Wprowadzenie do Scilaba

wersja 0.9999 α

Bruno Pinçon

Institut Elie Cartan Nancy
E.S.I.A.L.
Université Henri Poincaré
Email : Bruno.Pincon@iecn.u-nancy.fr

Przekład z języka francuskiego :

Piotr Fulmański

Wydział Matematyki i Informatyki
Uniwerstytetu Łódzkiego
Email : fulmanp@math.uni.lodz.pl

Katarzyna Szulc

Institut Elie Cartan Nancy 1
Université Henri Poincaré
Email : Katarzyna.Szulc@iecn.u-nancy.fr

Skrypt ten był początkowo opracowany przez studentów E.S.I.A.L. (Ecole Supérieure d'Informatique et Application de Lorraine). Opisuje on niewielką część możliwości Scilaba, w szczególności te, które pozwalają zastosować notacje analizy numerycznej i małych symulacji stochastycznych takich jak:

- operacje na macierzach i wektorach o współrzędnych rzeczywistych;
- programowanie w Scilabie;
- prosta grafika;
- niektóre funkcje dla dwóch wymienionych powyżej (generowanie liczb losowych, rozwiązywanie równań, ...)

Scilab umożliwia wykonanie wielu innych operacji, w szczególności w dziedzinie automatyki, obróbki sygnałów dźwiękowych, symulacji systemów dynamicznych (przy pomocy scicos)... Jakoże zamierzam systematycznie uzupełniać ten dokument, jestem w pełni otwarty na wszelkie uwagi, sugestie i krytykę pozwalające mi na jego uleprzenie (również w przypadku błędów ortograficznych), piszcie do mnie[1]. Mała historia tego dokumentu:

- wersja 0.999: modyfikacje rozdziału o grafice oraz uzupełnienie treści rozdziału o programowaniu; wersja relatywna do Scilab-2.7 ;
- wersja 0.9999 (ten dokument): dostosowanie rozdziału o grafice do ńowej grafigi obiektowejŚcilaba; wersja relatywna do Scilab-4.0.

W wyniku dopisania kilku paragrafów, dokument stracił na spójności. Istnieje jednak obecnie inny podręcznik, ktory jest dostępny na stronie internetowej Scilaba (czytaj dalej).

[1] Tę samą uwage chca przekazać czytelnikom autorzy tłumaczenia polskiego (*przyp. tłum.*)

Podziękowania

- dla Doc Scilab, który często pomagał mi na forum użytkowiników;
- dla Bertrand Guiheneuf, który dostarczył mi magiczną "ścieszkę"do skompilowania Scilaba 2.3.1 pod linuxem (kompilacja pod linuxem nowszych wersji nie powoduje problemów);
- dla moich kolegów i przyjaciół, Stéphane Mottelet[2] , Antoine Grall, Christine Bernier-Katzentsev oraz Didier Schmitt ;
- dla Patrice Moreaux za uważne przeczytanie i poprawienie błędów;
- dla Helmut Jarausch, który przetłumaczył ten dokument na język niemiecki i który zwrócił moją uwagę na kilka niejasności;
- i dla wszystkich czytelników za ich wsparcie, uwagi i korekty.

[2] dzięki za ten .pdf Stéphane!

Spis treści

1 Wiadomosci wstepne

1.1 Co to jest Scilab?

Dla osób znających już MATLAB-a odpowiedź jest prosta – Scilab jest jego darmowym (więcej szczegułów związanych z tym tematem w dalszej części) odpowiednikiem, powstałym[3] w I.N.R.I.A. (Institut National de Recherche en Informatique et Automatique). Składnia, z wyjątkiem nielicznych poleceń, jest taka sama (istotne różnice wystepują w przypadku grafiki). Osobom nie znającym MATLAB-a chcę powiedzieć, że Scilab jest przystepnym środowiskiem do wykonywania obliczeń numerycznych dzięki zaimplementowanym niezbednym metodom:

- rozwiazywanie ukladów liniowych,
- wyznaczanie wartości własnych, wektorów własnych,
- dekompozycja dla wartości osobliwych i psełdo-osobliwych,
- szybka transformacja Fouriera,
- rozwiazywanie równań różniczkowych,
- algorytmy optymalizacji,
- rozwiazywanie równan nieliniowych,
- generowanie liczb losowych,
- dla wielu niezaawansowanych zastosowań algebry liniowej w automatyce.

Ponadto Scilab wyposażony jest w funkcje służące do tworzenia grafiki, zarówno niskopoziomowej (wielokąty, odczytywanie współrzędnych położenia kursora, itp.) jak i wysokopoziomowej (krzywe, powierzchnie itp.). Wprowadzony język programowania, dzięki operowaniu notacją macierzową, jest prostym, ale potężnym i efektywnym narzędziem. Wyszukiwanie błędów w programach jest szybkie dzięki łatwemu operowaniu zmiennymi. W przypadku, gdy obliczenia będą zbyt czasochłonne (język jest interpretowany...) możliwe jest łatwe połączenie programu Scilab-a z podprogramami napisanymi w C czy FORTRAN-ie.

1.2 Jak korzystać z tego dokumentu?

W rozdziale drugim wyjaśniono podstawy pracy z Scilab-em jako narzędziem do obliczeń macierzowych. Wystarczy prześledzić proponowane przykłady. Sekcje oznaczone gwiazdką podczas pierwszego czytania można pominąć. Osoby zainteresowane zagadnieniami dotyczącymi grafiki, moga przeanalizowac pierwsze przykłady z rozdziału czwartego. Rozdział trzeci wyjaśnia podstawy programowania w Scilab-ie. Rozdział szósty pokazuje jak uniknąć najczęstrzych błędów popełnianych podczas pracy w Scilab-ie lub nieporozumień wynikających z jego specyfikacji (proszę podeślij mi także i swoje!). Istnieją nieznaczne różnice pomiędzy środowiskami graficznymi przeznaczonymi dla Unix i Windows polegające na odmiennym sposobie rozmieszczenia przycisków i menu. W tym dokumencie opieram sie na wersji Unix aczkolwiek użytkownicy wersji przeznaczonej dla systemu Windows nie powinni natrafić na problemy ze znalezieniem odpowiednich opcji / przyciskow.

1.3 Podstawy pracy z Scilab-ie

W najprostszym przypadku, Scilab może być wykorzystywany jako „kalkulator" zdolny wykonywać obliczenia na wektorach i macierzach liczb rzeczywistych i/lub zespolonych (ale także na zwykłych skalarach) oraz do wizualizacji krzywych i powierzchni. W najprostrzych zadaniech raczej nie ma potrzeby pisania programów. Dosyć szybko zaczniemy jednak korzystać ze skryptów (zbiorów instrukcji, poleceń Scilab-a) a następnie funkcji. Oczywiście niezbędne w takich sytuacjach staje sie użycie edytora tekstu, na przykład emacs (Unix, Windows), wordpad (Windows), vi lub vim (Unix a także Windows)...

1.4 Gdzie znaleźć informacje na temat Scilab-a?

W dlalszej części dokumentu zakłda się, że czytelnik dysponuje wersją 2.7 programu. W celu uzyskania dodatkowych informacji odsyłam do „Scilab home page":

`http://scilabsoft.inria.fr`

na której znaleźć można różnorodną dokumentację, oraz efekty pracy innych użytkowników.

„Scilab group" wydał (w latach 1999 – 2001) około dwudziestu artykułów w czasopiśmie „Linux magazie". Polecam je szczególnie, gdyż poruszają większość zagadnień związanych ze Scilab-em (większości z nich w tym opracowaniu nawet się nie porusza). Uzyskać je można pod adresem

[3] Scilab wykożystuje dużą ilość funkcji pochodzących z różnych miejsc, dostępnych często przez Netlib

`http://www.saphir-control.fr/articles/`

Na temat Scilab-a prowadzone jest również forum dyskusyjne, w ramach którego istnieje możliwość zadawania pytań, dokonywania uwag, udzielania odpowiedzi na wcześniej postawione pytania, itd:

`comp.sys.math.scilab`

Wszystkie zamieszczone tam wiadomości są archiwizowane i dostępne ze strony domowej Scilab-a po wybraniu *Scilab newsgroup archive*.

Wybierając jeden z dwóch odsyłaczy *Books and Articles on Scilab* lub *Scilab Related Links* umieszczonych na stronie głównej uzyskujemy dostęp do wielu różnch dokumentów. W szczególności:

- wprowadzenie B. Ycart (Démarrer en Scilab et statistiques en Scilab);
- „Scilab Bag Of Tricks" autorstwa Lydia E. van Dijk i Christoph L. Spiel, który jest raczej przeznaczony dla osób dobrze znających już Scilab-a (rozwój tej książki został niestety brutalnie przerwany kilka lat temu);
- *Travaux Pratiques sur Scilab classes par themes* umożliwia dostęp do projektów realizowanych przez studentów ENPC;
- wprowadzenie do informatyki z użciem Scilab-a (verb+http://kiwi.emse.fr/SCILAB/+).

Oczywiście w zależności od potrzeb znaleźć można dużo innych opracowań traktujących problem w nieco odmienny niż zamieszczony tutaj sposób.

1.5 Jaki jest statut programu Scilab?

Osoby dobrze znające oprogramowanie (na licencji GPL) z pewnością interesuje statut Scilab-a jako programu *darmowego*. Częstym pytaniem na forum jest :

Scilab: is it really free?

Yes it is. Scilab is not distributed under GPL or other standard free software copyrights (because of historical reasons), but Scilab is an Open Source Software and is free for academic and industrial use, without any restrictions. There are of course the usual restrictions concerning its redistribution; the only specific requirement is that we ask Scilab users to send us a notice (email is enough). For more details see Notice.ps or Notice.tex in the Scilab package.

Answers to two frequently asked questions: Yes, Scilab can be included a commercial package (provided proper copyright notice is included). Yes, Scilab can be placed on commercial CD's (such as various Linux distributions).

2 Operacje na macierzach i wektorach

Ta część daje podstawy do poznania zastosowań Scilab-a jako narzędzia do operacji macierzowych.

Aby rozpocząć pracę z Scilab-em wystarczy wpisać

```
scilab
```

w terminalu[4]. Jeśli wszystko przebiega prawidłowo, na ekranie ukaże się okno Scilab-a z głównym menu zawierającym w szczególności przyscisk `Help`, `Demos` a w oknie wprowadzania poleceń ukaże się

```
                 ==========
                 scilab-2.7
      Copyright (C) 1989-2003 INRIA/ENPC
                 ==========



Startup execution:
  loading initial environment

-->
```

gdzie `-->` jest znakiem zachęty.

[4] Albo kliknąć odpowiednią ikonę.

2.1 Wprowadzanie macierzy

Podstawowym typem danych w Scilab-ie jest macierz liczb rzeczywistych lub zespolonych. Najprostszym sposobem definiowania macierzy (wektora, skalara będących w istocie szczególnymi przypadkami macierzy) w środowisku Scilab jest wprowadzenie z klawiatury listy elementów macierzy, stosując następującą konwencję:

- elementy tego samego wiersza oddzielone są spacją lub przecinkiem;
- lista elementów musi być ujęta w nawias kwadratowy [];
- w przypadku wprowadzania skalarów nawias kwadratowy można pominąć;
- każdy wiersz macierzy, z wyjątkiem ostatniego, musi być zakończony średnikiem.

Dla przykładu komenda:

```
-->A=[1 1 1;2 4 8;3 9 27]
```

da na wyjściu

```
 A  =
!   1.    1.    1.  !
!   2.    4.    8.  !
!   3.    9.    27. !
```

W przypadku, gdy instrukcja zostanie zakończona średnikiem, wynik nie pojawi się na ekranie. Wpiszmy na przykład

```
-->b=[2 10 44 190];
```

Aby zobaczyć współrzędne wprowadzonego wektora, wystarczy napisać

```
-->b
```

a odpowiedzią będzie

```
 b =
!  2.  10.  44.  190. !
```

Bardzo długa instrukcja może być napisana w kilku linijkach, przy czym przechodząc do linii następnej, linię poprzednią należy zakończyć trzema kropkami, jak w poniższym przykładzie:

```
-->T = [ 1 0 0 0 0 0 ;...
-->      1 2 0 0 0 0 ;...
-->      1 2 3 0 0 0 ;...
-->      1 2 3 0 0 0 ;...
-->      1 2 3 4 0 0 ;...
-->      1 2 3 4 5 0 ;...
-->      1 2 3 4 5 6 ]
```

co daje

```
 T  =
!   1.    0.    0.    0.    0.    0. !
!   1.    2.    0.    0.    0.    0. !
!   1.    2.    3.    0.    0.    0. !
!   1.    2.    3.    4.    0.    0. !
!   1.    2.    3.    4.    5.    0. !
!   1.    2.    3.    4.    5.    6. !
```

W przypadku wprowadzania liczb zespolonych stosuje się następującą składnie:

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

-->Y = [ 1 + %i , -2 + 3*%i ; -1 , %i]
 Y  =
!   1. + i    - 2. + 3.i !
! - 1.          i        !
```

2.2 Typowe wektory i macierze

Istnieją funkcję do konstrukcji typowych macierzy i wektorów.

Macierz jednostkowa. Aby otrzymać macierz jednostkową o wymiarach 4 na 4 stosujemy instrukcję:

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

Argumentami funkcji `eye(n,m)` jest liczba wierszy (`n`) oraz kolumn (`m`) (Uwaga : jeżeli $n < m$ (odpowiednio $n > m$ wówczas otrzymamy macierz odwzorowania wzajemnie jednoznacznego, surjekcja (odpowiednio injekcja) przestrzeni kanonicznej K^m na K^n).

Macierz diagonalna, diagonalna macierzy. Aby otrzymać macierz diagonalną, w której elementy na głównej przekątnej pochodzą z wcześniej zdefiniowanego wektora `b` wpisujemy

```
-->B=diag(b)
 B  =
!   2.    0.     0.     0.   !
!   0.    10.    0.     0.   !
!   0.    0.     44.    0.   !
!   0.    0.     0.     190. !
```

Uwaga: Pisząc `b` nadal mamy dostęp do wcześniej zdefiniowanego wektora, co pokazuje, że Scilab rozróżnia wielkość liter.

Zastosowanie funkcji `diag` na dowolnej macierzy pozwala uzyskać główną przekątną macierzy jako wektor kolumnowy

```
-->b=diag(B)
 b  =
!   2.   !
!   10.  !
!   44.  !
!   190. !
```

(Funkcja ta przyjmuje także drugi, opcjonalny, argument – porównaj ćwiczenia.)

Macierze zerowa i jedynkowa. Funkcje `zeros` i `ones` pozwalają odpowiednio stworzyć macierze zerowe i macierze składające się z jedynek. Podobnie jak dla funkcji `eye` ich argumentami są liczba wierszy i liczba kolumn.

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

Można także użyć jako argument nazwę macierzy już zdefiniowanej w środowisku. W efekcie otrzymujemy macierz o takich wymiarach macierzy będącej argumentem

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

Macierz trójkątna. Funkcje `triu` i `tril` pozwalają otrzymać macierz trójkątną górną i dolną:

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

Macierze o elementach losowych. Funkcja `rand` pozwala utworzyć macierz o elementach peudolosowych (pochodzących z przedziału [0,1); możliwe jest także użycie rozkładu normalnego jak i podanie zarodka dla generatora liczb pseudolosowych):

```
-->M = rand(2, 5)
 M  =
! 0.2113249  0.0002211  0.6653811  0.8497452  0.8782165 !
! 0.7560439  0.3303271  0.6283918  0.6857310  0.0683740 !
```

n-elementowy wektor o stałej różnicy między elementami. Aby wprowadzić wektor (wierszowy) `x` o n współrzędnych równomiernie rozmieszczonych pomiędzy x_1 i x_n (innymi słowy tak aby $x_{i+1} - x_i = \frac{x_n - x_1}{n-1}$, n węzłów, zatem $n-1$), używamy funkcji `linspace`.

```
-->x = linspace(0,1,11)
 x  =
! 0.  0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1. !
```

Instrukcją analogiczną pozwalającą utworzyć wektor o zadanej wartości pierwszej współrzędnej, ustalonej różnicy pomiędzy współrzędnymi i ostatniej współrzędnej nie większej niż zadana wartość jest

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

Składnia jest następująca `y = wartPocz:przyrost:wartGranicz`. Tak długo jak pracujemy z liczbami całkowitymi nie ma problemu z ustaleniem wartości granicznej odpowiadającej ostatniej składowej wektora.

```
-->i = 0:2:12
 i  =
!  0.  2.  4.  6.  8.  10.  12. !
```

Dla liczb rzeczywistych jest to zdecydowanie trudniejsze do określenia:

(i) przyrost może posiadć nieskończone rozwinięcie w reprezentacji binarnej lub jego skończone rozwinięcie może wybiegać poza zakres reprezentacji maszynowej liczb rzeczywistych powodując ich zaokrąglenia;

(ii) błędy zaokrągleń numerycznych kumulują się w miarę obliczania kolejnych składowych wektora.

```
-->xx = 0:0.05:0.60
 ans  =
! 0.  0.05  0.1  0.15  0.2  0.25  0.3  0.35  0.4  0.45  0.5  0.55 !
```

Uwaga: w zależności od komputera na jakim uruchomiony zostanie Scilab powyższy przykład może dać różnie wyniki, to znaczy 0.6 *może pojawić się jako ostatnia składowa.* Często przyrost równy jest 1, można go w takiej sytuacji pominąć:

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

W przypadku gdy przyrost jest dodatni (ujemny) oraz `wartPocz>wartGrancz` (`wartPocz<wartGrancz`) otrzymujemy wektor bez współrzędnych nazywany w Scilab-ie macierzą pustą:

```
-->i=3:-1:4
 i  =
    []

-->i=1:0
 i  =
    []
```

2.3 Wyrażenia w Scilab-ie

Scilab jest językiem posiadającym bardzo prostą składnię, w której instrukcje przypisania mają postać

```
zmienna = wyrazenie
```

lub prościej

```
wyrazenie
```

gdzie w ostatnim przypadku wartość `wyrazenia` jest przypisana do domyślnej zmiennej o nazwie `ans`. Wyrażenia w Scilabie mogą być tak proste (jeśli chodzi o zapis) jak wyrażenia skalarne w innych językach programowania, ale mogą składać się z macierzy i wektorów co często sprawia trudność początkujacym użytkownikom tego języka. Dla wyrażeń „skalarnych" mamy standardowe operatory `+`, `-`, `*` , `/` i `^` i najczęściej stosowane funkcje przedstawione w tabeli 1. Zauważmy, że funkcje wymienione w tabeli poniżej funkcji Γ są dostępne od wersji 2.4 oraz, że Scilab proponuje także inne funkcje specjalne, wsród których można znaleźć funkcje Bessela, eliptyczne, itd.

`abs`	wartość bezwzględna, moduł
`exp`	eksponent
`log`	logarytm naturalny
`log10`	logarytm o podstawie 10
`cos`	cosinus (argument w radianach)
`sin`	sinus (argument w radianach)
`sinc`	$\frac{sin(x)}{x}$
`tan`	tangente (argument w radianach)
`cotg`	cotangente (argument w radianach)
`acos`	arccos
`asin`	arcsin
`atan`	arctg
`cosh`	cosinus hiperboliczny
`sinh`	sinus hiperboliczny
`tanh`	tangens hiperboliczny
`acosh`	argch
`asinh`	argsh
`atanh`	argth
`sqrt`	pierwiastek kwadratowy
`floor`	$E(x) = (\lfloor x \rfloor) = n \Leftrightarrow n \leq x < n+1, \qquad x \in N$
`ceil`	$\lceil x \rceil = n \Leftrightarrow n-1 < x \leq n, \qquad x \in N$
`int`	$int(x) = \lfloor x \rfloor$ jeśli $x > 0$ oraz $= \lceil x \rceil$ dla $x \leq 0$
`erf`	funkcja błędu $erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$
`erfc`	dopełnienie funkcji błędu określone przez $ercf(x) = 1 - erf(x) = \frac{2}{\sqrt{\pi}} \int_x^{+\infty} e^{-t^2} dt$
`gamma`	$\Gamma(x) = \int_0^{+\infty} t^{x-1} e^{-t} dt$
`lngamma`	$\ln(\Gamma(x))$
`dlgamma`	$\frac{d}{dx} \ln(\Gamma(x))$

`:fonc`

Tablica 1: Wybrane funkcj używane przez Scilab-a.

Definiując zmienną (będącą skalarem, wektorem, macierzą) jej wartość (wartości) nie musi być wyrażona przez konkretną liczbę, ale także przez wyrażenie którego wartość zostanie jej przypisana.

```
-->M = [sin(%pi/3) sqrt(2) 5^(3/2); exp(-1) cosh(3.7) (1-sqrt(-3))/2]
 M  =
!   0.8660254    1.4142136    11.18034          !
!   0.3678794    20.236014    0.5 - 0.8660254i  !
```

Uwaga: Powyższy przykład ilustruje potencjalne niebezpieczeństwo podczas obliczania pierwiastka kwadratowego z liczy ujemnej. Scilab rozważa czy ma do czynienia z liczbami zespolonymi i zwraca jeden z pierwiastów jako rezultat.

2.3.1 Kilka podstawowych przykładów wyrażeń macierzowych

Dostępne są wszystkie proste działania wykonywane na macierzach: suma dwóch macierzy, iloczyn macierzy, iloczyn skalarny i macierzowy itd. Oto kilka przykładów (w których wykorzystujemy wcześniej zdefiniowane macierze). *Uwaga: Tekst występujacy w danym wierszu po znaku `//` oznacza dla Scilab-a komentarz. Nie jest interpretowany a jedynie dostarcza pewnych uwag i wyjaśnień.*

```
-->D = A + ones(A)  // napisz A aby zobaczyc jej zawartosc
 D  =
!   2.    2.    2.  !
!   3.    5.    9.  !
```

```
!   4.    10.    28. !

-->A + M      // nie mozna wykonac dzialania dodawania (niezgodnosc wymiarow),
              // jaka jest odpowiedz Scilab-a?
      !--error     8
inconsistent addition

-->E = A*C   // C jest macierza jedynkowa o wymiarach (3,4)
 E  =
!   3.     3.     3.     3.  !
!   14.    14.    14.    14. !
!   39.    39.    39.    39. !

--> C*A   // nie mozna wykonac mnozenia (niezgodnosc wymiarow),
          // jaka jest odpowiedz Scilab-a?
     !--error    10
inconsistent multiplication

--> At = A'  // transpozycje otrzymuje sie stawiajac za nazwa macierzy znak apostrofu
 At  =
!   1.    2.    3.  !
!   1.    4.    9.  !
!   1.    8.    27. !

--> Ac = A + %i*eye(3,3)  // tworzymy macierz o elementach zespolonych
 Ac  =
!   1. + i      1.          1.        !
!   2.          4. + i      8.        !
!   3.          9.          27. + i   !

--> Ac_adj = Ac' // w ten sposob otrzymujemy macierz transponowana o elementach
                 // zespolonych zastapionych ich sprzezeniami
 Ac_adj  =
!   1. - i      2.          3.        !
!   1.          4. - i      9.        !
!   1.          8.          27. - i   !

-->x = linspace(0,1,5)'   // konstrukcja wektora kolumnowego
 x  =
!   0.   !
!   0.25 !
!   0.5  !
!   0.75 !
!   1.   !

-->y = (1:5)'     // inny wektor kolumnowy
 y  =
!   1. !
!   2. !
!   3. !
!   4. !
!   5. !

-->p = y'*x    // iloczyn skalarny wektorow x i y
 p  =
    10.

-->Pext = y*x' // otrzymujemy macierz 5x5 rzedu 1, dlaczego?
 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. !

--> Pext / 0.25   // macierz mozna podzielic przez skalar
 ans  =
!   0.    1.    2.     3.     4.  !
!   0.    2.    4.     6.     8.  !
!   0.    3.    6.     9.     12. !
!   0.    4.    8.     12.    16. !
!   0.    5.    10.    15.    20. !

--> A^2    // podniesienie do potegi drugiej macierzy
 ans  =
!   6.      14.     36.  !
!   34.     90.     250. !
!   102.    282.    804. !

--> [0 1 0] * ans   // mozna uzyc zmiennej ans, ktora zawiera wynik ostatniego
                    // dzialania gdy nie zostal on przypisany do zadnej zmiennej
 ans  =
!   34.    90.    250. !

--> Pext*x - y + rand(5,2)*rand(2,5)*ones(x) + triu(Pext)*tril(Pext)*y;
--> // wpisz ans aby zobaczyc wynik
```

Inną, bardzo interestującą cechą charakterystyczną, jest możliwość podania jako argumentu dla funkcji (z tabeli 1) macierzy zamiast kolejnych jej elementów. Innymi słowy wpisanie instrukcji `f(A)` oznacza obliczenie wartości funkcji `f` na kolejnych elementach macierzy `A`; otrzymamy w ten sposób macierz $[f(a_{ij})]$. Przykłady:

```
-->sqrt(A)
 ans  =
!   1.           1.    1.        !
!   1.4142136    2.    2.8284271 !
!   1.7320508    3.    5.1961524 !

-->exp(A)
 ans  =
!   2.7182818    2.7182818    2.7182818 !
!   7.3890561    54.59815     2980.958  !
!   20.085537    8103.0839    5.320D+11 !
```

Uwaga: dla funkcji, które mają sens dla macierzy (co innego dla funkcji które stosuje sie dla każdego elementu macierzy...) na przykład funkcja eksponent, nazwa funkcji jest poprzedzona literą `m` *w ten sposób aby otrzymać eksponent macierzy A wystarczy wprowadzić komendę* `expm`

2.3.2 Działania na elementach macierzy

Aby pomnożyć lub podzielić dwie macierz, A i B, o tych samych wymiarach, w taki spsób aby wynikiem była macierz, również o tych samych wymiarach, w której każdy element jest iloczynem (ilorazem) odpowiednich elementów macierzy A i B należy użyc operatorów `.*` lub `./`. `A.*B` jest macierzą o elementach $[a_{ij}b_{ij}]$ natomiast `A./B` jest macierzą o lementach $[a_{ij}/b_{ij}]$. Podobnie można podnieść do potęgi każdy z elementów macierzy wpisując operator `.^`: `A.^p` pozwoli otrzymać macierz o wyrazach $[a_{ij}^p]$. Rozważmy przykład:

```
-->A./A
ans =
!  1.  1.  1. !
!  1.  1.  1. !
!  1.  1.  1. !
```

Uwagi:

- *W przypadku gdy* `A` *nie jest macierzą kwadratową działanie* `A^n` *będzie wykonywane na kolejnych elementach macierzy* `A`*. Zaleca się jednak stosowanie zapisu* `A.^n` *ponieważ jest on bardziej czytelny.*
- *Jeśli* `s` *jest skalarem oraz* `A` *jest macierzą wówczas* `s.^A` *daje macierz o wyrazach* $s^{a_{ij}}$.

2.3.3 Rozwiązywanie układów równań liniowych

Aby rozwiązać układ równań liniowych gdzie macierz współczynników jest kwadratowa, Scilab stosuje rozkład LU z częściową zamianą wierszy prowadzącą do rozwiazania dwóch trójkątnych układów równań. Jest to jednak operacja niewidoczna dla użytkownika dzięki wykożystaniu operatora `\`:

```
-->b=(1:3)'     // tworzymy wektor b
 b  =
!   1. !
!   2. !
!   3. !

-->x=A\b        // rozwiazujemy Ax=b
 x  =
!   1. !
!   0. !
!   0. !

-->A*x - b    // sprawdzamy poprawnosc wyniku
 ans  =
!   0. !
!   0. !
!   0. !
```

Aby zapamiętać ten sposób postępowania, należy mieć na uwadze układ początkowy $Ax = b$ a następnie pomnożyć układ lewostronnie przez A^{-1} (co oznacza podzielenie przez macierz A). Sposób ten daje dokładny wynik, ale w ogólności występują błędy zaokrąglenia spowodowane arytmetyką liczb zmiennoprzecinkowych.

```
-->R = rand(100,100);  // stawiamy srednik na koncu aby uniknac zalewu ekranu liczbami

-->y = rand(100,1);    // srednik, jak wyzej

-->x=R\y;              // rozwiazanie ukladu Rx=y

-->norm(R*x-y)    // funkcja norm pozwala obliczyc norme wektorow (macierzy),
                  // (obliczyc mozemy dwie normy - euklidesowa i hermitea)
 ans  =
    1.134D-13
```

Uwaga: Nie otrzymacie identycznego z moim jeżeli funkcja `rand` *nie zostanie użyta tak jak w powyższym przykładzie... W momencie gdy rozwiązanie układu liniowego jest wątpliwe, Scilab wyświetla informacje ostrzegające i pozwalające podjąć odpowiednie w takiej sytuacji działania.*

2.3.4 Indeksowanie, wydobywanie podmacierzy, konkatenacji macierzy i wektorów

Aby odniesc się do konkretnego elemetnu macierzy wystarczy przy nazwie podać w nawiasie jego indeksy. Na przykład:

```
-->A33=A(3,3)
 A33  =
    27.

-->x_30 = x(30,1)
 x_30  =
  - 1.2935412
```

```
-->x(1,30)
        !--error    21
invalid index

-->x(30)
 ans  =
  - 1.2935412
```

Uwaga: Jeżeli macierz jest wektorem kolumnowym wystarczy jednynie wpisać numer linii, w której znajduje się szukany element; analogicznie postępujemy w przypadku wektora wierszowego.

Zaletą języka Scilab jest możliwość łatwego wydobywanie podmacierzy z macierzy wyjściowej.

```
-->A(:,2)   // aby uzyskac 2 kolumne,...
 ans  =
!   1. !
!   4. !
!   9. !

-->A(3,:)   // ... 3 wiersz
 ans  =
!   3.    9.    27. !

-->A(1:2,1:2)  // podmacierz glowna rzedu 2
 ans  =
!   1.    1. !
!   2.    4. !
```

Omówmy teraz ogólną składnię. Niech macierz A ma wymiary $(n.m)$, niech dalej $v1 = (i_1, \ldots, i_p)$ oraz $v2 = (j_1, \ldots, j_q)$ oznaczają wektory (wierszowe lub kolumnowe), w których wartości są takie, że $1 \leq i_k \leq n$ et $1 \leq j_k \leq m$, wówczas `A(v1,v2)` oznacza macierz o wymiarach (p, q) utowrzoną z wyrazów macierzy A odpowiadających wierszom $i_1, i_2, \ldots, i_p$ oraz kolumnom $j_1, j_2, \ldots, j_q$.

```
-->A([1 3],[2 3])
 ans  =
!   1.    1.  !
!   9.    27. !

-->A([3 1],[2 1])
 ans  =
!   9.    3. !
!   1.    1. !
```

W praktyce dokonujemy prostrzych ekstrakcji, wydobywa się elementy umieszczone w przylegających blokach na przykład w kolumnach lub wierszach. W takim przypadku użyjemy wyrażenia `i_poczatkowe:przyrost:i_koncowe` w celu wygenerowania wektora wskaźników. Natomiast aby wygenerować pełny obszar odpowiadający wymiarowi użyjemy operatora `:` (jak widać to w pierwszym przykładzie). Zatem aby otrzymać podmacierz złożoną z 1 i 3 wiersza zastosujemy

```
-->A(1:2:3,:)    // lub inaczej A([1 3],:)
 ans  =
!   1.    1.    1.  !
!   3.    9.    27. !
```

Przejdźmy teraz do operacji konkatencaji macierzy, która umożliwia połączenie (ustawiając obok siebie) wiele macierzy w celu otrzymania jednej zwanej macierzą blokową. Dla przykładu rozważmy następującą macierz podzieloną na bloki:

$$A = \left(\begin{array}{c|ccc} 1 & 2 & 3 & 4 \\ \hline 1 & 4 & 9 & 16 \\ 1 & 8 & 27 & 64 \\ 1 & 16 & 81 & 256 \end{array}\right) \qquad = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}.$$

Należy zatem zdefiniować podmacierze $A_{11}, A_{12}, A_{21}, A_{22}$:

```
-->A11=1;

-->A12=[2 3 4];

-->A21=[1;1;1];

-->A22=[4 9 16;8 27 64;16 81 256];
```

ostatecznie otrzymujemy macierz A powstałą z połączenia 4 bloków:

```
-->A=[A11 A12; A21 A22]
A =
!  1.  2.  3.   4.   !
!  1.  4.  9.   16.  !
!  1.  8.  27.  64.  !
!  1.  16. 81.  256. !
```

Z punktu widzenia syntaktyki, macierze blokowe traktowane są jak zwykłe skalary (należy przy tym oczywiście pamiętać o zgodności wymiarów odpowidnich macierzy blokowych).

Istnieje odrębna składnia służąca do jednoczesnego usunięcia z macierzy wierszy lub kolumn: niech $v = (k_1, k_2, \ldots, k_p)$ będzie wektorem składającym się z numerów wierszy lub kolumn macierzy M. Polecenie `M(v,:)=[]` spowoduje usunięcie wierszy o numerach $k_1, k_2, \ldots, k_p$ z macierzy M, natomiast `M(:,v)=[]` usunie kolumny $k_1, k_2, \ldots, k_p$. Ponadto jeśli u jest wektorem (wierszowym lub kolumnowym), `u(v)=[]` usunie odpowiednie składowe.

Kolejność elementów macierzy. Macierze w Scilab-ie są składowane kolumna za kolumną i ta kolejność elementów wykorzystywana jest w wielu funkcjach (porównaj dla przykładu polecenie `matrix`, która umożliwia zmianę wymiarów macierzy). W szczególności dla operacji wstawiania i wydobywania możliwe jest użycie domyślnego porządku przy wykorzystaniu jedynie pojedyńczego wektora indeksów (w miejsce dwóch, jako wskaźnik kolumn lub wierszy). Oto kilka przykładów opartych na wcześniej zdefiniowanej macierzy A:

```
-->A(5)
 ans  =
    2.

-->A(5:9)
 ans  =
!   2.  !
!   4.  !
!   8.  !
!   16. !
!   3.  !

-->A(5:9) = -1  // spowoduje wstawienie elementu -1
 A  =
!   1.  - 1.  - 1.     4.    !
!   1.  - 1.    9.     16.   !
!   1.  - 1.    27.    64.   !
!   1.  - 1.    81.    256.  !
```

2.4 Informacje na temat środowiska pracy(*)

Wystarczy wpisać `who` a otrzymamy w ten sposób następujące informacje

```
-->who
your variables are...

 Anew     A        A22      A21      A12      A11      x_30     A33      x
 y        R        b        Pext     p        Ac_adj   Ac       At       E
 D        cosh     ind      xx       i        linspace M        U        O
 zeros    C        B        I        Y        c        T        startup  ierr
 scicos_pal        home     PWD      TMPDIR   percentlib        fraclablib
 soundlib xdesslib utillib  tdcslib  siglib   s2flib   roblib   optlib   metalib
```

```
elemlib  commlib  polylib  autolib  armalib  alglib   mtlblib  SCI      %F
%T       %z       %s       %nan     %inf     old      newstacksize      $
%t       %f       %eps     %io      %i       %e       %pi
using      14875 elements  out of    1000000.
          and          75 variables out of       1023
```

- Zmienne, które zostały wprowadzone `Anew, A,A22, A21, ...,b` w porządku odwrotnym do ich wczytywania;
- Nazwy bibliotek Scilab-a (posiadających rozszerzenie lib) i funkcji: `cosh`. To znaczy funkcje (te opisane w języku Scilab-a) oraz biblioteki traktowane są przez Scilab-a jak zmienne.
  Uwaga: Procedury w Scilab-ie zaprogramowane w Fortran 77 i C nazywane są „prymitywami Scilab-a" i nie są uważane za zmienne w Scilab-ie. W dalszej części tego dokumentu używa się czasem przesadnie sformułowania „prymityw" aby zaznaczyć funkcje Scilab-a (programów w języku Scilab), które są zawarte w standardowo dostępnym środowisku.
- Stałe predefiniowane, takie jak π, e i i epsilon maszynowy eps oraz dwie inne stałe, klasyczne w arytmetyce zmiennoprzecinkowej: nan - *ang. not a number*, $inf - \infty$. Zmienne, których nazwa poprzedzona jest znakiem $\%$ nie mogą zostać usunięte.
- Bardzo istotna zmienna `newstacksize` określająca rozmiar stosu (to znaczy ilość dostępnej pamięci).
- Na koniec wyświetlona zostaje informacja o ilości użytych słów 8 bitowych w odniesieniu do ich całkowitej (dostępnej) ilości a następnie ilość użytych zmiennych w stosunku do maksymalnej dozwolonej ich ilości.

Rozmiar stosu można zmienić za pomocą polecenia `stacksize(nbmbts)` gdzie `nbmbts` oznacza nowy pożądany rozmiar. Ta sama komenda bez argumentu pozwala uzyskać rozmiar stosu mieszczący maksymalną dopuszczalną liczbę zmiennych.

Tak więc chcąc usunąć wektor $v1$ ze środowiska a więc zwolnić miejsce w pamięci należy użyć polecenia `clear v1`. Polecenie `clear` użyte bez argumentu usunie wszystkie zmienne. Chcąc natomiast usunąc więcej niż jedną zmienną moża podać ich nazwy jako kolejne argumenty polecenia `clear`, na przykład: `clear v1 v2 v3`.s

2.5 Wywołanie pomocy z linii poleceń

Pomoc można otrzymać wybierając przycisk `Help` z menu okna Scilab. Począwszy od wersji 2.7 strony pomocy są w formacie HTML[5]. Możliwe jest użycie innego programu do przeglądania dokumentacji w następujący sposób

```
-->global %browsehelp; %browsehelp=[];
-->help
```

W wierszu pierwszym modyfikowana jest zmienna lokalna `%browsehelp` w taki sposób, że zawiera ona macierz pustą. Zatem wprowadzenie polecenia `help` lub wybranie odpowiedniego przycisku, zostaną zaproponowane inne alternatywne możliwości. Można tą zmienną umieścić w pliku `.scilab` dzięki czemu uniknie się powyższych manipulacji[6].

Wpisując polecenie `help` (lub klikając na przycisk `Help`) pojawi się strona HTML zawierająca odnośniki do wszystkich stron pomocy (`Scilab Programming, Graphic Library, Utilities and Elementary functions,...`). Klikając na wybraną nazwę otrzymuje się listę szystkich funkcji związanych z danym tematem, do każdej z nich dołączono krótki opis. Wybierając daną funkcję otrzymuje się stronę z pytaniami dotyczącymi danej funkcji. Jeżeli natomiast znana jest nazwa funkcji wystarczy wpisać komendę `help` *NazwaFunkcji* w oknie Scilab-a aby wejść na odpowiedną stronę. Polecenie `apropos` *słowoKluczowe* wyświetli wszystkie strony na których pojawia się *słowoKluczowe* w nazwach funkcji lub ich opisach. Przeszukiwanie opcji może okazać sie niewygodne, gdyż często wiele funkcji zgrupowanych jest pod jedną nazwą (na przykład jako Elementary functions mojetutu występuje bardzo dużo różnych funkcji, które w jakimkolwiek stopniu na tę nazwę zasługują); nie wahaj sie zatem używać polecenia `apropos`

2.6 Generator prostych wykresów

Przypuśćmy, że chcemy narysować wykres funkcji $y = e^{-x}\sin(4x)$ dla $x \in [0, 2\pi]$. Można zdefiniować podział przedziału poprzez funkcję `linespace`

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

a następnie policzyć wartości funkcji dla każdego elementu podziału wykrzystując w tym celu instrukcję

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

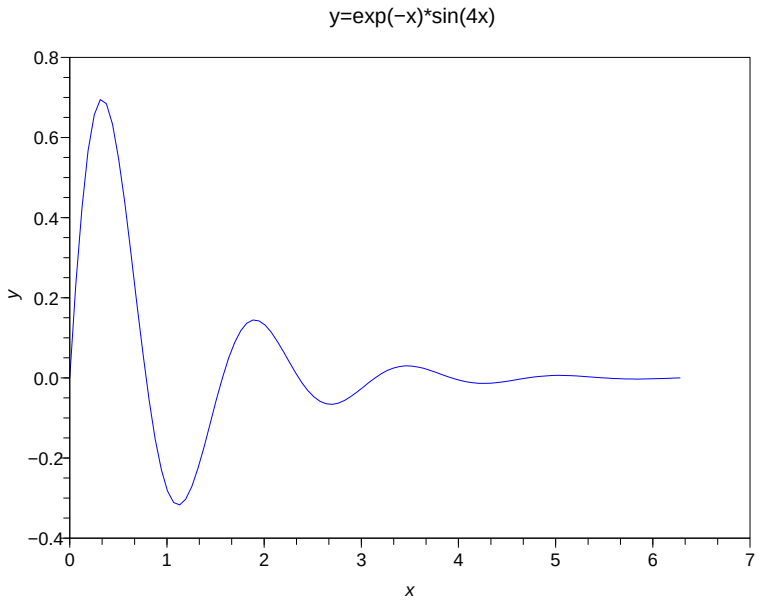
[5] Format podstawowy to XML a z niego otrzymuje się HTML.

[6] Jeśli wasza przeglądarka nie została uwzględniona należy dokonać modyfikacji pliku `SCI/macros/util/browsehelp.sci`

i ostatecznie

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

gdzie trzy ostatnie łańcuchy znaków (odpowiednio: opis osi rzędnych i odciętych oraz nazwa wykresu) są opcjonalne. Instrukcja ta pozwala narysować krzywą przechodzącą przez punkty, których współrzędne określa wektor `x` na osi odciętych oraz `y` na osi rzędnych. Jeżeli punkty układają się częściowo wzdłuż linii prostej wykres będzie tym wierniejszy im większej ilości punktów użyjemy



Rysunek 1: Prosty wykres.

Uwaga: Ta instrukcja jest ograniczona, zatem nie należy sugerować się jedynie tym rozwiązaniem. W rozdziale dotyczącym grafiki wprowadza się instrukcje `plot2d`, która jest o wiele lepsza.

2.7 Pisanie i wykonywanie skryptów

Możliwe jest wpisanie ciągu instrukcji do pliku `nazwaPliku` i ich wywołanie poprzez komendę

```
-->exec('nazwaPliku')    // lub exec("nazwaPliku")
```

Prostszą metodą jest wybranie pozycji `File Operation` znajdującą się w menu `File` okna Scilab. Menu pozwala wybrać insteresujący nas plik (ewentualnie w celu jego modyfikazji); aby go wykonać wystarczy wybrać `Exec`. Jako przykład skryptu rozważmy ponownie wykres funkcji $e^{-x}\sin(4x)$ zadając za każdym razem inny rozmiar przedziału $[a, b]$ i ilość podprzedzaiłów. Napiszmy zatem w pliku nazywającym się na przykład `script1.sce` następujące instrukcje

```
// pierwszy skrypt

a = input(" Podaj lewy kraniec przyedzialu : ");
b = input(" Podaj prawy kraniec przedzialu : ");
n = input(" Podaj ilosc podprzedzialow : ");

// obliczenie odcietych
x = linspace(a,b,n+1);

// obliczenie rzednych
y = exp(-x).*sin(4*x);

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

Uwaga:

- Aby odnaleźć się w naszych plikach Scilab-a zaleca się wykorzystać w nazwie pliku skryptowego rozszerzenie `*.sce` (w przypadku gdy plik zawiera funkcję, rozszerzeniem powinno być `*.scu`).
- Pewne edytory mogą być wyposażone w narzędzia specyficzne dla edycji w Scilab-ie (zobacz strona Scialba). Dla `emacs` istnieją dwa, z krórych lepszy pochodzi od Alexandra Vigodnera, którego najnowsze wersje można znaleźć pod adresem

  `http://www.geocities.com/avezunchik/scilab.html`

- Skrypt jest zazwyczaj pierwotnym programem dla aplikacji napisanych w Scilab-ie.

2.8 Dodatkowe informacje

2.8.1 Skracanie instrukcji przy zapisie macierzowym

Wcześniej dowiedzieliśmy się, że mnożenie macierzy przez skalar (podobnie dzielenie macierzy przez skalar) jest zdefiniowane w Scilab-ie. Natomiast Scilab stosuje uproszczeń mniej oczywistych, takich jak dodawanie skalara i macierzy. Wyrażenie `M + s` gdzie `M` jest macierza a `s` skalarem jest skrótem dla instrukcji

```
M + s*ones(M)
```

to znaczy skalar jest dodany do wszystkich elementów macierzy.

Inny skrót: w wyrażeniu typu `A./B` (oznaczającym dzielenie *element po elemencie* macierzy o tych samych wymiarach), jeżeli `A` jest skalarem wówczas wyrarzenie to jest skróconym zapisem dla instukcji :

```
A*ones(B)./B
```

otrzymuje sie macierz $[a/b_{ij}]$. Takie uproszczenia pozwalają na zapis bardziej syntetyczny w wielu przypadkach (por. ćwiczenia). Na przykład, jeśli f jest funkcją zdefiniowaną w środowisku, x jest wektorem a s jest zmienną, wówczas

```
s./f(x)
```

jest wektorem tych samych wymiarów co x, w którym i-ta współrzędna równa jest $s/f(x_i)$. W ten sposób, aby wyznaczyć wektor o współrzędnych $1/f(x_i)$, użyjemy składni :

```
1./f(x)
```

Taka składnia interpretuje `1.` jako jedynkę i wynik nie będzie właściwy. Dobrym rozwiązaniem aby otrzymać poszukiwany wynik jest ujęcie liczby w nawias lub oddzielenie liczby spacją :

```
(1)./f(x)       // lub takze 1 ./f(x)  lub  1.0./f(x)
```

2.8.2 Pozostałe uwagi dotyczące rozwiązywania układów równań liniowych (*)

1. W przypadku gdy mamy więcej wyrazów wolnych, można stosować następujące polecenie :

```
-->y1 = [1;0;0;0]; y2 = [1;2;3;4];  // przypadek gdy mamy 2 wyrazy wolne (mozna
                                    // ujac wiecej instrukcji w jednej lini)
-->X=A\[y1,y2]     // konkatenacja/laczenie y1 i y2
 X  =
!   4.          - 0.8333333 !
! - 3.           1.5       !
!   1.3333333   - 0.5       !
! - 0.25         0.0833333 !
```

   Pierwsza kolumna macierzy X jest rozwiązaniem układu równań $Ax^1 = y^1$, natomiast druga jest rozwiązaniem układu $Ax^2 = y^2$.

2. Wcześniej zobaczyliśmy, że jeżeli A jest macierzą kwadratową (n,n) oraz b jest wektorem kolumnowym o n współrzędnych (a więc macierzą (n,1)) to :

```
   x = A\b
```

da nam rozwiązanie układu równań liniowych postaci $Ax = b$. Jeżeli natomiast macierz A okaże sie macieżą osobliwą, Scilab zwróci błąd. Na przykład :

```
-->A=[1 2;1 2];

-->b=[1;1];

-->A\b
    !--error    19
singular matrix
```

Podczas gdy macierz A jest źle uwarunkowana (lub ewentualnie źle zrównoważona) odpowiedź będzie dostarczona ale wraz z komentarzem w postaci ostrzeżenia zawierającym oszacowanie odwrotności uwarunkowania ($cond(A) = ||A||\ ||A^{-1}||$):

```
-->A=[1 2;1 2+3*%eps];
-->A\b
 warning
 matrix is close to singular or badly scaled.
 results may be inaccurate. rcond =   7.4015D-17

 ans  =
!   1. !
!   0. !
```

Z drugiej strony, jeżeli macierz nie jest kwadratowa, ale posiada tę samą liczbę wirszy co wektor wyrazów wolnych, Scilab obliczy rozwiązanie (w postaci wektora kolumnowego, którego liczba współrzędnych będzie równa liczbie kolumn macierzy A) nie informując użytkownika o błędzie. W efekcie, jeżeli równanie $Ax = b$ nie ma w ogólności rozwiązania jednoznacznego [7], można zawsze wybrać wektor x który zweryfikuje pewne własności (x o minimalne normie i jednocześnie będący rozwiązaniem $\min ||Ax - b||$). W tym przypadku takie rozwiązanie jest wynikiem działanie innych algorytmów które umożliwiają (ewentualne) otrzymanie psełdo-rozwiązania[8]. Niedogodność jest taka, że jeśli zostanie popełniony błąd podczas definiowania macierzy (na przykład zdefiniowana zaostanie dodatkowa kolumna tak, że macierz będzie miała wymiar (n,n+1)) błąd ten nie zostanie wykryty natychmiast. Rozważmy przykład :

```
-->A(2,3)=1        // zaburzenie
A =
!  1.  2.  0. !
!  1.  2.  1. !

-->A\b
 ans  =
!   0.         !
!   0.5        !
! - 3.140D-16 !

-->A*ans - b
 ans  =
   1.0D-15 *

! - 0.1110223 !
! - 0.1110223 !
```

Poza zwróceniem naszej uwagi na konsekwencje tego typu zakłuceń, powyższy przykład ilustruje następujące fakty/aspekty :

[7] niech (m, n) będzie wymiarem macierzy A (gdzie $n \neq m$), mamy jedno rozwiązanie wtedy i tylko wtedy, gdy $m > n$, $KerA = \{0\}$ i w konsekwencji $b \in ImA$ ten ostatni warunek jest wyjątkiem jeśli b jest zadane z góty w K^m ; w każdym innym przypadku albo nie ma żadnego rozwiązania albo isnieje nieskończenie wiele rozwiązań

[8] w przypadkach trudnych, tzn. takich, gdzie macierz nie ma maksymalnego rzędu ($rg(A) < min(n, m)$ gdzie n i m są dwuwymiarowe) należy wyznaczyć rozwiązanie korzystając z psełdo-odwrotnej macierzy do A (`x = pinv(A)*b`).

- `x = A\y` pozwala rozwiązać problem mniej kwadratowy (kiedy macierz nie ma maksymalnego rzędu, należy lepiej użyć/wykożystać `x = pinv(A)*b`, pseudo-inwersja zostanie obliczona poprzez dekompozycję pojedynczych wartości A (tę dekompozycję można otrzymać za pomocą instukcji `svd`));
- instrukcja `A(2,3)=1` (błąd zakłucenia...) jest w rzeczywistości skrótem od :

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

  to znaczy Scilab domyśla się, że macierz A ma być uzupełniona (o 3 kolumnę) ale brakuje mu jednego elementu. W takim przypadku uzupełni je zerami.
- element na pozycji (2,2) jest równy $2 + 3\,\varepsilon_m$. Epsilon maszynowy (ε_m) maże być zdefiniowany jako bardzo duża liczba, dla której $1 \oplus \varepsilon_m = 1$ w arytmetyce zmiennoprzecinkowej[9]. W konsekwencji musimy mieć $2 \oplus 3\varepsilon_m > 2$ aby na ekranie pojawiła się 2. Wynika to z użytego domyślnie formatu danych, można go jednak zmodyfikawać używając instrukcji `format` :

  ```
  -->format('v',19)

  -->A(2,2)
  ans =
      2.0000000000000009
  ```

  W ten sposób zmienimy format domyślny `format('v',10)` (patrz `Help` aby poznać znaczenie argumentów).
- Rozwiązanie układu $Ax = b$, które nie zostało przypisane żadnej konkretniej zmniennej, jest domyślnie nazwane `ans`. Zmienną tę można następnie wykożystać do obliczenia wartości rezydualnej $Ax - b$.

3. Przy użyciu Scilab-a można również odpowiednio rozwiązać układ równań postaci $xA = b$ gdzie x i b są wektorami wierszowymi, a A jest macierzą kwadratową (poprzez transpozycję wracamy do układu klasycznego $A^\top x^\top = b^\top$), wystarczy pomnożyć prawostronnie przez A^{-1} (odpowiednikiem tej operacji jest prawostronne podzielenie przez macierz A) :

   ```
   x = b/A
   ```

   I podobnie jak poprzednio, jeśli A jest macierzą prostokątną (gdzie liczba kolumn jest równa liczbie wierszy wektora b), Scilab wyznaczy rozwiązanie.

2.8.3 Kilka prostych macierzy (*)

Suma, iloczyn współczynników macierzy, macierz pusta

Aby zsymować współczynniki macierzy, używamy funkcji `sum` :

```
-->sum(1:6)   // 1:6 = [1 2 3 4 5 6] : powinnismy zatem otrzymac 6*7/2 = 21  !!!!!
 ans  =
     21.
```

Funkcja ta przyjmuje argument dodatkowy dla obliczenia tylko wierszy lub tylko kolumn :

```
-->B = [1 2 3; 4 5 6]
 B  =
!   1.    2.    3. !
!   4.    5.    6. !

-->sum(B,"r")  // oblicza sume wyrazow w poszczegolnych kolumnach,
               // otrzymujemy wektor wierszowy postaci
 ans  =
!   5.    7.    9. !

-->sum(B,"c") // oblicza sume wyrazow w poszczegolnych wierszach,
              // otrzymujemy wektor kolumnowy postaci
 ans  =
!   6.  !
!   15. !
```

[9] ściśle mówiąc wszystkie liczby rzeczywiste x takie, że $m \leq |x| \leq M$ mogą być zakodowane/zapisane za pomocą liczby zmiennoprzecinkowej $fl(x)$ następująco : $|x - fl(x)| \leq \varepsilon_m |x|$ gdzie m i M są odpowiednio najmniejszą i największą liczbą dodatnią możliwą do zakodowania za pomocą znormalizowanej liczby zmiennoprzecinkowej.

Jednym z wyjątkowo użytecznych obiektów jest *macierz pusta*, którą definiuje sie następująco :

```
-->C =[]
 C  =
     []
```

Macierz ta ma następujące własności :`[] + A = A` i `[]*A = []`. Stosując teraz funkcję `sum` do macierzy pustej otrzymujemy naturalnie rezultat :

```
-->sum([])
 ans  =
    0.
```

według matematycznej konwencji sumowania :

$$S = \sum_{i \in E} u_i = \sum_{i=1}^{n} u_i \quad \text{gdy } E = \{1, 2, \ldots, n\}$$

jeżeli E jest zbiorem pustym wówczas $S = 0$ W analogiczny sposób, aby otrzymać iloczyn elementów macierzy, stosuje sie

funkcję `prod` :

```
-->prod(1:5)     // otrzymujemy 5! = 120
 ans  =
    120.

-->prod(B,"r")  // napisz B aby zobaczyc macierz...
 ans  =
!   4.    10.    18. !

-->prod(B,"c")
 ans  =
!   6.   !
!   120. !

-->prod(B)
 ans  =
    720.

-->prod([])
 ans  =
    1.
```

wynik otrzymuje siś zgodnie z konwencją znaną w matematyce :

$$\prod_{i \in E} u_i = 1, \text{ si } E = \emptyset.$$

Suma i iloczyn skumulowany

Funkcje `cumsum` i `cumprod` obliczają odpowiednio sumę i iloczyn skumiulowany macierzy lub wektora :

```
-->x = 1:6
 x  =
!   1.    2.    3.    4.    5.    6. !

-->cumsum(x)
 ans  =
!   1.    3.    6.    10.    15.    21. !

-->cumprod(x)
 ans  =
!   1.    2.    6.    24.    120.    720. !
```

Dla macierzy kumulowanie dokonuje siś zazwyczja jedynie w porządku kolumnowym :

```
-->x = [1 2 3;4 5 6]
 x  =
!   1.    2.    3. !
!   4.    5.    6. !

-->cumsum(x)
 ans  =
!   1.    7.     15. !
!   5.    12.    21. !
```

Podobnie jak w przypadku funkcji `sum` i `prod`, można dokonywać kumulowania według wierszy i kolumn :

```
-->cumsum(x,"r")   // suma skumulowana wedlug kolumn !
 ans  =
!   1.    2.    3. !
!   5.    7.    9. !

-->cumsum(x,"c")   // suma skumulowana wedlug wierszy !
 ans  =
!   1.    3.    6.  !
!   4.    9.    15. !
```

Pojawia się tutaj pozorna niezgodność pomiędzy nazwą a sposobem dokonywania obliczeń na wierszach i kolumnach[10] : dla funkcji `sum` i `prod` argument informuje o ostatecznej formie otrzymanego wyniku (`sum(x,"r")` pozwala otrzymać wektor wierszowy (r jak row, ang. wiersz) a więc sumę każdej kolumny) lecz takie rozumowanie dziala jedynie w przypadku tych funkcji.

minimum i maksimum wektora i macierzy

Najmniejszą i największą wartość wektora lub macierzy można wyznaczyć posługując się funkcjami `min` i `max`. Ich działanie jest takie jak w przypadku funkcji `sum` i `prod` gdzie dodatkowy argument określa czy minimum lub maksimum ma być wyznaczone dla wiersza czy dla kolumny. Dodatkowo możliwe jest określenie wskaźnika dla miejsca, w któtym znajduje sie owo minimum lub maksimum. Przykłady :

```
-->x = rand(1,5)
 x  =
!   0.7738714    0.7888738    0.3247241    0.4342711    0.2505764 !

-->min(x)
 ans  =
    0.2505764

-->[xmin, imin] = min(x)
 imin  =
    5.              // min jest osiagniete w x(5)
 xmin  =
    0.2505764

-->y = rand(2,3)
 y  =
!   0.1493971    0.475374     0.8269121 !
!   0.1849924    0.1413027    0.7530783 !

-->[ymin, imin] = min(y)
 imin  =
!   2.    2. !      // min jest osiagniete dla y(2,2)
 ymin  =
    0.1413027

-->[ymin, imin] = min(y,"r")  // minimum kazdej kolumny
```

[10] `sum`, `prod`, `cumsum`, `cumprod`, `mean`, `st_deviation`, `min`, `max`

```
 imin  =
!   1.    2.    2. !          // => min to  y(1,1) y(2,2) y(2,3)
 ymin  =
!   0.1493971    0.1413027    0.7530783 !

-->[ymin, imin] = min(y,"c")  // minimum kazdego wiersza
 imin  =
!   1. !           // min to  y(1,1)
!   2. !           //         y(2,2)
 ymin  =
!   0.1493971 !
!   0.1413027 !
```

średnia i odchylenie standardowe

Funkcje `mean` i `st_deviation` pozwalają wyznaczyć średnią i odchylenie standardowe współrzędnych wektora lub macierzy. Formuła używana dla odchylenia standardowego jest następująca :

$$\sigma(x) = \left(\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2 \right)^{1/2} , \; \bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

```
-->x = 1:6
 x  =
!   1.    2.    3.    4.    5.    6. !

-->mean(x)
 ans  =
    3.5

-->st_deviation(x)
 ans  =
    1.8708287
```

Podobnie można wyznaczyć średnią (a także odchylenie standardowe) wiersza lub kolumny macierzy :

```
-->x = [1 2 3;4 5 6]
 x  =
!   1.    2.    3. !
!   4.    5.    6. !

-->mean(x,"r")   // srednia wyrazow kazdej z kolumn macierzy
 ans  =

!   2.5    3.5    4.5 !

-->mean(x,"c")   // srednia kazdego z wierszy
 ans  =

!   2. !
!   5. !
```

Przekształcenie macierzy. Funkcja `matrix` umożliwia takie przekształcenie macierzy aby miała ona nowe wymiary (przy zachowaniu tych samych współczynników).

```
-->B = [1 2 3; 4 5 6]
 B  =
!   1.    2.    3. !
!   4.    5.    6. !

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

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

Jest to możliwe dzięki temu, że współczynniki macierzy zapamiętywane są kolumnowo. Jednym z zastowowań funkcji `matrix` jest transpozycja wektora wierszowego na kolumnowy i odwrotnie. Należy zaznaczyć, że w ogólności pozwala ona także przekształcić macierz `A` zamieniając (wektory wierszowe i kolumnowe) na wektor kolumnowy `v` : `v = A(:)`, przykład :

```
-->A = rand(2,2)
 A  =
!   0.8782165    0.5608486 !
!   0.0683740    0.6623569 !

-->v=A(:)
 v  =
!   0.8782165 !
!   0.0683740 !
!   0.5608486 !
!   0.6623569 !
```

Wektory z postępem logarytmicznym. Czasami istnieje potrzeba operownia wektorem którego współrzędne stanowią postęp logarytmiczny (tzn. takie, że stosunek dwuch sąsiednichc jest stały: $x_{i+1}/x_i = const$): można użyć w tym przypadku funkcję `logspace` : `logspace(a,b,n)`: pozwalającą otrzymać taki wektor o n współrzędnych w którym pierwsza i ostatnia wsółrzędna są odpowiednio 10^a i 10^b, np:

```
-->logspace(-2,5,8)
 ans  =
!   0.01    0.1    1.    10.    100.    1000.    10000.    100000. !
```

Wartości i wektory własne

Funkcja `spec` pozwala obliczyć wartości własne macierzy (kwadratowej!) :

```
-->A = rand(5,5)
 A  =
!   0.2113249    0.6283918    0.5608486    0.2320748    0.3076091 !
!   0.7560439    0.8497452    0.6623569    0.2312237    0.9329616 !
!   0.0002211    0.6857310    0.7263507    0.2164633    0.2146008 !
!   0.3303271    0.8782165    0.1985144    0.8833888    0.312642  !
!   0.6653811    0.0683740    0.5442573    0.6525135    0.3616361 !

-->spec(A)
 ans  =
!   2.4777836                !
! - 0.0245759 + 0.5208514i   !
! - 0.0245759 - 0.5208514i   !
!   0.0696540                !
!   0.5341598                !
```

oraz wyświetla wyniki w postaci wektora kolumnowego (Scilab stosuje metodę QR polegającą na interacyjnej dekompozycji) *Schur* macierzy). Wektory własne mogą być otrzymane za pomocą `bdiag`. W ogólności aby wyznaczyć wartości własne można wykożystać funkcję `gspec`.

2.8.4 Funkcje `size` i `length`

`size` pozwala uzyskać informację o wymiarach macierzy (w kolejności: liczba wierszy, liczba kolumn):

```
-->[nl,nc]=size(B)   // B jest macierza (2,3) z poprzedniego przykladu
 nc  =
    3.
 nl  =
    2.
```

```
-->x=5:-1:1
 x  =
!   5.    4.    3.    2.    1. !

-->size(x)
 ans  =
!   1.    5. !
```

natomiast `length` mówi o liczbie elementów macierzy (rzeczywistych lub zespolonych). W ten sposób dla wektora wierszowego lub kolumnowego otrzymuje się dokładnie liczbę jego współczynników :

```
-->length(x)
 ans  =
    5.

-->length(B)
 ans  =
    6.
```

Obie te instrukcje używane są po to aby poznać rozmiar macierzy lub wektora będącego argumentem tych funkcji. Należy dodać, że `size(A,'r')` (lub `size(A,1)`) oraz `size(A,'c')` (lub `size(A,2)`) mówią o liczbie wierszy (rows) oraz kolumn (columns) macierzy A.

2.9 Ćwiczenia

1. Zdefiniować macierz wymiaru n następująco (zob. szczegóły dotyczące funkcji `diag` za pomocą `Help`):

$$A = \begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & \ddots & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{pmatrix}$$

2. Niech A będzie macierzą kwadratową; czym będzie `diag(diag(A))` ?

3. Funkcja `tril` pozwala wydobyć trókątną dolną część macierzy (`triu` dla części trójkątnej górnej). Zdefiniować dowolną macierz kwadratową A (np. za pomocą `rand`) i skonstruować pojedynczą instrukcją macierz trójkątną górną T taką, że $t_{ij} = a_{ij}$ dla $i > j$ (części trójkątne górne macierzy A i T są jednakowe) i taką, że $t_{ii} = 1$ (T jest diagonalnie jednostkowa).

4. Niech `X` będzie macierzą (lub wektorem) zdefiniowanym w środowisku. Napisać instrukcję pozwalającą obliczyć macierz `Y` (tego zamego rozmiaru co `X`) w której element na pozycji (i, j) jest równy wartości $f(X_{ij})$ w następujących przypadkach :

   (a) $f(x) = 2x^2 - 3x + 1$
   (b) $f(x) = |2x^2 - 3x + 1|$
   (c) $f(x) = (x - 1)(x + 4)$
   (d) $f(x) = \frac{1}{1+x^2}$

5. Naszkicować wykres funkcji $f(x) = \frac{\sin x}{x}$ dla $x \in [0, 4\pi]$ (napisać skrypt).

6. *Mała ilustracja prawa wielkich liczb :* wyznaczyć za pomocą funkcji `rand` n próbek wedlug prawa zero-jedynkowego w postaci wektora x, obliczyć średnią skumulowaną tego wektora, tzn. wektor $\bar{x}$, którego każda z n współrzędnych ma wartość : $\bar{x}_k = \frac{1}{k}\sum_{i=1}^{k} x_i$ oraz naszkicować wykres ciągu otrzymanego w ten sposób. Zbadać jego zachowanie, zwiększając wartość n.

3 Programowanie w Scilabie

Scilab, oprócz prostych instrukcji (pozwalających, z wykożystaniem instrukcji macierzowch, na bardzo syntetnyczne programować bliskie językowi mamtematycznemu, co najmniej macierzowemu) dysponuje prostym aczkolwiek wystarczająco kompletnym językiem programowania. Zasadniczą różnicą w porównaniu z najpopularniejszymi dziś językami (C, C++, Pascal, ...) jest brak konieczności deklarowania zmiennych: w przeciwieństwie do operacji wcześniej wykonywanych, nie musimy w żadnym momencie precyzować rozmiaru macierzy ani jej typu (rzeczywista, zespolona, ...). To na interpreterze Scilaba spoczywa zadanie rozpoznawania każdego nowego obiektu. Takie podejście nie jest często brane pod uwagę, gdyż z punktu widzenia programisty prowadzić może do powstawania trudnych do wychwycenia błędów. W przypadku języków takich jak Scilab (czy MATLAB) nie powoduje to problemów dzięki używaniu zapisu wektorowego/macierzowego oraz gotowych instrukcji prostych, pozwalających na znaczne zmniejszenie ilości zmiennych oraz wielkości kodu (na przykład dzięki instrukcjom macierzowym unikamy znacznej ilości pętli służących do definiowania macierzy). Inna zaletą języków programowania tego typu jest dysponowanie instrukcjami graficznymi (dzięki czemu unikamy konieczności kożystania z innego programu lub stosowania bibliotek graficznych) oraz bycie interpretowanym (co pozwala na łatwe wyszukiwanie błędów bez pomocy debugera). Nie mniej jednak niedogodnym jest, że programy napisane w Scilabie są wolniejsze od tych napisanych w C (ale program w C wymaga 50 razy więcej czasu na jego poprawne napisanie). Z innej strony, w aplikacjach gdzie ilość zmiennych jest szczególnie istotna (na przykład macierz 1000 x 1000) lepiej powrócić do języka kompilowanego. Istotnym punktem wpływającym na szybkość wykonania jest konieczność stosowania maksymalnie często instrukcji prostych i instrukcji macierzowych (oznacza to ograniczenie do minimum ilość pętli)[11]. W efekcie Scilab odwołuje się także do skompilowanych funkcji (fortran) dzięki czekmu interpreter ma mniej pracy (pracuje mniej). Dla wykorzystania najlepszego z tych dwóch modeli (to znaczy szybkości języka kompilowanego i wygody środowiska takiego jak Scilab), użytkownik ma możliwość wywoływania podprogramów w fortranie(77) lub C.

3.1 Pętle

W Scilabie mamy możliwość użycia jednej z dwóch petli: `for` oraz `while`.

3.1.1 Pętla `for`

Pętla `for` służy do powtarzania pewnego ciągu instrukcji; zmienna starująca pętli przyjmuje kolejne wartości wektora liniowego

```
-->v=[1 -1 1 -1]
-->y=0; for k=v, y=y+k, end
```

Ilość iteracji (powtórzeń) określona jest przez ilość składników wektora; w i-tej iteracji wartość `k` jest równa `v(i)`. Opisowo pętle `for` przedstwić można następująco

dla $i := i_{start}$ **aż do** i_{end} **z krokiem** i_{step} **powtarzaj :**
instrukcje
koniec powtarzaj

Jako wektora można użyć wyrażenia `iStart:iStep:iEnd`; jeśli przyrost `iStep` ma wartość 1 to jak widzieliśmy wcześniej może zostać pominięty. Przedstawiona wcześniej jako przykład pętla `for` może zostać także napisana w bardziej naturalny sposób

```
-->y=0; for i=1:4, y=y+v(i), end
```

Kilka uwag:

- Zmienna sterująca może także przyjmować wartości pochodzące z macierzy. W takim przypadku ilość iteracji równa jest ilości kolumn macierzy, a zmienna sterująca w i-tym przebiegu pętli jest równa i-tej kolumnie macierzy

  ```
  -->A=rand(3,3);y=zeros(3,1); for k=A, y = y + k, end
  ```

- Dokładna składnia instrukcji `for` wygląda następująco

  ```
  for zmienna = macierz, ciag instrukcji, end
  ```

  gdzie instrukcje oddzielone są od siebie przecinkiem (lub średnikiem jeśli nie chcemy widzieć na ekranie rezultatów wykonania instrukcji). Ponieważ lista zmiennych rozdzielonych przecinkiem równoważna jest takiej samej liście rozdzielonej spacjami, dlatego w skryptach lub funkcjach częściej stosuje się wygodniejszy (bardziej czytelny) zapis:

[11] patrz również: sekcja "Kilka uwag dotyczących szybkości"

```
for  zmienna  =  macierz
   instrukcja1
   instrukcja2
   ...........
   instrukcjan
end
```

gdzie instrukcja może kończyć się średnikiem (będzie tak zawsze jesli chcemy uniknąć pojawiania się rezultaów instrukcji na ekranie)[12].

3.1.2 Pętla `while`

Pętla `while` pozwala na powtarzanie ciągu instrukcji tak długo dopóki prawdziwy jest pewnien warunek:

```
-->x=1 ; while x<14,x=2*x, end
```

Zadając warunek możemy wykorzystać następujące operatory porównania:

==	równe
<	mniejsze
>	większe
<=	mniejsze lub równe
>=	większe lub równe
~= lub <>	różne

Scilab posiada wbudowany typ logiczny, w którym dla oznaczenia prawdy stosuje się symbole `%t` lub `%T` zaś fałszu `%f` lub `%F`. Można oczywiście zdefiniować macierz lub wektor złożony z wartości logicznych. Dostępne są następujące operatory logiczne :

&	i
|	lub
~	nie

Formalna składnia pętli `while` przedstawia się jak poniżej

```
while warunek, instrukcja_1, ... ,instrukcja_N , end
```

lub (często w przypadku skryptu lub funkcji):

```
while warunek
   instrukcja_1
   ..............
   instrukcja_N
end
```

gdzie każda `instrukcja_k` może zostać zakończona średnikiem, `warunek` zaś jest wyrażeniem zwracającym wartość logiczną.

3.2 Instrukcja warunkowa

Są dwie instrukcje sterujące przebiegiem wykonania programu: *if-then-else* oraz *select-case*.

3.2.1 Instrukcja `if-then-else`

Spójrzmy na przykład poniżej :

```
-->if x>0 then, y=-x,else,y=x,end  // zmienna x powinna zostac wczesniej zdefiniowana
```

Podobnie jak to miało miejsce dla instrukcji pętli przecinki nie są znakami obowiązkowymi. Jak w większości spotykaych języków programowania, w przypadku gdy nie podejmujemy żadnego działania związanego z niespełnieniem warunku, możemy część `else` pominać. W przypadku gdy część `else` zawierać by miała kolejną instrukcję `if-then-else` zamiast ciagu `else` oraz `if` można użyć `elseif` co prowadzi do ogólnego schematu tej instrukcji

[12] jest tak tylko w przypadku, gdy mamy do czynienia ze skryptem, w przypadku funkcji, wynik instrukcji nie jest wyświetlany nawet gdy nie jest ona zakończona średnikiem; zasadę tę można zmodyfikować używając instrukcji `mode`

```
if warunek1  then
   ciag instrukcji wykonywanych
   gdy spelniony jest warunek1
elseif warunek2  then
   ciag instrukcji wykonywanych
   gdy spelniony jest warunek2

...

elseif warunekN  then
   ciag instrukcji wykonywanych
   gdy spelniony jest warunekN
else
   ciag instrukcji wykonywanych
   gdy nie jest spelniony zaden
   z warunk\'ow od 1 do N
end
```

Podobnie jak dla instrukcji `while` `warunek` jest wyrażeniem zwracającym wartość logiczną.

3.2.2 Instrukcja `select-case` (*)

Spójrzmy na przykład poniżej (proszę sprawdzić jego działanie dla różnych wartości zmiennej `num`)[13]

```
-->num = 1, select num, case 1, y = 'przypadek 1', case 2, y = 'przypadek  2',...
-->else, y = 'inne przypadki        ', end
```

który w skrypcie lub funkcji przyjmie raczej postać

```
// zakladamy, ze zmienna num zostala dobrze okreslona
select  num
case 1 y = 'przypadek 1'
case 2 y = 'przypadek 2'
else   y = 'inne przypadki'
end
```

Scilab sukcesywnie porównuje wartość zmiennej `num` z możliwymi wartościam przypadków (1, potem 2). Jeśli zajdzie równość, wykonywane są instrukcję począwszy od `case` równego zmiennej `num` aż do końca instrukcji `select-case` (tutu jak działa ten select-case??). Przekazanie sterowania do nieobowiązkowej gałęźi `else` ma miejsce w sytuacji gdy nie powiodą się wszystkie wcześniejsze testy. Formalnie instrukcja ta przedstawia się następująco

```
select zmiennaTestjaca
case wyrazenie1
    ciag instrukcji wykonywany gdy
    zmiennaTestuaca rowna jest wyrazeniu1
...

case wyrazenieN
    ciag instrukcji wykonywany gdy
    zmiennaTestujaca rowna jest wyrazeniuN
else
    ciag instrukcji wykonywany gdy
    zmiennaTestujaca nie jest rowna zadnemu
    z wyrazen od 1 do N
end
```

gdzie `wyrazeniei` jest wyrażeniem zwracającym wartość, która może zostać porównana ze `zmiennaTestujaca`. Konstrukcja `select-case` równoważna jest następującej konstrukcji `if-then-else`

```
if zmiennaTestujaca ==  wyrazenie1 then
   ciag instrukcji wykonywany gdy
```

[13] Zmienna `y` jest łańcuchem znaków – patrz kolejny rozdział.

```
   zmiennaTestujaca rowna jest wyrazeniu1

...

elseif zmiennaTestujaca = wyrazenieN then
   ciag instrukcji wykonywany gdy
   zmiennaTestujaca rowna jest wyrazeniuN
else
   ciag instrukcji wykonywany gdy
   nie jest speniony zaden z powyzszych
   warunkow
end
```

3.3 Inne rodzaje zmiennych

Do tej pory używaliśmy zmiennych będących macierzami (wektorami, skalarami) złożonymi z wartości typu rzeczywistego, zespolonego lub logicznego. Oprócz nich mamy również do dyspozycji zmienne reprezentujące łańcuchy znaków oraz listy.

3.3.1 Łańcuchy zanków

W przykładzie dotyczącym konstrukcji `select-case` zmienna `y` jest typu *łańcuch znaków*. W Scilabie łańcuchy znaków ograniczane są za pomocą znaku apostrofu lub cudzysłowu (jeśli chcemy któryś z tych znaków wykorzystać w takim łańcuchu, to należy napisać go dwukrotnie). Aby przypisać zmiennej `czyToPrawda` wartość

```
Czy Scilab jest "cool" ?
```

należy napisać

```
-->czyToPrawda = "Czy Scilab jest ""cool"" ?"
```

lub równoważnie

```
-->czyToPrawda = 'Czy Scilab jest ""cool"" ?'
```

Można także zdefiniować macierz złożoną z łańcuchów znaków

```
-->M = ["a" "bc" "def"]
 M  =
!a  bc  def  !

-->size(M)  // w celu otrzymania wymiarow skladowych
 ans  =
!   1.    3. !

-->length(M)
 ans  =
!   1.    2.    3. !
```

Zauważmy, że w tym przypadku `length` nie zachowuje się tak samo jak dla macierzy liczbowej. Zwraca bowiem jako wynik macierz o takim samym wymiarze jak `M`, w której współczynnik określony przez współrzędne (i, j) równy jest co do wartości ilości znaków w łańcuchu na pozycji (i, j).

Do łączenia (konkatenacji) łańcuchów używamy operatora `+`

```
-->s1 = 'abc'; s2 = 'def'; s = s1 + s2
s =
 abcdef
```

podciąg wydobywamy zaś (operacja extrakcja) przy pomocy funkcji `part`

```
-->part(s,3)
ans =
 c

-->part(s,3:4)
ans =
 cd
```

Drugim argumentem funkcji `part` jest wektor określający indeksy znaków jakie mamy wydobyć z łańcucha.

3.3.2 Listy (*)

Lista jest uporządkowanym zbiorem obiektów Scilaba (macierzy i skalarów rzeczywistych, zespolonych czy też złożonych z łąńcuchów znaków, wartości logicznych, funkcji, list, ...; każdemu elementowi przypisuje się numer). Dostępne są dwa rodzaje list: lista obiektow oraz lista typów. Oto przykład listy obiektów :

```
-->L=list(rand(2,2),["Obym skonczyl" " ten skrypt..."],[%t ; %f])
 L  =


       L(1)

!   0.2113249    0.0002211 !
!   0.7560439    0.3303271 !

       L(2)

!Obym skonczyl   ten skrypt...  !

       L(3)

! T !
! F !
```

W ten oto sposób zdefiniowaliśmy listę, w której pierwszy element jest macierzą o wymiarze $(2, 2)$, drugi wektorem łańcuchów znakowych a trzeci wektorem o składnikach logicznych. Oto kilka podstawowych operacji na listach

```
-->M = L(1)  // extrakcja (wydobycie) pierwszego sk\l{}adnika
 M  =

!   0.2113249    0.0002211 !
!   0.7560439    0.3303271 !

-->L(1)(2,2) = 100;  // modyfikowanie pierwszego skladnika

-->L(1)
 ans  =

!   0.2113249    0.0002211 !
!   0.7560439    100.      !

-->L(2)(4) = " przed wakacjami !";  // modyfikacja drugiego skladnika

-->L(2)
 ans  =

!Obym skonczyl   ten skrypt...      przed wakacjami !  !

-->L(4)="Aby dodac czwarty skladnik"
 L  =


       L(1)

!   0.2113249    0.0002211 !
!   0.7560439    100.      !

       L(2)
```

```
!Obym skonczyl   ten skrypt...      przed wakacjami !  !

       L(3)

! T !
! F !

       L(4)

  Aby dodac czwarty skladnik

-->size(L)   // ile elementow ma lista
 ans  =

    4.

-->length(L)  // podobnie jak wyzej
 ans  =

    4.

-->L(2) = null()  // usuniecie drugiego elementu
 L  =


       L(1)

!   0.2113249    0.0002211 !
!   0.7560439    100.      !

       L(2)

! T !
! F !

       L(3)

  Aby dodac czwarty skladnik

 -->Lbis=list(1,1:3)    // definicja innej listy
 Lbis  =


       Lbis(1)

    1.

       Lbis(2)

!   1.    2.    3. !


-->L(3) = Lbis  // 3 skladnik L jest teraz lista
 L  =


       L(1)
```

```
!   0.2113249    0.0002211 !
!   0.7560439    100.      !

       L(2)

! T !
! F !

       L(3)

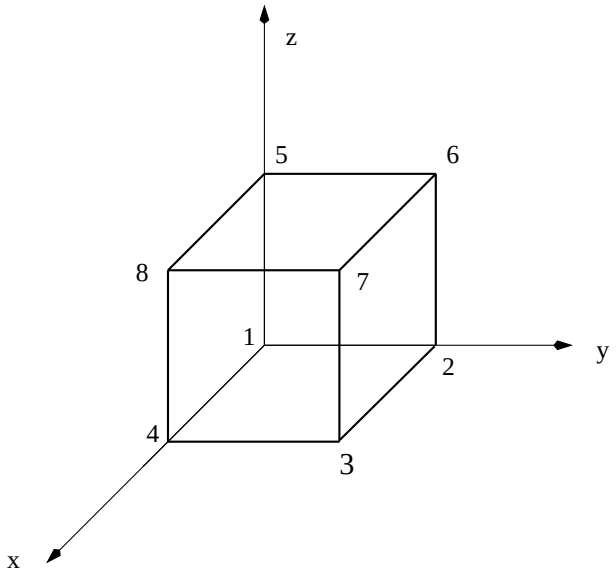

        L(3)(1)

    1.

        L(3)(2)

!   1.    2.    3. !
```

Przejdźmy teraz do list typów. W tych listach pierwszy element jest łańcuchem określającym typ elementów listy (co pozwala zdefiniować nowy typ danych a następnie określić operatory na tym typie danych), kolejne elementy mogą być dowolnymi obiektami Scilaba. Pierwszy element może być także wektorem złożonym z łańcuchów znaków; pierwszy określa typ elementów listy, pozostałe zaś służą jako indeksy elementów listy (w miejsce ich numerów na liście). Rozważmy następujacy przykład: chcemy reprezentować wielościan (w którym wszystkie ściany mają identyczną ilość krawędzi). W tym celu przechowujemy w macierzy współrzędne wszystkich wierzchołków (o wymiarze (3,ilośćWierzchołków) do której odwoływać się będziemy używając ciagu *coord*. Następnie definiujemy macierz (o wymiarze (ilośćWierzchołkówŚciany, ilośćŚcian)) określającą związek pomiędzy ścianą a wierzchołkami: dla każdej sciany podajemy numery wierzchołków ją opisujących w kolejności zodnej z regułą korkociągu dla zewnętrznego wektora normalnego. Do tej listy odwoływać się będziemy pisząc *fence*. Dla sześcianu (porównaj



Rysunek 2: Numéros des sommets du cube

```
:cube
```

rysunej 2) będzie to wyglądało jak poniżej

```
P=[ 0 0 1 1 0 0 1 1;...     // wspolrzedne wierzcholkow
    0 1 1 0 0 1 1 0;...
    0 0 0 0 1 1 1 1];

connect = [1 5 3 2 1 1;...  // zaleznosci pomiedzy wierzcholkami i scianami
           2 8 7 6 5 4;...
```

```
          3 7 8 7 6 8;...
          4 6 4 3 2 5];
```

Pozostaje zdefiniować listę tutu typee, która zawierać będzie wszystkie informacje o sześcianie

```
-->Cube = tlist(["polyedre","coord","fence"],P,connect)
 Cube  =


       Cube(1)

!polyedre  coord  fence  !

       Cube(2)

!   0.    0.    1.    1.    0.    0.    1.    1. !
!   0.    1.    1.    0.    0.    1.    1.    0. !
!   0.    0.    0.    0.    1.    1.    1.    1. !

       Cube(3)

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

W celu wskazania pewnego elementu z wykorzystniem jego numeru, można wykorzystać odpowiadający jemy łańcuch znaków:

```
-->Cube.coord(:,2)  // lub inaczej  Cube("coord")(:,2)
 ans  =

!   0. !
!   1. !
!   0. !

-->Cube.fence(:,1)
 ans  =

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

Pomijając te szczególne możliwości, listami typów manipuluje się tak samo jak innymi. Ich zaletą jest możliwość samodzielnego zdefiniowania (rozszerzenia) operatorów `+`, `-`, `*`, `/` itd. na listę danych tego typu (porównaj przyszłą wersję tej dokumentacji, albo lepiej stronę Scilaba z podobnymi informacjami.

3.3.3 Kilka przykładów z wektorami i macierzami logicznymi

Typ boolowski (używany do reprezentacji wartości logicznych typu *prawda* i *fałsz*) okazuje się często przydatny ze względów praktyznych podczas manipuowania macierzami. Gdy porównujemy dwie macierze tego samego wymiaru za pomocą jednego z operatorów porównania (<, >, <=, >=,==, ~=) jako wynik otrzymujemy macierz o wartościach logicznych, również tego samego formatu, której zawartość jest wynikiem porównania *element po elemencie* odpowiadających sobie elementów porównywanych macierzy

```
-->A = rand(2,3)
 A  =
!   0.2113249    0.0002211    0.6653811 !
!   0.7560439    0.3303271    0.6283918 !

-->B = rand(2,3)
 B  =
```

```
!   0.8497452    0.8782165    0.5608486 !
!   0.6857310    0.0683740    0.6623569 !

-->A < B
 ans  =
! T T F !
! F F T !
```

Jeśli natomiast porównujemy macierz z liczbą, a więc `A<s` gdzie `s` jest skalarem to faktycznie dokonujemy porównania `A<s*one(A)`

```
-->A < 0.5
 ans  =
! T T F !
! F T F !
```

Operatory logiczne również stosuje się dla macierzy na zasadzie *element po elemencie*

```
-->b1 = [%t %f %t]
 b1  =
! T F T !

-->b2 = [%f %f %t]
 b2  =
! F F T !

-->b1 & b2
 ans  =
! F F T !

-->b1 | b2
 ans  =
! T F T !

-->~b1
 ans  =
! F T F !
```

Ponadto istnieją dwie przydatne funkcje pozwalające dokonać wektoryzacji testu

1. Funkcja `bool2s` przekształca macierz boolowską w macierz o takich samych wymiarach przekształcając wartość logiczna *prawda* na 1, *fałsz* zaś na 0

   ```
   -->bool2s(b1)
    ans  =
   !   1.    0.    1. !
   ```

2. Funkcja `find` zwraca współrzędne wartości *prawda* w macierzy boolowskiej

   ```
   -->v = rand(1,5)
    v  =
   !   0.5664249    0.4826472    0.3321719    0.5935095    0.5015342 !

   -->find(v < 0.5)  // v < 0.5 daje wektor boolowski
    ans  =
   !   2.    3. !
   ```

Stosowanie tych funkcji jest dosyć kosztowne czasowo (porównaj "Kilka uwag związanych z szybkością wykonania"). Funkcje `and` oraz `or` oznaczające odpowiednio iloczyn logiczny (i) oraz sumę logiczną (lub) wszystkich elementów macierzy boolowskiej dając w efekcie jedną wartość logiczną. Funkcje to mogą przyjmować dodatkowy argument określający czy odpowiednie działanie logiczne ma zostać wykonane na wierszach czy kolumnach macierzy.

3.4 Funkcje

W celu zdefiniowania funkcji w Scilabie, najbardziej odpowiednią metodą jest zapisanie jej w utoworzonym pliku; będzie można do niego dopisywać kolejne funkcje grupując na przykład te związane z tym samym czy podobnym zagadnieniem czy aplikacją. Każda funkcja powinna rozpoczynać się w następujący sposób

```
function [y1,y2,y3,...,yn]=nazwaFunkcji(x1,....,xm)
```

gdzie `xi` są argumentami funkcji, natomiast `yi` wartościami przez nią zwracanymi. Dalej występuje ciało funkcji czyli wszystkie instrukcje niezbędne do wykonania przez funkcję określonego zadania. Funkcja powinna kończyc się słowem kluczowym `endfunction`. Oto pierwszy przykład

```
function [y] = silnia1(n)
   // silnia; zakladamy, ze n jest liczba naturalna
   y = prod(1:n)
endfunction
```

Załóżmy, że powyższą funkcję zapisaliśmy w pliku o nazwie `silnia1.sci`[14]. Aby można z niej korzystać w Scilabie należy ją najpierw wczytać

```
getf("silnia1.sci")  // lub inaczej exec("silnia1.sci")
```

Można tego dokonać także z menu `File operations` tak jak dla skryptów. Dopiero teraz możemy użyć zdefiniowanych w danym pliku funkcji, zarówno w „samodzielnych" poleceniach jak i w skryptach czy innych funkcjach.

```
-->m = silnia1(5)
 m  =
    120.

-->n1=2; n2 =3; silnia1(n2)
 ans  =
    6.

-->silnia1(n1*n2)
 ans  =
    720.
```

Zanim przejdziemy do kolejnych przykładów musimy ustalić jeszcze pewną terminologię. Wartości zwracane przez funkcję (`yi`) oraz te przez nią zwracane (`xi`) nazywać będziemy *argumentami formalnymi*. Używając funkcji na przykład w skrypcie czy innej funkcji

```
argS = silnia1(argE)
```

użyte argumenty (`argS` oraz `argE`) nazywać będzimy *argumentami efektywnymi*. I tak w pierwszym przykładzie argumentem wejściowym jest stała o wartości 5, w drugim zmienna `n2`, natomiast w trzecim wyrażenie `n1*n2`. Związek między argumentem efektywnym a formalnym może obiawiać się na różne sposoby (por. następny paragraf, gdzie precyzuje się te zagadnienia przy użyciu środowiska Scilaba). ce qui concerne Scilab).

`trinom`

Drugi przykład pokazuje funkcję znajdującą pierwiastki trójmianu kwadratowego

```
function [x1,x2] = trinom(a, b, c)
  // oblicza pierwiastki trojmianu kwadratowego postaci x^2 + b x + c = 0
  // a, b oraz c musza byc liczbami rzeczywistymi lub zespolonymi roznymi od zera
  delta = b^2 - 4*a*c
  x1 = (-b - sqrt(delta))/(2*a)
  x2 = (-b + sqrt(delta))/(2*a)
endfunction
```

Oto wyniki trzech prób jej użycia

```
-->[r1, r2] = trinom(1,2,1)
 r2  =
  - 1.
```

[14] Tradycyjnie plik zawierający funkcję posiada rozszerznie `.sci` natomiast skrypt `.sce`

```
 r1  =
  - 1.

-->[r1, r2] = trinom(1,0,1)
 r2  =
    i
 r1  =
  - i

-->trinom(1,0,1)  // wywolanie bez przypisania do zmiennej
 ans  =
  - i
```

Zauważmy, że w trzecim przypadku nie widzimy drugiego z pierwiastków. Jest to normalne zachowanie w sytuacji gdy funkcja zwraca więcej niż jedną wartość i nie zostaną one przypisane do zmiennych (tak jak ma to miejsce w drugim przypadku). W takiej sytuacji Scilab wykorzystuje domyślną zmienną `ans` aby przechowywać wynik; `ans` będzie przyjmowało kolejen zwracane wartości aby ostatecznie przyjąć wartość równą tej zwróconej jako ostaniej.

Teraz trzeci przykład. Należy rozwinąć w punkcie t wielomian zapisany w bazie Newtona (*Uwaga: Dla* $x_i = 0$ *otrzymujemy bazę kanoniczną.*)

$$p(t) = c_1 + c_2(t - x_1) + c_3(t - x_1)(t - x_2) + \cdots + c_n(t - x_1)\ldots(t - x_{n-1}).$$

Używając rozkładu na czynniki i wykonując obliczenia od prawej do lewej (tutaj dla $n = 4$)

$$p(t) = c_1 + (t - x_1)(c_2 + (t - x_2)(c_3 + (t - x_3)(c_4))),$$

otrzymujemy algorytm Hornera

(1) $p := c_4$

(2) $p := c_3 + (t - x_3)p$

(3) $p := c_2 + (t - x_2)p$

(4) $p := c_1 + (t - x_1)p.$

Uogólniając to na dowolne n otrzymujemy następującą funkcję Scilaba

```
function [p]=myhorner(t,x,c)
   // rozwija wielomian c(1) + c(2)*(t-x(1)) + c(3)*(t-x(1))*(t-x(2)) +
   //                   ... + c(n)*(t-x(1))*...*(t-x(n-1))
   // zgodnie z algorytmem Hornera
   n=length(c)
   p=c(n)
   for k=n-1:-1:1
      p=c(k)+(t-x(k))*p
   end
endfunction
```

Jeśli wektory `coef`, `xx`, `tt` zostały dobrze określone instrukcja

```
val = myhorner(tt,xx,coef)
```

spowoduje przypisanie do `val` wartości równej

$$coef_1 + coef_2(tt - xx_1) + \cdots + coef_m \prod_{i=1}^{m-1}(tt - xx_i)$$

Małe przypomnienie: instrukcja `length` zwraca iloczyn dwóch wymiarów macierzy dając liczbę elementów co w przypadku wektora (kolumnowego lub wierszowego) równoważne jest jego długości. Instrukcja ta, wraz z instrukcją `size` zwracającą ilość wierszy i ilość kolumn, pozwala zrezygnować z przekazywania do funkcji informacji o wymiarze przekazywanych obiektów.

3.4.1 Przekazywanie parametrów (*)

Przekazywanie parametrów odbywa się przez referencję jeśli wewnątrz funkcji nie są one modyfikowane oraz przez wartość[15] w przeciwnym razie (to znaczy, parametry wejściowe funkcji nie mogą być modyfikowane). Uwzględnienie tego faktu w waszych programach może spowodować w znacznym stopniu przyspieszenie pewnych funkcji. Poniżej przedstawiamy sztuczny, ale dobrze ilustrujący zagadnienie, przykład ukazujący koszt związany z przekazywaniem parametrów przez wartość.

```
function [w] = toto(v, k)
  w = 2*v(k)
endfunction

function [w] = titi(v, k)
  v(k) = 2*v(k)
  w = v(k)
endfunction

// skrypt testujacy
m = 200000;
ilPowtorzen = 2000;
x = rand(m,1);
timer(); for i=1:nb_rep; y = toto(x,300); end; t1=timer()/ilPowtorzen
timer(); for i=1:nb_rep; y = titi(x,300); end; t2=timer()/ilPowtorzen
```

Otrzymane wyniki będą różniły się w zależności od maszyny na jakiej działa Scilab; my otrzymaliśmy

```
 t1  =  0.00002
 t2  =  0.00380
```

Wraz z zakończeniem funkcji destrukcji ulegają wszystkie jej zmienne wewnętrzne.

3.4.2 Wstrzymywanie funkcji

Podczas debugowanie funkcji przydatna okazuje się funkcja `disp(v1, v2, ...)` pozwalająca wyświetlić wartości zmiennych `v1`, `v2`, ... Należy mieć na uwadze, że funkcja `disp` wyświetla wartości swoich argumentów w **odwrotnej kolejności**. W celu chwilowego wstrzymania wykonania programu można użyć funkcji `pause`; po jej napotkaniu mamy możliwość sprawdzenia wartości wszystkich zdefiniowanych do tej pory zmiennych (znak zachęty `-->` Scilaba zmieni się na `-1->`). Polecenie `resume` powoduje wznowienie przebiegu obliczeń.

Istnieje także inny sposób wskazywania miejsca wstrzymania programu (tak zwany *break point*, *ang.*) niż dodawanie instrukcji `pause`

`setbpt(`*NazwaFunkcji* `[`, *numerLinii* `])`

Argumentem funkcji `setbpt` jest `NazwaFunkcji`, która ma być wstrzymywana oraz opcjonalny numer wiersza, w którym ma to nastąpić (domyslną wartością jest 1). Po napotkaniu przez Scilaba wskazanego miejsca zatrzymania, dalej wszystko odbywa się tak jak gdyby zastosowano instrukcję `pause` **po** instrukcji wskazanej przez `numerLinii`. Punkt zatrzymania może zostać usnięty przez

`delbpt(`*NazwaFunkcji* `[`, *numerLinii* `])`

Jeśli nie zostanie wskazana żadna linia, wszystkie punkty zatrzymania zostaną usunięte. Zdefiniowane do tej pory punkty zatrzymania można zobaczyc korzystając z `dispbpt ()`. Oto przykład związany z wcześniej zdefiniowaną funcją `trinom` (patrz strona 33). Załóżmy, że wczytaliśmy uprzednio tę funkcję czy to za pomocą funkcji `getf` czy też `exec`.

```
-->setbpt("trinom")   // pierwszy punkt zatrzymania
-->[r1,r2] = trinom(1,2,7)  // rozpoczynamy obliczenia => zatrzymanie
Stop after row     1 in function trinom :

-1->a    // sprawdzamy wartosci zmiennych
 a  =
    1.

-1->b
```

[15] Przez wartość czyli we wnętrzu funkcji pracujemy na kopii przekazywanego argumentu. Przez referencję czyli we wnętrzu funkcji pracujemy na oryginalnym argumecie. (*przyp. tłum.*)

```
 b  =
    2.

-1->dispbpt()   // sprawdzamy punkty zatrzymania
 breakpoints of  function :trinom
          1

-1->setbpt("trinom",5)  // dodajemy kolejny punkt zatrzymania

-1->x1  // x1 nie zostal jeszcze zdefiniowany
x1
  !--error     4
undefined variable : x1

-1->resume   // wznawiamy wykonani az do nastepnego punktu zatrzymania
Stop after row     5 in function trinom :

-1->x1     // teraz mozemy sprawdzic jaka wartosc ma x1
 x1  =
   - 1. - 2.4494897i

-1->x2    // x2 nie zostal jeszcze zdefiniowany (nastapi to dopiero w linii 6)
x2
  !--error     4
undefined variable : x2

-1->dispbpt()   // sprawdzamy punkty zatrzymania
 breakpoints of  function :trinom
         1
         5

-1->resume   // wznawiamy wykonanie funkcji
 r2  =
   - 1. + 2.4494897i
 r1  =
   - 1. - 2.4494897i

-->delbpt("trinom")  // usuwamy wszystkie punkty zatrzymania
```

4 Grafika

W zakresie tworzenia grafiki Scilab posiada wiele możliwości zarówno jeśli brać pod uwagę operacje niskopoziomowe jak też bardziej złożone funkcje pozwalające operować skomplikowanymi obiektami. W dalszym ciągu zostanie wyjaśniona jedynie mała część dostępnych możliwości. *Uwaga:* Dla osób znających instrukcje graficzne MATLABA Stephane Mottelet napisała (tutu czy napisal) bibliotekę funkcji graficznych wywoływanych w stylu MATLABA; dostępna jest ona pod adresem

`http://www.dma.utc.fr/~mottelet/scilab/`

4.1 Okna graficzne

Wywołanie instrukcji graficznej takiej jak `plot` czy `plot3d` powoduje umieszczenie rysowanego obrazu w oknie raficznym o numerze 0. Wywołanie funkcji graficznej powoduje na ogół powstanie nowego obrazu na już istniejącym, stąd potrzeba uprzedniego wymazania zawartości okna przy pomocy funkcji `xbasc()`. Wymazania zawartości okna można także dokonać wybierając z menu `File` opcję `clear`. Manipulowanie oknami możliwe jest dzieki następującym funkcjom

`xset("window",num)`	okno o numerze `num` staje się oknem bieżącym; jeśli żadne okno nie istniało to zostanie utworzone
`xselect()`	uaktywnia bieżące okno; jeśli żadne okno nie istniało to zostanie utworzone
`xbasc([num])`	wymazuje zawartość okna o numerze `num` ; jeśli zostanie pominięty numer to wymazana zostanie zawartość bieżącego okna
`xdel([num])`	powoduje zamknięcie okna o numerze `num` ; jeśli zostanie pominięty numer to zamknięte zostanie bieżące okno

Jako generalną zasadę można przyjąć, że po wybraniu bieżącego okna (przez `xset("window",num)`) za pomoca instrukcji `xset("nom",a1,a2,...)` możemy dokonywać zmian w parametrach związanych z oknem. `nom` określa nazwę parametru, który chcemy zmienić jak na przykład `thickness` gdy chcemy dostosować grybość strzałek czy `colormap` gdy chcemy zmienić używaną paletę kolorów. Za nazwą podajemy jeden lub więcej argumentów wmaganych do ustawienia nowej wartości parmetru. Zbiór tych parametrów nazywany jest kontekstem graficznym; każde okno posiada swój taki kontekst. W celu zdobycia większej ilość informacji na temat parametów (których jest całkiem pokaźny zbiór) odsyłsamy do `Help`-u i tematu `Graphic Library`. W większości sytuacji mogą one być zmieniane interaktywnie przez menu graficzne, ukazujące się po wydaniu polecenia `xset()`. (*Uwaga: W tym menu dostępne jest również podokno opisujące kolory; nie jest jenak możliwe wprowadzanie zmian do tegoż opisu.* Rodzina funkcji `[a1,a2,...]=xget('nazwa')` pozwala otrzymać parametry związane z pewnym kontekstem graficznym.

4.2 Wprowadzenie do `plot2`

Wcześnie mieliśmy już do czynienia z prosta instrukcją `plot`. Jeśli jednak zamierzamy wykreślic więcej krzywych lepiej stosować funkcję `plot2d`. Najprościej można ją użyć jak to pokazano poniżej

```
x=linspace(-1,1,61)'; // odci\k{e}te (wektor kolumnowy)
y = x.^2; // rz\k{e}dne (r\'ownie\.z wektor kolumnowy)
plot2d(x,y) // --> il lui faut des vecteurs colonnes ! tutu
```

Dołóżmy teraz inną krzywą . . .

```
ybis = 1 - x.^2;
plot2d(x,ybis)
xtitle("Krzywe...")  // dodajemy tytu\l{}
```

. . . i jeszcze jedną, która tutu jest w innej skali[16] niż poprzednie

```
yter = 2*y;
plot2d(x,yter)
```

Zauważmy, że Scilab przeskalował okno tak aby zmieściła się w nim trzecia krzywa, ale także odrysował dwie poprzednie krzywe w nowej skali (wydaje się to naturalne, ale mechanizm ten nie był obecny aż do wersji 2.6 wprowadzając czesto w błąd). Możliwe jest wykreślenie tych trzech krzywych za jednym razem

```
xbasc() // aby wyczy\'scic obszar rysowania
plot2d(x,[y ybis yter]) // konkatenacja macierzy
xtitle("Krzywe...","x","y") // tytu\l{} i nazwy dla obu osi
```

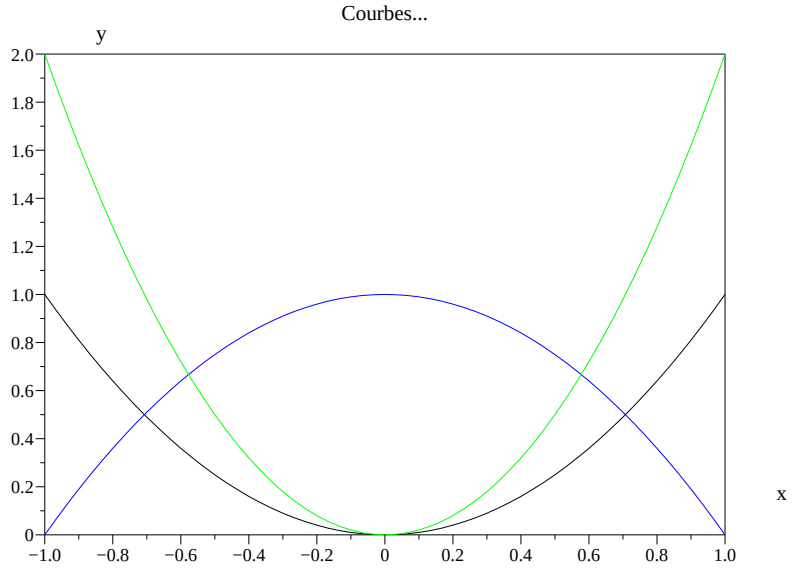
Otrzymujemy w ten sposób wykres podobny do tego przedstawionego na rysunku 3. W celu jednoczesnego wykreślenia kilku krzywych, instrukcja przyjmuje postać `plot2d(Mx,My)` gdzie `Mx` oraz `My` są macierzami o identycznym wymiarze, ilość kolumn `nc` jest równa ilości krzywych a *i*-ta krzywa jest otrzymywana na podstawie wektorów `Mx(:,i)` (odcięte) i `My(:,i)` (rzędne). W sytuacji gdy wektor odciętych jest taki sam dla wszystkich krzywych (jak w naszym przypadku), można podać go jeden raz zamiast powtarzać *nc* razy (`plot2d(x,My)` zamiast `plot2d([x x .. x],My)`).

4.3 `plot2d` z argumentami opcjonalnymi

Ogólna składnia przedstawia się następująco

```
plot2d(Mx,My <,opt_arg>*)
```

[16] Pod pojęciem skali rozumiemy tutaj obszar w jakim zostanie wykreślona krzywa oraz ewentualne dodatkowe cechy.



`fig:1`

Rysunek 3: Les fonctions x^2, $1-x^2$ et $2x^2$

gdzie `<,opt_arg>*` oznacza opcjonalny ciąg dodatkowych argumentów postaci[17]

słowoKluczowe=wartość

podanych w dowolnej kolejności. Pokażemy podstawowe epcje przy pomocy kilku przykładów

1. **Wybór koloru i definiowanie legendy** W poprzednim przykładzie można było zaobserwować, że Scilab wykreślił 3 krzywe przy użyciu 3 różnych (pierwszych) kolorów określonch w domyślnej palecie kolorów

1	czarny	5	czerwony	23	fioletowy
2	niebieski	6	fiołkowo-różowy	26	brązowy
3	jasno-zielony	13	ciemno-zielony	29	różowy
4	cyan	16	jasno-niebieski tutu	32	jaune orangé

   Instrukcja `xset()` pozwala na ich zmianę [18]. Celem wybrania koloru używamy zapisu `styl=vect`, gdzie `vect` jest wektorem liniowym zawierającym numery kolorów dla każdej krzywej. Legendę otrzymujemy pisząc `leg=str`, gdzie `str` jest łańcuchem znaków postaci `leg1@leg2@...` w którym `legi` jest podpisem dla `i`-tej krzywej. Oto przykład (porówaj rysunek (4)) :

```
x = linspace(0,15,200)';
y = besselj(1:5,x);
xbasc()
plot2d(x, y, style=[2 3 4 5 6], leg="J1@J2@J3@J4@J5")
xtitle("Funkcje Bessela J1, J2,...","x","y")
```

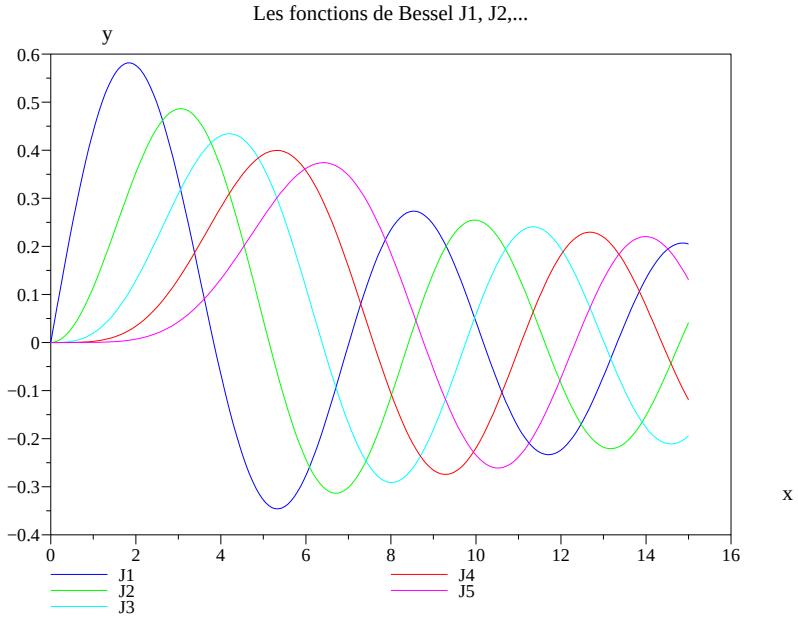
   Ponieważ kolejność argumentów opcjonalnych nie jest istotna równie dobrze można napisać

```
plot2d(x, y, leg="J1@J2@J3@J4@J5", style=[2 3 4 5 6])
```

2. **Wkresy zawierające symbole** Czasami, gdy ważne jest wyraźne wskazanie punktów przez które wykres przechodzi, przydatne mogą okazać się znaczniki tutu. Możemy wybrać jeden z kilku rodzajów zancznika przedstawionych poniżej

styl	0	-1	-2	-3	-4	-5	-6	-7	-8	-9
symbol	·	+	×	⊕	♦	♢	△	▽	♣	◯

[17] argumentFormalny = argumentEfektywny
[18] tutu



`style`

Rysunek 4: Wybrany styl i legenda

Dla przykładu spróbujmy (porównaj rysunek (5)) :

```
x = linspace(0,2*%pi,40)';
y = sin(x);
yp = sin(x) + 0.1*rand(x,"normal");
xbasc()
plot2d(x, [yp y], style=[-2 2], leg="y=sin(x)+perturbation@y=sin(x)")
```

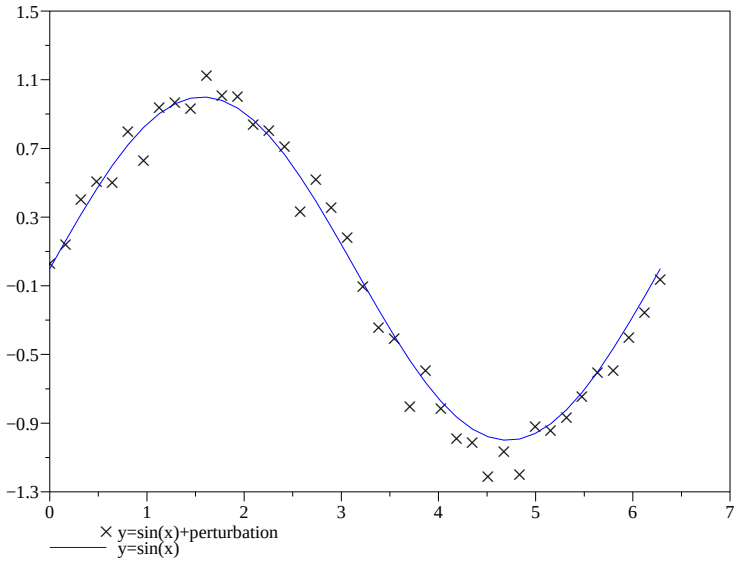
3. **Określanie skali** Oto przykład gdzie niezbędne jest narzucenie skali izometrycznej tutu bowiem chcemy narysować okrąg (patrz rysunek (6)). Dodatkowy parametr będzie postaci `frameflag=val`, gdzie `val` powinna przyjąć wartość 4 aby otrzymać skale izometryczną (dobraną na podstawie wartości minimalnej i naksymalnej)

```
t = linspace(0,2*%pi,60)';
x1 = 2*cos(t); y1 = sin(t);            // elipsa
x2 = cos(t);   y2 = y1;                // okr\k{a}g
x3 = linspace(-2,2,60)'; y3 = erf(x3); // funkcja b\l{}\k{e}du
legenda = "elipsa@okr\k{a}g@funkcja b\l{}\k{e}du"
plot2d([x1 x2 x3],[y1 y2 y3],style=[1 2 3], frameflag=4, leg=legenda)
xtitle("Znowu krzywe...","x","y")
```

W pewnych przypadkach `frameflag` towarzyszyć powinien parametr `rect`. Jeśli na przykład chcemy samodzielnie określić obszar rysowania (zwalniając z tego punkcję `plot2d` tutu) musimy napisać $rect = [x_{min}, y_{min}, x_{max}, y_{max}]$ w połączeniu z `frameflag=1` jak w przykładzie poniżej

```
x = linspace(-5,5,200)';
y = 1 ./(1 + x.^2);
xbasc()
plot2d(x, y, frameflag=1, rect=[-6,0,6,1.1])
xtitle("Funkcja Rungego tutu")
```

Oto wszystkie możliwe wartości dla `frameflag`



symbol

Rysunek 5: dessin en trait plain et avec des symboles non reliés tutu

`frameflag=0`	użycie wcześniej zdefiniowanej skali (lub domyślnej)
`frameflag=1`	skala zadana przez kwadrat
`frameflag=2`	skala obliczona jako maksimum i minimum z `Mx` i `My`
`frameflag=3`	échelle isométrique calculée en fonction de rect tutu
`frameflag=4`	skala izometryczna obliczona jako maksimum i minimum z `Mx` et `My`
`frameflag=5`	identycznie 1 ale z ewntualnym dostosowaniem graduation tutu
`frameflag=6`	identycznie 2 mais avec adaptation eventuelle pour la graduation
`frameflag=7`	identycznie 1 ale wcześniejsze krzywe są odrysowywane
`frameflag=8`	identycznie 2 ale wcześniejsze krzywe są odrysowywane

Uwaga: W przypadkach 4 i 5 może zajść (ewentualna) modyfikacja obszaru rysowania w taki sposób aby stopniowanie (tutu) (które jest zawsze tutu) tutu (tak zwane tutu).

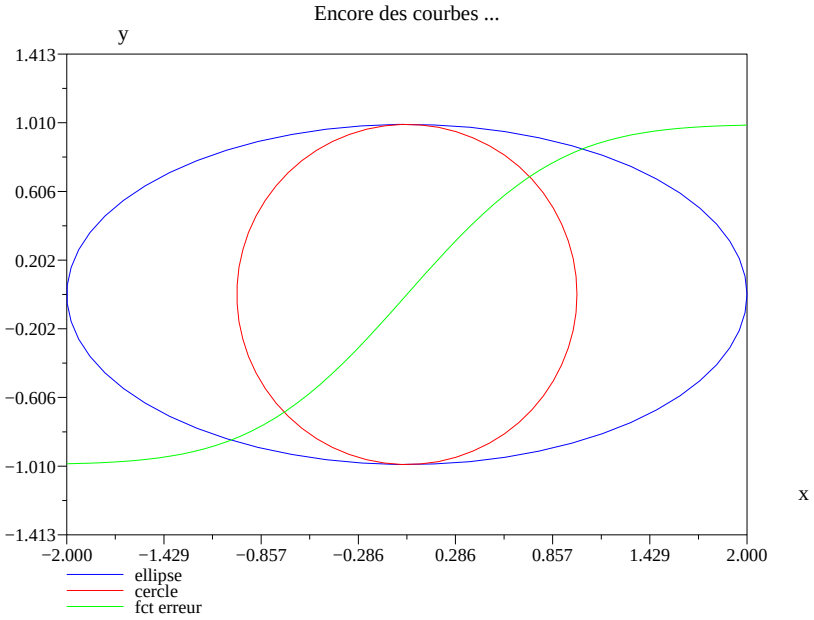
4. **Precyzowanie umieszczenis osi** Rozmieszczenie osi możemy określać pisząc `axesflag=val`. W kolenym przykładzie (patrz rysunek (7)) `val` ma wartość 5 co powoduje, że osie przetna się wpunkcie $(0,0)$[19] bez tutu boite englobante :

```
x = linspace(-14,14,300)';
y = sinc(x);
xbasc()
plot2d(x, y, style=2, axesflag=5)
xtitle("Funkcja sinc")
```

Oto tabela podająca wszytkie możliwości:

`axesflag=0`	bez pudełka, osi oraz jednostek
`axesflag=1`	z pudełkiem, osiami i jednostkami (x na dole, y po lewej)
`axesflag=2`	z pudelkiem, ale bez osi i jednostek
`axesflag=3`	z pudełkiem, osiami i jednostkami (x na dole, y po lewej)
`axesflag=4`	bez pudełka, ale z osiami i jednostkami (punkt przecięcia w środku)
`axesflag=5`	bez pudełka, ale z osiami i jednostkami (punkt przecięcia dla $x = 0$ i $y = 0$)

[19]Gdy punkt $(0,0)$ jest we wnętrzu obszaru rysowania.



Rysunek 6: Elipsa, okrąg i funkcja błędu

5. **Skala logarytmiczna** Odpowiedni parametr jest postaci `logflag=str`, gdzie `str` jest łańcuchem złożonym z dówch znaków – pierwszy dotyczy osi OX, drugi OY i każdy przyjmuje wartość `n` (nie logarytmiczna) lub `l` (logarytmiczna).

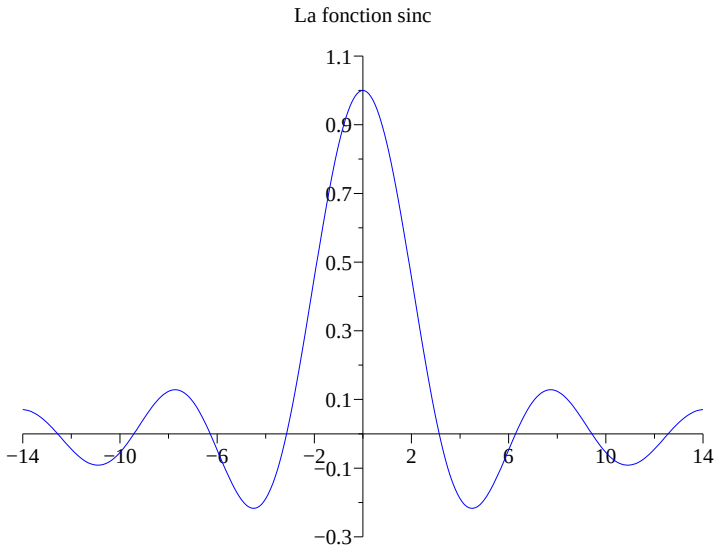
```
x = logspace(0,4,200)';
y = 1 ./x;
xbasc()
subplot(1,2,1)
plot2d(x, y, style=2, logflag= "ln")
xtitle("logflag=""ln""")
subplot(1,2,2)
plot2d(x, y, style=2, logflag= "ll")
xtitle("logflag=""ll""")
```

Przykład ten pokazuje także jak przy pomocy funkcji `subplot(m,n,num)` w jednym oknie umieścić kilka (niezależnych) wykresów. Parametr `m` informuje o podziale okna w pionie na `m` równych części, `n` decyduje o podziale w poziomie `num` jest natomiast kolejnym numerem okna spośród $m \times n$ okien. Okna numerowane są od lewej do prawej i od góry do dołu poczynając od numeru 1. Stąd okno na pozycji (i, j) ma numer[20] $n \times (i-1) + j$. Nic nie stoi na przeszkodzie aby modyfikować siatkę tutu celem dobranie najbardziej nam odpowiadającego układu wykresów. Na przykład

```
xbasc()
subplot(1,2,1)
titlepage("po lewej")
subplot(3,2,2)
titlepage(["po prawej";"powy\.zej"])
subplot(3,2,4)
titlepage(["po prawej";"w centrum"])
subplot(3,2,6)
titlepage(["po prawej";"poni\.zej"])
xselect()
```

tutu okna w pionie na dwie części (lewą i prawą), okna po prawej podzielono w poziomie na trzy częsci. Funkcję `subplot` można rozumieć jako dyrektywę pozwalającą wybrać pewien podobszar okna graficznego.

[20] A nie $m \times (i-1) + j$ jak napisane jest w pomocy!



esflag

Rysunek 7: Umieszczenie osi otrzymane dla `axesflag=5`

6. **Słowo kluczowe** `strf` Pozwala ono zastąpić zarazem `frameflag` i `axesflag` oraz, ze względu na kompatybilność z wcześniejszymi wersjami `plot2d` zawiera także flagę określającą czy ma ono zastosowanie do legendy, czy nie tutu. Podawana wartość składa się z trzech znaków `xyz`, gdzie

   x równe 0 (nie ma legendy) lub 1 (legenda tworzona jest przez podanie *leg=val*) ;

   y cyfra od 0 do 9, odpowiadająca wartości podawanej w *frameflag* ;

   z cyfra od 0 do 5, odpowiadająca wartości podawanej w *axesflag*.

   En fait il faut savoir l'utiliser car les séquences optionnelles de nombreuses primitives de dessin ne disposent pas encore des mots clés `frameflag` i `axesflag` tutu. Dodatkowo jest to bardzo praktyczne jeśli chcemy dodać nowy wykres do już istniejącego bez zmieniania skali i ramki co można osiągnać pisząc `strf="000"` (unikając w ten sposób pisania `frameflag=0, axesflag=0`).

4.4 Inne wersje `plot2d`: `plot2d2`, `plot2d3`

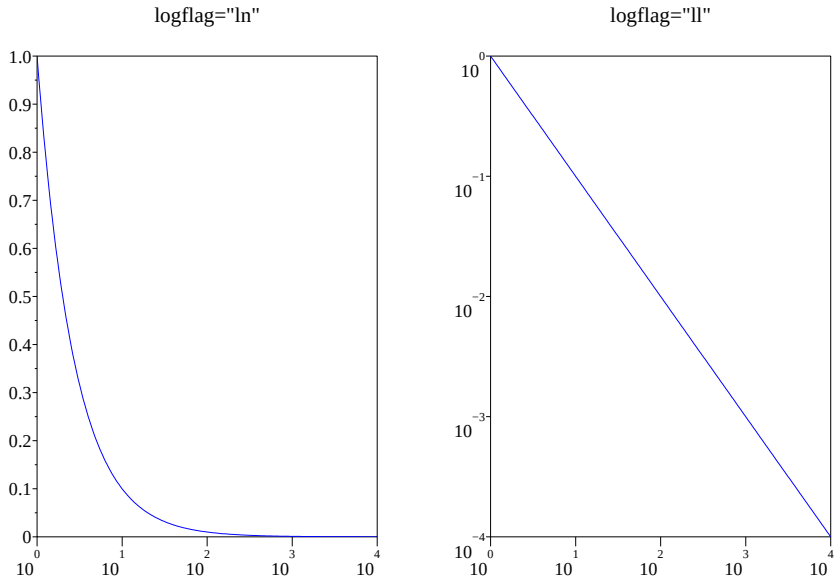
Zasadniczo używa się ich jak `plot2d` taka sama składnia, takie same argumenty opcjonalne.

1. **plot2d2** tutu pozwala narysować funkcję w oparciu o skalary: w miejsce wykresu prostego odcinka wprowadzamy punkty (x_i, y_i) et (x_{i+1}, y_{i+1}), `plot2d2` odcinek poziomy (od (x_i, y_i) do (x_{i+1}, y_i)) a następnie odcinek pionowy (od (x_{i+1}, y_i) do (x_{i+1}, y_{i+1})). Oto przykład (porównaj rysunek (9)):

   ```
   n = 10;
   x = (0:n)';
   y = x;
   xbasc()
   plot2d2(x,y, style=2, frameflag=5, rect=[0,-1,n+1,n+1])
   xtitle("plot2d2")
   ```

2. **plot2d3** Rysuje diagram tutu batonow: dla każdego punktu (x_i, y_i) `plot2d3` rysuje jeden segment pionowy od $(x_i, 0)$ do (x_i, y_i) (porównaj rysunek (10)) :

   ```
   n = 6;
   x = (0:n)';
   y = binomial(0.5,n)';
   ```



ogflag

Rysunek 8: Wykorzystanie parametru `logflag`

```
xbasc()
plot2d3(x,y, style=2, frameflag=5, rect=[-1,0,n+1,0.32])
xtitle("Prawdopodobienstwo dla prawa binomialnego tutu B(6,1/2)")
```

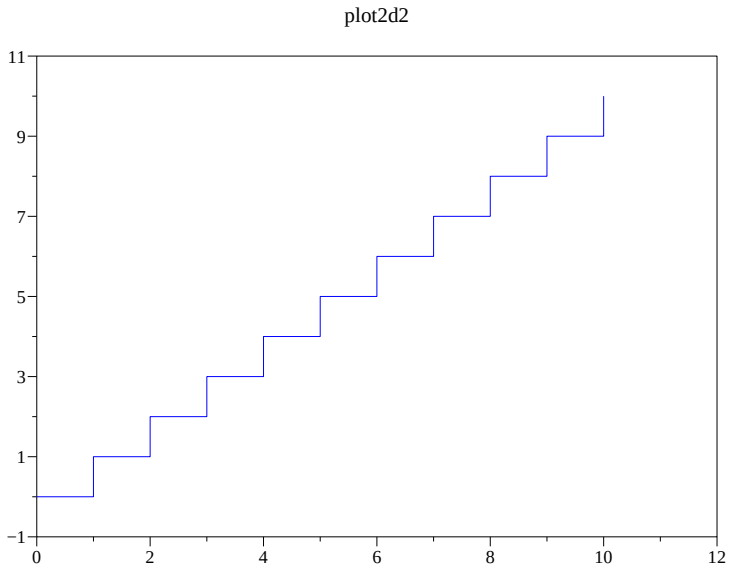
4.5 Rysowanie większej ilości krzywych złożonych z różnej ilości punktów

Za pomocą `plot2d` i jego wariantów nie można narysować za jednym razem większej ilości krzywych, które nie są dyskretyzowane na takiej samej ilości przedziałów (i chyba takich samych przedziałów tutu) co pociąga za soba konieczność kilkakrotnego użycia tej funkcji. Od wersji 2.6 można obyć się bez podawania skali bez obawy o przykre niespodzianki, ponieważ domyślnie (`frameflag=8`) wcześniejsze krzywe są odrysowywane w przypadku zmiany skali. Dlatego jeśli ktoś chce opanować skalę (chodzi tutaj chyba o to aby nie byla ona zmieniana automatycznie tutu), musi to zrobic przy pierwszym wywołaniu a dla następnych używać `frameflag=0`[21]. Oto odpowiedni przykład (porównaj rysunek (11))

```
x1 = linspace(0,1,61)';
x2 = linspace(0,1,31)';
x3 = linspace(0.1,0.9,12)';
y1 = x1.*(1-x1).*cos(2*%pi*x1);
y2 = x2.*(1-x2);
y3 = x3.*(1-x3) + 0.1*(rand(x3)-0.5);  // identyczne jak y2, ale z zaburzeniem
ymin = min([y1 ; y2 ; y3]); ymax = max([y1 ; y2 ; y3]);
dy = (ymax - ymin)*0.05;          // aby doda\'c margines tutu
rect = [0,ymin - dy,1,ymax+dy];   // okno wykresu
xbasc()                           // wyczyszczenie poprzednich wykres\'ow
plot2d(x1, y1, style=1, frameflag=5, rect=rect) // 1-sze wywo\l{}anie, kt\'ore ustali skal\k
plot2d(x2, y2, style=2, frameflag=0)            // 2-ie i 3-ie wywo\l{}anie;
plot2d(x3, y3, style=-1,frameflag=0)            // u\.zywamy poprzedniej skali
xtitle("Krzywe...","x","y")
```

Uwaga: Wyprúj ten przykład pisząc `frameflag=1` w miejsce `frameflag=5`. Nie można mieć osobnej legendy dla każdej krzywej niemniej jest to możliwe do osiągnięcia przy pomocy niewielkiej ilości progamowania.

[21] Metoda ta jest konieczna jeśli używamy skali iso-metrycznej.



lot2d2

Rysunek 9: Wykorzystanie `plot2d2`

4.6 Zabawa z kontekstem graficznym

moze lepiej „praca”? :))) Jeśli wyprobowaliśmy poprzednie przykłady bez wątpienia i z ochotą będziemy chcieli modyfikować niekóre rzeczy jak rozmiar symboli, rozmiar czy styl użtego fontu lub też grubość strzałek.

1. **Fonty :** Aby zmienić font należy użyć poniższego wywołania

   ```
   xset("font",font_id,fontsize_id)
   ```

   gdzie `font_id` i `fontsize_id` są liczbami całkowitymi odpowiadającymi odpowiednio z rodzaj i wielkość wybranego fontu. Bieżący font można otrzymać pisząc

   ```
   f=xget("font")
   ```

   gdzie `f` jest wektorem, `f(1)` opisuje rodzaj fontu a `f(2)` jego wielkość. Można prostozmieniać/uzyskać wielkość za pomocą `xset("font size",size_id)` i `fontsize_id = xget("font size")`. Oto te, które są teraz dostępne

Nazwa fontu	Courier	Symbol	Times	Times-Italic	Times-Bold	Times-Bold-Italic
Identyfikator	0	1	2	3	4	5

Wielkość	8 pts	10 pts	12 pts	14 pts	18 pts	24 pts
Identyfikator	0	1	2	3	4	5

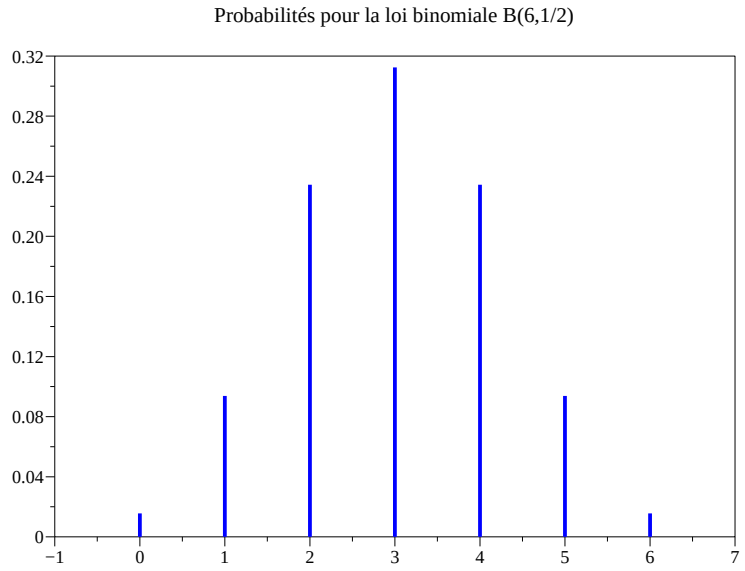
   Uwagi:

   - Courier est ŕ chasse fixe tutu;
   - font Symbol pozwala na użycie liter greckich (p odpowiada π, a – α, etc...);
   - Times jest fontem domyślnym a jego domyślny rozmiar to 10 punktów.

2. **rozmiar symboli:** zmieniamy poleceniem

   ```
   xset("mark size",marksize_id)
   ```

   bieżący otrzymujemy natomiast pisząc



Rysunek 10: Wykorzystanie plot2d3

lot2d3

```
marksize_id = xget("mark size")
```

gdzie podobnie jak dla fontów rozmiar określamy za pomoca cyfr od 0 do 5; 0 jest wielkością domyślną.

3. **grubość linii:** zmieniamy / otrzymujemy pisząc

```
xset("thickness",thickness_id)
thickness_id = xget("thickness")
```

Grubość jest wielkością dodatnią odpowiadającą liczbie pikseli (na grubość) jaką ma mieć krzywa. Dokonując zmiany grubości kreślonych linii wpływamy także, czy tego chcemy czy nie, na grubość kreślonej ramki i skali na osiach. Wyjściem w takiej sytuacji jest dwukrotne tworzenie rysunku. Za pierwszym razem pomijamy ramkę i skalę (`axesflag=0`). Następnie powracamy do normalnej grubości i nie zmieniając skali widocznego obszaru (`frameflag=0`) rysujemy coś poza nim (na przykład krzywą zredukowana do jednego punktu $(-\infty, -\infty)$):

```
xset("thickness", 3)
plot2d(x, y, axesflag=0, ...)
xset("thickness", 1)
plot2d(-%inf, -%inf, frameflag=0, ...)
```

Zatem drugie wywołanie służy jedynie do narysowania ramki i skali.

4.7 Tworzenie histogramów

Odpowiednia do tego celu funkcja scilaba nazywa się `histplot` a jej wywołanie jest następujące

```
histplot(n, X, <,opt_arg>*)
```

gdzie

- `n` jest albo liczbą całkowitą albo wektorem liniowym (w którym $n_i < n_{i+1}$):
  1. w przypadku gdy `n` jest wektorem liniowym dane dzielone są na k klas $C_i = [n_i, n_{i+1}[$ (wektor `n` ma więc $k+1$ składowych);

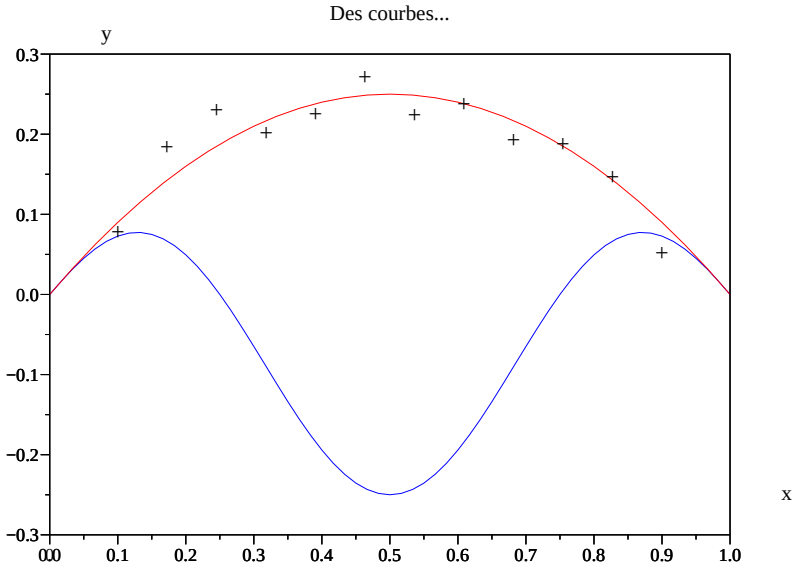


fig:3

Rysunek 11: Jeszcze krzywe...

2. w przypadku gdy `n` jest liczbą całkowitą, dane dzielone są na n równoodległych klas

$$C_1 = [c_1, c_2],\ C_i =]c_i, c_{i+1}],\ i = 2, ..., n, \text{ avec } \begin{cases} c_1 = min(X),\ c_{n+1} = max(X) \\ c_{i+1} = c_i + \Delta C \\ \Delta C = (c_{n+1} - c_1)/n \end{cases}$$

- `X` wektor (liniowy lub kolumnowy) z danymi;
- `<,opt_arg>*` ciąg opcjonalnych argumentów jak dla `plot2d` z dodatkową kombinacją *normalization = val* gdzie *val* jest stałą (zmienną lub wyrażeniem) boolowskim (domyślnie *true*). Kiedy histogram jest znormalizowany, tutu son intégrale vaut 1 et approche donc une densité (dans le cas contraire, la valeur d'une plage correspond au nombre de composantes de X tombant dans cette plage). Plus précisemment, la plage de l'histogramme correspondant ŕ l'intervalle C_i vaut donc (m étant le nombre de données, et $\Delta C_i = n_{i+1} - n_i$) :

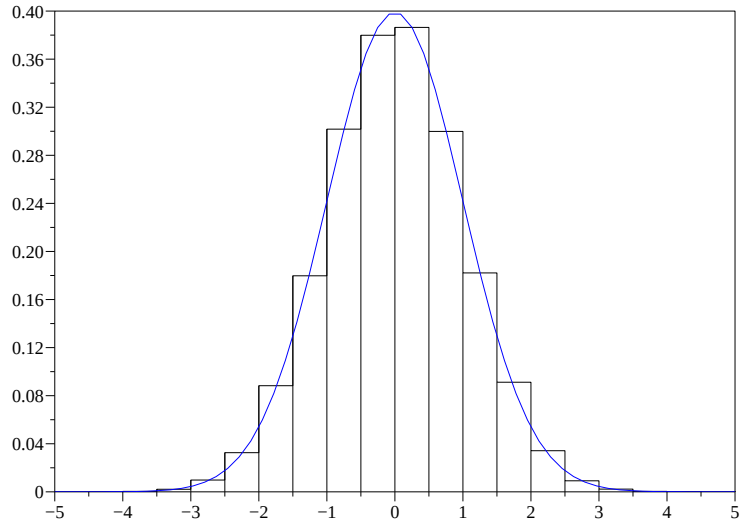
$$\begin{cases} \frac{\text{card } \{X_j \in C_i\}}{m \Delta C_i} & \text{si } normalization = vrai \\ \text{card } \{X_j \in C_i\} & \text{si } normalization = faux \end{cases}$$

Poniżej przedstawiamy mały przykład, tutu toujours avec la loi normale (patrz rysunek (12)) :

```
X = rand(100000,1,"normal"); classes = linspace(-5,5,21);
histplot(classes,X)
// tutu on lui superpose le tracé de la densité de N(0,1)
x = linspace(-5,5,60)'; y = exp(-x.^2/2)/sqrt(2*%pi);
plot2d(x,y, style=2, frameflag=0, axesflag=0)
```

4.8 Zapisywanie grafiki w różnych formatach

W celu zachowanie utworzonego rysunku wybieramy z menu `File` okna graficznego opcję `Export` a następnie wybieramy pożądany format. tutu - sprawdzić to Większość z nich opiera się na postscripcie... . Począwszy od wersji 2.5 mamy także możliwość eksportu grafiki do pliku typu `gif`.



`histo`

Rysunek 12: Histogram próbki liczb losowych wybranych z rozkładem $N(0,1)$ tutu

4.9 Prosta animacja

Realizacja animacji przy pomocy Scilaba jest dość prosta, jako że pozwala on na podwójne buforowanie dzięki czemu unikamy efektu migotania, gdyż wykres tworzony jest „w ukryciu” i dopiero potem pokazywany. Mamy do wyboru dwa „drivery” mające wpływ na tworzenie wykresu na ekranie[22]

- `Rec`, który powoduje, że wszystkie operacje graficzne związane są z oknem; jest to domyślny drvier;
- `X11`, tutu qui se contente simplement d’afficher les graphiques (il est alors imposible de „zoomer”).

Do celów tworzenia animacji najczęsciej odpowiedni będzie ten ostatni; wybieramy go pisząc `driver("X11")` (w celu powrócenia do drivera domyślnego piszemy `driver(Żec")`).

Przy podwójnym buforowaniu każdy kolejny obraz tworzony jest najpierw w pamięci (powstaje wówczas tak zwana pixmapa) a dopiero później przenoszony na ekran. Oto prosty schemat postępowania przy tworzeniu animacji

```
driver("X11")      // tutu pas d’enregistrement des opérations graphiques
xset("pixmap",1)   // przej\’scie do trybu podw\’ojnego buforowania
...........        // ewentualne instrukcje ustalaj\k{a}ce skal\k{e}
for i=1:nb_dessins
  xset("wwpc")    // wyczyszczenie pixmapy
  .......
  .......         // tworzenie i-tego rysunku
  .......
  xset("wshow")   // przeniesienie rysunku na ekran
end
xset("pixmap",0)   // powr\’ot do bezpo\’sredniego tworzenia rysunk\’ow na ekranie
driver("Rec")      // przej\’scie do domy\’slego drivera
```

Uwaga: W powyższym postępowaniu czyszczenie całego obszaru rysowania przy pomocy `xset("wwpc")` nie czynnością wymaganą. Zamiast tego można użyć na przykład funkcji `xclea`, co uchroni nas przed każdorazowym odrysowywaniem tych obszarów rysunku, które się nie zmieniają. Aby tutu użyć technik maskowania należy za pomocą wywołanie `xset('alufunction',nu` zmienić funkcję sterującą wyświetlaniem (gdzie `num` to liczba całkowita związana z wybraną funkcją); patrz Help i przykłady.

Poniżej przedstawiony zostanie przykład animacji: poruszający się środek ciężkości prostokąta (o długości L i szerokości l) po okręgu o promieniu r i środku w punkcie $(0,0)$; prostokąt dodatkowo obraca się wokół swojego środka ciężkości. tutu Il y a certain un nombre de détails qui se greffent autour du canevas général exposé ci-avant :

- l’ordre `plot2d` sert uniquement ŕ régler l’échelle (isométrique) pour les dessins ;

[22] Oprócz „driverów” pozwalających tworzyć rysunki jako `postscript`, `fig` czy `gif`.

- `xset("background",1)` impose la couleur 1 (du noir dans la carte des couleurs par défaut) comme couleur d'arričre plan, mais il est recommandé d'exécuter l'instruction `xbasr()` pour mettre effectivement ŕ jour la couleur du fond ;
- le dessin consiste ŕ appeler la fonction `xfpoly` suivi de `xpoly` pour dessiner le bord (avec ici 3 pixels ce qui est obtenu avec `xset("thickness",3)`) ; ŕ chaque fois on change la couleur ŕ utiliser avec `xset(ćolor",num)` ;
- l'instruction `xset("default")` repositionne le contexte graphique de la fenętre ŕ des valeurs par défaut ; ainsi la variable `pixmap` reprend la valeur 0, `thickness` la valeur 1, `background` sa valeur par défaut, etc...

```
n = 4000;
L = 0.6; l = 0.3; r = 0.7;
nb_tours = 4;
t = linspace(0,nb_tours*2*%pi,n)';
xg = r*cos(t); yg = r*sin(t);
xy = [-L/2  L/2 L/2 -L/2;...   // 4 punkty graniczne
      -l/2 -l/2 l/2  l/2];

xselect()
driver("X11")
xset("pixmap",1)
plot2d(\%inf,\%inf, frameflag=3, rect=[-1,-1,1,1], axesflag=0)
xset("background",1); // czarne t\l{}o
xbasr()               // tutu utile pour la mise a jour du background
xset("thickness",3)   // zwi\k{e}kszenie grubo\'sci kre\'slonych linii (3 pixele)

xset("font",2,4)
for i=1:n
   xset("wwpc")
   theta = 3*t(i);
   xyr = [cos(theta) -sin(theta);...
          sin(theta)  cos(theta)]*xy;
   xset("color",2)
   xfpoly(xyr(1,:)+xg(i), xyr(2,:)+yg(i))
   xset("color",5)
   xpoly(xyr(1,:)+xg(i), xyr(2,:)+yg(i),"lines",1)
   xset("color",32)
   xtitle("Animation simple")
   xset("wshow")   // przeniesienie pixmapy na ekran
end
driver("Rec")      // powrot do domy\'slnego drivera
xset("default")    // przywr\'ocenie domy\'slnego kontekstu graficznego
```

4.10 Powierzchnie

Podstawową funkcją pozwalającą na tworzenie wykresów powierzchni jest `plot3d`[23]. Przy preprezentacji powierzchni za pomoca facettes tutu mamy możliwość określenia innego koloru dla każdej z nich. Od wersji 2.6 można także, zarówno dla fscettes trójkątnych jak i kwadratowych określić kolor dla każdego z wierzchołków, przez co rendu tutu otrzymywane jest przez interpolację kolorów określonych w wierzchołkach.

4.10.1 Wprowadzenie do `plot3d`

Gdy powierzchnia opisane jest przez równanie typu $z = f(x, y)$, szczególnie łatwo można ją narysować jeśli argumenty są z obszaru prostokątnego. Jako przykład rozważmy funkcję $f(x, y) = cos(x)cos(y)$ dla $(x, y) \in [0, 2\pi] \times [0, 2\pi]$:

```
x=linspace(0,2*\%pi,31); // dyskretyzacja zmiennej x (a tak\.ze y : b\k{e}dzie to samo)
z=cos(x)'*cos(x);       // zbi\'or warto\'sci zmiennej z : macierz z(i,j) = f(x(i),y(j))
plot3d(x,x,z)           // rysunek
```

W efekcie otrzymamy coś podobnego do rysunku (13)[24]. W najbardziej ogólnej formie funkcji `plot3d` lub `plot3d1` używamy pisząc

[23] `plot3d1` używana analogicznie pozwala uzależnić wartość koloru od wartości przyjmowanej na osi OZ.

[24] Dokładnie rzecz biorąc rysunek ten przedstawia efekt użycia funkcji `plot3d1` gdzie na potrzeby publikacji zamiast kolorów użyto odcieni szarości a także ustawiono trochę inny punkt widzenia.

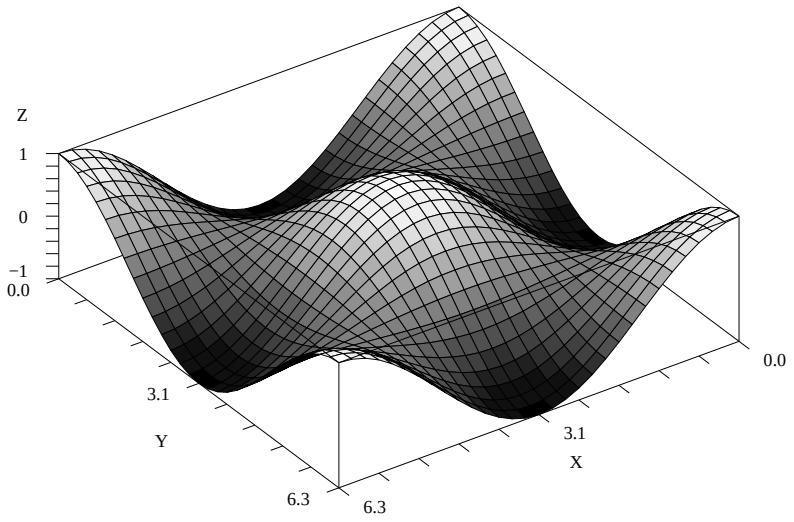


fig:4

Rysunek 13: Funkcja $z = cos(x)cos(y)$

```
plot3d(x,y,z <,opt_arg>*)
plot3d1(x,y,z <,opt_arg>*)
```

gdzie dla `plot2d`, `<,opt_arg>*` ozancza ciąg argumentów opcjonalnych, `opt_arg` przyjmuje formę *słowo_kluczowe=wartość*. W najprostszym przypadku, `x` i `y` są wektorami liniowymi ($(1, nx)$ i $(1, ny)$) odpowiadającymi dyskretyzacji zmiennej x oraz y, natomiast `z` jest macierzą (nx, ny) taką, że $z_{i,j}$ jest „wysokością" w punkcie (x_i, y_j).

Opcjonalne argumenty to:

1. `theta`=*val_theta* i `alpha`=*val_alpha* to dwa kąty (w stopniach) określające punkt widzenia we współrzędnych sferycznych (jeśli O jest środkiem pudełka englobante tutu, Oc kierunkiem patrzenia kamery, wówczas $\alpha = kt(Oz, Oc)$ i $\theta = kt(0x, Oc')$ gdzie Oc' jest rzutem Oc na płaszczyznę Oxy);

2. `leg`=*val_leg* pozwala określić nazwę dla każej z osi (na przykład `leg="x@y@z"`), argument efektywny *val_leg* jest łańcuchem znakowym, w którym `@` stanowi separator pomiędzy nazwami;

3. `flag`=*val_flag* gdzie *val_flag* jest wektorem o trzech składowych *[mode type box]* pozwalającym określić:

   (a) parametr $mode$ związany jest z rysunkiem faces tutu i siatki:

      i. dla $mode > 0$, faces niewidoczne są usuwane[25], siatka pozostaje widoczna;

      ii. dla $mode = 0$, otrzymujemy tutu un rendu (*fr. fil de fer, ang. wireframe*) de la surface;

      iii. dla $mode < 0$, faces niewidoczne są usuwane a siatka nie jest rysowana.

   Dodatnio określona strona face tutu (patrz dalej) będzie malowana z wykorzystaniem koloru numer $mode$ zaś strona przeciwna przy użyciu koloru, który możemy określić instrukcją `xset("hidden3d",coloris)` (domyślnie jest to 4 kolor z palety).

   (b) parametr $type$ pozwala określić skalę:

$type$	otrzymana skala
0	użycie poprzedniej skali (tutu ou par défaut)
1	skala wraz z `ebox`
2	skala otrzymana w oparciu o minimum i maximum zmiennych tutu
3	jak 1 ale skala jest izometryczna
4	jak 2 ale skala jest izometryczna
5	variante de 3
6	variante de 4

[25] tutu actuellement c'est l'algorithme du peintre qui est utilisé, c-a-d qu'un tri des facettes est effectué et les plus éloignées de l'observateur sont dessinées en premier.

(c) parametr box kontroluje tutu le pourtour du graphe :

box	otrzymany efekt
0	juste le dessin de la surface
2	osie pod powierzchnia są rysowane
3	jak dla 2 z dodatkowym tutu la boite englobante
4	jak dla 3 z dodatkowym tutu la graduation des axes

4. `ebox=`*val_ebox* pozwala określić tutu la boite englobante, *val_ebox* jest wektorem o 6 składowych $[x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{m}$

Oto mały przykład, w którym używa się prawie wszystkich parametrów `plot3d`. Jest to prosta animacja pozwalająca lepiej zrozumieć zmiane punktu widzenia przy pomocy parametrów `theta` i `alpha`. W skrypcie tym używamy `flag=[2 4 4]`, co oznacza

- $mode = 2$ powierzchnia będzie rysowana (jej dodatno określona strona) kolorem 2 oraz będzie widoczna siatka;
- $type = 4$ zostanie użyta skala izometryczna obliczona na podstawie danych (jest to równoważne wyborowi $type = 3$ z parametrem $ebox$ przyjmującym wartości równe minimum i maksimum danych);
- $box = 4$ rysunek będzie zawierał pudełko tutu boite oraz stopniowanie et des graduations.

```
x=linspace(-%pi,%pi,31);
z=sin(x)'*sin(x);
n = 200;
theta = linspace(30,390,n);  // pe\l{}ny obr\'ot
alpha = [linspace(60,0,n/2) linspace(0,80,n/2)]; // od g\'ory
                                                 // do do\l{}u
xselect()
xset("pixmap",1)   // aktywowanie podw\'ojnego buforowania
driver("X11")
// zmieniamy parametr theta
for i=1:n
   xset("wwpc") // wyczyszczenie bie\.z\k{a}cego bufora
   plot3d(x,x,z,theta=theta(i),alpha=alpha(1),leg="x@y@z",flag=[2 4 4])
   xtitle("zmiany punktu widzenia przy pomocy parametru theta")
   xset("wshow")
end
// zmieniamy parametr alpha
for i=1:n
   xset("wwpc") // wyczyszczenie bie\.z\k{a}cego bufora
   plot3d(x,x,z,theta=theta(n),alpha=alpha(i),leg="x@y@z",flag=[2 4 4])
   xtitle("zmiany punktu widzenia przy pomocy parametru alpha")
   xset("wshow")
end
xset("pixmap",0)
driver("Rec")
```

4.10.2 Kolory

Powróćmy jeszcze na chwilkę do ostatniego przykładu i zamiast `plot3d` napiszmy `plot3d1` uzależniając tym samym kolory od wartości zmiennej z. Narysowana powierzchnia powinna przypominać w tym momencie mozaike gdyż wykorzystywana plaeta kolorów domyślnie nie jest „ciągła”.

Paleta kolorów jest macierzą o wymiarze `(nb_couleurs,3)`, gdzie i-ta linia definiuje intensywności składowej czerwonej (wartość zawarta w przedziale od 0 do 1), zielonej i niebieskiej dla i-tego koloru. Mając taką macierz, którą nazwiemy `C`, polecenie `xset(ćolormap",C)` pozwala ją wczytać (załadować) do kontekstu graficznego bieżącego okna graficznego. Funkcje, `hotcolormap` oraz `greycolormap` dostarczają mapy ze stopniową zmianą kolorów[26]. Mała uwaga: jeśli dokonujemy zmiany palety kolorów po narysowaniu jakiegoś rysunku, zmiany nie będą widoczne natychmiast (jest to zachowania normalne); wystarczy zmienić rozmiar okna lub tutu d'envoyer l'ordre `xbasr(numero_fenetre)`. Oto nowy przykład

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

[26]Patrz także sekcja Contributions na stronie Scilab.

```
C = hotcolormap(32); // hot colormap z 32 kolorami
xset("colormap",C)
xset("hidden3d",30)  // wyb\'or koloru 30 do rysowania po ujemnie okre\'slonej stronie
xbasc()
plot3d1(x,x,z, flag=[1 4 4])  // spr\'obuj tak\.ze z flag=[-1 4 4]
```

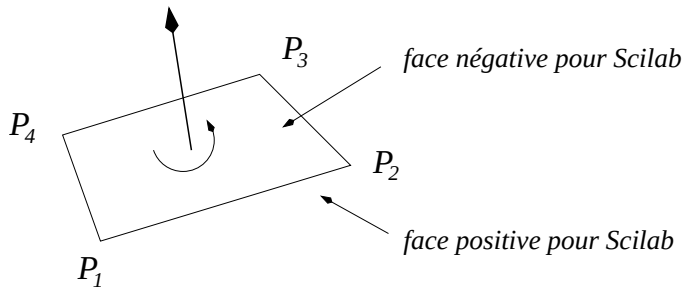
Uwaga : w `plot3d1`, wykorzystuje się jedynie znak parametru $mode$ (jeśli $mode \geq 0$ siatka zostanie narysowana, nie zostanie zaś gdy $mode < 0$).

4.10.3 `plot3d` z des facettes

Chcąc użyć tej funkcji w jak najogólniejszej postaci należy podać opis powierzchni uwzględniając pojedyńczy tutu facettes. Określany jest on przy pomocy 3 macierzy `xf, yf, zf` o wymiarze (nb_sommets_par_face, nb_faces), gdzie `xf(j,i),yf(j,i),zf(j,i)` są współrzędnymi j-tego wierzchołka i-tego tutu facette. Poza tymi zmianami dalsze użycie jest takie samo jak w poprzednich przykładach:

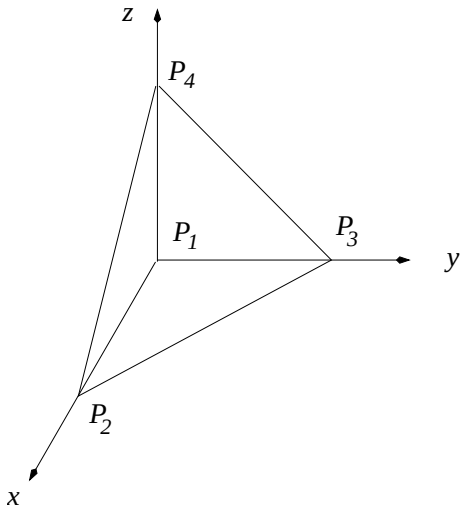
`plot3d(xf,yf,zf <,opt_arg>*)`

Orientacja tutu facettes jest odmienna od zwyczajowo przyjętej (patrz rysunek (14).



Rysunek 14: orientacja tut facettes w Scilab-ie

W celu zdefinowania kolorów dla każdego facette, trzeci argument musi być listą : `list(zf,colors)` gdzie `colors` jest wektorem o rozmiarze `nb_faces`, `colors(i)` określan numer (w palecie) koloru i-tego tutu facette.



Rysunek 15: Trójścian

Jak w pierwszym przykładzie, narysujemy ściany trójścianu z rysunku (15), dla którego:

$$P_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix},\ P_2 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix},\ P_3 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix},\ P_4 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix},$$

gdzie definicja ścian jest następujaca (w ten sposób otrzymujemy ściany zewnętrzne z orientacją dodatnią dla Scilab-a):

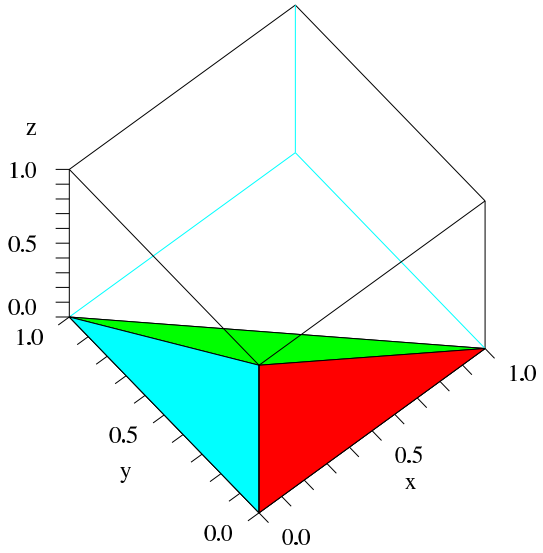
$$f_1 = (P_1, P_2, P_3),\ f_2 = (P_2, P_4, P_3),\ f_3 = (P_1, P_3, P_4),\ f_4 = (P_1, P_4, P_2)$$

Napiszmy więc:

```
//    f1 f2 f3 f4
xf = [ 0  1  0  0;
       1  0  0  0;
       0  0  0  1];
yf = [ 0  0  0  0;
       0  0  1  0;
       1  1  0  0];
zf = [ 0  0  0  0;
       0  1  0  1;
       0  0  1  0];  // tutu ouf !

xbasc()
plot3d(xf,yf,list(zf,2:5), flag=[1 4 4], leg="x@y@z",alpha=30, theta=230)
xselect()
```

Otrzymany efekt powinien przypominać rysunek 16. Można zauważyć, że `plot3d` używa prostej tutu projection orthographique et non une projection perspective plus réaliste.



Rysunek 16: Trójścian narysowany w Scilab-ie.

tetra

Na bieżące potrzeby, obliczenia tutu des facettes, mogą być efektywniejsze dzięki funkcjom:

- `eval3dp` i `nf3d` dla powierzchni zdefiniowanych przy pomocy $x = x(u, v), y = y(u, v), z = z(u, v)$ (patrz 4.10.4) ;
- `genfac3d` dla powierzchni definiowanych przy pomocy $z = f(x, y)$ (przykład pokazany jest trochę dalej (4.10.5)).

Jeśli powierzchnia (wielościan) zdefiniowana jest w taki sam sposób jak trójścian z przykładu nie można jego narysować bezpośrednio przy użyciu `plot3d`. W celu otrzymania opisu oczekiwanego przez Scilab-a można wykorzystać funkcję podobną do przedstawionej poniżej:

```
function [xf,yf,zf] = facettes_polyedre(P)
   // tutu transforme la structure Polyedre de l'exemple
   // sur les tlist en description de facettes pour
   // visualisation avec plot3d
   [ns, nf] = size(P.face)  // ns: ilo\'s\'c wierzcho\l{}k\'ow tworz\k{a}cych \'scian\k{e}
                            // nf: ilo\'s\'c \'scian
   xf=zeros(P.face); yf=zeros(P.face); zf=zeros(P.face)
```

```
   for j=1:ns
      num =  connect(ns+1-j,:) // dla odwr\'ocenia orientacji
      xf(j,:) = P.coord(1, num)
      yf(j,:) = P.coord(2, num)
      zf(j,:) = P.coord(3, num)
   end
endfunction
```

Mając tak określoną funkcję, rysunek wielościanu otrzymamy pisząc

```
   [xf,yf,zf] = facettes_polyedre(Cube);
   plot3d(xf,yf,list(zf,2:7), flag=[1 4 0],theta=50,alpha=60)
```

fparam

4.10.4 Rysowanie powierzchni opisanej przy pomocy $x = x(u, v)$, $y = y(u, v)$, $z = z(u, v)$

Odpowiedź: wziąć dyskretyzację dziedziny parametrów obliczyć tutu les facettes przy pomocy funkcji (`eval3dp`). Ze względów wydajnościowych, funkcja określająca parametry powierzchni powinna być napisana „wektorowo”. Jeśli $(u_1, u_2, \ldots, u_m)$ i $(v_1, v_2, \ldots, v_n)$ stanowią dyskretyzację dziedziny parametrów, funkcja powinna być wywoływana jednokrotnie z dwoma „dużymi” wektorami rozmiaru $m \times n$:

$$U = (\underbrace{u_1, u_2, \ldots, u_m}_{1}, \underbrace{u_1, u_2, \ldots, u_m}_{2}, \ldots\ldots, \underbrace{u_1, u_2, \ldots, u_m}_{n})$$
$$V = (\underbrace{v_1, v_1, \ldots, v_1}_{m \text{ fois } v_1}, \underbrace{v_2, v_2, \ldots, v_2}_{m \text{ fois } v_2}, \ \ldots\ \ldots\ , \underbrace{v_n, v_n, \ldots, v_n}_{m \text{ fois } v_n})$$

W oparciu o te dwa wektory, funkcja powinna tworzyć 3 wektory X, Y et Z wymiaru $m \times n$ takie, że:

$$X_k = x(U_k, V_k), Y_k = y(U_k, V_k), Z_k = z(U_k, V_k)$$

Oto kilka przykładów parametryzacji powierzchni, tutu écrite[27] de façon ŕ pouvoir ętre utilisée avec `eval3dp` :

```
function [x,y,z] = tore(theta, phi)
  // tutu paramétrisation classique d'un tore de rayons R et r et d'axe Oz
  R = 1; r = 0.2
  x = (R + r*cos(phi)).*cos(theta)
  y = (R + r*cos(phi)).*sin(theta)
  z = r*sin(phi)
endfunction

function [x,y,z] = helice_torique(theta, phi)
  // tutu paramétrisation d'une helice torique
  R = 1; r = 0.3
  x = (R + r*cos(phi)).*cos(theta)
  y = (R + r*cos(phi)).*sin(theta)
  z = r*sin(phi) + 0.5*theta
endfunction

function [x,y,z] = moebius(theta, rho)
  // wst\k{e}ga Moëbius-a
  R = 1;
  x = (R + rho.*sin(theta/2)).*cos(theta)
  y = (R + rho.*sin(theta/2)).*sin(theta)
  z = rho.*cos(theta/2)
endfunction

function [x,y,z] = tore_bossele(theta, phi)
  // tutu paramétrisation d'un tore dont le petit rayon r est variable avec theta
  R = 1; r = 0.2*(1+ 0.4*sin(8*theta))
  x = (R + r.*cos(phi)).*cos(theta)
```

[27] Piszemy je w naturalny sposób, pamiętając aby zamiast `*` i `/` napisać `.*` i `./` i wszystko powinno działać!

```
  y = (R + r.*cos(phi)).*sin(theta)
  z = r.*sin(phi)
endfunction
```

Oto przykład wykorzystujący ostatnią powierzchnię:

```
// skrypt rysuj\k{a}cy powierzchni\k{e} opisan\k{a} za pomoc\k{a} r\'ownania parametrycznego
theta = linspace(0, 2*%pi, 160);
phi = linspace(0, -2*%pi, 20);
[xf, yf, zf] = eval3dp(tore_bossele, theta, phi); // tutu calcul des facettes
xbasc()
plot3d1(xf,yf,zf)
xselect()
```

Jeśli chcemy użyć kolorów a nie otrzymujemy ich na rysunku, spowodowane jest to niewłaściwą orientacją; wystarczy odwrócić kierunek jednego z wektorów stanowiącego dyskretyzację dziedziny.

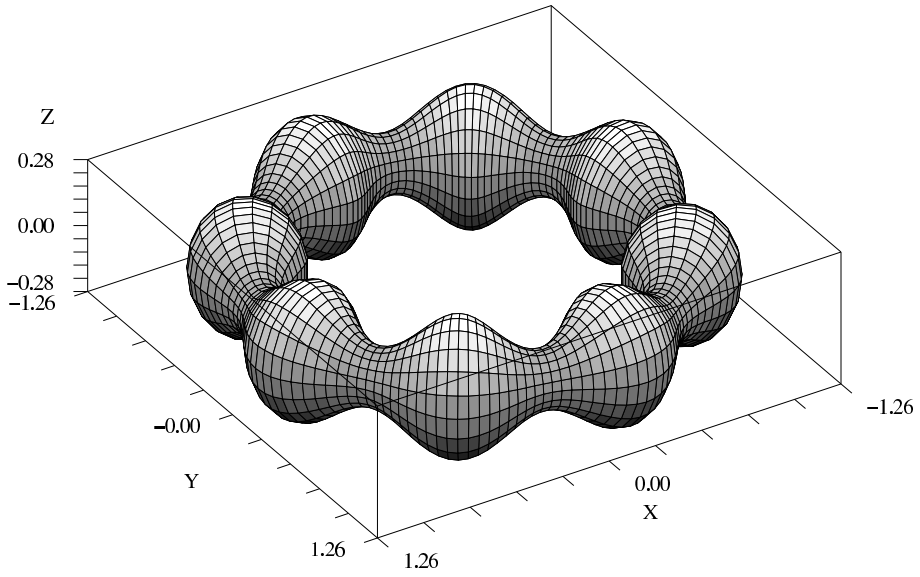


fig:6

Rysunek 17: Un tore bosselé... tutu

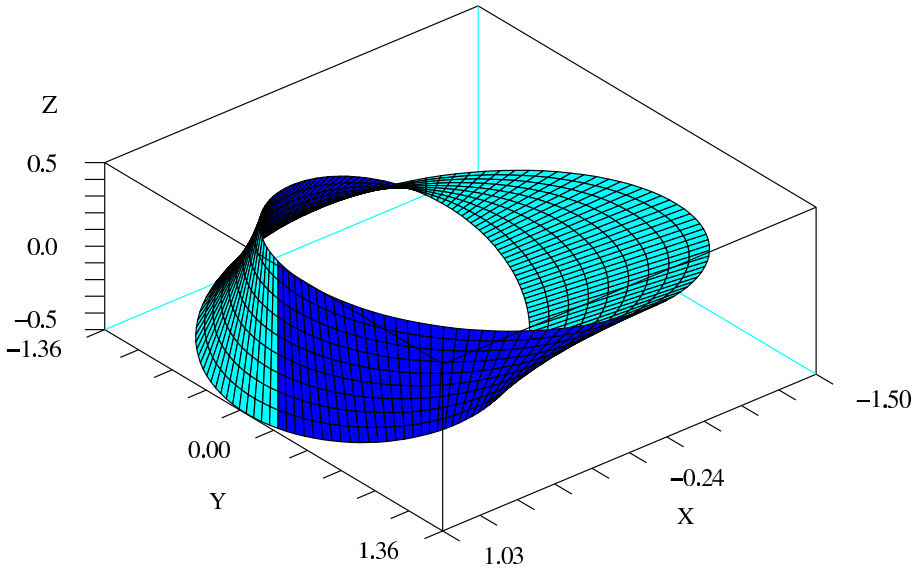
tutu Funkcja `nf3d` jest lekko podobna do `eval3dp`, ale mając dyskretyzację u i v trzeba zdefiniować soit-męme des matrices X, Y, Z taką, że:

$$X_{i,j} = x(u_i, v_j)$$
$$Y_{i,j} = y(u_i, v_j)$$
$$Z_{i,j} = z(u_i, v_j)$$

et vos facettes s'obtiennent alors avec `[xf,yf,zf] = nf3d(X,Y,Z)`. Jako przykład rozważmy wstęgę Moëbius-a définit juste avant :

```
nt = 120;
nr = 10;
rho = linspace(-0.5,0.5,nr);
theta = linspace(0,2*%pi,nt);
R = 1;
X = (R + rho'*sin(theta/2)).*(ones(nr,1)*cos(theta));
Y = (R + rho'*sin(theta/2)).*(ones(nr,1)*sin(theta));
Z = rho'*cos(theta/2);
[xf,yf,zf] = nf3d(X,Y,Z);
xbasc()
plot3d(xf,yf,zf, flag=[2 4 6], alpha=60, theta=50)
xselect()
```

Uwaga*: w celu otrzymania pawidłowej macierzy należało użyć funkcji* `ones`*, tutu ce qui ne rend pas le code trčs clair: funkcja* `eval3dp` *jest łatwiejsza w użyciu!*



Rysunek 18: Wstęga Moëbius-a

ebius

4.10.5 `plot3d` z interpolacją kolorów

pcolor

Od wersji 2.6 , możliwe jest określenie koloru dla każdego wierzchołka tutu d'une facette. Wystarczy okteślić macierz `colors` takiego samego wymiaru jak `xf, yf, zf` dającą opis każdego facette, to znaczy taką, że `colors(i,j)` jest kolorem związanym z i-tym wierzchołkiem j-tego face, i połączyć z trzecim argumentem (`zf`) w listę:

```
plot3d(xf,yf,list(zf,colors) <,opt_arg>*)
```

Oto przykład początkowy `plot3d` bez rysowania siatki:

- tutu une couleur par face pour le dessin de gauche,
- tutu une couleur par sommet pour celui de droite.

Aby obliczyć wartosci kolorów, wykorzystano pomcniczą funkcję wiążącą w sposób liniowy wartości na karcie graficznej tutu!!!. (użyto funkcji `dsearch` dostępnej od wersji 2.7 ale można łatwo tutu vous en passer). Zauważyć także będzie można wykorzytanie funkcji `genfac3d` pozwalającej na obliczenie tutu les facettes.

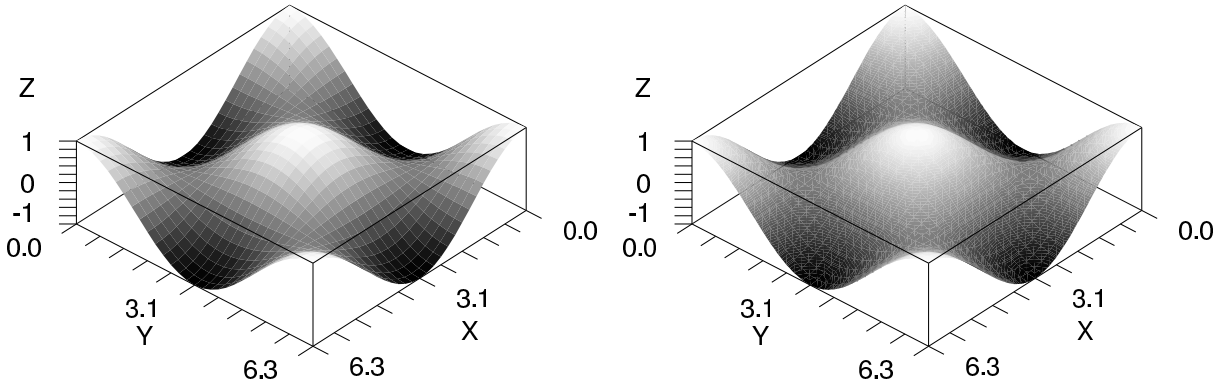
```
// przyk\l{}ad wykorzystania plot3d z interpolacj\k{a} kolor\'ow
function [col] = associe_couleur(val)
   // przypisanie koloru dla ka\.zdej z warto\'sci z val tutu przypsanie do warto\'sci?!!!
   n1 = 1                    // numer 1-ego koloru
   n2 =  xget("lastpattern") // numer ostatniego koloru
   nb_col = n2 - n1 + 1
   classes = linspace(min(val),max(val),nb_col)
   col = dsearch(val, classes)
endfunction

x=linspace(0,2*\%pi,31);
z=cos(x)'*cos(x);
[xf,yf,zf] = genfac3d(x,x,z);
xset("colormap",graycolormap(64))

zmeanf = mean(zf,"r");
zcolf = associe_couleur(zmeanf);
zcols = associe_couleur(zf);

xbasc()
xset("font",6,2)  // font 6 (helvetica) tutu n'est disponible
                  // que dans la version cvs de scilab
subplot(1,2,1)
   plot3d(xf,yf,list(zf,zcolf), flag=[-1 4 4])
```

```
   xtitle("Jeden kolor na tutu face")
subplot(1,2,2)
   plot3d(xf,yf,list(zf,zcols), flag=[-1 4 4])
   xtitle("Jeden kolor na wierzcho\l{}ek")
xselect()
```



shade

Rysunek 19: Z i bez interpolacji kolorów.

4.11 Krzywe w przestrzeni

Podstawową funkcją pozwalającą rysować krzywe w przestrzeni jest `param3d`. Oto klasyczny przykład tutu de l'hélice:

```
t = linspace(0,4*\%pi,100);
x = cos(t); y = sin(t) ; z = t;
param3d(x,y,z)  // ewentualne wymazanie okna graficznego za pomoc\k{a} xbasc()
```

tutu mais comme cette dernič̈re ne permet que d'affichier une seule courbe nous allons nous concentrer sur `param3d1` qui permet de faire plus de choses. Oto jej składnia:

```
param3d1(x,y,z <,opt_arg>*)
param3d1(x,y,list(z,colors) <,opt_arg>*)
```

Macierze `x, y` et `z` muszą być tego samego wymiaru (np,nc) a ilość krzywych (nc) jest określona przez ilość kolumn (jak dla `plot2d`). Parametry opcjonalne są takie same jak dla fnkcji `plot3d`, modulo le fait que `flag` tutu ne ne comporte pas de paramčtre *mode*.

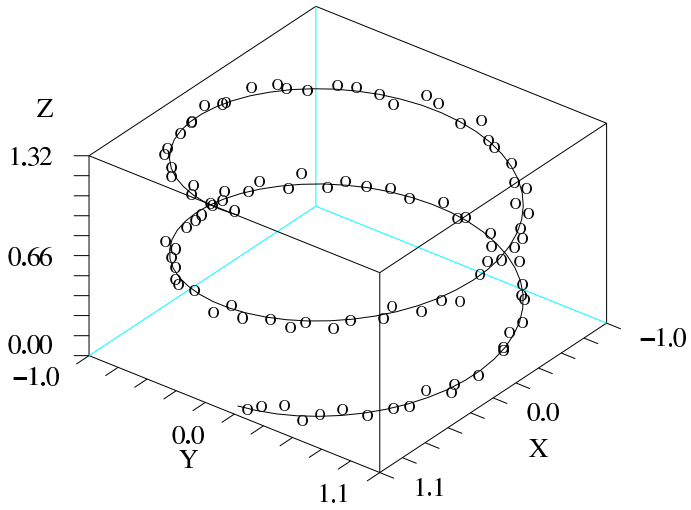
`colors` jest wektorem określającym styl dla każdej krzywej (dokładnie jak `plot2d`), to znaczy gdy `colors(i)` jest wartością całkowitą dodatnią, i-ta krzywa rysowana jest i-tym kolorem z bieżącej palety kolorów (tutu ou avec différents pointillés sur un terminal noir et blanc); jeśli zaś zawarta jest pomiędzy -9 i 0, otrzymujemy rysnek złożony z punktów (tutu non reliés) zaznaczonych odpowiednim symbolem. Oto przykład, który powinien doprowadzić nas do rysunku (20):

```
t = linspace(0,4*\%pi,100)';
x1 = cos(t); y1 = sin(t) ; z1 = 0.1*t;  // spirala
x2 = x1 + 0.1*(1-rand(x1));
y2 = y1 + 0.1*(1-rand(y1));
z2 = z1 + 0.1*(1-rand(z1));
xbasc();
xset("font",2,3)
param3d1([x1 x2],[y1 y2],list([z1 z2], [1,-9]), flag=[4 4])
xset("font",4,4)
xtitle("Spirala z per\l{}ami")
```

Jak dla `plot2d` funkcję tą należy wywołać kilkakrotnie jeśli różne krzywe nie mają takiej samej ilości punktówon. Oto skrypt, wyjaśniający jak utworzyć dwie grupy punktów ze różnymi znakami i kolorami:

```
n = 50;        // ilo\'s\'c punkt\'ow
P = rand(n,3); // losowanie punkt\'ow
// tutu on impose les dimensions de la boite englobante
ebox = [0 1 0 1 0 1];
```

Helice avec perles



Rysunek 20: Krzywa i punkty w przestrzeni.

helice

```
// rozdzielenie punkt\'ow na dwie grupy aby pokaza\'c jak przypisa\'c
// r\'o\.zne symbole i kolory do punkt\'ow
m = 30;
P1 = P(1:m,:) ; P2 = P(m+1:n,:);

// rysunek
xbasc()
// pierwsza grupa punkt\'ow
xset("color",2)  // tutu du bleu avec la carte par defaut niebieski w domy\'slnej palecie (?!)
param3d1(P1(:,1),P1(:,2),list(P1(:,3), -9), alpha=60, theta=30,...
         leg="x@y@z", flag=[3 4], ebox=ebox)
      // tutu flag=[3 4] : 3 -> echelle iso se basant sur ebox
      // tutu             4 -> boite + graduation
// druga grupa punkt\'ow
xset("color",5)  // tutu du rouge avec la carte par defaut
param3d1(P2(:,1),P2(:,2),list(P2(:,3), -5), flag=[0 0])
     // tutu -5 pour des triangles inverses
     // tutu [0 0]  : echelle fixée et cadre dessiné avec l'appel précédent
xset("color",1) // aby ustawi\'c czarny jako bie\.z\k{a}cy kolor
xtitle("Punkty...")
xselect()
```

4.12 Różności

Istnieje jeszcze wiele prymitywów graficznych:

1. `contour2d` i `contour` pozwalają rysować linie poziomicowe dla funkcji $z = f(x, y)$ określonej na prostokącie;
2. `grayplot` i `Sgrayplot` pozwalają reprezentować wartości funkcji tutu qui permettent de représenter les valeurs d'une telle fonction en utilisant des couleurs ;
3. `fec` odgrywaja taką samą rolę jak dwie poprzednie dla funkcji określonej tutu est définie sur une triangulation plane ;
4. `champ` pozwala określić pole wektorowe w 2D ;
5. tutu wiele funkcji wywoływanych w tym rozdziale dopuszcza różne parametry aby tworzyć wykresy funkcji bardziej bezpośrednio si on fournit une fonction scilab comme argument (le nom de ces fonctions commence par un f `fplot2d, fcontour2d, fplot3d, fplot3d1, fchamp`,...).

Aby zdać sobie sprawę z możliwości[28] wystarczy przejrzeć tutu rubrique (dział,sekcja (?!)) Graphic Library pomocy. Od wersji 2.7 istnieje nowy model graficzny „zorientowany obiektowo" pozwalający modyfikować właściwości grafiki po ujrzeniu rysunku. Domyślnie nie jest on aktywny, ale jeśli ktoś chce poeksperymentować należy wydać polecenie:

```
set("figure_style","new")
```

przed utworzeniem rysunku. Jako że ten nowy tryb jest w trakcie rozwijania zalecane jest używanie wersji tutu cvs de scilab.

Biblioteka Enrico Ségré, którą można tutu „ściągnąć" z jego strony:

`http://www.weizmann.ac.il/~fesegre/`

tutu complčte celle de Scilab et contient takze funkcje pozwalające uprościć tutu certaines taches.

5 Zastosowania i uzupełnienia

Rozdział ten przedstawia metody rozwiązywania w Scilabie pewnych typów problemów analizy numerycznej (aktualnie równań różniczkowych...) oraz dostarcza uzupełnień/dodatkowych informacji niezbędnych podczas projektowania prostych symulacji stochastycznych.

5.1 Równania różniczkowe

Scilab dysponuje potężnym interfejsem dla rozwiązywania numerycznego (w sposób przybliżony) równań różniczkowych przy zastosowaniu prostej instrucji `ode`. Rozważmy zatem następujące równanie różniczkowe z warunkiem początkowym :

$$\begin{cases} u' = f(t,u) \\ u(t_0) = u_0 \end{cases}$$

gdzie $u(t)$ jest wektorem w $\mathbb{R}^n$, f jest funkcją $\mathbb{R} \times \mathbb{R}^n \longrightarrow \mathbb{R}^n$, oraz $u_0 \in \mathbb{R}^n$. Zakładamy warunki brzegowe dla których rozwiązanie istnieje i jest jednoznaczne do okresu T.

5.1.1 Podstawowe użycie `ode`

Zdefiniujmy funkcj f jako funkcję Scilaba w następującyj sposób :

```
function [f] = MojaFunkcja(t,u)
   //
   tutaj kod definiujacy f jako funkcje t i u.
endfunction
```

Rmq : Podobnie, jeśli równanie jest ...autonome..., należy doprowadzić równanie do postaci w której mamy funkję zależne od zmiennej t Oto przykład kodu dla równania Van der Pola :

$$y'' = c(1-y^2)y' - y$$

kładąc $u_1(t) = y(t)$ oraz $u_2(t) = y'(t)$ przekształcamy je do układu dwuch równań różniczkowych pierwaszego rzędu :

$$\frac{d}{dt}\begin{bmatrix} u_1(t) \\ u_2(t) \end{bmatrix} = \begin{bmatrix} u_2(t) \\ c(1-u_1^2(t))u_2(t) - u_1(t) \end{bmatrix}$$

```
function [f] = VanDerPol(t,u)
  // second membre pour Van der Pol (c = 0.4)
  f(1) = u(2)
  f(2) = 0.4*(1 - u(1)^2)*u(2) - u(1)
endfunction
```

Następnie wywołujemy `ode` aby rozwiązać równanie (układ równań) względem t_0 w T, wychodząc od u_0 (wektor kolumnowy) i chcąc otrzymać rozwiązanie w chwili $t(1) = t_0,\ t(2),\ ...,\ t(m) = T$, wpiszemy instrukcję :

```
t = linspace(t0,T,m);
[U] = ode(u0,t0,t,MojaFunkcja)
```

[28] tutu attention risque de noyade !

Otrzymamy w ten sposób *macierz* `U` *o wymiarach* (n, m)*, w której* `U(i,j)` *jest rozwiązaniem częściowym/cząstkowym* $u_i(t(j))$ *(i-ta składowa w chwili* $t(j)$*).* Uwaga : *Liczba składowych branych dla* `t` *(czyli momenty w których otrzymuje się rozwiązanie) nie mają nic wspólnego z precyzją obliczeń. ..... Funkcja* `ode` *bazuje na wielu algorytmach pozwalających dostosowanie jej do wielu sytuacji... Aby wybrać odpowiednią metodę należy dodać odpowiedni parametr w trakcie wywołania (patrz Help). Domyślnie (tzn. bez wybierania explicite jednej metody) stosowana jest metoda Adamsa predykcji, natomiast w przypadku gdy Scilab określi równanie jako sztywne*[29] *algorytm zostaje zmieniony na metodę Gear-a.*

Oto kompletny przykład dla równania Van der Pola. W tym przypadku przestrzeń stanów jest płaska, można zatem otrzymać obraz dynamiki rysując pole wektorowe w prostokącie $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$ *za pomocą instrukcji graficznej* `fchamp` *w następujący sposób:*

```
fchamp(MojaFunkcja,t,x,y)
```

gdzie `MojaFunkcja` *oznacza nazwę funkcji Scilaba będącą składnikiem po prawej stronie równania różniczkowego,* `t` *jest momentem w którym chcemy naszkicować pole (w przypadku ciągłym równania możemy przyjąć dowolną wartość np. 0) oraz* `x` *i* `y` *są wektorami wierszowymi o nx i ny współrzędnych wyznaczaqcymi punkty siatki na której znajdują się kirunki wyznaczające pole wektorowe.*

```
// 1/ kre\'slimy pole wektorowe odpowiadaj\k{a}ce r\'ownaniu Van der Pola
n = 30;
delta = 5
x = linspace(-delta,delta,n); // tutaj y = x
xbasc()
fchamp(VanDerPol,0,x,x)
xselect()

// 2/ rozwi\k{a}zujemy r\'ownanie r\'o\.zniczkowe
m = 500 ; T = 30 ;
t = linspace(0,T,m);  // moment w kt\'orym znajduje sie rozwi\k{a}zanie
u0 = [-2.5 ; 2.5];    // warunek pocz\k{a}tkowy
[u] = ode(u0, 0, t, VanDerPol);
plot2d(u(1,:)',u(2,:)',2,"000")
```

5.1.2 Van der Pol jeszcze raz

W tej części zwrócimy naszą uwagę na możliwości graficzne Scilaba pozwalanqce otrzymać wiele żądanych trajektorii bez ponownego uruchamiania skryptu z inną wartością argumentu u_0 *..... Po wyświetleniu pola wektorowego, każdy warunek początkowy będzie zadany poprzez kliknięcie lewym przyciskiem myszy*[30]*; pojawi się wówczas punkt na żądanej pozycji początkowej. Jest to możliwe dzięki funkcji* `xclick`*, której składnia jest następująca :*

```
[c_i,c_x,c_y]=xclick();
```

Jak widać, Scilab dyspinuje odpowiednimi procedurami, które umożliwiają miedzy innymi reagowanie na tzw zdarzenia graficzne jak kliknięcie myszą . Kiedy takie zdarzenia ma miejsce, następuje automatczna lokalizacja pozycji punktu (w bieżącej skali) poprzez przypisanie jego współżędnych do zmiennych `c_x` *oraz* `c_y`*. Trzeci argument, czyli* `c_i` *oznacza odpowiedni klawisz myszy zgodnie z poniższą tabelką :*

wartość dla `c_i`	*klawisz*
0	*lewy*
1	*środkowy*
2	*prawy*

W skrypcie wykożystano kliknięcie na lewy klawisz myszy jako ten, wyznaczający punkt początkowy dla pęku zdarzeń.

Na koniec nadano każdej z wyrysowanych trajektorii inny kolor (tablica `couleur` *pozwala wybrać jeden z dostępnych kolorów). Aby otrzymać skalę izometryczną urzyjemy* `fchamp` *dodając opcjonalny argument jak dla* `plot2d` *(należy użyć* `strf=`wartosc_strf *jako że parametry* frameflag *i* axesflag *nie są .....). Ostatni aspekt : aby zaznaczyć punkt początkowy obrysowano go nałum kółkiem, natomiast aby pokazać wszystkie możliwości okna graficznego wykożystano sześcienny układ współrzędnych. Ostatnia uwaga : podczas gdy jest wyświetlone pole wektorowe, można maksymalizować okno graficzne! Klikając dwukrotnie otrzymano rysunek (21) wszystkie trajektorie zbieżne na jednej sferze, będącej ..... dla tego równania*

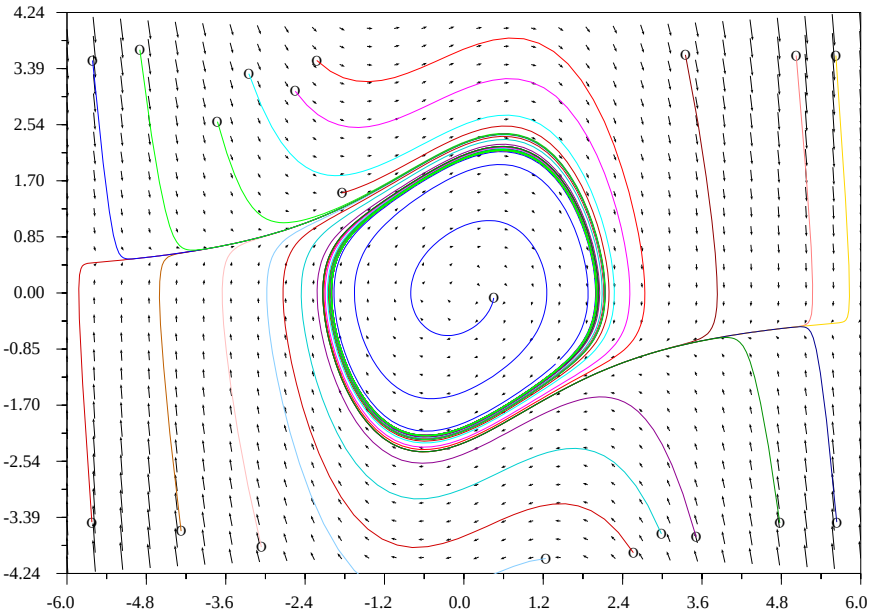
[29] równanie jest *raide w przypadku gdy (mniej lub bardziej) łączy się z metodami jednoznacznymi*

[30] jak sugerowano w jednym z artykułów dotyczących Scilaba, zamieszczonym w *Linux Magazine*

```
// 1/ wykres pola wektorowego dla rownania Van der Pola
n = 30;
delta_x = 6
delta_y = 4
x = linspace(-delta_x,delta_x,n);
y = linspace(-delta_y,delta_y,n);
xbasc()
fchamp(VanDerPol,0,x,y, strf="041")
xselect()

// 2/ resolution de l'equation differentielle
m = 500 ; T = 30 ;
t = linspace(0,T,m);

couleurs = [21 2 3 4 5 6 19 28 32 9 13 22 18 21 12 30 27]  // 17 couleurs
num = -1
while %t
   [c_i,c_x,c_y]=xclick();
   if c_i == 0 then
      plot2d(c_x, c_y, style=-9, strf="000")  // un petit o pour marquer la C.I.
      u0 = [c_x;c_y];
      [u] = ode(u0, 0, t, VanDerPol);
      num = modulo(num+1,length(couleurs));
      plot2d(u(1,:)',u(2,:)', style=couleurs(num+1),  strf="000")
   elseif c_i == 2 then
      break
   end
end
```



erpol

Rysunek 21: Quelques trajectoires dans le plan de phase pour l'équation de Van der Pol

5.1.3 Troche więcej o ode

W tym drugim przykładzie pokarzemy inne zastosowanie funkcji `ode` *dla równań z parametrem. ..... Oto nasze nowe równanie różniczkowe (*Równanie Brusselator*) :*

$$\begin{cases} \frac{du_1}{dt} = 2 - (6+\epsilon)u_1 + u_1^2 x_2 \\ \frac{du_2}{dt} = (5+\epsilon)u_1 - u_1^2 u_2 \end{cases}$$

które przyjmuje jako punkt krytyczny $P_{stat} = (2, (5+\epsilon)/2)$*. Ponieważ wartości parametru* ϵ *zmieniają sie z ujemnej na dodatnią, punkt stacjonarny zmienia swoją charakter (jest stały lub zmienny, wchodząc, dla* $\epsilon = 0$ *w zjawisko rozgałęzienia* Hopf*). Interesują nas trajektorie z warunkiemi początkowymi z sąsiedztwa tego punktu. Oto funkcja licząca to równanie :*

```
function [f] = Brusselator(t,u,eps)
   //
   f(1) = 2 - (6+eps)*u(1) + u(1)^2*u(2)
   f(2) = (5+eps)*u(1) - u(1)^2*u(2)
endfunction
```

Aby podać parametr, zastąpimy w wywołaniu `ode` *nazwę funkcji (tutaj Brusselator) przez uporządkowaną listę zawierającą nazwę funkcji oraz jej parametry :*

```
[x] = ode(x0,t0,t,list(MojaFunkcja, par1, par2, ...))
```

W naszym przypadku :

```
[x] = ode(x0,t0,t,list(Brusselator, eps))
```

a następnie zastosujemy `fchamp` *aby narysować pole.*

Aby ustalić zakres tolerancji dla błędu lokalnego rozwiązania użyjemy dodatkowych parametrów `rtol` *oraz* `atol` *zaraz po nazwie funckji (lub listy zawierającej tę funkcję wraz z jej parametrami. W każdym kroku czasowym,* $t_{k-1} \to t_k = t_{k-1} + \Delta t_k$*, obliczane jest oszacowanie błędu lokalnego* e *(tzn. błędu dla kroku czasowego wychodząc z warunku początkowego* $v(t_{k-1}) = U(t_{k-1})$*) :*

$$e(t_k) \simeq U(t_k) - \left(\int_{t_{k-1}}^{t_k} f(t, v(t))dt + U(t_{k-1}) \right)$$

(drugi składnik jest rozwiązaniem dokładnym zależnym od rozwiązania mumerycznego $U(t_{k-1})$ *otrzymanego w poprzednim kroku) a następnie sprawdza czy zawiera się on w zakresie tolerancji formuowane za pomocą dwóch parametrów* `rtol` *et* `atol` *:*

$$tol_i = rtol_i * |U_i(t_k)| + atol_i,\ 1 \le i \le n$$

w przypadku gdy dane są dwa wektory długości n *dla tych praramertów, oraz:*

$$tol_i = rtol * |U_i(t_k)| + atol,\ 1 \le i \le n$$

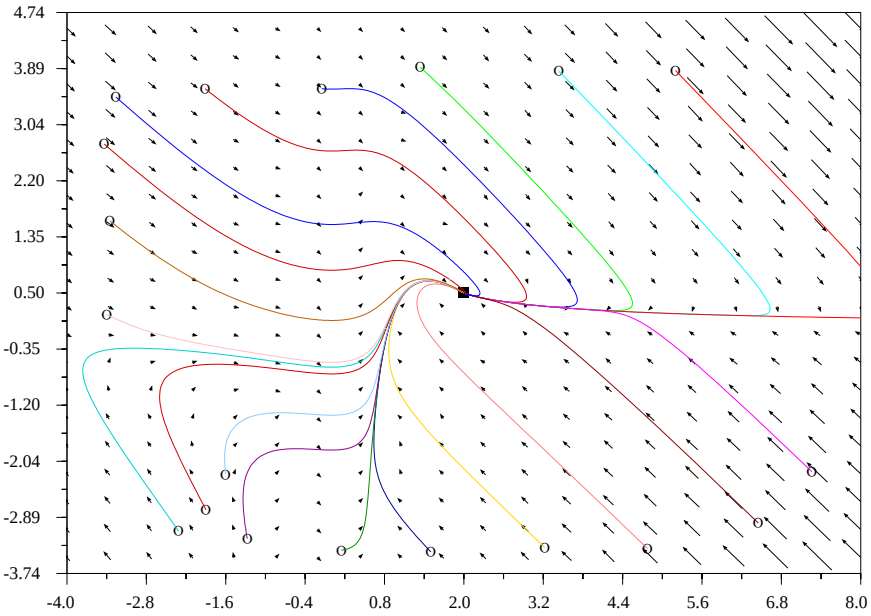
gdy dane są skalary. Jeżeli $|e_i(t_k)| \le tol_i$ *dla wszystkich* t_k*, krok jest akceptowany a następnie obliczony zostaje nowy krok czasowy w taki sposób, że kryterium nałożone na przyszły błąd będzie pewnym sposobem jego realizacji<–?????. W przeciwnym razie, ponownie całkuje sie wyrażenie dla* $t_(k-1)$ *przyjmując nowy, nieco mniejszy krok(.....). Metody tego typu oprócz zmiennych postępu czasowgo, manipulują również kolejnością równań w celu otrzymania dobrej skuteczności/wydajności informacji... Domyślnie stosowanymi wartościami są* $rtol = 10^{-5}$ *i* $atol = 10^{-7}$*(za wyjątkiem typów wybierających metodę Runge Kutta). Ważna uwaga : całkowanie maże często sie nie powiźć...*

Poniżej przedstawiamy przykładowy skrypt, w którym punkty krytyczne oznaczono małymi, czarnymi kwadratami (otrzymano je przy pomocy prostej funkcji graficznej `xfrect`*) :*

```
// Brusselator
eps = -4
P_stat = [2 ; (5+eps)/2];
// granice dla wykresu pola wektorowego
delta_x = 6; delta_y = 4;
x_min = P_stat(1) - delta_x; x_max = P_stat(1) + delta_x;
y_min = P_stat(2) - delta_y; y_max = P_stat(2) + delta_y;
n = 20;
x = linspace(x_min, x_max, n);
y = linspace(y_min, y_max, n);
// 1/ wykres pola wektorowego
xbasc()
fchamp(list(Brusselator,eps),0,x,y, strf="041")
xfrect(P_stat(1)-0.08,P_stat(2)+0.08,0.16,0.16) // dla zaznaczenia punktow krytycznych
```

```
xselect()

// 2/ rozwiazanie rownania rozniczkowego
m = 500 ; T = 5 ;
rtol = 1.d-09; atol = 1.d-10; // zakres tolerancji bledu
t = linspace(0,T,m);
couleurs = [21 2 3 4 5 6 19 28 32 9 13 22 18 21 12 30 27]
num = -1
while %t
   [c_i,c_x,c_y]=xclick();
   if c_i == 0 then
      plot2d(c_x, c_y, style=-9, strf="000")  // male o aby zanaczyc C.I.
      u0 = [c_x;c_y];
      [u] = ode(u0, 0, t, rtol, atol, list(Brusselator,eps));
      num = modulo(num+1,length(couleurs));
      plot2d(u(1,:)',u(2,:)', style=couleurs(num+1), strf="000")
   elseif c_i == 2 then
      break
   end
end
```



ator1

Rysunek 22: Niektóre trajektorie w polu fazowym dla Brusselatora ($\epsilon = -4$)

5.2 Generowniw liczb losowych

5.2.1 Funkcja `rand`

Do tej pory funkcja ta służyła nam głównie do wypełniania liczbami losowymi naszych macierzy i wektorów... Funkcja ta używa liniowego generatora następująco[31] *:*

$$X_{n+1} = f(X_n) = (aX_n + c) \; mod \; m, \; n \geq 0, \quad gdzie \begin{cases} m = 2^{31} \\ a = 843314861 \\ c = 453816693 \end{cases}$$

[31] Według tego, co autor rozumiał oglądanąc kod

Jej okres jest z pewnością równy m (oznacza to, że f jest permutacją cykliczną na przedziale $[0, m-1]$.) Zauważmy, że wszystkie generatory losowe w komputerze są ..... Aby przekształcić je do liczb rzeczywistych z przedziału $[0, 1[$, dzieli się otrzymaną daną przez m (i otrzymuje się generator liczb rzeczywistych.....). Składnik początkowy szeregu jest często zwany inicjatorem i przyjmuje domyślnie wartość $X_0 = 0$. Zatem pierwsze wywołanie `rand` *(pierwszy otrzymany współczynnik jeśli wypełniamy macierz lub wektor) jest zawsze :*

$$u_1 = 453816693/2^{31} \approx 0.2113249$$

Istnieje możliwość zmiany, w dowolnym momencie inicjatora za pomocą instrukcji :

```
rand("seed",inicjator)
```

gdzie inicjator jest zawarty w przedziale $[0, m-1]$. Często istnieje potrzeba zainicjowania szeregu poprzez wybór inicjatora mniej lub bardziej przypadkowo (sytuacja taka, aby nie mić tej samen liczby za każdym razem), możemy chcieć na przykład otrzymać losowo datę lub godzine czy też kombinaować inicjatory<-??? Scilab dysponuje funkcją `getdate` *która generuje wektor złożony z 9 elementów. Wśród nich są :*

- *drugi oznacza miesiąc (1-12),*
- *szusty, dzień miesiąca (1-31),*
- *siudmy, godzinę (0-23),*
- *ósmy, minyty (0-59),*
- *i dziewiąty, sekundy (0-61 ?).*

Aby otrzymać inicjator można na przykład dodawać powyższe elementy między sobą :

```
v = getdate()
rand("seed", sum(v([2 6 7 8 9])))
```

Zwróćmy uwagę również na możliwość zastąpienia bierzącego inicjatora przez :

```
germe = rand("seed")
```

Począwszy od rozkładu jednostajnego na $[0, 1[$, można otrzymać inne rozkłady a funkcja `rand` *dostarcza dostateczny interface pozwalający otrzymać rozkład normalny (średnia 0 i wariancja 1). Aby przejść od jednego do drugiego, stosuje sie następującą składnie :*

```
rand("normal")  // aby otrzyma\'c rozk\l{}ad normalny
rand("uniform") // aby powr\'oci\'c do rozk\l{}adu jednostajnego
```

Domyślnie generator przyjmuje rozkład jednostajny ale jest rozsądnie w każdej symulacji upewnić sie czy `rand` *daje oczekiwany wynik używając jedną w tych instrukcji. Można zrasztą sprawdzić aktualny rozkąad za pomocą :*

```
loi=rand("info")  // rozk\l{}ad jest jeden z dw\'och mo\.zliwo\k{a}ci "jednostajny" lub "n
```

Ponowne wywołanie `rand` *może przyjąć następujące formy :*

1. `A = rand(n,m)` *uzupełnia macierz* `A` *(n,m) liczbami losowymi ;*
2. *jeśli* `B` *jest macierzą już zdefiniowaną o wymiarach (n, m) to* `A = rand(B)` *daje ten sam efekt (pozwala uniknąć odzyskania<-???? wymiarów macierzy* `B`*) ;*
3. *wreszcie,* `u = rand()` *daje pojedynczą liczbę losową.*

Do dwuch pierwszych metod można dodać argument aby wskazać dodatkowo rozkład : `A = rand(n,m,loi)`, `A = rand(B,loi)`, *gdzie* `loi` *jest jednym z dwóch łańcuchów* `normal` *lub* `uniform`.

Wybrane zastosowania z użyciem funkcji `rand` *Począwszy od rozkładu jednostajnego, w prosty sposób można otrzymać macierz (n,m) liczb wdług :*

1. *rozkład jednostajny na $[a, b]$:*

```
X = a + (b-a)*rand(n,m)
```

2. *rozkład jednostajny na liczbach całkowitych z przedziału $[n_1, n_2]$:*

```
X = floor(n1 + (n2+1-n1)*rand(n,m))
```

(losujemy następujące liczby rzeczywiste rozkładu jednostajnego na przedziele rzczeywistym $[n_1, n_2 + 1[$ a następnie bierzemy ich część całkowitą).

Symulacja pojedynczej próby Bernulliego z prawdopodobieństwem sukcesu p :

```
succes = rand() < p
```

otrzymujemy dzięki prostej metodzie[32] *symulującej rozkład dwumianowy $B(N, p)$:*

```
X = sum(bool2s(rand(1,N) < p))
```

(`bool2s` *przekształca sykcesy w 1 i nie sumuje ich w funkcji* `sum`*). Z uwagi na fakt, że wykonywanie iteracji przez Scilaba jest dośc wolne, można otrzymać bezpośrednio wektor (kolumnowy) składający się z m realizacji tego rozkładu :*

```
X = sum(bool2s(rand(m,N) < p),"c")
```

ale kożyśtne będzie użycie funkcji `grand`*, która używa metodę bardziej wyczynowej<-???. Z drugiej strony, jeśli używacie tego sposobu*[33]*, jest bardziej przejżyście aby kodować go jako funkcję Scilaba. A oto prosta funkcja symulujące rozkład geometryczny (ilość prób Bernulliego potrzebnych aby otrzymać sukces)*[34] *:*

```
function [X] = G(p)
   // rozklad geometryczny
   X = 1
   while rand() > p   // pora\.zka
      X = X+1
   end
endfunction
```

Wreszcie, w nawiązaniu do rokładu Normalnego $\mathcal{N}(0, 1)$, otrzymamy rozkład Normalny $\mathcal{N}(\mu, \sigma^2)$ (średnia μ i odchylenie standardowe σ) za pomocą :

```
rand("normal")
X = mu + sigma*rand(n,m) // aby otrzymac macierz (n,m) tych liczb
// lub przy uzyciu pojedynczej instrukcji : X = mu + sigma*rand(n,m,"normal")
```

5.2.2 Funkcja `grand`

W skomplikowanych symulacjach wykożystujących wiele liczb losowych klasyczna funkcja `rand` *z jej okresem rzędu $2^{31}(\simeq 2.147\ 10^9)$ może okazać się trochę za ciasna. Jest zatem wskazane użycie* `grand` *która również umożliwia symulację wszystkich klasycznych rozkładów.* `grand` *używa sie prawie w taki sam sposób co* `rand`*, tzn. że można użyć jednej z dwóch następujących składni (oczywiście dla drugiej macierz* `A` *musi być zdefiniowana w momencie jaj wywołania) :*

```
grand(n,m,loi, [p1, p2, ...])
grand(A,loi, [p1, p2, ...])
```

oů `loi` *oznacza łańcuch znaków precyzujący rozkład po którym nasępują ewentualnie jego parametry. Kilka przykładów (aby otrzymać próbę n realizacji, pod postacią wektora kolumnowego) :*

1. *rozkład jednostajny na liczbach całkowitych z dużego przedziału $[0, m[$:*

   ```
   X = grand(n,1,"lgi")
   ```

   gdzie m zależy od generatora bazy (domyślnie $m = 2^{32}$) ;

2. *rozkład jednostajny na liczbach całkowitych z przedziału $[k_1, k_2]$:*

   ```
   X = grand(n,1,"uin",k1,k2)
   ```

   (musi być spełniony warunek aby $k_2 - k_1 \leq 2147483561$ bo w przeciwnym wypadku problem ten jest sygnalizowany jako błąd);

3. *dla rozkładu jednostajnego na przedziale $[0, 1[$:*

[32] ale bardzo efektywnej dla duchych N !
[33] w ogólności używać będziemy funkcji `grand` pozwala otrzymać wiekszość klasycznych rozkładów
[34] ta funkcja jest bardziej efektywna dla małych p.

```
X = grand(n,1,"def")
```

4. *dla rozkładu jednostajnego na przedziale* $[a, b[$ *:*

```
X = grand(n,1,"unf",a,b)
```

5. *dla rozkładu dwumianowego* $B(N, p)$ *:*

```
X = grand(n,1,"bin",N,p)
```

6. *dla rozkładu geometrycznego* $G(p)$ *:*

```
X = grand(n,1,"geom",p)
```

7. *dla rozkładu Poissona ze średnią* μ *:*

```
X = grand(n,1,"poi",mu)
```

8. *dla rozkładu wykładniczego ze średnią* λ *:*

```
X = grand(n,1,"exp",lambda)
```

9. *dla rozkładu normalnego ze średnią* μ *i odchylaniem standardowym* σ *:*

```
X = grand(n,1,"nor",mu,sigma)
```

Inne przykłady można znaleźć na stronach pomocy.
générateurs de base ??????.....
Pour opérer sur ces générateurs de base, vous pouvez utiliser les instructions suivantes :

`nom_gen = grand("getgen")`	*permet de récupérer le (nom du) générateur courant*
`grand(śetgen",nom_gen)`	*le générateur* `nom_gen` *devient le générateur courant*
`etat = grand("getsd")`	*permet de récupérer l'état interne du générateur courant*
`grand(śetsd",e1,e2,..)`	*impose l'état interne du générateur courant*

La dimension de l'état interne de chaque générateur dépend du type du générateur : de un entier pour urand *ŕ 624 entiers plus un index pour* Mersenne Twister[35]*. Si vous voulez refaire exactement la męme simulation il faut connaître l'état initial (avant la simulation) du générateur utilisé et le sauvegarder d'une façon ou d'une autre. Exemple :*

```
grand("setgen","kiss")   // kiss devient le générateur courant
e = [1 2 3 4];           // etat que je vais imposer pour kiss
                         // (il lui faut 4 entiers)
grand("setsd",e(1),e(2),e(3),e(4));    // voila c'est fait !
grand("getsd")           // doit retourner le vecteur e
X = grand(10,1,"def");   // 10 nombres
s1 = sum(X);
X = grand(10,1,"def");   // encore 10 nombres
s2 = sum(X);
s1 == s2                 // en général s1 sera different de s2

grand("setsd",e(1),e(2),e(3),e(4));   // retour ŕ l'état initial
X = grand(10,1,"def");   // de nouveau 10 nombres
s3 = sum(X);
s1 == s3                 // s1 doit etre egal a s3
```

[35] une procédure d'initialisation avec un seul entier existe néanmoins pour ce générateur.

5.3 Dystrubuanty i ich odwrotności

Dystrybuanty są często używane przy testach statystycznych (χ^2_r, ...) ponieważ pozwalają na obliczenia, niech :

1. *dystrybuanta w 1 lub kilku punktach ;*
2. *jej odwrotność w 1 lub kilku punktach ;*
3. *jeden z parametrów rozkładu jest dany przez pozostałe i parę $(x, F(x))$;*

W Helpie, dystrybuanty można znaleźć pod hasłem Cumulative Distribution Functions... , wszystkie te funkcje poprzedzone są skrótem `cdf`*. Przykładowo dla rozkładu Normalnego $\mathcal{N}(\mu, \sigma^2)$, funkcja która nas interesuje nosi nazwę* `cdfnor` *i jej składnia jest następująca :*

1. `[P,Q]=cdfnor("PQ",X,mu,sigma)` *aby otrzymać $P = F_{\mu,\sigma}(X)$ i $Q = 1 - P$,* `X`*,* `mu` *oraz* `sigma` *mogą być wektorami (o tych zamych wymiarach) i wówczas otrzymuje sie dla* `P` *i* `Q` *wektor za pomocą $P_i = F_{\mu_i,\sigma_i}(X_i)$;*
2. `[X]=cdfnor("X",mu,sigma,P,Q)` *aby otrzymać $X = F^{-1}_{\mu,\sigma}(P)$ (podobnie jak poprzednio argumentami mogą być wektory o tych zamych wymiarach i wówczas otrzymuje się $X_i = F^{-1}_{\mu_i,\sigma_i}(P_i)$;*
3. `[mu]=cdfnor("Mean",sigma,P,Q,X)` *aby otrzymać średnią ;*
4. *i ostatecznie* `[sigma]=cdfnor("Std",P,Q,X,mu)` *aby dostać odchylenie standardowe.*

Dwie ostatnie składnie funkcjonują także gdy argumentami są wektory o jednakowych wymiarach.
Uwagi :

- *możliwość jednoczesnej pracy z p oraz $q = 1 - p$ pozwala uzyskać precyzję w obszarze gdzie p jest bliskie 0 lub 1. Dla p bliskiego 0 funkcja wykożystuje wartość p, natomiast dla p bliskiego 1 funkcja wykożystuje wartość q ;*
- *lańcuch znaków pozwalający otrzymać funkcję odwrotną nie zawsze ma postać* `X`*... zobacznie na odpowiednich stronach pomocy.*

5.4 Proste symulacje stochastyczne

5.4.1 Wprowadzenie i notacja

Często symulacja polega przede wszystkim na otrzymaniu wektora :

$$x^m = (x_1, ..., x_m)$$

którego składowe są rozumiane jako realizacje niezależnych zmiennych losowych i podobnie rozkład $X_1, X_2, ..., X_m$ (będziemy zapisywać przez X zmienną losową mającą ten sam rozkład <-???). W praktyce wektor x^m otrzymuje sie bezpośrednio lub pośrednio za pomocą funkcji `rand` *lub* `grand`[36]*.*

W przypadku próby x^m poszukujemy zbliżonych charakterystyk jej rozkładu takich jak wartość oczekiwana, odchylenie standardowe, dystrybuanta (lub funkcja gęstości) lub też gdy stawiamy hipotezę dotyczącą rozkładu, aby określić jej parametry, albo, gdy parametry są już znane, weryfikować za pomocą testów statystycznych, że nasza próba jest (możliwie) dobrze reprezentatywną zmienną losową która podąża efektywnie za rozkładem itd... <-?????????????????????????? Dla przypadków, które nas interesują, znane są przeważnie dokładne wyniki teoretyczne, a symulacja służy jedynie do zilustrowania wniosków, twierdzeń (lgn?, TCL =? CTG, itd...), działania metody lub algorytmu, ...

5.4.2 Przedziały ufności

Niekiedy o empirycznej wartości oczekiwanej otrzymanej z naszej próby (w Scilabie : `x_bar_m = mean(xm)`*) chcielibyśmy powiedzieć, że mieści się w pewnym przedziale. Ponadto chcielibyśmy znać ów przedział aby móc zapisać, żę :*

$$E[X] \in I_c \text{ z prawdopodobieństwem } 1 - \alpha$$

[36] w większości przypadków próba statystyczna dotyczy miar fizycznych (temperatura, ciśnienie,...), biometrycznych (wzrost,waga), sondaży, itd... otrzymane dane są zbierane w plikach (lub bazach danych); pewne programy (jak R) proponują dla nich układ danych (dla których studenci będą mogli robić ręcznie !) niestety Scilab nie dysponuje taką możliwością; niemniej przypadków gdzie używamy takich symulacji jest równie wiele, na przykład aby przestudiować zachowanie pewnych systemów wejściowych (lub zakłuceń) losowych lub podobnie aby rozwiązać problemy czysto deterministyczne ale dla których metody analizy numerycznej są zbyt skomplikowane lub niemożliwe do zaimplementowania<-??? .

gdzie najczęściej $\alpha = 0.05$ *lub* 0.01 *(przedziały ufności odpowiednio 95% i 99%). W celu wyznaczenia tych przedziałów za punkt wyjścia posłuży na C.T.G.. Jeśli przyjmiemy :*

$$\bar{X}_m = \frac{1}{m}\sum_{i=1}^{m} X_i$$

za zmienną losową dla wartości średniej (gdzie $\bar{x}_m$ *jest pojedynczą realizacją), wówczas prowo wielkich liczb mówi nam, że* $\bar{X}_m$ *jest zbieżne do* $E[X]$ *i na mocy C.T.G (przy pewnych założeniach...) mamy :*

$$\lim_{m\to+\infty} P(a < \frac{\sqrt{m}(\bar{X}_m - E[X])}{\sigma} \leq b) = \frac{1}{\sqrt{2\pi}}\int_a^b e^{-t^2/2}dt$$

(przyjmuje się, że $Var[X] = \sigma^2$*). Dla dostatecznie dużych* m *przyjmuje się, że :*

$$P(a < \frac{\sqrt{m}(\bar{X}_m - E[X])}{\sigma} \leq b) \approx \frac{1}{\sqrt{2\pi}}\int_a^b e^{-t^2/2}dt$$

Jeżeli chcemy aby przedział ufności był "symetryczny" :

$$\frac{1}{\sqrt{2\pi}}\int_{-a}^{a} e^{-t^2/2}dt = F_{N(0,1)}(a) - F_{N(0,1)}(-a) = 2F_{N(0,1)}(a) - 1 = 1 - \alpha$$

wówczas :

$$a_\alpha = F^{-1}_{N(0,1)}(1 - \frac{\alpha}{2}) \text{ albo } -F^{-1}_{N(0,1)}(\frac{\alpha}{2})$$

co zapisuje się w scilabie :

```
a_alpha = cdfnor("X", 0, 1, 1-alpha/2, alpha/2)
```

Otrzymujemy ostatecznie[37] *:*

$$E[X] \in [\bar{x}_m - \frac{a_\alpha\sigma}{\sqrt{m}}, \bar{x}_m + \frac{a_\alpha\sigma}{\sqrt{m}}]$$

z prawdopodobieństwem $1 - \alpha$ *(jeżeli przybliżenie granicy jest poprawne...).*

Problem pojawia się w momencie gdy nieznane jest odchylenie standardowe... Wykożystuje się wówczas etymator :

$$S_m = \sqrt{\frac{1}{m-1}\sum_{i=1}^{m}(X_i - \bar{X}_m)^2}$$

Zastępując odchylenie standardowe σ *przez* s_m *(gdzie* s_m *jest pojedynczą realizacją* S_m*), otrzymujemy przedział ufności, który przyjmujemy również dla wartości empirycznej.*

Szczególne przypadki :

1. *jeśli* X_i *ma rozkład normalny* $N(\mu, \sigma^2)$ *wówczas :*

   $$\sqrt{m}\frac{\bar{X}_m - \mu}{S_m} \sim t(m-1)$$

   gdzie $t(k)$ *ma rozkład Studenta o* k *stopniach swobody. w tym przypadku wszelkie poprzednie przybliżenia tracą moc (przybliżenie granicy i przybliżenie odchylenia standardowego) oraz otrzymuje sie :*

   $$\mu \in [\bar{x}_m - \frac{a_\alpha s_m}{\sqrt{m}}, \bar{x}_m + \frac{a_\alpha s_m}{\sqrt{m}}] \text{ avec probabilité } 1 - \alpha$$

   gdzie s_m *jest empirycznym odchyleniem stndardowym z próby (*`sm = st_deviation(xm)` *w scilabie) gdzie* a_α *jest wartością krytyczną rozkładu Studenta :*

   $$a_\alpha = F^{-1}_{t(m-1)}(1 - \frac{\alpha}{2})$$

   otrzymaną w scilacie[38] *przez :*

   ```
   a_alpha = cdft("T", m-1, 1-alpha/2, alpha/2)
   ```

2. *w momencie gdy wariancja wyraża się przez funkję wartości oczekiwanej (Bernouilli, Poisson, wykładniczy,...) można pzsłużyć die przybliżeniem odchylenia standardowego i otrzymuje się przedział ufności poprzez rozwiązanie odpowiedniej nierówności.*

[37] dla przedziału z 95%, mamy $a_\alpha \simeq 1.96$ często zastępowane przez 2.

[38] zobacz na odpowiednich stronach pomocy : właściwie funkcje `cdf` nie mają regularnej/uniwersalnej składni i `X` nie musi być zawsze oznaczeniem pozwalającym otrzymać funkcję odwrotną do dystybuanty, tutaj jest to `T` !

5.4.3 Wykres dystrubuanty empirycznej

Dystrybuantą zmienne losowej X nazywamy funkcję :

$$F(x) = \textit{Probabilité que } X \leq x$$

Dystrybuanta empiryczna pochodząca z próby x^m jest definiowana jako :

$$F_{x^m}(x) = card\{x_i <= x\}/m$$

Jest to funkcja schodkowa, którą liczy się łatwo jeśli posortujemy wektor x^m w porządku rosnącym (mamy więc $F_{x^m}(x) = i/m$ dla $x_i \leq x < x_{i+1}$). Standardowym algorytmem sortującym w scilabie jest funkcja `sort` *która sortuje malejąco*[39]*. Aby posortować wektor x^m w porządku rosnącym użyjemy :*

```
xm = - sort(-xm)
```

Łatwym sposobem narysowania wykresu jest funkcja `plot2d`*. Oto przykładowy kod :*

```
dystrybuanta_empiryczna(xm)
   //  wykres dystrybuanty (empirycznej)
   //  na pr\'obie xm
   m = length(xm)
   xm = - sort(-xm(:))
   ym = (1:m)'/m
   plot2d2(xm, ym, leg="dystrybuanta empiryczna")
endfunction
```

Zwróćmy uwagę na zapis : `xm(:)`*, który jest analogiczny do zapisywania wektora wierszowego.*

A teraz przykład dla rozkładu normalnego $\mathcal{N}(0,1)$:

```
m = 100;
xm = grand(m,1,"nor",0,1);
xbasc()
dystrybuanta_empiryczna(xm);  // wykres fct dystrybuanty empirycznej
// dane dla wykresu dystrybuanty "dokladnej"
x = linspace(-4,4,100)';
y = cdfnor("PQ", x, zeros(x), ones(x));
plot2d(x, y, style=2)      // nanosimy krzywa na pierwszy wykres
xtitle("Dystrybyanty dokladna i empiryczna")
```

który daje następujący wykres (23).

5.4.4 Test χ^2

Niech $x^m = (x_1, ..., x_m)$ będzie naszą próbą dla dalszej analizy. Niech będzie dana hipoteza $\mathcal{H}$: zmienne losowe postaci $(X_1, ..., X_m)$ mają rozkład $\mathcal{L}$. Chcemy wiedzieć czy hipoteza jest prawdziwa czy nie. Dla naszej próby można bez wątpienia obliczyć statystyki elementarne (średnią i odchylenie standardowe empiryczne) i jeżeli są one dostatecznie bliskie wartości oczekiwanej oraz odchyleniu standardowemu rozkładu $\mathcal{L}$, można wówczas zastosować test statystyczny. Test χ^2 stosuje się dla rozkładów dyskretnych o skończonej liczbie wartości. Na przykład zakładając, że rozkład $\mathcal{L}$ jest zadany przez $\{(v_i, p_i), 1 \leq i \leq n\}$. Test polega na obliczeniu wielkości :

$$y = \frac{\sum_{i=1}^{n}(o_i - mp_i)^2}{mp_i}$$

gdzie o_i jest liczbą realizacji x_j równych v_i i na przyrównaniu otrzymanej wielkości y do wartości progowej y_α, test będzie pozytywny[40] *jeśli $y \leq y_\alpha$.*

Jeżeli do równania na y wstawiwy w miejsce próby $x_1, ..., x_m$, zmienne losowe $X_1, ..., X_m$ w ten sposób zdefiniujemy[41] *zmienną losową Y, którą można przybliżać (dla dostatecznie dużych m) rozkładem χ^2 o $n-1$ stopniach swobody. Wartość progową otrzymujemy przez :*

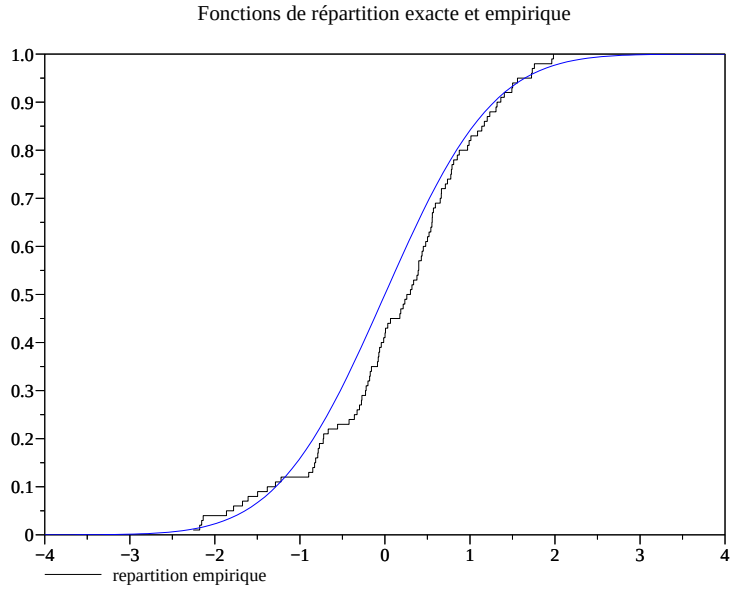
$$y_\alpha = F^{-1}_{\chi^2_{n-1}}(1-\alpha)$$

przy $\alpha = 0.05$ lub $\alpha = 0.01$. W Scilabie wartość tę otrzymamy poprzez :

[39] zobacz także funkcję `gsort`, która robi więcej rzeczy

[40] z intuicyjnego punktu widzenia jeżeli hipoteza jest dobra oczekujemy, że o_i nie odbiega zbytnio od mp_i, zatem jeżeli hipoteza jest fałszywa oczekujemy że otrzymamy wysoką wartość y, skąd odrzucimy hipotezę dla $y > y_\alpha$...

[41] poprzez zmienną losową wektorową $O = (O_1, ..., O_n)$ mającą rozkład wielomianowy<-?????



repart

Rysunek 23: Systrybuanta dokładna i empiryczna rozkładu normalnego

```
y_progowe = cdfchi("X", n-1, 1-alpha, alpha)
```

Aby obliczyć częstość o_i za pomocą Scilaba możemy użyć następującej metody :

```
occ = zeros(n,1);
for i=1:n
   occ(i) = sum(bool2s(xm == v(i))); // albo length(find(xm==v(i)))
end
if sum(occ) ~= m then, error("b\l{}\k{a}d oblicze\'n"), end
```

I aby otrzymać wielkość y można napisać (stosując zapis wktorowy[42]) :

```
y = sum( (occ - m*p).^2 ./ (m*p) )
```

pod warunkiem, że `p` *(wektor przwdopodobieństw rozkładu $\mathcal{L}$), będzie tych samych wymiarów co* `occ` *(tutaj wektor kolumnowy ??????????).*
Uwagi :

- *przybliżanie rozkładem χ^2 jest ważne gdy m jest dostatecznie duże... żadamy często $mp_{min} > 5$ ($p_{min} = \min_i p_i$) jako minimalny warunek zastosowania tego testu; w ten sposób możecie weryfikować to założenie i napisać wiadomość aby uprzedzić użytkownika jeżeli nie jest spełnione.*
- *można łatwo pogrupować te obliczenia w funkcji;*
- *dla rozkładu ciągłego test może się stosować grupując wielkości na przedziały, na przykład dla rozkładu $U_{[0,1]}$, stosuje się n równomiernie rozłożonych przedziałów; podobnie dla rozkładu dyskretnego o nieskończonej liczbie wielkości, można pogrupować je w kolejki rozkładu; podobną procedurę można zastosować dla roskładów skończonych dla których warynek zastosowania testu nie jest spełniony.*
- *jeżeli testujecie rozkład w którym pewne parametry zostały wcześniej obliczone na podstawie danych, należy określić liczbę stopni swobody dla rozkładu χ^2 ; na przykład jeśli spodziewamy się rozkładu $B(n-1,p)$ i użyjemy $p = \bar{x}_m/(n-1)$ wówczas nieprzekraczalną wartością progową dla testowanej hipotezy zerowej będzie $y_\alpha = F^{-1}_{\chi^2_{n-2}}(1-\alpha)$ a nie $y_\alpha = F^{-1}_{\chi^2_{n-1}}(1-\alpha)$.*

[42] ćwiczenie: przekształć tę instrukcję wektorową aby zrozumieć jak (i dlaczego) działa.

5.4.5 Test Kołmogorowa-Smirnova

Niech X będzie rzeczywistą zmienną losową dla której dystrybuanta rozkładu jest funkcją ciągłą F oraz X_1, X_2,..., X_m, m niezależnych zmiennych losowych. Dla realizacji X_i (mówimy wektor $x^m = (x_1, ..., x_m)$), można skonstruować dystrybuantę empiryczną która przy ($m \to +\infty$) dąży do dystrybuanty teoretycznej. Test KS polega na określeniu różnicy pomiędzy dystrybuantą teoretyczną i empiryczną (otrzymaną na podstawie naszej próby $(x_1, ..., x_m)$) w następujący sposób :

$$k_m = \sqrt{m} \sup_{-\infty<x<+\infty} |F(x) - F_{x^m}(x)|$$

i na porównaniu jej z wielkością dopuszczalną . Jeżeli zastąpimy naszą realizacę przez odpowiednie zmienne losowe, wówczas k_m staje się również zmienną losową (którą zapisujemy jako K_m). Zgodnie z teorią jej dystrybuanta jej rozkładu jest następująca :

$$\lim_{m\to+\infty} P(K_m \leq x) = H(x) = 1 - 2\sum_{j=1}^{+\infty}(-1)^{j-1}e^{-2j^2x^2}$$

Jak przy teście χ^2, jeżeli rozważana hipoteza jest fałszywa, otrzymane wartości k_m będą duże i odrzycimy tę hipoteze gdy:

$$k_m > H^{-1}(1-\alpha)$$

z $\alpha = 0.05$ lub 0.01 na przykład. Jeśli używamy przybliżenia $H(x) \simeq 1 - 2e^{-2x^2}$ wówczas nieprzekraczalna wartość progowa jest :

$$k_{seuil} = \sqrt{\frac{1}{2}ln(\frac{2}{\alpha})}$$

Obliczenie k_m nie sprawia problemu jeżeli posortuje się wektor $(x_1,\ x_2,\ \ldots,\ x_m)$. Sortowaniw będzie efektywne gdy zwrócimy uwagę ne fakt aby :

$$\sup_{x\in[x_i,x_{i+1}[} F_{x^m}(x) - F(x) = \frac{i}{m} - F(x_i),\ et \sup_{x\in[x_i,x_{i+1}[} F(x) - F_{x^m}(x) = F(x_{i+1}) - \frac{i}{m}$$

dwie następne wielkości można łatwo policzyć :

$$k_m^+ = \sqrt{m} \sup_{-\infty<x<+\infty} (F_{x^m}(x) - F(x)) = \sqrt{m} \max_{1\leq j\leq m} (\frac{j}{m} - F(x_j))$$

$$k_m^- = \sqrt{m} \sup_{-\infty<x<+\infty} (F(x) - F_{x^m}(x)) = \sqrt{m} \max_{1\leq j\leq m} (F(x_j) - \frac{j-1}{m})$$

i otrzymujemy w ten sposób $k_m = \max(k_m^+,\ k_m^-)$.

5.4.6 Ćwiczenia

Czy to sprawka kostki ?

Rzucamy 200 razy symetryczną kostą do gry, doświadczenie jest następujące : rzucamy tyle razy aż wypadnie 1 (ale nie więcej niż 10 rzutów). Otrzymaliśmy następujące wyniki :

liczba rzutów	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	≥ 11
ilość doświadczeń	*36*	*25*	*26*	*27*	*12*	*12*	*8*	*7*	*8*	*9*	*30*

na przykład 36 razy jedynka pojawiła sie w pierwszym rzycie, 25 razy jedynka pojawiła sie w drugim żucie, itd...

Wykonać test χ^2 aby udzielić odpowiedzi na postawione pytanie.

Urna Polya

Losujemy N razy z urny Polya *zawierającej początkowo r kul czerwonych i v kul zielonych. Każde losowanie polega na wyciągnięciu jednej kuli i jej odłożeniu spowrotem do urny wraz z c kulami tego samego koloru. Przez X_k oznaczamy stosunek kul zielonych po k losowaniach do sumy kul, V_k oznacza ilość kul zielonych :*

$$X_0 = \frac{v}{v+r}, V_0 = v.$$

Jeżeli $v = r = c = 1$, wynik będzie następujący :

1. $E(X_N) = E(X_0) = X_0 = 1/2$;
2. X_N *ma rozkład normalny na zbiorze* $\{\frac{1}{N+2}, ..., \frac{N+1}{N+2}\}$;

3. *dla $N \to +\infty$, X_N jest zbieżne do rozkładu jednostajnego na przedziale $[0,1)$.*

Wykożystacie symulację aby zilustrować dwa pierwsze wyniki.

1. *Aby przeprowadzić różne symulacje, można napisać funkcję zależną od parametru N i która wykonuje N kolejnych losowań. Funkcja ta zwraca więc X_N i V_N :*

```
function [XN, VN] = Urna_Polya(N)
   // symulacja N losowan z "Urny Polya" :
   VN = 1 ; V_plus_R = 2 ; XN = 0.5
   for i=1:N
      u = rand()   // losowanie jednej kuli
      V_plus_R = V_plus_R + 1   // zwieksza ilosc kul
      if (u <= XN) then  // wylosowalislmy kule zielona : mamy XN prawdopodobienstwo
                         // wylosowania kuli zielonej (1 - XN dla kuli czerwonej)
         VN = VN + 1
      end
      XN = VN / V_plus_R // aktualizujemy stosunek kul zielonych
   end
endfunction
```

   niestety skuteczne wykonanie statystyk wymaga wielokrotnego wywołania tej funkcji, a zatem, z uwagi na stosunkowo powolne wykonywanie iteracji przez Scilab, należy naposać funkcję wykonującą m procesów równoległych . Można użyć funkcji `find` *aby naprawiś????? urny z których losujemy kule zielone.*

2. *Napisać skrypt znajdujący poprzez symulacę wartość oczekiwaną (wraz z jej przedziełem ufności).*

3. *Rozwinąć poprzedni skrypt testując hipotezę H dotyczącą rozkładu zmiennej losowej X_N $H : X_N$ ma rozkład jednostajny na zbiorze $\{\frac{1}{N+2}, ..., \frac{N+1}{N+2}\}$ za pomocą testu χ^2.*

4. *Porównać wyniki graficznie, na przykład rysując na jednym rysunku dystrybuantę empiryczną i teoretyczną, lub też narysować wykres funkcji gęstości rozkładu χ^2 dla otrzymanych wielkości przez ten test, wielkości progowe zaznaczyć liniami pionowymi.*

Most ?????
Proces stochastyczny (w którym U oznacza rozkład jednostajny na przedziale $[0,1]$) :

$$X_n(t) = \frac{1}{\sqrt{n}} \sum_{i=1}^{n} (1_{\{U_i \leq t\}} - t)$$

jest taki, że dla ustalonego $t \in]0,1[$ mamy :

$$\lim_{n \to +\infty} X_n(t) = Y(t) \sim \mathcal{N}(0, t(1-t))$$

Chcemy zilustrować wyniki za pomocą symulacji.
Schemat pracy :

1. *Napisać funkcję Scilaba* `function [X] = pont_brownien(t,n)` *pozwalającą otrzymać realizację $X_n(t)$; w ciągu wywołamy tę funkcję m razy z wartością n dostatecznie dużym*[43] *należy zatem napisać ją bez użycia pentli.*

2. *Napisać sktypt scilaba wykożystując tę symulację mostu Brownien : wykonać m symulacji $X_n(t)$ (z dużym n) i przedstawić graficznie zachowanie rozkładu (rysując jego dystrybuantę empiryczną i nakładając na nią dystrybuantę teoretyczną $Y(t)$) a następnie zastosować test Kołmogorowa-Smirnova. Można umieścić poprzednią funkcję na początku skryptu aby pracować tylko z jednym plikiem.*

Uwaga *: nie jest łatwo dobrać odpowiednie wielkości dla m i n : kiedy m wierzy ...?????????????.....gdzie test się nie powiódł ponieważ uznał, że n nie jest dostatecznie duże (ponieważ rozkład normalny otrzymuje się z granicy gdy $n \to +\infty$), należy zatem zwiększyć n... Dla $t = 0.3$, możecie na przykład przetestować z $n = 1000$ i $m = 200, 500, 1000$, i test daje dobre wyniki (żadko odżuca hipotezę zerową) ale dla $m = 10000$ test jest prawie zawsze negatywny. Jeżeli zwiększymy n (na przykład do $n = 10000$ ale obliczenia zaczynają być nieco dłuższe na komputerze PC wykonanym w 2003) wówczas test ponownie staje się normalnie pozytywny.*

[43] aby przybliżyć $Y(t)$!

6 Zbiór drobiazgów

W tej części przedstawiono/przytoczono przegląd niektórych błędów często spotykanych w Scilabie...

6.1 Definiowanie wektora i macierzy *współczynnik po współczynniku*

Ten błąd jest jednym z najczęstrzych. Rozważmy następujący skrypt :

```
K = 100  // jedyny parametr w tym skrypcie
for k=1:K
   x(k) = cos
   y(k) = cos innego
end
plot(x,y)
```

Jeżeli uruchomimy ten skrypt po raz pierwszy, zostaną zdefiniowane dwa wektory x *i* y*. Pojawia się jeden mały błąd ponieważ w każdej iteracji Scilab redefiniuje rozmiary tych wektorów (nie wie, że ich wymiarem końcowym będzie (K,1)). Zauważmy również, że domyślnie stworzy wektor kolumnowy. Podczas drugiero wywołania (zmieniając parametr* K*...) wektory* x *i* y *są znane i takie, że* k *jest mniejsze od 100 (początkowa wielkość parametru K) zatem zmieniane są równeież ich współrzędne. W konsekwencji jeżeli nowa wartość* K *jest taka, że :*

- `K < 100` *wówczas nasze wektory* x *i* y *mają zawsze 100 współrzędnych (zmodyfikowane jest tylko* K *pierwszych) a wykres nie pokazuje tego co oczekujemy ;*
- `K > 100` *nie pojawia się żaden problem (za wyjątkiem tego, że rozmiar wektora jest za każdym razem aktualizowany począwszy od 101 iteracji)*

Najlepszym sposobem jest zdefiniowanie wektorów x *i* y *za pomocą inicjalizacji ich rodzaju :*

```
x = zeros(K,1) ; y = zeros(K,1)
```

w ten sposób uniknie się wszelkich błędów. Nasz skrypt zatem przyjmie postać :

```
K = 100  // jedyny parametr w tym skrypcie
x = zeros(K,1); y = zeros(K,1);
for k=1:K
   x(k) = cos
   y(k) = cos innego
end
plot(x,y)
```

6.2 Na temat wartości zwracanych przez funkcję

Załóżmy, że mamy zaimplementowaną funkcję w Scilabie zwracającą dwa argumenty, na przykład :

```
function [x1,x2] = resol(a,b,c)
   // rozwi\k{a}zanie r\'ownania kwadratowego a x^2 + b x + c = 0
   // formu\l{}ad poprawiona dla wi\k{e}kszo\'sci metod numerycznych
   // (w celu unikni\k{e}cia odejmowania dw\'och s\k{a}siednich liczb)
   if (a == 0) then
      error(" nie rozwa\.zamy przypadku gdy a=0 !")
   else
      delta = b^2 - 4*a*c
      if (delta < 0) then
         error(" nie rozwa\.zamy przypadku gdy delta < 0 ")
      else
         if (b < 0) then
            x1 = (-b + sqrt(delta))/(2*a) ; x2 = c/(a*x1)
         else
            x2 = (-b - sqrt(delta))/(2*a) ; x1 = c/(a*x2)
         end
      end
   end
endfunction
```

W ogólnym przypadku jeżeli wywołamy funkcję w Scilab-ie w następujący sposób :

```
-->resol(1.e-08, 0.8, 1.e-08)
 ans  =
   - 1.250D-08
```

jej wynik jest przypisany zmiennej `ans`. *Ale zmienna ta jest pojedynczą liczbą, a funkcja zwraca dwie wartości z których tylko pierwsza jast przypisana zmienne* `ans`. *Aby zobaczyć drugą wartość użyjemy składni :*

```
-->[x1,x2] = resol(1.e-08, 0.8, 1.e-08)
 x2  =
   - 80000000.
 x1  =
   - 1.250D-08
```

Inna, niebezpieczniejsza pułapka jest następująca. Załóżmy, że znamy precyzję z jaką działa ta formuła (w stosunku do precyzji działania formół klasycznych). Aby dobrać wartości a i c (na przykład $a = c = 10^{-k}$ przyjmując różne warotści k) obliczymy wszystkie pierwiastki dla kolejnych parametrów równania i ustawimy je jako dwa wektory służce do późniejszej analizy. Najprostrzy schemat jest następujący :

```
b = 0.8;
kmax = 20;
k = 1:kmax;
x1 = zeros(kmax,1); x2=zeros(kmax,1);
for i = 1:kmax
   a = 10^(-k(i)); // c = a
   [x1(i), x2(i)] = resol(a, b, a); // BLAD !
end
```

Taka składnia nie zadziała (jedynie w przypadku gdy funkcja zwraca jedną wartość). Należy wykonać to w dwóch krokach :

```
[rac1, rac2] = resol(a, b, a);
x1(i) = rac1; x2(i) = rac2;
```

Uwaga *: Problem ten jest rozwiązany w wercjach rozwijających (cvs) dla Scilaba.*

6.3 Funkcja została zmidyfikowana ale...

wszystko działa tak jak przed modyfikacją ! Być może zapomnieliście zapisać zmiany w waszym edytorze albo, co jest bardziej prawdopodobne, zapomnieliście załadować plik zawierający funkcję w Scilabie za pomocą instrukcji `getf` *(lub* `exec`*) ! Jedna mała wskazówka : instrukcja* `getf` *(lub* `exec`*) jest z pewnoącią zapisana niedaleko w historii komend, wystarczy zatem za pomocą strzałki ↑ odszukać tę instrukcję.*

6.4 Problem z `rand`

Domyślnie funkcja `rand` *dostarcza liczby losowe według rozkładu jednostajnego na przedziale $[0, 1[$, można jednak otrzymać rozkład normalny $\mathcal{N}(0, 1)$ za pomocą :* `rand(ńormal")`. *Chcąc powrócić do rozkładu jednostajnego należy wpisać instrukcję* `rand("uniform")`. *Aby uniknąć tego problemu najlepiej pamiętać aby każde wywołanie funkcji* `rand`. *precyzowało rozkład który nas aktualnie interesuje (patrz poprzedni rozdział).*

6.5 Wektory wierszowe, wektory kolumnowe...

W kontekście macierzowym ich postać jest sprecyzowana, ale dla innych zastosowań nie zawsze są one rozrózniane i stosuje sie funkcję która działa w obydwu przypadkach. Aby przyspieszyć obliczenia dobrze jest odwołać się do skłakni macierzowej i wybrać jedną lub drugą formę wektora. Można wykożystać funkcję `matrix` *w następujący sposób :*

```
x = matrix(x,1,length(x)) // aby otrzyma\'c wektor wierszowy
x = matrix(x,length(x),1) // aby otrzyma\'c wektor kolumnowy
```

Chcąc otrzymać w prosty sposób wektor kolumnowy można wykośystać instrukcję :

```
x = x(:)
```

6.6 Operator porównania

W pewnyc przypadkach Scilab akceptuje symbol = *jako operator porównania :*

```
-->2 = 1
 Warning: obsolete use of = instead of ==
    !
 ans  =
   F
```

ale lepiej jest zawsze używać symbolu ==*.*

6.7 Liczby zespolone a liczby rzeczywiste

Scilab jest tak skonstruowany aby traktować liczby rzeczywiste w taki sam sposób jak liczby zespolone ! Jest to calkiem praktyczne ale może niekiedy być przyczyną niespodzianek, na przykład podczas szacowania funkcji rzeczywistej poza jej dziedziną (na przykład $\sqrt{x}$ *i* $\log(x)$ *dla* $x < 0$*, acos*(x) *i asin*(x) *dla* $x \notin [-1, 1]$*, acosh*(x) *dla* $x < 1$*) ponieważ Scilab zwraca wówczas oszacowanie części zespolonej tej funkcji. Aby dowiedzieć się czy działamy na zmiennej zespolonej czy rzeczywistej należy użyć funkcji* `isreal` *:*

```
-->x = 1
 x  =
    1.

-->isreal(x)
 ans  =
  T

-->c = 1 + %i
 c  =
    1. + i

-->isreal(c)
 ans  =
  F

-->c = 1 + 0*%i
 c  =
    1.

-->isreal(c)
 ans  =
  F
```

6.8 Proste instrukcje a funkcje Scilaba

W niniejszym opracowaniu częstu stosuje sie zamiennie terminów prosta instrukcja *i* funkcja *aby wskazać procedury dostępne w bieżącej werksji Scilaba. Istnieje jednak fundamentalna różnica pomiędzy prostą instrukcją która jest kodowana w fortranie 77 lub w C a funkcją (zwaną również makro) która jest kodowana w języku Scilaba : funkcja jest wówczqs rozumiana jako zmienna Scilaba, i dzięki temu można ją użyć jako argumentu dla inne funkcji. Począwszy od wersji 2.7, proste instrukcje bywają zmiennymi w Scilabie (fptr). Niemniej jednak przysparzają wiele problemów (aktualnie rozwiązanych w rozwinięciach).*

Oto przykład problemu jaki można spotkać : funkcja następująca pozwalająca przybliżać zbiór przy zastosowaniu metody Monte Carlo *:*

```
function [im,sm]=MonteCarlo(a,b,f,m)
   //               /b
   //  przybli\.zenie |  f(x) dx przy zastosowaniu metody Monte Carlo
   //               /a
   //  zwracane jest r\'ownie\.z empiryczne odchylenie standardowe
   xm = grand(m,1,"unf",a,b)
   ym = f(xm)
```

```
    im = (b-a)*mean(ym)
    sm = (b-a)*st_deviation(ym)
endfunction
```

Jako argument f oczekuje się funkcji Scilaba ale ten kod powienien również działać dla funkcji matematycznych które należą do prostych instrukcji Scilaba[44] *ponieważ są one teraz rozważane jako zmienne. Niemniej jednak test z użyciem prostej instrukcji* `exp` *nie powiedzie się :*

```
-->[I,sigma]=MonteCarlo(0,1,exp,10000)  // bug !
```

Wywołując funkcję `MonteCarlo` *za pomocą* `getf` *z opcją nie kompilowania :*

```
-->getf("MonteCarlo.sci","n")
```

błąd ten [bug] (skorygowany w aktualnym cvs) będzie wówczas ominięty.

6.9 Obliczanie wyrażeń logicznych

W przeciwieństwie do języka C, obliczenie wartości logicznej wyrażeń :

$$a \text{ lub } b$$
$$a \text{ i } b$$

poprzedzone jest wyliczeniem wartości logicznej a i b, dopiero potem zostanie wykonana na nich operacja sumy czy iloczynu logicznego (Priorytety operatorów zamieszczono w rozdziale Programowanie).

A Odpowiedzi do ćwiczeń z rozdziału 2

1. ```
   --> n  = 5 // dla ustalenia warto\'sci n...
   --> A = 2*eye(n,n) - diag(ones(n-1,1),1) - diag(ones(n-1,1),-1)
   ```
   Szybszym sposobem jest użycie funkcji `toeplitz` *:*

   ```
   -->n=5; // pour  fixer une valeur a n...

   -->toeplitz([2 -1 zeros(1,n-2)])
   ```

2. *Jeżeli A jest macierzą (n, n),* `diag(A)` *zwróci wektor kolumnowy zawierający elementy diagonalne macierzy A (a zatem otrzymamy wektor kolumnowy o wymiarze n).*
   tt diag(diag(A)) zwróci macierz kwadratową diagonalną o wymiarze n, w której elementy diagonalne będą takie jak w macierzy wyjściowej.

3. *Oto jedna z możliwości :*

   ```
   --> A = rand(5,5)
   --> T = tril(A) - diag(diag(A)) + eye(A)
   ```

4. *(a)*
   ```
   --> Y = 2*X.^2 - 3*X + ones(X)
   --> Y = 2*X.^2 - 3*X + 1   // wyko\.zystuj\k{a}c zapis skr\'ocony
   --> Y = 1  + X.*(-3 + 2*X) // oraz schemat Hornera
   ```

   (b)
   ```
   --> Y = abs(1 + X.*(-3 + 2*X))
   ```

   (c)
   ```
   --> Y = (X - 1).*(X + 4)  // wyko\.zystuj\k{a}c zapis skr\'ocony
   ```

   (d)
   ```
   --> Y = ones(X)./(ones(X) + X.^2)
   --> Y = (1)./(1 + X.^2) // z skr\'otami
   ```

5. *Oto skrypt :*

[44] na przykład `sin`, `exp` i wiele innych.
   ```

```
n = 101;                          // dyskretyzacja
x = linspace(0,4*%pi,n);
y  =  [1  , sin(x(2:n))./x(2:n)]; // aby zapobiec dzieleniu przez zero...
plot(x,y,"x","y","y=sin(x)/x")
```

6. *Możliwe rozwiązanie (dla różnych wartości n) :*

```
n = 2000;
x = rand(1,n);
xbar = cumsum(x)./(1:n);
plot(1:n, xbar, "n","xbar", ...
    "ilustracja lgn : xbar(n) -> 0.5 qd n -> + oo")
```

B Rozwiązania ćwiczeń z rozdziału 3

1. *Klasyczny algorytm wykożystuący dwie pętle :*

```
function  [x] =  sol_tri_sup1(U,b)
  //
  //  rozwi\k{a}zanie  Ux  =   b gdzie U jest macierz\k{a} tr\'ojk\k{a}tn\k{a} g\'orn\k{a
  //
  [n,m] =  size(U)
  // kilka wyj\k{a}tk\'ow ....
  if  n  ~=  m then
     error('  Macierz nie jest kwadratowa')
  end
  [p,q] =  size(b)
  if p ~=  m then
     error(' Wyraz wolny ma nieprawid\l{}owy wymiar')
  end
  // pocz\k{a}tek algorytmu
  x = zeros(b) // rezerwuje miejsce dla x
  for i = n:-1:1
     suma = b(i,:)
     for j =  i+1:n
        suma = suma -  U(i,j)*x(j,:)
     end
     if U(i,i) ~= 0 then
        x(i,:) = suma/U(i,i)
     else
        error(' Macierz jest nieodwracalna')
     end
  end
endfunction
```

A oto wersja wykorzystująca pojedyńczą pętlę

```
function [x] = sol_tri_sup2(U,b)
  //
  //
  [n,m] = size(U)
  //  niekt\'ore wyj\k{a}tki ....
  if  n ~= m then
     error('  Macierz nie jest kwadratowa')
  end
```

```
  [p,q] = size(b)
  if p ~= m then
     error(' Wektor wyraz\'ow wolnych ma z\l{}y wymiar)
  end
  // pocz\k{a}tek algorytmu
  x = zeros(b) // rezerwuje miejsce dla x
  for i = n:-1:1
     suma = b(i,:) - U(i,i+1:n)*x(i+1:n,:) // zobacz komentarz ko\'ncowy
     if U(i,i) ~= 0 then
        x(i,:) = suma/U(i,i)
     else
        error(' Nie mo\.zna odwr\'oci\'c macierzy')
     end
  end
endfunction
```

Komentarz : w pierwszej iteracji (dla $i = n$) macierze `U(i,i+1:n)` *et* `x(i+1:n,:)` *są puste. Odpowiadają one obiektom zdefiniowanym w Scilabie (macierz pusta), które oznaczone są przez* `[]`*. Dodawanie macierzy pustej jest zdefiniowane i daje :* `A = A + []`*. Zatem w pierwszej iteracji mamy* `suma =  b(n,:) + []` *to znaczy* `suma = b(n,:)`.

2.
```
//
// skrypt rozwiazujacy x" + alpha*x' + k*x = 0
//
// Aby przeksztalcic rownanie do postaci ukladu rownan pierwszego rzedu
// kladziemy  : X(1,t) =  x(t) i X(2,t) = x'(t)
//
// Otrzymujemy wowczas : X'(t) = A X(t) z  :  A  = [0 1;-k -alpha]
//
k =  1;
alpha =  0.1;
T = 20;    //  moment koncowy
n   = 100; //  dyskretyzacja czasowa :   przedzial  [0,T] zostanie
           //  podzielony na n przedzialow
t  = linspace(0,T,n+1); // momenty czasowe : X(:,i) bedzie korespondowal z  X(:,t(i))
dt = T/n;  // zmiana czasu
A = [0 1;-k -alpha];
X = zeros(2,n+1);
X(:,1) = [1;1]; // warunki poczatkowe
M = expm(A*dt); // oblicza exponent A dt

// obliczenia
for i=2:n+1
  X(:,i) =  M*X(:,i-1);
end

// wyswitlenie rezultat\'ow
xset("window",0)
xbasc()
xselect()
plot(t,X(1,:),'czas','pozycja','Courbe x(t)')
xset("window",1)
xbasc()
xselect()
plot(X(1,:),X(2,:),'pozycja','predkosc','Trajektoria w przestrzeni stanow')
```

3.
```
function [i,info]=przedzial(t,x)
   // poszukuje dychotomii przedzialu i takiej, ze: x(i)<=  t <= x(i+1)
   // jezeli t nie jest z przedzialu [x(1),x(n)] zwraca info = %f
   n=length(x)
   if  t < x(1) |  t > x(n)  then
```

```
      info = %f
      i = 0 // kladziemy wartosc domyslna
   else
      info = %t
      i_dolne=1
      i_gorne=n
      while  i_gorne - i_dolne > 1
         itest = floor((i_gorne + i_dolne)/2 )
         if ( t >= x(itest) ) then, i_dolne= itest, else, i_gorne=itest, end
      end
      i=i_dolne
   end
endfunction
```

4.
```
function [p]=myhorner(t,x,c)
   // rozwiniecie wielomianu c(1) +  c(2)*(t-x(1)) +  c(3)*(t-x(1))*(t-x(2)) + ...
   // algorytmem Hornera
   // t jest wektorem (lub macierza) moment\'ow czasu
   n=length(c)
   p=c(n)*ones(t)
   for k=n-1:-1:1
      p=c(k)+(t-x(k)).*p
   end
endfunction
```

5. *Rozwinięcie w szereg Fouriera :*

```
function [y]=signal_fourier(t,T,cs)

  // ta funkcja zwraca T-okresowy sygnal
  // t  : wektor momentow dla kt\'orych obliczmy
  //      sygnal y ( y(i) odpowiadaj\k{a}cy t(i) )
  // T  : okres sygnalu
  // cs : jest wektorem ktory wskazuje amplitude kazdej funkcji f(i,t,T)

  l=length(cs)
  y=zeros(t)
  for j=1:l
     y=y + cs(j)*f(j,t,T)
  end
endfunction

function [y]=f(i,t,T)

  // wielomiany trygonometryczne dla sygna\l{}u o okresie T :

  // jesli i jest parzyste   : f(i)(t)=sin(2*pi*k*t/T)  (z k=i/2)
  // jesli i jest nieparzyste : f(i)(t)=cos(2*pi*k*t/T)  (z k=floor(i/2))
  // stad w szczeg\'olnosci  f(1)(t)=1 jest traktowany ponizej
  // jako przypadek szczegolny i niezbyt wa\.zny
  // t jest wektorem mament\'ow czasowych

  if i==1 then
     y=ones(t)
  else
     k=floor(i/2)
     if modulo(i,2)==0 then
        y=sin(2*%pi*k*t/T)
     else
        y=cos(2*%pi*k*t/T)
```

```
      end
    end
  endfunction
```

6. *Spotkanie : niech T_A i T_B będą momentami przybycia Pana A i Pani B. Są to dwie zmienne losowe niezależne o rozkładzie $U([17,18])$ a spotkanie ma miejsce jezeli $[T_A, T_A + 1/6] \cup [T_B, T_B + 1/12] \neq \emptyset$. Doswiadczenie spotkania odpowiada realizacji zmienne losowej wektorowej $R = (T_A, T_B)$ która, zgodnie z hipotezami, ma rozkład zero-jedynkowy na kwadracie $[17,18] \times [17,18]$. Prawdopodobieństwo spotkania wyznacza sie więc w oparciu o zależność pomiędzy powieżchnią obszaru (odpowiadającą pojedynczemu spotkaniu) zdefiniowaną jako:*

$$\begin{cases} T_A \leq T_B + 1/12 \\ T_B \leq T_A + 1/6 \\ T_A, T_B \in [17,18] \end{cases}$$

oraz powieżchnią kwadratową (1), przez co otrzymuje sie $p = 67/288$. Aby obliczyć to prawdopodobieństwo, można przyjąć ze spotkanie powinno miec miejsce pomiędzy połódniem a godziną pierwszą. Poprzez symulację otrzymuje sie m doświadczeń a prawdopodobieństwo empiryczne jest liczbą przypadków w których spotkanie ma miejce podzieloną przez m. A oto rozwiązanie :

```
function [p] = rdv(m)
   tA = rand(m,1)
   tB = rand(m,1)
   spotkanie = tA+1/6 > tB  &  tB+1/12 > tA;
   p = sum(bool2s(spotkanie))/m
endfunction
```

można także wykożystać funkcje `grand` *opisaną w rozdziale dotyczącym zastosowań :*

```
function [p] = rdv(m)
   tA = grand(m,1,"unf",17,18)
   tB = grand(m,1,"unf",17,18)
   spotkanie = tA+1/6 > tB  &  tB+1/12 > tA;
   p = sum(bool2s(spotkanie))/m
endfunction
```

C Rozwiązania ćwiczeń z rozdziału 4

Czy to sprawka kostki?

Jeżeli przez J oznaczymy zmienną losową będącą wynikiem doświadczenia, wówczas J ma rozkład geometryczny $G(p)$ gdzie $p = 1/6$ przy założeniu, że kostka jest symetryczna. W ten sposób możemy określić prawdopodobieństwo poszczególnych zmiennych losowych jako (przymujemy $q = 1 - p = 5/6$) :

$$P(J=1) = p,\ P(J=2) = qp,\ P(J=3) = q^2p,\ ....,\ P(J=10) = q^9p,\ P(J>10) = q^{10}$$

W drugiej części tabeli podano częstość występowania poszczególnych zmiennych losowych, co w skrypcie zapiszemy następująco :

```
occ= [36 ; 25 ; 26 ; 27 ; 12 ; 12 ; 8 ; 7 ; 8 ; 9 ; 30];
p = 1/6; q = 5/6;
pr = [p*q.^(0:9) , q^10]';
y = sum( (occ - m*pr).^2 ./ (m*pr) );
y_progowy = cdfchi("X", 10, 0.95, 0.05);
mprintf("\n\r Test dla :")
mprintf("\n\r y = %g, et y_progowy (95 \%) = %g", y, y_progowy)
mprintf("\n\r 200*min(pr) = %g", 200*min(pr))
```

Otrzmujemy $y = 7.73915$ gdy $y_{seuil} = 18.307$, zatem kostka wydaje się być prawidłowa. Niemniej jednak mamy $200 \times \min(pr) \simeq 6.5$ co jest znacznie poniżej warunku pozwalającego zastosować test (należałoby wykonać jeszcze kilka dodatkowych doświadczeń).

Urna Polya

***Funkcja scilaba*

```
   function [XN, VN] = Urna_Polya_rownolegla(N,m)
      VN = ones(m,1) ; V_plus_R = 2 ; XN = 0.5*ones(m,1)
      for i=1:N
         u = grand(m,1,"de"f)     // losowanie jednej kuli
         V_plus_R = V_plus_R + 1  // dodaje kule
         ind = find(u <= XN)      // znajduje numer urny z kt\'orej
                                  // wylosowano kule zielona
         VN(ind) = VN(ind) + 1    // zwiekszamy liczbe kul zielonych w tej urnie
         XN = VN / V_plus_R       // aktualizujemy proporcje kul zielonych
      end
   endfunction
```

***Skrypt Scilaba*

```
// symulacja polya :
N = 10;
m = 5000;
[XN, VN] = Urna_Polya_rownolegla(N,m);

// 1/ obliczenie wartosci oczekiwanej i przedzialu ufnosci
EN = mean(XN);            // wartosc oczekiwana empiryczna
sigma = st_deviation(XN); // odchylenie standardowe empiryczne
delta =  2*sigma/sqrt(m); // zaokraglenie dla przedzialu empirycznego (2 dla 1.9599..)
mprintf("\n\r  E teoretyczne = 0.5");
mprintf("\n\r  E oszacowane = %g",EN);
mprintf("\n\r  Przedzial (empiryczny) ufnosci przy 95\% : [%g,%g]",EN-delta,EN+delta);

// 2/ test chi2
alpha = 0.05;
p = 1/(N+1);  //  prawdopodobienstwo kazdego wyniku (rozklad jednostajny)
occ = zeros(N+1,1);
for i=1:N+1
  occ(i) = sum(bool2s(XN == i/(N+2)));
end
if sum(occ) ~= m then, error(" Problem..."), end  // mala poprawka
Y = sum( (occ - m*p).^2 / (m*p) );
Y_seuil = cdfchi("X",N,1-alpha,alpha);
mprintf("\n\r Test chi 2 : ")
mprintf("\n\r ----------- ")
mprintf("\n\r wielkosc otrzymana w tescie : %g", Y);
mprintf("\n\r wielkosc progowa : %g", Y_progowe);
if (Y > Y_progowe) then
  mprintf("\n\r Wniosek : Hipoteza odrzucona !")
else
  mprintf("\n\r Wniosek : Hipoteza nie odrzucona !")
end

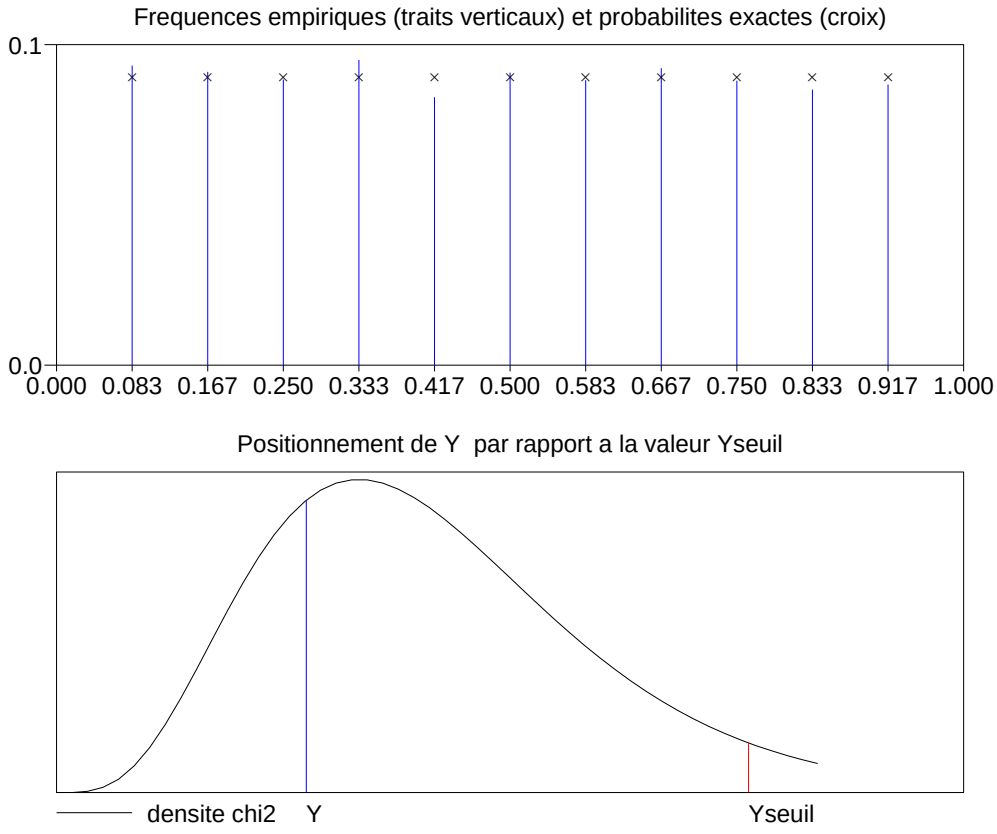
// 3/ ilustracja graficzna
function [d] = gestosc_chi2(X,N)
   d = X.^(N/2 - 1).*exp(-X/2)/(2^(N/2)*gamma(N/2))
endfunction
czestosc = occ/m;
ymax = max([czestosc ; p]); rect = [0 0 1 ymax*1.05];
xbasc(); xset("font",2,2)
subplot(2,1,1)
   plot2d3((1:N+1)'/(N+2), czestosc, style=2 , ...
   frameflag=1, rect=rect, nax=[0 N+2 0 1])
   plot2d((1:N+1)'/(N+2),p*ones(N+1,1), style=-2, strf="000")
   xtitle("Czestosci empiryczne (kreski pionowe) i prawdopodobienstwa teoretyczne (krzy\.zyki)")
subplot(2,1,2)
   // rysujemy g\k{e}sto\'s\'c chi2 ???a N ddl
   X = linspace(0,1.1*max([Y_seuil Y]),50)';
```

```
  D = densite_chi2(X,N);
  plot2d(X,D, style=1, leg="densite chi2", axesflag=2)
  // kreski pionowe dla Y
  plot2d3(Y, densite_chi2(Y,N), style=2, strf="000")
  xstring(Y,-0.01, "Y")
  // kreski pionowe dla Yseuil
  plot2d3(Y_seuil, densite_chi2(Y_seuil,N), style=5, strf="000")
  xstring(Y_seuil,-0.01, "Yseuil")
  xtitle("Pozycja Y  w odniesieniu do warto\'sci Yseuil")
```

Poniżej przedstawiono rezultaty otrzymane dla $N = 10$ *et* $m = 5000$ *(patrz wykres (24)) :*

```
E dok\l{}adne = 0.5
E oszacowane = 0.4959167
Przedzia\l{} ufno\'sci dla 95% : [0.4884901,0.5033433]

Test du chi 2 :
--------------
warto\'s\'c otrzymana w te\'scie : 8.3176
warto\'s\'c progowa : 18.307038
Wniosek : Hipoteza nie odrzucona !
```



polya

Rysunek 24: Ilustracja dla testu χ^2 urny Polya...

Most ?brownien?

***Funkcja*

```
function [X] = most_brownien(t,n)
   X = sum(bool2s(grand(n,1,"def") <= t) - t)/sqrt(n)
endfunction
```

***Skrypt** Skrypt ten wykonuje m symulacji zmiennej losowej $X_n(t)$, przedstawia wykres funkcji rozkładu empirycznego, nakładając na niego wykres funkcji rozkładu oczekiwanego (dla $n = +\infty$) i wreszcie oblicza test KS :*

```
t = 0.3;
sigma = sqrt(t*(1-t));  // oczekiwane odchylenie standardowe
n = 4000;               // n "du\.ze"
m = 2000;               // licczba symulacji
X = zeros(m,1);         // zainicjowanie wektora realizacji
for k=1:m
   X(k) = most_brownien(t,n);  // p\k{e}tla licz\k{a}ca kolejne realizacje
end

// wykres funkcji rozk\l{}adu empirycznego
X = - sort(-X); // tri
prcum = (1:m)'/m;
xbasc()
plot2d2(X, prcum)
x = linspace(min(X),max(X),60)';               // odci\k{e}te i
[P,Q]=cdfnor("PQ",x,0*ones(x),sigma*ones(x));  // rz\k{e}dne dla funkcji dok\l{}adenj
plot2d(x,P,style=2, strf="000")                // rzutujemy je na wykres pocz\k{a}tkowy

// zastosowanie testu KS
alpha = 0.05;
FX = cdfnor("PQ",X,0*ones(X),sigma*ones(X));
Dplus = max( (1:m)'/m - FX );
Dminus = max( FX - (0:m-1)'/m );
Km = sqrt(m)*max([Dplus ; Dminus]);
K_progowe = sqrt(0.5*log(2/alpha));

// prezentacja wynik\'ow
//
mprintf("\n\r Test KS : ")
mprintf("\n\r -------- ")
mprintf("\n\r  warto\'s\'c otrzymana w te\'scie : %g", Km);
mprintf("\n\r  warto\'s\'c progowa : %g",K_seuil);
if (Km > K_progowe) then
   mprintf("\n\r Wniosek : Hipoteza odrzycona !")
else
   mprintf("\n\r Wniosek : Hipoteza nie odrzucona !")
end
```

Oto rezultaty otrzymane dla $n = 1000$ i $m = 4000$:

```
Test KS :
--------
warto\'s\'c otrzymana w te\'sci : 1.1204036
warto\'s\'c progowa : 1.2212382
Wniosek : Hipoteza nie odrzucona !
```