Maxima Mínimo

Autor - Robert Dodier Tradutor - Jorge Barros de Abreu

7 de janeiro de 2006

1 O que é Maxima?

Maxima[1] é um sistema que trabalha com expressões, tais como $x+y$, $sin(a+b\pi)$, e $u \cdot v - v \cdot u$.

Maxima não está muito preocupado sobre o significado de uma expressão. Se uma expressão é representativa ou não quem decide é o usuário.

Algumas vezes você quer atribuir valores a entidades desconhecidas e avaliar a expressão. Maxima está disponível para fazer isso. Mas Maxima está também disponível para adiar atribuições de valores específicos; você pode realizar muitas manipulações de uma expressão, e somente mais adiante (ou nunca) atribuir valores a entidades desconhecidas.

Vamos ver alguns exemplos.

1. Quero calcular o volume de uma esfera.

```
(%i1) V: 4/3 * %pi * r^3;
                                      3
                                4 %pi r
(%o1)                           --------
                                   3
```

2. O raio é 10.

[1] Home page: http://maxima.sourceforge.net
Documentos: http://maxima.sourceforge.net/docs.shtml
Manual de referência: http://maxima.sourceforge.net/docs/manual/en/maxima.html

```
(%i2) r: 10;
(%o2)                          10
```

3. V é o mesmo que antes; Maxima não muda V até que eu diga a ele para fazer isso.

```
(%i3) V;
                                    3
                             4 %pi r
(%o3)                        --------
                                3
```

4. Digo ao maxima para re-avaliar V.

```
(%i4) ''V;
                             4000 %pi
(%o4)                        --------
                                3
```

5. Gostaria de ver um valor numérico em lugar de uma expressão.

```
(%i5) ''V, numer;
(%o5)                  4188.79020478639
```

2 Expressões

Tudo no Maxima é uma expressão, incluindo expressões matemáticas, objetos, e construções de programação. uma expressão é ou um átomo, ou um operador junto com seus argumentos.

Um átomo é um símbolo (um nome), uma seqüência de caracteres entre apóstrofos, ou um número (ou inteiro ou em ponto flutuante).

Todas as expressões não atômicas são representadas como $op(a_1, \ldots, a_n)$ onde op é o nome de um operador e $a_1, \ldots, a_n$ são seus argumentos. (A expressão pode ser mostrada de forma diferente, mas a representação interna é a mesma.) Os argumentos de uma expressão podem ser átomos ou expressões não atômicas.

Expressões matemáticas possuem operadores matemáticos, tais como $+ \; - \; * \; / \; < \; = \; >$ ou uma avaliação de função tal como $\mathbf{sin}(x), \mathbf{bessel\_j}(n, x)$. Em tais casos, o operador é a função.

Objetos no Maxima são expressões. Uma lista $[a_1, \ldots, a_n]$ é uma expressão $\mathbf{list}(a_1, \ldots, a_n)$. Uma matriz é uma expressão

$$\mathbf{matrix}(\mathbf{list}(a_{1,1}, \ldots, a_{1,n}), \ldots, \mathbf{list}(a_{m,1}, \ldots, a_{m,n}))$$

Construções de programação são expressões. Um bloco de código $\mathbf{block}(a_1, \ldots, a_n)$ é uma expressão com operador **block** e argumentos $a_1, \ldots, a_n$. Uma declaração condicional **if** a **then** b **elseif** c **then** d é uma expressão $\mathbf{if}(a, b, c, d)$. Um ciclo **for** a **in** L **do** S é uma expressão similar a $\mathbf{do}(a, L, S)$.

A função do Maxima **op** retorna o operador de uma expressão não atômica. A função **args** retorna os argumentos de uma expressão não atômica. A função **atom** diz se uma expressão é um átomo ou não.

Vamos ver alguns exemplos mais.

1. Átomos são símbolos, seqüências de caracteres, e números. Agrupei muitos exemplos em uma lista então podemos vê-los todos juntos.

```
(%i2) [a, foo, foo_bar, "Hello, world!", 42, 17.29];
(%o2)      [a, foo, foo_bar, Hello, world!, 42, 17.29]
```

2. Mathematical expressions.

```
(%i1) [a + b + c, a * b * c, foo = bar, a*b < c*d];
(%o1)       [c + b + a, a b c, foo = bar, a b < c d]
```

3. Listas e matrizes. Os elementos de uma lista ou matriz podem ser qualquer tipo de expressão, mesmo outra lista ou matriz.

```
(%i1) L: [a, b, c, %pi, %e, 1729, 1/(a*d - b*c)];
                                                1
(%o1)           [a, b, c, %pi, %e, 1729, ---------]
                                             a d - b c
(%i2) L2: [a, b, [c, %pi, [%e, 1729], 1/(a*d - b*c)]];
                                                 1
(%o2)         [a, b, [c, %pi, [%e, 1729], ---------]]
                                              a d - b c
(%i3) L [7];
                                 1
(%o3)                        ---------
                             a d - b c
```

```
(%i4) L2 [3];
                                                1
(%o4)            [c, %pi, [%e, 1729], ---------]
                                      a d - b c
(%i5) M: matrix ([%pi, 17], [29, %e]);
                              [ %pi  17 ]
(%o5)                         [         ]
                              [ 29   %e ]
(%i6) M2: matrix ([[%pi, 17], a*d - b*c], [matrix ([1, a], [b, 7]), %e]);
                     [ [%pi, 17]  a d - b c ]
                     [                      ]
(%o6)                [ [ 1  a ]             ]
                     [ [      ]      %e     ]
                     [ [ b  7 ]             ]
(%i7) M [2][1];
(%o7)                             29
(%i8) M2 [2][1];
                              [ 1  a ]
(%o8)                         [      ]
                              [ b  7 ]
```

4. Construção de programação são expressões. $x : y$ significa atribua y a x; o valor da expressão de atribuição é y. **block** agrupa muitas expressões, e as avalia uma vez após outra; o valor do bloco é o valor da última expressão.

```
(%i1) (a: 42) - (b: 17);
(%o1)                             25
(%i2) [a, b];
(%o2)                          [42, 17]
(%i3) block ([a], a: 42, a^2 - 1600) + block ([b], b: 5, %pi^b);
                                 5
(%o3)                         %pi  + 164
(%i4) (if a > 1 then %pi else %e) + (if b < 0 then 1/2 else 1/7);
                                      1
(%o4)                          %pi + -
                                      7
```

5. **op** retorna o operador, **args** retorna os argumentos, **atom** diz se uma expressão é um átomo ou não.

```
(%i1) op (p + q);
```

```
(%o1)                           +
(%i2) op (p + q > p*q);
(%o2)                           >
(%i3) op (sin (p + q));
(%o3)                          sin
(%i4) op (foo (p, q));
(%o4)                          foo
(%i5) op (foo (p, q) := p - q);
(%o5)                          :=
(%i6) args (p + q);
(%o6)                        [q, p]
(%i7) args (p + q > p*q);
(%o7)                    [q + p, p q]
(%i8) args (sin (p + q));
(%o8)                        [q + p]
(%i9) args (foo (p, q));
(%o9)                       [p, - q]
(%i10) args (foo (p, q) := p - q);
(%o10)                [foo(p, q), p - q]
(%i11) atom (p);
(%o11)                         true
(%i12) atom (p + q);
(%o12)                         false
(%i13) atom (sin (p + q));
(%o13)                         false
```

6. Operadores e argumentos de construções de programação. O apóstrofo simples diz ao Maxima para construir a expressão mas não fazer a avaliação para um momento posterior escolhido pelo usuário. Vamos voltar paa aquele último.

```
(%i1) op ('(block ([a], a: 42, a^2 - 1600)));
(%o1)                          block
(%i2) op ('(if p > q then p else q));
(%o2)                           if
(%i3) op ('(for x in L do print (x)));
(%o3)                          mdoin
(%i4) args ('(block ([a], a: 42, a^2 - 1600)));
                                    2
(%o4)              [[a], a : 42, a  - 1600]
(%i5) args ('(if p > q then p else q));
(%o5)                  [p > q, p, true, q]
(%i6) args ('(for x in L do print (x)));
```

```
(%o6)     [x, L, false, false, false, false, print(x)]
```

3 Avaliação

O valor de um símbolo é uma expressão associada ao símbolo. Todo símbolo tem um valor; se não for de outra forma atribuido um valor, um símbolo avalia para si mesmo. (E.g., x avalia para x se não for de outra forma atribuido um valor.)

Números e seqüências de caractere avaliam para si mesmos.

Um expressão não atômica é avaliada aproximadamente como segue.

1. Cada argumento do operadro da expressão é avaliado.
2. Se um operador está associado a uma função que pode ser chamada, a função é chamada, e o valor de retorno da função é o valor da expressão.

Avaliação é modificada de várias maneiras. Algumas modificações causam menos avaliação:

1. Algumas funções não avaliam algumas ou todos os seus argumentos, ou de outra forma modificam a avaliação de seus argumentos.
2. Um apóstrofo simples $'$ evita avaliação.
   (a) $'a$ avalia para a. Qualquer outro valor de a é ignorado.
   (b) $'f(a_1, \ldots, a_n)$ avaliam para $f(\mathbf{ev}(a_1), \ldots, \mathbf{ev}(a_n))$. É isso, os argumetos são avaliados mas f não é chamada.
   (c) $'(\ldots)$ evita avaliação de quaisquer expressões dentro de (...).

Algumas modificações causam mais avaliação:

1. Dois apóstrofos $''a$ causam uma avaliação extra na hora em que a expressão a é passada.
2. $\mathbf{ev}(a)$ causa uma avaliação extra de a toda vez que $\mathbf{ev}(a)$ for avaliado.
3. O idioma $\mathbf{apply}(f, [a_1, \ldots, a_n])$ causa a avaliação dos argumentos $a_1, \ldots, a_n$ mesmo se f comumente colocar um apoóstrofo nos argumentos $a_1, \ldots, a_n$.

4. **define** constrói uma definição de função da mesma forma que :=, nas **define** avalia o corpo da função enquanto := coloca um apóstrofo nesse mesmo corpo não avaliando-o portanto.

Vamos considerar como algumas funções são avaliadas.

1. Símbolos avaliam para simesmos se não forem de outra forma atribuidos a um valor.

```
(%i1) block (a: 1, b: 2, e: 5);
(%o1)                          5
(%i2) [a, b, c, d, e];
(%o2)                  [1, 2, c, d, 5]
```

2. Argumentos de um operador são comumente avaliados (a menos que a avaliação seja evitada de uma forma ou de outra).

```
(%i1) block (x: %pi, y: %e);
(%o1)                           %e
(%i2) sin (x + y);
(%o2)                       - sin(%e)
(%i3) x > y;
(%o3)                       %pi > %e
(%i4) x!;
(%o4)                         %pi!
```

3. Se um operador corresponde a uma função que pode ser chamada, a função é chamada (a menos que isso seja evitado por algum meio). De outra forma eavaliação retorna outra expressão com o mesmo operador.

```
(%i1) foo (p, q) := p - q;
(%o1)                 foo(p, q) := p - q
(%i2) p: %phi;
(%o2)                         %phi
(%i3) foo (p, q);
(%o3)                       %phi - q
(%i4) bar (p, q);
(%o4)                     bar(%phi, q)
```

4. Algumas funções colocam apóstrofo em seus argumentos. Exemplos: **save**, :=, **kill**.

```
(%i1) block (a: 1, b: %pi, c: x + y);
(%o1)                          y + x
(%i2) [a, b, c];
(%o2)                     [1, %pi, y + x]
(%i3) save ("tmp.save", a, b, c);
(%o3)                         tmp.save
(%i4) f (a) := a^b;
                                      b
(%o4)                        f(a) := a
(%i5) f (7);
                               %pi
(%o5)                         7
(%i6) kill (a, b, c);
(%o6)                         done
(%i7) [a, b, c];
(%o7)                        [a, b, c]
```

5. Um apóstrofo simples evita avaliação mesmo se isso puder comumente acontecer.

```
(%i1) foo (x, y) := y - x;
(%o1)                   foo(x, y) := y - x
(%i2) block (a: %e, b: 17);
(%o2)                           17
(%i3) foo (a, b);
(%o3)                         17 - %e
(%i4) foo ('a, 'b);
(%o4)                          b - a
(%i5) 'foo (a, b);
(%o5)                        foo(%e, 17)
(%i6) '(foo (a, b));
(%o6)                         foo(a, b)
```

6. Dois apóstrofos simples (apóstrofo-apóstrofo) fazem com que ocorra uma avaliação extra na hora em que a expressão for passada.

```
(%i1) diff (sin (x), x);
(%o1)                          cos(x)
(%i2) foo (x) := diff (sin (x), x);
(%o2)                foo(x) := diff(sin(x), x)
(%i3) foo (x) := ''(diff (sin (x), x));
(%o3)                    foo(x) := cos(x)
```

7. **ev** faz com que uma avaliação extra ocorra toda vez que isso for avaliado. Contrasta com o efeito de apóstrofo-apóstrofo.

```
(%i1) block (xx: yy, yy: zz);
(%o1)                          zz
(%i2) [xx, yy];
(%o2)                        [yy, zz]
(%i3) foo (x) := ''x;
(%o3)                       foo(x) := x
(%i4) foo (xx);
(%o4)                          yy
(%i5) bar (x) := ev (x);
(%o5)                    bar(x) := ev(x)
(%i6) bar (xx);
(%o6)                          zz
```

8. **apply** faz com que ocorra a avaliação do argumento mesmo mesmo se eles estiverem comumente com apóstrofo.

```
(%i1) block (a: aa, b: bb, c: cc);
(%o1)                          cc
(%i2) block (aa: 11, bb: 22, cc: 33);
(%o2)                          33
(%i3) [a, b, c, aa, bb, cc];
(%o3)               [aa, bb, cc, 11, 22, 33]
(%i4) apply (kill, [a, b, c]);
(%o4)                         done
(%i5) [a, b, c, aa, bb, cc];
(%o5)               [aa, bb, cc, aa, bb, cc]
(%i6) kill (a, b, c);
(%o6)                         done
(%i7) [a, b, c, aa, bb, cc];
(%o7)                 [a, b, c, aa, bb, cc]
```

9. **define** avalia o corpo de uma definição de função.

```
(%i1) integrate (sin (a*x), x, 0, %pi);
                         1   cos(%pi a)
(%o1)                    - - ----------
                         a       a
(%i2) foo (x) := integrate (sin (a*x), x, 0, %pi);
(%o2)       foo(x) := integrate(sin(a x), x, 0, %pi)
```

```
(%i3) define (foo (x), integrate (sin (a*x), x, 0, %pi));
                                 1   cos(%pi a)
(%o3)                foo(x) := - - ----------
                               a       a
```

4 Simplificação

Após avaliar uma expressão, Maxima tenta encontrar uma expressão equivalente que é "mais simples." Maxima aplica muitas regras que abrange noções convencionais de simplicidade. Por exemplo, $1 + 1$ simplifica para 2, $x + x$ simplifica para $2x$, and **sin**(%**pi**) simplifica para 0.

Todavia, muitas bem conhecidas identidades não são aplicadas automaticamente. Por exemplo, fórmulas de arco duplo para funções trigonométricas, ou rearranjos de razões tais como $a/b + c/b \rightarrow (a + c)/b$. Existem muitas funções que podem aplicar identidades.

Simplificação é sempre aplicada a menos que explicitamente evitada. Simplificação é aplicada mesmo se uma expressão não for avaliada.

tellsimpafter estabelece regras de simplificação definidas pelo usuário.

Vamos ver alguns exemplos de simplificação.

1. Apóstrofo evita avaliação mas não simplificação. Quando o sinalizador global **simp** for **false**, simplificação é evitada mas não a avaliação.

```
(%i1) '[1 + 1, x + x, x * x, sin (%pi)];
                                      2
(%o1)                     [2, 2 x, x , 0]
(%i2) simp: false$
(%i3) block ([x: 1], x + x);
(%o3)                           1 + 1
```

2. Algumas identidade não são aplicadas automaticamente. **expand**, **ratsimp**, **trigexpand**, **demoivre** são algumas funções que aplicam identidades.

```
(%i1) (a + b)^2;
                                     2
(%o1)                         (b + a)
(%i2) expand (%);
```

```
                          2            2
(%o2)                    b  + 2 a b + a
(%i3) a/b + c/b;
                             c   a
(%o3)                        - + -
                             b   b
(%i4) ratsimp (%);
                             c + a
(%o4)                        -----
                               b
(%i5) sin (2*x);
(%o5)                      sin(2 x)
(%i6) trigexpand (%);
(%o6)                  2 cos(x) sin(x)
(%i7) a * exp (b * %i);
                                 %i b
(%o7)                        a %e
(%i8) demoivre (%);
(%o8)              a (%i sin(b) + cos(b))
```

5 apply, map, e lambda

1. **apply** constrói e avalia uma expressão. Os argumentos da expressão são sempre avaliados (mesmo se eles não puderem ser avaliados de outra forma).

```
(%i1) apply (sin, [x * %pi]);
(%o1)                      sin(%pi x)
(%i2) L: [a, b, c, x, y, z];
(%o2)                  [a, b, c, x, y, z]
(%i3) apply ("+", L);
(%o3)                z + y + x + c + b + a
```

2. **map** constrói e avalia uma expressão usando ítens individuais de uma lista de argumentos. Os argumentos de uma expressão sào sempre avaliados (mesmo se eles não puderem ser avaliados de outra forma). O resultado é uma lista.

```
(%i1) map (foo, [x, y, z]);
(%o1)              [foo(x), foo(y), foo(z)]
(%i2) map ("+", [1, 2, 3], [a, b, c]);
```

```
(%o2)                  [a + 1, b + 2, c + 3]
(%i3) map (atom, [a, b, c, a + b, a + b + c]);
(%o3)           [true, true, true, false, false]
```

3. **lambda** constrói uma expressão lambda (i.e., uma função sem nome). A expressão lambda pode ser usada em alguns contextos como uma função comum que possui nome. **lambda** não avalia o corpo da função.

```
(%i1) f: lambda ([x, y], (x + y)*(x - y));
(%o1)            lambda([x, y], (x + y) (x - y))
(%i2) f (a, b);
(%o2)                      (a - b) (b + a)
(%i3) apply (f, [p, q]);
(%o3)                      (p - q) (q + p)
(%i4) map (f, [1, 2, 3], [a, b, c]);
(%o4)  [(1 - a) (a + 1), (2 - b) (b + 2), (3 - c) (c + 3)]
```

6 Tipos de objetos internos

Um objeto é representado como uma expressão. Como outra expressão, um objeto compreende um operador e seus argumentos.

Os mais importantes tipos de objets internos são listas, matrizes, e conjuntos.

6.1 Listas

1. Uma lista é indicada dessa forma: $[a, b, c]$.
2. If L é uma lista, $L[i]$ é seu i'ésimo elemento. $L[1]$ é o primeiro elemento.
3. **map**(f, L) aplica f a cada elemento de L.
4. **apply**$(" + ", L)$ é a soma dos elementos de L.
5. **for** x **in** L **do** *expr* avalia *expr* para cada elemento de L.
6. **length**(L) é o número de elementos em L.

6.2 Matrizes

1. Uma matriz é definida da seguinte forma: **matrix**$(L_1, \ldots, L_n)$ onde $L_1, \ldots, L_n$ são listas que representam as linhas da matriz.
2. Se M for uma matriz, $M[i, j]$ ou $M[i][j]$ é seu (i, j)'ésimo elemento. $M[1, 1]$ é o elemento no canto superior esquerdo.
3. O operador . representa multiplicação não comutativa. $M.L$, $L.M$, e $M.N$ são produtos não comutativos, onde L é uma lista e M e N são matrizes.
4. **transpose**(M) é a transposta de M.
5. **eigenvalues**(M) retorna o autovalor de M.
6. **eigenvectors**(M) retorna o autovetor de M.
7. **length**(M) retorna o nuúmero de linhas de M.
8. **length**(**transpose**(M)) retorna o número de colunas de M.

6.3 Conjuntos

1. Maxima entende conjuntos finitos explicitamente definidos. Conjuntos não são o mesmo qque listas; uma conversão explícita é necessária para mudar de um para outro.
2. Um conjunto é especificado dessa forma: **set**$(a, b, c, \ldots)$ onde os elementos do conjunto são $a, b, c, \ldots$.
3. **union**(A, B) é a união dos conjuntos A e B.
4. **intersection**(A, B) é a intersecção dos conjuntos A e B.
5. **cardinality**(A) é o número de elementos no conjunto A.

7 Como fazer para...

7.1 Definir uma função

1. O operador := define uma função, colocando um apóstrofo no corpo da função.

Nesse exemplo, **diff** é reavalado toda vez que a função for chamada. O argumento é substituído por x e a expressão resultante é avaliada. Quando o argumento for alguma outra coisa que não um símbolo, isso causa um erro: para **foo**(1) Maxima tenta avaliar $\mathbf{diff}(\mathbf{sin}(1)^2, 1)$.

```
(%i1) foo (x) := diff (sin(x)^2, x);
                                          2
(%o1)              foo(x) := diff(sin (x), x)
(%i2) foo (u);
(%o2)                     2 cos(u) sin(u)
(%i3) foo (1);
Non-variable 2nd argument to diff:
1
#0: foo(x=1)
 -- an error.
```

2. **define** define uma função, avaliando o corpo da função.

   Nesse exemplo, **diff** é avaliado somente uma vez (quando a função é definida). **foo**(1) is OK now.

```
(%i1) define (foo (x), diff (sin(x)^2, x));
(%o1)               foo(x) := 2 cos(x) sin(x)
(%i2) foo (u);
(%o2)                     2 cos(u) sin(u)
(%i3) foo (1);
(%o3)                     2 cos(1) sin(1)
```

7.2 Resolver uma equação

```
(%i1) eq_1: a * x + b * y + z = %pi;
(%o1)                  z + b y + a x = %pi
(%i2) eq_2: z - 5*y + x = 0;
(%o2)                    z - 5 y + x = 0
(%i3) s: solve ([eq_1, eq_2], [x, z]);
                (b + 5) y - %pi      (b + 5 a) y - %pi
(%o3)   [[x = - ---------------, z = -----------------]]
                     a - 1                 a - 1
(%i4) length (s);
(%o4)                           1
(%i5) [subst (s[1], eq_1), subst (s[1], eq_2)];
       (b + 5 a) y - %pi   a ((b + 5) y - %pi)
```

```
(%o5) [----------------- - ------------------- + b y = %pi,
            a - 1                a - 1
                  (b + 5 a) y - %pi   (b + 5) y - %pi
                  ----------------- - --------------- - 5 y = 0]
                        a - 1               a - 1
(%i6) ratsimp (%);
(%o6)                   [%pi = %pi, 0 = 0]
```

7.3 Integrar e diferenciar

integrate calcular integrais definidas e indefinidas.

```
(%i1) integrate (1/(1 + x), x, 0, 1);
(%o1)                          log(2)
(%i2) integrate (exp(-u) * sin(u), u, 0, inf);
                                1
(%o2)                           -
                                2
(%i3) assume (a > 0);
(%o3)                         [a > 0]
(%i4) integrate (1/(1 + x), x, 0, a);
(%o4)                       log(a + 1)
(%i5) integrate (exp(-a*u) * sin(a*u), u, 0, inf);
                                1
(%o5)                           ---
                                2 a
(%i6) integrate (exp (sin (t)), t, 0, %pi);
                         %pi
                        /
                        [      sin(t)
(%o6)                   I    %e       dt
                        ]
                        /
                         0
(%i7) 'integrate (exp(-u) * sin(u), u, 0, inf);
                       inf
                      /
                      [      - u
(%o7)                 I    %e    sin(u) du
                      ]
                      /
                       0
```

diff calcular derivadas.

```
(%i1) diff (sin (y*x));
(%o1)         x cos(x y) del(y) + y cos(x y) del(x)
(%i2) diff (sin (y*x), x);
(%o2)                        y cos(x y)
(%i3) diff (sin (y*x), y);
(%o3)                        x cos(x y)
(%i4) diff (sin (y*x), x, 2);
                               2
(%o4)                        - y  sin(x y)
(%i5) 'diff (sin (y*x), x, 2);
                            2
                           d
(%o5)                      --- (sin(x y))
                             2
                           dx
```

7.4 Fazer um gráfico

plot2d desenhar gráficos bidimensionais.

```
(%i1) plot2d (exp(-u) * sin(u), [u, 0, 2*%pi]);
(%o1)
(%i2) plot2d ([exp(-u), exp(-u) * sin(u)], [u, 0, 2*%pi]);
(%o2)
(%i3) xx: makelist (i/2.5, i, 1, 10);
(%o3)  [0.4, 0.8, 1.2, 1.6, 2.0, 2.4, 2.8, 3.2, 3.6, 4.0]
(%i4) yy: map (lambda ([x], exp(-x) * sin(x)), xx);
(%o4) [0.261034921143457, 0.322328869227062, .2807247779692679,
.2018104299334517, .1230600248057767, .0612766372619573,
.0203706503896865, - .0023794587414574, - .0120913057698414,
- 0.013861321214153]
(%i5) plot2d ([discrete, xx, yy]);
(%o5)
(%i6) plot2d ([discrete, xx, yy], [gnuplot_curve_styles, ["with points"]]);
(%o6)
```

Veja também **plot3d**.

7.5 Gravar e chamar um arquivo

save escreve expressões em um arquivo.

```
(%i1) a: foo - bar;
(%o1)                          foo - bar
(%i2) b: foo^2 * bar;
                                      2
(%o2)                          bar foo
(%i3) save ("minha.sessao", a, b);
(%o3)                         minha.sessao
(%i4) save ("minha.sessao", all);
(%o4)                         minha.sessao
```

load lê expressões de um arquivo.

```
(%i1) load ("minha.sessao");
(%o4)                         minha.sessao
(%i5) a;
(%o5)                          foo - bar
(%i6) b;
                                      2
(%o6)                          bar foo
```

Veja também **stringout** e **batch**.

8 Programando no Maxima

Existe um ambiente, que contém todos os símbolos do Maxima. Não existe como criar outro ambiente.

Todas as variáveis são globais a menos que pareçam em uma declaração de variáveis locais. Funções, expressões lambda, e blocos podem ter variáveis locais.

O valor de uma variável é aquele que foi atribuído mais recentemente, ou por uma atribuição explícita ou por atribuição de um valor a uma variável local em um bloco, função, ou expressão lambda. Essa política é conhecida como *escopo dinâmico*.

Se uma variável é uma variável local em uma função, expressão lambda, ou bloco, seu valor é local mas suas outras propriedades (como estabelecidas

através de **declare**) são globais. A função **local** faz uma variável local com relação a todas as propriedades.

Por padrão uma definição de função é global, mesmo se isso aparecer dentro de uma função, expressão lambda, ou bloco. **local**$(f), f(x) := \ldots$ cria uma definição de função local.

trace(*foo*) faz com que o Maxima mostre uma mensagem quando a função *foo* for iniciada e terminada.

Vamos ver alguns exemplos de programação no Maxima.

1. Todas as variáveis são globais a menos que apareçam em uma declaração de variáveis locais. Funções, expressões lambda, e blocos podem ter variáveis locais.

```
(%i1) (x: 42, y: 1729, z: foo*bar);
(%o1)                          bar foo
(%i2) f (x, y) := x*y*z;
(%o2)                    f(x, y) := x y z
(%i3) f (aa, bb);
(%o3)                       aa bar bb foo
(%i4) lambda ([x, z], (x - z)/y);
                                        x - z
(%o4)                 lambda([x, z], -----)
                                         y
(%i5) apply (%, [uu, vv]);
                                 uu - vv
(%o5)                            -------
                                  1729
(%i6) block ([y, z], y: 65536, [x, y, z]);
(%o6)                      [42, 65536, z]
```

2. O valor de uma variável é aquele que foi atribuido mais recentemente, ou por atribuição explícita ou por atribuição de um valor a uma variável local.

```
(%i1) foo (y) := x - y;
(%o1)                     foo(y) := x - y
(%i2) x: 1729;
(%o2)                           1729
(%i3) foo (%pi);
(%o3)                        1729 - %pi
```

```
(%i4) bar (x) := foo (%e);
(%o4)                    bar(x) := foo(%e)
(%i5) bar (42);
(%o5)                          42 - %e
```

9 Lisp e Maxima

A construção **:lisp** *expr* diz ao interpretador Lisp para avaliar *expr*. Essa construção é reconhecida em entradas através da linha de comando e em arquivos processados por **batch**, mas não é reconhecida por **load**.

O símbolo Maxima **foo** corresponde ao símbolo Lisp $foo, e o símbolo Lisp foo corresponde ao símbolo Maxima ?**foo**.

:lisp (**defun** $foo (a) (...)) define uma função Lisp foo que avalia seus argumentos. A partir o Maxima, a função é chamada como **foo**(a).

:lisp (**defmspec** $foo (e) (...)) define uma função Lisp **foo** que coloca uma apóstrofo em seus argumentos não avaliando-os portanto. A partir do Maxima, a função é chamada como **foo**(a). Os argumentos de $foo são (**cdr** e), e (**caar** e) é sempre $foo por si mesma.

A partir do Lisp, ta construção (**mfuncall** '$foo $a_1 \ldots a_n$) chama a função **foo** definida no Maxima.

Vamos extender ao Lisp a partir do Maxima e vice-versa.

1. A construção **:lisp** *expr* diz ao interpretador Lisp para avaliar *expr*.

   ```
   (%i1) (aa + bb)^2;
                                        2
   (%o1)                        (bb + aa)
   (%i2) :lisp $%
   ((MEXPT SIMP) ((MPLUS SIMP) $AA $BB) 2)
   ```

2. **:lisp** (**defun** $foo (a) (...)) define uma função Lisp **foo** que avalia seus argumentos.

   ```
   (%i1) :lisp (defun $foo (a b) `((mplus) ((mtimes) ,a ,b) $%pi))
   $FOO
   (%i1) (p: x + y, q: x - y);
   (%o1)                            x - y
   (%i2) foo (p, q);
   (%o2)                  (x - y) (y + x) + %pi
   ```

3. **:lisp** (**defmspec** $foo (e) (...)) define uma função Lisp **foo** que coloca um apóstrofo em seus argumentos não avaliando-os portanto.

```
(%i1) :lisp (defmspec $bar (e) (let ((a (cdr e))) `((mplus) ((mtimes) ,@a)
#<CLOSURE LAMBDA (E) (LET ((A (CDR E))) `((MPLUS) ((MTIMES) ,@A) $%PI))>
(%i1) bar (p, q);
(%o1)                         p q + %pi
(%i2) bar (''p, ''q);
(%o2)                         p q + %pi
```

4. Partinfo do Lisp, a construção (**mfuncall** '$foo $a_1 \ldots a_n$) chama a função **foo** definida no Maxima.

```
(%i1) blurf (x) := x^2;
                                          2
(%o1)                       blurf(x) := x
(%i2) :lisp (displa (mfuncall '$blurf '((mplus) $grotz $mumble)))
                 2
(mumble + grotz)
NIL
```