Лекция 4. Численные методы и программирование с Maxima

4.1 Программирование на встроенном макроязыке

4.1.1 Условные операторы

Основная форма условного оператора: if $cond_1$ then $expr_1$ else $expr_0$. Если условие $cond_1$ истинно, то выполняется выражение $expr_1$, иначе — выполняется выражение $expr_2$. Пакет **Maxima** позволяет использовать различные формы оператора if, например: if $cond_1$ then $expr_1$ elseif $cond_2$ then $expr_2$ elseif ...else $expr_0$

Если выполняется условие $cond_1$, то выполняется выражение $expr_1$, иначе — проверяется условие $cond_2$, и если оно истинно — выполняется выражение $expr_2$, и т.д. Если ни одно из условий не является истинным — выполняется выражение $expr_0$.

Альтернативные выражения $expr_1, expr_r2, ..., expr_k$ — произвольные выражения **Maxima** (в т.ч. вложенные операторы if). Условия — действительно или потенциально логические выражения, сводимые к значениям $true$ или $false$. Способ интерпретации условий зависит от значения флага $prederror$. Если $prederror = true$, выдаётся ошибка, если значения какого-либо из выражений $cond_1, ..., cond_n$ отличается от $true$ или $false$. Если $prederror = false$ и значения какого-либо из выражений $cond_1, ..., cond_n$ отличается от $true$ или $false$, результат вычисления if — условное выражение.

4.1.2 Операторы цикла

Для выполнения итераций используется оператор texttt{do. Могут использоваться три варианта его вызова, отличающиеся условием окончания цикла:

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

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

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

Здесь $variable$ — переменная цикла; $init_value$ — начальное значение; $increment$ — шаг (по умолчанию равен 1); $limit$ — конечное значение переменной цикла; $body$ — операторы тела цикла.

Ключевые слова thru, while, unless указывают на способ завершения цикла:

- по достижении переменной цикла значения $limit$;
- пока выполняется условие $condition$;
- пока не будет достигнуто условие $condition$.

Параметры $init_value, increment, limit$, и $body$ могут быть произвольными выражениями. Контрольная переменная по завершении цикла предполагается положительной (при этом начальное значение может быть и отрицательным). Выражения $limit, increment$, условия завершения ($condition$) вычисляются на каждом шаге цикла, поэтому их сложность влияет на время выполнения цикла.

При нормальном завершении цикла возвращаемая величина — атом $done$. Принудительный выход из цикла осуществляется при помощи оператора return, который может возвращать произвольное значение.

Контрольная переменная цикла — локальная внутри цикла, поэтому её изменение в цикле не влияет на контекст (даже при наличии вне цикла переменной с тем же именем).

Примеры:

```
(%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)$
```

$$count = 2$$
$$count = 6$$
$$count = 18$$

Условия инициализации и завершения цикла можно опускать. **Пример** (цикл без явного указания переменной цикла):

```
(%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));
```

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

За 20 итераций достигается точное значение $\sqrt{5}$.

Несколько более изощрённый пример — реализация метода Ньютона для уравнения с одной неизвестной (вычисляется та же величина — корень из пяти):

```
(%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;
```

$$(\%o2) \qquad f(x) := x^2 - 5$$

```
(%i3)   float(newton(f,1000));
```

$$(\%o3) \qquad 2.236068027062195$$

Ещё одна форма оператора цикла характеризуется выбором значений переменной цикла из заданного списка. Синтаксис вызова: for $variable$ in $listend_testsests$ do $body$

Проверка условия завершения $end_testsests$ до исчерпания списка $list$ может отсутствовать.

Пример:

```
(%i1)   a:[];
```

$$(\%o1) \qquad [\,]$$

```
(%i2)   for f in [1,4,9,16] do a:cons(sqrt(f),a)$
(%i3)   a;
```

$$(\%o3) \qquad [4, 3, 2, 1]$$

4.1.3 Блоки

Как в условных выражениях, так и в циклах вместо простых операторов можно писать составные операторы, т.е. блоки. Стандартный блок имеет вид: block([r,s,t],r:1,s:r+1,t:s+1,x:t,t*t); Сначала идет список локальных переменных блока (глобальные переменные с теми же именами никак не связаны с этими локальными переменными). Список локальных переменных может быть пустым. Далее идет набор операторов. Упрощенный блок имеет вид: (x:1,x:x+2,a:x); Обычно в циклах и в условных выражениях применяют именно эту форму блока. Значением блока является значение последнего из его операторов. Внутри данного блока допускаются оператор перехода на метку и оператор return. Оператор return прекращает выполнение текущего блока и возвращает в качестве значения блока свой аргумент block([],x:2,x:x*x, return(x), x:x*x);

В отсутствие оператора перехода на метку, операторы в блоке выполняются последовательно. (В данном случае слово "метка" означает отнюдь не метку типа "%i5" или "%o7"). Оператор go выполняет переход на метку, расположенную в этом же блоке:

```
(%i1)   block([a],a:1,metka, a:a+1,
        if a=1001 then return(-a),go(metka));
```

$$(\%o1) - 1001$$

В этом блоке реализован цикл, который завершается по достижении "переменной цикла" значения 1001. Меткой может быть произвольный идентификатор.

Следует иметь в виду, что цикл сам по себе является блоком, так что (в отличие от языка **C**) прервать выполнение циклов (особенно вложенных циклов) с помощью оператора go невозможно, т.к. оператор go и метка окажутся в разных блоках. То же самое относится к оператору return. Если цикл, расположенный внутри блока, содержит оператор return, то при исполнении оператора return произойдет выход из цикла, но не выход из блока:

```
(%i1)   block([],x:for i:1 thru 15 do
        if i=2 then return(555),display(x),777);
```

$$x = 555$$
$$(\%o1) \qquad 777$$

```
(%i2)   block([],x:for i:1 thru 15 do
        if i=52 then return(555),display(x),777);
```

$$x = done$$

```
(%o2)                                777
```

Если необходимо выйти из нескольких вложенных блоков сразу (или нескольких блоков и циклов сразу) и при этом возвратить некоторое значение, то следует применять блок catch

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

В данном блоке выполнение цикла завершается, как только значение i достигает 2. Возвращаемое блоком catch значение равно 555.

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

В данном блоке выполнение цикл выполняется полностью, и возвращаемое блоком catch значение равно 777 (условия выхода из цикла при помощи throw не достигаются).

Оператор throw — аналог оператора return, но он обрывает не текущий блок, а все вложенные блоки вплоть до первого встретившегося блока catch.

Наконец, блок errcatch позволяет перехватывать некоторые (к сожалению, не все!) из ошибок, которые в нормальной ситуации привели бы к завершению счета.

Пример:

```
(%i1)   errcatch(a:1, b:0, log(a/b), c:7);
expt:   undefined: 0 to a negative exponent.
(%o1)                                [ ]
(%i2)   c;
(%o2)                                 c
```

Выполнение последовательности операций прерывается на первой операции, приводящей к ошибке. Остальные выражения блока не выполняются (значение с остаётся неопределённым). Сообщение об возникшей ошибке может быть выведено функцией $errormsg()$.

4.1.4 Функции

Наряду с простейшим способом задания функции, **Maxima** допускает создание функции в виде последовательности операторов: f(x) :=$(expr_1, expr_2, ..., expr_n)$; Значение, возвращаемое функцией — значение последнего выражения $expr_n$.

Чтобы использовать оператор return и изменять возвращаемое значение в зависимости от логики работы функции, следует применять конструкцию block, например: f(x)=block ([], $expr_{1, ...,}$ if (a > 10) then return(a), ..., $expr_n$).

При $a > 10$ выполняется оператор return и функция возвращает значение a , в противном случае — значение выражения $expr_n$.

Формальные параметры функции или блока — локальные, и являются видимыми только внутри них. Кроме того, при задании функции можно объявить локальные переменные (в квадратных скобках в начале объявления функции или бока).

Пример:

block ([a: a], $expr_{1, ...,}$ a: a+3, ..., $expr_n$)

В данном случае при объявлении блока в локальной переменной a сохраняется значение глобальной переменной a , определённой извне блока.

Пример:

```
(%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
```

В данном примере значение переменной a задаётся вне тела функции, но результат, возвращаемый ею, зависит от значения a .

Начальные значения локальных переменных функции могут задаваться двумя способами:

- Задание функции f(x) := ($expr_{1, ...,}$ $expr_n$);, вызов функции f(1); — начальное значение локальной переменной x равно 1.
- Задание блока **block ([x: 1]**, $expr_{1, ...,}$ $expr_n$), при этом начальное значение локальной переменной x также равно 1.

Наряду с именованными функциями, **Maxima** позволяет использовать и безымянные функции (лямбда-функции). Синтаксис использования лямбда-выражений (правда, при использовании с лямбдавыражениями всё-таки ассоциируется имя — см. пример):

$f1 : lambda([x_1, \ldots, x_m], expr_1, \ldots, expr_n)$
$f2 : lambda([[L]], expr_1, \ldots, expr_n)$
$f3 : lambda([x_1, \ldots, x_m, [L]], expr_1, \ldots, expr_n)$

Пример:

```
(%i1)   f: lambda ([x], x^2);
(%o1)                        lambda ([x], x^2)
(%i2)   f(a);
(%o2)                                a^2
```

Более сложный пример (лямбда-выражения могут использоваться в контексте, когда ожидается имя функции):

```
(%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]
```

Аргументы лямбда-выражений — локальные переменные. Другие переменные при вычислении лямбда-выражений рассматриваются как глобальные. Исключения отмечаются специальным символом — прямыми кавычками (см. лямбда-функцию $g2$ в примере).

```
(%i6)   a: %pi$ b: %e$ g: lambda ([a], a*b);
(%o8)                        lambda ([a], a b)
(%i9)   b: %gamma$ g(1/2);
(%o10)                               %gamma/2
(%i11)  g2: lambda ([a], a*'b);
(%o11)                       lambda ([a], a 'b)
(%i12)  b: %e$ g2(1/2);
(%o13)                               %gamma/2
```

Лямбда-функции могут быть вложенными. При этом локальные переменные внешнего выражения доступны как глобальные для внутреннего (одинаковые имена переменных маскируются).

Пример:

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

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

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

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

Подобно обычным функциям, лямбда-функции могут иметь список параметров переменной длины.

Пример:

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

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

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

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

Список [cc] при вызове лямбда-функции f включает три элемента: [a, b, c]. Формула для расчёта f применяется к каждому элементу списка.

Локальные переменные могут быть объявлены и посредством функции $local$ (переменные $v_1, v_2, ..., v_n$ объявляются локальными вызовом $local(v_1, v_2, ..., v_n)$ независимо от контекста).

4.1.5 Транслятор и компилятор в Maxima

Определив ту или иную функцию, можно заметно ускорить ее выполнение, если ее оттранслировать или откомпилировать. Это происходит потому, что если Вы не оттранслировали и не откомпилировали определенную Вами функцию, то при каждом очередном ее вызове **Maxima** каждый раз заново выполняет те действия, которые входят в определение функции, т.е. фактически разбирает соответствующее выражение на уровне синтаксиса **Maxima.**

4.1.5.1 Функция translate

Функция $translate$ транслирует функцию **Maxima** на язык **Lisp.** Например, выражение: $f(x) := 1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7$ транслируется командой: translate(f);. После этого функция, как правило, начинает вычисляться быстрее.

Пример, иллюстрирующий выигрыш по времени после трансляции функции:

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

Функция $f(n)$, организованная в виде блока, позволяет вычислить сумму $\sum_{k=1}^{k=n} k^2$.

Для выполнения тестов использовался один и тот же ноутбук (ОС Linux, Maxima 5.24). При непосредственном обращении к функции f время вычисления f (1000000) составило 7,86 с, после трансляции — 3,19 с. Для оценки времени вычисления использована функция $time$.

```
(%i2)   f(1000000);
(%o2)   33333833333500000
(%i3)   time(%o2);
(%o3)   [7.86]
(%i4)   translate(f);
(%o4)   [f]
(%i5)   f(1000000);
(%o5)   33333833333500000
(%i6)   time(%o5);
(%o6)   [3.19]
```

Функция $time(\%o1, \%o2, ...)$ возвращает список периодов времени в секундах, израсходованных для вычисления результатов $\%o1, \%o2, \cdots$ Аргументом функции $time$ могут быть только номера строк вывода, для любых других переменных функция возвращает значение $unknown$.

4.1.5.2 Функция compile

Функция $compile$ сначала транслирует функцию **Maxima** на язык **Lisp**, а затем компилирует эту функцию $Lisp$ до двоичных кодов и загружает их в память.

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]
```

После этого функция (как правило) начинает считаться еще быстрее, чем после трансляции. Например, после компиляции функции f из последнего примера время вычисления f (1000000) составило 2.17 с.

Следует иметь в виду, что как при трансляции, так и при компиляции Maxima старается оптимизировать функцию по скорости. Однако Maxima работает преимущественно с целыми числами произвольной длины либо текстовыми выражениями. Поэтому при работе с большими по объёму функциями могут возникнуть проблемы, связанные с преобразованием типов данных. В этом случае следует отказаться от трансляции или компиляции, либо переписать функцию, упорядочив использование типов.

Пример: Рассмотрим две функции, вычисляющие одно и то же выражение. В функции $f2$ явно указано, что функция возвращает действительные значения (в формате с плавающей точкой)

```
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)$
```

Время выполнения функции $f1$ при запуске $f1$(5, 10000) составило 1,8 с. После компиляции время выполнения составило 1,49 с, после трансляции — 1,39 с. Попытка обратиться к откомпилированной функции $f1$ командой $f1$(5.0, 10000.0) завершилась неудачей вследствие возникающей ошибки (плавающее переполнение).

При использовании функции с декларированным типом результата (f2) время выполнения f2(5, 10000) оказалось меньше, чем $f1$ (1,65 с вместо 1,8 с). Однако время выполнения той же функции после трансляции или компиляции превышает 10 с. Следует учесть, что в данном случае результат расчёта — рациональное число. Преобразование его к форме с плавающей точкой при вычислении очередного значения суммы требует дополнительных вычислительных затрат. При обращении к $f2$ с действительными аргументами $f2$(5.0, 10000.0) время счёта составило всего 0,16 с.

Для функции, возвращающей результат, который представляется в виде числа с плавающей точкой, компиляция или трансляция может дать уменьшение времени счёта в несколько раз.

Пример: Рассмотрим функции, вычисляющую действительное выражение (в данном случае суммируются иррациональные 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)$
```

Время вычисления выражения $f3$(5, 2000) для неоткомпилированной и не оттранслированной функции составило 7,47 с., после трансляции время вычисления $f3$(5, 2000) составило 0,03 с, после компиляции — 0,02 с.

Рассмотрим ещё один пример:

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

Время вычисления выражения $f4$(5, 1000000) составило 10,89 с, время вычисления выражения $f4$(5.0, 1000000) составило 6,71 с. После трансляции $f4$ время вычисления выражения $f4$(5, 1000000) составило 9,1 с (выигрыш по времени практически отсутствует), а для f4(5.0, 1000000) — 2,49 с (выигрыш по времени за счёт выполнения вычислений с плавающей точкой примерно в 2,5 раза).

4.2 Ввод-вывод в пакете Maxima

В этом разделе рассматриваются конструкции, позволяющие осуществить обмен данными между **Maxima** и другими приложениями.

4.2.1 Ввод-вывод данных в консоли

Основная функция для считывания вводимых пользователем данных: $read(expr_1, ..., expr_n)$. Вводимые выражения $expr_1, expr_2, \cdots$ при вводе интерпретируются. Поля ввода разделяются точками с запятой или знаком $. Аргументы функции $read$ могут включать подсказку.

Пример:

```
(%i1)   a:42$
(%i2)   a:read("Значение a = ",a," введите новую величину")
;
Значение a = 42 введите новую величину (p+q)^3;
```

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

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

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

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

Аналогичная функция $readonly$ осуществляет только ввод данных (без их интерпретации).

Пример (сравнение использования функций $read$ и $readonly$):

```
(%i1)   a:7$
(%i2)   readonly("Введите выражение:");

Введите выражение: 2^a;
```

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

```
(%i3)   read("Введите выражение:");

Введите выражение: 2^a;
```

$(\%o3)\quad 128$

Вывод на экран осуществляется функцией $display$. Синтаксис её вызова: $display(expr_1, expr_2, ...)$.

Выражения из списка аргументов выводятся слева направо (сначала само выражение, а затем после знака равенства — его значение).

Аналогичная функция $disp$ (синтаксис вызова: $disp(expr + 1, expr + 2, ...)$) выводит на экран только значение выражения после его интерпретации.

Функция $grind$ осуществляет вывод в консоль **Maxima** аналогично $disp$, но в форме, удобной для ввода с клавиатуры.

```
(%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);
```

$$1$$
$$(\%o6)\ done$$

Управление консольным вводом/выводом осуществляется посредством установки флагов $display2d, display_format_internal$ и т.п.

Вывод на экран длинных выражений по частям (одна часть над другой) осуществляется функцией $dispterms$ (синтаксис вызова: $dispterms(expr)$).

Кроме того, для вывода результатов вычислений используется функция $print$. Синтаксис вызова: $print(expr_1, ..., expr_n)$. Выражения $expr_1, ..., expr_n$ интерпретируются и выводятся последовательно в строчку (в отличие от вывода, генерируемого функцией $display$). Функция $print$ возвращает значение последнего интерпретированного выражения.

Пример:

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

Пример: 1 2 5
```

$$(\%o4)\ 5$$

```
(%i5)   rez;
```

$$(\%o5)\ 5$$

```
(%i6)   display("Пример:",a,b,c);

Пример:=Пример:
```

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

4.2.2 Файловые операции ввода-вывода

4.2.2.1 Ввод-вывод текстовых данных

Сохранение текущего состояния рабочей области **Maxima** осуществляется при помощи функции $save$. Эта функция позволяет сохранить в файле отдельные объекты с указанными именами. Варианты вызова $save$ [1]Не забудьте, что имя файла должно быть строкой и заключено в прямые кавычки, например: save("foo.l", all)$, или вычислено в свое строковое значение с помощью двух одинарных кавычек: s:"foo.l"$ save(s,all)$.:

$save(filename, name_1, name_2, name_3, ...)$ — сохраняет текущие значения переменных $name_1, name_2, name_3, \cdots$ в файле $filename$. Аргументы должны быть именами переменных, функций или других объектов. Если имя не ассоциируется с какой-либо величиной в памяти, оно игнорируется. Функция save возвращает имя файла, в который сохранены заданные объекты.

$save(filename, values, functions, labels, ...)$ — сохраняет все значения переменных, функций, меток и т.п.

$save(filename, [m, n])$ — сохраняет все значения меток ввода/вывода в промежутке от m до n (m, n — целые литералы).

$save(filename, name_1 = expr_1, ...)$ — позволяет сохранить объекты **Maxima** с заменой имени $expr_1$ на имя $name_1$.

$save(filename, all)$ — сохраняет все объекты, имеющиеся в памяти.

Глобальный флаг $file_output_append$ управляет режимом записи. Если $file_output_append = true$, результаты вывода $save$ добавляются в конец файла результатов. Иначе файл результата переписывается. Вне зависимости от $file_output_append$, если файл результатов не существует, то он создаётся.

Данные, сохранённые функцией $save$, могут быть снова загружены функцией $load$ (см. ниже).

Варианты записи при помощи save могут совмещаться друг с другом (пример: $save($ $filename, aa, bb, cc = 42, functions$ [11,17]).

Загрузка предварительно сохранённого функцией $save$ файла осуществляется функцией $load(filename)$.

Аналогичный синтаксис и у функции $stringout$, которая предназначена для вывода в файл выражений **Maxima** в формате, пригодном для последующего считывания **Maxima.**

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

Функция $load(filename)$ вычисляет выражения в файле $filename$, создавая таким образом переменные, функции, и другие объекты **Maxima.** Если объект с некоторым именем уже присутствует в **Maxima**, при выполнении $load$ он будет замещён считываемым. Чтобы найти загружаемый файл, функция $load$ использует переменные $file_search, file_search_maxima$ и $file_search_lisp$ как справочники поиска. Если загружаемый файл не найден, печатается сообщение об ошибке.

Загрузка работает одинаково хорошо для кода на **Lisp** и кода на макроязыке **Maxima.** Файлы, созданные функциями $save, translate_file, compile_file$ содержат код на Lisp, а созданные при помощи функции $stringout$ содержат код **Maxima**. Все эти файлы могут с равным успехом быть обработаны функцией $load$. $Load$ использует функцию $loadfile$, чтобы загрузить файлы **Lisp** и $batchload$, чтобы загрузить файлы **Maxima.**

Load не распознаёт конструкции :lisp в файлах, содержащих код на **Maxima**, а также глобальные переменные $\_$, $\_\_$, $\%$, и $\%th$, пока не будут созданы соответствующие объекты в памяти.

Функция $loadfile(filename)$ предназначена для загрузки файлов, содержащих код на **Lisp**, созданные функциями $save, translate_file, compile_file$. Для задач конечного пользователя удобнее функция $load$.

Протокол сессии **Maxima** может записываться при помощи функции $writefile$ (он записывается в формате вывода на консоль). Для тех же целей используется функция $appendfile$ (запись в конец существующего файла). Завершение записи и закрытие файла протокола осуществляется функцией $closefile$. Синтаксис вызова: $writefile(filename), closefile(filename)$.

4.2.2.2 Ввод-вывод командных файлов

Основная функция, предназначенная для ввода и интерпретации командных файлов — функция $batch(filename)$ Функция $batch$ читает выражения **Maxima** из файла $filename$ и выполняет их. Функция $batch$ отыскивает $filename$ в списке $file_search_maxima$. имя файла $filename$ включает последовательность выражений **Maxima**, каждое из которых должно оканчиваться ; или $. Специальная переменная $\%$ и функция $\%th$ обращаются к предыдущим результатам в пределах файла. Файл может включать конструкции :lisp. Пробелы, табуляции, символы конца строки в файле игнорируются. Подходящий входной файл может быть создан редактором текста или функцией $stringout$.

Функция $batch$ считывает каждое выражение из файла $filename$, показывает ввод в консоли, вычисляет соответствующие выражения и показывает вывод также в консоли. Метки ввода назначаются входным выражениям, метки вывода — результатам вычислений, функция $batch$ интерпретирует каждое входное выражение, пока не будет достигнут конец файла. Если предполагается реакция пользователя (ввод с клавиатуры), выполнение $batch$ приостанавливается до завершения ввода. Для остановки выполнения batch-файла используется Ctrl-C.

Функция $batchload(filename)$ считывает и интерпретирует выражения из командного файла, но не выводит на консоль входных и выходных выражений. Метки ввода и вывода выражениям, встречающимся в командном файле, также не назначаются. Специальная переменная $\%$ и функция $\%th$ обращаются к предыдущим диалоговым меткам, не имея результатов в пределах файла. Кроме того, файл $filename$ не может включать конструкции :lisp.

4.3 Встроенные численные методы

4.3.1 Численные методы решения уравнений

4.3.1.1 Решение уравнений с одним неизвестным

Для решения уравнения с одним неизвестным в пакете Maxima предусмотрена функция $find_root$. Синтаксис вызова:

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

Поиск корня функции f или выражения expr относительно переменной x осуществляется в пределах $a \leqslant x \leqslant b$.

Для поиска корней используется метод деления пополам или, если исследуемая функция достаточно гладкая, метод линейной интерполяции.

4.3.2 Решение уравнений методом Ньютона: пакет newton1

Основная функция пакета newton1 предназначена для решения уравнений методом Ньютона.

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

Данная функция возвращает приближенное решение уравнения $expr = 0$ методом Ньютона, рассматривая $expr$ как функцию одной переменной x. Поиск начинается с $x = x_0$ и производится, пока не будет достигнуто условие $abs(expr) < eps$. Функция $newton$ допускает наличие неопределенных переменных в выражении $expr$, при этом выполнение условия $abs(expr) < eps$, оценивается как истинное или ложное. Таким образом, нет необходимости оценивать $expr$ только как число.

Для использования пакета необходимо загрузить его командой load(newton1).

Примеры использования функции $newton$:

```
(%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);
```

$$(\%o5)\ 1.00030487804878\,a$$

```
(%i6)   ev (x^2 - a^2, x = %);
```

$$(\%o6)\ 6.098490481853958\,10^{-4}\,a^2$$

4.3.2.1 Решение уравнений с несколькими неизвестными: пакет mnewton

Мощная функция для решения систем нелинейных уравнений методом Ньютона входит в состав пакета mnewton. Перед использованием пакет необходимо загрузить:

```
(%i1)   load("mnewton");
```

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

После загрузки пакета mnewton становятся доступными основная функция — $mnewton$ и ряд дополнительных переменных для управления ею: $newtonepsilon$ (точность поиска, величина по умолчанию $10.0^{-\frac{fpprec}{2}}$), $newtonmaxiter$ (максимальное число итераций, величина по умолчанию 50).

Синтаксис вызова: $mnewton(FuncList, VarList, GuessList)$, где $FuncList$ — список функций, образующих решаемую систему уравнений, $VarList$ — список имен переменной, и $GuessList$ — список начальных приближений.

Решение возвращается в том же самом формате, который использует функция $solve()$. Если решение не найдено, возвращается пустой список.

Пример использования функции $mnewton$:

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

$$(\%o2)\ [[x1 = 3.75683400801276 9, x2 = 2.779849592817897]]$$

```
(%i3)   mnewton([2*a^a-5],[a],[1]);
```

$$(\%o3)\ [[a = 1.70927556786144]]$$

Как видно из второго примера, функция $mnewton$ может использоваться и для решения единичных уравнений.

4.3.3 Интерполяция

Для выполнения интерполяции функций, заданных таблично, в составе **Maxima** предусмотрен пакет расширения interpol, позволяющий выполнять линейную или полиномиальную интерполяцию Пакет включает служебную функцию $charfun2(x, a, b)$, которая возвращает $true$, если число x принадлежит интервалу $[a, b)$, и $false$ в противном случае.

4.3.3.1 Линейная интерполяция

Линейная интерполяция выполняется функцией $linearinterpol$ Синтаксис вызова: $linearinterpol(points)$ или $linearinterpol(points, option)$.

Аргумент $points$ должен быть представлен в одной из следующих форм:

- матрица с двумя столбцами, например p:matrix([2,4], [5,6], [9,3]), при этом первое значение пары или первый столбец матрицы — это значения независимой переменной,
- список пар значений, например p:[[2,4], [5,6], [9,3]],
- список чисел, которые рассматриваются как ординаты интерполируемой функции, например p:[4,6,3], в этом случае абсциссы назначаются автоматически (принимают значения 1, 2, 3 и т.д.).

В качестве опции указывается имя независимой переменной, относительно которой строится интерполяционная функция.

Примеры выполнения линейной интерполяции:

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

$$(\%o3)\ \left(\frac{13}{2}-\frac{3\,x}{2}\right)\ \mathrm{charfun2}\left(x,-\infty,3\right)+2\,\mathrm{charfun2}\left(x,7,\infty\right)+$$
$$\left(37-5\,x\right)\ \mathrm{charfun2}\left(x,6,7\right)+\left(\frac{5\,x}{3}-3\right)\ \mathrm{charfun2}\left(x,3,6\right)$$

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

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

```
(%i5)   map(f,[7.3,25/7,%pi]);
```

$$(\%o5)\ [2,\frac{62}{21},\frac{5\,\pi}{3}-3]$$

4.3.3.2 Интерполяция полиномами Лагранжа

Интерполяция полиномами Лагранжа выполняется при помощи функции $lagrange$.

Синтаксис вызова: $lagrange(points)$ или $lagrange(points, option)$.

Смысл параметров $points$ и $options$ аналогичен указанному выше.

Пример использования интерполяции полиномами Лагранжа:

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

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

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

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

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

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

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

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

4.3.3.3 Интерполяция сплайнами

Интерполяция кубическими сплайнами выполняется при помощи функции $cspline$.

Синтаксис вызова: $cspline(points)$ или $cspline(points, option))$.

Смысл параметров $points$ и $options$ аналогичен указанному выше.

Пример использования интерполяции кубическими сплайнами:

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

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

```
(%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 Оптимизация с использованием пакета lbfgs

Основная функция пакета ($lbfgs(FOM, X, X0, epsilon, iprint)$) позволяет найти приближенное решение задачи минимизации без ограничений целевой функции, определяемой выражением FOM, по списку переменных X с начальным приближением $X0$. Критерий окончания поиска определяется градиентом нормы целевой функции (градиент нормы $FOM < epsilonmax(1, normX)$).

Данная функция использует квазиньютоновский алгоритм с ограниченной памятью (алгоритм BFGS). Этот метод называют методом с ограниченным использованием памяти, потому что вместо полного обращения матрицы Гессе (гессиана) используется приближение с низким рангом. Каждая итерация алгоритма — линейный (одномерный) поиск, то есть, поиск вдоль луча в пространстве переменных X с направлением поиска, вычисленным на базе приближенного обращения матрицы Гессе. В результате успешного линейного поиска значение целевой функции (FOM) уменьшается. Обычно (но не всегда) норма градиента FOM также уменьшается.

Параметр функции $iprint$ позволяет контролировать вывод сообщений о прогрессе поиска. Величина $iprint[1]$ управляет частотой вывода ($iprint[1] < 0$ — сообщения не выводятся; $iprint[1] = 0$ — сообщения на первых и последних итерациях; $iprint[1] > 0$ — вывод сообщений на каждой $iprint[1]$ итерации). Величина $iprint[2]$ управляет объёмом выводимой информации (если $iprint[2] = 0$, выводится счётчик итераций, число вычислений целевой функции, её величину, величину нормы градиента FOM и длины шага). Увеличение $iprint[2]$ (целая переменная, принимающая значения 0,1,2,3) влечёт за собой увеличение количества выводимой информации.

Обозначения колонок выводимой информации:

- I — число итераций, которое увеличивается после каждого линейного поиска;
- NFN — количество вычислений целевой функции;
- FUNC — значение целевой функции в конце линейного поиска;
- GNORM — норма градиента целевой функции в конце очередного линейного поиска;
- STEPLENGTH — длина шага (внутренний параметр алгоритма поиска).

Функция $lbfgs$ реализована разработчиками на **Lisp** путём перекодирования классического алгоритма, первоначально написанного на Фортране, поэтому сохранила некоторые архаичные черты. Однако используемый алгоритм обладает высокой надёжностью и хорошим быстродействием.

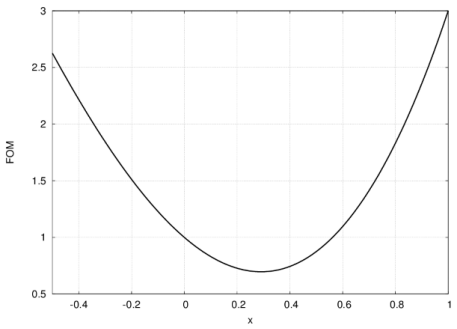

Рис. 4.1. График исследуемой функции в окрестности минимума

Рассмотрим примеры использования $lbfgs$.

Простейший пример — минимизация функции одной переменной. Необходимо найти локальный минимум функции $f(x) = x^3 + 3x^2 - 2x + 1$. Результаты расчётов:

```
(%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
```

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

Рассмотрим результаты минимизации функции нескольких переменных при помощи $lbfgs$:

```
(%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 Оптимизация с ограничениями методом неопределённых множителей Лагранжа

Для решения задач минимизации с ограничениями в составе **Maxima** предусмотрен пакет augmented_lagrangian_method, реализующий метод неопределённых множителей Лагранжа.

Синтаксис вызова функции:

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

Рассматриваемая функция возвращает приближенное решение задачи минимизации функции нескольких переменных с ограничениями, представленными в виде равенств. Целевая функция задаётся выражением FOM, варьируемые переменные — списком xx, их начальные значения — списком yy, ограничения — списком C (предполагается, что ограничения приравниваются к 0). Переменные $optional_args$ задаются в форме символ=значение.

Распознаются следующие символы:

- $niter$ — число итераций метода неопределённых множителей Лагранжа;
- $lbfgs_tolerance$ — точность поиска LBFGS;
- $iprint$ — тот же параметр, что и для $lbfgs$;
- $\%lambda$ — начальное значение неопределённого множителя для метода Лагранжа.

Для использования функции $augmented_lagrangian_method$ необходимо загрузить её командой $load(augmented_lagrangian)$.

Данная реализация метода неопределённых множителей Лагранжа базируется на использовании квазиньютоновского метода LBFGS.

4.3.5 Численное интегрирование: пакет romberg

Для вычисления определённых интегралов численными методами в **Maxima** есть простая в использовании и довольно мощная функция $romberg$ (перед использованием её необходимо загрузить).

Синтаксис вызова:

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

Функция $romberg$ вычисляет определённые интегралы методом Ромберга. В форме $romberg(expr, x, a, b)$ возвращает оценку полного интеграла выражения $expr$ по переменной x в пределах от a до b. Выражение $expr$ должно возвращать действительное значение (число с плавающей запятой).

В форме $romberg(F, a, b)$ функция возвращает оценку интеграла функции $F(x)$ по переменной x в пределах от a до b (x представляет собой неназванный, единственный аргумент F; фактический аргумент может быть отличен от x). Функция F должна быть функцией **Maxima** или **Lisp**, которая возвращает значение с плавающей запятой.

Точностью вычислений при выполнении $romberg$ управляют глобальные переменные $rombergabs$, и $rombergtol$. Функция $romberg$ заканчивается успешно, когда абсолютное различие между последовательными приближениями — меньше чем $rombergabs$, или относительное различие в последовательных приближениях — меньше чем $rombergtol$. Таким образом, когда rombergabs равна 0.0 (это значение по умолчанию), только величина относительной ошибки влияет на выполнение функции $romberg$.

Функция $romberg$ уменьшает шаг интегрирования вдвое по меньшей мере $rombergit$ раз, поэтому максимальное количество вычислений подинтегральной функции составляет $2^{rombergit}$. Если критерий точности интегрирования, установленный $rombergabs$ и $rombergtol$, не удовлетворен, $romberg$ печатает сообщение об ошибке. Функция $romberg$ всегда делает по крайней мере $rombergmin$ итерации; это — эвристическое правило, предназначенное, чтобы предотвратить преждевременное завершение выполнения функции, когда подинтегральное выражение является колебательным.

Вычисление при помощи $romberg$ многомерных интегралов возможно, но заложенный разработчиками способ оценки точности приводит к тому, что методы, разработанные специально для многомерных задач, могут привести к той же самой точности с существенно меньшим количеством оценок функции.

Рассмотрим примеры вычисления интегралов с использованием $romberg$:

```
(%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);
```

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

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

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

Как видно из полученных результатов вычисления двойного интеграла, точное и приближённое решение совпадают до 7 знака включительно.