# Wstęp do Maximy

## Materiały pomocnicze do wizualizacji obiektów matematycznych część I

Opracowanie: Lech Sławik

30 września 2011

# Spis treści

# Rozdział 1

# Interfejs wxMaxima

W tym notatniku dowiemy się jak używać wxMaxima, graficznego interfejsu do Maximy. Z Maximą można komunikować się na wiele sposobów. Od najprostszego, poprzez użycie konsoli tekstowej, do bardzo wyrafinowanych edytorsko, jak na przykład program TeXmacs. Obecnie największą popularnością cieszy się wxMaxima ze względu na zgrabne połączenie możliwości graficznych, tekstowych i obliczeniowych przy zachowaniu prostoty użytkowania. Dlatego jako narzędzie do pisania tych notatek wybraliśmy ten program i przy nim pozostaniemy w dalszej pracy.

## 1.1 Struktura dokumentu wxMaxima

Podobnie jak większość komercyjnych programów algebry komputerowej, wxMaxima realizuje koncepcję interaktywnego dokumentu matematycznego, w którym swobodnie można przeplatać tekstowe komentarze z obliczeniami i wykresami.

### 1.1.1 Komórki

Dokument wxMaxima składa sie z tzw. komórek (cells). Każda komórka ma nawias kwadratowy z lewej strony, który wskazuje jej poczatek i koniec. Komórki są różnych typów. Pierwsza komórka w tym dokumencie to komórka tytułu dokumentu (title cell), następna to komórka tekstowa (gdzie umieszczamy np. opisy), trzecia to komórka tytułu rozdziału

(section cell) i tak dalej. Najważnieszym typem komórki jest komórka obliczeniowa (input cell), w której wpisujemy polecenia do wykonania przez program Maxima. Każde polecenie Maximy powinno kończyć się:

znakiem średnika ';' - jeżeli chcemy zobaczyć wynik na ekranie lub znakiem dolara '$' - jeżeli z jakiś powodów nie chcemy aby wynik obliczeń pojawił się na ekranie.

Kliknięcie w dowolnym miejscu komórki obliczeniowej (powinien pojawić się migający kursor tekstowy), a następnie wciśnięcie klawiszy SHIFT-ENTER powoduje przesłanie kodu do Maximy w celu wykonania go. Przed wyslaniem wxMaxima sprawdza czy kod kończy się znakiem średnika ';' lub dolara '$' - jeżeli nie, dodaje automatycznie ';' na końcu. Tekst znajdujacy się między znakami /* oraz */ stanowi komentarz i jest ignorowany w trakcie obliczeń. Rezultat, który wxMaxima otrzymuje z Maximy jest dołaczany do części wynikowej komórki obliczeniowej. Kolejne komórki obliczeniowe i wynikowe są poprzedzane etykietami %i1, %i2,... oraz %o1,%o2,... W rzeczywistości te etykiety są zmiennymi, pod które podstawiana jest zawartość kolejnych komórek i do których można się odwoływać.

```
(%i1)  /*przykład polecenia Maximy: */
       integrate(x^2, x);
```

(%o1) $\frac{x^3}{3}$

```
(%i2)  integrate(x^2, x)$
```

Ponieważ na końcu jest $ wynik nie jest wyświetlany.

```
(%i3)  integrate(x^3, x);
```

(%o3) $\frac{x^4}{4}$

```
(%i4)  %o2;
```

(%o4) $\frac{x^3}{3}$

Mimo, że %o2 nie zostało wyświetlone, to zostało obliczone i zapamiętane.

### 1.1.2 Edycja komórki

Edytowanie komórki jest łatwe - aby zmodyfikować jej zawartość kliknij w miejscu, ktore chcesz edytować. Powinien pojawić się kursor oraz ograniczający komórkę nawias kwadratowy przyjmuje kolor czerwony. Zauważmy, że jeżeli przesuniemy kursor w obszar miedzy komórkami pojawia się pozioma linia (nazwijmy ją poziomym kursorem). Wciśnięcie dowolnego klawisza powoduje wprowadzenie nowej komórki obliczeniowej. Natomiast wciśnięcie SHIFT-UP/DOWN markuje komórki w wybranym kierunku. Pojedyńczą komorkę możemy zamarkować klikając na ograniczajacy ją nawias. Na zamarkowanych komórkach możemy wykonywać różne operacje (np. kopiowanie czy usuwanie) podobnie jak w innych edytorach tekstu. Proszę otworzyć pole menu "Edycja" i wykorzystując podane tam komendy (i ich skróty) przećwiczyć podstawowe operacje. Kiedy poziomy kursor jest aktywny bezpośrednia edycja powoduje pojawienie się nowej komórki obliczeniowej. Jeżeli chcemy w tym miejscu wstawić komórkę innego typu klikamy pole menu "Cell" i wybieramy jeden z dostępnych tam typów (zwróćmy uwagę na skróty klawiaturowe, które przyspieszają pracę).

### 1.1.3 Ukrywanie komórek

Czasami wynik obliczeń jest bardzo długi. Wykonajmy poniższy przykład (klikamy w komórkę i wciskamy SHIFT-ENTER).

```
(%i5)  expand( (a+b)^40 );
```

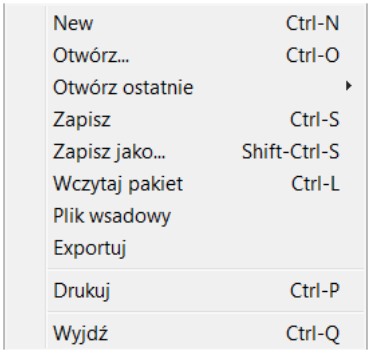
(%o5) $b^{40}+40\,a\,b^{39}+780\,a^2\,b^{38}+9880\,a^3\,b^{37}+91390\,a^4\,b^{36}+658008\,a^5\,b^{35}+3838380\,a^6\,b^{34}+18643560\,a^7\,b^{33}+76904685\,a^8\,b^{32}+273438880\,a^9\,b^{31}+847660528\,a^{10}\,b^{30}+2311801440\,a^{11}\,b^{29}+5586853480\,a^{12}\,b^{28}+12033222880\,a^{13}\,b^{27}+23206929840\,a^{14}\,b^{26}+40225345056\,a^{15}\,b^{25}+62852101650\,a^{16}\,b^{24}+88732378800\,a^{17}\,b^{23}+113380261800\,a^{18}\,b^{22}+131282408400\,a^{19}\,b^{21}+137846528820\,a^{20}\,b^{20}+131282408400\,a^{21}\,b^{19}+113380261800\,a^{22}\,b^{18}+88732378800\,a^{23}\,b^{17}+62852101650\,a^{24}\,b^{16}+40225345056\,a^{25}\,b^{15}+23206929840\,a^{26}\,b^{14}+12033222880\,a^{27}\,b^{13}+5586853480\,a^{28}\,b^{12}+2311801440\,a^{29}\,b^{11}+847660528\,a^{30}\,b^{10}+273438880\,a^{31}\,b^{9}+76904685\,a^{32}\,b^{8}+18643560\,a^{33}\,b^{7}+3838380\,a^{34}\,b^{6}+658008\,a^{35}\,b^{5}+91390\,a^{36}\,b^{4}+9880\,a^{37}\,b^{3}+780\,a^{38}\,b^{2}+40\,a^{39}\,b+a^{40}$

Jeżeli chcemy zaoszczędzić miejsce na ekranie ukrywając wynik (bez usuwania go) klikamy w pusty trójkąt znajdujący się u góry nawiasu

ograniczającego komórkę. Komórka zostaje zwinięta co sygnalizuje czarne wypełnienie trójkąta. Ponowne kliknięcie w trójkąt rozwija zawartość komórki. Podobny mechanizm jest zaimplementowany przy tytułach rozdziałów i podrozdziałów (rolę trójkąta przejmuje tu kwadrat). Jest to wygodne narzędzie przy porzadkowaniu materiału przedstawianego np. w ramach prezentacji.

## 1.2 Pole menu "Plik"

W polu "Plik" mamy dostęp do następujących poleceń (Maxima 5.25): Większość z nich występuje prawie w każdym edytorze i ich działanie

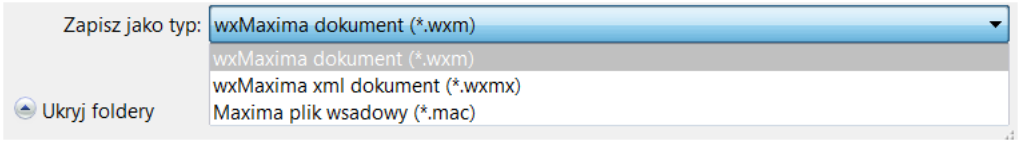



Rysunek 1.1:

jest dobrze znane. Ograniczymy się tu do omówienia elementów specyficznych dla wxMaximy.

### 1.2.1 Format zapisu dokumentu wxMaximy

Chcąc zachować dokument wxMaximy w formacie umożliwiającym ponowne otwarcie w wxMaxima wybieramy pozycję "Zapisz jako...". Po wpisaniu tytułu mamy do wyboru trzy typy dokumentu: Plik z rozszerzeniem *.wxm jest plikiem tekstowym, w którym zostaną zachowane jedynie wprowadzone polecenia i teksty (bez wyników i obrazków). Aby odtworzyć wyniki należy ponownie wykonać wszystkie komórki obliczeniowe. Zaletą tego typu jest nieduży rozmiar i mała wrażliwość na uszkodzenia.




Rysunek 1.2:

W pliku z rozszerzeniem *.wxmx zostaną zachowane wszystkie elementy dokumentu - polecenia, wyniki, teksty, wstawione i wytworzone obrazki. Z technicznego punktu widzenia jest to skompresowane archiwum zawierające pliki graficzne i obraz dokumentu zapisany w formacie xml.

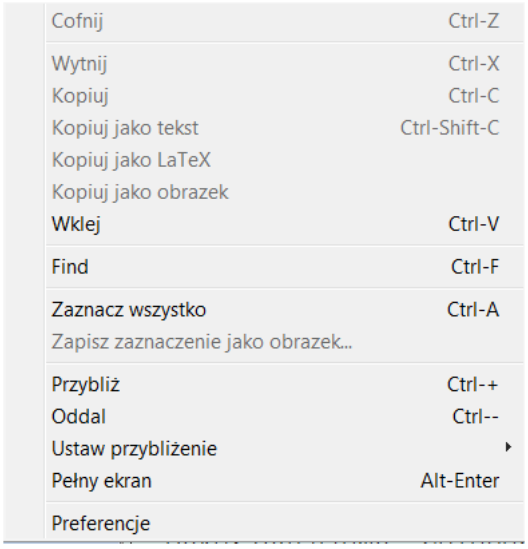
Plik z rozszerzeniem .mac to tak zwany plik wsadowy. Zawiera jedynie polecenia i komentarze, a ponowne jego otwarcie powoduje wykonanie zapisanych komend. Na obecnym etapie nie będziemy zajmować się plikami tego typu. Pozycje "Wczytaj pakiet", "Plik wsadowy" pozwalają na załadowanie plików *.mac, *.lisp zawierających, których poznanie odkładamy na później.

### 1.2.2 Eksport do html i pdfLaTeX

Jeżeli dokument jest w postaci finalnej i chcemy go opublikować, bądź na WWW, bądź w wersji drukowanej wykorzystujemy polecenie "Eksportuj", które pozwala zapisać dokument jako stronę WWW w formacie html lub plik w formacie pdfLaTeX.

## 1.3 Pole menu "Edycja"

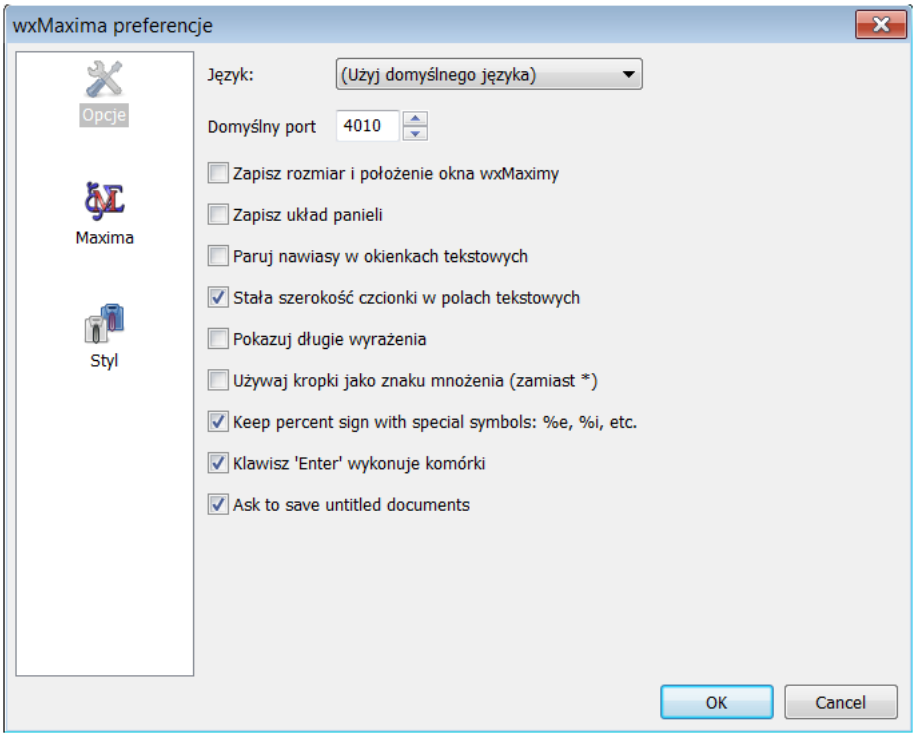
W polu "Edycja" mamy dostęp do następujących poleceń (Maxima 5.25): Tu również większość funkcji jest doskonale znana. Warto zwrócić uwagę na pozycję "Kopiuj jako LaTeX", która pomaga w wykorzystaniu wxMaximy w opracowywaniu dokumentu w systemie LaTeX. Temu tematowi poświęcimy osobny notatnik.




Rysunek 1.3:

### 1.3.1 Dostosowywanie programu wxMaxima

Ostatnia pozycja "Preferencje" daje możliwości dopasowania programu do własnych upodobań i potrzeb. Podkreślmy, że wszystkie dotychczasowe uwagi były pisane pod założeniem, że opcje dostępne w ramach preferencji mają wartość standardową. W zakładkach preferencji można te wartości zmienić. Przykładowo zaznaczając polecenie "Klawisz ’Enter’ wykonuje komórki" zmieniamy działanie klawisza Enter w komórkach obliczeniowych (chcąc przejść do nowej linii stosujemy wówczas kombinację Shift+Enter). Działanie większości opcji jest zrozumiałe z ich opisu dlatego ograniczamy się tylko wskazania pozycji, które mogłyby umknąć uwadze. Standardowo, przy otwieraniu nawiasu wxMaxima automatycznie wstawia nawias zamykający. Jeżeli nie chcemy tej funkcjonalności usuwamy zaznaczenie przy pozycji "Paruj nawiasy w okienkach tekstowych". Dla oszczędności miejsca wxMaxima wyświetla długie wyrażenia w postaci skróconej. Jeżeli chcemy widzieć pełną postać zaznaczamy pozycję "Pokazuj długie wyrażenia". Pozycja Preferencje ma trzy zakładki: Opcje, Maxima i Styl. Zakładką Maxima przy standardowej instalacji nie mamy potrzeby zajmować się. W zakładce Styl zwracamy uwagę na pozycję " Używaj czcinek jsMath". Jeżeli czcionki jsMath są w systemie wxMaxima wybiera je automatycznie podczas instalacji.




Rysunek 1.4:

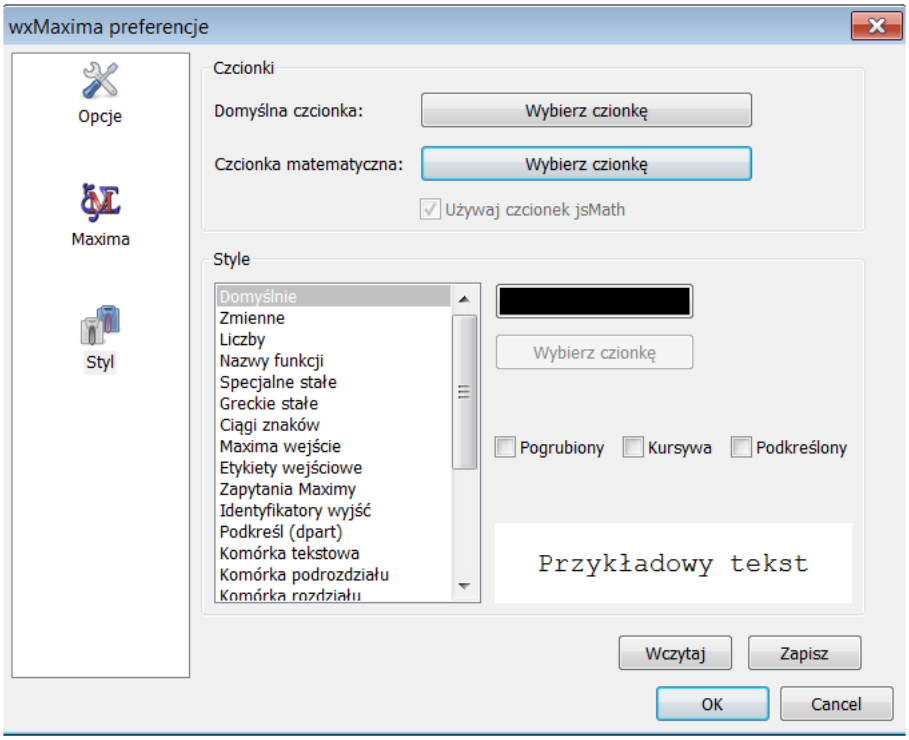
Jeżeli nie, warto je znaleźć w internecie (np. przy pomocy Google), zainstalować i zaznaczyć opcję używania. To poprawi jakość wyświetlanych wzorów matematycznych.

## 1.4 Pole menu "Cell" (Komórka)

W tym miejscu zgrupowane są polecenia ułatwiające pracę z komórkami. Polecenie "Wykonaj wszystkie komórki" jest wygodne gdy otwieramy plik zachowany w formacie *.wxm i chcemy szybko odtworzyć wszystkie wyniki.

### 1.4.1 Przyspieszanie edycji

Opiszemy dokładniej polecenia drugiej grupy. "Kopiuj ostatnie wejście" (Ctrl-I) Typowa sesja z programem Maxima polega na pracy interak-



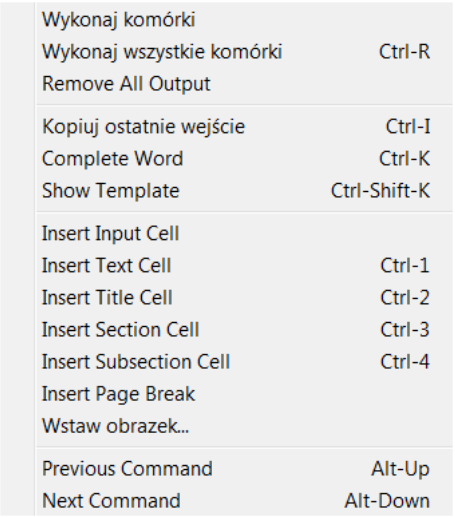

Rysunek 1.5:

tywnej. Wpisujemy polecenie, na podstawie wyniku wpisujemy następne, itd. Dlatego kolejne polecenie jest często drobną modyfikacją poprzedniego. Aby zaoszczędzić czas możemy skopiować ostatnie wejście (najlepiej używając skrótu Crl-I), dokonać potrzebnych zmian i wykonać komórkę.

Warto wspomnieć tu, że symbol % jest specjalną zmienną, pod którą pamiętany jest ostatni wynik i której można używać w dalszych obliczeniach.

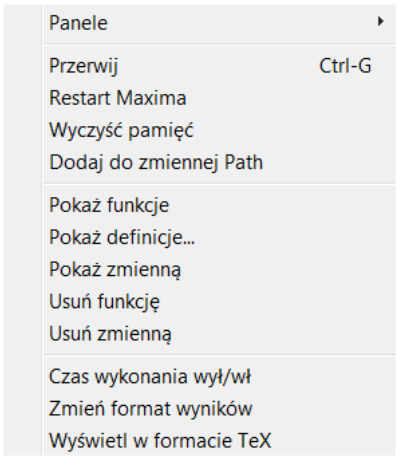
```
(%i6)  2+2;

(%o6)  4

(%i7)  %+5;

(%o7)  9
```

"Complete Word" (Ctrl-K) Napiszmy początkowy fragment nazwy funk-




Rysunek 1.6:

cji lub zmiennej. Następnie wciśnijmy Ctrl-K. Możliwe są dwie reakcje: 1. Jeżeli wpisany fragment jednoznacznie określa dalszy ciąg nazwy wxMaxima automatyczznie wpisze go. 2. Jeżeli jest nazwy wielu funkcji i/lub zmiennych zaczynają się w ten sposób pojawi się menu z wszystkimi możliwościami (o ile jest ich mniej niż 25), z którego dokonujemy właściwego wyboru. "Show template" (Ctrl-Shift-K) Jeżeli po wpisaniu nazwy funkcji wciśniemy Ctrl-Sfift-K w komórce pojawi się szablon składni tej funkcji (jeżeli dana funkcja ma kilka form, to wcześniej pojawi się menu, z którego wybieramy potrzebną nam formę). Argumenty funkcji ujęte są w nawiasy <>. Pierwszy argument jest zamarkowany. Wpisujemy właściwą wartość argumentu i wciskamy klawisz TAB, co powoduje zamarkowanie następnego argumentu. Operację powtarzamy do wyczerpania wszystkich argumentów. Wygodnie jest łączyć obie operacje. Wpisujemy fragment nazwy, uzupełniamy poprzez Ctrl-K, a następnie wciskamy Ctrl-Shift-K i uzupełniamy szablon. Sens poleceń w pozostałych dwóch grupach nie powinien budzić wątpliwości.




Rysunek 1.7:

## 1.5 Pole menu "Maxima"

### 1.5.1 Panele

Dla początkującego użytkownika interesująca może być pozycja panele, gdzie zgromadzono narzędzia pozwalające na wykorzystanie Maximy jako zaawansowanego kalkulatora naukowego możliwego do wykorzystania z minimalną wiedzą o programie (ale nie zerową!). Przykładowo zobaczmy panel Podstawowa Matematyka:


Rysunek 1.8:

### 1.5.2 Przerywanie działania programu

Kliknięcie na wybranej pozycji powoduje wykonanie wskazanych obliczeń (lub pojawienie się pomocniczego okna dialogowego). Panele pojawiły się niedawno w wxMaxima i są ciągle ulepszane. Jeżeli chcemy przerwać wykonanie komórki (np. obliczenia trwają zbyt długo) możemy spróbować użyć pozycji "Przerwij" (lub skrótu Ctrl-G). Ponieważ nie zawsze odnosi to pożądany skutek, to wtedy musimy użyć bardziej radykalnego środka jakim jest pozycja "Restart Maxima". Pamiętajmy o wcześniejszym zapisaniu dokumentu, nad którym pracujemy. Pozycja "Wyczyść pamięć" jest jednym z wariantów komendy "kill" służącej do usuwania nadanych wcześniej wartości. W tym przypadku jest to "kill(all)", co praktycznie prowadzi do czystego środowiska bez restartowania Maximy. Pozostałe elementy tego pola będziemy wykorzystywać w bardziej zaawansowanych częściach podręcznika.

## 1.6 Pola menu "Równania", "Algebra", "RRC", "Kreślenie", "Numeryczne"

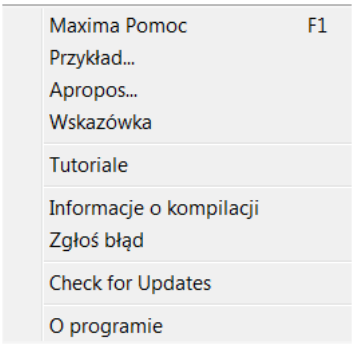
Spełniają podobną rolę jak panele, ale zawierają więcej funkcji i są bardziej elastyczne. Okienka dialogowe wspomagają użytkownika minimalizując potrzebę nauki składni poleceń programu i skracając czas wpisywania komend. Szczegóły udostępnionych tam funkcji poznamy w następnych notatnikach.

## 1.7 Pomoc

Standardowa pomoc (w języku angielskim) znajduje się w menu:

### 1.7.1 Maxima Pomoc

Jeżeli w notatniku ustawimy kursor za interesującym nas słowem i wciśniemy F1 otworzy się "Maxima Pomoc" na interesującym nas haśle (o ile takie hasło istnieje). Analogiczną pomoc możemy uzyskać bezpośrednio w notatniku używając komendy "descibe(nazwa)". Przykładowo:




Rysunek 1.9:

```
(%i8)  describe(diff);
```

$0 : diff(FunctionsandVariablesforDifferentiation)1 : diff < 1 > (FunctionsandVariablesf$
$separatednumbers, `all'or`none' : 0; --Specialsymbol : diffWhen`diff'ispresentasan`evflag'i$

(%o8) *true*

Równoważną formą tego polecenia jest ? nazwa (spacja po "?" jest obowiązkowa!)

```
(%i9)  ? diff;
```

$0 : diff(FunctionsandVariablesforDifferentiation)1 : diff < 1 > (FunctionsandVariablesf$
$separatednumbers, `all'or`none' : 0; --Specialsymbol : diffWhen`diff'ispresentasan`evflag'i$

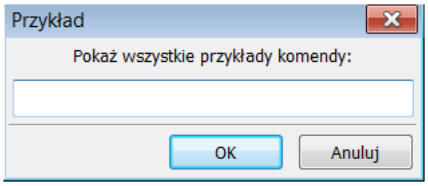
(%o9) *true*

### 1.7.2 Przykład...

Bardzo praktycznym źródłem informacji jest pozycja "Przykład...", która ułatwia korzystanie z systemowej komendy "example(nazwa)". Przy aktywnym poziomym kursorze klikamy tę pozycję, co powoduje otwarcie się okienka Wpisujemy interesującą nas komendę np. diff (pochodna) i otrzymujemy listę przykładów:

```
(%i10) example(diff);
```

Wszystkie komendy, dla których są przygotowane przykłady możemy zobaczyć wykonując:




Rysunek 1.10:

```
(%i19) example();
```

### 1.7.3 Apropos...

Jeżeli pamiętamy tylko część nazwy polecenia, to komenda "apropos(tekst)" (wpisana bezpośrednio do komórki lub za pomocą pozycji menu w podobny sposób jak wyżej) dostarczy listę nazw znanych Maximie zawierających podany fragment:

```
(%i20) apropos("diff");
```

Teraz możemy wybrać pełną nazwę komendy i uzyskać dalsze informacje używając Pomocy Maximy. Podobną, ale bardziej rozbudowaną, funkcjonalność ma polecenie ?? nazwa W pierwszym etapie pokazuje się ponumerowana lista nazw zawierających podany fragment. Wpisujemy wybrany numer ("all" jeżeli chcemy zobaczyć wszystko, "none" jeżeli rezygnujemy). Proszę przetestować to wykonując poniższą komórkę.

```
(%i21) ?? diff;
```

Standardowa pomoc (w języku angielskim) znajduje się w menu.

### 1.7.4 Maxima Pomoc i operator pomocy "?"

Jeżeli w notatniku ustawimy kursor za interesującym nas słowem i wciśniemy F1 otworzy się "Maxima Pomoc" na interesującym nas haśle (o ile takie hasło istnieje). Analogiczną pomoc możemy uzyskać bezpośrednio w notatniku używając komendy "descibe(nazwa)". Przykładowo:

```
(%i22) describe(diff);
```

Równoważną formą tego polecenia jest ? nazwa (spacja po "?" jest obowiązkowa!)

```
(%i23) ? diff;
```

### 1.7.5 Przykład...

Bardzo praktycznym źródłem informacji jest pozycja "Przykład...", która ułatwia korzystanie z systemowej komendy "example(nazwa)". Przy aktywnym poziomym kursorze klikamy tę pozycję, co powoduje otwarcie się okienka Wpisujemy interesującą nas komendę np. diff (pochodna) i otrzymujemy listę przykładów:

```
(%i24) example(diff);
```

Wszystkie komendy, dla których są przygotowane przykłady możemy zobaczyć wykonując:

```
(%i33) example();
```

### 1.7.6 Apropos...

Jeżeli pamiętamy tylko część nazwy polecenia, to komenda "apropos(tekst)" (wpisana bezpośrednio do komórki lub za pomocą pozycji menu w podobny sposób jak wyżej) dostarczy listę nazw znanych Maximie zawierających podany fragment:

```
(%i34) apropos("diff");
```

Teraz możemy wybrać pełną nazwę komendy i uzyskać dalsze informacje używając Pomocy Maximy.

### 1.7.7 Operator pomocy "??"

Podobną, ale bardziej rozbudowaną, funkcjonalność ma polecenie ?? nazwa W pierwszym etapie pojawia się ponumerowana lista nazw zawierających podany fragment. Wpisujemy numer (lub numery oddzielone spacją) interesującej nas nazwy (all jeżeli chcemy zobaczyć wszystkie opisy, none jeżeli rezygnujemy) i wciskamy Enter. Przykładowo:

```
(%i35) ?? gamma;
```

# Rozdział 2

# Wyrażenia w Maximie

## 2.1 Wstępne definicje

Podstawową pracą jaką wykonuje Maxima, to przetwarzanie wyrażeń. Wyrażenie w sensie Maximy jest pojęciem szerszym niż to się rozumie w potocznym języku i obejmuje m.in. wzory matematyczne, różnego rodzaju obiekty oraz konstrukcje programistyczne. Spróbujmy bardziej precyzyjnie opisać te pojęcia.

- Wyrażenie składa się z atomów i operatorów wraz z argumentami.
- Atomem jest symbol (nazwa), łańcuch znakowy (tekst) ujęty w znaki cudzysłowia lub numerał wyrażający liczbę całkowitą lub zmiennoprzecinkową.
  *Komentarz.* Przypomnijmy, że symbol lub słowo języka naturalnego wyrażające liczbę nazywamy numerałem lub cyfrą (ang. numeral, digit), a w języku potocznym, po prostu liczbą. Numerały różnią się od liczb tak jak słowa różnią się od rzeczy, które określają. Symbole „11”, „jedenaście” oraz „XI” są różnymi numerałami reprezentującymi tę samą liczbę.
  Użycie w definicji atomu pojęcia numerału (zamiast potocznie używanego słowa liczba) nie jest przypadkowe. Ma na celu podkreślenie, że np. atom 12 czy 31.12 jest traktowany przez program jako nazwa (symbol) specjalnego typu. Świadomość tego faktu pozwoli lepiej zrozumieć działanie programu.

- Wszystkie wyrażenia różne od atomów (krótko nieatomowe) reprezentowane są w postaci op(a_1,..., a_n), gdzie op jest nazwą operatora, a a_1,..., a_n są jego argumentami.
  (Postacie wyrażeń wyświetlonych na ekranie mogą być różne, ale ich reprezentacja wewnętrzna jest taka sama).

- Argumenty operatora mogą być atomami lub nieatomowymi wyrażeniami.

## 2.2 Funkcje analizujące strukturę wyrażenia

Maxima dostarcza funkcji, które pozwalają zbadać strukturę wyrażeń. Przeanalizujmy następujące przykłady.

Sprawdzanie czy dane wyrażenie jest atomem:

```
(%i1)  atom(-2); atom(%pi); atom("bleble");
       atom(1.23); atom(x); atom(sin(x)); atom(1/2);
```

(%o1) $true$

(%o2) $true$

(%o3) $true$

(%o4) $true$

(%o5) $true$

(%o6) $false$

(%o7) $false$

Zwróćmy uwagę, że liczby wymierne nie są atomami. Funkcja op(expr) zwraca operator główny wyrażenia "expr":

```
(%i8)  op(a+b);op(sin(x));op(a[2]);op(1/2);
```

(%o8) $+$

(%o9) $sin$

(%o10) $a$

(%o11) $/$

Te wyniki są oczywiste, ale poniższe chyba już mniej...

```
(%i12) op(2+2); op(diff(x^2,x));

part: argument must be a non-atomic expression; found 4 -- an error.
To debug this try: debugmode(true);
```

(%o13) $*$

Wyjaśnienie sensu komunikatu związane jest z zasadami przetwarzania wyrażeń, które poznamy w następnych częściach.

Funkcja args(expr) zwraca listę argumentów operatora głównego wyrażenia "expr".

```
(%i14) args(cos(x)); args(a+b);args((a+b)*(c+d));
```

(%o14) $[x]$

(%o15) $[b, a]$

(%o16) $[b + a, d + c]$

## 2.3 Typy wyrażeń

Wszystkie wyrażenia Maximy są jednego z trzech typów:

1. Wyrażenia matematyczne.
2. Wyrażenia będące obiektami.
3. Konstrukcje programistyczne.

W tym miejscu powiemy tylko kilka słów dla ogólnej orientacji czym są poszczególne typy. Szczegółom poświęcone są następne części.

### 2.3.1 Wyrażenia matematyczne

Wyrażenie matematyczne to wyrażenie, które składa się z nazw (symboli), liczb (numerałów), operatorów matematycznych (np. arytmetycznych, logicznych, itp.), standardowych funkcji matematycznych i funkcji definiowanych przez użytkownika.

### 2.3.2 Podstawowe obiekty: listy, macierze, zbiory

Obiekt podobnie jak inne wyrażenia składa się z operatora i jego argumentów. Najważniejszymi obiektami wbudowanymi w program Maxima są listy, macierze i zbiory.

#### Listy

Lista, to ciąg, którego elementami są wyrażenia dowolnego typu, ograniczony nawiasami kwadratowymi.

Listy są podstawowymi obiektami w Maximie, które służą do budowy bardziej skomplikowanych obiektów.

Warto wiedzieć, że Maxima jest napisana w języku programowania Lisp, którego nazwa jest skrótem od "lists processing"

```
(%i17) L:[a, raz, raz_dwa, "Witajcie", 42, 17.29,[x,y] ];
```

(%o17) $[a, raz, raz\_dwa, Witajcie, 42, 17.29, [x, y]]$

Jeżeli L jest listą to L[i] oznacza i-ty element listy

```
(%i18) L[1];L[3];L[5];L[7];
```

(%o18) $a$

(%o19) $raz\_dwa$

(%o20) $42$

(%o21) $[x, y]$

Ostatni element jest listą, której elementy możemy uzyskiwać w podobny sposób

```
(%i22) L[7][1];
```

(%o22) $x$

#### Macierze

Macierz zadajemy przy pomocy wyrażenia

```
matrix(L_1 , . . . , L_n )
```

gdzie `L_1 , . . . , L_n` są listami reprezentującymi wiersze macierzy.

Jeżeli M jest macierzą, to M[i, j] lub M[i][j] oznacza element o wskaźnikach (i,j).

```
(%i23) M:matrix([a, b, c],[ %pi, %e, 1729]);
```

$$\text{(\%o23)} \quad \begin{pmatrix} a & b & c \\ \pi & e & 1729 \end{pmatrix}$$

```
(%i24) M[1,1];M[2,2];
```

(%o24) $a$

(%o25) $e$

Macierze są argumentami wielu operatorów wbudowanych w Maximę związanych z algebrą liniową. Część z nich mamy dostępnych z pozycji menu "Algebra".

### Zbiory

Skończona liczba elementów ograniczona nawiasami klamrowymi jest rozumiana przez Maximę jako zbiór, w tym sensie, że na tych obiektach można wykonywać operacje znane z teorii mnogości (suma, przecięcie, itp.)

```
(%i26) {a,f,e,d,c,b,a};
```

(%o26) $a, b, c, d, e, f$

Zwróćmy uwagę, że zgodnie z teorią zbiorów każdy element jest uwzględniany tylko raz (mimo, że a wpisaliśmy dwukrotnie). Przykłady sumy i części wspólnej zbiorów:

```
(%i27) union({a,b,c,d,e},{b,d,f,g});
```

(%o27) $a, b, c, d, e, f, g$

```
(%i28) intersection({a,b,c,d,e},{b,d,f,g});
```

(%o28) $b, d$

### 2.3.3 Konstrukcje programistyczne

Uniwersalność Maximy wynika z faktu, że dostarcza nie tylko gotowych funkcji, z których możemy korzystać podobnie do kalkulatora, ale również jest językiem programowania. Język ten posiada typowe narzędzia jak instrukcje warunkowe, pętle czy możliwość budowania procedur. W połączeniu z dużą liczbą gotowych procedur matematycznych, graficznych i numerycznych daje to środowisko, w którym analiza modeli matematycznych jest relatywnie szybka i wygodna.

Dla ułatwienia pracy składnia instrukcji programistycznych jest podobna do innych języków. Na przykład instrukcja warunkowa ma kształt

```
if a then b elseif c then d
```

a pętla programowa

```
for a in L do S
```

Dla zrozumienia działania programu warto jednak wiedzieć, że te instrukcje również są wyrażeniami i wewnętrznie są pamiętane (i interpretowane) jako operator z argumentami.

Dla potwierdzenia tej uwagi użyjmy poznanych funkcji "op" i "args"

```
(%i29) kill(all);
```

(%o0) $done$

```
(%i1)  op(if a then b elseif c then d);
```

(%o1) $if$

```
(%i2)  args(if a then b elseif c then d);
```

(%o2) $[a, b, c, d, true, false]$

Tak więc wewnętrznie instrukcja warunkowa widziana jest jako wyrażenie if(a,b,c,d,true,false). W dalszym ciągu będziemy poznawać i używać instrukcje w ich klasycznej postaci, głowny cel tej uwagi, to chęć podkreślenia jednorodnej struktury całego programu Maxima.

## 2.4 Wartość wyrażenia

Każdemu wyrażeniu jest przypisywana wartość (ang. value), która też jest wyrażeniem (niekoniecznie tego samego typu)

`wyrażenie -> wartość`

Proces przypisywania wartości nazywamy ewaluacją wyrażenia (wymiennie będziemy mówić o wyliczaniu wyrażenia).

### 2.4.1 Ewaluacja atomów

Dla najprostszych wyrażeń (czyli atomów) zasady przypisywania są następujące:

- liczby (numerały) i łańcuchy tekstowe mają wartość równą samym sobie,
- nazwy (symbole) mają wartość określoną przy pomocy operatora przypisywania wartości, którego symbolem jest znak dwukropka :
- Jeżeli nazwa nie ma wartości określonej przy pomocy operatora : , to jej wartością jest ona sama.

Przykłady:

```
(%i3)  x : 1 ; waga : a/b;
```

(%o3) $1$

(%o4) $\frac{a}{b}$

### 2.4.2 Ewaluacja wyrażeń nieatomowych

Wyrażenia nieatomowe op(a_1,..., a_n) standardowo ewaluowane są w następujący sposób

1. W pierwszej kolejności wyliczane są argumenty operatora głównego "op".
2. Jeżeli z operatorem związana jest funkcja (wewnętrzna lub określona przez użytkownika) jest ona wykonywana z wyliczonymi argumentami, a jej rezultat jest wartością wyrażenia.

**Uwaga 1.** Ponieważ argumenty operatora mogą wyrażeniami nieatomowymi punkt 1 ma charakter pętli wykonywanej aż do osiągnięcia poziomu, na którym argumenty są atomami.

**Uwaga 2.** To jest standardowa pętla obliczeniowa dla większości operatorów/funkcji. Są jednak operatory, których wyliczanie jest niestandardowe (na przykład nie są ewaluowane argumenty). Zapoznając się z nowymi funkcjami należy zwrócić szczególną uwagę na informację jak wyliczają swoje argumenty. Ponadto użytkownik ma do dyspozycji kilka funkcji, którymi może modyfikować standardowy proces ewaluacji wyrażenia.

**Uwaga 3.** Kiedy do komórki obliczeniowej wxMaximy wpisujemy interesujące nas wyrażenie i wydajemy polecenie wykonania komórki (np. Shift+Enter), to otrzymany wynik jest, z dokładnością do uproszczeń, o których za chwilę, wartością wprowadzonego wyrażenia.

**Uwaga 4.** W przypadku kilku (niewielu, ale za to ważnych) funkcji bardziej od wartości wyrażenia interesuje nas tzw. efekt uboczny uzyskany w trakcie ewaluacji. Tak jest z procedurami graficznymi, czy drukującymi informację na ekranie.

## 2.5 Upraszczanie wyrażenia (simplification)

Wyrażenie będące wynikiem ewaluacji rzadko ma postać użyteczną do dalszej pracy. Dlatego przed podaniem wyniku Maxima (podobnie jak inne programy algebry komputerowej) próbuje go uprościć. Na przykład, nie wyświetla $x + x$ tylko $2x$ (o ile $x$ nie ma przypisanej wartości), podobne nie $x/x$ tylko $1$.

Taką procedurę znamy dobrze ze szkolnych czasów. Zadanie sprowadź dane wyrażenie do najprostszej postaci było rutynowym ćwiczeniem. Jednak próba algorytmizacji i implementacji komputerowej tej rutyny, to bardzo trudne (a w ogólnym przypadku nierozwiązywalne) zadanie algebry komputerowej. Jest wiele przyczyn takiego stanu, w tej chwili zwrócimy uwagę na trzy podstawowe.

1. Pojęcie "najprosza postać wyrażenia" nie jest dobrze określone. W szkole średniej zgoda na podaną odpowiedź była wynikiem konsensusu, a nie wynikiem jakiejkolwiek ścisłej definicji. Oczywiście dużą rolę odgrywała długość wzoru, ale nikt przecież nie liczył znaków w odpowiedzi. Byłoby to pozbawione sensu merytorycznego i praktycznego i w szczególności nie może być podstawą algorytmu. Sens słów "najprosza postać wyrażenia" w dużym stopniu zależy od kontekstu i celu naszej pracy. Przykładowo, jeżeli chcemy rozwiązać równanie, to najprostsza postać będzie związana z iloczynem, jeżeli chcemy policzyć pochodną, to raczej zmierzać będziemy do sumy.

2. Programy algebry komputerowej przetwarzają symbole. Zastanówmy się jak taki program ma uprościć na przykład wyrażenie sqrt(x^2). Czy to ma być x, czy |x|, czy jeszcze coś innego? Odpowiedź zależy od tego jak interpretujemy symbol x (np. zmienna rzeczywista, zespolona, macierz, operator itp.) i niestety dobra odpowiedź w jednym kontekście będzie fałszywa w innym.

3. Nawet jeżeli zdecydujemy się mocno ograniczyć możliwości naszego programu np. do dziedziny liczb rzeczywistych, to wiele reguł upraszczających (tożsamości) ma specyficzne dla siebie dziedziny, które bardzo trudno uwzględnić w ogólnym algorytmie.
   Praktycznie jest tylko jedna radykalna rada na wymienione problemy - udostępnić bogate i różnorodne procedury upraszczające, ale decyzję, które w danej sytuacji zastosować oraz odpowiedzialność za poprawne ich stosowanie w danym kontekście, przenieść na użytkownika programu.
   Niestety, tak skrajne stanowisko wpływa zniechęcająco na użytkownika, który np. ręcznie musiałby skracać (1+x^2)/(1+x^2). Dlatego autorzy takich programów (również Maximy) szukają złotego środka, starają się stosować automatycznie przynajmniej te reguły upraszczające, które są poprawne w możliwie szerokim kontekście (np. rozdzielność dodawania względem mnożenia, łączność, przemienność,itd.).
   W Maximie zakres tej automatyzacji upraszczania jest regulowany wartościami pewnych zmiennych systemowych. W szczególności jeżeli zmienna "simp" ma wartość false (standardowo ma wartość true), to wszelkie automatyczne uproszczenia są wyłączone.

Zobaczmy kilka prostych przykładów

```
(%i5)  kill(all);
```

(%o0) $done$

```
(%i1)  simp : true;  x+x;
```

(%o1) $true$

(%o2) $2\,x$

```
(%i3)  simp :false; x+x;
```

(%o3) $false$

(%o4) $x + x$

```
(%i5)  simp : true;  x/x;
```

(%o5) $true$

(%o6) $1$

```
(%i7)  simp : false;  x/x;
```

(%o7) $false$

(%o8) $\frac{x}{x}$

```
(%i9)  simp : true;  x+(b+x);
```

(%o9) $true$

(%o10) $2\,x + b$

```
(%i11) simp : false;  x+(b+x);
```

(%o11) $false$

(%o12) $x + (b + x)$

Te krótkie uwagi są bardzo dalekie od wyczerpania delikatnego i trudnego tematu upraszczania wyrażeń. Ich cel jest bardzo skromny: uświadomienie Czytelnikowi, że wyrażenie które wpisujemy do komórki podlega dwóm różnym procesom — ewaluacji i uproszczeniu:

```
wyrażenie -> ewaluacja -> uproszczenie -> wynik
```

Aby uniknąć różnych pułapek trzeba umieć rozróżniać te dwa procesy i być świadomym ich ograniczeń. Oczywiście wymaga to praktyki, ponieważ granica między tymi procesami nie jest w pełni intuicyjna. Przykładowo, operacje arytmetyczne są traktowane przez Maximę jako uproszczenia, a nie wyliczenia. Sprawdźmy

```
(%i13) simp:false; 2+2;
```

(%o13) $false$

(%o14) $2+2$

Inny przykład, w którym ewaluacja jest wykonywana, ale uproszczenia nie:

```
(%i15) x:1; b:2; simp : false;  x+(b+x);
```

(%o15) $1$

(%o16) $2$

(%o17) $false$

(%o18) $1+(2+1)$

# Rozdział 3

# Podstawowe wyrażenia matematyczne

## 3.1 Nazwy

Nazwy (zmiennych, stałych, funkcji itp.) w Maximie mogą zawierać małe i duże litery alfabetu łacińskiego, cyfry od 0 do 9, znak % oraz znak podkreślenia. W nazwie mogą wystąpić inne znaki, ale muszą być poprzedzone znakiem ukośnika \. Jeżeli pierwszym znakiem nazwy jest cyfra, to musi być poprzedzona znakiem ukośnika \. Cyfry wewnątrz nazwy nie muszą być poprzedzona znakiem ukośnika \. Maxima rozróżnia duże i małe litery. Pod zmienną sytemową "values" zapisywane są wprowadzone nazwy, którym przypisano wartość. Wartość zmiennej "values" możemy odczytać przy pomocy pozycji menu "Maxima->Pokaż zmienną".

## 3.2 Nadawanie wartości

Jak już wiemy nazwom zmiennych możemy przypisywać wartości przy pomocy operatora : (znak dwukropka, bez znaku równości!)

```
(%i1)  x : 1 /* spacje nie są konieczne,
       ale poprawiają czytelność */;

(%o1)  1
```

```
(%i2)  x;
```

(%o2) 1

Operator : wylicza wartość prawej strony i przypisuje ją do zmiennej. Zwróćmy uwagę, że przypisanej wartości nie zmieniają późniejsze modyfikacje składowych prawej strony. Przykładowo, zmiennej hf nadajemy wartość określoną prawą stroną

```
(%i3)  hf:f*(L/D)*(V^2/(2*g));
```

(%o3) $$\frac{f\,L\,V^2}{2\,g\,D}$$

Aktualna wartość hf:

```
(%i4)  hf;
```

(%o4) $$\frac{f\,L\,V^2}{2\,g\,D}$$

Przypisujemy wartości zmiennym f,L,D,V,g (aby zachować czytelność używamy $)

```
(%i5)  f:0.017 $ L:1000 $ D:0.2 $ V:2.5 $ g:32.2 $
```

Mimo tego wartość hf nie ulega zmianie:

```
(%i10) hf;
```

(%o10) $$\frac{f\,L\,V^2}{2\,g\,D}$$

Jeszcze raz definiujemy hf (dla zachowania efektu używamy $)

```
(%i11) hf:f*(L/D)*(V^2/(2*g))$
```

W tym momencie przy wyliczaniu prawej strony (i przypisywaniu wartości do hf) zostają uwzględnione nadane wartości

```
(%i12) hf;
```

(%o12) 8.249223602484472

## 3.3 Usuwanie wartości zmiennej

Funkcja

```
kill(a\_1,...., a\_n)
```

anuluje przypisania wartości do nazw a_1,...,a_n.

Funkcja "kill" rozpoznaje specjalne typy argumentów, przykładowo

```
 kill(all)
```

usuwa wszelkie przypisania poczynione w danej sesji.

Polecenie "kill(all)" jest podpięte pod pozycję w menu "Maxima->Wyczyść pamięć".

*Uwaga.* Komenda "kill(all)" usuwa również definicje innych obiektów, w szczególności funkcji użytkownika, o których będziemy mówić w następnej części.

## 3.4 Nazwy zarezerwowane

Tworząc nowe nazwy trzeba pamiętać, że pewne słowa są już zarezerwowane w Maximie. Przykładowo

```
integrate   next    from    diff    in      at
limit       sum     for     and         elseif
then    else    do      or      if
unless  product  while       thru    step
```

## 3.5 Predefiniowane stałe matematyczne

Pewnym nazwom są przypisane w Maximie wartości i własności typowych stałych matematycznych. Przykładowo:
`%e` podstawa logarytmu naturalnego (=exp(1))
`%i` jedność urojona (=sqrt(-1))

`inf` plus nieskonczoność (rzeczywista)
`minf` minus nieskonczoność (rzeczywista)
`infinite` nieskończoność zespolona
`%phi` złoty podział
`%pi` pi
`%gamma` stała Eulera
`false, true` stałe logiczne (fałsz, prawda)

## 3.6 Operatory arytmetyczne, porównania, logiczne

**Operatory arytmetyczne**

+ dodawanie
- odejmowanie
* mnożenie ( w Maximie znak mnożenia jest obowiązkowy, np. 2*x NIE 2x)
/ dzielenie
^ potęgowanie

**Operatory porównania**

= równa się
# różny
> większy
< mniejszy
>= większy lub równy
<= mniejszy lub równy

Powyższe operatory porównania mają charakter pomocniczy (np. do budowania warunków), w szczególności nie są ewaluowane (tzn. nie testują w żadnym sensie prawdziwości formuły):

```
(%i13) 2=2; a=b;3=5;
```

(%o13) $2 = 2$

(%o14) $a = b$

(%o15) $3 = 5$

Do bezpośredniego testowania służy funkcja "is":

```
(%i16) is(2=2); is(a=b);is(3=5);
```

(%o16) $true$

(%o17) $false$

(%o18) $false$

Zwróćmy uwagę, że funkcja "is" nie rozpoznaje wyrażeń równoważnych

```
(%i19) kill(x)$ is((x^2 - 1 = (x + 1) * (x - 1)));
```

(%o19) $false$

Do tego celu należy użyć funkcji "equal"

```
(%i20) is(equal(x^2 - 1, (x + 1) * (x - 1)));
```

(%o20) $true$

### Operatory logiczne

`and` koniunkcja
`or` alternatywa
`not` zaprzeczenie

Operatory logiczne wyliczają wartość operatorów porównania bez konieczności użycia funkcji "is"

```
(%i21) 1>2 or 3>2;
```

(%o21) $true$

**Zasady preferencji**

Kolejność wykonywania różnych operatorów wchodzących w skład złożonego wyrażenia jest określona przez zasady preferencji wbudowane w program. Jeżeli chcemy zmienić tę kolejność używamy standardowego grupowania przy pomocy nawiasów okrągłych ( ).

## 3.7 Predefiniowane funkcje matematyczne

Przykłady funkcji matematycznych dostępnych w Maximie:
`sqrt` pierwiastek kwadratowy
`sin` sinus
`cos` cosinus
`tan` tangens
`cot` cotangens
`sec` secans
`csc` kosecans
`asin` arcus sinus
`acos` arcus cosinus
`atan` arcus tangens
`acot` arcus cotangens
`asec` arcus secans
`acsc` arcus kosecans
`exp` wykładnicza $e^x$
`log` logarytm naturalny
`sinh` sinus hiperboliczny
`cosh` cosinus hiperboliczny
`tanh` tangens hiperboliczny
`asinh` arcus sin hiperboliczny
`acosh` hiperboliczny arcus cosinus
`atanh` arcus tangens hiperboliczny
`floor` "podłoga"
`ceiling` "sufit"
`fix` część całkowita
`float` przybliżenie dziesiętne
`abs` wartość bezwzględna

## 3.8 Arytmetyka liczb rzeczywistych

### 3.8.1 Obliczenia symboliczne (arytmetyka dokładna)

Jeżeli w wyrażeniu nie występują numerał (liczba) w zapisie dziesiętnym, to standardowo Maxima daje wyniki w postaci symbolicznej. Aby otrzymać przybliżenie zmiennoprzecinkowe ( w arytmetyce dziesiętnej 16-to cyfrowej) można (m.in.) użyć funkcji "float". Przykłady:

```
(%i22) 5/3; float(%);sqrt(3^2+5);float(%);
```

(%o22) $\frac{5}{3}$

(%o23) 1.666666666666667

(%o24) $\sqrt{14}$

(%o25) 3.741657386773941

Jeżeli jeden z argumentów jest liczbą dziesiętną, wynik (z małymi wyjątkami) też jest w postaci dziesiętnej:

```
(%i26) 3.5/2; sqrt(3.2^2+5);
```

(%o26) 1.75

(%o27) 3.903844259188627

### 3.8.2 Zmiana standardowego typu

Pozycja menu "Numeryczne"->"Wyjście..." pozwala na przełączanie standardowego typu wyniku (z symbolicznego na numeryczne i vice versa):

```
(%i28) if numer#false then numer:false else numer:true;
       exp(-2);
```

(%o28) *true*

(%o29) 0.13533528323661

i na odwrót:

```
(%i30) if numer#false then numer:false else numer:true;
       exp(-2);
```

(%o30) $false$

(%o31) $e^{-2}$

### 3.8.3 Arytmetyka o dowolnej precyzji

Standardowo Maxima wykonuje obliczenia w arytmetyce dziesiętnej 16-to cyfrowej. Jezeli chcemy wykonywać obliczenia z większą (lub mniejszą) dokładnością używamy innego typu liczb rzeczywistych o nawie "bfloat". Komenda "bfloat" dokonuje konwersji argumentu do zapisu w arytmetyce dziesiętnej z liczbą cyfr wskazana przez zmienna "fpprec", np. przez podstawienie "fpprec:32" (dostepne również z pozycji menu "Numeryczne".

*UWAGA 1:* wartość "fpprec" nie ma wpływu na obliczenia typu "float", które zawsze są 16-to cyfrowe.

*UWAGA 2:* fakt, ze dana liczba jest interpretowana jako "bfloat" jest sygnalizowane w zapisie "mantysa-cecha" literą b (dla "float" jest to litera e).

*UWAGA 3:* obliczenia w których wystepują liczby "float" i "bfloat" daja w wyniku "bfloat".

Przykłady: zadanie liczby cyfr w arytmetyce bfloat

```
(%i32) fpprec : 32;
```

(%o32) 32

Mix

```
(%i33) 3.5b-4*2.7e2; 2.2e-2+1.2b-1; -3.2b2/2e3;
```

(%o33) $9.45b-2$

(%o34) $1.4199999999999999872324352168107b-1$

(%o35) $-1.6000000000000000033306690738754 7b-1$

Kilka przykładów z $\pi$:

```
(%i36) float(%pi); bfloat(%pi);
```

(%o36) 3.141592653589793

(%o37) $3.14159265358979323846264338327950b0$

```
(%i38) fpprec:64; bfloat(%pi);
```

(%o38) 64

(%o39) $3.1415926535897932384626433832795028841971693993751058209749445920b0$

```
(%i40) fpprec:128; bfloat(%pi);
```

(%o40) 128

(%o41) $3.14159265358979323846264338327[71 digits]98214808651328230664709384460b0$

Sprawdzenie aktualnej arytmetyki

```
(%i42) fpprec;
```

(%o42) 128

powrót do standardu:

```
(%i43) fpprec:16 $ fpprec;
```

(%o44) 16

Kilka przykładów obliczeń sybolicznych i dziesiętnych:

```
(%i45) 2+(1/(2+1/(2+1/2)));
       4./3.+3./4.+1./6.;
       sqrt(1+(3/2)^3);
       abs(-2.5+1/2.5);
       sin(%pi/3+cos(%pi/3));
       sqrt(exp(-2)+log(abs(-2+1/2)));
       ceiling(3.25); floor(3.25); fix(3.25);
       3.25-fix(3.25); sinh(2.5);
```

(%o45) $\frac{29}{12}$

(%o46) $\frac{9}{4}$

(%o47) $\frac{\sqrt{35}}{2^{\frac{3}{2}}}$

(%o48) 2.1

(%o49) $\sin\left(\frac{\pi}{3}+\frac{1}{2}\right)$

(%o50) $\sqrt{\log\left(\frac{3}{2}\right)+e^{-2}}$

(%o51) 4

(%o52) 3

(%o53) 3

(%o54) 0.25

(%o55) 6.050204481039788

## 3.9 Liczby zespolone

Liczby zespolone zapisujemy w tradycyjnej postaci x+%i*y. Program wykonuje działania na liczbach zespolonych (w arytmetyce zależnej od typu części rzeczywistej i urojonej)

```
(%i56) z1: 3+5*%i; z2:-2+6*%i;z1+z2;z1-z2;z1*z2;z1^2;
```

(%o56) $5\,i+3$

(%o57) $6\,i-2$

(%o58) $11\,i+1$

(%o59) $5-i$

(%o60) $(5\,i+3)\,(6\,i-2)$

(%o61) $(5\,i+3)^2$

Ostatnie dwa przykłady wymagają wskazania stosownego uproszczenia. Na przykład

```
(%i62) expand(z1*z2);
```

(%o62) $8\,i-36$

Maxima dostarcza wielu funkcji związanych z liczbami zespolonymi. Przykładowo:
`cabs` (complex absolute value) moduł liczby zespolonej
`carg` (complex argument)argument liczby zespolonej
`rectform` przedstawia wyrażenie zespolone w postaci x+%i*y
`polarform` przedstawia wyrażenie zespolone w postaci wykładniczej
`realpart` zwraca część rzeczywistą
`imagpart` zwraca część urojoną
`conjugate` sprzężenie liczby zespolonej
Przykłady

```
(%i63) cabs(z1);
       arg(z1);
       z2;
       -z2;
       conjugate(z2);
       expand(z2*conjugate(z2));
       rectform(z1/z2);
       rectform(sqrt(z1));
       polarform(z1);
       polarform(z2);
```

(%o63) $\sqrt{34}$

(%o64) $\arg\left(5\,i+3\right)$

(%o65) $6\,i-2$

(%o66) $2-6\,i$

(%o67) $-6\,i-2$

(%o68) $40$

(%o69) $\frac{3}{5}-\frac{7\,i}{10}$

(%o70) $\frac{\sqrt{\sqrt{34}-3}\,i}{\sqrt{2}}+\frac{\sqrt{\sqrt{34}+3}}{\sqrt{2}}$

(%o71) $\sqrt{34}\,e^{i\,\mathrm{atan}\left(\frac{5}{3}\right)}$

(%o72) $2\,\sqrt{10}\,e^{i\,\left(\pi-\mathrm{atan}\left(3\right)\right)}$

Funkcje wbudowane w Maximę działają zarówno w dziedzinie rzeczywistej jak i zespolonej.

```
(%i73) rectform(sin(z1));
```

(%o73) $i\cos\left(3\right)\,\sinh\left(5\right)+\sin\left(3\right)\,\cosh\left(5\right)$

```
(%i74) rectform(log(-1));
```

(%o74) $i\,\pi$

# Rozdział 4

# Uproszczenia

Jak wspominaliśmy w części poświęconej ogólnym własnościami wyrażeń, przekształcenie wyrażenia do postaci użytecznej do dalszej analizy (potocznie zwanej uproszczeniem wyrażenia) jest pozostawione użytkownikowi. W tym celu ma on do dyspozycji wiele narzędzi, z których część omówimy w tym notatniku.

Podstawowe funkcje upraszczające zebrane są w menu pod nazwą "Uprość".

## 4.1 Uproszczenia automatyczne

Dla zaoszczędzenia czasu użytkownika Maxima oferuje kilka funkcji, które starają się automatycznie dać w wyniku, to co intuicyjnie rozumiemy przez uproszczone wyrażenie. Poniżej pokazujemy trzy przykłady takich funkcji. Jeżeli wynik nas satysfakcjonuje (i spełnia przyjęte założenia, za co w dużej mierze odpowiada użytkownik), to możemy na tym poprzestać. Jeżeli nie, możemy stosować bardziej wyspecyfikowane funkcje omówione dalej.

### 4.1.1 ratsimp

Funkcja: `ratsimp(<expr>)` upraszcza wyrażenie <expr> i wszystkie jego podwyrażenia (również argumenty funkcji niewymiernych).

MENU: "Uprość->Uprość wyrażenie"

```
--> sin(x/(x+x^2)) = %e^((1+log(x))^2-log(x)^2);
```

(%o12) $$\sin\left(\frac{x}{x^2+x}\right) = e^{(\log(x)+1)^2-\log(x)^2}$$

```
--> ratsimp(%);
```

(%o13) $$\sin\left(\frac{1}{x+1}\right) = e\,x^2$$

```
--> a+b*x+b*(a/b-x);
```

(%o14) $$b\,x + b\left(\frac{a}{b} - x\right) + a$$

```
--> ratsimp(%);
```

(%o15) $$2\,a$$

```
--> ((x-1)^(3/2)-(1+x)*sqrt(x-1))/sqrt(x-1)/sqrt(1+x);
```

(%o16) $$\frac{(x-1)^{\frac{3}{2}} - \sqrt{x-1}\,(x+1)}{\sqrt{x-1}\,\sqrt{x+1}}$$

```
--> ratsimp(%);
```

(%o17) $$-\frac{2}{\sqrt{x+1}}$$

### 4.1.2 fullratsimp

Funkcja `fullratsimp(<expr>)` wykonuje polecenie `ratsimp` dla wyrażenie <expr>, a następnie zestaw standardowych uproszczeń niewymiernych. Tę procedurę powtarza rekurencyjnie. Jeżeli wyrażenie nie ulega już zmianom jest wyświetlane jako wynik uproszczenia.

```
--> expr: (x^(a/2) + 1)^2*(x^(a/2) - 1)^2/(x^a - 1);
```

(%o20) $$\frac{\left(x^{\frac{a}{2}} - 1\right)^2\left(x^{\frac{a}{2}} + 1\right)^2}{x^a - 1}$$

```
--> ratsimp (expr);
```

(%o21) $$\frac{x^{2\,a}-2\,x^a+1}{x^a-1}$$

```
-->     fullratsimp (expr);
```

(%o22) $x^a-1$

### 4.1.3 radcan

Funkcja: `radcan(<expr>)` Upraszcza wyrażenie <expr>, które może zawierać funkcje logarytmiczne, wykładnicze i pierwiastkowe do specyficznej dla Maximy postaci kanonicznej.

MENU: "Uprość->Uprość Pierwiastki"

```
-->     radcan(%e^((1+log(x))^2-log(x)^2));
```

(%o1) $e\,x^2$

## 4.2 Przekształcanie do postaci iloczynów lub sum

### 4.2.1 Postać iloczynowa nad ciałem liczb całkowitych

Funkcja `factor(<expr>)` przedstawia liczbę całkowitą lub wyrażenie <expr>, zawierające dowolną liczbę zmiennych i funkcji, w postaci iloczynu czynników nierozkładalnych nad ciałem liczb całkowitych

MENU: "Uprość->Faktoryzuj wyrażenie"

```
-->     factor(51575319651600);
```

(%o10) $2^4\,3^2\,5^2\,1583\,9050207$

```
-->     factor((1-x^2)/(1+x));
```

(%o13) $-\left(x-1\right)$

```
-->     factor(sqrt((1-x^2)/(1+x)));
```

(%o14) $\sqrt{1-x}$

```
-->    factor (1 + x^99);
```

(%o16) $$(x+1)\left(x^2-x+1\right)\left(x^6-x^3+1\right)\left(x^{10}-x^9+x^8-x^7+x^6-x^5+x^4-x^3+x^2-x+1\right)\left(x^{20}+x^{19}-x^{17}-x^{16}+x^{14}+x^{13}-x^{11}-x^{10}-x^9+x^7+x^6-x^4-x^3+x+1\right)\left(x^{60}+x^{57}-x^{51}-x^{48}+x^{42}+x^{39}-x^{33}-x^{30}-x^{27}+x^{21}+x^{18}-x^{12}-x^9+x^3+1\right)$$

```
-->    factor(x^2-1/4);
```

(%o18) $$\frac{(2\,x-1)\,(2\,x+1)}{4}$$

```
-->    factor(x^4+x^2+1);
```

(%o27) $$\left(x^2-x+1\right)\left(x^2+x+1\right)$$

```
-->    factor(x^2-2);
```

(%o35) $x^2-2$

Tego wielomianu nie da się rozłożyć do postaci iloczynowej nad ciałem liczb całkowitych.

*Uwaga:* Bardziej rozbudowana postać funkcji "factor" umożliwia rozkład nad innymi ciałami (zob. help), Szczególnym przypadkiem tej ogólnej możliwości jest "gfactor".

### 4.2.2 Postać iloczynowa nad cialem liczb zespolonych

Funkcja `gfactor(<expr>)` przedstawia wielomian <expr> w postaci iloczynu czynników nierozkładalnych nad ciałem liczb zespolonych o częściach rzeczywistych i urojonych będących liczbami całkowitymi.

MENU: "Uprość->Faktoryzuj zespolenie"

```
-->    gfactor(x^2+1);
```

(%o36) $(x-i)\,(x+i)$

```
-->     factor(x^4-1);
```

(%o37) $(x-1)\,(x+1)\left(x^2+1\right)$

```
-->     gfactor(x^4-1);
```

(%o38) $(x-1)\,(x+1)\,(x-i)\,(x+i)$

### 4.2.3 Przekształcanie do postaci sumy

Funkcja `expand(<expr>)` jest operacją odwrotną do "factor". Wykonuje mnożenia i potęgowania występujące w wyrażeniu <expr>.

MENU: "Uprość->Rozwiń wyrażenie"

```
-->     w1:(x+1)^2*(y+1)^3;
```

(%o41) $(x+1)^2\,(y+1)^3$

```
-->     expand(w1);
```

(%o45) $x^2\,y^3+2\,x\,y^3+y^3+3\,x^2\,y^2+6\,x\,y^2+3\,y^2+3\,x^2\,y+6\,x\,y+3\,y+x^2+2\,x+1$

```
-->     factor(%);
```

(%o46) $(x+1)^2\,(y+1)^3$

### 4.2.4 Rozklad na ułamki proste

Funkcja `partfrac(<expr>,<var>)` rozkłada wyrażenie <expr> na ułamki proste względem zmiennej <var>.

MENU: "RRC->Ułamek prosty"

```
-->     partfrac(1/(1-x^2), x);
```

(%o8) $$\frac{1}{2\,(x+1)}-\frac{1}{2\,(x-1)}$$

## 4.3 Przekształcenia logartytmów

### 4.3.1 Zwijanie logarytmów

Funkcja `logcontract(<expr>)` rekurencyjnie analizuje wyrażenie <expr> przekształcając podwyrażenia postaci

$$a_1 \log(b_1) + a_2 \log(b_2) + c$$

w

$$\log(\texttt{ratsimp}(b_1^{a_1} {b_2}^{a_2})) + c$$

.

MENU: "Uprość->Zwiń logarytmy"

```
--> 2*(a*log(x) + 2*a*log(y));
```

(%o3) $2\ (2\,a\log(y) + a\log(x))$

```
--> logcontract(%);
```

(%o2) $a\log\left(x^2\,y^4\right)$

### 4.3.2 Rozwijanie logarytmów

Przekształcenia logarytmu iloczynu, potęgi, itp. jest realizowana nie przez funkcję, ale przez nadawanie wartości zmiennej systemowej `logexpand`. Domyślnie jej wartością jest `false`.

- Jeżeli `logexpand` nadamy wartość true, to 'log(a^b)' będzie upraszczane do 'b*log(a)'.
- Jeżeli `logexpand` nadamy wartość 'all', to dodatkowo 'log(a*b)' będzie upraszczane do 'log(a)+log(b)'.
- Jeżeli `logexpand` nadamy wartość 'super',to dodatkowo 'log(a/b)' będzie upraszczane do 'log(a)-log(b)' dla liczb wymiernych 'a/b'.
- Jeżeli `logexpand` ma standardową wartość 'false', żadne z tych uproszczeń nie jest wykonywane.

MENU: "Uprość->Rozwiń logarytmy"

Poniższy przykład pokazuje jak wygodnie zmieniać wartość zmiennej systemowej (taką postać dostajemy używając menu). Proszę zwrócić uwagę, że

- wyrażenie i zadanie wartości jest rozdzielone przecinkiem (nie średnikiem!),
- wartość zmiennej jest nadawana chwilowo na czas wykonania komendy, w następnym kroku wartość "logexpand" ponownie ma poprzednią wartość.

Ta forma jest przykładem użycia funkcji "ev", której poświęcimy osobny rozdział.

```
--> log(x^2/y^3), logexpand=super;
```

(%o6) $2\log(x) - 3\log(y)$

```
--> logexpand;
```

(%o7) $true$

## 4.4 Przekształcenia trygonometryczne

### 4.4.1 Podstawowe uproszczenie

Funkcja `trigsimp(<expr>)` upraszcza wyrażenie trygonometryczne <expr> do postaci zależnej od ‘sin’, ‘cos’, ‘sinh’, ‘cosh’ wykorzystując tożsamość sin(x)^2 + cos(x)^2 = 1 oraz cosh(x)^2 - sinh(x)^2 = 1.

Użycie ‘trigreduce’, ‘ratsimp’, oraz ‘radcan’ może dodatkowo uprościć wyrażenie.

MENU: "Uprość->Uproszczenia trygonometryczne->"Uprość trygonometrycznie"

```
--> tan(x)*sec(x)^2+((1-sin(x)^2)*cos(x))/cos(x)^2;
```

(%o46) $\sec(x)^2\tan(x) + \dfrac{1-\sin(x)^2}{\cos(x)}$

```
-->    trigsimp(%);
```

(%o47) $$\frac{\sin\left(x\right)+\cos\left(x\right)^4}{\cos\left(x\right)^3}$$

```
-->    (3+cos(x)^4+8*sin(x)+4*(cos(x)^2-
       sin(x)^2)-6*cos(x)^2*sin(x)^2+
       sin(x)^4)/(8*cos(x)^3);
```

(%o48) $$\frac{\sin\left(x\right)^4-6\cos\left(x\right)^2\sin\left(x\right)^2+4\left(\cos\left(x\right)^2-\sin\left(x\right)^2\right)+8\sin\left(x\right)+\cos\left(x\right)^4+3}{8\cos\left(x\right)^3}$$

```
-->    trigsimp(%);
```

(%o49) $$\frac{\sin\left(x\right)+\cos\left(x\right)^4}{\cos\left(x\right)^3}$$

### 4.4.2 Przekształcenie do funkcji jednego kąta

Funkcja `trigexpand(<expr>)` rozwija funkcje trygonometryczne i hiperboliczne argumentów będących sumami i wielokrotnościami kątów do funkcji pojedyńczych kątów.

MENU: "Uprość->Uproszczenia trygonometryczne->"Rozwiń trygonometrycznie"

*Uwaga.* "trigexpand" upraszcza w jednym kroku, jeden poziom wyrażenia. Dla pełnego rozwinięcia trzeba czasami powtórzyć komendę wielokrotnie.

```
-->    trigexpand(sin(10*x+y));
```

(%o51) $$\cos\left(10\,x\right)\,\sin\left(y\right)+\sin\left(10\,x\right)\,\cos\left(y\right)$$

```
-->    trigexpand(%);
```

(%o52) $$\begin{aligned}&\left(-\sin\left(x\right)^{10}+45\cos\left(x\right)^2\sin\left(x\right)^8-210\cos\left(x\right)^4\sin\left(x\right)^6\right.\\&\left.+210\cos\left(x\right)^6\sin\left(x\right)^4-45\cos\left(x\right)^8\sin\left(x\right)^2+\cos\left(x\right)^{10}\right)\sin\left(y\right)\\&+\left(10\cos\left(x\right)\,\sin\left(x\right)^9-120\cos\left(x\right)^3\sin\left(x\right)^7+252\cos\left(x\right)^5\sin\left(x\right)^5\right.\\&\left.-120\cos\left(x\right)^7\sin\left(x\right)^3+10\cos\left(x\right)^9\sin\left(x\right)\right)\cos\left(y\right)\end{aligned}$$

Jeżeli chcemy zrobić to w jednym kroku możemy wykorzystać zmienną systemową "trigexpand" nadając jej wartość true w sposób podobny do omówionego przy okazji "logexpand":

```
-->    sin(10*x+y), trigexpand=true;
```

$$\begin{aligned}(\%o54)\ & (-\sin(x)^{10}+45\cos(x)^2\sin(x)^8-210\cos(x)^4\sin(x)^6\\ & +210\cos(x)^6\sin(x)^4-45\cos(x)^8\sin(x)^2+\cos(x)^{10})\sin(y)\\ & +(10\cos(x)\ \sin(x)^9-120\cos(x)^3\sin(x)^7+252\cos(x)^5\sin(x)^5\\ & -120\cos(x)^7\sin(x)^3+10\cos(x)^9\sin(x))\cos(y)\end{aligned}$$

### 4.4.3 Przekształcanie do funkcji wielokrotności kąta

Funkcja `trigreduce(<expr>)` zwija funkcje trygonometryczne i hiperboliczne argumentu <x> do funkcji argumentów będących wielokrotnością <x>.

MENU: "Uprość->Uproszczenia trygonometryczne->Redukuj trygonometrycznie"

```
-->    trigreduce(-sin(x)^2+3*cos(x)^2+x);
```

$$(\%o56)\ \frac{\cos(2\,x)}{2}+3\left(\frac{\cos(2\,x)}{2}+\frac{1}{2}\right)+x-\frac{1}{2}$$

## 4.5 Podstawienia

**Funkcja subst.** Funkcja `subst(<a>, <b>, <c>)` podstawia <a> w miejsce <b> w wyrażeniu <c>. <b> musi być atomem lub pełnym podwyrażeniem <c>.

Dla przykładu, ‘x+y+z’ jest pełnym podwyrażeniem wyrażenia ‘2*(x+y+z)/w’, natomiast ‘x+y’ nie jest.

MENU: "Uprość->Podstaw..."

Równoważna postać to (uwaga na kolejność!):

```
subst(<b> = <a>, <c>)
```

```
-->    kill(all);
```

(%o0) $done$

```
-->    subst(a, x+y, x + (x+y)^2 + y);
```

(%o1) $y + x + a^2$

```
-->    subst(-%i, %i, a + b*%i);
```

(%o2) $a - i\,b$

**Podstawienia szeregowe.** Kilka podstawień w jednym poleceniu można wykonać podając je w postaci listy. Wówczas są one realizowane szeregowo (tzn. wykonywane jest pierwsze podstawienie, do wyniku stosuje się drugie podstawienie, itd.)

```
-->    subst([a=b, b=c], a+b);
```

(%o3) $2\,c$

**Podstawienia równoległe** Jeżeli chcemy aby podstawienia były wykonywane równolegle (tzn. na wykonywanie następnych podstawień nie mają wpływu poprzednie) należy użyć funkcji `psubst`. Porównajmy:

```
-->    subst ([a^2=b, b=a], sin(a^2) + sin(b));
```

(%o8) $2\sin\left(a\right)$

```
-->    psubst ([a^2=b, b=a], sin(a^2) + sin(b));
```

(%o6) $\sin\left(b\right) + \sin\left(a\right)$

**Funkcja ratsubst.** Funkcja `ratsubst(<a>, <b>, <c>)` podstawia <a> w miejsce <b> w wyrażeniu <c>. W przeciwieństwie do `subst`, <b> nie musi być pełnym podwyrażeniem <c>.

```
-->    subst(a, x+y,2*(x+y+z)/w);
```

(%o12) $$\frac{2\,\left(z + y + x\right)}{w}$$

```
-->     ratsubst(a,x+y,2*(x+y+z)/w);
```

(%o13) $$\frac{2\,z+2\,a}{w}$$

# Rozdział 5

# Listy

Wstępne informacje o listach były podane w rozdziale poświęconym ogólnym własnościom wyrażeń. Podkreśliliśmy tam fundamentalną rolę list w programie Maxima (i wielu innych programach algebry komputerowej). Nic więc dziwnego, że wracamy do tego tematu, aby dokładniej poznać użyteczne funkcje przy pracy z listami.

## 5.1 Tworzenie nowych list

Nową listę możemy tworzyć ręcznie wypisując jawnie jej elementy ujęte w nawiasy kwadratowe. Ale lepiej ten żmudny proces zautomatyzować (jeżeli to jest możliwe) np. przy pomocy jednej z poniższych funkcji.

### 5.1.1 makelist

Funkcja: `makelist(<expr>, <i>, <i_0>, <i_1>)`

Wynikiem jest lista, której elementy są budowane poprzez kolejne wyliczanie <expr> z <i> zmieniającym się od <i_0> do <i_1>.

```
(%i1)  makelist(x^i,i,3,10);
```

(%o1) $[x^3, x^4, x^5, x^6, x^7, x^8, x^9, x^{10}]$

Funkcja: `makelist(<expr>, <x>, <lista>)`|

Wynikiem jest lista, której elementy są budowane poprzez kolejne wyliczanie <expr> z <x> przyjmującym wartości z <lista>.

```
(%i2)  makelist(x=y,y,[a,b,c]);
```

(%o2) $[x = a, x = b, x = c]$

### 5.1.2 create_list

Funkcja:
`create_list(<form>, <x_1>, <lista_1>, ..., <x_n>, <lista_n>)`

Wynikiem jest lista elementów powstałych przez wyliczenie wyrażenia <form> ze zmienną <x_1> przyjmującą kolejne wartości z listy <lista_1>. Następnie to samo jest wykonywane <x_2> i <lista_2>, itd. aż do <x_n> i <lista_n>.

Wyrażenia <x_i> muszą być symbolami ponieważ nie są wyliczane w trakcie wykonywania. Elementy <lista_i> są wyliczane raz, na początku procesu iteracji.

```
(%i3)  create_list(x^i,i,[1,3,7]);
```

(%o3) $[x, x^3, x^7]$

```
(%i4)  create_list([i,j],i,[a,b],j,[e,f,h]);
```

(%o4) $[[a, e], [a, f], [a, h], [b, e], [b, f], [b, h]]$

Argument <lista_i> moe być zastąpiony przez parę wyrażeń, które wyliczają się do liczby. W tym przypadku odgrywają one rolę dolnej i górnej granicy zmiany <x_i>.

```
(%i5)  create_list([i,j],i,[1,2,3],j,1,i);
```

(%o5) $[[1, 1], [2, 1], [2, 2], [3, 1], [3, 2], [3, 3]]$

Granice dla j mogą zależeć od aktualnej wartości i.

## 5.2 Modyfikacja istniejących list

Aby utworzyć nowe listy możemy modyfikować istniejące poprzez dodawanie i usuwanie elementów czy łączenie na różne sposoby gotowych list.

### 5.2.1 Wybieranie podlisty

Funkcja: `sublist(<lista>, <p>)` Wynikiem jest lista tych elementów z listy <lista>, dla których predykat <p> przyjmuje wartość true.

```
(%i6)  L: [1, 2, 3, 4, 5, 6];
```

(%o6) $[1, 2, 3, 4, 5, 6]$

```
(%i7)  sublist (L, evenp);
```

(%o7) $[2, 4, 6]$

Podlista liczb parzystych.

### 5.2.2 Bezpośrednie łączenie dwóch list

Funkcja: `append(<lista\_1>, ..., <lista\_n>)`

Wynikiem jest lista złożona z elementów <lista_1>, po których następują elementy <lista_2>, itd. aż do elementów <lista_n>.

```
(%i8)  append([a,b,c,d], [%pi,2,"tekst"]);
```

(%o8) $[a, b, c, d, \pi, 2, tekst]$

### 5.2.3 Naprzemienne łączenie dwóch list

Funkcja: `join(<l>, <m>)`

Buduje nową listę zawierającą elementy list <l> i <m> wstawiane naprzemiennie. W wyniku otrzymujemy listę [<l>[1], <m>[1], <l>[2], <m>[2], ...]. Jeżeli listy są różnej długości "join" ignoruje elementy dłuższej listy.

```
(%i9)  join([a,b,c,d],[1,2,3,4,5]);
```

(%o9) $[a, 1, b, 2, c, 3, d, 4]$

### 5.2.4 Dodawanie elementu na początku i na końcu listy

Funkcja: `cons(<expr>,<lista>)`

Wynikiem jest lista, której pierwszym elementem jest <expr>, a następnymi są elementy listy <lista>.

```
(%i10) cons(a,[1,2,3]);
```

(%o10) $[a, 1, 2, 3]$

Funkcja: `endcons(<expr>,<lista>)`

Wynikiem jest lista, złożona z elementów listy <lista> i elementu <expr> umieszczonego na końcu

```
(%i11) endcons(a,[1,2,3]);
```

(%o11) $[1, 2, 3, a]$

### 5.2.5 Usuwanie z listy wskazanego elementu

Funkcja: `delete(<expr\_1>, <expr\_2>)`

Usuwa z wyrażenia <expr_2> dowolny argument jego głównego operatora, który jest taki sam jak <expr_1> (taki sam w sensie operatora "=", tzn. mamy tu do czynienia ze składniową równością, a nie identycznością w matematycznym sensie).

Najprostsze zastosowanie funkcji "delete" to usuwanie wskazanego elementu z listy.

```
(%i12) delete(y, [w, x, y, z, z, y, x, w]);
```

(%o12) $[w, x, z, z, x, w]$

```
(%i13) delete(sin(x), [sin(x),cos(sin(x)), tan(x)]);
```

(%o13) $[\cos(\sin(x)), \tan(x)]$

Drugi sin(x) nie został usunięty, ponieważ nie jest argumentem głównego operatora.

```
(%i14) delete(1, [sin(x)^2+cos(x)^2,11,x/x]);
```

(%o14) $[\sin(x)^2 + \cos(x)^2, 11]$

x/x zostało usunięte ponieważ stosują się automatyczne reguły upraszczania, jedynka trygonometryczna nie, ponieważ jej uproszczenie wymaga wezwania specjalnego polecenia.

### 5.2.6 Usuwanie z listy pierwszych lub ostatnich n elementów

Funkcja: `rest(<lista>, n)`

Zwraca listę <lista> z usuniętymi n pierwszymi elementami. Jeżeli n jest ujemne, usuwane jest n ostatnich elementów.

```
(%i15) rest([a,b,c,d,e],3);
```

(%o15) $[d, e]$

```
(%i16) rest([a,b,c,d,e],-3);
```

(%o16) $[a, b]$

### 5.2.7 "Wypłaszczanie" listy

Funkcja: `flatten(<expr>)`

Jeżeli <expr> jest listą złożoną (zawierającą listy jako elementy) flatten usuwa wszystkie wewnętrzne nawiasy tworząc jedną prostą listę. Zakres działania "flatten" obejmuje ogólne wyrażenia, ale tym na razie nie będziemy się zajmować.

```
(%i17) flatten([a, b, [c, [d, e], f], [[g, h]], i, j]);
```

(%o17) $[a, b, c, d, e, f, g, h, i, j]$

## 5.3 Wybieranie elementu listy

Wcześniej poznaliśmy ogólne zasady wybierania elementów danej listy. Do pierwszych dziesięciu elementów oraz ostatniego elementu danej listy możemy odwoływać się używając odpowiednich słów angielskich.

```
first
second
third
fourth
fifth
sixth
seventh
eighth
ninth
tenth
last
```

```
(%i18) second([a,b,c,d]);
```

(%o18) $b$

## 5.4 Własności listy

### 5.4.1 Długość listy

Funkcja: `length(<expr>)`

Podaje liczbę składowych w wyświetlanej postaci wyrażenia <expr>. Dla to jest to liczba elementów. Dla macierzy jest to liczba wierszy. Dla sum jest to liczba składników.

```
(%i19) length([a,b,c,d]);
```

(%o19) 4

```
(%i20) length(a+b+c+d);
```

(%o20) 4

### 5.4.2 Minimum i maksimum listy

Funkcja: `lmax(<e>)`

Funkcja: `lmin(<e>)`

Wynikiem jest maksimum (odpowiednio minimum) listy <e>.

```
(%i21) lmax([1.2, 34,-2]);
```

(%o21) 34

```
(%i22) lmin([1.2, 34,-2]);
```

(%o22) $-2$

```
(%i23) lmax([a,b,c]);
```

(%o23) $\max(a, b, c)$

### 5.4.3 Predykaty

Funkcja: `listp(<e>)`

Predykat sprawdzający czy <e> jest listą.

Funkcja: `member(<expr>, <lista>)`

Testuje czy <expr> jest elementem listy <lista> (test opiera się na formalnym porównaniu składni, podobnie jak dla funkcji "is")

```
(%i24) member(a, [x,y,a]);
```

(%o24) *true*

```
(%i25) member(1,[1.0,2,3]);
```

(%o25) $false$

### 5.4.4 Odwracanie porządku listy

Funkcja: `reverse(<lista>)`

Odwraca porządek elementów listy <lista>

```
(%i26) reverse([a,b,c,d]);
```

(%o26) $[d, c, b, a]$

```
(%i27) reverse([b,a,c,d]);
```

(%o27) $[d, c, a, b]$

## 5.5 Działania arytmetyczne na listach

Jeżeli zmienna systemowa `listarith` ma wartość `true`, to listy mogą być argumentami operacji arytmetycznych. Wynikiem takiego działania jest lista, której elementy są wynikiem danej operacji wykonanej na elementach o tych samych indeksach (mówimy krótko, że działania na listach wykonywane są "poelementowo").

Jeżeli "listarith" ma wartość "false" operacje arytmetyczne na listach nie są wykonywane.

Domyślną wartością `listarith` jest `true`.

*UWAGA.* Te operacje, mimo zewnętrznego podobieństwa, nie zawsze pokrywają się z wynikami, których spodziewamy się na bazie przyzwyczajeń np. z algebry liniowej.

Uzupełnijmy powyższe ogólne uwagi przykładami.

```
(%i28) [a,b,c]+[1,2,3];
```

(%o28) $[a + 1, b + 2, c + 3]$

```
(%i29) 2*[a,b,c];
```

(%o29) $[2\,a, 2\,b, 2\,c]$

```
(%i30) op([a,b,c]+[1,2,3]);
```

(%o30) [

Przy dodawaniu i mnożeniu przez "skalar" zasady są takie jak dla wektorów (ostatni przykład pokazuje, że wynik jest listą). Ale następne przykłady pokazują zasadnicze różnice:

```
(%i31) [a,b,c]+1;
```

(%o31) $[a+1, b+1, c+1]$

```
(%i32) [a,b,c]*[1,2,3];
```

(%o32) $[a, 2\,b, 3\,c]$

```
(%i33) [a,b,c]^2;
```

(%o33) $[a^2, b^2, c^2]$

```
(%i34) sin([a,b,c]);
```

(%o34) $[\sin(a), \sin(b), \sin(c)]$

"Poelementowe" działanie na listach jest bardzo użyteczną cechą list przy tworzeniu zwartych i czytelnych kodów programowych, ale trzeba uważać, aby nie pomylić operacji macierzowych (wektorowych) z operacjami na listach.

Dla uniknięcia tego typu kłopotów Maxima wyraźnie odróżnia typ "matrix" od typu "list" (np. lista list nie jest macierzą). Jednak dla zaoszczędzenia czasu, Maxima w wielu sytuacjach dokonuje automatycznej konwersji do odpowiedniego typu.

Przykładowo, w operacjach zawierających listy i macierze, listy są automatycznie konwertowane do typu "matrix" i wynik zawsze jest typu "matrix".

```
(%i35) listp([[a,b],[c,d]]);
```

(%o35) $true$

```
(%i36) A : [[a,b],[c,d]]+matrix([1,2],[3,4]);
```

$$(\%o36)\quad \begin{pmatrix} a+1 & b+2 \\ c+3 & d+4 \end{pmatrix}$$

```
(%i37) listp(A);
```

$$(\%o37)\quad false$$

# Rozdział 6

# Definiowanie funkcji

## 6.1 Definiowanie funkcji przy pomocy operatora :=

Najczęściej definiujemy funkcje przy pomocy operatora := (znak równości bezpośrednio po dwukropku, bez spacji). Po lewej stronie znajduje się nazwa funkcji wraz z argumentami ujętymi w nawiasy okrągłe, po prawej wyrażenie zależne od argumentów i (ewentualnie) innych parametrów:

```
nazwa(arg_1,arg_2,...,arg_n) := wyrażenie
```

- Definicja jest pamiętana dokładnie w tej postaci jak została wprowadzona, bez wyliczania prawej strony.
- Wartością funkcji jest wartość prawej strony, która jest wyliczana za każdym razem gdy funkcja jest używana. Do wyliczeń brane są aktualne wartości argumentów i parametrów.
- Pod zmienną sytemową "functions" zapisywane są nazwy definiowanych funkcji. Wartość zmiennej "functions" możemy odczytać przy pomocy pozycji menu "Maxima->Pokaż funkcje".
- Tam również odnajdziemy pozycję "Maxima->Pokaż definicje...", która przy pomocy polecenia "fundef" odtwarza definicję wskazanej funkcji.

Przykłady:

```
(%i1)  a : %pi;
```

(%o1) $\pi$

Definiujemy funkcję

```
(%i2)  f(x):=a*x^2;
```

(%o2) $\mathrm{f}\left(x\right) := a\,x^2$

Mimo, że a ma wartość pi, w definicji f parametr a nie jest wyliczany

```
(%i3)  y :2;
```

(%o3) 2

```
(%i4)  f(y);
```

(%o4) $4\,\pi$

Po wezwaniu funkcji prawa strona jest wyliczana

```
(%i5)  f(2); f(z); f(x+1);
```

(%o5) $4\,\pi$

(%o6) $\pi\,z^2$

(%o7) $\pi\left(x+1\right)^2$

Podobnie definiujemy funkcje wielu zmiennych

```
(%i8)  p(x,y):=sqrt(x^2+y^2);
```

(%o8) $\mathrm{p}\left(x,y\right) := \sqrt{x^2+y^2}$

## 6.2 Definiowanie funkcji przy pomocy polecenia "define"

Odsuwanie w czasie wyliczania prawej strony funkcji do chwili jej wezwania, które cechuje operator :=, jest na ogół tym zachowaniem, którego potrzebujemy. Są jednak sytuacje gdzie prowadzi to do błedu. Zilustrujmy to prostym przykładem.

Funkcja $g$ jest zdefiniowana jako pochodna funkcji x^2:

```
(%i9)  g(x) := diff(x^2,x);
```

(%o9) $\mathrm{g}\left(x\right) := \mathrm{diff}\left(x^2, x\right)$

Obliczamy wartość g(b)

```
(%i10) g(b);
```

(%o10) $2\,b$

Wynik jest poprawny. Ale dla g(1) zaczynają się kłopoty

```
(%i11) g(1);


diff: second argument must be a variable; found
1#0: g(x=1) -- an error. To debug this try:
debugmode(true);
```

Jeżeli rozumiemy zasadę działania operatora :=, wynik nie jest zaskakujący – próba obliczenie prawej strony z aktualną wartością argumentu prowadzi do bezsensownej komendy diff(1^2,1).

W tym przypadku pożądane byłoby, aby prawa strona definicja funkcji była zapamiętana po obliczeniu prawej strony.

Cel ten realizuje polecenie "define" o składni

```
define(nazwa_funkcji(arg_1,...,arg_n), wyrażenie)
```

Jeżeli poprawimy definicję funkcji g w ten sposób, jej zachowanie będzie zgodne z oczekiwaniami:

```
(%i12) define(g1(x),diff(x^2,x));
```

(%o12) $\mathrm{g1}\left(x\right) := 2\,x$

```
(%i13) g1(b);g1(1);
```

(%o13) $2\,b$

(%o14) $2$

## 6.3 Usuwanie definicji funkcji

Podobnie jak dla zmiennych niepotrzebną definicję funkcji możemy usunąć (m.in.) przy pomocy polecenia "kill"

```
(%i15) kill(g1);
```

(%o15) *done*

```
(%i16) g1(1);
```

(%o16) $\mathrm{g1}\,(1)$

## 6.4 Funkcje anonimowe

### 6.4.1 lambda-wyrażenia

Maxima dopuszcza definiowanie funkcji bez nadawania im nazwy (stąd tytuł funkcje anonimowe). Do tego celu służy polecenie "lambda" o podstawowej składni

```
lambda([x_1, ..., x_m], wyrazenie_1, ..., wyrazenie_n)
```

Dla argumentów `x_1,...,x_n` wyliczane są kolejno `wyrazenie_1,...,wyrazenie_n`, wynikiem działania jest wartość ostatniego wyrażenia `wyrazenie_n`.

Jeżeli lambda-wyrażeniu nadamy nazwę, to otrzymamy zwyczajną funkcję o tej nazwie. Przykład:

```
(%i17) h : lambda([x,y],x^2+y^2);
```

(%o17) $\mathrm{lambda}\left([x,y],x^2+y^2\right)$

```
(%i18) h(a,b);
```

(%o18) $b^2+\pi^2$

```
(%i19) h(2,3);
```

(%o19) $13$

Jedno z częstszych zastosowań tej konstrukcji związane jest z faktem, że w poleceniach, które jako argumenty wymagają nazwy funkcji, w miejsce tej nazwy możemy wstawić lambda-wyrażenie. Dwie szczególnie użyteczne funkcje tego typu omówimy poniżej.

### 6.4.2 Funkcja "apply"

Funkcja

```
apply(f,[x1, x2, ... , xn])
```

buduje i wylicza wyrażenie postaci

```
f(arg1, arg2, ... , argn),
```

gdzie `arg1=x1, arg2=x2, ... , argn=xn`.

Argumenty wyrażenia są zawsze wyliczane, nawet w sytuacji, gdy nie są wyliczane w innych kontekstach.

```
(%i20) kill(all)$
```

Dla uniknięcia konfliktu czyścimy środowisko nazw.

```
(%i1)  apply(sin, [x * %pi]);
```

(%o1) $\sin\left(\pi\, x\right)$

"Aplikujemy" sin do x*%pi (oczywiście tylko dla ilustracji).

```
(%i2)  L1:[a, b, c, x, y, z];
```

(%o2) $[a, b, c, x, y, z]$

Nadanie nazwy liście używanej w dalszych rachunkach

```
(%i3)  apply("+", L1);
```

(%o3) $z + y + x + c + b + a$

"Aplkujemy" + to elementów listy L1, czyli sumujemy elementy L1

```
(%i4)  L2:[1, 5, -10.2, 4, 3];
```

(%o4) $[1, 5, -10.2, 4, 3]$

Definiujemy nową listę L2 i wyliczamy jej minimum (ponownie dla ilustracji kontrukcji apply)

```
(%i5)  apply(min,L2);
```

(%o5) $-10.2$

```
(%i6)  F(x,y) := sin(x+y);
```

(%o6) $\mathrm{F}(x, y) := \sin(x + y)$

Wyliczenie wartości nowej funkcji F przy pomocy apply:

```
(%i7)  apply(F,[2,%e]);
```

(%o7) $\sin(e + 2)$

To samo bez konieczności wcześniejszego definiowania funkcji F.

```
(%i8)  apply(lambda([x,y],sin(x+y)),[2,%e]);
```

(%o8) $\sin(e + 2)$

### 6.4.3 Funkcja "map"

W najprostszej wersji funkcja "map" ma postać

```
map(f,[x1, x2, ... , xn])
```

Jej wynikiem jest lista postaci

```
[f(arg1), f(arg2), ... , f(argn)],
```

gdzie `arg1=x1, arg2=x2, ... , argn=xn`.

Argumenty wyrażenia są zawsze wyliczane, nawet w sytuacji gdy nie są wyliczane w innych kontekstach.

```
(%i9)  kill(all)$
```

```
(%i1)  map(f, [x, y, z]);
```

(%o1) $[\mathrm{f}(x), \mathrm{f}(y), \mathrm{f}(z)]$

Nic nie jest wyliczone ponieważ funkcja f nie jest zdefiniowana.

```
(%i2)  map(atom, [a, b, c, a + b, a + b + c]);
```

(%o2) $[true, true, true, false, false]$

Funkcja "atom" jest odwzorowana na podaną listę.

```
(%i3)  f(x):=x^2;
```

(%o3) $\mathrm{f}(x) := x^2$

```
(%i4)  map(f,[a,b,c,d]);
```

(%o4) $[a^2, b^2, c^2, d^2]$

Chcemy podnieść elementy listy do kwadratu. W tym celu definiujemy funkcję f i odwzorowujemy ją na listę.

```
(%i5)  map(lambda([x],x^2),[a,b,c,d]);
```

(%o5) $[a^2, b^2, c^2, d^2]$

To samo w jednym ruchu przy użyciu funkcji lambda.

*Uwaga.* Powyższe przykłady mają charakter ilustracyjny, więc są bardzo proste. Ale nawet z nich wynika ważna obserwacja. Chociaż można się obyć bez funkcji "apply" i "map", to warto ćwiczyć ich używanie, ponieważ dzięki nim kod jest krótszy i unikamy pętli programowych. To z kolei znakomicie poprawia czytelność i efektywność naszych procedur.

## 6.5 Zmiana standardowego procesu ewaluacji wyrażeń

### 6.5.1 Wstrzymywanie ewaluacji

Znak apostrofu (') umiejscowiony przed wyrażeniem wstrzymuje ewaluację tego wyrażenia. Dokładniej:

(a) 'x , gdzie x jest nazwą, wylicza się do x, bez względu na to co było przypisane do x wcześniej.

(b) 'f(a1, . . . , an) , gdzie f jest nazwą funkcji, wylicza się do f(ev(a1), . . . , ev(an)), to znaczy argumenty są wyliczane, natomiast definicja funkcji nie jest realizowana.

(c) '(. . .) wstrzymuje ewaluację dowolnego wyrażenia zawartego wewnątrz okrągłych nawiasów.

(d) Apostrof wstrzymuje ewaluację, ale nie ma wpływu na uproszczenia.

Przykłady:

```
(%i6)  kill(all);
```

(%o0) $done$

```
(%i1)  f(x, y) := y - x;
```

(%o1) $\mathrm{f}\left(x, y\right) := y - x$

```
(%i2)  a: %e; b: 17;
```

(%o2) $e$

(%o3) $17$

```
(%i4)  f(a, b);
```

(%o4) $17 - e$

Zgodnie ze standardową procedurą argumenty funkcji f są ewaluowane, a następnie wykonywana jest definicja funkcji.

```
(%i5)  f('a, 'b);
```

(%o5) $b - a$

W tym przypadku apostrofy wstrzymują wyliczenie argumentów i definicja funkcja jest realizowana dla argumentów a i b.

```
(%i6)  'f(a, b);
```

(%o6) $\mathrm{f}\left(e, 17\right)$

Tym razem apostrof wstrzymuje wykonanie funkcji, ale argumenty są wyliczane.

```
(%i7)  '(f(a, b));
```

(%o7) $\mathrm{f}\left(a, b\right)$

Użycie nawiasów wstrzymało wyliczanie argumentów i wartości funkcji.

```
(%i8)  '(sin(2+3));
```

(%o8) $\sin\left(5\right)$

Mimo użycia nawiasów apostrof nie wstrzymał zamiany 2+3 na 5. Przyczyną jest, że (w Maximie) działania arytmetyczne są uproszczeniami, a nie ewaluacjami.

Jest wiele sytuacji, w których niezbędne (bądź wygodne) jest użycie '. Na razie ograniczymy się do technicznej ciekawostki, która poprawia czytelność prowadzonych obliczeń.

```
(%i9)  integrate(1/(1+x^3),x);
```

(%o9) $$-\frac{\log\left(x^2-x+1\right)}{6}+\frac{\operatorname{atan}\left(\frac{2\,x-1}{\sqrt{3}}\right)}{\sqrt{3}}+\frac{\log\left(x+1\right)}{3}$$

Wartością komendy "integrate" jest całkę nieoznaczoną z zadanej funkcji względem zadanej zmiennej. Wynik jest czytelny natomiast odczytanie polecenia wymaga pewnej wprawy. Ponadto zadanie i wynik są w różnych wierszach.

```
(%i10) 'integrate(1/(1+x^3),x)=integrate(1/(1+x^3),x);
```

(%o10) $$\int\frac{1}{x^3+1}\,dx=-\frac{\log\left(x^2-x+1\right)}{6}+\frac{\operatorname{atan}\left(\frac{2\,x-1}{\sqrt{3}}\right)}{\sqrt{3}}+\frac{\log\left(x+1\right)}{3}$$

Teraz w pierwszej komendzie zakazaliśmy wyliczania funkcji `integrate` więc program zwrócił tylko nazwę funkcji (dzięki własnościom `wxMaximy` ta nazwa jest wyświetlana jako znak całki), po znaku równości nie ma apostrofu, więc całka jest wyliczana.

### 6.5.2 Wymuszanie dodatkowej ewaluacji

1. Dwa apostrofy (′) umieszczone jeden bezpośrednio za drugim przed wyrażeniem powodują dodatkową ewaluację wyrażenia. To oznacza, że wyrażenie zostaje wyliczone zgodnie ze standardową procedurą, a następnie otrzymana wartość ponownie jest wyliczana.

Zobaczmy to na przykładzie:

```
(%i11) kill(all);
```

(%o0) *done*

```
(%i1)  a : b;
```

(%o1) $b$

```
(%i2)  b : c;
```

(%o2) $c$

```
(%i3)  sin(a);
```

(%o3) $\sin(b)$

Wynik jest efektem standardowej procedury: argument a wylicza się do b, więc wynik to sin(b)

```
(%i4)  sin(''a);
```

(%o4) $\sin(c)$

Teraz a wylicza się do b, podwójny apostrof powoduje ponowne wyliczenie b, które ma wartość c.

```
(%i5)  ''sin(a);''(sin(a));
```

(%o5) $\sin(b)$

(%o6) $\sin(c)$

Te przykłady pokazują, że zakresy działania podwójnego apostrofu są podobne jak dla pojedyńczego. 2. Podobne rezultaty daje zastosowanie funkcji "ev"

```
(%i7)  ev(sin(a));
```

(%o7) $\sin(c)$

Jednak funkcja ev ma większe możliwości. Użycie opcji "eval" wymusza następną ewaluację.

```
(%i8)  c:d;
```

(%o8) $d$

```
(%i9)  ev(sin(a));
```

(%o9) $\sin(c)$

```
(%i10) ev(sin(a),eval);
```

(%o10) $\sin(d)$

Opcja "eval" może być powtórzona wielokrotnie, dając możliwość wymuszania dowolnej liczby ewaluacji danego wyrażenia. Powyższe uwagi to tylko mała część możliwościfunkcji "ev". Więcej na jej temat zawiera następna część.

## 6.6 Rozbudowane definicje funkcji

Do tej pory prawe strony definicji funkcji składały się z jednego wyrażenia. Oczywistym jest, że szybko natrafimy na potrzebę użycia bardziej skomplikowanych funkcji do określenia, których potrzeba wielu poleceń. Maxima udostępnia kilka możliwości definiowania takich funkcji.

Najprostsza to umieszczenie po prawej stronie definicji ciągu poleceń oddzielonych przecinkami i ujętych w nawiasy okrągłe. Wówczas rezultatem funkcji jest wartość ostatniego wyrażenia.

Przykład

```
(%i11) kill(all);
```

(%o0) *done*

```
(%i1)  f(x) := (a:x, b:sin(x), a+b);
```

(%o1) $\mathrm{f}\left(x\right) := \left(a : x, b : \sin\left(x\right), a + b\right)$

```
(%i2)  f(1);
```

(%o2) $\sin\left(1\right) + 1$

Sposób ten ma bardzo poważną wadę. Zmienne pomocnicze użyte w definicji są zmiennymi globalnymi tzn. na ich wartości mają wpływ wcześniejsze przypisania, a co ważniejsze, przypisania dokonane w trakcie wykonywania fumkcji zostają zachowane po jej wykonaniu. Sprawdźmy

```
(%i3)  a;b;
```

(%o3) $1$

(%o4) $\sin\left(1\right)$

Takie zachowanie jest częstym źródłem błędnych wyników w trakcie dalszego ciągu sesji z Maximą. Dlatego bezpieczniejszym sposobem jest używanie konstrukcji, w których zmienne pomocnicze są lokalne, tzn. przypisania dokonane w trakcie wykonania funkcji są anulowane po jej zakończeniu.

Poznamy dwie takie konstrukcje:

- tzw. lokalne środowisko obliczeniowe zadawane przez funkcję "ev"
- oraz funkcję "block", która pozwala budować procedury w sposób podobny do klasycznych języków programowania.

Ze względu na znaczenie tych konstrukcji poświęcimy im osobne części.

# Rozdział 7

# Lokalne środowisko obliczeniowe "ev"

Wszystkie operacje w Maximie są wykonywane w pewnym (globalnym) środowisku, tzn. ustalone są wartości zmiennych, definicje funkcji, poczynione założenia, itp. Często nasze obliczenia wymagają chwilowej zmiany tego środowiska. Aby uniknąć konfliktów nazw należy możliwie szybko wrócić do wyjściowego środowiska (np. wyczyścić niepotrzebne zmienne czy funkcje), co bywa uciążliwe. Funkcja "ev" dostarcza możliwości wykonania obliczeń w lokalnym środowisku, które istnieje tylko w czasie wykonywania pojedynczej komendy i nic nie zmienia w skali globalnej.

Według opinii twórców programu to jedna z bardziej użytecznych funkcji programu, chociaż opanowanie jej niuansów wymaga programistycznej wiedzy i doświadczenia. Może dlatego opinie na współczesnych forach dyskusyjnych nie są już tak entuzjastyczne, a *część ekspertów zaleca użycie rozwiązań alternatywnych do* `ev`.

Poniżej opisano część funkcjonalności funkcji "ev". Pełny opis można znaleźć w "Maxima Pomoc" pod hasłem "ev". Decyzję czy i w jakim stopniu używać tej funkcji pozostawiamy czytelnikowi.

Ogólna składnia tej funkcji jest następująca:

```
ev(wyr, arg\_1, ..., arg\_n).
```

Funkcja "ev" oblicza wyrażenie "wyr" w środowisku określonym przez argumenty arg_1, ..., arg_n. Argumenty mogą być (m.in.) tzw. przełącz-

nikami (funkcje boolowskie),przypisanianiami, podstawieniami, równaniami, funkcjami matematycznymi i programowymi. Lista wybranych dopuszczalnych argumentów znajduje się poniżej.

Argumenty ‘ev’ mogą być podane w dowolnej kolejności za wyjątkiem rownań zadających podstawienia, które są wykonywane w kolejności od lewej do prawej oraz funkcji, ktore są składane, np. ‘ev (<expr>, ratsimp, realpart)’ daje w wyniku ‘realpart (ratsimp(<expr>))’.

Wartością funkcji "ev" jest wyrażenie będące wynikiem obliczeń.

Obliczenia prowadzone są sukcesywnie w następującej kolejności:

# Krok 1

Zadane jest lokalne środowisko poprzez uwzględnienie argumentów, które mogą mieć następującą postać:

## Przypisania

`V:wart` (lub alternatywnie `V=wart`) powoduje, że do ‘V’ jest przypisana lokalnie (chwilowo) wartość ‘wart’ podczas obliczania wyrażenia.

Jeżeli więcej niż jeden argument ‘ev’ jest tego typu wówczas przypisanie jest wykonywane "równolegle".

Jeżeli ‘V’ jest nie-atomowym wyrażeniem wówczas wykonywane jest (raczej) podstawienie, a nie przypisanie.

*Uwaga.* Słowo przypisanie rezerwujemy dla zmiany wartości wyrażenia, słowo podstawienie związane jest ze zmianą kształtu samego wyrażenia.

```
(%i1)  ev(a*x+b*y+c*z, a=2,b=a,c=1);
```

(%o1) $z + a\,y + 2\,x$

Ilustracja "równoległości" przypisywania wartości. Zmienna b została astąpiona przez a mimo, że wcześniej a nadano wartość 2.

```
(%i2)  ev(a*x+b*y+c*z, a=2,b=a,c=1,eval);
```

(%o2) $z + 2\,y + 2\,x$

Wymuszenie uwzględnienia wartości a poprzez dodatkowe wyliczenie.

```
(%i3)  ev(a*x+b*y+c*z, a=3,a=2,c=1);
```

(%o3) $z + b\,y + 3\,x$

Uwzględnione jest tylko pierwsze przypisanie.

```
(%i4)  ev(a*x+b*y+c*z, a=2,a=3,c=1);
```

(%o4) $z + b\,y + 2\,x$

Przypisania wykonywane są od lewej do prawej.

```
(%i5)  ev(sin(2*x+y), 2*x=z);
```

(%o5) $\sin\left(z + y\right)$

Podstawienie za 2*x symbolu z.

## Uproszczenia

Jeżeli argument "ev" jest nazwą jednej z funkcji: expand, factor, fullratsimp, logcontract, polarform, radcan, ratexpand, ratsimp, rectform, rootscontract, trigexpand, trigreduce, to wyrażenie jest przekształcane ("upraszczane") zgodnie ze specyfiką danej funkcji. Przykłady są podobne do analizowanych w czści poświęconej funkcjom upraszczającym.

## Konwersje

Jeżeli jako argumentu użyjemy nazwy float to liczby wymierne i liczby typu "big floats" są konwertowane do liczby dziesiętnej typu float (liczba dziesiętna w arytmetyce 16-cyfrowej) niektóre liczby niewymierne są przybliżane liczbą typu float.

```
(%i6)  ev(4/7*x^2+sqrt(2)*x+2,float);
```

(%o6) $0.57142857142857\,x^2 + 1.414213562373095\,x + 2$

```
(%i7)  float(4/7*x^2+sqrt(2)*x+2);
```

(%o7) $0.57142857142857\,x^2 + 1.414213562373095\,x + 2.0$

Jak widać działanie funkcji "float" i argumentu funkcji "ev" o tej samej nazwie jest podobne, ale nie daje w pełni identycznych wyników (flaga float ma skromniejszy zakres).

```
(%i8)  ev([exp(2),sin(1),%pi],float);
```

(%o8) $[e^2, \sin(1), \pi]$

```
(%i9)  float([exp(2),sin(1),%pi]);
```

(%o9) $[7.38905609893065, 0.8414709848079, 3.141592653589793]$

Te liczby niewymierne nie podlegają konwersji przy pomocy flagi float (ale funkcja float spełnia swoją rolę. Mocniejszą wersją flagi float jest flaga numer, co pokazuje poniższy przykład:

```
(%i10) ev([exp(2),sin(1),%pi],numer);
```

(%o10) $[7.38905609893065, 0.8414709848079, 3.141592653589793]$

Flafa bfloat konwertuje wszystkie liczby i wartości funkcji o argumentach liczbowych do liczb dziesiętnych typu bfloat.

```
(%i11) ev([exp(2),sin(1),%pi],fpprec = 30, bfloat);
```

(%o11) $[7.38905609893065022723042746058b0, 8.4147098480789650665250232163b-1, 3.14159265358979323846264338328b0]$

Zauważmy, że dokładność arytmetyki została zmieniona tylko lokalnie

```
(%i12) fpprec;
```

(%o12) 16

## Modyfikacja sposobu ewaluacji wyrażenia

**eval** – wymusza dodatkowe obliczenie wyrazenia (zob. krok (5) ponizej). ‘eval’ może występować wielokrotnie, za kazdym razem powodując dodatkowe obliczenie wyrażenia.

**noeval** – zawiesza obliczenia przy wykonywaniu "ev" (zob. krok 4 poniżej). Bywa użyteczne w połączeniu z innymi argumentami, również przy potrzebie uproszczenia wyrażenia bez ponownego obliczania.

**nazwa funkcji** – jeżeli argumentem ev jest nazwa funkcji występująca w wyrażeniu (np. diff, sum, integrate), to ta funkcja jest wyliczana, nawet jeżeli przed nią stoi operator ’.

**nouns** – wymusza obliczenie wszystkich operatorów z wstrzymanym obliczeniem przez użycie apostrofu ’ zawartych w wyrażenie.

```
(%i13) ev('sum(k^2, k, 1, 10), eval);
```

$$\text{(\%o13)} \quad \sum_{k=1}^{10} k^2$$

### Definiowanie funkcji wewnątrz "ev"

Funkcje występujące w "wyrazenie" (powiedzmy ‘F(x)’) mogą być definiowane lokalnie w postaci ‘F(x) :=prawa_strona’ jako argument polecenia ‘ev’.

### Wyrażenia jako argumenty "ev"

Jeżeli atom lub wyrazenie nieatomowe (nie wymienione wyżej) wystepują jako argument, to są one wyliczane.

Jezeli wynik jest równaniem lub przypisaniem, to wtedy jest on uwzględniany w dalszych obliczeniach związanych z "ev" .

Jeżeli wynik jest listą, to elementy listy są traktowane jako dodatkowe argumenty polecenia ‘ev’. To umożliwia użycie np. listy równań otrzymanych w wyniku komendy ‘solve’.

## Krok 2

Podczas wykonywania kroku (1) tworzona jest lista zmiennych znajdujacych sie po lewej stronie rownań bedących argumentami. Wszystkim zmiennym globalnym wystepującym w wyrażeniu nadawana jest ich wartość.

## Krok 3

Jezeli argumenty wskazują na podstawienie jest ono teraz wykonywane.

## Krok 4

Otrzymane wyrażenie jest ponownie wyliczane (chyba, że jeden z argumentów jest ‘noeval’) i upraszczane zgodnie z wartością argumentów funkcji ev. Pamietajmy, że przed wyliczeniem wartości funkcji w wyrażeniu wyliczane są wartości jej argumentów, czyli ‘ev(F(x))’ zachowuje sie jak ‘F(ev(x))’.

## Krok 5

Dla każdego wystapienia argumentu ‘eval’ kroki (3) i (4) sa powtarzane.

```
(%i14) ev (sin(x) + cos(y) + (w+1)^2 + 'diff (sin(w), w),
       numer, expand, diff, x=2, y=1);
```

(%o14) $\cos(w) + w^2 + 2\,w + 2.449599732693821$

## Alternatywny zapis funkcji "ev"

Wygodnym (i często używanym) jest alternatywny zapis polecenia ‘ev’ polegajacy na opuszczeniu ‘ev()’, tzn. można pisać po prostu

```
wyrazenie, arg\_1, ..., arg\_n
```

*Uwaga.* Taki zapis nie może być częścią innego wyrażenia (np. funkcji, bloku, itp.)

# Rozdział 8

# Programowanie

Alternatywą dla lokalnego środowiska obliczeniowego funkcji "ev" jest użycie języka programowania `Maximy`, a w szczególności konstrukcji `block`.

Język programowania `Maximy` jest zbliżony do popularnych języków proceduralnych (`Pascal`, `C`, itp.), więc doświadczenia zdobyte przy okazji nauki tych języków mogą być wykorzystane w Maximie. Należy jedynie zwrócić uwagę na poprawność składni i pamiętać o specyfice przekształcania wyrażeń.

## 8.1 Procedura "block"

Podstawową komendą, której używamy do budowy nowych modułów (funkcji,procedur) jest polecenie "block":

```
block([var\_1, ... , var\_m], <expr\_1>, ..., <expr\_n>)
```

— Funkcja "block" wylicza sekwencyjnie wyrażenia <expr_1>, ..., <expr_n>. Proszę zwrócić uwagę, wewnątrz bloku wyrażenia odzielane są przecinkami (nie średnikami!)

— Wynikiem tej funkcji jest wartość ostatniego wyliczonego wyrażenia.

— Kolejność wyliczania wyrażeń może być modyfikowana przez funkcję "return" (również przez funkcje "go", "throw", ale o nich w tej chwili nie będziemy mówić).

— Funkcja `return(<value>)` może być użyta do bezpośredniego wyjścia z bloku z wartością wyrażenia będącego jej argumentem>

— Zmienne var_1, ... , var_m, podane w formie listy jako pierwszy argument, są lokalne względem bloku, tzn. ich wartość nie wpływa na wartość zmiennych globalnych o tych samych nazwach. Jeżeli nie ma zmiennych lokalnych pierwszy argument może zostać pominięty.

— Wewnątrz bloku wszystkie zmienne różne od var_1, ... , var_m są globalne (ich wartości są zachowane nawet po zakończeniu wykonywania bloku).

— Najczęściej funkcji "block" używamy jako prawej strony definicji nowej funkcji, ale może być również używana samodzielnie.

Przykłady:

```
(%i1)  f1(a1,b1) := block(a : a1^2,
                b : b1^2,
                a*x^4+b*x^3+a1*x^2+b1*x);
```

(%o1) $\mathrm{f1}\left(a1, b1\right) := \mathrm{block}\left(a : a1^2, b : b1^2, a\,x^4 + b\,x^3 + a1\,x^2 + b1\,x\right)$

Prosty przykład funkcji definiowanej przy pomocy block.

```
(%i2)  f1(3,2);
```

(%o2) $9\,x^4 + 4\,x^3 + 3\,x^2 + 2\,x$

```
(%i3)   f1(x,y);
```

(%o3) $x^3\,y^2 + x\,y + x^6 + x^3$

Wynikiem jest wartość ostatniego wyrażenia z uwzględnieniem wcześniejszych przypisań

```
(%i4)  a :2;
```

(%o4) 2

W tej chwili a jest globalną zmienną o wartości 2.

```
(%i5)  f1(u,v);
```

(%o5) $u^2\,x^4+v^2\,x^3+u\,x^2+v\,x$

```
(%i6)  a;
```

(%o6) $u^2$

Widzimy, że wykonanie funkcji (bloku) zmodyfikowało wartość globalnej zmiennej! Jest to źródło częstych błędów, trudnych do odszukania. Bez istotnego powodu nie należy używać zmiennych globalnych wewnątrz bloku, a raczej zadeklarować je jako zmienne lokalne.

Poprawiona definicja funkcji:

```
(%i7)  f2(a1,b1) := block([a,b],a : a1^2,
                           b : b1^2,
                           a*x^4+b*x^3+a1*x^2+b1*x);
```

(%o7) $\mathrm{f2}\left(a1,b1\right):=\mathrm{block}\left(\left[a,b\right],a:a1^2,b:b1^2,a\,x^4+b\,x^3+a1\,x^2+b1\,x\right)$

```
(%i8)  a : 2;
```

(%o8) 2

```
(%i9)  f2(u,v);
```

(%o9) $u^2\,x^4+v^2\,x^3+u\,x^2+v\,x$

```
(%i10) a;
```

(%o10) 2

Widzimy, że na wartość zmiennej globalnej a nie wpłynęła wartość zmiennej lokalnej o tej samej nazwie.

## 8.2 Funkcje wyświetlające wyniki

Wartością funkcji "block" jest (na ogół) wartość ostatniego wyrażenia. Jednak często chcemy wyświetlić wyniki uzyskane w trakcie wykonywania bloku, które nie są związane z ostatnim wyrażeniem. Poniżej omawiamy najczęściej używane w tym celu polecenia.

### 8.2.1 Funkcja print

```
print(<expr_1>, ..., <expr_n>)
```

Wylicza i wyświetla kolejno wartości wyrażeń <expr_1>, ..., <expr_n> w jednej (lub więcej) liniach. Wartością "print" jest wartość ostatniego wyrażenia.

```
(%i11) kill(a,b);
```

(%o11) *done*

```
(%i12) r: print ("(a+b)^3 to ", expand ((a+b)^3),
       "log(a^10/b) to ", radcan (log (a^10/b)))$
```

```
(a+b)^3 to b^3+3ab^2+3a^2b+a^3
log (a^10/b) to 10log(a) - log(b)
```

```
(%i13) r;
```

(%o13) $10\log(a) - \log(b)$

Proszę nie mylić wyświetlanego napisu, który jest tzw. efektem ubocznym z wartością funkcji "print".

### 8.2.2 Funkcja display

```
display(<expr_1>, ..., <expr_n>)
```

Wyświetla równania, których lewe strony są niewyliczonymi wyrażeniami <expr_1>, ..., <expr_n>, a prawymi stronami są wyliczone wartości tych wyrażeń.

W jednej linii znajduje się jedno równanie.

Wartością funkcji jest słowo "done".

Jest to wygodne narzędzie do wyświetlania pośrednich wyników wewnątrz bloku, w szczególności dla pętli "for".

```
(%i14) x:123$

(%i15) display(x, sin(1.0));


x=123
sin(1.0) =0.8414709848079

(%o15) done
```

### 8.2.3 Funkcja disp

```
disp(<expr_1>, ..., <expr_n>)
```

Funkcja, której działanie jest podobne do "display", ale wyświetla tylko wartości wyrażeń.

```
(%i16) x:123$

(%i17) disp(x, sin(1.0));

1230.8414709848079

(%o17) done
```

## 8.3 Instrukcje warunkowe

Ogólna idea budowania instrukcji warunkowych jest podobna do znanej z innych języków programowania.

### 8.3.1 Funkcja if-then-else

```
if <warunek> then <akcja1> else <akcja2>
```

Działanie: jeżeli <warunek> wylicza się do wartości true, to akcja1, jeżeli do wartości false, to akcja2.

```
(%i18) f(x,y):=if x<y then print(x, " jest mniejsze od ", y)
       else print(x, " jest większe lub równe od ", y)$

(%i19) f(1,2);


1 jest mniejsze od 2

(%o19) 2

(%i20) f(1,1);


1 jest większe lub równe od 1

(%o20) 1
```

Jeden ze sposobów definicji liczb Fibonnaciego

```
(%i22) fib(n):=if n = 1 or n = 2 then 1 else fib(n-2)+fib(n-1)$

(%i23) fib(4);

(%o23) 3

(%i24) fib(10);

(%o24) 55
```

### 8.3.2 Funkcja if-then-elseif-then-else

Składnia:

```
if <warunek 1> then <akcja 1> elseif <warunek 2>
then <akcja 2> else <akcja 3>
```

Działanie:

Jeżeli <warunek 1> wylicza się do wartości true, to <akcja 1>.

Jeżeli <warunek 2> wylicza się do wartości true, to < akcja 2>.

W pozostałych przypadkach <akcja 3>.,

"if-then-elseif-then-else" może mieć więcej niż jedną "elseif-then" składową.

```
(%i25) kill(all);
```

(%o0) *done*

```
(%i1)  f(mu,nu):=if mu = nu then mu else (if mu > nu
       then mu-nu else nu+mu);
```

(%o1) $\mathrm{f}\left(\mu,\nu\right):=\text{if}\quad \mu=\nu\quad \text{then}\quad \mu\quad \text{else}\quad \text{if}\quad \mu>\nu\quad \text{then}\quad \mu-\nu\quad \text{else}\quad \nu+\mu$

## 8.4 Iteracje

### 8.4.1 Operator "do"

Podstawowym operatorem do tworzenia iteracji jest specjalny operator "do":

```
do (polecenie_1,...,polecenie_n)
```

Po nazwie operatora dajemy *obowiązkową spację*, a następnie piszemy ciąg poleceń oddzielonych przecinkami i ujętych w nawiasy okrągłe. Działanie operatora polega na nieskończonym wykonywaniu w pętli ciągu poleceń zawartych w nawiasach okrągłych, dlatego wsród poleceń musi być takie, które wymusi zakończenie pętli (najczęściej jest to "return").

Jako przykład zbudujmy funkcję, która liczy iloczyn kolejnych liczb naturalnych od n do m (oczywiście jest to ilustracja komendy "do", ten sam cel można zrealizować na wiele sposobów, w tym bezpośrednio używając funkcji "product").

```
(%i2)  iloczyn(n,m):= block(
            [i:n, k:n],
       /*i,k zm. lokalne z przypisanymi wartościami początk.*/
             /* pętla do */
            do (
                k: k*(i+1),
                if i+1=m then return(k) else i : i+1
                /*warunek zakończenia pętli*/
               )
       )$
```

Pamiętajmy, że w każdym języku programowania formatowanie kodu pomaga w uniknięciu pomyłek.

```
(%i3)  iloczyn(3,5);
```

(%o3) 60

Przy testowaniu tej funkcji bardzo łatwo wpaść w nieskończoną pętlę (np. wpisując niefrasobliwie iloczyn(5,3). Jeżeli tak się stanie, proszę przerwać obliczenia wybierając z menu Maxima->Przerwij (lub Ctrl-G)

Ze względu na to ryzyko rzadko używamy operatora "do" samodzielnie. Najczęściej jest on składnikiem konstrukcji pętli "for" znanej z innych języków programowania. W dalszym ciągu poznamy podstawowe warianty "for" dostępne w Maximie.

### 8.4.2 Funkcja for-from-step-thru-do

Składnia:

```
for <zmienna> from <wartość_początkowa>
step <krok> thru <wartość_końcowa> do <instrukcje>
```

Alternatywnie:

```
  for <zmienna> : <wartość_początkowa> step <krok>
  thru <wartość_końcowa> do <instrukcje>
```

```
for <zmienna> : <wartość_początkowa>  thru
<wartość_końcowa> step <krok> do <instrukcje>
```

Jeżeli <krok> jest 1, to fragment step 1 może być pominięty.

<wartość_początkowa>, <krok>, <wartość_końcowa>, <instrukcje> mogą być dowolnymi wyrażeniami.

W pierwszym kroku <wartość_początkowa> jest przypisana do <zmienna>. Następnie:

1. jeżeli <zmienna> jest większa niż <wartość_końcowa> pętla kończy działanie,
2. w przeciwnym przypadku zestaw poleceń <instrukcje> jest wykonywany,
3. <krok> jest dodawany do <zmienna>

Kroki (1)-(3) są powtarzane aż <zmienna> jest większa niż <wartość_końcowa>.

*Uwaga*: <zmienna> jest zawsze lokalna w stosunku do operatora "do". Dlatego możemy tu użyć dowolnej nazwy bez obawy zmodyfikowania zmiennych globalnych.

```
(%i4)  for a:-3 thru 26 step 7 do print("a = ", a)$

a = -3
a = 4
a = 11
a = 18
a = 25
```

### 8.4.3 Funkcja for-from-step-while-do

Składnia:

```
  for <zmienna> from <wartość_początkowa> step
<krok> while <warunek> do <instrukcje>
```

Alternatywnie:

```
 for <zmienna> : <wartość_początkowa> step <krok>
while <warunek> do <instrukcje>
```

```
  for <zmienna> : <wartość_początkowa>  while <warunek> step <krok> do <inst
```

Jeżeli <krok> jest 1, to fragment step 1 może być pominięty.

<wartość_początkowa>, <krok>, <instrukcje> mogą być dowolnymi wyrażeniami.

W pierwszym kroku <wartość_początkowa> jest przypisana do <zmienna>. Następnie:

1. jeżeli <warunek> wylicza się do wartości false pętla kończy działanie,
2. w przeciwnym przypadku zestaw poleceń <instrukcje jest wykonywany,
3. <krok> jest dodawany do <zmienna>

Kroki (1)-(3) są powtarzane aż <warunek> wyliczy się do wartości false.

*Uwaga*: <zmienna> jest zawsze lokalna w stosunku do operatora "do".

```
(%i5)  s: 0$
        for i: 1 while i <= 10 do s: s+i;
(%o6)  done
```

### 8.4.4 Funkcja for-from-step-unless-do

Składnia:

```
 for <zmienna> from <wartość_początkowa> step
<krok> unless <warunek> do <instrukcje>
```

Alternatywnie:

```
for <zmienna> : <wartość_początkowa> step <krok>
unless <warunek> do <instrukcje>
```

```
for <zmienna> : <wartość_początkowa> unless
<warunek> step <krok> do <instrukcje>
```

Jeżeli <krok> jest 1, to fragment step 1 może być pominięty.

<wartość_początkowa>, <krok>, <instrukcje> mogą być dowolnymi wyrażeniami.

W pierwszym kroku <wartość_początkowa> jest przypisana do <zmienna>. Następnie:

1. jeżeli <warunek> wylicza się do wartości true pętla kończy działanie,
2. w przeciwnym przypadku zestaw poleceń <instrukcje jest wykonywany,
3. <krok> jest dodawany do <zmienna>

Kroki (1)-(3) są powtarzane aż <warunek> wyliczy się do wartości true.

*Uwaga*: <zmienna> jest zawsze lokalna w stosunku do operatora "do".

Dla przykładu policzymy wielomian Taylora stopnia 7 dla zadanej funkcji.

```
(%i12) szereg:1; wyraz:exp(sin(x));
```

(%o12) 1

(%o13) $e^{\sin(x)}$

```
(%i14) for p from 1 unless p > 7 do
         (wyraz : diff(wyraz,x)/p,
          szereg:subst(x = 0,wyraz)*x^p+szereg
         );
```

(%o14) *done*

Standardową wartością dla "do" jest atom "done". Ale zmienna "szereg" jest globalna, więc możemy ją w każdej chwili wyliczyć:

```
(%i15) szereg;
```

$$(\%o15)\quad \frac{x^7}{90}-\frac{x^6}{240}-\frac{x^5}{15}-\frac{x^4}{8}+\frac{x^2}{2}+x+1$$

### 8.4.5 Funkcja for-in-do

Ciekawą formą pętli "for" jest iteracja, gdzie do zmiennej nie dodajemy stałej wartości <krok>, ale przypisujemy jej w każdym kroku iteracji wyrażenia będące kolejnymi elementami podanej listy:

```
for <zmienna> in <lista>  do <instrukcje>
```

Przykład:

```
(%i16) kill(all);
```

(%o0) *done*

```
(%i1)  for f in [log,sin,atan] do disp(f(1.0))$
```

0.00.841470984807900.78539816339745

### 8.4.6 Uwagi końcowe

Powyżej przedstawiliśmy jedynie ogólne uwagi na temat podstawowych poleceń wykorzystywanych przy konstrukcji bardziej rozbudowanych funkcji z użyciem "block". Ich efektywne użycie jest kwestią praktyki, którą będziemy zdobywać analizując i budując bardziej złożone przykłady.

Warto pamiętać, że wiele konstrukcji, które w pierwszym odruchu kojarzą się z pętlą "for" mogą być zrealizowane przy pomocy funkcji "create_list", "makelist", "map", "apply", "sum", "product", itp. Takie rozwiązania dają zawyczaj bardziej przejrzysty i efektywny kod.

# Rozdział 9

# Podstawowa grafika 2D i 3D

Przy wykonywaniu procedur graficznych Maxima współpracuje z zewnętrznymi programami, które praktycznie umożliwiają realizację każdego projektu. Z tego punktu widzenia, polecenia graficzne Maximy są rodzajem interfejsu między Maximą, a danym programem.

Te interfejsy (funkcje graficzne) mają różny stopień złożoności - od najprostszych ukierunkowanych na rysowanie wykresów funkcji, do bardzo skomplikowanych umożliwiających wykorzystanie pełnej mocy skojarzonego programu.

W tym notatniku ograniczymy się do opisu podstawowych funkcji, które od lat udostępnia program Maxima i dostępnych bezpośrednio z menu wxMaximy. Procedury te realizują podstawowe cele przy wizualizacji obiektów matematycznych, jednak ich możliwości są mocno ograniczone. Dlatego w ostatnich latach powstały dodatkowe biblioteki poleceń graficznych o znacznie większym zakresie zastosowań. jednej z takich bibliotek poświęcimy osobny notatnik.

Programem zewnętrznym, którego będziemy używać do realizacji procedur graficznych jest "gnuplot".

Specyfiką interfejsu wxMaxima jest to, że omawiane polecenia mogą występować w dwóch wersjach: - z przedrostkiem "wx" , wówczas rysunek zostaje wklejony do notatnika i po wyborze rozszerzenia "wxmx" zachowany na wskazanym nośniku, - bez przedrostka, wówczas program gnuplot otwiera nowe okienko, w którym prezentowany jest rysunek. Aby wrócić do sesji Maximy trzeba zamknąć okienko gnuplot.

Każda z tych wersji ma swoje zalety i wady. Funkcjonalność i zakres możliwści formatu "gnuplot" są znacznie większa niż formatu wewnętrznego. Natomiast wygoda tworzenia i ewentualnego modyfikowania notatnika jest niewątpliwie po stronie formatu wewnętrznego. Wybór zależy od kontekstu i użytkownika. Ze względu na specyfikę opracowania, w większości przypadków będziemy używać wersji z przedrostkiem "wx".

## 9.1 Grafika 2D

Funkcje graficzne zgromadzone są w menu pod nazwą "Kreślenie". Pole to zawiera trzy pozycje , "Wykres 3D...", "Format wykresu". Zaczynamy od poleceń związanych z pozycją "Wykres 2D...".

### 9.1.1 Składnia polecenia wxplot2d (lub plot2d)

Funkcja:

```
wxplot2d(<rys>, <arg_zakres>, ... ,<opcje>, ...)
```

rysuje wykres jednej funkcji jednej zmiennej.

Funkcja:

```
wxplot2d([<rys_1>, ..., <rys_n>], <x_zakres>, ...,
<opcje>, ...)
```

rysuje n wykresów funkcji jednej zmiennej na jednym rysunku.

Pierwszy argument `<rys>` lub elementy listy `[<rys_1>, ..., <rys_n>]` mogą być:

- wyrażeniami, np. x^2, sin(x), exp(log(x)+x;
- nazwami funkcji, np. sin, cos
- listami postaci:
  - `[discrete, [<x1>, ..., <xn>], [<y1>, ..., <yn>]]`, (wykres dyskretny)

- `[discrete, [[<x1>, <y1>],..., [<xn>, ..., <yn>]]`, (inne zadanie punktów do wykresu dyskretnego)
- `[parametric, <x_expr>, <y_expr>, <t_range>]` (wykres krzywej zadanej parametrycznie)

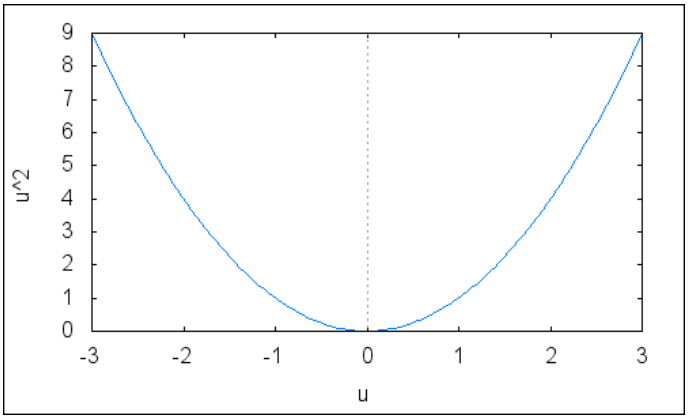
Drugi argument `<arg_zakres>`, dla funkcji zadanej poprzez wyrażenie lub nazwę, jest obowiązkowy. Jego postać to lista `[arg_funkcji, min, max]`, gdzie "min" i "max" określają przedział, nad którym rysujemy wykres (lub wykresy).

Dalsze argumenty pod wspólną nazwą `<opcje>` są potrzebne jeżeli chcemy modyfikować standardowy wygląd rysunku. Przykłady opcji i ich składnia zostaną podane poniżej.

### Wykresy funkcji zadanych jawnie

Najprostsza postać funkcji `wxplot2d` :

```
(%i1)  wxplot2d( u^2, [u,-3,3]);
```

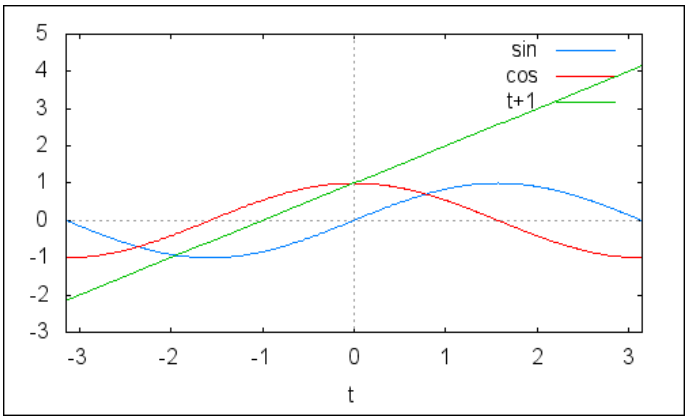



(%t1)

(%o1)

Kilka wykresów zadanych wyrażeniem i nazwami funkcji:

```
(%i2)  wxplot2d( [sin, cos, t+1], [t,-%pi,%pi]);
```




(%t2)

(%o2)

### Wykresy punktowe

Przy pomocy funkcji `wxplot2d` możemy również ilustrować dane eksperymentalne, interpretować geometrycznie ciągi liczbowe oraz inne obiekty, które przedstawiamy przy pomocy punktów na płaszczyźnie.

W tym celu wybieramy wariant pierwszego argumentu, który jest listą zawierającą słowo kluczowe `discrete` oraz współrzędne punktów, które mają być narysowane:

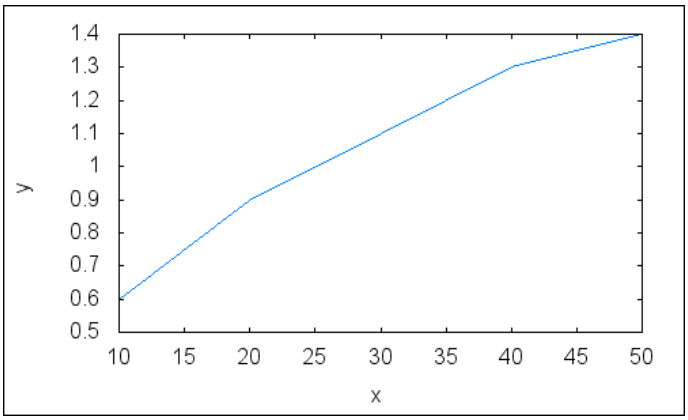
```
[discrete, [<x1>, ..., <xn>], [<y1>, ..., <yn>]]
```

lub

```
[discrete, [[<x1>, <y1>],..., [<xn>, ..., <yn>]]
```

W pierwszym przypadku lista `[<x1>, ..., <xn>]` zawiera odcięte punktów, natomiast lista `[<y1>, ..., <yn>]` odpowiadające im rzędne. W drugim przypadku lista `[<x1>, <y1>],..., [<xn>, ..., <yn>]` zawiera współrzędne kolejnych punktów. Przykład:

```
(%i3)  wxplot2d ([discrete,
                      [10, 20, 30, 40, 50],
                      [.6, .9, 1.1, 1.3, 1.4]
                     ])$
```

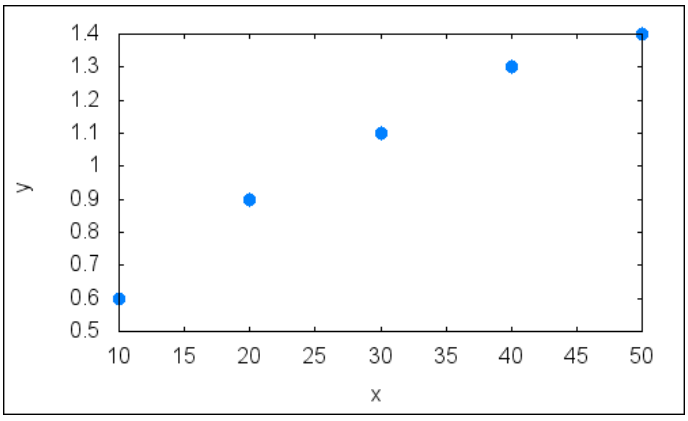



(%t3)

Dla czytelności kod został sformatowany.

Jak widzimy standardowo punkty są łączone odcinkami. Aby uzyskać same punkty trzeba użyć opcji `[style, points]`. Zilustrujemy to używając drugiego wariantu zadawania punktów:

```
(%i4)  wxplot2d([discrete,
                  [ [10, .6], [20, .9], [30, 1.1],
                  [40, 1.3], [50, 1.4]]
                ],
                [style, points])$
```




(%t4)

Zwróćmy uwagę, że mimo znaku `$` na końcu mamy wyświetlony rysunek. Przyczyną jest to, że (podobnie jak `print`) polecenia graficzne wyświetlają rysunek jako efekt uboczny (nie są wartością tego wyrażenia). W szczególności, nie muszą być ostatnią komendą `block`.

**Krzywe zadane parametrycznie**

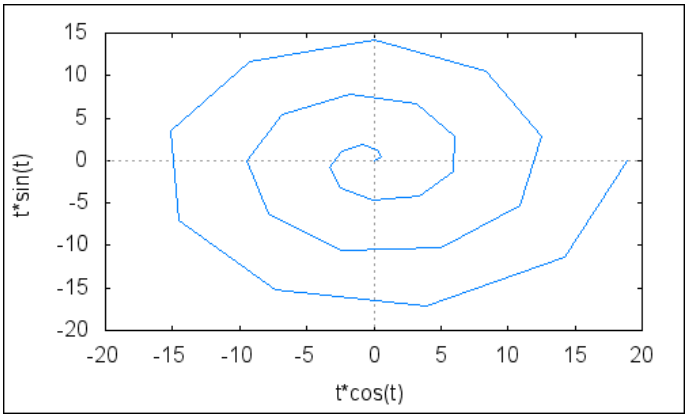
Jeżeli chcemy narysować krzywą płaską daną równaniami parametrycznymi (x(t),y(t)), gdzie parametr t zmienia się od t_0 do t_1, to używamy ostatniego wariantu pierwszego argumentu ze słowem kluczowym `parametric`:

```
[parametric, <x_expr>, <y_expr>,  <t_range>],
```

gdzie `<x_expr>`, `<y_expr>` są wyrażeniami zadającymi parametryzację, a `<t_range>` ma postać listy `[<param>, <min>,<max>]`.

Przykład:

```
(%i5)  wxplot2d([parametric,
       t*cos(t), t*sin(t),
       [t,0,6*%pi]]);
```




(%t5)

(%o5)

Wykres nie spełnia naszych oczekiwań. Żeby go poprawić i ewentualnie zmodyfikować musimy użyć właściwych opcji. Jakie mamy opcje do dyspozycji i jak ich używać stanowi treść następnego punktu.

## 9.2 Opcje

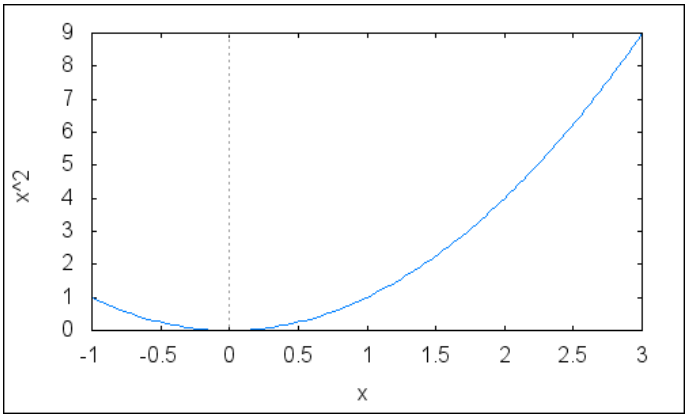
Wszystkie opcje są listami złożonymi ze słowa kluczowego oraz jednej lub kilku wartości. Poniej opiszemy kilka najczęściej używanych opcji .

### 9.2.1 Specjalna rola zmiennych x i y

Jak widzimy na przykładach rysunek zamknięty jest w prostokąt, którego wymiary są automatycznie dopasowane do wielkości wykresu. Aby zadać własne wymiary tego prostokąta, w ramach opcji podajemy dodatkowe zakresy dla zmiennych x i/lub y. Wyjaśnijmy to na przykładzie:

```
(%i6)  wxplot2d(x^2,[x,-1,3])$
```

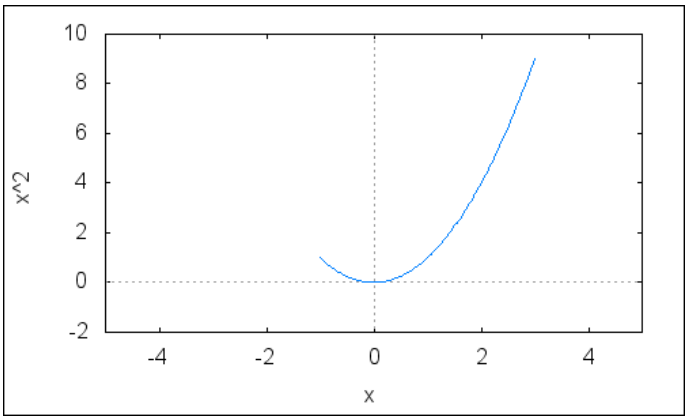
(%t6)




Rysunek bez dodatkowych opcji z automatycznie dobranym prostokątem.

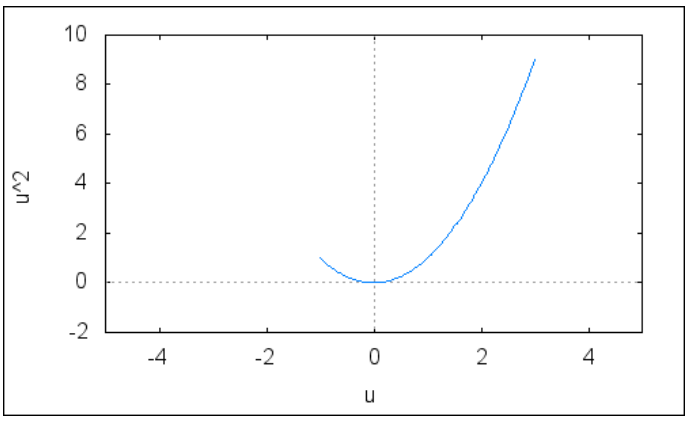
```
(%i7)  wxplot2d(x^2,[x,-1,3],[x,-5,5],[y,-2,10])$
```

(%t7)




Rysunek umieszczony w zadanym prostokącie. Zwróćmy uwagę na różne znaczenie zakresów zmiennej x - pierwszy określa zakres argumentu funkcji, drugi podaje wymiar prostokąta.

```
(%i8)  wxplot2d(u^2,[u,-1,3],[x,-5,5],[y,-2,10])$
```

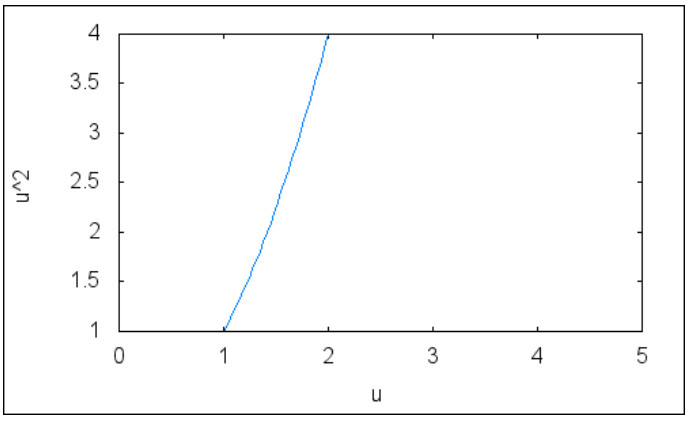



(%t8)

Oczywiście argument funkcji możemy nazwać dowolnie, natomiast w zakresach oznaczających wymiary prostokąta musimy użyć zmiennych x i y.

```
(%i9)  wxplot2d(u^2,[u,-1,3],[x,0,5],[y,1,4])$
```

*plot2d : somevalueswereclipped.*




(%t9)

Manewrując wymiarami prostokąta możemy wyświetlać fragmenty rysunku. Program ostrzega nas, że część rysunku została obcięta.

### 9.2.2 Opcje modyfikujące wygląd wykresów

#### Wygladzanie krzywej

`[nticks, <n>]`

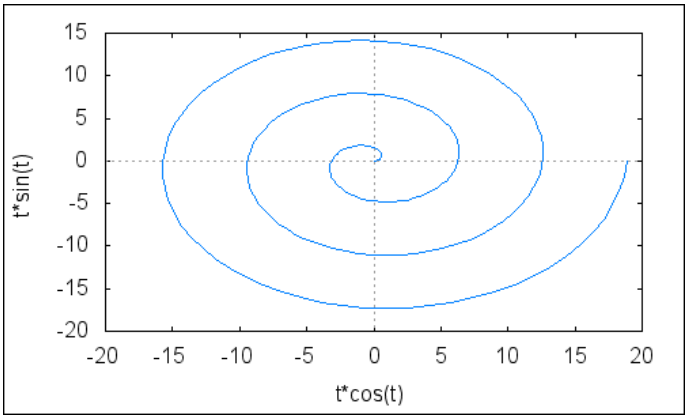
wskazuje liczbę punktów, które są brane pod uwagę przez procedury rysujące wykresy.

Standardowa wartość to 29. Jak widzieliśmy, im większe `nticks` tym wykres jest bardziej gładki (za cenę czasu wykonania).

Wróćmy do przykładu.

```
(%i10) wxplot2d([parametric,
          t*cos(t), t*sin(t),
          [t,0,6*%pi]
        ],
       [nticks, 200])$
```

(%t10)




Jest lepiej, ale rysunek jest zdeformowany, ponieważ stosunek jednostek osi x i y nie jest 1/1. Czasami jest to użyteczne, ale nie w tym przypadku.

**Furtka do gnuplot**

`gnuplot_preamble`

daje pełny dostęp do możliwości programu `gnuplot`. Na obecnym poziomie nie chcemy się w to zagłębiać, poza niezbędnymi wyjątkami. Przykładowo

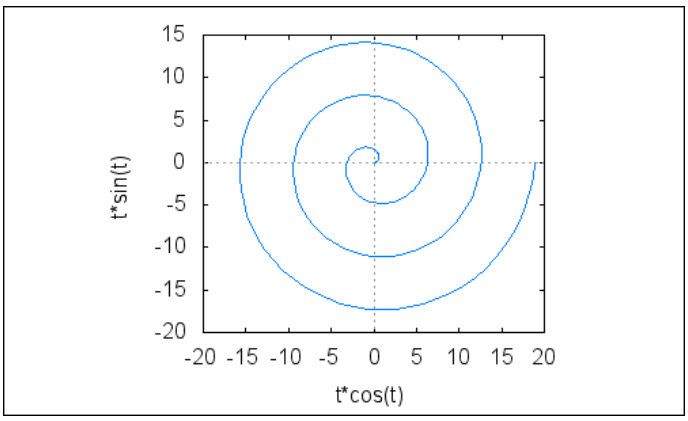
```
[gnuplot_preamble, "set size ratio -1"]
```

ustawia stosunek jednostek osi x i osi y na 1:1, co jest rozwiązaniem problemu.

Zastosujmy to do naszego przykładu.

```
(%i11) wxplot2d([parametric, t*cos(t), t*sin(t), [t,0,6*%pi] ],
                    [nticks, 200],
                    [gnuplot_preamble, "set size ratio -1"]) $
```

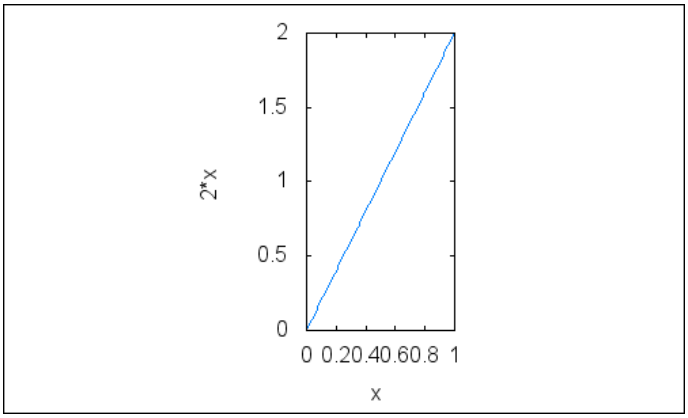
(%t11)




Warto podkreślić różnicę między "set size ratio -1", które ustawia stosunek jednostek osi na 1:1 oraz "set size ratio 1", które ustawia stosunek szerokości rysunku do długości na 1:1. Zobaczmy różnicę na przykładzie:

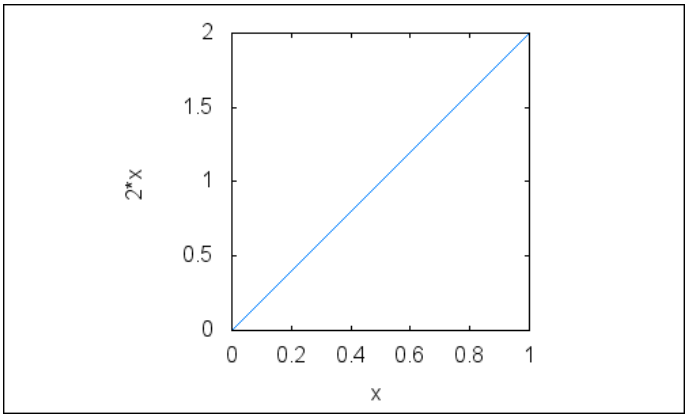
```
(%i12) wxplot2d(2*x,[x,0,1],
                [gnuplot_preamble, "set size ratio -1"]);
```

(%t12)




(%o12)

```
(%i13) wxplot2d(2*x,[x,0,1],
       [gnuplot_preamble, "set size ratio 1"]);
```




(%t13)

(%o13)

**Styl wykresu**

`[style, <style_1>, ..., <style_n>]`

opisuje jak przedstawiać wykresy lub punkty w grafice 2d. Po słowie kluczowym `style` musi nastąpić jeden lub więcej opisów stylów `<style_i>`. Każdy styl odpowiada kolejnemu wykresowi. Jeżeli wykresów jest więcej niż liczba stylów, to style są powtarzane w pętli. Styl `<style_i>` jest jest jednym z słów kluczowych:

- `lines` - wykres składa się z odcinków,
- `points` - wykres składa się z izolowanych punktów,
- `linespoints` - wykres jest rysowany przy pomocy odcinków i punktów,
- `dots` - wykres jest rysowany linią kropkowaną.

Każde z powyższych słów może mieć dodatkowe parametry, wtedy niezbędne jest ujęcie całego stylu w nawiasy kwadratowe.

Słowo `lines` może mieć jeden lub dwa dodatkowe parametry:

- pierwszy określa grubość linii,

- drugi jest liczbą naturalną określającą kolor linii.

Standardowe kody kolorów są następujące:

1 - blue, 2 - red, 3 - green, 4 - magenta, 5 - black, 6 - cyan.

Słowo `points` może mieć jeden, dwa lub trzy parametry:

- pierwszy określa promień obiektu rysowanego jako punkt,
- drugi kolor według tego samego schematu jak dla linii,
- trzeci rodzaj obiektu rysowanego jako punkt.

Mamy do dyspozycji następujące rodzaje obiektów:

1-bullet, 2-circle, 3-plus, 4-times, 5-asterisk, 6-box,7-square, 8-triangle, 9-delta, 10-wedge, 11-nabla,12-diamond,13-lozenge

(Jeżeli rodzaj kształtu nie jest jasny z angielskiej nazwy, proszę przeanalizować przykłady).

Słowo `linespoints` może mieć jeden, dwa, trzy lub cztery parametry: szerokość linii, promień punktu, kolor, rodzaj obiektu użytego jako punkt. Zadawanie parametrów jest analogiczne jak dla `lines` i `points`.

Standardowy styl (używany jeżeli nie podamy żadnej własnej opcji) to `lines` o grubości 1 i kolorze będącym na pierwszym miejscu w opcji `color`.

W przykładach automatycznie wyprodukujemy kombinacje różnych grubości i kolorów oraz przykładowe wykresy (zob.notatnik poświęcony listom).

Dla przejrzystości wyniki przypisujemy do zmiennych.

```
(%i14) grubosci_i_kolory : cons(style,
       makelist([lines,2*i,i],i,1,6));
```

(%o14) $[style, [lines, 2, 1], [lines, 4, 2], [lines, 6, 3], [lines, 8, 4], [lines, 10, 5], [lines, 12, 6]]$

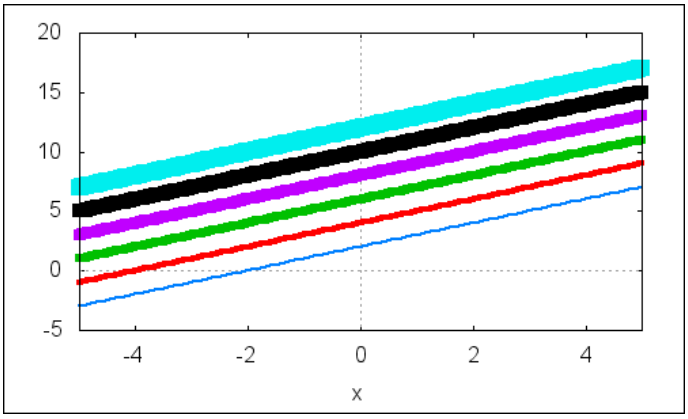
```
(%i15) linie : makelist(x+2*i,i,1,6);
```

(%o15) $[x + 2, x + 4, x + 6, x + 8, x + 10, x + 12]$

A teraz linie w kolejnych grubościach i kolorach (opcja `[legend,false]` opisana jest niżej.

```
(%i16) wxplot2d( linie, [x,-5,5], [legend, false],grubosci_i_kolory);
```

(%t16)




(%o16)

Budujemy 13 "serii danych" (na użytek przykładu każda seria jest jednopunktowa):

```
(%i17) punkty_13_serii : makelist([discrete,[[i,4]]],i,1,13);
```

$$(\%o17)\ [[discrete,[[1,4]]],[discrete,[[2,4]]],[discrete,[[3,4]]],$$
$$[discrete,[[4,4]]],[discrete,[[5,4]]],[discrete,[[6,4]]],$$
$$[discrete,[[7,4]]],[discrete,[[8,4]]],[discrete,[[9,4]]],$$
$$[discrete,[[10,4]]],[discrete,[[11,4]]],[discrete,[[12,4]]],$$
$$[discrete,[[13,4]]]]$$

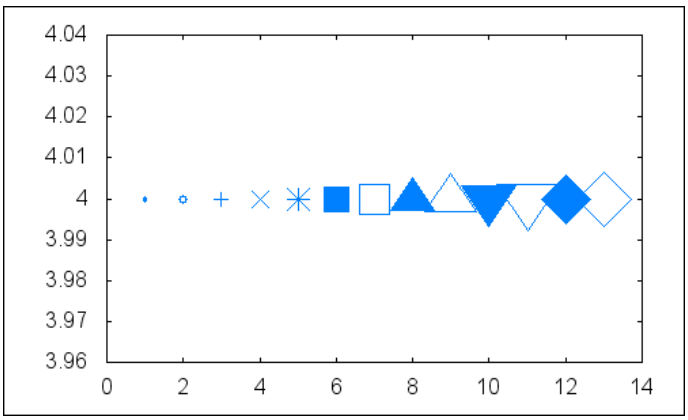
Kombinacje promieni i kształtów (kolor stały):

```
(%i18) promienie_i_typy: cons(style,
       makelist([points,i,1,i],i,1,13));
```

$$(\%o18)\ [style, [points, 1, 1, 1], [points, 2, 1, 2], [points, 3, 1, 3], [points, 4, 1, 4], [points, 5, 1, 5], [points, 6, 1, 6], [points, 7, 1, 7], [points, 8, 1, 8], [points, 9, 1, 9], [points, 10, 1, 10], [points, 11, 1, 11], [points, 12, 1, 12], [points, 13, 1, 13]]$$

Rysujemy:

```
(%i19) wxplot2d(punkty_13_serii,
       promienie_i_typy,[legend,false],
       [gnuplot_preamble, "set size ratio -1"]);
```




(%t19)

(%o19)

*Uwaga*. Alternatywnie parametry związane z kolorem i kształtem można zadawać za pomocą osobnych (niżej opisanych) opcji `color` i `point_type`.

**Kolor**

```
[color, <color_1>, ..., <color_n>]
```

określa jakimi kolorami są rysowane kolejne wykresy. Jeżeli wykresów jest więcej niż n, kolory są powtarzane w pętli.

Dopuszczalne kolory: `blue, red, green, magenta, black, cyan.`

Wartość standardowa: `[color,blue, red, green, magenta, black, cyan]`

**Typ punktu**

`[point_type, <typ_1>,...,<typ_n>]`

wskazuje jak mają być rysowane izolowane punkty w kolejnych wykresach.

Dopuszczalne wartości to:

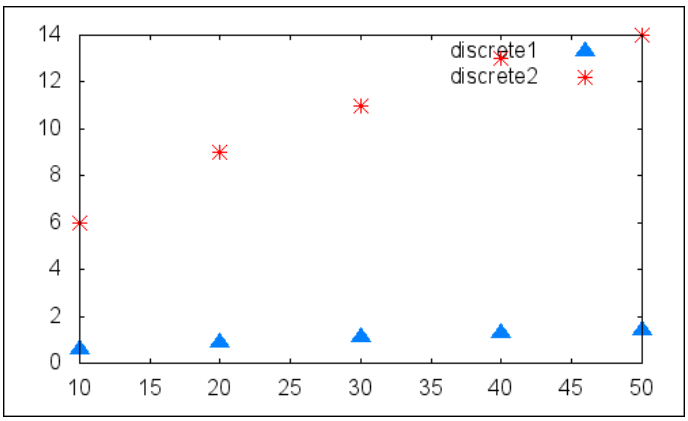
```
bullet,circle,plus,times,asterisk,
box,square,triangle,delta,wedge,nabla,
diamond,lozenge.
```

Standardowa wartość tej opcji (kolejność używania znaków), to:

```
[point_type,bullet,circle,plus,times,
asterisk,box,square,triangle,delta,wedge,
nabla,diamond,lozenge]
```

```
(%i20) wxplot2d([
                [ discrete, [ [10, .6], [20, .9],
                [30, 1.1], [40, 1.3], [50, 1.4]] ],
                [ discrete, [ [10, 6], [20, 9],
                [30, 11], [40, 13], [50, 14]] ]
                  ],
                [point_type,triangle,asterisk],
                [style, points])$
```

(%t20)



### 9.2.3 Opcje modyfikujące opis wykresów

#### Osie

`[axes, <symbol>]`

W zależności od wartości `<symbol>` mamy:

- `true` - obie osie są rysowane (wartość standardowa),
- `x` - tylko oś pozioma,
- `y` - tylko oś pionowa,
- `false` - osie nie są rysowane.

#### Legenda

`[legend, <tekst_1>, ..., <tekst_n>]`

Opis wykresów na rysunku. Standardowo wypisywane są wyrażenia lub nazwy funkcji zadające wykresy. Zmieniamy to wpisując własne wersje dla `<tekst_1>, ..., <tekst_n>`

- `[legend, false]` powoduje brak wyświetlania opisu rysunku.
- `[xlabel, "text"]`, `[ylabel, "text"]` opisy osi poziomej i pionowej.

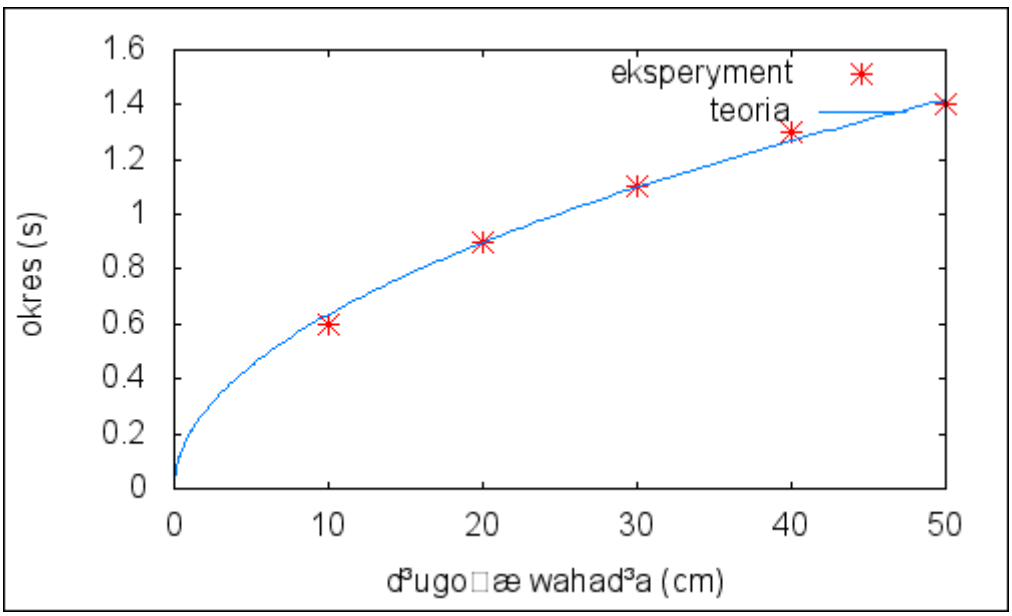
Standardowo nazwa zmiennej niezależnej oraz y dla zmiennej zależnej.

Dla przykładu narysujemy wykres danych eksperymentalnych wraz z funkcją teoretyczną związaną z danymi:

```
(%i21) xy: [[10, 0.6], [20, 0.9], [30, 1.1], [40, 1.3],
       [50, 1.4]]  /*dane eksperymentalne */$
```

```
(%i22)     wxplot2d(
                    [ [discrete, xy],
                     2*%pi*sqrt(l/980)
                    ],
                     /* zadanie wykresów*/
                    [l,0,50],
                    /* zakres */
                    [style, points, lines],
                    /* styl kolejnych wykresów (punkty, linia)*/
                   [color, red, blue],
                   /* kolor kolejnych wykresów */
                    [point_type, asterisk],
                   /* kształt punktu */
                    [legend, "eksperyment", "teoria"],
                    /* legenda */
                    [xlabel, "długość wahadła (cm)"],
                    /* opis osi  x*/
                    [ylabel, "okres (s)"])
                    /* opis osi  y*/
          $
```

(%t22)




Aby uniknąć kłopotów z polskimi literami można wykonać ten przykład w wersji bez wx.

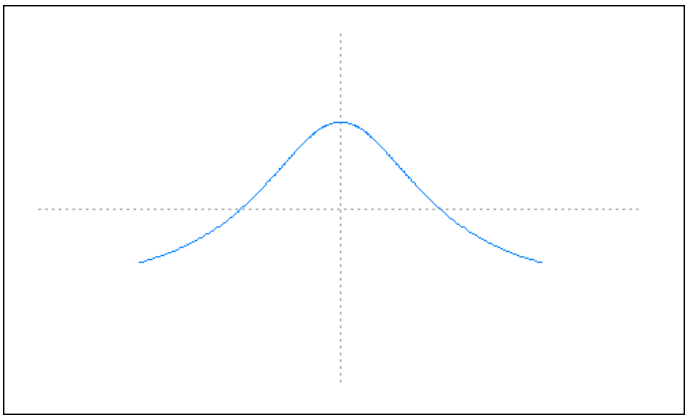
**Ramka**

```
[box, <symbol>]
```

Jeżeli `<symbol>` jest równy `true` rysowany jest prostokąt ograniczający wykres, jeżeli `false` prostokąt nie jest rysowany.

```
(%i24) wxplot2d(1/(1+v^2)-1/2 , [v,-2,2], [x,-3,3],[y,-1,1], [box,false])$
```

(%t24)



## 9.3 Grafika 3D

W tej części omówimy polecenia związane z pozycją menu "`Kreślenie->Wykres 3D...`". Poniżej używamy wersji poleceń z przedrostkiem `wx`, ale warto w ramach własnych ćwiczeń powtórzyć te komendy w wersji bez `wx` (w okienku `gnuplot`), ponieważ mamy wtedy, manewrując myszką, możliwość dynamicznej zmiany punktu widzenia.

### 9.3.1 Składnia polecenia wxplot3d (lub plot3d)

**Powierzchnie zadane jawnym wzorem**

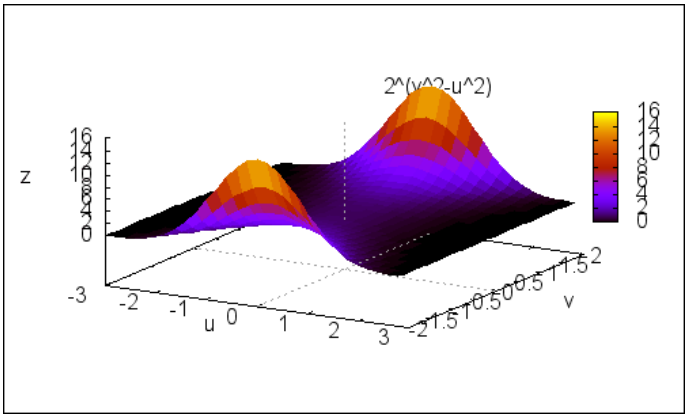
Funkcja

```
wxplot3d ([<expr_1>, ..., <expr_n>],  <x_zakres>,
<y_zakres>,  ..., <opcje>, ...)
```

wyświetla jedną lub więcej powierzchni będących wykresami funkcji dwóch zmiennych. Funkcje mogą być zadane wzorami (wyrażeniami) `<expr_1>, ..., <expr_n>` lub nazwami. Zakres podajemy tak jak w poleceniu `plot2d`.

```
(%i25) wxplot3d (2^(-u^2 + v^2), [u, -3, 3], [v, -2, 2])$
```

(%t25)




**Powierzchnie zadane parametrycznie**

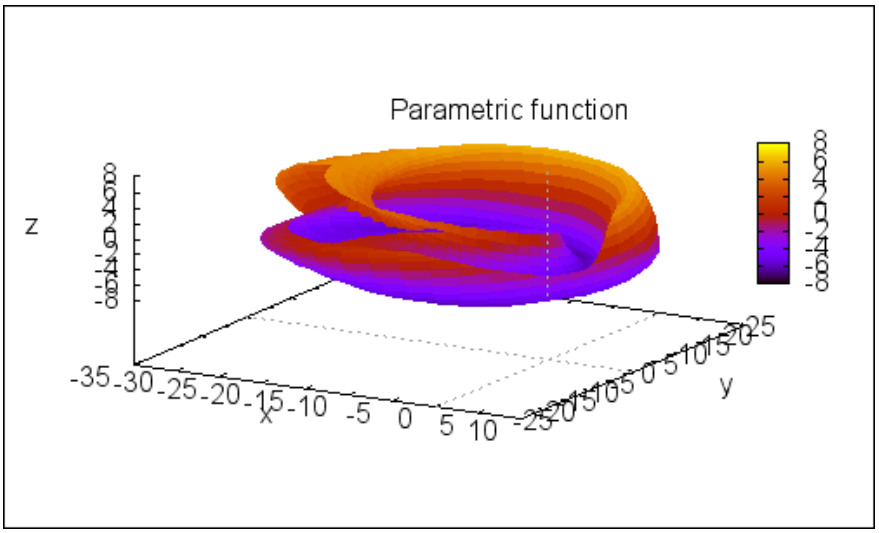
Funkcja

```
wxplot3d([x(u,v),y(u,v),z(u,v)], u_zakres, v_zakres, opt)
```

rysuje powierzchnię zadaną równaniami x(u,v),y(u,v),z(u,v), gdzie parametry zmieniają się we wskazanych zakresach.

Typowym przykładem jest butelka Kleina (opcja `grid` odgrywa podobną rolę jak `nticks` w plot2d, zob. niżej)

```
(%i26) expr_1:5*cos(x)*(cos(x/2)*cos(y)+
              sin(x/2)*sin(2*y)+3.0)-10.0$
       expr_2:-5*sin(x)*(cos(x/2)*cos(y) +
               sin(x/2)*sin(2*y) + 3.0)$
       expr_3: 5*(-sin(x/2)*cos(y) + cos(x/2)*sin(2*y))$
       wxplot3d ([expr_1, expr_2, expr_3],
                 [x, -%pi, %pi],
                 [y, -%pi, %pi],
                 [grid, 40, 40]
                )$
```




(%t29)

Ponieważ rysunek w wewnętrznym formacie nie jest zbyt czytelny, proponujemy wykonać ten rysunek w okienku `gnuplot`

### Poziomice

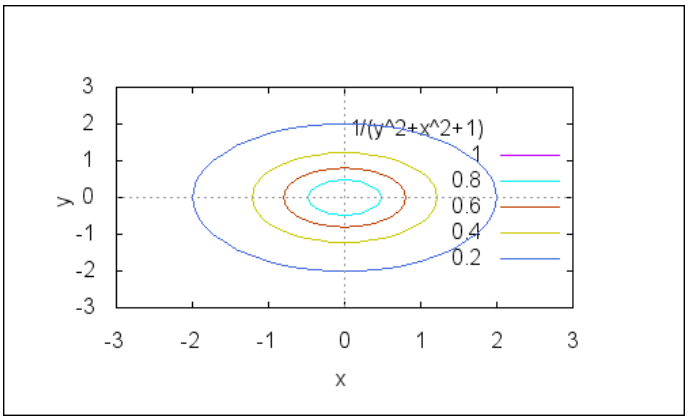
Funkcja

`contour_plot(f(x,y), <x_zakres>,  <y_zakres>,opcje)`

rysuje poziomice fukcji f(x,y) w podanym zakresie. Przykład:

```
(%i31) wxcontour_plot(1/(1+x^2+y^2), [x,-3,3], [y,-3,3])$
```

(%t31)




### 9.3.2 Opcje

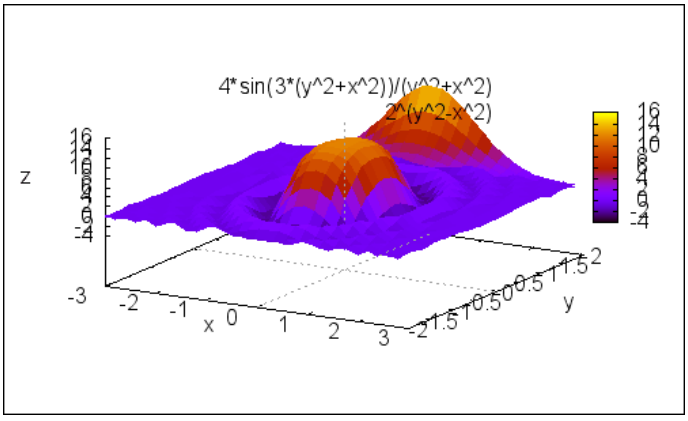
Ograniczamy się do podania jednej opcji, ponieważ bardziej zaawansowaną grafikę będziemy tworzyć przy pomocy pakietu `draw`.

```
[grid,nx,ny]
```

Liczba linii siatki, odpowiednio w kierunku "osi x" i "osi y". Im wyższe są te liczby, tym bardziej gładki jest wykres. Standardowo zadane jest `[grid,30,30]`. Porównajmy poniższe rysunki:

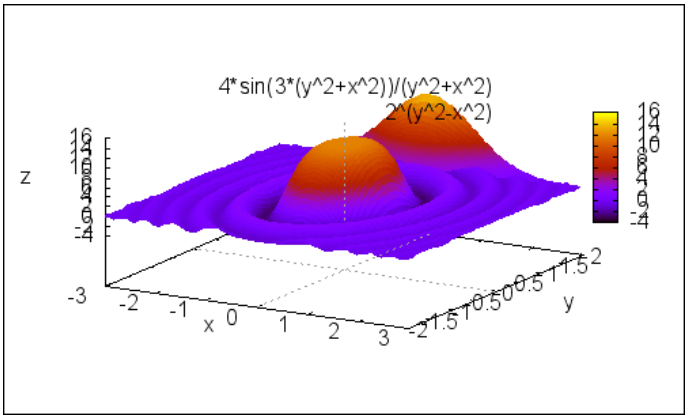
```
(%i32)  wxplot3d ([2^(-x^2 + y^2),
         4*sin(3*(x^2+y^2))/(x^2+y^2),
                   [x, -3, 3], [y, -2, 2]])$
```

(%t32)



```
(%i33)  wxplot3d ([2^(-x^2 + y^2),
         4*sin(3*(x^2+y^2))/(x^2+y^2),
                   [x, -3, 3], [y, -2, 2]],
                   [grid, 100,100]
                   )$
```

(%t33)



# Rozdział 10

# Pakiet `draw`

W przeciwieństwie do poprzednich procedur (`plot2d`, `plot3d`), które były dostępne bezpośrednio po otwarciu `Maximy`, ten zbiór komend stanowi osobną bibliotekę. Niniejsze notatki są roboczym wyciągiem z dokumentacji autora pakietu Mario Rodríguez Riotorto. Ich zawartość wykracza poza planowany zakres opracowania, a w aktualnej chwili są w trakcie testowania i opracowywania. Mimo to, ze względu na użyteczność pakietu zdecydowano się je dołączyć. Rozwinięcie tego tematu planowane jest w następnej części materiałów poświęconych wizualizacji obiektów matematycznych.

Jeżeli chcemy użyć dodatkowej biblioteki (a jest ich dużo, w większości bardzo ciekawych) trzeba ją załadować do sytemu. Służy do tego polecenie `load`, którego argumentem jest nazwa biblioteki (na razie zakładamy, że pakiety znajdują się w standardowym miejscu na dysku, więc nie trzeba podawać ścieżki dostępu).

```
-->    load(draw);

(%o1)

C:/PROGRA~2/MAXIMA~2.0/share/maxima/5.25.0/share/draw/draw.lisp
```

Jeżeli wszystko dzieje się bez kłopotu pojawia się ścieżka dostępu do głównego pliku.

Ogólna zasada pracy pakietu jest następująca:

- używając podstawowych obiektów graficznych oraz opcji budujemy opis dwu- lub trójwymiarowego rysunku. Taki opis jest argumentem funkcji `gr2d` (w przypadku obiektów dwuwymiarowych) lub `gr3d` (w przypadku obiektów trójwymiarowych). Wynik działania tych funkcji, którym jest przekształcenie opisu do wewnętrznego formatu, nazywamy odpowiednio dwu- lub trójwymiarową sceną.
- Sceny wyświetlamy na ekranie (lub zachowujemy w wybranym formacie na dysku) przy pomocy polecenia `draw`:

  ```
  draw(gr2d(opis_1), ..., gr3d(opis_n).
  ```

  Każda scena jest przedstawiana w osobnej tablicy prostokątnej.

Dla skrócenia zapisu przy grafikach złożonych z jednej sceny stworzono funkcje:

- `draw2d(opis)`, która jest równoważna poleceniu `draw(gr2d(opis))`,
- `draw3d(opis)`, która jest równoważna poleceniu `draw(gr3d(opis))`.

Ponieważ opisy często będą bardzo skomplikowane warto jeszcze raz przypomnieć ogólną zasadę poprawiania czytelności kodu przez stosowne formatowanie oraz wprowadzanie nazw pomocniczych (warto naśladować przykłady).

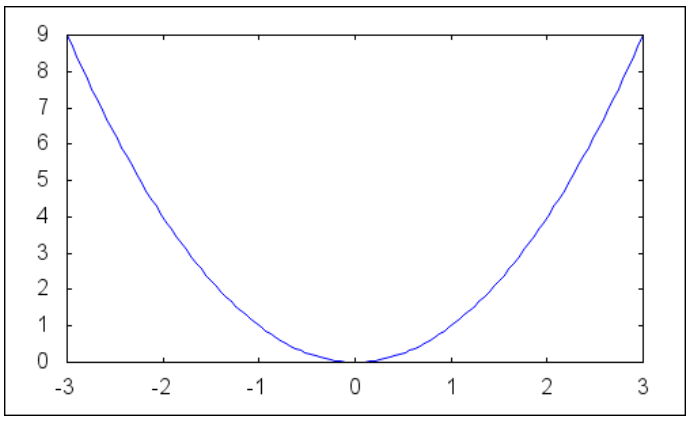
Podobnie jak w przypadku grafik opartych na `plot` możemy alternatywnie używać komend z przedrostkiem `wx` ( rysunek zostaje wklejony do notatnika),lub bez przedrostka (otwiera się osobne okienko `gnuplot`).

## 10.1 Dwuwymiarowe obiekty graficzne

### 10.1.1 Wykres funkcji zadanej jawnie

`explicit(f(x),x,x1,x2)` wykres funkcji f(x) zmiennej x w zakresie od x1 do x2 (oczywiście nazwa argumentu może być inna)

```
--> wxdraw2d(explicit(x^2,x,-3,3) )$
```

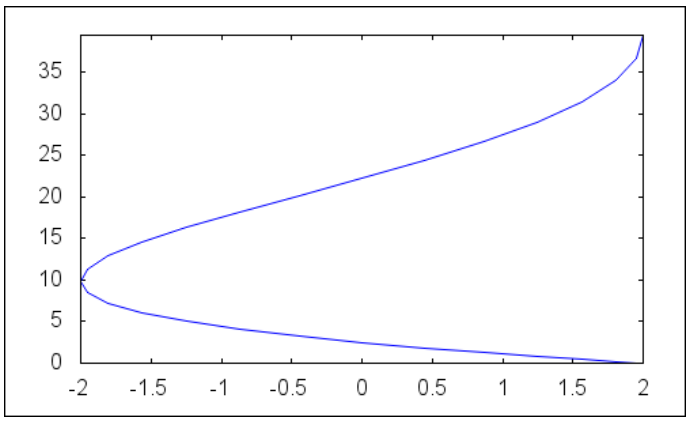



(%t2)

### 10.1.2 Krzywa zadana parametrycznie

`parametric(x(t),y(t),t,t1,t2)` rysunek krzywej zadanej parametrycznie równaniami x(t), y(t) z parametrem t zmieniającym się od t1 do t2

```
--> wxdraw2d( parametric(2*cos(u),u^2,u,0,2*%pi))$
```

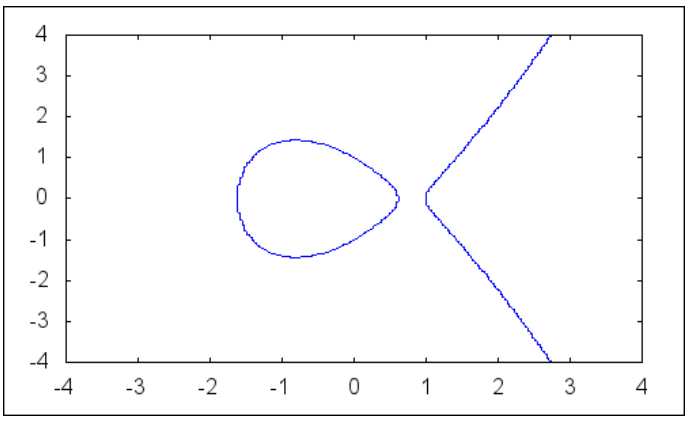



(%t3)

### 10.1.3 Krzywa zadana w sposób uwikłany

`implicit(równanie,x,x1,x2,y,y1,y2)` rysunek krzywej zadanej w sposób uwikłany równaniem `równanie` ze zmiennymi `x` i `y` zmieniającymi się w zakresach `x1...x2`, `y1...y2`.

```
--> wxdraw2d(implicit(y^2=x^3-2*x+1, x, -4,4, y, -4,4));
```




(%t4)

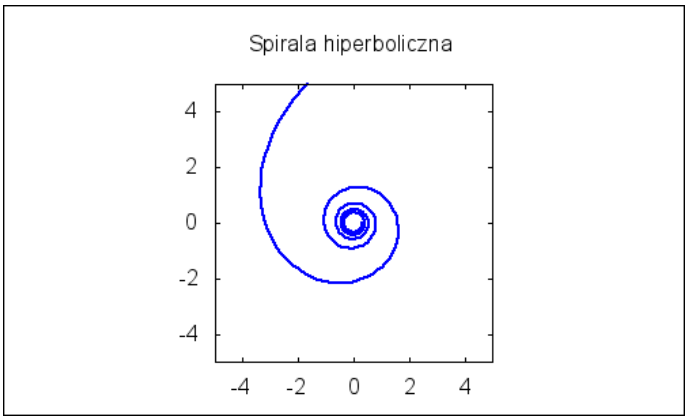
(%o4)

### 10.1.4 Krzywa we współrzędnych biegunowych

`polar(r(fi),fi,fi_1,fi_2)` rysunek krzywej zadanej we współrzędnych biegunowych wzorem `r(fi)` gdzie kąt `fi` zmienia się od `fi_1` do `fi_2`.

```
--> wxdraw2d(user_preamble="set size ratio -1",
             nticks         = 300,
             xrange         = [-5,5],
             yrange         = [-5,5],
             color          = blue,
             line_width     = 2,
             title          = "Spirala hiperboliczna",
             polar(10/theta,theta,1,10*%pi) )$
```

(%t5)

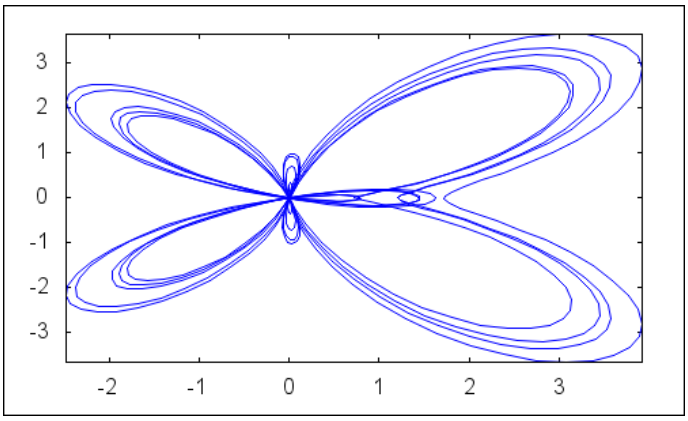



Znaczenie opcji wyjaśnimy za chwilę.

Inny przykład zwany motylem.

```
--> wxdraw2d(nticks   = 600,
      polar(exp(cos(t))-2*cos(4*t)+sin(t/12)^5,
            t, 0, 30)  )$
```

(%t6)

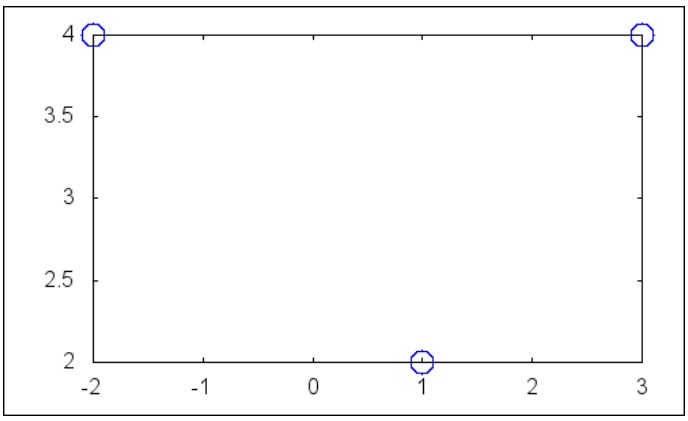



### 10.1.5 Punkty

`points(wsp_x,wsp_y)` punkty, których współrzędne x-owe dane są w liście `wsp_x`, a odpowiednie współrzędne y-owe w liście `wsp_y`

`points(p1,p2,...)` alternatywny sposób zadania punktów - p1, p2, ... są listami dwuelementowymi zadającymi współrzędne kolejnych punktów

```
--> wxdraw2d( point_size    = 3,
                point_type    = circle,
               points([[1,2], [3,4], [-2,4]]));
```



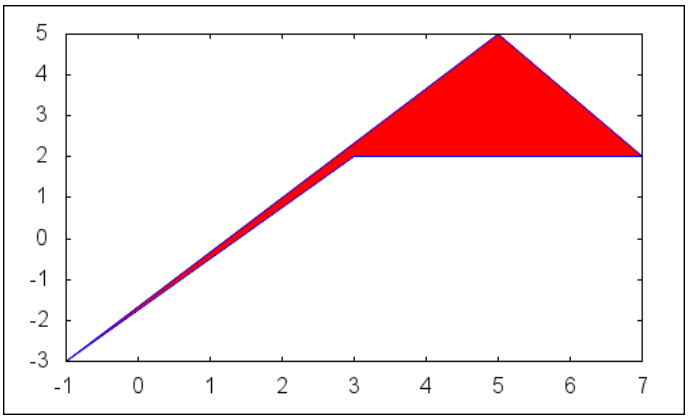

(%t7)

(%o7)

### 10.1.6 Wielokąty

`polygon(wsp_x,wsp_y)` lub `polygon(p1,p2,...)` wielokąt, gdzie współrzędne wierzchołków zadawane są analogicznie jak dla punktów.

```
--> wxdraw2d(polygon([[3,2],[7,2],[5,5], [-1,-3]]))$
```



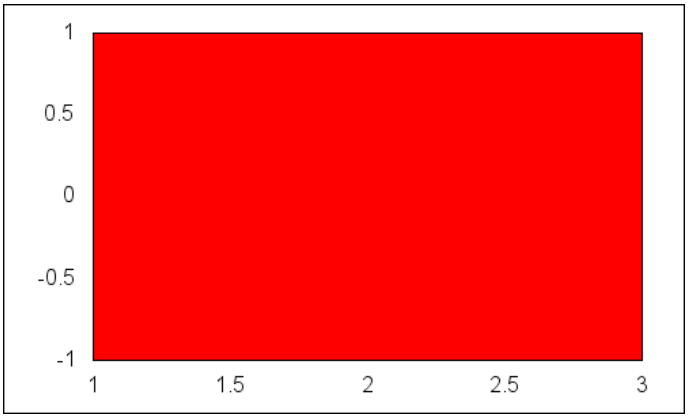

(%t8)

### 10.1.7 Prostokąt

`rectangle(p1,p2)` prostokąt o przeciwległych wierzchołkach p1, p2 (zadanych jako pary współrzędnych - `[p1_x,p1_y], [p2_x,p2_y]`)

```
--> wxdraw2d(rectangle([1,1],[3,-1]))$
```

(%t9)

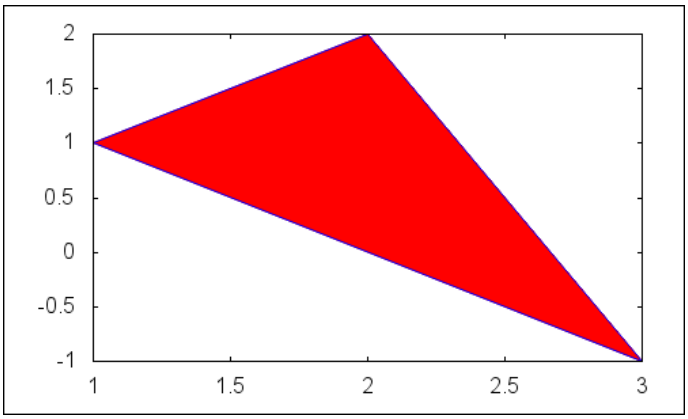



### 10.1.8 Trójkąt

`triangle(p1,p2,p3)` trójkąt o wierzchołkach p1, p2, p3

```
--> wxdraw2d(triangle([1,1],[2,2],[3,-1]))$
```

(%t10)

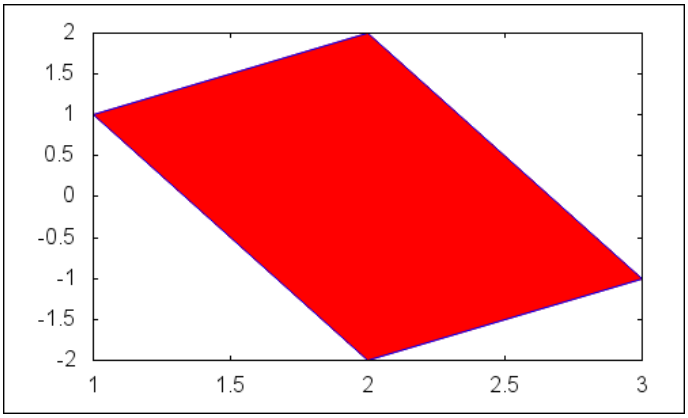


### 10.1.9 Czworokąt

`quadrilateral(p1,p2,p3,p4)` czworokąt o wierzchołkach p1, p2, p3, p4

```
--> wxdraw2d(quadrilateral([1,1],[2,2],[3,-1],[2,-2]))$
```

(%t11)

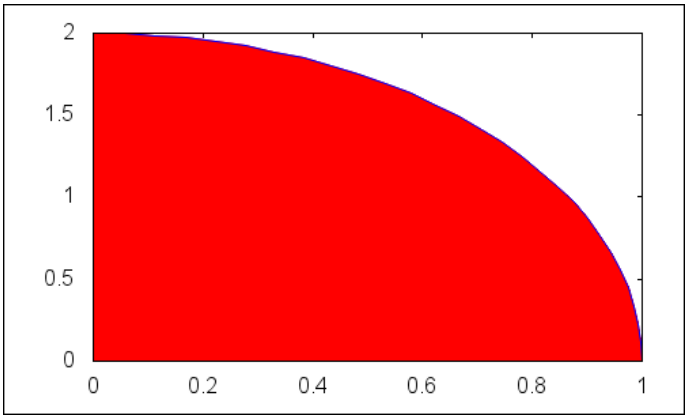



### 10.1.10 Elipsa

`ellipse(x0,y0,a,b,w1,w2)` rysuje część elipsy o środku x0,y0, półosiach a,b, kącie początkowym w1 i końcowym w2 (w stopniach)

```
--> wxdraw2d(ellipse(0,0,1,2,0,90));
```

(%t12)




(%o12)

### 10.1.11 Etykieta i jej opcje

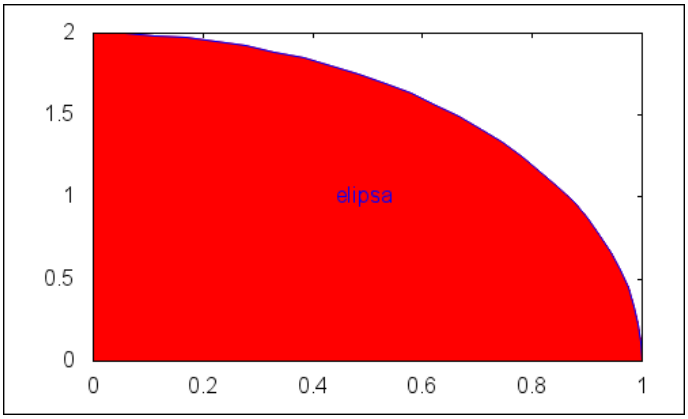
`label([\verbtext|,x,y],...)|` napisy umieszczone w kolejnych pozycjach (x,y).

Atrybuty etykiety ustawiamy przy pomocy opcji:

`label_alignment=value` wyrównanie etykiety, możliwa wartość: `center` (default), `left`, `right`

`label_orientation=value` orientacja etykiety, możliwa wartość: `horizontal` (default), `vertical`

```
--> wxdraw2d(ellipse(0,0,1,2,0,90),
             label(["elipsa", 0.5,1]));
```




(%t13)

(%o13)

### 10.1.12 Wektor i jego opcje

`vector([x,y],[dx,dy])` wektor o początku w `[x,y]` i współrzędnych `[dx,dy]`

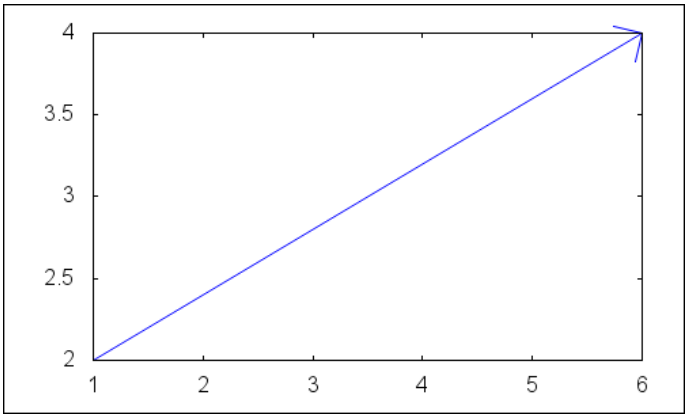
`head_length=len` długość strzałki w jednostkach osi x (default: 2)

`head_angle=kąt` kąt strzałki (default: 45)

`head_type=typ` typ strzałki, wartości: `filled` (default), `empty`, `nofilled`

`head_both=true/false` czy rysować dwie strzałki

```
--> wxdraw2d(head_length = 0.3,
               head_type   =  nofilled,
               vector([1,2],[5,2]));
```




(%t14)

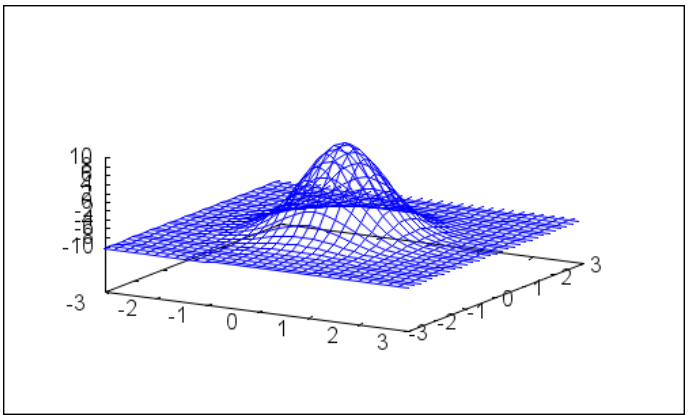
(%o14)

## 10.2 Trójwymiarowe obiekty graficzne

### 10.2.1 Funkcje jawne

`explicit(f(x,y),x,x1,x2,y,y1,y2)` wykres funkcji `f(x,y)` w zakresie `x1...x2` i `y1...y2`

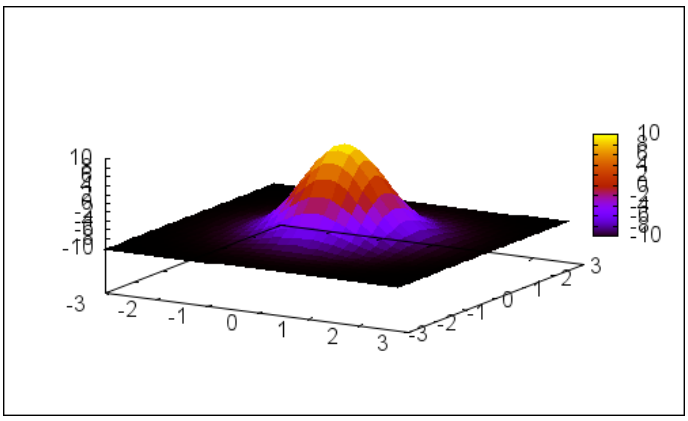
```
--> wxdraw3d(explicit(20*exp(-x^2-y^2)-10,x,-3,3,y,-3,3));
```




(%t15)

(%o15)

```
-->    wxdraw3d(surface_hide = true,
              enhanced3d   = true,
              explicit(20*exp(-x^2-y^2)-10,x,-3,3,y,-3,3));
```

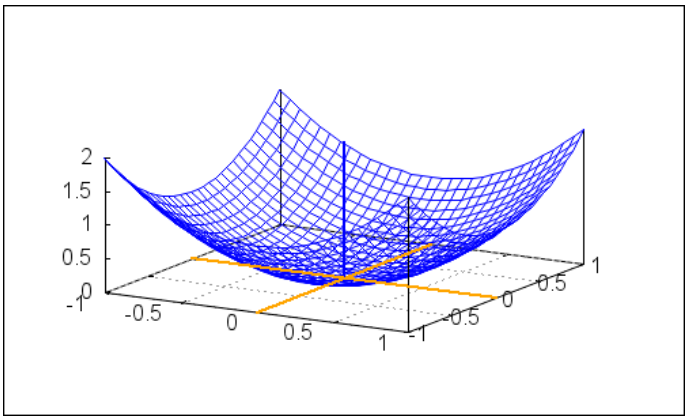



(%t16)

(%o16)

```
-->    wxdraw3d(explicit(x^2+y^2,x,-1,1,y,-1,1),
         xaxis_width = 2,
         xaxis_color = orange,
         xaxis_type  = solid,
         xaxis=true,
         yaxis_width = 2,
         yaxis_color = orange,
         yaxis_type  = solid,
         yaxis=true,
         zaxis_width = 2,
         zaxis_color = blue,
         zaxis_type  = solid,
         zaxis=true,
         grid=true,
         user_preamble= "set xyplane at 0" )$
```

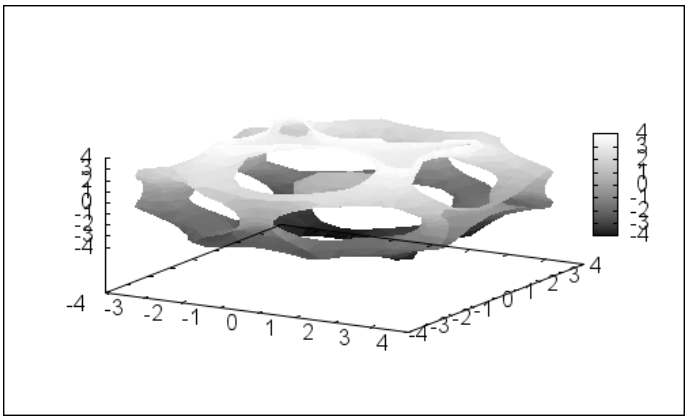
(%t17)




### 10.2.2 Funkcje uwikłane

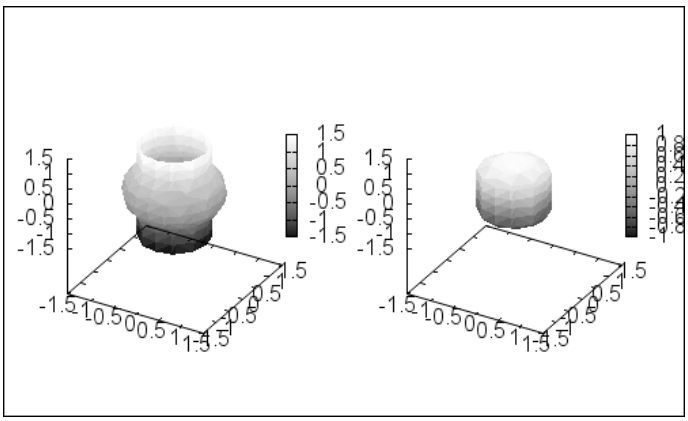
`implicit(equation,x,x1,x2,y,y1,y2,z,z1,z2)` krzywa uwikłana zadana równaniem `equation`, względem zmiennych x, y, z w zakresie `x1...x2`, `y1...y2`, `z1...z2`

```
-->    wxdraw3d(user_preamble="set size ratio 1",
              enhanced3d   = true,
              x_voxel      = 15,
              y_voxel      = 15,
       z_voxel      = 15,
              implicit(2=(cos(x+%phi*y)+cos(x-%phi*y)+
              cos(y+%phi*z)+cos(y-%phi*z)+
              cos(z-%phi*x)+cos(z+%phi*x)),
              x,-4,4,y,-4,4,z,-4,4),
              palette      = gray,
              surface_hide = true)$
```



(%t18)

```
--> wxdraw(
      columns  = 2,
       /* suma */
      gr3d(enhanced3d   = true,
           implicit(min(x^2+y^2+z^2,2*x^2+2*y^2)=1,
                    x,-1.5,1.5,y,-1.5,1.5,z,-1.5,1.5),
           surface_hide = true,
           palette      = gray),
      /* przecięcie */
      gr3d(enhanced3d   = true,
           implicit(max(x^2+y^2+z^2,2*x^2+2*y^2)=1,
                    x,-1.5,1.5,y,-1.5,1.5,z,-1.5,1.5),
           surface_hide = true,
           palette      = gray) )$
```



(%t19)

### 10.2.3 Krzywe zadane parametrycznie

`parametric(x(t),y(t),z(t),t,t1,t2)` krzywa zadana parametrycznie równaniami `x(t),y(t),z(t)`, gdzie parametr t zmienia się od t1 do t2.

`parametric_surface(x(u,v),y(u,v),z(u,v),u,u1,u2,v,v1,v2)`

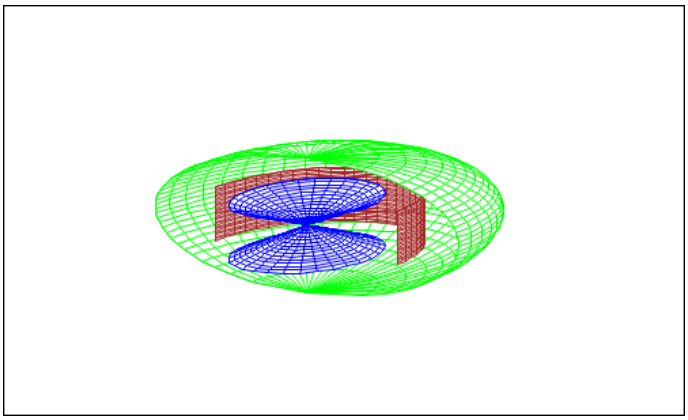
powierzchnia zadana parametrycznie równaniami `x(u,v),y(u,v),z(u,v)`, gdzie parametry u i v zmieniają się w zakresach `u1...u2` i `v1...v2`

### 10.2.4 Współrzędne walcowe

`cylindrical(r(z,fi),z,z1,z2,fi,fi1,fi2)` powierzchnia zadana we współrzędnych walcowych równaniem `r(z,fi)`, gdzie współrzędna z zmienia się w zakresie od z1 do z2, a kąt fi od fi1 do fi2.

```
--> wxdraw3d(
      surface_hide = true,
      axis_3d      = false,
      xtics        = none,
      ytics        = none,
      ztics        = none,
      color        = blue,
      cylindrical(z,z,-2,2,a,0,2*%pi), /*stożek*/
      color        = brown,
      cylindrical(3,z,-2,2,az,0,%pi),  /*cylinder*/
      color        = green,
      cylindrical(sqrt(25-z^2),z,-5,5,a,0,%pi) /*sfera*/ )$
```

(%t20)



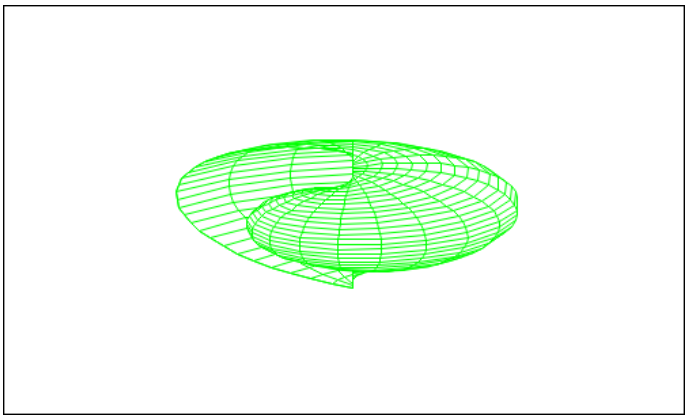
### 10.2.5 Współrzędne sferyczne

`spherical(r(fi,psi),fi,fi1,fi2,psi,psi1,psi2)` powierzchnia zadana we współrzędnych sferycznych równaniem r(fi, psi) , gdzie kąt fi zmienia się w zakresie od fi1 do fi2, a kąt psi od psi1 do psi2.

Przykład zwany muszlą.

```
--> wxdraw3d(
      color        = green,
      surface_hide = true,
      axis_3d      = false,
      xtics        = none,
      ytics        = none,
      ztics        = none,
      spherical(a+z,a,0,3*%pi,z,0,%pi))$
```

(%t21)



### 10.2.6 Punkty

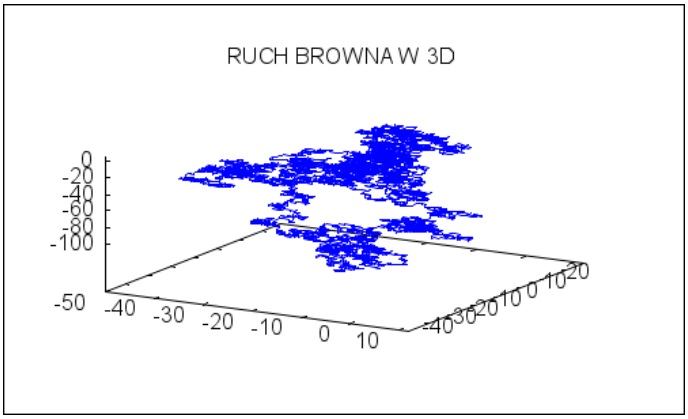
`points(xvals,yvals,zvals)`

`points(p1,p2,...)`

punkty zadawane analogicznie jak w 2D

Przykład ilustrujący ruch Browna w 3D.

```
-->    block([history:[[0,0,0]], lst, pos],
          for k:1 thru 10000 do
             (lst: copylist(last(history)),
              pos: random(3)+1,
              lst[pos]: lst[pos] + random(2)*2-1,
              history: endcons(lst, history) ),
          wxdraw3d(title         = "RUCH BROWNA W 3D",
          point_size    = 0,
          points_joined = true,
          points(history) )  )$
```




(%t22)

### 10.2.7 Opisy

`label([text,x,y,z],...)` etykieta `text` umieszczona w pozycji `[x,y,z]`; wyrównanie, fonty, etc. mogą być ustawione innymi opcjami.

### 10.2.8 Wektory

`vector([x,y,z],[dx,dy,dz])` wektor o początku `[x,y,z]` i współrzędnych `[dx,dy,dz]`

## 10.3 Opcje

Wszystkie opcje mają postać

```
nazwa = wartość
```

Część opcji ma charakter globalny, co w praktyce oznacza, że mogą być umieszczone w dowolnym miejscu opisu.

Opcje wpływające na formę obiektów graficznych mają charakter lokalny, muszą być podane przed obiektem i obowiązują do kolejnej zmiany ich wartości.

Jeżeli nie podajemy jawnie wartości opcji to używana jest jej wartość standardowa (default).

## Użyteczne przykłady ogólnych opcji

- `set_draw_defaults(opts,. . . )`
  w ten sposób możemy modyfikować wartości standardowe wybranych opcji na czas całej sesji,
- `file_name=plik`
  nazwa pliku, do którego zapisywana jest grafika (jeżeli zostanie wybrana opcja zapisu), standardowa nazwa, to `maxima_out`,
- `user_preamble=tekst`
  dodatkowe polecenia przekazywane do gnuplot (ważny przykład: `user_preamble="set size ratio -1"`,
  co ustawia właściwe proporcje długości i szerokości)
- `dimensions=[szer,wys]`
  wymiar grafiki (w pixelach dla formatu bitmapowego, w 1/10mm dla grafiki wektorowej)
- `columns=n`
  liczba kolumn, jeżeli grafika składa się z wielu scen
- `color=nazwa_koloru`
  kolor linii
- `background_color= nazwa_koloru` kolor tła
- `fill_color=nazwa_koloru`
  kolor wypełnienia

- `xrange=[min,max]`
  zakres dla osi x
- `yrange=[min,max]`
  zakres dla osi y
- `zrange=[min,max]`
  zakres dla osi z
- `logx=true/false`
  skala logarytmiczna dla osi x (jeżeli true)
- `logy=true/false`
  skala logarytmiczna dla osi y (jezeli true)
- `logz=true/false`
  skala logarytmiczna dla osi z (jezeli true)
- `grid=true/false`
  rysuje siatkę (jeżeli true)
- `xtics=true/false`
  rysuje znaczniki na osi x (jezeli true)
- `ytics=true/false`
  rysuje znaczniki na osi y (jezeli true)
- `ztics=true/false`
  rysuje znaczniki na osi z (jezeli true)
- `xtics_rotate=true/false`
  obraca znaczniki osi x o 90 stopni
- `ytics_rotate=true/false`
  obraca znaczniki osi y o 90 stopni
- `ztics_rotate=true/false`
  obraca znaczniki osi z o 90 stopni
- `title=text`
  główny tytuł dla sceny (standardowo pusty)
- `key=text`
  nazwa funkcji wpisywana w legendzie (standardowo pusty)

- `xlabel="text"`
  Etykieta dla oxi x
- `ylabel="text"`
  Etykieta dla oxi y
- `zlabel="text"`
  Etykieta dla oxi z
- `xaxis=true/false`
  decyduje czy rysować oś x
- `yaxis=true/false`
  decyduje czy rysować oś y
- `zaxis=true/false`
  decyduje czy rysować oś z
- `x(yz)axis_width=width`
  grubość linii odpowiedniej osi
- `x(yz)axis_color=color`
  kolor odpowiedniej osi
- `x(yz)axis_type=solid/dots`
  typ linii odpowiedniej osi (standard dots)
- `line_width=width`
  grubość linii
- `line_type=solid/dots`
  typ linii (standard solid)
- `point_size=size`
  wymiar punktu w rysunkach "punktowych"
- `point_type=n`
  typ punktu, możliwe wartości: -1,0,1,2,. . . 13.
- `points_joined=true/false`
  czy łączyć punkty odcinkami (standard `false`)
- `nticks=n`
  parametr odpowiedzialny za gładkość krzywej (standard 30)

## 10.4 Opcje dla 2D

- `axis_bottom=true/false`
  czy pokazywać dolną oś
- `axis_top=true/false`
  czy pokazywać górną oś
- `axis_left=true/false`
  czy pokazywać lewą oś
- `axis_right=true/false`
  czy pokazywać prawą oś
- `filled_func=true/false`
  czy wypełniać kolorem obszar pomiędzy rysowaną funkcją i dolną osią
- `filled_func=f`
  czy wypełniać kolorem obszar pomiędzy rysowaną funkcją i funkcją f
- `transparent=true/false`
  wielokąty są kolorowane zgodnie z opcją fill_color (jeżeli true)
- `border=true/false`
  czy rysować brzegi wielokątów
- `transform`
  jeżeli wartość jest postaci
  `[f1(x,y), f2(x,y), x, y]`
  (dla grafiki 2D), to stosowana jest transformacja płaszczyzny według podanych wzorów (podobnie dla grafiki 3D), standardowa wartość 'none'.

## 10.5 Opcje dla 3D

- `view=[a,b]`
  kąty widzenia: a wokół osi x, b wokół osi z

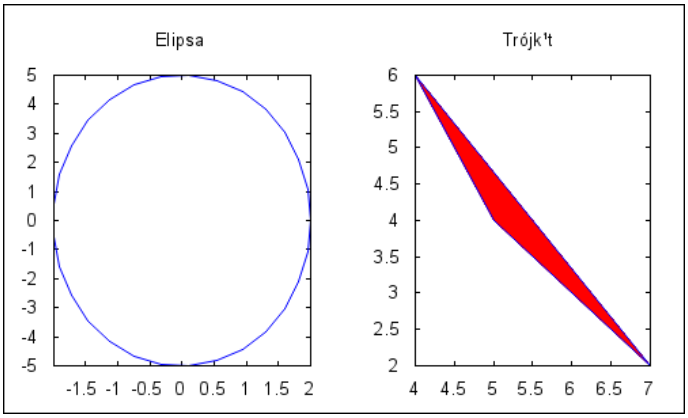
- `axis_3d=true/false`
  czy wyświetlać wszystkie osie w 3D (default: true)
- `xu_grid=n`
  wpływa na gęstość próbkowania wzdłuż pierwszej osi (standard =30),
- `yv_grid=n`
  wpływa na gęstość próbkowania wzdłuż drugiej osi (standard =30), ma zastosowanie do obiektów: 'explicit' i 'parametric_surface'.
- `surface_hide=true/false`
  czy wyświetlać ukryte części powierzchni
- `enhanced3d=true/false`
  czy kolorować powierzchnie
- `palette=[r,g,b]`
  określenie palety kolorów
- `colorbox=true/false`
  czy rysować skalę wyrażoną w kolorach
- `contour=value`
  poziomice dla powierzchni możliwe wartości: `none` (default), `base`, `surface`, `both`, `map`
- `contour_levels=n`
  n poziomic jest rysowanych w równych odstępach
- `contour_levels=[x1,dx,x2]`
  poziomice są rysowane od x1 do x2 z krokiem dx
- `contour_levels=\{x1,x2,. . . \}`
  poziomice są rysowane na wskazanych poziomach x1, x2, etc.

## 10.6 Przykłady grafiki 2d złożonej z kilku scen

```
-->    scena1: gr2d(title  = "Elipsa",
                    nticks = 30,
                  parametric(2*cos(t),5*sin(t),t,0,2*%pi))$
```

```
--> scena2: gr2d(font = "Arial",
                 title  = "Trójkąt",
                 color      = blue,
                 fill_color = red,
                 polygon([4,5,7],[6,4,2]))$

--> wxdraw( columns = 2,  scena1, scena2)$
```

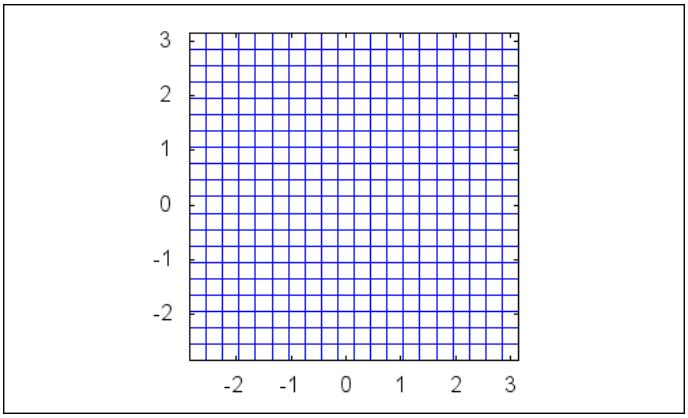
(%t25)




## 10.7 Transformacje w 2D

```
--> /* poprawiamy opcje dla zaoszczędzenia czasu */
    set_draw_defaults(
      points_joined = true,
      point_type = dot,
      color = blue,
      proportional_axes = 'xy) $
```

```
-->    /* budujemy siatkę */
       malla(x0,x1,y0,y1) :=
         block([dx, dy, n:21],
           dx: (x1-x0)/n,
           dy: (y1-y0)/n,
           append(
              makelist(
                 points(makelist([x0+j*dx, y0+k*dy],
                                 k, 1, n)),
                 j, 1, n),
              makelist(
                 points(makelist([x0+j*dx, y0+k*dy],
                                 j, 1, n)),
                 k, 1, n)) )$

-->    /* funkcja pomocnicza */
       module(x,y) := sqrt(x^2+y^2) $

-->    /* siatka przed transformacją */
       wxdraw2d(malla(-%pi,%pi,-%pi,%pi)) $
```

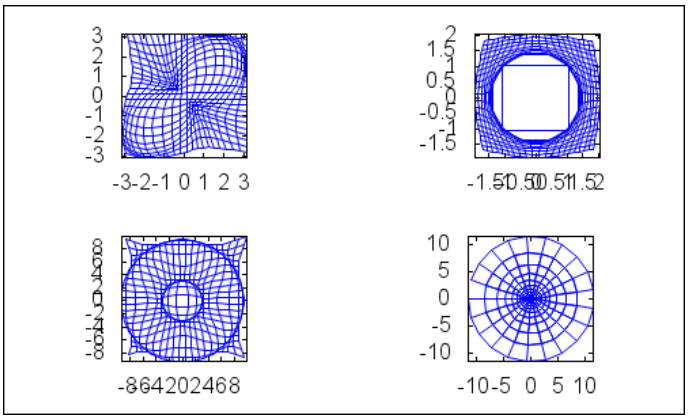



(%t29)

```
-->    tr1 :
         block([k : sin(x)*sin(y)/module(x,y)],
           gr2d(transform = [k*x+x, k*y+y, x, y],
                malla(-%pi,%pi,-%pi,%pi) )) $
```

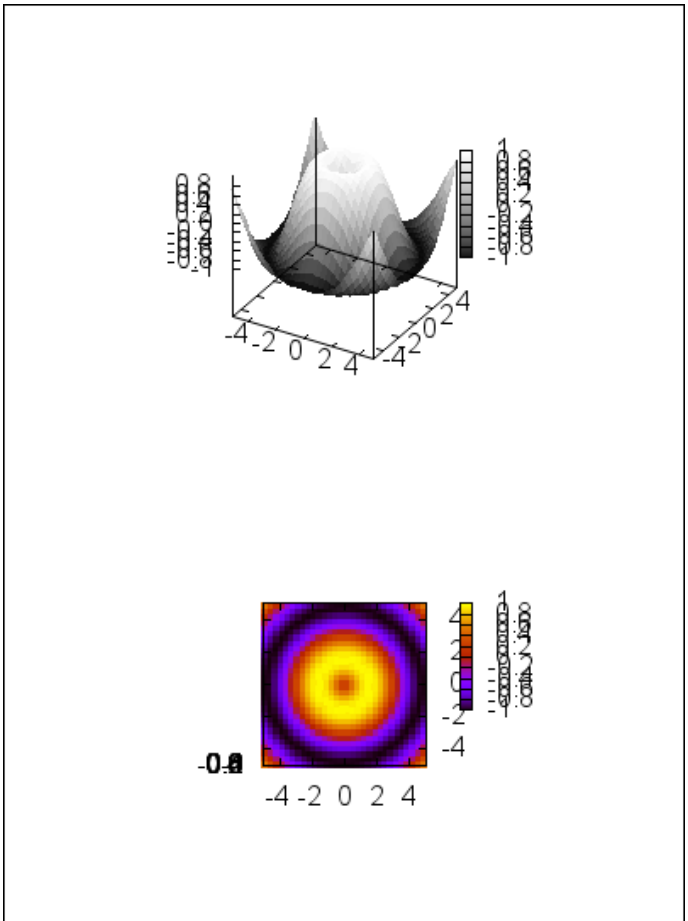
```
-->    tr2 :

         block([k : 1.25/(module(x,y)+0.8)/module(x,y) ],

           gr2d(transform = [k*x+x, k*y+y, x, y],

                malla(-3/2,3/2,-3/2,3/2) )) $

-->    tr3 :
         block([k : sin(module(x,y))/module(x,y) ],
           gr2d(transform = [k*x+x, k*y+y, x, y],
                malla(-3*%pi,3*%pi,-3*%pi,3*%pi) )) $

-->    tr4 :
         gr2d(transform = [realpart(cos(x+y*%i)),
                           imagpart(cos(x+y*%i)),
                           x, y],
              malla(-%pi, %pi, -%pi, %pi) ) $

-->    wxdraw(
          columns  = 2,
           tr1, tr2, tr3, tr4 ) $
```

(%t34)



## 10.8 Przykład grafiki 3d złożonej z kilku scen

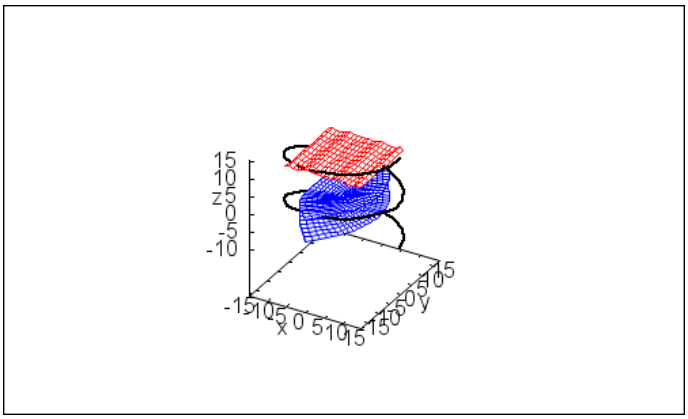
```
--> /* scena w szarościach */
    scena1: gr3d(surface_hide = true,
                 enhanced3d   = true,
                 palette      = gray,
                explicit(sin(sqrt(x^2+y^2)),x,-5,5,y,-5,5))$
    /*kolor jako 3 wymiar */
    scena2: gr3d(surface_hide   = true,
                 enhanced3d     = true,
                 view = [0,360],
                 explicit(sin(sqrt(x^2+y^2)),x,-5,5,y,-5,5))$
    /* szarości jako 3 wymiar */
    scena3: gr3d(surface_hide  = true,
       enhanced3d    = true,
                 palette       = gray,
                 user_preamble = "set pm3d map",
                explicit(sin(sqrt(x^2+y^2)),x,-5,5,y,-5,5))$
    /* A teraz all together */
    wxdraw( dimensions = [400,800],
    scena1,
    scena2,
    scena3);
```



(%o38)

### 10.8.1 Mieszanka wedlowska

```
-->    wxdraw3d(surface_hide = true,
              xlabel       = "x",
              ylabel       = "y",
              zlabel       = "z",
              color        = "blue",
              parametric_surface(u**2-v**2,2*u*v,u, u,-3,3,v,-3,3),
              color   = red,
              xu_grid = 20,
              yv_grid = 10,
              explicit(10+exp(0.3*sin(x^2/10)+0.2*cos(x^3/5)),
              x,-10,10,y,-10,10),
              color      = black,
              line_width = 2,
              nticks     = 40,
             parametric(15*cos(r),10*sin(r),2*r-10,r,0,4*%pi) );
```




(%t39)

(%o39)

## 10.9 Animacja w wxMaxima

Słabo udokumentowane, na razie poznajemy metodą "learning by watching".

Jeżeli klikniemy na rysunek utworzony przy pomocy procedury `with_slider` możemy uruchomić animację przy pomocy standardowych przycisków znajdujących się na pasku ikon (strzałka — start, kwadrat — stop). Jeżeli mamy myszkę z kółkiem, to przy jego pomocy możemy wyświetlać animację klatka po klatce.

Proszę przeanalizować i uruchomić poniższe przykłady w `wxMaximie`.

### Przykład 1

```
tay(n, x) := block(
   [ts : taylor(sin(x__), x__, 0, n)],
   subst(x__=x, ts)
)$
with_slider(
/*pierwszy argument to parametr i jego wartości*/
   n, 2*makelist(i, i, 1, 10)-1,
/*pozostałe jak dla komend typu plot*/
   [sin(x), '(tay(n, x))],
   [x, -9, 9],
   [y, -2, 2]
)$
```

### Przykład 2

```
with_slider_draw(
/*pierwszy argument to parametr i jego wartości*/
   a, makelist(i, i, 0, 30)/5,
/*pozostałe jak dla komend typu draw */
   fill_color = green,
   color = black,
       polygon([[0,0], [6,0], [6,8]]),
   fill_color = red,
       polygon([[a, 0], [6,0], [6, 4*a/3], [a,4*a/3]]),
   color = black,
   line_width = 3,
   key = "Area of inscribed rectangle",
       explicit(4*x/3*(6-x), x, 0, a),
```

```
   yrange = [0, 14],
   xrange = [-2, 8]
)$
```

**Przykład 3**

```
with_slider_draw(
 /*pierwszy argument to parametr i jego wartości*/
   a, makelist(i, i, 0, 20)*%pi/10,
/*pozostałe jak dla komend typu draw */
       parametric(-1+cos(t), sin(t), t, 0, 2*%pi),
   line_width = 2,
   color = blue,
       explicit(sin(x), x, 0, a),
   color = green,
   points_joined = true,
   point_size = 0,
   line_width = 1,
       points([[-1,0], [-1+cos(a), sin(a)]]),
   color = black,
       explicit(sin(a), x, -1+cos(a), a),
dimensions= [600,400],
      yrange = [-2, 2],
   xrange = [-3, 2*%pi+1],
   xaxis = true,
   yaxis = true
)$
```

**Przykład 4**

```
with_slider_draw(
   ang, makelist(i,i,0,20)*%pi/10,
   dimensions= [600,400],
   xrange     = [-10,10],
   yrange     = [-11,10],
   nticks     = 80,
   color      = red,
```

```
      parametric(6*cos(u),
                 6*sin(u),
                 u,0,2*%pi),
      parametric((6+2)*sin(ang)+2*cos(u),
                 (6+2)*cos(ang)+2*sin(u),
                 u,0,2*%pi),
   color = blue,
   line_width = 2,
      parametric((6+2)*sin(a)-2*sin(a*(1+6/2)),
                 (6+2)*cos(a)-2*cos(a*(1+6/2)),
                 a,0,ang),
   color      = black,
   point_size = 1,
   points_joined = true,
   line_width = 1,
proportional_axes = xy,
      points([
              [0,0],
              (6+2)*[sin(ang),cos(ang)],
              [(6+2)*sin(ang)-2*sin(ang*(1+6/2)),
               (6+2)*cos(ang)-2*cos(ang*(1+6/2))]
             ]))$
```

**Przykład 5**

```
kill(all)$  z(x, y):=y^2 - x^2;
wxdraw3d(view=[55,120], xaxis=true, yaxis=true,
enhanced3d=true, contour = both,
ztics={-9, 0, 9}, xlabel="x", ylabel="y",zlabel="z",
explicit(z(x, y),x, -3, 3, y, -3, 3)
)$
```

**Przykład 6**

```
with_slider_draw3d(t, makelist(0.2*i,i,-10,10),
xaxis=true, yaxis=true,
enhanced3d=true, contour = both,
```

```
ztics={-9, 0, 9}, xlabel="x", ylabel="y",zlabel="z",
contour_levels = {t-1, t+1},
explicit(z(x, y),x, -3, 3, y, -3, 3)  )$
```

# Rozdział 11

# Równania liniowe i nieliniowe

## 11.1 Symboliczne rozwiązanie równania z jedną niewiadomą

### Funkcja `solve`

Funkcja `solve(<expr>)`

MENU: "Równania->Rozwiąż..."

Rozwiązuje równanie algebraiczne <expr> względem niewiadomej <x>. Wynikiem jest lista rozwiązań, której elementami są równania dla <x>.

Jeżeli <expr> nie jest równaniem, rozwiązywane jest równanie <expr>=0.

Niewiadoma <x> może być funkcją (np. f(x)) lub innym wyrażeniem nieatomowym za wyjątkiem sum i iloczynów.

<x> może być pominięte, jeżeli <expr> zawiera tylko jedną zmienną.

<expr> może zawierać funkcje wymierne, trygonometryczne, wykładnicze, itp.

```
-->    solve (x^3 - 1);
```

(%o10) $[x = \frac{\sqrt{3}\,i - 1}{2}, x = -\frac{\sqrt{3}\,i + 1}{2}, x = 1]$

```
-->     solve(a*x^2+b*x+c=0,x);
```

(%o24) $[x = -\frac{\sqrt{b^2 - 4\,a\,c} + b}{2\,a}, x = \frac{\sqrt{b^2 - 4\,a\,c} - b}{2\,a}]$

```
-->     solve(5^f(x) = 125,f(x));
```

(%o25) $[\mathrm{f}\,(x) = \frac{\log\,(125)}{\log\,(5)}]$

```
-->     %, radcan;
```

(%o26) $[\mathrm{f}\,(x) = 3]$

```
-->     solve(x^6+2*x^4+x^3-4*x-4.2,x);
```

$rat : replaced - 4.2 by - 21/5 = -4.2$

(%o21) $[0 = 5\,x^6 + 10\,x^4 + 5\,x^3 - 20\,x - 21]$

## 11.2 Symboliczne rozwiązywanie układu równań

### Funkcja `linsolve`

Funkcja `linsolve([<eqn_1>, ..., <eqn_n>], [<x_1>, ..., <x_n>])`

Rozwiązuje układ równań liniowych zadanych w postaci pierwszej listy względem niewiadomych wskazanych w drugiej liście.

MENU: Równania->Rozwiąż układ równań liniowych

```
-->     e1: x + z = y;
        e2: 2*a*x - y = 2*a^2;
        e3: y - 2*z = 2;
        linsolve ([e1, e2, e3], [x, y, z]);
```

(%o4) $z + x = y$

(%o5) $2\,a\,x - y = 2\,a^2$

(%o6) $y - 2\,z = 2$

(%o7) $[x = a + 1, y = 2\,a, z = a - 1]$

## Funkcja algsys

Funkcja `algsys([<eqn_1>, ..., <eqn_n>], [<x_1>, ..., <x_n>])`

Rozwiązuje układ równań wielomianowych zadanych w postaci pierwszej listy względem niewiadomych wskazanych w drugiej liście.

Wynikiem jest lista list równań przedstawiających rozwiązania.

Jeżeli `algsys` nie potrafi wyznaczyć rozwiązania wynikiem jest pusta lista [].

Jeżeli układ ma nieskończenie wiele rozwiązań zależnych od dowolnych stałych, to stałe te oznaczane są symbolami `%r1, %r2, ...` .

Jeżeli `algsys` nie potrafi znaleźć rozwiązania dokładnego, to próbuje wyznaczyć rozwiązanie przybliżone.

Rodzaj rozwiązania przybliżonego zależy od wartości zmiennej systemowej `realonly`. Jeżeli `realonly` ma wartość `true`, wyznaczane są przybliżone rozwiązania rzeczywiste (przy pomocy funkcji `realroots`), jeżeli `realonly` ma wartość `false` wyznaczane są przybliżone rozwiązania zespolone (przy pomocy funkcji `allroots`).

```
-->     e1: 2*x*(1 - a1) - 2*(x - 1)*a2=0;
        e2: a2 - a1=0;
        e3: a1*(-y - x^2 + 1)=0;
        e4: a2*(y - (x - 1)^2)=0;
        algsys ([e1, e2, e3, e4], [x, y, a1, a2]);
```

(%o19) $2\,\left(1-a1\right)\,x-2\,a2\,\left(x-1\right)=0$

(%o20) $a2-a1=0$

(%o21) $a1\,\left(-y-x^2+1\right)=0$

(%o22) $a2\,\left(y-\left(x-1\right)^2\right)=0$

(%o23) $[[x=0,y=\%r3,a1=0,a2=0],[x=1,y=0,a1=1,a2=1]]$

```
-->     e1: x^2 - y^2=0;
        e2: -1 - y + 2*y^2 - x + x^2=0;
        algsys ([e1, e2], [x, y]);
```

(%o24) $x^2-y^2=0$

(%o25) $2\,y^2-y+x^2-x-1=0$

(%o26) $[[x=-\frac{1}{\sqrt{3}},y=\frac{1}{\sqrt{3}}],[x=\frac{1}{\sqrt{3}},y=-\frac{1}{\sqrt{3}}],[x=-\frac{1}{3},y=-\frac{1}{3}],[x=1,y=1]]$

### Funkcja `solve` dla układów równań

Funkcja `solve([<eqn_1>, ..., <eqn_n>], [<x_1>, ..., <x_n>])`

Rozwiązuje układ równań liniowych lub nieliniowych (wielomianowych) wykorzystując funkcję `linsolve` lub `algsys`.

```
-->    solve([4*x^2 - y^2 = 12, x*y - x = 2], [x,y]);
```

### Funkcja `eliminate`

Funkcja `eliminate([<eqn_1>, ..., <eqn_n>], [<x_1>, ..., <x_k>])`

Eliminuje zmienne `[<x_1>, ..., <x_k>]` z równań `[<eqn_1>,..., <eqn_n>]` dając w wyniku n-k równań.

```
-->    expr1: 2*x^2 + y*x + z=0;
       expr2: 3*x + 5*y - z - 1=0;
       expr3: z^2 + x - y^2 + 5=0;
       eliminate ([expr3, expr2, expr1], [y, z]);
```

## 11.3 Numeryczne rozwiązanie równania z jedną niewiadomą

### Funkcja `findroot`

Funkcja `find_root(<expr>, <x>, <a>, <b>)`

Funkcja `find_root(<nazwa_funkcji>, <a>, <b>)`

Wyznacza miejsca zerowe wyrażenia `<expr>` lub funkcji o podanej nazwie w domkniętym przedziale `[<a>, <b>]`. Wyrażenie `<expr>` może być równaniem, wtedy wyznaczane jest miejsce zerowe wyrażenia `f(x)` po doprowadzeniu równania do postaci `f(x)=0`.

Funkcja wykorzystuje metodę bisekcji. Za spełnienie założeń gwarantujących znalezienie miejsca zerowego odpowiedzialny jest użytkownik. W przypadku kilku miejsc zerowych metoda znajduje tylko jedno z nich.

W pomocy można znaleźć opis bardziej rozbudowanych wersji tej funkcji umożliwiających (m.in.) sterowanie błędem przybliżenia.

```
-->     f(x) := sin(x) - x/2;
        find_root (sin(x) - x/2, x, 0.1, %pi);
        find_root (sin(x) = x/2, x, 0.1, %pi);
        find_root (f(x), x, 0.1, %pi);
        find_root (f, 0.1, %pi);
```

(%o28) $\mathrm{f}\left(x\right) := \sin\left(x\right) - \frac{x}{2}$

(%o29) 1.895494267033981

(%o30) 1.895494267033981

(%o31) 1.895494267033981

(%o32) 1.895494267033981

Różne sposoby zadania tego samego problemu:

```
-->     find_root (exp(x) = y, x, 0, 100);
```

(%o33) $\mathrm{find\_root}\left(e^{x} = y, x, 0.0, 100.0\right)$

Zmienna y nie ma nadanej wartości, nie można zrealizować metody bisekcji.

```
-->     find_root (exp(x) = y, x, 0, 100), y = 10;
```

(%o36) 2.302585092994046

Nadanie wartości y w lokalnym środowisku obliczeniowym.

```
-->     fpprec:32;
        bf_find_root (exp(x) = y, x, 0, 100), y = 10;
```

(%o37) 32

(%o38) 2.3025850929940456840179914546844*b*0

Wykorzystanie arytmetyki dużej dokładności.

## Funkcja allroots

Funkcja allroots(<eqn>)

Funkcja bfallroots(<eqn>)

Oblicza numeryczne przybliżenia pierwiastków rzeczywistych i zespolonych wielomianu <eqn> (drugi wariant używa arytmetyki dużej dokładności).

```
--> eqn: (1 + 2*x)^3 = 13.5*(1 + x^5);
```

(%o2) $$(2\,x+1)^3 = 13.5\,\left(x^5+1\right)$$

```
--> allroots (eqn);
```

(%o3) $$[x = 0.829674990212936, x = -1.015755543828121,$$
$$x = 0.965962515219637\,i - 0.406959723192407,$$
$$x = -0.965962515219637\,i - 0.406959723192407, x = 1.0]$$

## Funkcja realroots

Funkcja realroots (<eqn>)

Oblicza wymierne przybliżenia pierwiastków rzeczywistych wielomianu <eqn> (zapisanego jako wyrażenie lub równanie).

Współczynniki wielomianu muszą być liczbami zmiennoprzecinkowymi lub dokładnymi wymiernymi (nie może wystąpić np. %pi).

Liczby zmiennoprzecinkowe są konwertowane do wymiernych i wynik zawsze jest podawany w postaci liczby wymiernej.

Algorytm opiera się na konstrukcji ciągu Sturma i metodzie bisekcji.

```
--> realroots (-1 - x + x^5);
```

(%o9) $$[x = \frac{39168221}{33554432}]$$

Mamy jeden pierwiastek, ponieważ pozostałe są rzeczywiste:

```
-->    allroots(-1 - x + x^5);
```

(%o7) $[x = 1.08395410131771\,i + 0.181232444469875, x = 0.181232444469875 - 1.08395410131771\,i, x = 0.352471546031726\,i - 0.764884433600585, x = -0.352471546031726\,i - 0.764884433600585, x = 1.167303978261419]$

```
-->    ev (realroots (-1 - x + x^5), float);
```

(%o10) $[x = 1.167303949594498]$

Wykorzystanie funkcji ev do konwersji na liczbę zmienno przecinkową.

## 11.4 Użyteczne funkcje pomocnicze

Funkcja: `lhs(expr)`

Zwraca lewą stronę wyrażenia `expr`, jeżeli operator główny wyrażenia jest jednym z operatorów relacyjnych

```
< <= =  #  equal
notequal  >=  >
```

operatorem przypisania lub operatorem definicji funkcji := .

Funkcja: `rhs(expr)`

Jak wyżej dla prawej strony.

## 11.5 Przypisywanie wartości rozwiązań

Przeanalizujmy jeszcze raz znany przykład

```
-->    kill(all);
```

(%o0) *done*

```
-->    rozw : solve (x^3 - 1);
```

(%o1) $[x = \frac{\sqrt{3}\,i - 1}{2}, x = -\frac{\sqrt{3}\,i + 1}{2}, x = 1]$

```
-->     x;
```

(%o2) $x$

Polecenia rozwiązujące równania (układy równań) przedstawiają wynik w postaci listy równań opisujących niewiadome, natomiast nie przypisują wartości do zmiennych. Jeżeli chcemy użyć ich w dalszych obliczeniach musimy to zrobić samodzielnie. W tym celu listę wyników nazwaliśmy `rozw` i teraz możemy korzystać z odwoływania się do elementów list.

```
-->     x1 : rhs(rozw[1]); x2 : rhs(rozw[2]); x3 : rhs(rozw[3]);
```

(%o3) $\frac{\sqrt{3}\,i-1}{2}$

(%o4) $-\frac{\sqrt{3}\,i+1}{2}$

(%o5) $1$

```
-->     x1; x2; x3;
```

(%o6) $\frac{\sqrt{3}\,i-1}{2}$

(%o7) $-\frac{\sqrt{3}\,i+1}{2}$

(%o8) $1$

Przykład: wyprowadzić wzory na sumę i iloczyn pierwiastków równania kwadratowego w zależności od współczynników (wzory Vieta)

```
-->     kill(all);
```

(%o0) $done$

```
-->     rozw : solve(a*x^2+b*x+c=0,x);
```

(%o1) $[x=-\frac{\sqrt{b^2-4\,a\,c}+b}{2\,a},x=\frac{\sqrt{b^2-4\,a\,c}-b}{2\,a}]$

```
-->     x1 : rhs(rozw[1]); x2 : rhs(rozw[2]);
```

(%o2) $-\frac{\sqrt{b^2-4\,a\,c}+b}{2\,a}$

(%o3) $$\frac{\sqrt{b^2-4\,a\,c}-b}{2\,a}$$

```
--> '(x1+x2)=ratsimp(x1+x2);
```

(%o7) $$x2+x1=-\frac{b}{a}$$

```
--> '(x1*x2)=ratsimp(x1*x2);
```

(%o8) $$x1\,x2=\frac{c}{a}$$

Podobne zadanie dla równania stopnia trzeciego

```
--> kill(all);
```

(%o0) *done*

```
--> rozw : solve(a*x^3+b*x^2+c*x+d=0,x)$
```

```
--> x1 : rhs(rozw[1])$ x2 : rhs(rozw[2])$ x3 : rhs(rozw[3])$
```

```
--> '(x1+x2+x3)=ratsimp(x1+x2+x3);
```

(%o14) $$x3+x2+x1=-\frac{b}{a}$$

```
--> '(x1*x2*x3)=fullratsimp(x1*x2*x3);
```

(%o16) $$x1\,x2\,x3=-\frac{d}{a}$$

Tu ratsimp okazało się zbyt słabe ...

```
--> '(x1*x2+x1*x3+x2*x3)=ratsimp(x1*x2+x1*x3+x2*x3);
```

(%o15) $$x2\,x3+x1\,x3+x1\,x2=\frac{c}{a}$$

Komentarz: do rozwiązania wykorzystaliśmy "brutalną siłę" obliczeniową programu Maxima. Siła ta ma wiele ograniczeń, zarówno sprzętowych jak i czysto matematycznych. Na przykład, spróbujmy rozwiązać w podobny sposób zadanie dla wielomianu stopnia piątego.

```
--> kill(all);
```

(%o0) *done*

```
-->    rozw : solve(a_5*x^5+a_4*x^4+a_3*x^3+ a_2*x^2+a_1*x + a_0=0,x);
```

(%o2) $[0 = a\_5\,x^5 + a\_4\,x^4 + a\_3\,x^3 + a\_2\,x^2 + a\_1\,x + a\_0]$

Maxima jako wynik podaje nieprzetworzone równanie, co oznacza, że nie potrafi rozwiązać takiego równania. Raczej nic dziwnego dla każdego kto słyszał o Ewaryście Galois. Ale co z naszymi eksperymentami związanymi z wzorami Vieta? Nic straconego, trzeba chwilę pomyśleć i zmienić taktykę.

```
-->    w : expand(a_5*x^5+a_4*x^4+a_3*x^3+ a_2*x^2+a_1*x
        + a_0 = a_5*(x-x1)*(x-x2)*(x-x3)*(x-x4)*(x-x5))$

-->    coeff(lhs(w),x^5)=coeff(rhs(w),x^5);
```

(%o12) $a\_5 = a\_5$

```
-->    coeff(lhs(w),x^4)=coeff(rhs(w),x^4), factor;
```

(%o15) $a\_4 = -a\_5\ (x5 + x4 + x3 + x2 + x1)$

```
-->    coeff(lhs(w),x^3)=coeff(rhs(w),x^3), factor;
```

(%o16) $$a\_3 = a\_5\,(x4\,x5 + x3\,x5 + x2\,x5 + x1\,x5 + x3\,x4 + x2\,x4 + x1\,x4 + x2\,x3 + x1\,x3 + x1\,x2)$$

```
-->    coeff(lhs(w),x^2)=coeff(rhs(w),x^2), factor;
```

(%o17) $a\_2 = -a\_5\,(x3\,x4\,x5 + x2\,x4\,x5 + x1\,x4\,x5 + x2\,x3\,x5 + x1\,x3\,x5 + x1\,x2\,x5 + x2\,x3\,x4 + x1\,x3\,x4 + x1\,x2\,x4 + x1\,x2\,x3)$

```
-->    coeff(lhs(w),x)=coeff(rhs(w),x), factor;
```

(%o18) $$a\_1 = a\_5\,(x2\,x3\,x4\,x5 + x1\,x3\,x4\,x5 + x1\,x2\,x4\,x5 + x1\,x2\,x3\,x5 + x1\,x2\,x3\,x4)$$

```
-->    coeff(lhs(w),x,0)=coeff(rhs(w),x,0), factor;
```

(%o19) $a\_0 = -a\_5\,x1\,x2\,x3\,x4\,x5$

## 11.6 Sprawdzanie wyniku

Pracując z systemem algebry komputerowej powinniśmy zachować ograniczone zaufanie do uzyskanych wyników. Z różnych powodów mogą one nie być całkowicie poprawne z matematycznego punktu widzenia, chociaż komputer nie sygnalizuje żadnego błędu. Dlatego możliwie często powinniśmy je weryfikować. Poniżej podajemy przykład jednej z wielu technik sprawdzania rozwiązań równań.

```
-->    kill(all);
```

(%o0) *done*

```
-->    rozw : solve (x^3 - 1);
```

(%o1) $[x = \frac{\sqrt{3}\,i - 1}{2}, x = -\frac{\sqrt{3}\,i + 1}{2}, x = 1]$

Sprawdzenie pierwszego rozwiązania (przypominamy, że zgodnie z definicją `ev` jest to wyliczenie wyrażenia z chwilowym przypisaniem danym w `rozw[1]` i uproszczeniem wyniku)

```
-->    ev(x^3-1,rozw[1],ratsimp);
```

(%o3) 0

Drugie i trzecie rozwiązanie sprawdzimy przy pomocy uproszczonej formy `ev`:

```
-->    x^3-1,rozw[2],ratsimp; x^3-1,rozw[3],ratsimp;
```

(%o5) 0

(%o6) 0

# Rozdział 12

# Algebra liniowa - podstawowe narzędzia

## 12.1 Definiowanie macierzy

Podstawowym obiektem używanym w obliczeniach związanych z algebrą liniową jest `matrix` (macierz). Obiekt ten zadajemy przy pomocy polecenia :

`matrix(<wiersz_1>,...,<wiersz_n>)`

Argumenty `<wiersz_1>,...,<wiersz_n>` są listami o równej długości zadającymi wiersze macierzy. Elementami tych list mogą być dowolne wyrażenia. Przykładowo:

```
(%i1)  matrix([a,b,c,d], [1,2,3,4]);
```

$$\text{(\%o1)} \quad \begin{pmatrix} a & b & c & d \\ 1 & 2 & 3 & 4 \end{pmatrix}$$

Wygodnym sposobem wprowadzania macierzy jest skorzystanie z menu "Algebra->Wprowadź macierz", gdzie w okienku dialogowym zadajemy kolejno wymiar macierzy, jej rodzaj i nazwę. Następnie wypełniamy tablicę o stosownym kształcie (warto pamiętać, że klawisz Tab przenosi nas do kolejnych pozycji).

*Uwaga.* Wcześniej w menu "Algebra" mamy pozycje "Generuj macierz" i "Generate Matrix from expression" związane z poleceniem "genmatrix".

Omówienie tych pozycji odłożymy na później, ponieważ związane są z bardziej zaawansowanymi elementami Maximy.

Dla wielu macierzy specjalnego typu istnieją komendy przyspieszające ich wprowadzanie. Poniżej podajemy kilka przykładów takich poleceń.

### 12.1.1 Macierz diagonalna (i jednostkowa)

Funkcja `diagmatrix (<n>, <x>)`
daje w wyniku macierz diagonalną wymiaru `n x n` z elementem <x> na przekątnej.

<n> musi wyliczać się do liczby naturalnej, <x> może być dowolnym wyrażeniem. W szczególności `diagmatrix(n, 1)` daje w wyniku n-wymiarową macierz jednostkową.

```
(%i2)  diagmatrix(4,a);
```

(%o2)
$$\begin{pmatrix} a & 0 & 0 & 0 \\ 0 & a & 0 & 0 \\ 0 & 0 & a & 0 \\ 0 & 0 & 0 & a \end{pmatrix}$$

```
(%i3)  diagmatrix(4,1);
```

(%o3)
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

### 12.1.2 Macierz Cauchy'ego

Funkcja `cauchy_matrix ([x_1, x_2,... , x_m])`

daje w wyniku macierz wymiaru m x n, o elementach `a[i,j] = 1/(x_i+x_i)`.
Przykłady:

```
(%i4)  cauchy_matrix([a,b,c,d]);
```

(%o4) $$\begin{pmatrix} \frac{1}{2\,a} & \frac{1}{b+a} & \frac{1}{c+a} & \frac{1}{d+a} \\ \frac{1}{b+a} & \frac{1}{2\,b} & \frac{1}{c+b} & \frac{1}{d+b} \\ \frac{1}{c+a} & \frac{1}{c+b} & \frac{1}{2\,c} & \frac{1}{d+c} \\ \frac{1}{d+a} & \frac{1}{d+b} & \frac{1}{d+c} & \frac{1}{2\,d} \end{pmatrix}$$

```
(%i5)  cauchy_matrix([1,2,3,4,5]);
```

(%o5) $$\begin{pmatrix} \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \\ \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} \end{pmatrix}$$

### 12.1.3 Macierz o jednym nietrywialnym elemencie

Funkcja `ematrix( <m>, <n>, <x>, <i>, <j>)`

daje w wyniku macierz o wymiarach m x n, której wszystkie elementy są równe 0, za wyjątkiem elementu o indeksach (i,j), który równy jest `<x>`.

```
(%i6)  ematrix(3,2,a^2+b^2,2,2);
```

(%o6) $$\begin{pmatrix} 0 & 0 \\ 0 & b^2+a^2 \\ 0 & 0 \end{pmatrix}$$

### 12.1.4 Macierz zerowa

`zeromatrix (m, n)`

macierz o wymiarze m x n , której wszystkie elementy są równe 0.

```
(%i7)  zeromatrix(3,4);
```

(%o7) $$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

## 12.2 Działania na macierzach

Macierz w wewnętrznym języku Maximy jest zaimplementowana jako lista list. Dlatego na macierzach można wykonywać działania arytmetyczne w sposób omówiony przy okazji list, tzn. "element po elemencie". W przypadku dodawania macierzy i mnożenia przez skalar zgadza się to z standardowymi działaniami algebry liniowej:

```
(%i8)  A: matrix(
         [a,b,c],
         [u,v,w],
         [x,y,z]
       );
```

$$\text{(\%o8)}\quad \begin{pmatrix} a & b & c \\ u & v & w \\ x & y & z \end{pmatrix}$$

```
(%i9)  B: matrix(
         [1,2,3],
         [4,5,6],
         [7,8,9]
       );
```

$$\text{(\%o9)}\quad \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

```
(%i10) A+B;
```

$$\text{(\%o10)}\quad \begin{pmatrix} a+1 & b+2 & c+3 \\ u+4 & v+5 & w+6 \\ x+7 & y+8 & z+9 \end{pmatrix}$$

```
(%i11) k*A;
```

$$\text{(\%o11)}\quad \begin{pmatrix} a\,k & b\,k & c\,k \\ k\,u & k\,v & k\,w \\ k\,x & k\,y & k\,z \end{pmatrix}$$

Jednak mnożenie (w konsekwencji potęgowanie), element odwrotny, funkcje argumentu macierzowego (np. znana z teorii równań różniczkowych exp(A))są zdefiniowane w algebrze w zupełnie inny sposób i nieostrożne

stosowanie zwykłych operatorów arytmetycznych (działających element po elemencie) może prowadzić do wyników całkowicie pozbawionych sensu. Zobaczmy to na przykładach:

```
(%i12) A*B; A^2 ; exp(A); A^(-1);
```

$$\text{(\%o12)}\quad \begin{pmatrix} a & 2\,b & 3\,c \\ 4\,u & 5\,v & 6\,w \\ 7\,x & 8\,y & 9\,z \end{pmatrix}$$

$$\text{(\%o13)}\quad \begin{pmatrix} a^2 & b^2 & c^2 \\ u^2 & v^2 & w^2 \\ x^2 & y^2 & z^2 \end{pmatrix}$$

$$\text{(\%o14)}\quad \begin{pmatrix} e^a & e^b & e^c \\ e^u & e^v & e^w \\ e^x & e^y & e^z \end{pmatrix}$$

$$\text{(\%o15)}\quad \begin{pmatrix} \frac{1}{a} & \frac{1}{b} & \frac{1}{c} \\ \frac{1}{u} & \frac{1}{v} & \frac{1}{w} \\ \frac{1}{x} & \frac{1}{y} & \frac{1}{z} \end{pmatrix}$$

Dlatego dla obiektów typu "matrix" zaimplementowano w Maximie specjalne operatory, które działają zgodnie z regułami algebry liniowej. Mnożenie macierzy wykonujemy przy pomocy operatora "dot", którego wizualną formą jest kropka.

```
(%i16) A.B;
```

$$\text{(\%o16)}\quad \begin{pmatrix} 7\,c+4\,b+a & 8\,c+5\,b+2\,a & 9\,c+6\,b+3\,a \\ 7\,w+4\,v+u & 8\,w+5\,v+2\,u & 9\,w+6\,v+3\,u \\ 7\,z+4\,y+x & 8\,z+5\,y+2\,x & 9\,z+6\,y+3\,x \end{pmatrix}$$

```
(%i17) B.A;
```

$$\text{(\%o17)}\quad \begin{pmatrix} 3\,x+2\,u+a & 3\,y+2\,v+b & 3\,z+2\,w+c \\ 6\,x+5\,u+4\,a & 6\,y+5\,v+4\,b & 6\,z+5\,w+4\,c \\ 9\,x+8\,u+7\,a & 9\,y+8\,v+7\,b & 9\,z+8\,w+7\,c \end{pmatrix}$$

Operatorem potęgowania macierzy jest operator oznaczony symbolem ^^

```
(%i18) A^^2;
```

(%o18) $$\begin{pmatrix} c\,x+b\,u+a^2 & c\,y+b\,v+a\,b & c\,z+b\,w+a\,c \\ w\,x+u\,v+a\,u & w\,y+v^2+b\,u & w\,z+v\,w+c\,u \\ x\,z+u\,y+a\,x & y\,z+v\,y+b\,x & z^2+w\,y+c\,x \end{pmatrix}$$

W szczególności przy pomocy tego operatora możemy liczyć macierz odwrotną (o ile istnieje):

```
(%i19) B^^(-1);
```

$solve : singularmatrix. -- anerror. To debug this try : debugmode(true);$

Program sygnalizuje, że macierz jest osobliwa. Weźmy inny przykład:

```
(%i20) C : matrix([1,2],[3,4]);
```

(%o20) $$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

```
(%i21) D : C^^(-1);
```

(%o21) $$\begin{pmatrix} -2 & 1 \\ \frac{3}{2} & -\frac{1}{2} \end{pmatrix}$$

Sprawdźmy:

```
(%i22) C.D;
```

(%o22) $$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Alternatywnie, do liczenia macierzy odwrotnej możemy użyć polecenia "invert" (z menu: "Algebra->Macierz odwrotna)

```
(%i23) invert(D);
```

(%o23) $$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

## 12.3 Pojęcia związane z macierzami dostępne z menu

### 12.3.1 Wyznacznik

Wyznacznik macierzy możemy policzyć przy pomocy polecenia `determinant(<M>)` gdzie <M> jest zadaną macierzą (menu: "Algebra->Wyznacznik")

```
(%i24) determinant(matrix([a,b],[c,d]));
```

(%o24) $a\,d - b\,c$

### 12.3.2 Macierz transponowana

Transponowanie macierzy wykonuje funkcja
`transpose(<M>)`
(menu Algebra->Transponuj macierz")

```
(%i25) transpose(matrix([a,b],[c,d]));
```

(%o25) $\begin{pmatrix} a & c \\ b & d \end{pmatrix}$

### 12.3.3 Macierz dołączona

Funkcja
`adjoint <M>)`
daje w wyniku transponowaną macierz dopełnień algebraicznych (menu "Algebra -> Macierz dołączona").

```
(%i26) adjoint(matrix([a,b],[c,d]));
```

(%o26) $\begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$

### 12.3.4 Wielomian charakterystyczny

Funkcja
`charpoly(<M>, <x>)`

wylicza wielomian charakterystyczny macierzy <M> ze zmienną <x>. Inaczej, jest to funkcja o kodzie

```
determinant (<M> - diagmatrix(length (<M>), <x>))
```

```
(%i27) charpoly(cauchy_matrix([1,2,3]),u);
```

$$(\%o27)\quad \left(\left(\frac{1}{6}-u\right)\left(\frac{1}{4}-u\right)-\frac{1}{25}\right)\left(\frac{1}{2}-u\right)+\frac{\frac{1}{15}-\frac{\frac{1}{4}-u}{4}}{4}-\frac{\frac{\frac{1}{6}-u}{3}-\frac{1}{20}}{3}$$

Dla uproszczenia wyniku można połączyć ją np. z "expand" (tak to jest zaprojektowane w menu)

```
(%i28) charpoly(cauchy_matrix([1,2,3]),u), expand;
```

$$(\%o28)\quad -u^3+\frac{11\,u^2}{12}-\frac{131\,u}{3600}+\frac{1}{43200}$$

### 12.3.5 Wartości własne

Funkcja

`eigenvalues (<M>)`

lub równoważnie

`eivals(<M>)`,

oblicza wartości własne macierzy <M>.

Zwróćmy uwagę na strukturę wyniku!

Pozycja w menu "Algebra->Wartości własne"

```
(%i29) eigenvalues(matrix(
         [2,4,5],
         [1,-1,1],
         [2,1,-1]
       ));
```

$$(\%o29)\quad [[-\frac{3\sqrt{5}-3}{2},\frac{3\sqrt{5}+3}{2},-3],[1,1,1]]$$

Pierwsza lista podaje wartości własne, druga lista podaje ich krotności.

### 12.3.6 Wektory własne

Funkcja

`eigenvectors (<M>)`

lub równoważnie

`eivects (<M>)`,

oblicza wektory własne macierzy <M>.

Zwróćmy uwagę na strukturę wyniku!

Pozycja w menu "Algebra->Wektory własne"

```
(%i30) uw : eigenvectors(matrix(
         [2,4,5],
         [1,-1,1],
         [2,1,-1]
        ));
```

(%o30) $[[[-\frac{3\sqrt{5}-3}{2},\frac{3\sqrt{5}+3}{2},-3],[1,1,1]],[[1,-\sqrt{5}-2,\frac{\sqrt{5}+3}{2}]],[[1,\sqrt{5}-2,-\frac{\sqrt{5}-3}{2}]],[[1,0,-1]]]$

Wynik jest listą złożoną, która ma dwa elementy. Pierwszy jest listą o tej samej strukturze jak wynik funkcji `eigenvalues`, drugi jest listą wektorów własnych odpowiadających kolejnym wartościom własnym.

Jeżeli w dalszym ciągu chcemy używać konkretnych wektorów (wartości) własnych, trzeba się do nich odwołać w standardowy sposób. Na przykład, wprowadzamy nazwę dla pierwszego wektora własnego:

```
(%i31) w1 : uw[2][1][1];
```

(%o31) $[1,-\sqrt{5}-2,\frac{\sqrt{5}+3}{2}]$

Potrójny indeks jest potrzebny, ponieważ jednej wartości własnej może odpowiadać kilka liniowo niezależnych wektorów własnych.

### 12.3.7 Znormalizowane wektory własne

Funkcja

`uniteigenvectors (<M>)`

działa w ten sam sposób jak `eigenvectors`, dodatkowo wyliczając wersory własne.

```
(%i32) uniteigenvectors(matrix(
         [2,4,5],
         [1,-1,1],
         [2,1,-1]
       ));
```

$$(\%o32)\ [[[-\frac{3\sqrt{5}-3}{2},\frac{3\sqrt{5}+3}{2},-3],[1,1,1]],$$
$$[[[\frac{\sqrt{2}}{\sqrt{11\sqrt{5}+27}},-\frac{\sqrt{2}\sqrt{5}+2^{\frac{3}{2}}}{\sqrt{11\sqrt{5}+27}},\frac{\sqrt{5}+3}{\sqrt{2}\sqrt{11\sqrt{5}+27}}]],$$
$$[[\frac{\sqrt{2}}{\sqrt{27-11\sqrt{5}}},\frac{\sqrt{2}\sqrt{5}-2^{\frac{3}{2}}}{\sqrt{27-11\sqrt{5}}},-\frac{\sqrt{5}-3}{\sqrt{2}\sqrt{27-11\sqrt{5}}}]],[[\frac{1}{\sqrt{2}},0,-\frac{1}{\sqrt{2}}]]]]$$

# Rozdział 13

# Analiza matematyczna

## 13.1 Całkowanie

### 13.1.1 Całka nieoznaczona

Funkcja `integrate(<expr>,<x>)`
liczy symbolicznie (a raczej usiłuje policzyć) całkę nieoznaczoną z wyrażenia `<expr>` względem zmiennej `<x>`. Jeżeli próby nie dadzą rezultatu wyświetlana jest niewyliczona postać całki (lub częściowo wyliczona).

Zacznijmy od prostego przykładu

```
(%i1)  integrate(a*x^n,x);
```

$$Isn+1zeroornonzero?\,n;$$

(%o1)
$$\frac{a\,x^{n+1}}{n+1}$$

Jeżeli `Maxima` ma zbyt mało informacji zadaje dodatkowe pytanie - wpisujemy odpowiedź (tu "nonzero") i wciskamy (standardowo) Shift-Enter

```
(%i2)  integrate(a*x^n,x);
```

$$Isn+1zeroornonzero?\,n;$$

(%o2)
$$\frac{a\,x^{n+1}}{n+1}$$

Dla porównania wynik po wybraniu "zero". Jeżeli chcemy uniknąć pytań (na przykład zakładamy że n+1>0) możemy skorzystać z funkcji `assume` (załóż):

```
(%i3)  assume(n+1>0);
```

(%o3) $[n > -1]$

```
(%i4)  integrate(a*x^n,x);
```

(%o4) $$\frac{a\,x^{n+1}}{n+1}$$

Oczywiście poczynione założenia program wykorzystuje również w innych obliczeniach:

```
(%i5)  integrate((a+b)*x^(n+1),x);
```

(%o5) $$\frac{(b+a)\;x^{n+2}}{n+2}$$

Polecenie `forget` odwołuje poczynione założenia:

```
(%i6)  forget(n+1>0);
```

(%o6) $[n > -1]$

Proszę sprawdzić działanie wykonując ponownie poprzednie przykłady. Możemy jednocześnie robić więcej założeń:

```
(%i7)  assume(n+1>0, m+1>0);
```

(%o7) $[n > -1, m > -1]$

```
(%i8)  integrate(a*x^n+b*x^m,x);
```

(%o8) $$\frac{a\,x^{n+1}}{n+1}+\frac{b\,x^{m+1}}{m+1}$$

```
(%i9)  forget(n+1>0, m+1>0);
```

(%o9) $[n > -1, m > -1]$

Zobaczmy kilka przykładów, gdzie dodatkowo używamy operatora ’ dla czytelności całego rachunku:

```
(%i10) assume(b>0)$

       '(integrate(a*exp(x*b),x) )= integrate(a*exp(x*b),x);
```

$$\text{(\%o11)}\quad \int a\,e^{b\,x}dx = \frac{a\,e^{b\,x}}{b}$$

```
(%i12) assume(c>0)$

       '(integrate(a*b^(x*c),x) )= integrate(a*b^(x*c),x);
```

$$\text{(\%o13)}\quad \int a\,b^{c\,x}dx = \frac{a\,b^{c\,x}}{\log\left(b\right)\,c}$$

```
(%i14) assume(m>0)$
       'integrate(x^m*(a+b*x)^5,x) = integrate(x^m*(a+b*x)^5,x);
```

$$\text{(\%o15)}\quad \int x^m\left(b\,x+a\right)^5dx = \frac{b^5\,x^{m+6}}{m+6}+\frac{5\,a\,b^4\,x^{m+5}}{m+5}+\frac{10\,a^2\,b^3\,x^{m+4}}{m+4}+\frac{10\,a^3\,b^2\,x^{m+3}}{m+3}+\frac{5\,a^4\,b\,x^{m+2}}{m+2}+\frac{a^5\,x^{m+1}}{m+1}$$

```
(%i16) 'integrate(x/(a+b*x)^n,x) = integrate(x/(a+b*x)^n,x);
```

$$\text{(\%o16)}\quad \int \frac{x}{\left(b\,x+a\right)^n}dx = -\frac{\left(b^2\,\left(n-1\right)\,x^2+a\,b\,n\,x+a^2\right)\,e^{-n\log\left(b\,x+a\right)}}{b^2\,\left(n^2-3\,n+2\right)}$$

```
(%i17) 'integrate(1/(x^2-c^2)^5,x) = integrate(1/(x^2-c^2)^5,x);
```

$$\text{(\%o17)}\quad \int \frac{1}{\left(x^2-c^2\right)^5}dx = -\frac{35\log\left(x+c\right)}{256\,c^9}+\frac{35\log\left(x-c\right)}{256\,c^9}+\frac{105\,x^7-385\,c^2\,x^5+511\,c^4\,x^3-279\,c^6\,x}{384\,c^8\,x^8-1536\,c^{10}\,x^6+2304\,c^{12}\,x^4-1536\,c^{14}\,x^2+384\,c^{16}}$$

```
(%i18) 'integrate(1/(a+b*x^2)^4,x) = integrate(1/(a+b*x^2)^4,x);
```

$Is\,a\,positive\,or\,negative?\,p;$

(%o18) $$\int \frac{1}{\left(b\,x^2+a\right)^4}dx = \frac{5\,\mathrm{atan}\left(\frac{\sqrt{b}\,x}{\sqrt{a}}\right)}{16\,a^{\frac{7}{2}}\,\sqrt{b}} + \frac{15\,b^2\,x^5+40\,a\,b\,x^3+33\,a^2\,x}{48\,a^3\,b^3\,x^6+144\,a^4\,b^2\,x^4+144\,a^5\,b\,x^2+48\,a^6}$$

### 13.1.2 Całka oznaczona

Funkcja: `integrate (expr, x, a, b)`

liczy symbolicznie (a raczej usiłuje policzyć) całkę oznaczoną z wyrażenia <expr> względem zmiennej <x> w granicach od a do b.

Jeżeli próby nie dadzą rezultatu wyświetlana jest niewyliczona postać całki (lub częściowo wyliczona).

```
(%i19) integrate(a+x^3,x,0,5);
```

(%o19) $$\frac{20\,a+625}{4}$$

```
(%i20) 'integrate(sqrt(a+b*x)/(x^5),x,0,1) =
        integrate(sqrt(a+b*x)/(x^5),x);
```

$Is\,a\,positive\,or\,negative?\,p;$

(%o20) $$\int_0^1 \frac{\sqrt{b\,x+a}}{x^5}dx = -\frac{5\,b^4\log\left(\frac{2\sqrt{b\,x+a}-2\sqrt{a}}{2\sqrt{b\,x+a}+2\sqrt{a}}\right)}{128\,a^{\frac{7}{2}}} - \frac{15\,b^4\left(b\,x+a\right)^{\frac{7}{2}}-55\,a\,b^4\left(b\,x+a\right)^{\frac{5}{2}}+73\,a^2\,b^4\left(b\,x+a\right)^{\frac{3}{2}}+15\,a^3\,b^4\sqrt{b\,x+a}}{192\,a^3\left(b\,x+a\right)^4-768\,a^4\left(b\,x+a\right)^3+1152\,a^5\left(b\,x+a\right)^2-768\,a^6\left(b\,x+a\right)+192\,a^7}$$

### 13.1.3 Zmiana zmiennych

Funkcja `changevar(exp,f(x,y),y,x)`

dokonuje zmiany zmiennych w całce zgodnie z równaniem `f(x,y)= 0`, gdzie x jest starą zmienną, a y nową zmienną.

Zilustrujmy to na przykładzie

```
(%i21) 'integrate(exp(sqrt(5*x)),x,0,4)+
       'integrate(exp(sqrt(5*x+1)),x,0,5)+
       'integrate(exp(sqrt(z*x)),z,0,4);
```

$$(\%o21)\quad \int_0^4 e^{\sqrt{x\,z}}dz+\int_0^5 e^{\sqrt{5\,x+1}}dx+\int_0^4 e^{\sqrt{5}\,\sqrt{x}}dx$$

```
(%i22) changevar(%,x-y^2/5,y,x);
```

$$(\%o22)\quad \int_0^4 e^{\sqrt{x\,z}}dz-\frac{2\int_{-2\,\sqrt{5}}^0 y\,e^{|y|}dy}{5}-\frac{2\int_{-5}^0 y\,e^{\sqrt{y^2+1}}dy}{5}$$

### 13.1.4 Całki wielokrotne

Całki wielokrotne możemy liczyć iterując odpowiednią liczbę razy komendę `integrate`:

```
(%i23) 'integrate('integrate(1/(x+y+1), y, 0,x),x,0,1) =
        integrate(integrate(1/(x+y+1), y, 0,x),x,0,1);
```

$$Is\,x\,positive,\,negative,\,or\,zero?\;p;$$

$$(\%o23)\quad \int_0^1\int_0^x \frac{1}{y+x+1}dydx=\frac{3\log\left(3\right)-4\log\left(2\right)}{2}$$

## 13.2 Różniczkowanie

### 13.2.1 Pochodne zwyczajne

Funkcja `diff(expr,var)` oblicza pochodną wyrażenia `expr` względem zmiennej `var`.

```
(%i24) 'diff(x*sin(x)^2,x) = diff(x*sin(x)^2,x);
```

$$(\%o24)\quad \frac{d}{d\,x}\left(x\sin\left(x\right)^2\right)=\sin\left(x\right)^2+2\,x\cos\left(x\right)\,\sin\left(x\right)$$

```
(%i25) 'diff((x^2+1)/(x^2-1),x) = diff((x^2+1)/(x^2-1),x);
```

$$(\%o25)\quad \frac{d}{d\,x}\,\frac{x^2+1}{x^2-1}=\frac{2\,x}{x^2-1}-\frac{2\,x\,\left(x^2+1\right)}{\left(x^2-1\right)^2}$$

Funkcja `diff(expr,var,n)`

oblicza pochodną n-tego rzędu wyrażenia `expr` względem zmiennej `var`.

```
(%i26) diff(sin(x)*cos(x),x,2), trigreduce;
```

(%o26) $-2\sin\left(2\,x\right)$

```
(%i27) 'diff(sin(x)*cos(x),x,3) = diff(sin(x)*cos(x),x,3);
```

(%o27) $\frac{d^3}{d\,x^3}\left(\cos\left(x\right)\,\sin\left(x\right)\right)=4\sin\left(x\right)^2-4\cos\left(x\right)^2$

To samo z uproszczeniami:

```
(%i28) 'diff(sin(x)*cos(x),x,3) =
        diff(sin(x)*cos(x),x,3), trigreduce, ratsimp;
```

(%o28) $\frac{d^3}{d\,x^3}\frac{\sin\left(2\,x\right)}{2}=-4\cos\left(2\,x\right)$

### 13.2.2 Pochodne cząstkowe

Funkcja `diff(expr,var1, k,var2, l)`

oblicza pochodną rzędu k+l wyrażenia `expr`, k razy po `var1` oraz l razy po `var2`.

```
(%i29) 'diff(exp(x*y^2),x,1,y,2) = diff(exp(x*y^2),x,1,y,2);
```

(%o29) $\frac{d^3}{d\,x\,d\,y^2}\,e^{x\,y^2}=4\,x^2\,y^4\,e^{x\,y^2}+10\,x\,y^2\,e^{x\,y^2}+2\,e^{x\,y^2}$

Podobnie postępujemy przy pochodnych cząstkowych względem większej liczby zmiennych.

### 13.2.3 Funkcja depends

Jeżeli nie znamy wzoru na funkcję, a jedynie wiemy, że zależy ona od danych argumentów, to informację taką możemy przekazać do programu przy pomocy polecenia `depends`. Umożliwia to poprawne przekształcanie pochodnych takich funkcji.

```
(%i30) depends([U],[r,theta],[r,theta],[x,y]);
```

(%o30) $[\mathrm{U}\left(r,\theta\right),\mathrm{r}\left(x,y\right),\theta\left(x,y\right)]$

```
(%i31) diff(U,x)+diff(U,y);
```

(%o31) $$\left(\frac{d}{d\,y}\,\theta\right)\,\left(\frac{d}{d\,\theta}\,U\right)+\left(\frac{d}{d\,x}\,\theta\right)\,\left(\frac{d}{d\,\theta}\,U\right)+\left(\frac{d}{d\,y}\,r\right)\,\left(\frac{d}{d\,r}\,U\right)+\left(\frac{d}{d\,x}\,r\right)\,\left(\frac{d}{d\,r}\,U\right)$$

Jak widać mamy tu wzór na pochodną funkcji złożonej.

## 13.3 Granice

Funkcja
`limit(expr, x, val, dir)`
lub
`limit(expr, x, val)`

oblicza granicę wyrażenia `expr` przy `x` zmierzającym do wartości `val`. Jeżeli symbol `dir` ma wartość plus jest to granica prawostronna, jeżeli wartość minus - granica lewostronna.

Jeżeli argument `dir` jest pominięty, funkcja liczy granicę obustronną.

Symbol `val` może być symbolem `inf` (plus nieskończoność) lub `minf` (minus nieskończoność).

W wyniku mogą wystąpić symbole `und` (granica nieokreślona), `ind` (nieokreślona ale wyrażenie jest ograniczone) oraz `infinity` (nieskończoność zespolona).

Symbol `infinity` (nieskończoność zespolona) oznacza, że granica modułu wyrażenia jest plus nieskończoność, ale granicą wyrażenia nie jest ani plus ani minus nieskończoność.

```
(%i32) limit(x*log(x),x,0,plus);
```

(%o32) $0$

```
(%i33) limit((x+1)^(1/x),x,0);
```

(%o33) $e$

```
(%i34) limit(%e^x/x,x,inf);
```

(%o34) $\infty$

```
(%i35) limit(sin(1/x),x,0);
```

(%o35) $ind$

```
(%i36) limit((-2)^n,n,inf);
```

(%o36) $infinity$

## 13.4 Szeregi potęgowe

`Maxima` ma dwie komendy do rozwijania funkcji w szereg potęgowy.

Funkcja `taylor(expr,x,a,n)`

wylicza n pierwszych wyrazów rozwinięcia w szereg potęgowy wyrażenia `expr`, względem zmiennej `x`, w otoczeniu `a`.

```
(%i37)            taylor (sqrt (x + 1), x, 0, 5);
```

(%o37)/T/

$$1+\frac{x}{2}-\frac{x^2}{8}+\frac{x^3}{16}-\frac{5\,x^4}{128}+\frac{7\,x^5}{256}+\ldots$$

Funkcja `powerseries(expr,x,a)`

daje (stara się dać) ogólną postać szeregu potęgowego wyrażenia `expr`, względem zmiennej `x`, w otoczeniu `a`.

```
(%i38) powerseries(exp(x), x, 0);
```

(%o38) $$\sum_{i1=0}^{\infty}\frac{x^{i1}}{i1!}$$

Często bardziej czytelną formę szeregu daje użycie funkcji pomocniczej `niceindices`:

```
(%i39) niceindices(powerseries(exp(x), x, 0));
```

$$(\%o39)\quad \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

```
(%i40) taylor(exp(x), x, 0, 8);
```

(%o40)/T/

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \frac{x^6}{720} + \frac{x^7}{5040} + \frac{x^8}{40320} + \ldots$$

```
(%i41) powerseries(exp(x), x, 0);
```

$$(\%o41)\quad \sum_{i3=0}^{\infty} \frac{x^{i3}}{i3!}$$

```
(%i42) niceindices(powerseries(%, x, 0));
```

$$(\%o42)\quad \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

## 13.5 Sumy i iloczyny

Funkcja `sum(<expr>, <i>, <i_0>, <i_1>)`

sumuje wyrażenie `<expr>` względem wskaźnika sumacyjnego zmieniającego się od `<i_0>` do `<i_1>`. Suma może mieć skończoną lub nieskończoną liczbę składników.

```
(%i43) sum(l,l,1,10);
```

(%o43) 55

```
(%i44) sum(x^l,l,1,10);
```

$$(\%o44)\quad x^{10} + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x$$

```
(%i45) sum(1/k^2, k, 1, inf), simpsum;
```

$$(\%o45)\quad \frac{\pi^2}{6}$$

Funkcja `product(<expr>, <i>, <i_0>, <i_1>)`

mnoży wyrażenie `<expr>` względem wskaźnika `<j>` zmieniającego się od `<i_0>` do `<i_1>`.

```
(%i46)              product (x + i*(i+1)/2, i, 1, 4);
```

(%o46) $(x+1)\,(x+3)\,(x+6)\,(x+10)$

```
(%i47) product (i^2, i, 1, 7);
```

(%o47) 25401600

# Rozdział 14

# Matematyczny model populacji cykad

Do tej pory podawane przykłady ilustrowały proste funkcje. Na zakończenie tej części skryptu chcemy przedstawić wykorzystanie poznanych narzędzi do przedstawienia modelu matematycznego aktualnie wykorzystywanego w naukach biologicznych.

## 14.1 Kontekst biologiczny

U pewnych owadów obserwuje się zjawisko okresowego pojawiania sie bardzo licznej populacji dorosłych osobników przy bardzo niewielkiej liczbie owadów we wczesniejszych latach. W potocznym języku nazywamy to plagami (plaga komarów, plaga szarańczy, itp.). Co ciekawe, często zjawisko takie obserwuje się systematycznie.

Na przykład, w College Park, Maryland, regularnie co 17 lat pojawia się plaga cykad. Ponieważ jest to teren znanego uniwersytetu więc nic dziwnego, że zjawisko stało się obiektem naukowych badań. Między innymi starano się zbadać czy z modelu matematycznego opisującego zależności parametrów środowiska wynika tego typu zjawisko i jaka jest rola poszczególnych parametrów.

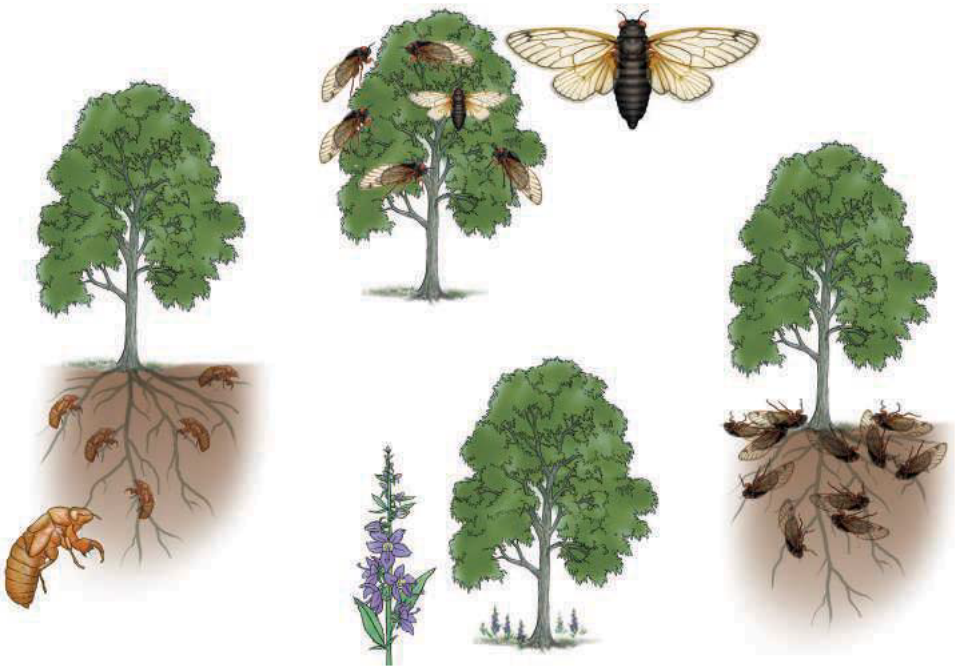
Poniżej przedstawiamy model pochodzący od Hoppensteadta i Kellera, zapisany w sposób wygodny do dalszej analizy przy pomocy poznanych przez nas narzędzi.

Zacznijmy od przedstawienia kilku informacji o fazach życia cykad. Przeważającą część wieloletniego życia cykada spędza pod powierzchnią ziemi w postaci poczwarki. Po osiągnięciu dojrzałości, poczwarka wychodzi na zewnątrz, gdzie spędza kilka tygodni już w postaci dorosłej cykady .

W tym czasie odbywają się gody i składanie jajeczek w szparach gałęzi drzew, po czym cykada umiera.

Jajeczka spadają na ziemię, gdzie wykluwają się z nich poczwarki. Poczwarki zagrzebują się w ziemi i przyczepiają się do korzeni. Tam spędzają większość życia czerpiąc pokarm z korzeni.

Po osiągnięciu dojrzałości (17 lat dla cykad Magicicada septendecim) przedostają tunelami się na powierzchnię i masowo wylatują, gdy temperatura osiągnie ok. 64 st. F (połowa maja).



Rysunek 14.1:

Cykady mają naturalnych wrogów (ptaki, wiewiórki, nawet psy), dla których są łatwym łupem. Ich masowe pojawianie się jest prawdopodobnie realzacją strategii przetrwania przez "nasycenie drapieżników" - przy

tak dużej liczbie jest duża szansa, że przeżyje dostatecznie wiele, aby zachował się gatunek.

W modelu należy zbilansować wielkość populacji cykad w poszczególnych fazach, zasoby pokarmowe środowiska i wpływ wrogów na zmniejszenie populacji.

## 14.2 Budowa modelu

### 14.2.1 Wybór parametrów i zmiennych

W modelu występują następujące stałe (w praktyce przyjmowane na drodze eksperymentalnej):

- `k` okres życia danego gatunku (w latach)
- `%mu` współczynnik przeżycia poczwarek w ciągu roku (po k latach ok. %mu^k), %mu jest wyświetlane jako grecka litera $\mu$,
- `%phi` współczynnik rozmnażania dorosłych osobników,%phi jest wyświetlane jako grecka litera $\phi$ ,
- `d` maksymalna pojemność środowiska (wyrażona liczbą poczwarek, dla których starczy pokarmu)

Z konteksty biologicznego wynika, że należy uwzględnić (co najmniej) pięć zmiennych wzajemnie ze sobą powiązanych:

- cykady (tzn. dorosłe osobniki wychodzące na powierzchnię);
- ofiary (inny sposób wyrażenia populacji wrogów),
- potomstwo (jajeczka),
- zapasy (pojemność środowiska),
- poczwarki (tzn. osobniki przebywające pod ziemią).

Bezpośrednio z natury problemu wynika, że jest on dyskretny – zmienne zależą od czasu wyrażonego w latach, ich wartościami są liczebności danej grupy w kolejnych latach.

Dla uproszczenia przyjmujemy, że wyznaczane eksperymentalnie wartości stałych (współczynników) można traktować jako dokładne (element losowości nie ma istotnego wpływu).

Tak więc, nasz model matematyczny będzie deterministycznym układem równań rekurencyjnych z pięcioma niewiadomymi:

- `cykady` = ciąg, którego elementami są liczby dorosłych osobników pojawiających się na powierzchni w kolejnych latach
- `ofiary` = ciąg, którego elementy oznaczają maksymalna liczbę cykad jaka mogą skonsumować wrogowie w kolejnych latach
- `potomstwo` = ciąg, którego elementami są liczby potomstwa w kolejnych latach
- `zapasy` = ciąg, którego elementami są liczby potomostwa dla którego starcza pożywienia w kolejnych latach
- `poczwarki` = ciąg, którego elementami są liczby poczwarek ulokowanych w korzeniach w kolejnych latach

Analizując treść biologiczną problemu staramy się wyznaczyć potrzebną liczbę relacji wiążących niewiadome.

### 14.2.2 Zapasy (pojemność środowiska)

Zapasy będziemy mierzyć liczbą poczwarek, które w danym momencie mogą dołączyć do grupy mając zagwarantowane pożywienie.

Zakładamy, że maksymalna pojemność środowiska wynosi `d` i ta pojemność jest redukowana przez liczbę poczwarek, które przeżyły do danego roku `j`.

Prosty model pojemności w roku `j` uwzględniający, że pojemność musi być nieujemna – od wyjściowej pojemności odejmujemy liczbę poczwarek, które przeżyły do roku j:

```
(%i1)  'zapasy[j] =
       'max(0, d - 'sum(%mu^i*poczwarki[j - i], i, 1, j - 1));
```

(%o1) $$zapasy_j = \max\left(0, d - \sum_{i=1}^{j-1} \mu^i \, poczwarki_{j-i}\right)$$

### 14.2.3 Cykady (osobniki wychodzące na powierzchnię)

Na powierzchnię w roku j wychodzi tyle dorosłych osobników ile przeżyło od roku j-k do roku j (k jest jednocześnie okresem życia i dojrzewania)

```
(%i2)  'cykady[j] = '%mu^k*poczwarki[j - k];
```

(%o2) $$cykady_j = \mu^k \, poczwarki_{j-k}$$

### 14.2.4 Potomstwo (jajeczka)

Liczba potomstwa w roku j dana jest iloczynem współczynnika rozrodczości i liczby dorosłych osobników w roku j pomniejszoną o liczbę osobników skonsumowanych w roku j:

```
(%i3)  'potomostwo[j] = 'max(0, %phi*(cykady[j] - ofiary[j]));
```

(%o3) $$potomostwo_j = \max\left(0, \phi\,\left(cykady_j - ofiary_j\right)\right)$$

### 14.2.5 Przyrost poczwarek w korzeniach

Mniejsza z liczb odpowiadająca pojemności i potomstwu (musi być nieujemna)

```
(%i4)  'poczwarki[j] = 'max(0,min(zapasy[j], potomstwo[j]));
```

(%o4) $$poczwarki_j = \max\left(0, \min\left(potomstwo_j, zapasy_j\right)\right)$$

### 14.2.6 Maksymalna liczba cykad "skonsumowanych" w roku j

Założenie: wielkości populacji drapieżników (wyrażona poprzez liczbę ofiar) zależy od

- jej wielkości w zeszłym roku ze współczynnikiem %nu
- liczby cykad w zeszłym roku ze współczynnikiem a.

```
(%i5)  'ofiary[j] ='%nu*ofiary[j - 1] + a*cykady[j - 1];
```

(%o5) $ofiary_j = \nu\, ofiary_{j-1} + a\, cykady_{j-1}$

## 14.3 Warunki początkowe

Zarówno z natury problemu jak i jego struktury matematycznej wynika, że wartości niewiadomych w pierwszych k latach muszą być zadane.

Jeżeli jednak założymy, że w naszej analizie startujemy "od początku", to wystarczy zadać k wartości zmiennej "poczwarki", ponieważ zmienne "ofiary", "cykady", "potomstwo", w tym okresie mają wartości zerowe (nic się nie dzieje dopóki pierwsze poczwarki po k latach nie wyjdą na powierzchnię), a wartości "zapasy" wyliczamy z zadanych wartości "poczwarki".

## 14.4 Matematyczne rozwiązanie problemu

Najbardziej pożądaną formą byłoby przedstawienie rozwiązania w postaci jawnych wzorów na niewiadome w zależności od liczby lat.Niestety najczęściej jest to niemożliwe (jak jest w tym przypadku, dla autora tych słów pozostaje otwartym problemem). Ograniczamy się jedynie do obserwacji, że przedstawione zależności pozwalają wyznaczać rekurencyjnie wartości niewiadomych dla zadanej liczby lat t i korzystamy ze wspomagania komputerowego przy obliczeniach i wizualizacji wyników.

## 14.5 Kod w Maximie

Budujemy procedurę `populacja`, której argumentami są:

- `poczwarki_0` lista będąca rozkładem zmiennej poczwarki w początkowych k latach (dla czytelności k zadajemy dalej jako argument)
- `t` liczba lat obserwacji ewolucji populacji
- `k` czas życia dla danego gatunku
- `mi` współczynnik przeżycia poczwarek w ciągu roku (wyżej oznaczana przez %mu)
- `f` współczynnik rozmnażania dorosłych osobników. (wyżej oznaczana przez %phi)
- `d` maksymalna pojemność środowiska
- `ni, a` współczynniki związane z liczbą ofiar (ni wyżej oznaczana przez %nu)

Po wykonaniu procedury zmienne `cykady`, `ofiary`, `potomstwo`, `zapasy`, `poczwarki` mają wartości będące listami o długości t, których elementy przedstawiają rozkład danej cechy w t latach.

```
(%i6)  populacja(poczwarki_0, t, k, mi, f, d, ni , a) :=
         block([i, j],
       /* rozkład poczwarek w pierwszych k latach*/
         poczwarki : reverse(rest(reverse(poczwarki_0),
                             length(poczwarki_0)-k)),
       /* rozkład pojemności środowiska  w pierwszych k
        latach*/
         zapasy :
             makelist(max(0,
             d - sum(mi^i*poczwarki[j - i],
             i, 1, j - 1)), j, 2, k),
         zapasy : append([d],zapasy),
       /* rozkład ofiar, cykad, jajeczek w pierwszych k
        latach, zakładamy same zera*/
         ofiary : cykady : potomstwo : makelist(0, i,1,k),
       /*budowanie list w pętli od roku k+1 do zadanej
        chwili końcowej t */
            for j:k+1 thru t do
               (cykady :
               append(cykady, [mi^k*poczwarki[j - k]]),
          ofiary : append(ofiary, [ni*ofiary[j - 1] +
                    a*cykady[j - 1]]),
          potomstwo : append(potomstwo, [max(0,
                      f*(cykady[j] - ofiary[j]))]),
          zapasy : append(zapasy, [max(0, d -
              sum(mi^i*poczwarki[j - i], i, 1, k - 1))]),
          poczwarki : append(poczwarki,
              [max(0,min(zapasy[j], potomstwo[j]))])
       )
           )$
```

Przetestujemy zbudowaną procedurę na argumentach sugerowanych przez specjalistów zajmujących się problemem.

Zakładamy, że gatunek żyje 7 lat, poczwarki początkowo rozmieszczone są równomiernie po 100 każdego roku.

Pytamy jak wygląda ewolucja w ciągu 130 lat.

```
(%i7)  poczwarki_0 : makelist(100, i,1,7)$
```

```
(%i8)  populacja(poczwarki_0, 130, 7, 0.95, 10, 10000, 0.95, 0.042);
```

(%o8) *done*

Z ciekawości odczytujemy wartości zmiennej cykady (podobnie możemy zrobić dla pozostałych zmiennych)

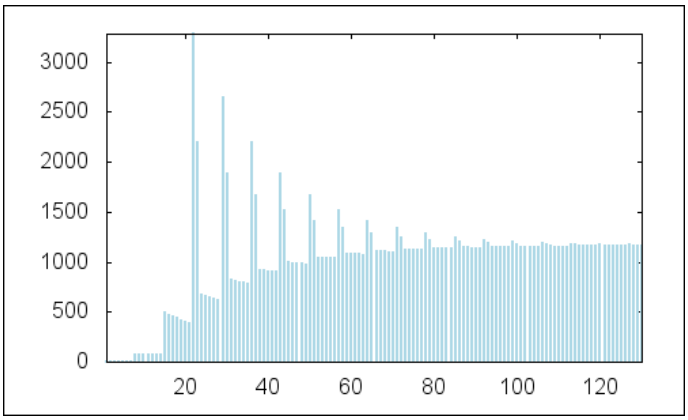
```
(%i9)  cykady;
```

$$\begin{aligned}(\%o9)\quad [&0,0,0,0,0,0,0,69.833729609375,69.833729609375,\\ &69.833729609375,69.833729609375,69.833729609375,\\ &69.833729609375,69.833729609375,487.6749791155298,\\ &467.1926299926776,447.7343983259679,429.2490782425937,\\ &411.6880241633883,395.0050227881432,379.1561714816602,\\ &3282.041046506122,2187.984942303423,661.8382771419915,\\ &648.9292886975438,636.6657496753151,625.0153876041949,\\ &613.9475436366367,2641.140318132662,1877.120136548887,\\ &811.3550009575589,802.3401728719593,\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad itd.\end{aligned}$$

Oczywiście bardziej czytelną informację daje nam wizualizacja

```
(%i10) load(draw)$
```

```
(%i11) wxdraw2d(points_joined = impulses,
                line_width    = 2,
                color         = "light-blue",
                points(cykady) )$
```




(%t11)

Wniosek: przy tej długości życia wielkość populacji stabilizuje się, nie ma żadnych "plag".

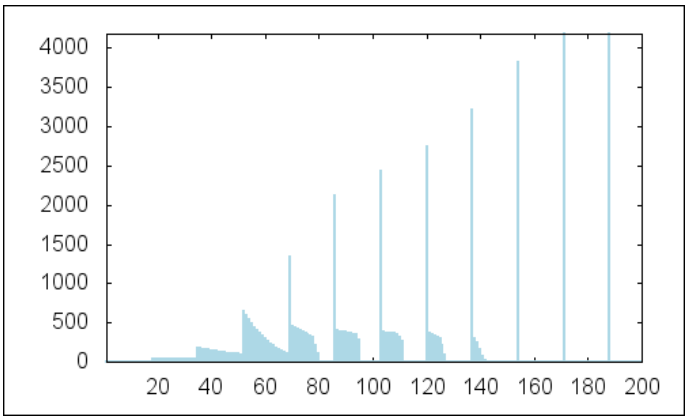
A teraz dla gatunku o długości życia 17 lat (czyli te z College Park), na przestrzeni 200 lat.

```
(%i12) poczwarki_0 : makelist(100, i,1,17)$

(%i13) populacja(poczwarki_0, 200, 17, 0.95, 10, 10000, 0.95, 0.042);
```

(%o13) *done*

```
(%i14) wxdraw2d(points_joined = impulses,
               line_width    = 2,
               color         = "light-blue",
               points(cykady) )$
```

(%t14)



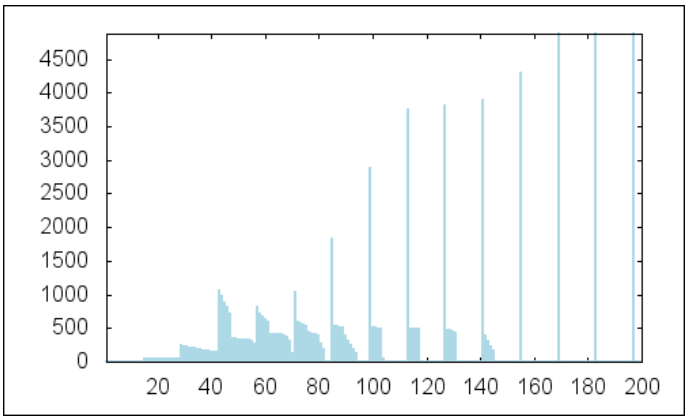

*Wniosek:* Przedstawiony model zgadza się z obserwacjami cyklicznych "plag".

Wykonajmy jeszcze jeden test dla potecjalnego gatunku o długości życia 14 lat

```
(%i15) poczwarki_0 : makelist(100, i,1,14)$

(%i16) populacja(poczwarki_0, 200, 14, 0.95, 10, 10000, 0.95, 0.042);
```

(%o16) *done*

```
(%i17) wxdraw2d(points_joined = impulses,
                line_width    = 2,
                color         = "light-blue",
                points(cykady) )$
```




(%t17)

Jak widać charakter jakościowy jest bardzo podobny. Jednak w rzeczywistości obserwuje się jedynie gatunki, których długości cykli są stosunkowo duzymi liczbami pierwszymi. Budzi to ożywione dyskusje wśród specjalistów. Przypuszczalny wyjaśnieniem jest próba uniknięcia jednoczesnego pojawienia się dużych populacji różnych gatunków.

Zapraszamy do własnych eksperymentów z różnymi parametrami i różnymi wizualizacjami zmiennych.