx,yy,zz)

// wielokrotny wykres
xset("window",3);
clf()
plot3d([xx xx],[yy yy],[zz 4+zz])
```

```
// wielokrotnuy wykres z uzyciem koloru
xset("window",4);
clf()
plot3d([xx xx],[yy yy],list([zz zz+4],[4*ones(1,400) 5*ones(1,400)]))

// prosty wykres z ustawieniem punktu widzenia i opisem osi
xset("window",5);
clf()
plot3d(1:10,1:20,10*rand(10,20),alpha=35,theta=45,flag=[2,2,3])

// kreslenia sfery tworzenie fsetek przy uzyciu eval3dp
xset("window",6);
deff("[x,y,z]=sph(alp,tet)",["x=r*cos(alp).*cos(tet)+orig(1)*ones(tet)";..
"y=r*cos(alp).*sin(tet)+orig(2)*ones(tet)";..
"z=r*sin(alp)+orig(3)*ones(tet)"]);
r=1;
orig=[0 0 0];
[xx,yy,zz]=eval3dp(sph,linspace(-%pi/2,%pi/2,40),linspace(0,%pi*2,20));
clf();
plot3d(xx,yy,zz)
```

4.5. Uwagi dodatkowe

Ze względu na bardzo duże możliwości graficzne nie jest możliwe nawet skrótowe ich omówienie. Dlatego gorąco zachęcamy do studiowania helpu i testowaniu zamieszczonych tam przykładów.

4.5.1. Różne warianty kreślenia grafiki

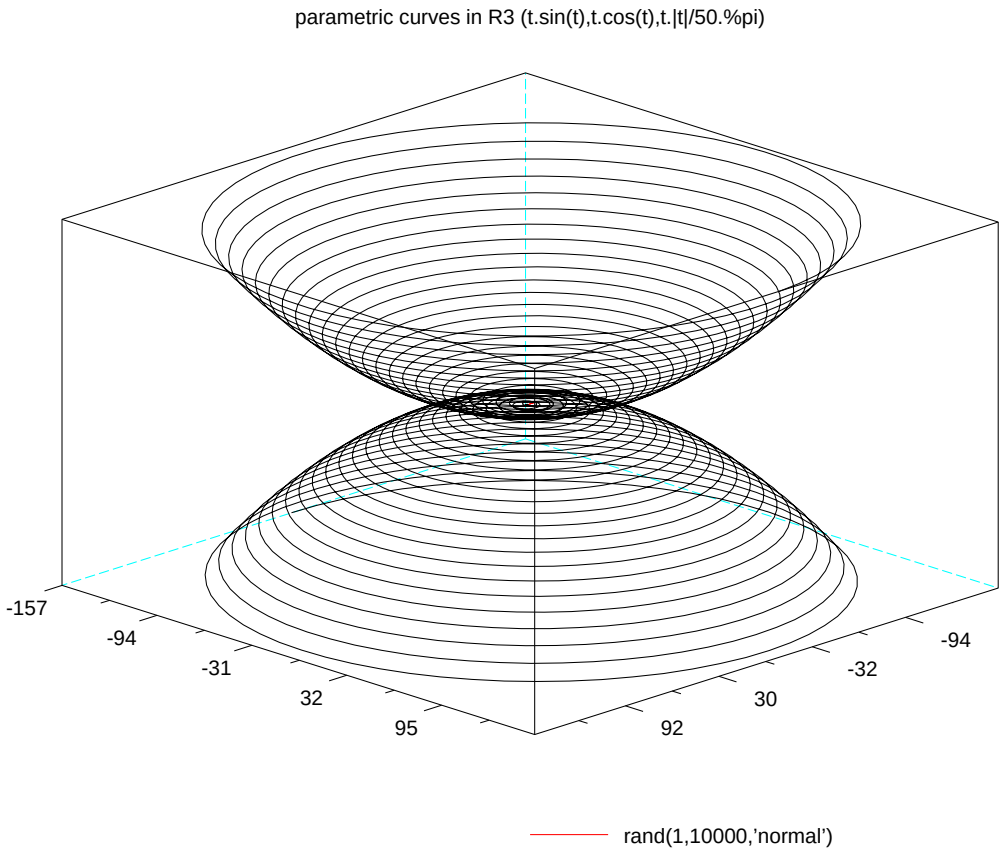
Wykres parametryczny

```
t=-50*%pi:0.1:50*%pi;
x=t.*sin(t);y=t.*cos(t);z=t.*abs(t)/(50*%pi);
param3d(x,y,z,45,60);
title='parametric curves in R3 (t.sin(t),t.cos(t),t.|t|/50.%pi)';
xtitle(title,' ',' ');
```

Polecenie `subplot()`

Polecenie `subplot()` pozwala kreślić na jednym rysunku kilka różnych wykresów, każdy w innym oknie.

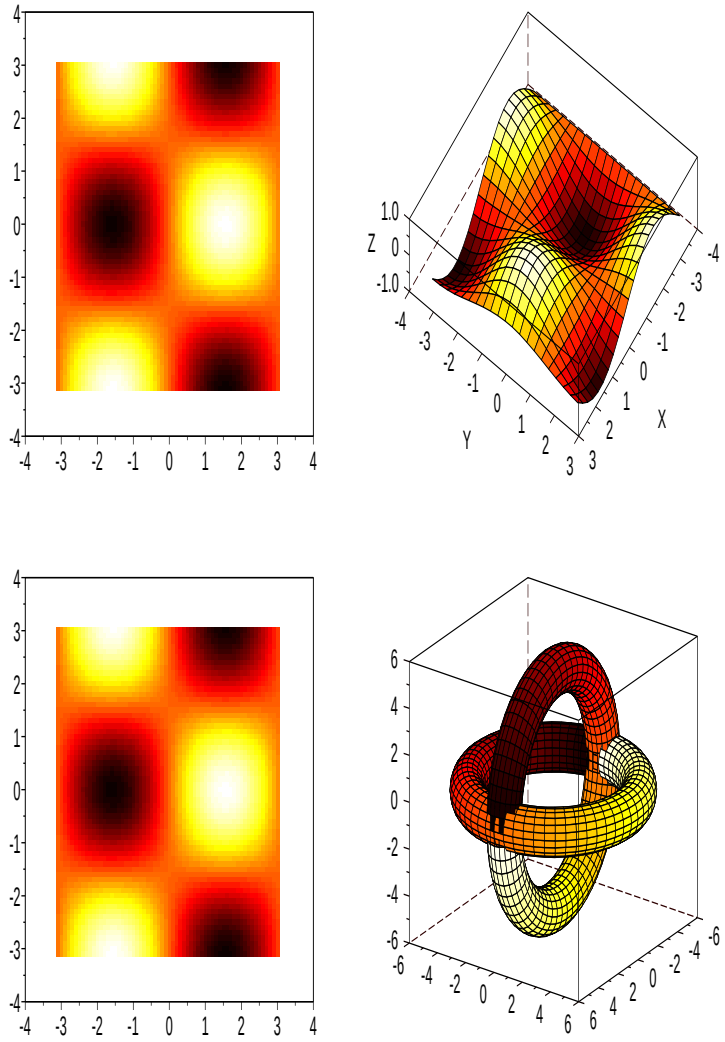
```
ncolor=228;
// pierwszy rysunek
xsetech([0.,0.,0.5,0.5],[-1,1,-1,1]); h=hotcolormap(ncolor);
```



Rysunek 4.9. Wykres funkcji zadanej parametrycznie

```
xset('colormap',h);
t=-%pi:0.1:%pi;m=sin(t)'*cos(t);grayplot(t,t,m);
//drugi rysunek
xsetech([0.5,0.,0.5,0.5],[-1,1,-1,1]); h1=((1:ncolor)'/ncolor)*[1
0 0]; xset('colormap',h1);
t=-%pi:0.3:%pi;plot3d1(t,t,sin(t)'*cos(t),35,45,'X@Y@Z',[2,2,4]);

// trzeci rysunek
xsetech([0.,0.5,0.5,0.5],[-1,1,-1,1]); h2=graycolormap(ncolor);
xset('colormap',h2);
t=-%pi:0.1:%pi;m=sin(t)'*cos(t);grayplot(t,t,m);

// czwarty rysunek
xsetech([0.5,0.5,0.5,0.5],[-1,1,-1,1]);
deff('[x,y,z]=knot(u,v)',['vv=ones(v)';'uu=ones(u);';
    'x=(5.*cos(u)+cos(u).*cos(v))';
    'y=(5.*sin(u)+sin(u).*cos(v))';
    'z=(uu.*sin(v))';])
nx=60;Nx=(0:nx)/nx ;ny=20;Ny=(0:ny)/ny;
```

```
[xx,yy,zz]=eval3dp(knot,2*%pi*Nx,2*%pi*Ny);
XXX=[-xx xx];YYY=[-yy zz];ZZZ=[-zz yy]; kk1=[1:size(zz,2)];
ncolor=60; hh=hotcolormap(ncolor); xset('colormap',hh) ;
kk1=modulo(kk1,60);kk2=kk1; KKK=list(ZZZ,[kk1 kk2]);
plot3d(XXX,YYY,KKK,35,70," @ @ ",[2,1,4],[-6,6,-6,6,-6,6]);
```



Rysunek 4.10. Wykorzystanie polecenia `subplot()`

4.6. Ćwiczenia

1. Zdefiniuj funkcję $f(x) = sinx + x$
2. Wykreśl funkcję w obszarze $x \in [0, 4 * \pi]$

3. Opisz osie utworzonego wykresu.
4. Wkreśl i opisz funkcje $f(x) = cos(x)/x$.

Rozdział 5

Wstęp do programowania

5.1. Instrukcje proste

Instrukcje są tym elementem języka który przekształca go, ze sprawnego ale tylko kalkulatora, do roli narzędzia do rozwiązywania dowolnie złożonych problemów, zapisanych za pomocą algorytmów.

5.1.1. Instrukcje pusta

W tym przypadku składa się ona tylko ze znaku „ ; ” lub pustej linii w skrypcie. Powoduje ona tylko przekazanie sterowania dalej do następnej linii.

5.1.2. Instrukcja przypisania

Instrukcja ta powoduje przypisanie zawartości zmiennej znajdującej się po prawej stronie znaku „ = ” zmiennej znajdującej się po jego lewej stronie. Nie należy mylić działania tej instrukcji z funkcją porównywania wielkości znajdujących się po obu jej stronach.

5.1.3. Sekwencja kilku instrukcji

Jak w każdym języku programowania dopuszcza się zapisanie kilku instrukcji w jednej linii, rozdzielonych średnikami „ ; ” lub przecinkiem „ , ”.

5.2. Instrukcje warunkowe

Z pomocą instrukcji warunkowych możemy sterować przebiegiem procesu obliczeń w zależności od spełniania lub nie spełniania pewnych warunków logicznych. Najprostszą instrukcją tego typu jest instrukcja warunkowa.

```
if war_log1, inst1
```

Jeśli wyrażenie *war_log1* jest prawdą to wykonywane są instrukcje *inst1*. Poniżej pokazano przykład działania instrukcji warunkowej. Jeżeli wartość zmiennej `a` jest równa `4` to w wyniku jej działania zostanie powiększona o kolejne `4`.

```
a=4; if a==4, a=a+4,end
```

Jeśli wartość zmiennej `a` jest równa `6` a więc inna niż `4`, to w wyniku działania tej instrukcji wartość `a` pozostanie bez zmiany, czyli równa `4`.

```
a=6; if a==4, a=a+4;end
```

Powyższą grupę instrukcję można uprościć zapisując jak poniżej.

```
if a==4  a=a+4, else disp(a) end
```

Jej ogólna wersja w przypadku pisania skryptów jest następująca.

MATLAB	SCILAB†
`if war_log1`	`if war_log1 then`
`  inst1`	`  inst1`
`else`	`else`
`  inst2`	`  inst2`
`end`	`end`

† Polecenia `then` może być zastąpione przez polecenie `do`, "`,`" lub można z niego zrezygnować wciskając klawisz *<RETURN>*.

W dalszej części pracy można zrezygnować z „SCILABOWYCH" poleceń `then`, `do` na korzyść możliwej zgodności poleceń warunkowych `if ...else ...end` z MATLABEM. W miejsce `inst1` lub `inst2` możemy wstawić kolejną instrukcję warunkową. Należy pamiętać o konieczności kończenia każdej instrukcji warunkowej poleceniem `end`. Jeżeli instrukcja warunkowa jest zapisywana w jednej linii, lub wyrażenie `inst` składa się z kilku poleceń, to należy rozdzielić je znakiem „ , " lub „ ; ".

Bardziej rozbudowana wersja instrukcji warunkowej może być następująca:

```
if war_log1
  inst1
 else
   if war_log2
      inst2
     else
       if war_log3
          inst3
       else
          inst4
     end
 end
end
```

Powyższą konstrukcję można uprościć komendą `elseif`.

```
if war_log1
  ciąg_inst1
 elseif war_log2
  inst2
  elseif war_log3
  inst3
  else      // w Matlabie możemy również napisać otherwise
  inst4
end
```

Wykorzystanie instrukcji warunkowej wiąże się z koniecznością wykorzystywania operatorów operacji logicznych (Tablica 5.1).

Tablica 5.1. Operatory operacji logicznych

Matlab	Scilab	Opis
==	==	równość lewej i prawej strony
<	<	mniejsze
<=	<=	mniejsze bądź równe
>	>	większe
>=	>=	większe równe
<>	<>	nierówne
&	&	logiczne i (and)
|	|	logiczne lub (or)
~	~	negacja (zaprzeczenie)

5.2.1. Przykłady

Przykład pokazuje sposób wykorzystania instrukcji warunkowej do sprawdzania czy dana macierz jest macierzą rzeczywistą czy zespoloną. Załóżmy że w programie pojawia się macierz A.

```
 A  =

!   0.2113249    0.3303271    0.8497452 !
!   0.7560439    0.6653811    0.6857310 !
!   0.0002211    0.6283918    0.8782165 !
```

sekwencja poleceń sprawdza rodzaj macierzy

```
if imag(A)==0 then
disp('A jest macierzą rzeczywistą')
else
disp('A jest macierzą zespoloną');
end
```

5.3. Instrukcja wariantowego wyboru

Inną instrukcją warunkową jest to polecenie `switch ... case` (MATLAB) i `select ... case` (SCILAB). Prostą wersję tej instrukcji, wpisywanej bezpośrednio z klawiatury, pokazano poniżej.

```
liczba=1, select  liczba, case 1, y='war 1',case 2, y='war 2, ...
else, y='inna war', end
```

Należy pamiętać o konieczności pisania przecinków lub średników po słowach kluczowych i instrukcjach. Trzy kropki oznaczają tu kontynuację instrukcji przy jej wpisywania bezpośrednio z poziomu interpretera. W przypadku wykorzystywania skryptów instrukcja może być zapisana jak to pokazano niżej:

```
liczba=1
select liczba // w Matlabie piszemy switch
  case 1
    y=war1
  case 2
    y=war2
 else
  y=innawar
end
```

Bardziej złożoną wersje obu instrukcji pokazano dalej.

```
if war1
 inst1
elseif
if war2
 else
select wyr_1
 case war_wyr1
inst3
 case war_wyr2
inst4 else
 inst5
```

5.3.1. Przykłady

W pokazanym poniżej przykładzie wygenerowano losowo liczbę naturalną `x` z zakresu $\{1, 2, \ldots, 10\}$. Jeśli $x = 1$ lub $x = 2$ to liczba jest wyświetlana na ekranie w pozostałych przypadkach wyświetlany jest napis ,,`inna wartość`''.

MATLAB †	SCILAB‡
`n=round(10*rand(1,1));`	`n=round(10*rand(1,1));`
`switch n`	`select n`
`case 0`	`case 0`
`disp(0);`	`disp(0);`
`case 1`	`case 1`
`disp(1);`	`disp(1);`
`otherwise`	`else`
`disp('inna wartosc');`	`disp('inna wartosc');`
`end`	`end`

† Polecenia `select otherwise` są charakterystyczne tylko dla MATLABA.
‡ Polecenia `select else` są dostępne tylko w SCILABIE.

5.4. Pętle

Instrukcje definiujące pętle pozwala użytkownikowi powtarzać grupę instrukcji. Istnieją dwie postacie tej instrukcji, pętla `for` i pętla `while`.

5.4.1. Pętla `for`

Pętla typu `for` pozwala powtarzać pojedynczą instrukcję lub ich grupę tak długo aż zmienna (dalej nazwana *index*) kontrolująca ilość wykonanych powtórzeń pętli osiągnie wartość ustawioną jako końcową. Składnia tej instrukcji jest następująca:

```
for indeks = wart_początkowa:krok:wart_końcowa do ciąg_instrukcji end
```

W powyższym zapisie pojawiają się słowa kluczowe `for do end` definiujące pętlę. Konstrukcja *indeks = wart_początkowa:krok:wart_końcowa* która może pojawiać się również w wersji uproszczonej *indeks = wart_początkowa:wart_końcowa* określa ilość powtórzeń bloku instrukcji oznaczonego słowem *ciąg_instrukcji*. Zmienna *wart_początkowa* odpowiada wartości początkowej zmiennej *index*, *wart_końcowa* wartość końcową która nie zostanie przekroczona, jej osiągniecie oznacza koniec pętli, *krok* określa przyrost zmiennej *index* w jednym kroku iteracyjnym. Jeśli jej wartość jest równa 1 to można ją pominąć.

5.4.2. Przykłady

Poniżej pokazano sposób obliczania iloczynu liczb naturalnych w zakresie od 1 do 20.

```
a=1;
for n=1:  20 do
a=a*n
end
a=
a=2.433E+8
```

Drugi przykład pokazuje jak można obliczyć sumę odwrotności pierwszych 10 liczb

naturalnych $\sum_{i=1}^{i=10} \frac{1}{i}$.

```
suma=0;
for i=1:10 do suma=suma+1/i; end
suma
suma=2.9289683
```

Trzeci przykład pokazuje że wartość końcowa pętli może nie być osiągnięta - zmienna k nigdy nie jest równa 10.

```
for k=1:4:10 do disp(k); end
1.
5.
9.
```

5.4.3. Pętla *while*

Inną ale równie użyteczną instrukcją definiującą pętle jest instrukcja *while* która pozwala nam powtarzać grupę instrukcji aż pewien określony w niej warunek logiczny nie zostanie spełniony.

```
while wyrażenie do ciąg_instrukcji end
```

Dopóki *wyrażenie* przyjmuje wartość logiczną prawda (true) wykonywany jest *ciąg_instrukcji* zawarty pomiędzy słowami kluczowymi `do ... end`. Pętla `while ... do ... end` może być przerwana za pomocą polecenia `break`.
Instrukcję `while` można uzupełnić o komendę `else` i wspomnianą już instrukcją `break`.

```
while wyrażenie do ciąg_instrukcji1 else ciąg_instrukcji2 end
```

W tym przypadku niespełnienie warunku logicznego *wyrażenie* nie powoduje ukończenia pętli a przejście do wykonywania instrukcji zapisanych symbolicznie *ciąg_instrukcji2*. Opuszczenie pętli wymaga więc użycia polecenia `break`.

5.4.4. Przykład

Pierwszy przykład pokazuje wykorzystanie pętli *while* do obliczania sumy liczb od 1 do 100.

```
a=0;
n=1; while n<=100 do
a=a+n;
n=n+1;
end
```

```
a
a=5050
```

Przeznaczenie drugiego przykładu powinien odgadnąć czytelnik

```
i=10; while i>0
do
i = i-3
if i==7 then break; end
else
disp(i)
end
```

5.5. Ćwiczenia

1. Napisz program (funkcję) obliczającą wartość całki metodą MonteCarlo, jeśli funkcja jest zadawana w postaci `deff()`.

Rozdział 6

Sztuka programowania

6.1. Kilka zdań na początek

Uznając programowania, może nieco na wyrost, za formę sztuki nie jest możliwe jej nauczenie. Można jednak pokusić się o podanie pewnych wskazówek dla jej adeptów, które mogą ułatwić pisanie programów krótkich, optymalnych i szybkich realizujących bezbłędnie postawione im zadania.

6.1.1. Niebezpieczeństwa

Podstawowym ograniczeniem na jakie napotyka programista, jest brak czasu jaki może poświęcić na pisanie programu. Z reguły nie ma czasu na przemyślenie problemu, brak go sformułowanie optymalnego algorytmu, a sam program powstaje w pośpiechu drogą kolejnych ulepszeń - niestety nieulepszalnego kodu wstępnego. Miary trudności dopełnia zawiła struktura pozbawionych komentarza[1] linii kodu. Z własnego doświadczenia autora wynika, że program nie mieszczący się na kartce formatu A4, po tygodniu najwyżej dwu staje się, niejasny a czasami wręcz niezrozumiały - nawet dla jego autora.
Żartobliwie można sformułować kilka prawd, które dotyczą programów - a co za tym idzie programistów.

— Błędy niewykrywalne są nieskończenie razy częstsze niż te które możemy wykryć.
— Większa część czasu działania programu przypada na czynności niezwiązane z rozwiązywanym przez niego problemem, który i tak będzie rozwiązany źle.
— Zadania testowe pokazują że program działa źle, nigdy że działa dobrze.

6.1.2. Kilka prostych zasad

— Zanim zabierzemy się do pisanie programu komputerowego, analizowany problem należy rozwiązać w sposób klasyczny na kartce papieru, oczywiście w takim zakresie jakim jest to możliwe. Pominięcie tego kroku jest gwarancją klęski.

— Każdy program musi mieć jasno określone parametry wejściowe i wyjściowe.
— Nazwy zmiennych w programie powinny odzwierciedlać ich role w algorytmie.

[1] Czasami twórca uzupełnia swój program o tak zwany User Manual napisany w formacie RTFm « Read The F*ck*ng manual»

6.2. Pisanie programów wydajnych

Dobrze napisany program jest wynikiem spełnienia dwu warunków:
— dobrego zrozumienia wybranego algorytmu,
— biegłej znajomości używanego narzędzia.

6.2.1. Znaczenie algorytmu

Rozwiązanie problemu jest niejednokrotnie utożsamiane z znalezieniem właściwego algorytmu. Podstawowym niebezpieczeństwem na jakie napotykamy jest skłonność do bezkrytycznego wykorzystywanie znalezionego rozwiązania w swoim programie. Jako ilustracje rozważmy zadanie wyznaczania liczb Fibonacciego. Definicja liczby Fibonacciego jest następująca, każda następna jest sumą dwu poprzednich.

$$F_k = \begin{cases} 1, & \text{k=1,2} \\ F_{k-1} + F_{k-2}, & \text{k}\geq 3 \end{cases} \tag{6.2.1}$$

Bezpośrednia implementacja zależności (6.2.1) prowadzi do następującej definicji funkcji.

```
function r=fib(n)
 if n <=2
   r=1;
 else
   r=fib(n-1)+fib(n-2);
 end
```

wyznaczenie piątej z kolei , licząc od 1, liczby Fibonacciego wymaga 9 wywołań funkcji `fib()`, jest to dużo i wydaje się że procedura jest bardzo „czasożerna”. Znaczące uproszczenie można uzyskać drogą uproszczenia podwójnego wywołania rekurencyjnego `r=fib(n-1)+fib(n-2)` zastępujemy go pozornie bardziej skomplikowanym rozwiązaniem jakim jest funkcja obliczając wartość `fib(n+1)`

```
function Z=fib_aux(N,n,Un,Un_1)
 if N>n
   z=fib_aux(N,n+1,Un+Un_1,Un);
  else
   Z=Un;
 end
```

Funkcję `fib_aux()` wywołujemy w nowej poprawionej wersji funkcji obliczającej liczby Fibonacciego.

```
function Z=fib2(N)
 U0=0;
 U1=1;
 if N==0
    Z=U0;
  else
```

```
    Z=fib_aux(N,1,U1,U0);
 end
```

Funkcję `fib2()` można uprościć

```
 Z=fib_aux(N,0,0,1);
```

Znaczące przyśpieszenie czasu obliczeń uzyskamy rezygnując z rekurencyjnego sposobu obliczania liczb Fibonacciego na korzyść metody iteracyjnej.

```
function Z=fib3(N)
 Un=0;
 Un_1=1;
 n=0;
 while n<N do
  New=Un+Un_1;
  Un_1=Un;
  Un=New;
  n=n+1;
 end
 Z=Un;
```

Czas obliczeń 30 liczby Fibonacciego spada z 108090 milisekund dla `fib()` do 1.014 milisekundy dla `fib3()`.

6.3. Silne i słabe strony interpretowanych języków programowania

6.3.1. Punkty silne

Podstawową siłą języka jest jego macierzowy charakter. Należy więc konsekwentnie wykorzystywać operatory działań na macierzach jako szybsze bardziej optymalne niż jakiekolwiek konstrukcje przeniesione z klasycznych języków programowania. Najmocniejszą stroną języka jest ogromna ilość predefiniowanych funkcji, gruntownie przetestowanych, które mogą ułatwić rozwiązanie każdego problemu obliczeniowego.

6.3.2. Punkty słabe

Słabe punkty języka ujawniają się w chwili gdy zmuszeni jesteśmy pisać własne funkcje. Niestety kompilowane języki wysokiego poziomu dają tu o wiele większe możliwości. Może dlatego większość z wbudowanych funkcji napisano w języku C lub Fortranie.

Rozdział 7

Wejście wyjście

Zdecydowana większość programów współpracuje z zbiorami plikowymi. Prędzej czy później użytkownik zawsze stanie przed koniecznością zapisu bądź odczytu zbioru danych. Zbiory plikowe są najczęściej używaną formą wymiany danych pomiędzy różnymi programami. Zważywszy na fakt, że programy mogą przechowywać dane w różny sposób, to i sposób ich zapisu lub odczytu może odbywać się na wiele różnych sposobów. Generalnie zbiory mogą być przechowywane w formacie ASCII i binarnym. Pliki ASCII można otwierać i modyfikować posługując się dowolnym edytorem tekstu. Pliki binarne nie mogą być podglądane w edytorach tekstowych, ale możemy mieć dostęp do ich zawartości, poprzez odpowiednie polecenia programowe, umożliwiające czytanie i poprawną interpretacje ciągów bajtów.

7.1. Zapis odczyt zmiennych używanych w programie

Zapis zmiennych występujących w programie możemy zrealizować w następujący sposób. Na początek generujemy oczywiście macierz a.

```
a = rand(3,3)
a  =
 !   0.2113249    0.3303271    0.8497452 !
 !   0.7560439    0.6653811    0.6857310 !
 !   0.0002211    0.6283918    0.8782165 !
```

Zapisujemy macierz `a` na dysku w pliku `a`.

```
save a
```

Usuwanie zmiennych z pamięci programu.

```
clear


a
  !--error     4
undefined variable : a
```

Powtórne wczytywanie macierzy a, tym razem z pliku.

```
load a


a
```

```
 a  =
 !   0.2113249    0.3303271    0.8497452 !
 !   0.7560439    0.6653811    0.6857310 !
 !   0.0002211    0.6283918    0.8782165 !
```

Jeśli teraz spojrzymy do bieżącego katalogu, to powinniśmy spostrzec plik o nazwie `a`, w którym zapisano zawartość zmiennej `a` występującej w naszym programie. Możemy również zapisywać naszą zmienną `a` poleceniem `save('namefile', nazwa_zmiennej)`, gdzie `'namefile'` oznacza nazwę pliku w którym umieściliśmy zawartość zmiennej `a`. Wszystkie zmienne występujące w naszym programie zapisujemy posługując się poleceniem `save('namefile')`. Aby odczytać zapisane wcześniej zmienne z pliku dyskowego wykorzystujemy polecenie `load`, które może odczytać wybraną lub wszystkie zmienne umieszczone z zbiorze plikowym. Przykład poniżej pokazuje sposób użycia poleceń `save` i `load`

```
 a = 1                        // tworzenie zmiennej a
 a  =
    1.

 b = 7:-1:5                   // tworzenie zmiennej b
 b  =
  !   7.    6.    5. !

 save('variables')            // zapis a i b w  'variables'

 clear                        // usuwanie zmiennych

 a                            // a nie może być znalezione
  !--error     4
undefined variable : a

 b                            // b nie może być znalezione
  !--error     4
undefined variable : b

load('variables','b')        // odczyt b z zbioru 'variables'

 b                            // wyświetlenie b
 b  =

!   7.    6.    5. !

 a                            // a wciąż jest niedostępne
  !--error     4
  undefined variable : a
```

```
clear                          // usuwanie wszystkich zmiennych

load('variables')              // odczyt  wszystkich zmiennych

a                              // wyświetlenie a
a  =
   1.

b                              // wyświetlenie  b
b  =

!   7.    6.    5. !
```

Zauważmy, że polecenia `save` i `load` nie wymagają żadnych dodatkowych poleceń otwierających lub zamykających zbiory. Dlatego ich wykorzystywanie jest względnie proste. Poleceniem ułatwiającym pracę z zbiorami plikowymi jest polecenie
`filename = xgetfile()`
gdzie: `filename` zawiera nazwę interesującego nas pliku wraz z pełną ścieżką dostępu do niego, może być ona wykorzystywana później jako argument w poleceniu `load(filename)`.

7.2. Zbiory typu ASCII

Bardzo często chcemy wykorzystywać zbiory danych wygenerowane przez inne aplikacje. Sprawa jest stosunkowo łatwa do rozwiązania, jeśli format danych jest kompatybilny pomiędzy różnymi aplikacjami. Tak jest z reguły, jeśli dane zapisane są w formacie ASCII. Zazwyczaj dane są zapisywane w postaci struktury macierzowej, gdzie poszczególne kolumny lub wiersze mają swoją ścisłą interpretacje, na przykład: jako wyniki pomiarów z kolejnych kanałów urządzenia pomiarowego. Załóżmy że chcemy uzyskać dostęp do danych zapisanych w formacie ASCII w pliku o nazwie `'test.dat'` który umieszczono w katalogu do którego aktualnie ma dostęp nasz program. Dane mogły być generowane przez inny program Matlabowy lub Scilabowy, utworzone w edytorze tekstowym lub zapisane w inny sposób. Ważna jest tu znajomość formatu tego zapisu i umiejętność interpretacji danych zapisanych w poszczególnych kolumnach lub wierszach.
Załóżmy że zawartość pliku `test.dat` jest następująca:

```
test.dat

0.2113249 0.5608486 0.3076091 0.5015342 0.2806498
0.7560439 0.6623569 0.9329616 0.4368588 0.1280058
0.0002211 0.7263507 0.2146008 0.2693125 0.7783129
0.3303271 0.1985144 0.3126420 0.6325745 0.2119030
0.6653811 0.5442573 0.3616361 0.4051954 0.1121355
0.6283918 0.2320748 0.2922267 0.9184708 0.6856896
0.8497452 0.2312237 0.5664249 0.0437334 0.1531217
```

```
0.6857310 0.2164633 0.4826472 0.4818509 0.6970851
0.8782165 0.8833888 0.3321719 0.2639556 0.8415518
0.0683740 0.6525135 0.5935095 0.4148104 0.4062025
```

Pokazana poniżej sekwencja poleceń pozwala na odczyt danych.

```
fid = file ('open','test.dat','unknown')
fid  =

   2.

data = read (fid, -1, 5)
data  =
  !   0.2113249    0.5608486    0.3076091    0.5015342   0.2806498 !
  !   0.7560439    0.6623569    0.9329616    0.4368588   0.1280058 !
  !   0.0002211    0.7263507    0.2146008    0.2693125   0.7783129 !
  !   0.3303271    0.1985144    0.312642     0.6325745   0.211903  !
  !   0.6653811    0.5442573    0.3616361    0.4051954   0.1121355 !
  !   0.6283918    0.2320748    0.2922267    0.9184708   0.6856896 !
  !   0.8497452    0.2312237    0.5664249    0.0437334   0.1531217 !
  !   0.685731     0.2164633    0.4826472    0.4818509   0.6970851 !
  !   0.8782165    0.8833888    0.3321719    0.2639556   0.8415518 !
  !   0.068374     0.6525135    0.5935095    0.4148104   0.4062025 !

file ('close' , fid)
```

Pewne elementy pokazanego powyżej kodu wymagają komentarza. Polecenie fortranowskie `file` jest wykorzystywane do otwierania i zamykania zbioru plikowego. Polecenie `read` jest wykorzystywane do odczytu. Cyfra `5` określa ile liczb zapisanych jest w każdym wierszu. Ponieważ nie wiemy ile wierszy jest w pliku dlatego pojawia się parametr `-1` wymuszający odczyt aż do końca pliku. Milcząco zakłada się, że format wiersza nie zmieni się do końca pliku. Należy również zauważyć, że próba odczytu kolejnych wierszy pliku nie powiedzie się, ponieważ osiągnięto koniec pliku.

Pewnym uproszczeniem powyższego polecenia zwalniającego nas od obowiązku znajomości ilości liczb w wierszu oraz liczby wierszy jest polecenie `fscanfMat('filename')`, linie niezawierające danych numerycznych są ignorowane. Istnieje odpowiednik tego polecenia pozwalający na zapis macierzy w formacie ASCII jest to polecenie `fprintfMat()`. Obie funkcje służą do wymiany informacji pomiędzy programami. Należy podkreślić że `fscanfMat` ignoruje tylko pierwszą linie nie zawierająca dane numeryczne. Jeśli takie linie powtarzają się wielokrotnie, polecenie zakończy odczyt po drugim pojawieniu się linie z znakami nie numerycznym. Przykładowo jeśli zmodyfikujemy plik znakowy `test.dat` wprowadzając kilka linii z tekstem

```
to jest pierwsza linia z tekstem
0.2113249 0.5608486 0.3076091 0.5015342 0.2806498
0.7560439 0.6623569 0.9329616 0.4368588 0.1280058
0.0002211 0.7263507 0.2146008 0.2693125 0.7783129
0.3303271 0.1985144 0.3126420 0.6325745 0.2119030
```

```
 0.6653811 0.5442573 0.3616361 0.4051954 0.1121355
 kolejna linia z tekstem...
 0.6283918 0.2320748 0.2922267 0.9184708 0.6856896
 0.8497452 0.2312237 0.5664249 0.0437334 0.1531217
 0.6857310 0.2164633 0.4826472 0.4818509 0.6970851
 0.8782165 0.8833888 0.3321719 0.2639556 0.8415518
 0.0683740 0.6525135 0.5935095 0.4148104 0.4062025
```

to ich odczyt w Scilabie wygląda następująco

```
 data2 = fscanfMat ('test2.dat') // polecenie automatycznie
                                 // otwiera plik
 data2  =
! 0.2113249 0.5608486 0.3076091 0.5015342 0.2806498 !

! 0.7560439 0.6623569 0.9329616 0.4368588 0.1280058 !

! 0.0002211 0.7263507 0.2146008 0.2693125 0.7783129 !

! 0.3303271 0.1985144 0.312642  0.6325745 0.211903  !

! 0.6653811 0.5442573 0.3616361 0.4051954 0.1121355 !
```

Widać że zignorowano linie 6 i wszystkie pozostałe. Polecenie `fprintfMat()` pozwala również na formatowany zapis macierzy. Wygenerujmy macierz `data`.

```
 data = rand (20,5,'normal')
 data  =
! - 0.7616491  - 0.7004486  0.0116391    0.5163254     0.2685505 !
!   1.4739762  - 0.8262233  - 1.4344474    1.0422456   0.8780630 !
!   0.8529775    1.1323911    1.748736     2.4976094 - 1.0179838 !
!   0.7223316  - 0.2330131    0.1645912    0.6450695 - 1.9598192 !
!   0.6380837  - 0.2343923    0.9182207  - 0.4483985   0.0927515 !
!   0.2546697    1.4027611    0.0259119  - 0.7218988   1.2311603 !
! - 0.6834217    0.3268713    0.7953703  - 0.5485232   0.6697461 !
!   0.8145126    0.0613533  - 1.3770621    1.1316247   0.2823040 !
! - 0.1884803  - 0.1890526  - 0.9063738  - 1.3667445 - 2.0049863 !
! - 1.0327357    0.4249849    1.2296215  - 1.3850463   1.1758127 !
! - 0.9239258  - 0.7460990  - 0.4577385  - 1.6280467 - 0.2570480 !
!   2.7266682  - 1.721103   - 0.5875092    0.7757721 - 0.1595813 !
! - 1.7086774  - 1.7157583    0.2301981  - 1.1049895   0.7131577 !
!   0.0698768    0.1023021  - 0.2563031    0.7604477 - 1.2140245 !
! - 1.3772844  - 1.2858605    1.1937458  - 0.6588100 - 2.343241  !
! - 0.1728369    0.6107784    1.8655072    0.1068464 - 2.0499752 !
! - 0.6019869  - 0.3778182  - 1.3189198    0.7469882 - 0.4183469 !
! - 1.5619521    2.5749104  - 0.8575199    0.0979236 - 0.2768419 !
! - 0.3888655  - 0.4575284    0.2973099    0.2946056 - 1.2710279 !
!   0.6543045  - 0.6453261  - 1.5404673    1.1474321 - 0.9639153 !
```

i zapiszmy ją na dysku w pliku test3.dat.

```
fprintfMat('test3.dat',data,'%15.10f')
```

gdzie `%15.10f` oznacza format zapisu liczb z macierzy `dane`, 15 oznacza że liczba ma piętnaście cyfr w tym dziesięć po przecinku. W pliku ujrzymy coś podobnego jak niżej

```
test3.dat
  -0.7616490874   -0.7004486348    0.0116391089    0.5163254498    0.2685505359
   1.4739762439   -0.8262233278   -1.4344473643    1.0422456176    0.8780630158
   0.8529775253    1.1323910660    1.7487360025    2.4976094313   -1.0179837547
   0.7223315777   -0.2330131163    0.1645911738    0.6450695409   -1.9598191507
   0.6380837232   -0.2343923439    0.9182207350   -0.4483984815    0.0927514850
   0.2546696793    1.4027611492    0.0259118507   -0.7218988418    1.2311603033
  -0.6834217347    0.3268713158    0.7953703383   -0.5485232377    0.6697460667
   0.8145126073    0.0613532795   -1.3770621196    1.1316247373    0.2823039722
  -0.1884802915   -0.1890525732   -0.9063738301   -1.3667444990   -2.0049863130
  -1.0327357336    0.4249848564    1.2296215292   -1.3850462843    1.1758126732
  -0.9239258062   -0.7460990484   -0.4577385334   -1.6280466584   -0.2570479823
   2.7266682098   -1.7211029770   -0.5875091875    0.7757720724   -0.1595812688
  -1.7086773593   -1.7157583243    0.2301980611   -1.1049895223    0.7131576895
   0.0698768244    0.1023020996   -0.2563031358    0.7604476679   -1.2140244569
  -1.3772844122   -1.2858605195    1.1937458319   -0.6588100458   -2.3432410362
  -0.1728369461    0.6107784158    1.8655071775    0.1068464105   -2.0499751583
  -0.6019868859   -0.3778182298   -1.3189197768    0.7469881736   -0.4183469030
  -1.5619521175    2.5749103741   -0.8575198721    0.0979236367   -0.2768419233
  -0.3888655235   -0.4575283716    0.2973098569    0.2946055949   -1.2710279057
   0.6543044610   -0.6453261045   -1.5404673140    1.1474320785   -0.9639152877
```

7.3. Pliki binarne

Wykorzystywanie plików binarnych jest częstą metodą wykorzystywaną do przechowywania informacji. Rozważmy prosty przykład: tablica zawierająca dane pochodzące z karty analogowo-cyfrowej. Sygnał jest zapamiętywany na dysku bez jakiejkolwiek kompresji. Zmienne są zapamiętywane w sposób znany z innych języków programowania. Poleceniem które może służyć do odczytu takich plików jest `mget()`. Jego składnia jest następująca:

```
x=mget([n,type,fd])
x=mgeti([n,type,fd])
```

gdzie: n – stała dodatnia: liczba danych do odczytu
fd – skalar. Parametr zwracany przez polecenie `mopen()` -1 dla ostatnio otwartego zbioru. domyślna wartość -1.
type – łańcuch , określa format czytanych liczb. 'l','i','s','ul','ui','us','d','f','c",'uc' - long, int, short, unsigned long, unsigned int, unsigned short, double, float, char i unsigned char
x – wektor liczb zmienno lub stałoprzecinkowych

Dwa kolejne przykłady objaśniają sposób pracy z plikami binarnymi Zwróćmy uwagę że zmienna typu char zajmuje tylko jeden bajt.

```
fid = mopen ('filechar1.test','w');       // tworzenie pliku
x =
 83;

mput ( x, 'c', fid)                       // zapis x


mclose(fid)                               // zamknięcie zbioru
 ans  =

    0.

fid = mopen ('filechar1.test','r');       // otwarcie pliku do
                                          // czytania

y = mget( x, 'c', fid)                    // czytanie x
 y  =

    83.

mclose(fid)                               // closes file
 ans  =

    0.
```

Drugi przykład w postaci skryptu do wykonania. Czytelnik zechce ustalić co właściwie robi program.

```
file1 = 'test1.bin';
file2 = 'test2.bin';
fd1=mopen(file1,'wb');
fd2=mopen(file2,'wb');
mput(1996,'ull',fd1);
mput(1996,'ull',fd2);
mclose(fd1);
mclose(fd2);
```

```
fd1=mopen(file1,'rb');
if 1996<>mget(1,'ull',fd1);write(%io(2),'Bug');end;
fd2=mopen(file2,'rb');
if 1996<>mget(1,'ull',fd2);write(%io(2),'Bug');end;
mclose(fd1);
mclose(fd2);
```

7.3.1. Przykład I

Przyjrzyjmy się strukturze zbioru plikowego. Na początek utwórzmy tablicę z liczbami całkowitymi z zakresu od -10 do 10. Dla zapisu wykorzystamy polecenie `mput()` a zmienne zapisujemy w formacie char.

```
fid = mopen ('filechar2.test','w');    // tworzenie zbioru

x = [ -10 , -5, 0, 5, 10];             // definowanie wektora

mput(x,'c',fid);                       // zapis wartości

mclose(fid)                            // zamknięcie zbioru
 ans  =

    0.
```

Dostęp do zapisanych wcześniej liczb uzyskujemy pisząc.

```
fid = mopen ('filechar2.test','r');    // otwarcie do czytania

y =mget(1,'c',fid)               // czytanie pierwszej wartości
 y  =

  - 10.

mtell(fid)               // wskaźnik pozycji czytania w zbiorze
ans =

    1.

meof(fid)                              // testowanie końca zbioru
 ans  =

    0.
```

Polecenie `mtell()` wskazuje aktualnie dostępną pozycje w pliku, która była odczytana, polecenie `meof()` sprawdza czy osiągnieto koniec zbioru.

```
y =mget(1,'c',fid)                // czytanie drugiej wartości
 y  =

  - 5.

mtell(fid)               // wskaźnik pozycji czytania w zbiorze
 ans  =

    2.

meof(fid)                          // testowanie końca zbioru
 ans  =

    0.
```

Dotychczas czytaliśmy po jednym elemencie, teraz spróbujmy odczytać jednocześnie trzy wartości

```
y =mget(3,'c',fid)    // czytanie trzech wartości jednocześnie
 y  =

!   0.    5.    10. !

mtell(fid)              // wskaźnik pozycji czytania w zbiorze
 ans  =

    6.

meof(fid)                        // testowanie końca zbioru
 ans  =

    0.
```

Mimo że zakończyliśmy odczyt wartości zapisanych do zbioru kontynuujemy odczyt.

```
y =mget(1,'c',fid)          // próba czytania kolejnej wartości
 y  =

     []

mtell(fid)             // wskaźnik pozycji czytania w zbiorze
 ans  =

    6.

meof(fid)                // testowanie końca zbioru
 ans  =
```

```
    16.
```

Jak należało oczekiwać odczyt nie powiódł się, osiągnęliśmy bowiem koniec zbioru. Przetestujmy teraz polecenie `mseek()` służące do ustawiania wskaźnika czytania w zbiorze.

```
mseek(0)                          // ustawiamy na pierwsza pozycję

mtell(fid)                  // wskaźnik pozycji czytania w zbiorze
 ans  =

    0.

y =mget(1,'c',fid)                               // czytamy wartość
 y  =

  - 10.

mseek(0)                          // ustawiamy na pierwsza pozycję /

mtell(fid)                    // wskaźnik pozycji czytania w zbiorze
 ans  =

    0.
```

Odczytajmy wartość z pliku raz traktując ją jako bajt ze znakiem raz jako bajt bez znaku.

```
y =mget(1,'uc',fid)          // czytamy wartość jako bajt bez znaku
 y  =

    246.

mclose(fid)                  // zamykamy zbiór
 ans  =

    0.
```

Należy zwrócić uwagę że w naszym przykładzie każda wartość jest zapamiętywana jako bajt ale też ten sam bajt może być interpretowany różnie zależnie od składni polecenia `mget()`.

7.3.2. Przykład II

W kolejnym przykładzie pokażemy jak interpretowane są liczby zapisane jako dwu bajtowe integery.

```
fid = mopen ('filechar3.test','w');   // otwieramy plik
```

```
x = [ -10 , -5, 0, 5, 10];

mput(x,'s',fid); // wysyłamy wektor x jako 2 bajtowe integery
                 // short)
mclose(fid)                    // zamknięcie zbioru ans  =

    0.
```

W formacie oznaczonym literą ('s') liczba całkowita jest zapisywana na dwy bajtach.

```
fid = mopen ('filechar3.test','r');    // otwieramy plik

mtell(fid)               // wskaźnik pozycji czytania w zbiorze
 ans  =

    0.

y =mget(1,'s',fid)        // czytamy pierwsza liczbę (short)
 y  =

  - 10.

mtell(fid)               // wskaźnik pozycji czytania w zbiorze
 ans  =

    2.
```

Pozycja wskaźnika czytania jest 2 w odróżnieniu od poprzedniej która była 1, jest to wynikiem że czytaliśmy po dwa bajty.

```
y =mget(1,'s',fid)           // czytamy druga wartość
 y  =

  - 5.

mtell(fid)              // wskaźnik pozycji czytania w zbiorze
 ans  =

    4.
```

Ustawiamy wskaźnik na początek zbioru.

```
mseek(0);                 // ustawiamy wskaźnik na początek

y =mget(1,'s',fid)        // czytamy pierwszą wartość
 y  =

  - 10.
```

```
mseek(0);                    // ustawiamy wskaźnik na pozycję 0

y =mget(1,'us',fid)          // czytamy liczbę bez znaku
 y  =

    65526.

mclose(fid)                     // zamykamy plik
 ans  =

    0.
```

Jak łatwo zauważyć czytając wartość '-10' jako liczbę całkowita bez znaku otrzymaliśmy wartość niepoprawną. Oznacza to że w tym konkretnym przypadku każda wartość jest zapisana na dwu bajtach. Dotyczy to zarówno liczb bez znaku jak i liczb ze znakiem.

7.3.3. Przykład III

```
fid = mopen ('filetypes.test','w'); // tworzenie pliku

i = 5                               // wartość do zapisu
 i  =

    5.

mput (i,'c',fid);                         // zapis jako znak
character

mput (i,'s',fid);                    // zapis jako short integer

mput (i,'l',fid);                    // zapis jako  long integer

mput (i,'f',fid);                   // zapis jako  float

mput (i,'d',fid);                  // zapis jako double

mclose(fid)                         // zamknięcie pliku
 ans  =

    0.
```

W powyższym przykładzie zapisaliśmy różne rodzaje danych, teraz sprawdźmy jak należy je odczytywać. Czytamy 'i1' jako jedno bajtową daną ('c')

```
fid = mopen ('filetypes.test','r');  // otwarcie zbioru do odczytu
```

```
mtell(fid)                  // wskaźnik pozycji czytania w zbiorze
 ans  =

    0.

i1 = mget(1,'c',fid)        // czytamy wartość jako znak (char)

 i1  =

    5.

mtell(fid)                 // wskaźnik czytania pokazuje  1 byte
 ans  =

    1.
```

Czytamy 'i2' ja liczbę zapisaną na dwu bajtach, wskaźnik czytania będzie na 3.

```
i2 = mget(1,'s',fid)           // czytamy dane jako  short
 i2  =

    5.

mtell(fid)                // wskaźnik  czytania
 ans  =

    3.
```

Czytamy 'i3' jako liczbę zapisaną na czterech bajtach ('l') wskaźnik czytania będzie na 7.

```
i3 = mget(1,'l',fid)                    // czytamy dane jako long
 i3  =

    5.
mtell(fid)                              // wskaźnik czytania
 ans  =

    7.
```

Zmienna rzeczywista ('f') jest zapisywana na czterech bajtach, wskaźnik pozycji czytania będzie na 11.

```
f1 = mget(1,'f',fid)              // czytamy zmienną rzeczywistą
 f1  =

    5.

mtell(fid)                       // wskaźnik czytania
```

```
 ans  =

    11.
```

Zmienna typu double (‘d’) jest zapisywana na 8 bajtach, wskaźnik czytania osiąga więc 19.

```
f1 = mget(1,'d',fid)        // czytamy zmienną typu double
 f1  =

    5.

mtell(fid)                    // wskaźnik czytania

 ans  =

    19.

mclose(fid)                    // zamknięcie pliku
 ans  =

    0.
```

Na koniec większy przykład skryptu pokazujący zapis odczyt pliku binarnego

```
//Przyklad uzycia polecen mopen, mclose,
//        mfprintf, mfscanf,meof
//Otwarcie zbioru plikowego do zapisu
//
mprintf('\nTworzenie pliku i drukowanie do niego\n');
// wydruk na ekranie
nazwa='c:\test\test.dta';
[plik,err]=mopen(nazwa,'w');
//
// Zapis danych do plikow
for k=1:5
    x=k^2; y=2*k-2; z=sin(5*k);
    mfprintf(plik,'%6f %6.3f %6.3f %6.3f \n',k,x,y,z);
end
mfprintf(plik,'\n');
mclose(plik);
//
mprintf('\nOtwarcie zbioru do czytania');
[plik,err]=mopen(nazwa,'r');
// Inicjacja pewnych zmiennych
licznik=0; k=[]; x=[]; y=[]; z=[]; m=[];
// Petla odczytujaca dane z pliku
while ~meof(plik)
```

```
      licznik=licznik+1;
      r=mfscanf(plik,'%f %f %f %f');
      m=[m;r];
      k=[k r(1)]; x=[x r(2)]; y=[y r(3)]; z=[z r(4)];
end
//
licznik=licznik-1;
mclose(plik);
//
// Wydruk na ekranie
mprintf('\n Czytanie danych z pliku\n');
mprintf('liczba linii w pliku %4d',licznik);
[nr,nc]=size(m);
// Okreslanie formatu wydruku na ekranie
sformat='';
// Konstruowanie formatu linii
for j=1:nc
    sformat=sformat+'%6.3f';
end;
// Druk poszczegolnych lini macierzy
for i=1:nr
    printf('\n');
    printf(sformat,m(i,:));
 end;
 //
 // Drukowanie poj wierszy macierzy
 mprintf('\n wiersz , k:\n');
 mprintf(sformat,k);
 mprintf('\n wiersz , x:\n');
 mprintf(sformat,x);
 mprintf('\n wiersz, y:\n');
 mprintf(sformat,y);
 mprintf('\n wiersz , z:\n');
 mprintf(sformat,z);
```

Rozdział 8

Ogólne zastosowania

Scilab został pomyślany jako środowisko do realizacji obliczeń naukowych i inżynierskich. Jak wiadomo podstawową strukturą używaną w Scilabie jest macierz, stąd też znajomość chociażby w minimalnym stopniu metod algebry liniowej wydaje się być konieczną.

8.1. Algebra liniowa

8.1.1. Obliczanie wyznaczników

Obliczenie wyznacznika macierzy dowolnego stopnia jest łatwe dzięki poleceniu `det()`, przy czym nie ma większego znaczenia czy argumentami macierzy są liczby (rzeczywiste lub zespolone), czy wielomiany dowolnego stopnia.

```
A=[1,3,2;1,1,1;0,2,3]
 A  =

!   1.    3.    2. !
!   1.    1.    1. !
!   0.    2.    3. !

det(A)
 ans  =

  - 4.
```

Dla macierzy, której elementami są wielomiany.

```
x=poly(0,'x');

A=[x,1+x;2-x,x^2]
 A  =

!   x         1 + x  !
!                    !
!                 2  !
!2 - x            x   !

det(A)
```

```
 ans

             2   3
  - 2 - x + x + x
```

8.1.2. Odwracanie macierzy

Odwracanie macierzy kwadratowej o elementach rzeczywistych, zespolnych czy wielomianowych jest możliwe dzięki poleceniu `inv()`. Wykonajmy przykład

```
A=rand(3,3)
 A  =

!   0.2113249    0.3303271    0.8497452 ! !   0.7560439
0.6653811    0.6857310 ! !   0.0002211    0.6283918    0.8782165 !
```

i obliczmy macierz odwrotną

```
 inv (A)
 ans  =

!   0.7079801    1.1252416  - 1.5636415 !
! - 3.0628747    0.8554450    2.2956276 !
!   2.1914059  - 0.6123814  - 0.5035294 !
```

Polecenie A^{-1} daje ten sam efekt. Dla macierzy o argumentach wielomianowych mamy

```
x=poly(0,'x');

A=([x,1+x;2-x,x^2]);
inv(A)
 ans  =

!          2                          !
!         x             - 1 - x       !
!   -------------     -------------   !
!             2   3             2   3 !
! - 2 - x + x + x   - 2 - x + x + x   !
!                                     !
!     - 2 + x               x         !
!   -------------     -------------   !
!             2   3             2   3 !
! - 2 - x + x + x   - 2 - x + x + x   !
```

8.1.3. Ranga, norma macierzy

Funkcją pozwalająca obliczyć rangę macierzy to `rank()`, natomiast normę macierzy obliczamy poleceniem norm().

```
A=[1,2,3;1,1,1;0,2,3]
 A  =

!   1.    2.    3. !
!   1.    1.    1. !
!   0.    2.    3. !

rank(A)
 ans  =

    3.

norm(A)
 ans  =

    5.3719323
```

8.1.4. Ślad macierzy

Polecenie `trace()` zastosowane do macierzy o dowolnych argumentach pozwala wyznaczyć jej ślad (suma elementów na głównej przekątnej), wystarczy napisać:

```
-->trace(A)
 ans  =

    5.
```

8.1.5. Wektory i wartości własne

Wartości własne i odpowiadające im wektory własne mają ogromne znaczenie w mechanice czy teorii sprężystości. Istnieje kilka poleceń umożliwiających ich wyznaczenie, najprościej osiągamy to wykorzystując polecenia `spec()` i `bdiag()`.Dla uprzednio zdefiniowanej macierzy A

```
spec(A)
 ans  =

!   0.3173850 + 0.3582594i !
!   0.3173850 - 0.3582594i !
!   4.36523                !

bdiag(A)
 ans  =
```

```
!   0.1434740    1.8371989    0.       !
! - 0.0863242    0.4912959    0.       !
!   0.           0.           4.36523 !
```

8.1.6. Triangularyzacja macierzy - tworzenie macierzy trójprzekątniowej

Przkształcenie danej macierzy do macierzy trójprzekątniowej umożliwia polecenie `htrianr()`.

```
-->htrianr(A)
 ans  =

!   1     8.882E-16     1           !
!                                   !
!   0     1             0.3333333   !
!                                   !
!   0     0             1           !
```

Poniżej pokazano bardziej złożony przykład trangularyzacji dotyczący macierzy, której elementami są wielomiany.

```
x=poly(0,'x') x  =
     x

M=[x;x^2;2+x^3]*[1,x-2,x^4]
 M  =

!                   2              5      !
!   x       - 2x + x              x       !
!                                         !
!    2          2   3              6      !
!   x       - 2x + x              x       !
!                                         !
!        3               3   4      4   7 !
!   2 + x   - 4 + 2x - 2x + x     2x + x  !


[Mu,U,rk]=htrianr(M)
 rk  =

    1.

 U  =

!              4       !
! - 2 + x     x     1  !
```

```
!                        !
! - 1          0      0  !
!                        !
!   0         -1      0  !

 Mu  =

!   0    0    x       !
!                      !
!                2     !
!   0    0    x       !
!                      !
!                    3 !
!   0    0    2 + x   !

det(U)
 ans  =
    1

M*U(:,1:2)
 ans  =

!   0    0 !
!           !
!   0    0 !
!           !
!   0    0 !
```

8.2. Rozwiązywanie układu równań liniowych

W wielu zagadnieniach statyki mamy doczynienia z koniecznością rozwiązywania układów równań liniowych dlatego umiejętność ich rozwiązywania jest ważna.

8.2.1. Metoda podstawowa

Aby rozwiązać układ równań liniowych n-tego stopnia, współczynniki przy zmiennych niezależnych powinny zapisane być w postaci macierzy kwadratowej. Scilab wykorzystuje algorytm LU z wyborem elementu podstawowego. Jeśli macierz współczynników oraz wyrazów wolnych wygenerujemy za pomocą polecenia `rand()` to rozwiązanie układu równań sprowadza się do poniższej sekwencji poleceń.

```
R=rand (5,5);  // macierz współczynników

y=rand(5,1);   // wektor wyrazów wolnych
```

```
x=R\y  // rozwiązanie Rx=y
  x  =

! - 0.9448615 !
!   1.8150216 !
!   1.0487447 !
!   0.7257294 !
! - 0.4538296 !
```

8.2.2. Rozszerzenia metody podstawowej

W przypadku, kiedy istnieje konieczność rozwiązania układu równań dla kilku różnych zestawów wyrazów wolnych, można skorzystać z ułatwień proponowanych przez system:

```
A=rand(4,4);
y1=[1;0;0;0];
y2=[1;2;3;4];
X=A\[y1,y2];
```

Pierwsza kolumna macierzy `X` jest rozwiązaniem odpowiadającym systemowu `Ax1=y1`, a druga systemowi `Ax2=y2`.

8.2.3. Rozwiązanie układu z wykorzystaniem poleceni `linsolve()`

Do rozwiązania układu równań liniowych można wykorzystać polecenie `linsolve()`. Polecenie to rozwiązuje układ: `R*x+y=0`. Sekwencja poleceń jest następująca.

```
R=rand (5,5);
y=rand(5,1);
linsolve(R,-y)
ans  =

! - 0.9448615 !
!   1.8150216 !
!   1.0487447 !
!   0.7257294 !
! - 0.4538296 !
```

8.2.4. Rozwiązaniu układu równań AX=B gdzie macierze A i B są macierzami symbolicznymi

Metoda ta wymaga wykonania trzech kroków i wykorzystania trzech poleceń
— `trianfml()`
— `trisolve()`
— `evstr()`
`trianfml()` umożliwia triangularyzacje macierzy symbolicznej A. `trisolve()` umożliwia rozwiązanie AX=B, kiedy A jest macierzą trójkątna górną. `evstr()` umożliwia zastąpienie symboli w macierzy A i B przez wartości liczbowe.

Przykład:

```
A=['1','2';'a','b']; // macierz współczynników
 A  =

!1  2  !
!      !
!a  b  !

B=['x','y']  // wektor wyrazów wolnych
 B  =

!x  y  !

W=trianfml(A)
 W  =

!a  b      !
!          !
!0  a*2-b  !

T=trisolve(W,B)
 T  =

!a\x  a\y  !

a=1;b=2;x=3;y=4; \\

evstr(T)  \\
 ans  =

!   3.    4. !
```

Umiejętność wykorzystywania tych poleceń daje nam szansę na testowanie rozwiązania dla wielu różnych układów wektorów wyrazów wolnych i macierzy współczynników, odzwierciedlających różne konfiguracje pracy analizowanych układów.

8.3. Interpolacja

8.3.1. Interpolacja liniowa

Aby wykonać interpolację liniową wykorzystujemy polecenie `interpln()`. Znaczenie poszczególnych parametrów formalnych tego polecenia jest następujące:

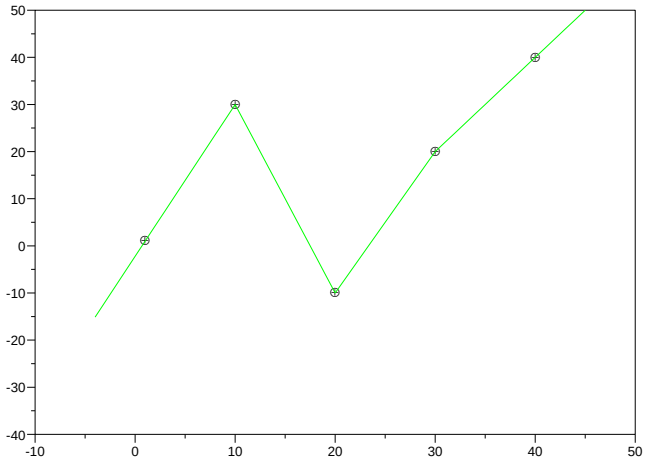
```
[y]=interpln(xyd,x)
```

gdzie: xyd – dwuwierszowa macierz zawierająca współrzędne punktów eksperymentalnych
x – wektor zmiennych niezależnych
y – wektor wartości funkcji otrzymany poprzez interpolację liniową

Przykład poniżej pokazuje sposób wykorzystania polecenia `interpln()`.

```
x=[1 10 20 30 40];  // wartości eksperymentalne x

y=[1 30 -10 20 40]; // wartości eksperymentalne y

plot2d(x',y',[-3],"011"," ",[-10,-40,50,50]);

//polecenie plot2d kreśli wartości eksperymentlne na rysunku

yi=interpln([x;y],-4:45);  //Obliczmy wartości intertpolowane w
                           //zakresie -4:45
plot2d((-4:45)',yi',[3],"000"); //kreślimy funkcję interpolującą w
```

Ostatecznie otrzymujemy rysunek 8.1.



Rysunek 8.1. Interpolacja liniowa

8.3.2. Interpolacja splajnami

Wykonania interpolacji wartości splajnami wymaga wykorzystania dwu poleceń `splin()` i `interp()`. Polecenie `splin()` ma następującą składnię:

```
d=splin(x,f)
```

gdzie: x – wektor liczb rzeczywistych
f – wektor liczb rzeczywistych tego samego stopnia co x

Należy zauważyć że wartości $f(x_i)$ obliczane są dla x_i. Polecenie `spline()` oblicza

funkcję trzeciego rzędu S interpolującą wartości f.

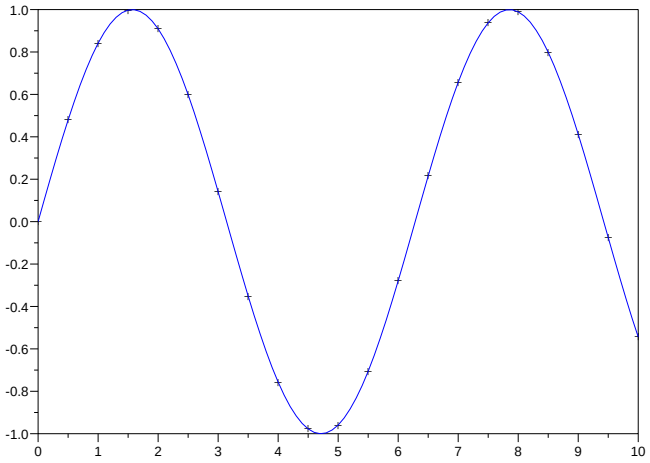
$$\begin{cases} f_i &= S(x_i) \\ d_i &= S'(x_i) \end{cases}$$

Funkcja `d` musi być wykorzystana wspólnie z poleceniem `interp()`. Polecenie `interp()` umożliwia wyznaczenie wartości funkcji S dla danych argumentów x_d. Sposób wykorzystania jest następujący:

```
[f0 [,f1 [,f2 [,f3]]]]=interp(xd,x,f,d)
```

gdzie: xd – wektor liczb rzeczywistych
x,f,d – wektor liczb rzeczywistych, współczynniki splajnu
fi – wektor pochodnych
Przykład użycia.

```
x=0:0.5:10;f=sin(x); \\punkty interpolacji
d=splin(x,f); \\definiowanie funkcji S
S=interp(0:0.1:10,x,f,d);  \\oblicznia funkcji interpolującej
plot2d(x',f',-1);  \\kreślenie punktów eksperymentalnych
plot2d((0:0.1:10)',S',2,'000')  \\kreślenie splajnu
```



Rysunek 8.2. Interpolacja splajnami

8.4. Całkowanie numeryczne

Obliczenie wartości całki oznaczonej jest stosunkowo proste i sprowadza się do wykorzystania polecenia `integrate()`.

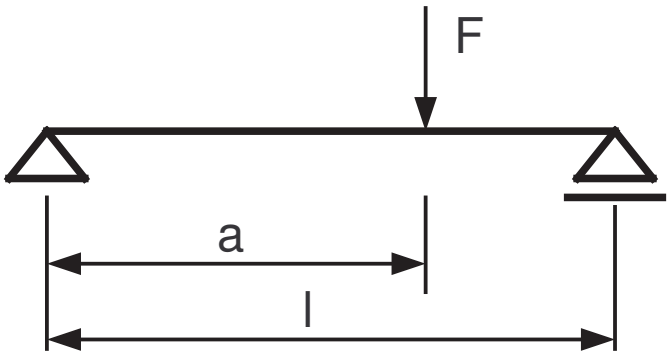
```
[x]=integrate(expr,v,x0,x1)
```

gdzie: x – wartość całki,
expr – funkcja pod całkowa,
v – zmienna niezależna,
x0,x1 – granice całkowania.
Polecenie `integrate()` oblicza więc:

$$x = \int_{x_0}^{x_1} expr(v)dv$$

8.5. Ćwiczenia

1. Dla układu jak na rysunku 8.3 wyznaczyć reakcje dla zmieniającej się w czasie siły $fF = \sin(t)$.



Rysunek 8.3. Belka zginana

2. Obliczyć całkę oznaczoną funkcji $e^{-x}ln(1/x)$ w zakresie od $x \in < 0; 2 >$

Spis treści

Bibliografia

Spis rysunków

Spis tablic