4. Чисельні методи та програмування з Maxima

4.1 Програмування на вбудованій макро мові

4.1.1 Умовні оператори

Основна форма умовного оператора: if$cond_1$**then** $expr_1$**else** $expr_0$. Якщо умова$cond_1$істинно, то виконується вираз$expr_1$, інакше - виконується вираз $expr_2$.

Пакет Maxima дозволяє використовувати різні форми оператораif наприклад: if$cond_1$**then** $expr_1$**elseif** $cond_2$**then** $expr_2$ **elseif...**`else`$expr_0$

Якщо виконується умова$cond_1$, то виконується вираз$expr_1$, інакше - перевіряється умова$cond_2$, і якщо воно істинне - виконується вираз$expr_2$, і т.д. Якщо жодна з умов не є істинною — виконується вираз$expr_0$.

Альтернативні вирази$expr_1, exp_r2, ..., expr_k$ - довільні вирази Maxima (в т.ч. вкладені операториif). Умови — справді чи потенційно логічні вирази, що зводяться до значень$true$ або $false$. Спосіб інтерпретації умов залежить від значення прапора$prederror$. Якщо$prederror = true$, видається помилка, якщо значення якогось із виразів$cond_1, ..., cond_n$ відрізняється від$true$ або $false$. Якщо$prederror = false$ і значення якогось із виразів $cond_1, ..., cond_n$ відрізняється від$true$ або $false$, результат обчисленняif - Умовний вираз.

4.1.2 Оператори циклу

Для виконання ітерацій використовують оператор text tt do. Можуть використовуватися три варіанти його виклику, що відрізняються умовою закінчення циклу:

for $variable : init_valuestepincrement$ **thru** $limit$ **do** $body$

for $variable : init_valuestepincrement$ **while** $condition$ **do** $body$

for $variable : init_valuestepincrement$ **unless** $condition$ **do** $body$

Тут$variable$ - Змінна циклу;$init_value$ - Початкове значення; $increment$ - крок (за замовчуванням дорівнює 1);$limit$ - Кінцеве значення змінної циклу;$body$ - Оператори тіла циклу.

Ключові слова thru, while, unless вказують на спосіб завершення циклу:

- після досягнення змінної циклу значення$limit$;
- поки виконується умова$condition$;
- поки не буде досягнуто умови$condition$.

Параметри $init_value, increment, limit$, і $body$ можуть бути довільними виразами. Контрольна змінна після завершення циклу передбачається позитивною (при цьому початкове значення може бути негативним). Вирази $limit, increment$, умови завершення ($condition$) обчислюються кожному кроці циклу, тому їх складність впливає час виконання циклу.

При нормальному завершенні циклу величина, що повертається — атом $done$. Примусовий вихід із циклу здійснюється за допомогою оператора return, який може повертати довільне значення.

Контрольна змінна циклу - локальна всередині циклу, тому її зміна в циклі не впливає на контекст (навіть за наявності поза циклом змінної з тим самим ім'ям).

Приклади:

```
(%i1) for a:-3 thru 26 step 7 do display(a)$
```

$$a = -3$$

$$a = 4$$

$$a = 11$$

$$a = 18$$

$$a = 25$$

```
(%i2) s: 0$for i: 1 while i <= 10 do s: s+i;
```

$$(\%o3) \qquad done$$

```
(% i4)     s;
```

$$(\%o4) \qquad 55$$

```
(% i5)     series: 1$ term: exp (sin (x))$
(%i7) for p:1 unless p > 7 do
      (term: diff (term, x)/p, series: series + subst
      (x=0,term)*x^p)$
(% i8)     series;
```

$$(\%o8) \qquad \frac{x^7}{90} - \frac{x^6}{240} - \frac{x^5}{15} - \frac{x^4}{8} + \frac{x^2}{2} + x + 1$$

```
(% i9)     for count: 2 next 3*count thru 20 do display
(count)$
```

$$count = 2$$

$$count = 6$$

$$count = 18$$

Умови ініціалізації та завершення циклу можна опускати.

Приклад (цикл без явної вказівки змінної циклу):

```
(%i10)     x:1000;
```

$$(\%o10) \qquad 1000$$

```
(%i11)     thru 20 do x: 0.5*(x + 5.0/x)$(%i12) x;
```

$$(\%o12) \qquad 2.23606797749979$$

```
(%i12)     float(sqrt(5));
```

```
(%o12)                2.23606797749979
```

За 20 ітерацій досягається точне значення $\sqrt{5}$.

Дещо витонченіший приклад — реалізація методу Ньютона для рівняння з однією невідомою (обчислюється та ж величина — корінь з п'яти):

```
(%i1) Newton (f, x): = ([y, df, dfx], df: diff (f ('x),
'x),
      do (y: ev(df), x: x - f(x)/y,
      if abs (f(x)) < 5e-6 then return (x)))
$(%i2)     f(x):=x^2-5;
(%o2)                  f(x) := x^2 - 5
(%i3) float(newton(f,1000));
(%o3)              2.236068027062195
```

Ще одна форма оператора циклу характеризується вибором значень змінної циклу із заданого списку. Синтаксис виклику:

for $variable$ **in** $listendtests$ **do** $body$

Перевірка умови завершення$endtests$ до вичерпання списку$list$ може бути відсутнім.

Приклад:

```
(%i1)    a:[];
(%o1)                       []
(%i2)    for f in [1,4,9,16] a:cons(sqrt(f),a)$
(%i3)    a;
(%o3)                    [4, 3, 2, 1]
```

4.1.3 Блоки

Як умовних висловлюваннях, і у циклах замість простих операторів можна писати складові оператори, тобто. блоки. Стандартний блок має вигляд: block([r,s,t],r:1,s:r+1,t:s+1,x:t,t*t);

Спочатку йде список локальних змінних блоку (глобальні змінні з тими самими іменами не пов'язані з цими локальними змінними). Список локальних змінних може бути пустим. Далі йде набір операторів.

Спрощений блок має вигляд: (x:1,x:x+2,a:x); Зазвичай у циклах та в умовних виразах застосовують саме цю форму блоку. Значення блоку є значення останнього з його операторів. Всередині цього блоку допускаються оператор переходу на мітку та оператор return. Оператор return припиняє виконання поточного блоку і повертає як значення блоку свій аргумент block([],x:2,x:x*x, return(x), x:x*x);

Без оператора переходу на мітку, оператори в блоці виконуються послідовно.

(У даному випадку слово "мітка" означає аж ніяк не мітку типу "%i5" або "%o7").

Оператор go виконує перехід на мітку, розташовану в цьому ж блоці:

```
(%i1) block([a],a:1,metka, a:a+1,
```

```
      if a=1001 then return(-a),go(metka));
```
$$(\%o1) \quad -1001$$

У цьому блоці реалізований цикл, який завершується після досягнення "змінної циклу" значення 1001. Міткою може бути довільний ідентифікатор.

Слід пам'ятати, що цикл сам собою блоком, отже (на відміну мови С) перервати виконання циклів (особливо вкладених циклів) з допомогою оператора go неможливо, т.к. оператор go та мітка виявляться у різних блоках. Те саме стосується оператора return. Якщо цикл, розташований усередині блоку, містить оператор return, то при виконанні оператора return відбудеться вихід із циклу, але не вихід із блоку:

```
(%i1) block([],x:for i:1 thru 15 do
      if i=2 then return(555),display(x),777);
```
$$x = 555$$
$$(\%o1) \quad 777$$
```
(%i2) block([],x:for i:1 thru 15 do
      if i=52 then return(555),display(x),777);
```
$$x = done$$
$$(\%o2) \quad 777$$

Якщо необхідно вийти з декількох вкладених блоків одразу (або кількох блоків і циклів одразу) і при цьому повернути деяке значення, слід застосовувати блок catch

```
(%i3) catch( block([],a:1,a:a+1, throw(a),a:a+7),a:a+9 );
```
$$(\%o3) \quad 2$$
```
(% i4)      a;
```
$$(\%o4) \quad 2$$
```
(% i5)      catch(block([],for i:1 thru 15 do
      ifi=2 then throw(555)),777);
```
$$(\%o5) \quad 555$$

У цьому блоці виконання циклу завершується, як тільки значення i досягає 2. Повертається блоком catch значення дорівнює 555.

```
(%i6) catch(block([],for i:1 thru 15 do
      if i=52 then throw(555)),777);
```
$$(\%o6) \quad 777$$

У даному блоці виконання цикл виконується повністю, і повертається блоком catch значення 777 (умови виходу з циклу за допомогою throw не досягаються).

Оператор throw — аналог оператора return, але він обриває не поточний блок, а всі вкладені блоки аж до першого catch, що зустрівся.

Нарешті, блок errcatch дозволяє перехоплювати деякі (на жаль, не всі!) помилки, які в нормальній ситуації призвели б до завершення рахунку.

Приклад:

```
(%i1) errcatch(a:1, b:0, log(a/b), c:7);
expt: undefined: 0 до негативного exponent.
```

```
(%o1)                          []
(%i2) c;
(%o2)                           c
```

Виконання послідовності операцій переривається на першій операції, яка призводить до помилки. Інші вирази блоку не виконуються (значення с залишається невизначеним). Повідомлення про помилку може бути виведено функцією $errormsg()$.

4.1.4 Функції

Поряд із найпростішим способом завдання функції, Maxima допускає створення функції у вигляді послідовності операторів:

f(x) :=$(expr_1, expr_2, ..., expr_n)$;

Значення, що повертається функцією - значення останнього виразу $expr_n$.

Щоб використовувати оператор return і змінювати значення, що повертається в залежності від логіки роботи функції, слід застосовувати конструкцію block, наприклад:

f(x)=block([], $expr_1$, ..., if (a > 10) then return(a), ..., $expr_n$).

При $a > 10$ виконується оператор return та функція повертає значення a, в іншому випадку - значення виразу $expr_n$.

Формальні параметри функції або блоку – локальні, і є видимими лише всередині них. Крім того, при завданні функції можна оголосити локальні змінні – у квадратних дужках на початку оголошення функції або боки.

Приклад:

block ([a: a], $expr_1$**, ..., a: a+3, ...,** $expr_n$**)**

В даному випадку при оголошенні блоку в локальній змінній a зберігається значення глобальної змінної a, Визначеної ззовні блоку.

Приклад:

```
(%i1) f(x):=([a:a],if a>0 then 1 else (if a<0 then -1
else 0));
(%o1)
f(x) := ([a : a], if a > 0 then 1 else if a < 0 then  − 1 else 0)
(%i2) a:1;
(%o2)                           1
(%i3) f(0);
(%o3)                           1
(% i4)      a:-4;
(%o4)                          −4
(% i5)      f(0);
(%o5)                          −1
(%i6) a:0;
(%o6)                           0
(%i7) f(0);
```

(%o7) 0

У цьому прикладі значення змінної a задається поза тілом функції, але результат, що повертається нею, залежить від значення a .

Початкові значення локальних змінних функції можуть задаватися двома способами:

1. Завдання функції f(x) := ($expr_1$, ..., $expr_n$);виклик функції f(1); - Початкове значення локальної змінної x дорівнює 1.
2. Завдання блоку block ([x: 1], $expr_1$, ..., $expr_n$), при цьому початкове значення локальної змінної x також одно 1.

Поряд з іменованими функціями Maxima дозволяє використовувати і безіменні функції (лямбда-функції).

Синтаксис використання лямбда-виразів (щоправда, при використанні з лямбда виразами таки асоціюється ім'я - див. приклад):

$f1 : lambda([x_1, \ldots, x_m], expr_1, \ldots, expr_n)$

$f2 : lambda([[L]], expr_1, \ldots, expr_n)$

$f3 : lambda([x_1, \ldots, x_m, [L]], expr_1, \ldots, expr_n)$

Приклад:

```
(%i1)    f: lambda ([x], x^2);
```

(%o1) $lambda\left([x], x^2\right)$

```
(%i2)    f(a);
```

(%o2) a^2

Більш складний приклад (лямбда-вирази можуть використовуватися в контексті, коли очікується ім'я функції):

```
(%i3)    lambda ([x], x^2) (a);
```

(%o3) a^2

```
(% i4)   apply (lambda ([x], x^2), [a]);
```

(%o4) a^2

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

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

Аргументи лямбда-виразів – локальні змінні. Інші змінні під час обчислення лямбда-выражений розглядаються як глобальні. Винятки відзначаються спеціальним символом - прямими лапками (див. лямбда-функцію $g2$ у прикладі).

```
(%i6)    a: %pi$ b: %e$ g: lambda ([a], a*b);
```

(%o8) $lambda\left([a], a\, b\right)$

```
(% i9)   b: %gamma$ g(1/2);
```

(%o10) $\frac{\gamma}{2}$

```
(%i11)   g2: lambda ([a], a*"b);
```

(%o11) $lambda\left([a], a\, \gamma\right)$

```
(%i12)   b: % e $ g2 (1/2);
```

(%o13) $\frac{\gamma}{2}$

Лямбда функції можуть бути вкладеними.

При цьому локальні змінні зовнішнього виразу доступні як глобальні для внутрішнього (однакові імена змінних маскуються).

Приклад:

```
(%i1)    h: lambda ([a, b], h2: lambda ([a],
a*b),h2(1/2));
```

(%o1) $lambda\left([a,b], h2 : lambda\left([a], a\,b\right), h2\left(\frac{1}{2}\right)\right)$

```
(%i2)    h(%pi, %gamma);
```

(%o2) $\frac{\gamma}{2}$

Подібно до звичайних функцій, лямбда-функції можуть мати список параметрів змінної довжини.

Приклад:

```
(%i1)    f: lambda ([aa, bb, [cc]], aa * cc + bb);
```

(%o1) $lambda\left([aa, bb, [cc]], aa\,cc + bb\right)$

```
(%i2)    f(3,2,a,b,c);
```

(%o2) $[3\,a + 2, 3\,b + 2, 3\,c + 2]$

Список $[cc]$ при виклику лямбда-функції f включає три елементи: $[a, b, c]$. Формула для розрахунку f застосовується до кожного списку.

Локальні змінні можуть бути оголошені за допомогою функції $local$. Змінні $v_1, v_2, ..., v_n$ оголошуються локальними викликом $local(v_1, v_2, ..., v_n)$ незалежно від контексту.

4.1.5 Транслятор і компілятор Maxima

Визначивши ту чи іншу функцію, можна помітно прискорити виконання, якщо її від транслювати або відкомпілювати. Це відбувається тому, що якщо Ви не транслювали і не відкомпілювали певну Вами функцію, то при кожному черговому її виклику Maxima щоразу знову виконує ті дії, які входять у визначення функції, тобто. фактично розбирає відповідний вираз лише на рівні синтаксису Maxima.

4.1.5.1 Функція translate

Функція $translate$ транслює функцію Maxima мовою Lisp. Наприклад, вираз: $f(x) := 1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7$ транслюється командою: translate(f); Після цього функція зазвичай починає обчислюватися швидше.

Приклад, що ілюструє виграш за часом після трансляції функції:

```
(%i1) f(n):=block([sum,k],sum:0,
      for k:1 thru n do (sum:sum+k^2),sum)$
```

Функція $f(n)$, організована у вигляді блоку, дозволяє обчислити суму $\sum_{k=1}^{k=n} k^2$.

Для виконання тестів використовувався той самий комп'ютер і Maxima 5.24. При безпосередньому зверненні до функції f час обчислення f (1000000) становило 7,86 с, після трансляції - 3,19 с. Для оцінки часу обчислення використано функцію $time$.

```
(%i2)    f(1000000);
(%o2)    333333833333500000
(%i3)    time(%o2);
(%o3)    [7.86]
(% i4)   translate(f);
(%o4)    [f]
(% i5)   f(1000000);
(%o5)    333333833333500000
(%i6)    time(%o5);
(%o6)    [3.19]
```

Функція $time(\%o1, \%o2, ...)$ повертає список періодів часу в секундах, витрачених для обчислення результатів $\%o1, \%o2, \ldots$. Аргументом функції $time$ можуть бути тільки номери рядків виводу, для будь-яких інших змінних функція повертає значення $unknown$.

4.1.5.2 Функція compile

Функція $compile$ спочатку транслює функцію Maxima на мову Lisp, а потім компілює цю функцію $Lisp$ до двійкових кодів та завантажує їх на згадку.

Пример :

```
(% i9)     compile(f);
Compiling /tmp/gazonk_1636_0.lsp.
End of Pass 1.
End of Pass 2.
OPTIMIZE levels: Safety=2,
Space=3, Speed=3
Finished compiling /tmp/gazonk_1636_0.lsp.
(%o92)     [f]
```

Після цього функція (зазвичай) починає вважатися набагато швидше, ніж після трансляції. Наприклад, після компіляції функції f з останнього прикладу час обчислення f (1000000) становило 2.17 с.

Слід пам'ятати, що як із трансляції, і при компіляції Maxima намагається оптимізувати функцію за швидкістю. Проте Maxima працює переважно з цілими числами довільної довжини чи текстовими висловлюваннями. Тому при роботі з великими функціями можуть виникнути проблеми, пов'язані з

перетворенням типів даних. У цьому випадку слід відмовитись від трансляції чи компіляції, або переписати функцію, упорядкувавши використання типів.

Приклад: Розглянемо дві функції, що обчислюють один і той самий вираз. У функції $f2$ явно зазначено, що функція повертає дійсні значення (у форматі з плаваючою точкою)

```
f1(x,n):=block([sum,k], sum:1,
      for k:1 thru n do (sum:sum+1/x^k),sum)$

f2(x,n):=block([sum,k],
      mode_declare ([function (f2), x], float),
      sum:1,for k:1 thru n do (sum:sum+1/x^k),sum)$
```

Час виконання функції $f1$ під час запуску $f1$(5, 10000) становило 1,8 с. Після компіляції час виконання становив 1,49 с, після трансляції - 1,39 с. Спроба звернутися до відкомпільованої функції $f1$ командою $f1$(5.0, 10000.0) завершилася невдачею внаслідок помилки (плаваюче переповнення).

При використанні функції з декларованим типом результату (f2) час виконання f2(5, 10000) виявився меншим, ніж $f1$(1,65 с замість 1,8 с). Однак час виконання тієї ж функції після трансляції чи компіляції перевищує 10 секунд. Слід врахувати, що у разі результат розрахунку — раціональне число. Перетворення його до форми з плаваючою точкою для обчислення чергового значення суми вимагає додаткових обчислювальних витрат. При зверненні до $f2$ із дійсними аргументами $f2$(5.0, 10000.0) час рахунку становив лише 0,16 з.

Для функції, що повертає результат, який подається у вигляді числа з плаваючою точкою, компіляція або трансляція може дати зменшення часу рахунку в кілька разів.

Приклад: Розглянемо функції, що обчислюють дійсний вираз (у даному випадку підсумовуються ірраціональні числа)

```
f3(x,n):=block([sum,k],
    mode_declare ([function (f3), x], float),
    sum:1, for k:1 thru n do (sum:sum+sqrt(x^k)),sum)$
```

Час обчислення виразу $f3$(5, 2000) для невідкомпільованої та не відтрансльованої функції склало 7,47 с., після трансляції час обчислення $f3$(5, 2000) становило 0,03 с, після компіляції - 0,02 с.

Розглянемо ще один приклад:

```
f4(x,n):=block([sum,k], sum:1,
      for k:1 thru n do (sum:sum+k/x),sum)$
```

Час обчислення виразу $f4$(5, 1000000) склало 10,89 с, час обчислення виразу $f4$(5.0, 1000000) становило 6,71 с. Після трансляції $f4$ час обчислення виразу $f4$(5, 1000000) склало 9,1 с (виграш за часом практично відсутня), а для f4 (5.0, 1000000) - 2,49 с (виграш за часом за рахунок виконання обчислень з плаваючою точкою приблизно в 2,5 рази).

4.2 Введення-виведення у пакеті Maxima

У цьому розділі розглядаються конструкції, що дозволяють здійснити обмін даними між Maxima та іншими програмами.

4.2.1 Введення-виведення даних у консолі

Основна функція для зчитування даних, що вводяться користувачем: $read(expr_1, ..., expr_n)$. Вирази, що вводяться $expr_1, expr_2, \cdots$ при введенні інтерпретуються. Поля введення розділяються точками з комою або знаком $. Аргументи функції $read$ можуть містити підказку.

Приклад:

```
(%i1) a:42$
(%i2) a:read("Значення a = ",a," введіть нову величину")
;
Значення a = 42 уведіть нову величину (p+q)^3;
```

$$(\%o2) \quad (q+p)^3$$

```
(%i3) display(a);
```

$$a = (q+p)^3$$

$$(\%o3) \quad done$$

Аналогічна функція $readonly$ здійснює лише введення даних (без їхньої інтерпретації).

приклад(порівняння використання функцій $read$ і $readonly$):

```
(%i1) a:7$
(%i2) readonly("Введіть вираз:");

Введіть вираз: 2^a;
```

$$(\%o2) \quad 2^a$$

```
(%i3) read("Введіть вираз:");

Введіть вираз: 2^a;
```

$$(\%o3) \quad 128$$

Виведення на екран здійснюється функцією $display$. Синтаксис її виклику: $display(expr_1, expr_2, ...)$.

Вирази зі списку аргументів виводяться зліва направо (спочатку саме вираз, та був після знака рівності — його значення).

Аналогічна функція $disp$ - синтаксис виклику:

$disp(expr+1, expr+2, ...)$

виводить на екран лише значення виразу після його інтерпретації.

Функція $grind$ здійснює виведення в консоль Maxima аналогічно $disp$ але у формі, зручній для введення з клавіатури.

```
(%i1) a:1$ b:2$ c:3$
(% i4)     display(a,b,c);
```
$$a = 1$$
$$b = 2$$
$$c = 3$$
$$(\%o4)\ done$$
```
(% i5)     disp(a,b,c);
```
$$1$$
$$2$$
$$3$$
$$(\%o5)\ done$$
```
(%i6) grind(a);
```
$$1$$
$$(\%o6)\ done$$

Управління консольним введенням/виводом здійснюється за допомогою встановлення прапорів $display2d, display_format_internal$ і т.п.

Виведення на екран довгих виразів частинами (одна частина над іншою) здійснюється функцією $dispterms$. Синтаксис виклику:

$dispterms(expr)$).

Крім того, для виведення результатів обчислень використовується функція $print$. Синтаксис виклику:

$print(expr_1, ..., expr_n)$.

Вирази $expr_1, ..., expr_n$ інтерпретуються і виводяться послідовно в рядок (на відміну від виведення, що генерується функцією $display$). Функція $print$ повертає значення останнього інтерпретованого виразу.

Приклад:

```
(%i1) a:1$ b:2$ c:(a^2+b^2)$
(% i4)     rez:print("Приклад:",a,b,c);

Приклад: 1 2 5
```
$$(\%o4)\ 5$$
```
(% i5)     rez;
```
$$(\%o5)\ 5$$
```
(%i6) display("Приклад:",a,b,c);

Приклад: = Приклад:
```

$a = 1$
$b = 2$
$c = 5$
$(\%o6)\ done$

4.2.2 Файлові операції введення-виводу

4.2.2.1 Введення-виведення текстових даних

Збереження поточного стану робочої області Maxima здійснюється за допомогою функції$save$. Ця функція дозволяє зберегти у файлі окремі об'єкти із зазначеними іменами. Варіанти виклику$save$[1]:

$save(filename, name_1, name_2, name_3, ...)$

— зберігає поточні значення змінних$name_1, name_2, name_3, \cdots$у файлі $filename$. Аргументи мають бути іменами змінних, функцій чи інших об'єктів. Якщо ім'я не асоціюється з будь-якою величиною пам'яті, воно ігнорується. Функція save повертає ім'я файлу, у якому збережено задані об'єкти.

$save(filename, values, functions, labels, ...)$

- Зберігає всі значення змінних, функцій, міток тощо.

$save(filename, [m, n])$

— зберігає всі значення позначок введення/виводу в проміжку відmдоn(m, n - Цілі літерали).

$save(filename, name_1 = expr_1, ...)$

- дозволяє зберегти об'єкти Maxima із заміною імені$expr_1$на ім'я$name_1$.

$save(filename, all)$

— зберігає всі об'єкти, які є у пам'яті.

Глобальний прапор $file_output_append$ керує режимом запису. Якщо $file_output_append = true$, результати висновку$save$додаються до кінця файлу результатів. Інакше файл результату переписується. Незалежно від $file_output_append$, якщо файл результатів немає, він створюється.

Дані, збережені функцією$save$, можуть бути знову завантажені функцією $load$ (Див. нижче).

Варіанти запису за допомогою save можуть поєднуватися один з одним (приклад:$save$($filename, aa, bb, cc = 42, functions$, [11,17]).

Завантаження попередньо збереженого функцією$save$файлу здійснюється функцією$load(filename)$.

Аналогічний синтаксис у функції$stringout$ яка призначена для виведення у файл виразів Maxima у форматі, придатному для подальшого зчитування Maxima. Синтаксис виклику

$stringout:$
$stringout(filename, expr_1, expr_2, expr_3, \ldots)$
$stringout(filename, [m, n])$
$stringout(filename, input)$
$stringout(filename, functions)$
$stringout(filename, values)$

Функція$load(filename)$обчислює вирази у файлі$filename$, створюючи таким чином змінні, функції та інші об'єкти Maxima. Якщо об'єкт з деяким ім'ям вже присутній у Maxima, під час виконання$load$він буде заміщений зчитуваним. Щоб знайти файл, що завантажується, функція$load$використовує змінні$file_search, file_search_maxima$і$file_search_lisp$як довідники пошуку. Якщо файл, який завантажується, не знайдено, друкується повідомлення про помилку.

Завантаження працює однаково добре для коду на Lisp та коду на макромові Maxima. Файли, створені функціями$save, translate_file, compile_file$містять код на Lisp, а створені за допомогою функції$stringout$містять код Maxima. Всі ці файли можуть з рівним успіхом бути оброблені функцією$load$. $Load$використовує функцію$loadfile$, щоб завантажити файли Lisp та$batchload$, щоб завантажити файли Maxima.

Loadне розпізнає конструкції :lisp у файлах, що містять код на Maxima, а також глобальні змінні$_, __, \%$, і$\%th$, доки не буде створено відповідних об'єктів у пам'яті.

Функція$loadfile(filename)$призначена для завантаження файлів, що містять код на Lisp, створені функціями$save, translate_file, compile_file$. Для завдань кінцевого користувача зручніша функція$load$.

Протокол сесії Maxima може записуватись за допомогою функції$writefile$(Він записується у форматі виведення на консоль). Для тих же цілей використовується функція$appendfile$(запис на кінець існуючого файла). Завершення запису та закриття файлу протоколу здійснюється функцією$closefile$. Синтаксис виклику: $writefile(filename), closefile(filename)$.

4.2.2.2 Введення-виведення командних файлів

Основна функція, призначена для введення та інтерпретації командних файлів – функція$batch(filename)$Функція$batch$читає вирази Maxima з файлу$filename$та виконує їх. Функція$batch$шукає$filename$у списку $file_search_maxima$. ім'я файлу$filename$включає послідовність виразів

Maxima, кожне з яких має закінчуватися; або $. Спеціальна змінна $\%$ та функція $\%th$ звертаються до попередніх результатів у межах файлу. Файл може включати конструкції: lisp. Пробіли, табуляції, символи кінця рядка у файлі ігноруються. Підходящий вхідний файл може бути створений редактором тексту або функцією $stringout$.

Функція $batch$ зчитує кожен вираз із файлу $filename$, показує введення консолі, обчислює відповідні вирази і показує висновок також у консолі. Мітки введення призначаються вхідним виразам, мітки виведення – результатам обчислень, функція $batch$ інтерпретує кожен вхідний вираз, доки досягне кінець файла. Якщо передбачається реакція користувача (введення з клавіатури), виконання $batch$ зупиняється до завершення введення. Для зупинення виконання batch-файлу використовується Ctrl-C.

Функція $batchload(filename)$ зчитує та інтерпретує вирази з командного файлу, але не виводить на консоль вхідних та вихідних виразів. Мітки введення та виведення виразів, що зустрічаються в командному файлі, також не призначаються. Спеціальна змінна $\%$ та функція $\%th$ звертаються до попередніх діалогових міток, не маючи результатів у межах файлу. Крім того, файл $filename$ не може включати конструкції: lisp.

4.3 Вбудовані чисельні методи

4.3.1 Чисельні методи розв'язання рівнянь

4.3.1.1 Вирішення рівнянь з одним невідомим

Для вирішення рівняння з одним невідомим у пакеті Maxima передбачено функцію $find_root$. Синтаксис виклику:

- $find_root(expr, x, a, b)$
- $find_root(f, a, b)$

Пошук кореня функції f або вираз expr щодо змінної x здійснюється в межах $a \leqslant x \leqslant b$.

Для пошуку коренів використовується метод поділу навпіл або, якщо досліджувана функція досить гладка, метод лінійної інтерполяції.

4.3.2 Вирішення рівнянь методом Ньютона: пакет newton1

Основна функція пакета newton1 призначена для вирішення рівнянь методом Ньютона.

Синтаксис виклику: $newton(expr, x, x0, eps)$

Ця функція повертає наближене рішення рівняння $expr = 0$ методом Ньютона, розглядаючи $expr$ як функцію однієї змінної x . Пошук починається з $x = x_0$ і проводиться, доки не буде досягнуто умови $abs(expr) < eps$.

Функція $newton$ допускає наявність невизначених змінних у вираженні $expr$, при цьому виконання умови $abs(expr) < eps$, оцінюється як справжнє чи хибне. Таким чином, немає необхідності оцінювати $expr$ лише як число.

Для використання пакета потрібно завантажити його командою load(newton1).

Приклади використання функції $newton$:

```
(%i1) load (newton1);
```
$$(\%o1)\ /usr/share/maxima/5.26.0/share/numeric/newton1.mac$$
```
(%i2) Newton (cos (u), u, 1, 1/100);
```
$$(\%o2)\ 1.570675277161251$$
```
(%i3) ev (cos(u), u = %);
```
$$(\%o3)\ 1.2104963335033529\,10^{-4}$$
```
(% i4)     assume (a > 0);
```
$$(\%o4)\ [a > 0]$$
```
(% i5)     newton (x^2 - a^2, x, a/2, a^2/100);
```
$$(\%o5)\ 1.000304878048 78\,a$$
```
(%i6) ev (x^2 - a^2, x = %);
```
$$(\%o6)\ 6.0984904818539 58\,10^{-4}\,a^2$$

4.3.2.1 Вирішення рівнянь з кількома невідомими: пакет mnewton

Потужна функція для вирішення систем нелінійних рівнянь методом Ньютона входить до складу пакету mnewton. Перед використанням пакет необхідно завантажити:

```
(%i1) load("mnewton");
```
$$(\%o1)$$
$$/usr/share/maxima/5.13.0/share/contrib/mnewton.mac$$

Після завантаження пакета mnewton стають доступними основна функція. $mnewton$ та ряд додаткових змінних для керування нею: $newtonepsilon$ (точність пошуку, величина за замовчуванням $10.0^{-\frac{fpprec}{2}}$), $newtonmaxiter$ (Максимальне число ітерацій, величина за замовчуванням 50).

Синтаксис виклику: $mnewton(FuncList, VarList, GuessList)$, де $FuncList$ - Список функцій, що утворюють розв'язувану систему рівнянь, $VarList$ - Список імен змінної, і $GuessList$ - Список початкових наближень.

Рішення повертається в тому ж форматі, який використовує функція $solve()$. Якщо рішення не знайдено, повертається пустий список.

Приклад використання функції $mnewton$:

```
(%i1)    load("mnewton")$
(%i2)    mnewton([x1+3*log(x1)-x2^2,2*x1^2-x1*x2-5*x1+1],
         [x1, x2], [5, 5]);
```
$$(\%o2)\ [[x1 = 3.756834008012769, x2 = 2.779849592817897]]$$

```
(%i3)    mnewton([2*a^a-5],[a],[1]);
```
$$(\%o3)\ [[a = 1.709275567861 44]]$$

Як видно з другого прикладу, функція $mnewton$ може використовуватись і для вирішення одиничних рівнянь.

4.3.3 Інтерполяція

Для виконання інтерполяції функцій, заданих таблично, у складі Maxima передбачено пакет розширення interpol, що дозволяє виконувати лінійну або поліноміальну інтерполяцію. Пакет включає службову функцію $charfun2(x, a, b)$, яка повертає $true$, якщо число x належить інтервалу $[a, b)$, і $false$ інакше.

4.3.3.1 Лінійна інтерполяція

Лінійна інтерполяція виконується функцією $linearinterpol$ Синтаксис виклику: $linearinterpol(points)$ або $linearinterpol(points, option)$.

Аргумент $points$ має бути представлений в одній з наступних форм:

- матриця з двома стовпцями, наприклад p:matrix([2,4], [5,6], [9,3]), при цьому перше значення пари або перший стовпець матриці - це значення незалежної змінної,
- список пар значень, наприклад p:[[2,4], [5,6], [9,3]],
- список чисел, які розглядаються як ординати функції, що інтерполюється, наприклад p:[4,6,3], в цьому випадку абсциси призначаються автоматично (приймають значення 1, 2, 3 і т.д.).

Як опція вказується ім'я незалежної змінної, щодо якої будується інтерполяційна функція.

Приклади виконання лінійної інтерполяції:

```
(%i1)    load("interpol")$
(%i2)    p: matrix([7,2],[8,2],[1,5],[3,2],[6,7])$
(%i3)    linearinterpol(p);
```
$$(\%o3)\ \left(\frac{13}{2} - \frac{3x}{2}\right)\ \text{charfun2}\,(x, -\infty, 3) + 2\,\text{charfun2}\,(x, 7, \infty) +$$

$$(37 - 5\,x)\ \text{charfun2}\,(x, 6, 7) + \left(\frac{5x}{3} - 3\right)\ \text{charfun2}\,(x, 3, 6)$$

```
(% i4)   f(x):="%;
```
$$(\%o4)\ f(x) := \left(\frac{13}{2} - \frac{3x}{2}\right)\ \text{charfun2}\,(x, -\infty, 3) + 2\,\text{charfun2}\,(x, 7, \infty) + (37 - 5\,x)\ \text{charfun2}\,(x, 6, 7) + \left(\frac{5x}{3} - 3\right)\ \text{charfun2}\,(x, 3, 6)$$

```
(% i5)   map(f, [7.3,25/7, % pi]);
```
$$(\%o5)\ \left[2, \frac{62}{21}, \frac{5\,\pi}{3} - 3\right]$$

4.3.3.2 Інтерполяція поліномами Лагранжа

Інтерполяція поліномами Лагранжа виконується за допомогою функції $lagrange$.

Синтаксис виклику: $lagrange(points)$ або $lagrange(points, option)$.

Сенс параметрів $points$ і $options$ аналогічний зазначеному вище.

Приклад використання інтерполяції поліномами Лагранжа:

```
(%i1)    load("interpol")$
(%i2)    p:[[7,2],[8,2],[1,5],[3,2],[6,7]]$
(%i3)    lagrange(p);
```

$$(\%o3)\ \frac{(x-7)(x-6)(x-3)(x-1)}{35}-\frac{(x-8)(x-6)(x-3)(x-1)}{12}+\frac{7(x-8)(x-7)(x-3)(x-1)}{30}-\frac{(x-8)(x-7)(x-6)(x-1)}{60}+\frac{(x-8)(x-7)(x-6)(x-3)}{84}$$

```
(% i4)   f(x):="%;
```

$$(\%o4)\ \mathrm{f}(x):=\frac{(x-7)(x-6)(x-3)(x-1)}{35}-\frac{(x-8)(x-6)(x-3)(x-1)}{12}+\frac{7(x-8)(x-7)(x-3)(x-1)}{30}-\frac{(x-8)(x-7)(x-6)(x-1)}{60}+\frac{(x-8)(x-7)(x-6)(x-3)}{84}$$

```
(% i5)   map(f, [2.3,5/7, % pi]);
```

$$(\%o5)\left[-1.567535,\frac{919062}{84035},\frac{(\pi-7)(\pi-6)(\pi-3)(\pi-1)}{35}-\frac{(\pi-8)(\pi-6)(\pi-3)(\pi-1)}{12}+\frac{7(\pi-8)(\pi-7)(\pi-3)(\pi-1)}{30}-\frac{(\pi-8)(\pi-7)(\pi-6)(\pi-1)}{60}+\frac{(\pi-8)(\pi-7)(\pi-6)(\pi-3)}{84}\right]$$

```
(%i6)    %,numer;
```

$$(\%o6)\ [-1.567535, 10.9366573451538, 2.893196551256924]$$

4.3.3.3 Інтерполяція сплайнами

Інтерполяція кубічними сплайнами виконується за допомогою функції $cspline$.

Синтаксис виклику: $cspline(points)$ або $cspline(points, option))$.

Сенс параметрів $points$ і $options$ аналогічний зазначеному вище.

Приклад використання інтерполяції кубічними сплайнами:

```
(%i1)    load("interpol")$
(%i2)    p:[[7,2],[8,2],[1,5],[3,2],[6,7]]$
(%i3)    cspline(p);
```

$$(\%o3)\ \left(\frac{1159x^3}{3288}-\frac{1159x^2}{1096}-\frac{6091x}{3288}+\frac{8283}{1096}\right)\ charfun2(x,-\infty,3)+\left(-\frac{2587x^3}{1644}+\frac{5174x^2}{137}-\frac{494117x}{1644}+\frac{108928}{137}\right)\ charfun2(x,7,\infty)+\left(\frac{4715x^3}{1644}-\frac{15209x^2}{274}+\frac{579277x}{1644}-\frac{199575}{274}\right)\ charfun2(x,6,7)+\left(-\frac{3287x^3}{4932}+\frac{2223x^2}{274}-\frac{48275x}{1644}+\frac{9609}{274}\right)\ charfun2(x,3,6)$$

```
(% i4)   f(x):="%;
```

$$\%o4)\ f(x) := \left(\frac{1159\,x^3}{3288} - \frac{1159\,x^2}{1096} - \frac{6091\,x}{3288} + \frac{8283}{1096}\right)\ charfun2(x, -\infty, 3) +$$

$$\left(-\frac{2587\,x^3}{1644} + \frac{5174\,x^2}{137} - \frac{494117\,x}{1644} + \frac{108928}{137}\right)\ charfun2(x, 7, \infty) +$$

$$\left(\frac{4715\,x^3}{1644} - \frac{15209\,x^2}{274} + \frac{579277\,x}{1644} - \frac{199575}{274}\right)\ charfun2(x, 6, 7) +$$

$$\left(-\frac{3287\,x^3}{4932} + \frac{2223\,x^2}{274} - \frac{48275\,x}{1644} + \frac{9609}{274}\right)\ charfun2(x, 3, 6)$$

```
(% i5)   map(f, [2.3,5/7, % pi]);
```

$$(\%o5)$$

$$[1.991460766423356, \frac{273638}{46991}, -\frac{3287\,\pi^3}{4932} + \frac{2223\,\pi^2}{274} - \frac{48275\,\pi}{1644} + \frac{9609}{274}]$$

```
(%i6)    %,numer;
```

$$(\%o6)$$

$$[1.991460766423356, 5.823200187269903, 2.227405312429507]$$

4.3.4 Оптимізація з використанням пакету lbfgs

Основна функція пакета ($lbfgs(FOM, X, X0, epsilon, iprint)$) дозволяє знайти наближене рішення задачі мінімізації без обмежень цільової функції, що визначається виразом FOM, за списком змінних X з початковим наближенням $X0$. Критерій закінчення пошуку визначається градієнтом норми цільової функції (градієнт норми $FOM < epsilonmax(1, normX)$).

Ця функція використовує квази ньютоновский алгоритм з обмеженою пам'яттю (алгоритм BFGS). Цей метод називають методом з обмеженим використанням пам'яті, тому що замість повного обігу матриці Гессе (гесіана) використовується наближення з низьким рангом. Кожна ітерація алгоритму — лінійний (одномірний) пошук, тобто пошук вздовж променя у просторі змінних X з напрямом пошуку, обчисленим з урахуванням наближеного звернення матриці Гессе. В результаті успішного лінійного пошуку значення цільової функції (FOM) зменшується. Зазвичай (але не завжди) норма градієнта FOM також зменшується.

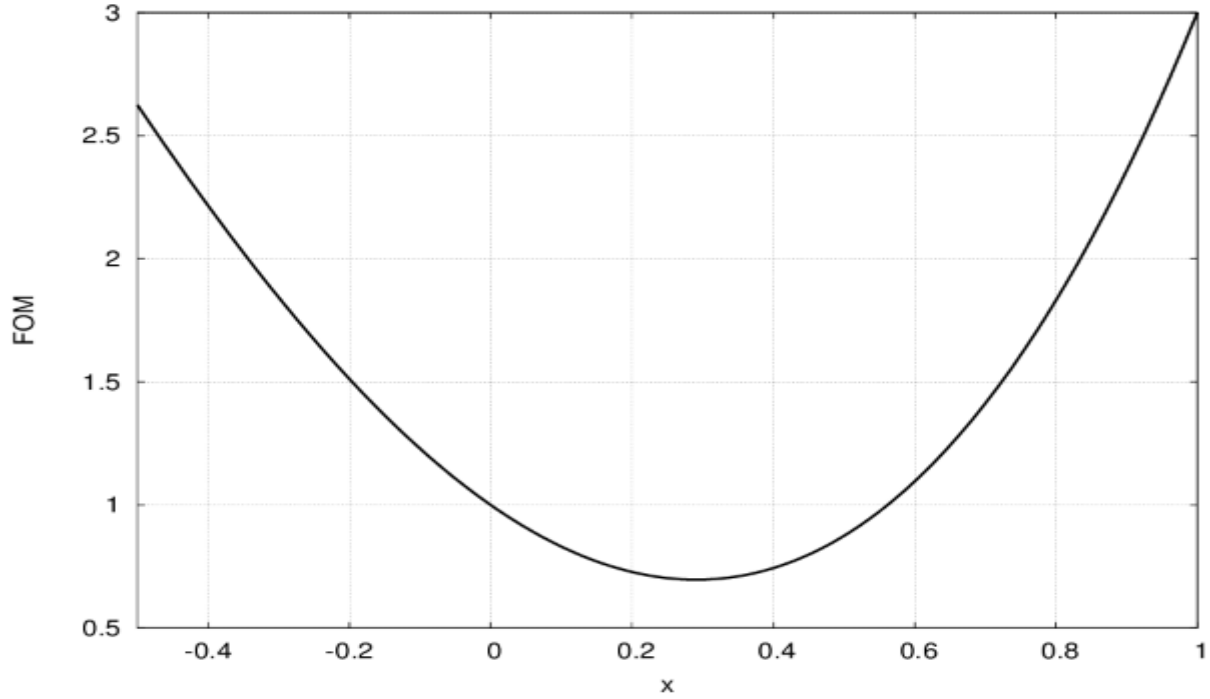
Параметр функції $iprint$ дозволяє контролювати виведення повідомлень про прогрес пошуку. Величина $iprint[1]$ керує частотою виведення ($iprint[1] < 0$ - Повідомлення не виводяться; $iprint[1] = 0$ - Повідомлення на перших та останніх ітераціях; $iprint[1] > 0$ - Виведення повідомлень на кожній $iprint[1]$ ітерації). Величина $iprint[2]$ управляє обсягом виведеної інформації (якщо $iprint[2] = 0$, виводиться лічильник ітерацій, кількість обчислень цільової функції, її величину, величину норми градієнта FOM та

довжини кроку). Збільшення $iprint[2]$ (ціла змінна, що приймає значення 0,1,2,3) спричиняє збільшення кількості виведеної інформації.

Позначення колонок інформації, що виводиться:

- I - число ітерацій, що збільшується після кожного лінійного пошуку;
- NFN - кількість обчислень цільової функції;
- FUNC - значення цільової функції наприкінці лінійного пошуку;
- GNORM - норма градієнта цільової функції наприкінці чергового лінійного пошуку;
- STEPLENGTH – довжина кроку (внутрішній параметр алгоритму пошуку).

Функція $lbfgs$ реалізована розробниками Lisp шляхом перекодування класичного алгоритму, спочатку написаного на Фортрані, тому зберегла деякі архаїчні риси. Однак алгоритм, що використовується, має високу надійність і хорошу швидкодію.



Мал. 4.1.Графік досліджуваної функції на околиці мінімуму

Розглянемо приклади використання $lbfgs$.

Найпростіший приклад - мінімізація функції однієї змінної. Необхідно знайти локальний мінімум функції $f(x) = x^3 + 3x^2 - 2x + 1$. Результати розрахунків:

```
(%i1) load (lbfgs);
```

$$(\%o1)\ /usr/share/maxima/5.13.0/share/lbfgs/lbfgs.mac$$

```
(%i2) FOM:x^3+3*x^2-2*x+1;
```

$$(\%o2)\qquad x^3+3\,x^2-2\,x+1$$

```
(%i3) lbfgs(FOM,[x],[1.1], 1e-4, [1, 0]);
```

```
************************************************
  N= 1 NUMBER OF CORRECTIONS=25
       INITIAL VALUES
F= 3.761000000000001D+00 GNORM= 8.230000000000001D+00
************************************************
I NFN FUNC GNORM STEPLENGTH
1 2 8.309999999999997D-01 1.370000000000000D+00
1.215066828675577D-01
2 3 7.056026396574796D-01 3.670279947916664D-01
1.000000000000000D+00
3 4 6.967452517789576D-01 3.053950958095847D-02
1.000000000000000D+00
4 5 6.966851926112383D-01 5.802032710369720D-04
1.000000000000000D+00
5 6 6.966851708806983D-01 8.833119583551152D-07
1.000000000000000D+00
THE MINIMIZATION TERMINATED WITHOUT DETECTING ERRORS.
IFLAG = 0
```

$$(\%o3) \qquad [x = 0.29099433470072]$$

Розглянемо результати мінімізації функції кількох змінних за допомогою $lbfgs$:

```
(%i1)    load (lbfgs)$
(%i2)    FOM:2*x*y+8*y*z+12*x*z+1e6/(x*y*z);
```

$$(\%o2) \qquad 8\,y\,z + 12\,x\,z + \frac{1000000.0}{x\,y\,z} + 2\,x\,y$$

```
(%i3)    lbfgs(FOM,[x,y,z],[1,1,1],1e-4,[-1,0]);
```

$$\%o3)\ [x = 13.47613086835734, y = 20.21398622934409,$$
$$z = 3.369022781547174]$$

4.3.4.1 Оптимізація з обмеженнями методом невизначених множників Лагранжа

Для вирішення задач мінімізації з обмеженнями у складі Maxima передбачено пакет augmented_lagrangian_method, який реалізує метод невизначених множників Лагранжа.

Синтаксис виклику функції:

- $augmented_lagrangian_method(FOM, xx, C, yy);$
- $augmented_lagrangian_method(FOM, xx, C, yy, optional_args)$
  .

Ця функція повертає наближене рішення задачі мінімізації функції кількох змінних з обмеженнями, представленими у вигляді рівностей. Цільова функція задається виразом FOM, змінні, що варіюються, — списком xx, їх початкові значення - списком yy, обмеження - списком C (передбачається, що обмеження

прирівнюються до 0). Змінні $optional_a rgs$ задаються у формі символ = значення.

Розпізнаються такі символи:

- $niter$ - Число ітерацій методу невизначених множників Лагранжа;
- $lbfgs_t olerance$ - Точність пошуку LBFGS;
- $iprint$ - Той самий параметр, що і для $lbfgs$;
- $\%lambda$ - Початкове значення невизначеного множника для методу Лагранжа.

Для використання функції $augmented_l agrangian_m ethod$ необхідно завантажити її командою $load(augmented_l agrangian)$.

Дана реалізація методу невизначених множників Лагранжа виходить з використання квази ньютоновского методу LBFGS.

4.3.5 Чисельне інтегрування: пакет romberg

Для обчислення певних інтегралів чисельними методами Maxima є проста у використанні і досить потужна функція $romberg$ (Перед використанням її необхідно завантажити).

Синтаксис виклику:

- $romberg(expr, x, a, b)$
- $romberg(F, a, b)$

Функція $romberg$ обчислює певні інтеграли методом Ромберга. У формі $romberg(expr, x, a, b)$ повертає оцінку повного інтегралу виразу $expr$ по змінній x в межах від a до b . Вираз $expr$ має повертати дійсне значення (число з плаваючою комою).

У формі $romberg(F, a, b)$ функція повертає оцінку інтегралу функції $F(x)$ по змінній x в межах від a до b (x є неназваним, єдиним аргументом F ; фактичний аргумент може бути відмінний від x). Функція F має бути функцією Maxima або Lisp, яка повертає значення з плаваючою комою.

Точністю обчислень під час виконання $romberg$ керують глобальні змінні $rombergabs$, і $rombergtol$. Функція $romberg$ закінчується успішно, коли абсолютна відмінність між послідовними наближеннями менша ніж $rombergabs$, або відносна відмінність у послідовних наближеннях - менше ніж $rombergtol$. Таким чином, коли ромбергаб дорівнює 0.0 (це значення за замовчуванням), тільки величина відносної помилки впливає на виконання функції $romberg$.

Функція $romberg$ зменшує крок інтегрування вдвічі щонайменше $rombergit$ раз, тому максимальна кількість обчислень підінтегральної

функції становить $2^{rombergit}$. Якщо критерій точності інтегрування встановлено $rombergabs$ і $rombergtol$, Не задоволений, $romberg$ друкує повідомлення про помилку. Функція $romberg$ завжди робить принаймні $rombergmin$ ітерації; це – евристичне правило, призначене, щоб запобігти передчасному завершенню виконання функції, коли підінтегральний вираз є коливальним.

Обчислення за допомогою $romberg$ багатовимірних інтегралів можливо, але закладений розробниками спосіб оцінки точності призводить до того, що методи, розроблені спеціально для багатовимірних завдань, можуть призвести до тієї ж точності з істотно меншою кількістю оцінок функції.

Розглянемо приклади обчислення інтегралів із використанням $romberg$:

```
(%i1)    load (romberg);
```

$$(\%o1)\ /usr/share/maxima/5.13.0/share/numeric/romberg.lisp$$

```
(%i2)    g(x, y):= x * y / (x + y);
```

$$(\%o2)\qquad g(x,y) := \frac{x\,y}{x+y}$$

```
(%i3)    estimate : romberg (romberg (g(x, y), y, 0,
x/2), x, 1, 3);
```

$$(\%o3)\qquad 0.81930228643245$$

```
(% i4)   assume (x > 0);
```

$$(\%o4)\qquad [x > 0]$$

```
(% i5)   integrate (integrate (g(x, y), y, 0, x/2), x, 1,
3);
```

$$(\%o5)\qquad -9\,log\left(\frac{9}{2}\right) + 9\,log\,(3) + \frac{2\,log\left(\frac{3}{2}\right) - 1}{6} + \frac{9}{2}$$

```
(%i6)    float(%);
```

$$(\%o6)\qquad 0.81930239639591$$

Як очевидно з отриманих результатів обчислення подвійного інтеграла, точне і наближене рішення збігаються до 7 включно.