Métodos Numéricos para Equações Diferenciais

Fernando Deeke Sasse
Departamento de Matemática, UDESC - Joinville
2008/2

Noções Básicas de Programação em Maple

1. Revisão e Estrutura de Dados

Nota sobre Autoria

A maior parte das Seções abaixo foi obtida do curso do Dr. Francis Wright , Lecture Notes for MAS104 Computational Mathematics II), e Renato Portugal (LNCC).

Introdução

Terminologia do Maple

- ***active/inert*** : faz algo/nada
- uma expressão do tipo ***função*** é uma aplicaçãom (*mapping*) ou operação sobre argumentos :

a *função* $f(x, y)$ consiste de

$$\begin{bmatrix} \textit{mapping} & \textit{application} & \textit{arguments} \\ f & (\ ) & x,y \end{bmatrix}$$

[Cuidado, pois *mapping* também é denota *function*!]

- ***execute*** : executa um programa
- ***evaluate*** : avalia ou obtém um valor de uma *variável* ou *função*.
- ***simplificação*** : um termo técnico para uma mudança de forma, que pode ser automática:

```
> 4/6;
```

$$\frac{2}{3} \qquad (1.2.1.1)$$

oou o resultado de um comando do Maple:

```
> cos(x)^2+sin(x)^2;  simplify(%);
```

$$\cos(x)^2 + \sin(x)^2$$

$$1 \qquad (1.2.1.2)$$

Worksheet

Grupos de execução : denotado pelo colchete à esquerda. Todo o código em um grupo de execução é executado em conjunto quando a tecla *Enter* é apertada.
Novas linhas use *Shift-Enter*.
Pressione a tecla *Enter* somente quando o grupo de execução estiver completo. (O cursor pode estar em qualquer lugar dentro do grupo)

```
> x := y;    # Shift-Enter for newline
  y := 23;   # Shift-Enter for newline
  x;         # Enter to execute
Error, recursive assignment
```

$$y := 23$$

$$x \quad (1.2.2.1)$$

```
> unassign('x,y');
```

Juntar/Separar: Você pode juntar (F4) ou separar (F3) grupos de execução (menu *Edit*) e deletar *prompts*.
Terminadores: utilize ponte e vírgula (**;**) para checar a saída. Depois dois pontos (**:**) pode ser utilizado ou a saída pode ser deletada.
Reabrindo: Reexecute partes relevantes ao reabrir uma *worksheet*.
Salvar: utilize *autosave* (veja Preferences... no File menu).
Recomeçar/executar: utilize o comando **restart** para começar um diferente cálculo na mesma *worksheet* limpando atribuições (ou utilize *restart* no canto direito da barra de ferramentas). O último botão à direita do menu de contexto executa toda a *worksheet*.

Caracteres, operadores, erros comuns, etc.

Aspas (Quotes)

Sempre é utilizado aos pares em volta do objeto.

Aspas simples (Forward quotes ou apostrophes ') – uneval

Previne a avaliação uma vez:

```
> x := y:  x;
```

$$y \quad (1.3.1.1.1)$$

```
> 'x';
```

$$x \quad (1.3.1.1.2)$$

```
> ''x'';
```

$$'x' \quad (1.3.1.1.3)$$

```
> %;   # % reavalia
```

$$x \quad (1.3.1.1.4)$$

```
> %;
```
y (1.3.1.1.5)

```
> sin(Pi);
```
0 (1.3.1.1.6)

```
> 'sin'(Pi);  %;
```
$\sin(\pi)$

0 (1.3.1.1.7)

```
> 'sin(Pi)';  %;
```
$\sin(\pi)$

0 (1.3.1.1.8)

Principal uso: limpar (*unassigning)* a atribuição de variáveis:

```
> x := 'x';  y := 'y';
```
$x := x$

$y := y$ (1.3.1.1.9)

ou

```
> unassign('x,y');
```

Aspas agudas (Backward quotes ou backquotes `) – nome

Transforma qualquer coisa em um símbolo (identificador ou nome) , escondendo caracteres especiais ou propriedades:

```
> `x/y` := x/y;
```
$x/y := \frac{x}{y}$ (1.3.1.2.1)

```
> `sin(x)` := sin(x);
```
$sin(x) := \sin(x)$ (1.3.1.2.2)

```
> `+`;  # operador
```
`` `+` `` (1.3.1.2.3)

```
> `end`;  # palavra reservada
```
end (1.3.1.2.4)

```
> `sin(Pi)`;  %;  # Nada é interpretado matematicamente
```
sin(Pi)

sin(Pi) (1.3.1.2.5)

Aspas duplas (") – string

Produz *strings*:

```
> "Este é um string";
```
(1.3.1.3.1)

$$\text{"Este é um string"} \quad (1.3.1.3.1)$$

Utilizado para plotar títulos, legendas, nomes de arquivos, processamento de texto.

Strings comportam-se como constantes, não podendo ter valores atribuídos.

Brackets

Cada tipo pode ser utilizado somente para seu propósito especial

Parênteses

Agrupamento de uma subexpressão.

```
> (a+b)^2, 2*(a+b);
```

$$(a+b)^2, 2\,a+2\,b \quad (1.3.2.1.1)$$

Aplicação (ou função):

```
> f(x), sin(theta);
```

$$\frac{1}{x}, \sin(\theta) \quad (1.3.2.1.2)$$

Colchetes

Construção de listas:

```
> L := [a,b,c];
```

$$L := [a, b, c] \quad (1.3.2.2.1)$$

Seleção, indexação ou subscrição de variáveis:

```
> L[2];
```

$$b \quad (1.3.2.2.2)$$

```
> A[0];
```

$$A_0 \quad (1.3.2.2.3)$$

Qualificação ou subscrição de funções:

```
> evalf[20](Pi);
```

$$3.1415926535897932385 \quad (1.3.2.2.4)$$

```
> simplify(log[10](100));
```

$$2 \quad (1.3.2.2.5)$$

```
> D[1](f)(x,y);  convert(%, diff);
```

$$-\frac{1}{x^2}$$

$$-\frac{1}{x^2} \quad (1.3.2.2.6)$$

Chaves

Construção de conjuntos:

```
> S := {a,b,c,b,a};
```
$S := \{a, b, c\}$ **(1.3.2.3.1)**

Note que conjuntos são automaticamente simplificados de acordo com uma ordem canônica, sem duplicações:

```
> T := {c,b,a};
```
$T := \{a, b, c\}$ **(1.3.2.3.2)**

Portanto,

```
> evalb(S = T); # evalb = avaliação booleana
```
true **(1.3.2.3.3)**

Alguns operadores importantes

Ponto (.)

Construção de números de ponto flutuante (*floats*):

```
> 3.14;
```
3.14 **(1.3.3.1.1)**

Multiplicação não comutativa (deixe espaço em torno do ponto):

```
> a . b <> b . a;
```
$a.b \neq b.a$ **(1.3.3.1.2)**

a menos que um dos multiplicandos seja numérico:.

```
> 4 . a = a . 4;   # both are the same as 4*a
```
$4\,a = 4\,a$ **(1.3.3.1.3)**

Útil na álgebra matricial no novo pacote LinearAlgebra (veja *dot* em *help*)

Vírgula (,)

Construção e extensão de sequências :

```
> S := a,b;
```
$S := a, b$ **(1.3.3.2.1)**

```
> S := S,c;
```
$S := a, b, c$ **(1.3.3.2.2)**

```
> S := 0,S;
```
$S := 0, a, b, c$ **(1.3.3.2.3)**

Também útil para construir listas, conjuntos, vetores e matrizes

Dollar ($)

Construção de sequências simples:

Repetição:.

```
> x$3;
```
x, x, x **(1.3.3.3.1)**

Útil em

```
> diff(f(x), x$3);
```

$$-\frac{6}{x^4} \qquad (1.3.3.3.2)$$

Expansões:

```
> $1..5;
```

$$1, 2, 3, 4, 5 \qquad (1.3.3.3.3)$$

Use `seq` para sequências mais gerais.

Avaliação e substituição

O Maple normalmente avalia cada nova expressão que é entrada diretamente. Quando necessário a avaliação pode ser forçada aplicando-se `eval` :

```
> F := x -> x^2;
```

$$F := x \rightarrow x^2 \qquad (1.3.4.1)$$

```
>  F;
```

$$F \qquad (1.3.4.2)$$

```
> eval(F);
```

$$x \rightarrow x^2 \qquad (1.3.4.3)$$

```
> F(2);
```

$$4 \qquad (1.3.4.4)$$

`eval` é utilizado também para avaliar uma expressão em valores específicos de suas variáveis:

```
> E := a*sin(x) + b;
```

$$E := a \sin(x) + b \qquad (1.3.4.5)$$

```
> eval(E, x = Pi);
```

$$b \qquad (1.3.4.6)$$

A avaliação de uma função é similar a uma aplicação:

```
> F := x -> a*sin(x) + b;
```

$$F := x \rightarrow a \sin(x) + b \qquad (1.3.4.7)$$

```
> F(Pi);
```

$$b \qquad (1.3.4.8)$$

Note que `subs` usa uma sintaxe diferente e não reavalia: a

```
> subs(x = Pi, E);
```

$$a \sin(\pi) + b \qquad (1.3.4.9)$$

É necessário o uso de `eval` se a expressão `subs` contém qualquer função ativa:

```
> eval(subs(x = Pi, E));
```

(1.3.4.10)

$$b \qquad (1.3.4.10)$$

Não confunda **eval** com avaliadores especiais tais como o avaliador de ponto flutuante **evalf** ou o avaliador booleano **evalb**.

Erros comuns

Justaposição não é permitida

```
> 2x;
Error, missing operator or `;`
> sin x;
Error, missing operator or `;`
```

Utilize *****, **.** (or **&***) para multiplicação e **()** para argumentos de funções:

```
> 2*x, sin(x);
```

$$2\,x, \sin(x) \qquad (1.3.5.1.1)$$

Maiúsculas/minúsculas são diferentes; funções inertes

Temos **Int <> int, Pi <> pi**

```
> Int(x, x) = int(x, x);
```

$$\int x\,dx = \frac{1}{2}\,x^2 \qquad (1.3.5.2.1)$$

Funções que começam com letra maiúsculas normalmente são inertes; use **value** para ativá-las. Isso proporciona um útil idioma:

```
> Int(x, x):  % = value(%);
```

$$\int x\,dx = \frac{1}{2}\,x^2 \qquad (1.3.5.2.2)$$

Símbolos cujos nomes são similares em grego e inglês são mostrados em grego, mas isso é independente do fato do Maple associar algum significado matemático especial a eles. Por exemplo, somente **Pi** representa a constante matemática:

```
> Pi, pi, PI;  evalf(%);
```

$$\pi, \pi, \Pi$$

$$3.141592654, \pi, \Pi \qquad (1.3.5.2.3)$$

Exponenciais requerem a função **exp**

```
> exp(x) <> e^x, exp(1) <> e;
```

$$\mathrm{e}^x \neq e^x, \mathrm{e} \neq e \qquad (1.3.5.3.1)$$

As letras **e** e **E** não têm qualquer significado especial (exceto em constantes *float*)

Atribuições e igualdades

Não os confunda: atribuições (`:=`) são ativas e dão valores a variáveis; igualdades (`=`) são estruturas inertes (usadas em equações)

Manipulando Estruturas de Dados Básicas

nops, op, [], subsop, NULL

Expressões ou estruturas de dados, tais como expressões algébricas, funções, sequências, listas e conjuntos, geralmente envolvem operadores, como
`+ - * / ^ . ,`

- `nops` dá o número de operandos (ou elementos):

```
> nops(a+b+c+d);
```

$$4 \quad (1.4.1.1)$$

```
> nops(f(a,b));
```

$$2 \quad (1.4.1.2)$$

```
> nops([a,b,c]);
```

$$3 \quad (1.4.1.3)$$

- `op` fornece os operandos (ou elementos) de uma sequência:

```
> op(a+b+c+d);
```

$$a, b, c, d \quad (1.4.1.4)$$

```
> op(f(a,b));
```

$$a, -1 \quad (1.4.1.5)$$

```
> op([a,b,c]);
```

$$a, b, c \quad (1.4.1.6)$$

`op` aceita um índice numérico ou domínio de valores como primeiro argumento opcional e retorna uma subsequência de operandos:

```
> op(2, [a,b,c]);
```

$$b \quad (1.4.1.7)$$

```
> op(2..3, [a,b,c]);
```

$$b, c \quad (1.4.1.8)$$

```
> op(2,[a,{3,4},c]);
```

$$\{3, 4\} \quad (1.4.1.9)$$

O *Operando zero* é o operador ou tipo de estrutura:

```
> op(0, a+b);
```

$$`+` \quad (1.4.1.10)$$

```
> op(0, [a,b,c]);
```

$list$ (1.4.1.11)

```
> op(0,{a,{3,4},c});
```

set (1.4.1.12)

Note essas duas anomalias. Como $a - b$ é representado como $a + (-b)$, temos

```
> a-b: op(0,%),op(%);
```

$`+`, a, -b$ (1.4.1.13)

e como $\frac{a}{b}$ é representado por $a\,(b)^{-1}$,

```
> a/b: op(0,%),op(%);
```

$`*`, a, \frac{1}{b}$ (1.4.1.14)

```
> 1/b:
> op(0,%);
```

$`\^{}`$ (1.4.1.15)

```
> op(%%);
```

$b, -1$ (1.4.1.16)

`[ ]` é como `op`, mas preserva a estrutura de dados:

```
> [a,b,c][2..3];
```

$[b, c]$ (1.4.1.17)

lembrando que

```
> op(2..3, [a,b,c]);
```

b, c (1.4.1.18)

exceto quando não há índices:

```
> [a,b,c][];
```

a, b, c (1.4.1.19)

Somente `[ ]` pode ser usado diretamente com uma sequência:

```
> (a,b,c)[2];
```

b (1.4.1.20)

```
> op(2,(a,b,c));
Error, invalid input: op expects 1 or 2 arguments, but received 4
```

`subsop` combina `subs` e `op`:

```
> subsop(2=B, [a,b,c]);
```

$[a, B, c]$ (1.4.1.21)

Índices Negativos contam em ordem inversa na maioria dos casos:

```
> L := [a,b,c,d,e]:
> op(-1, L) = op(nops(L), L);
```

$$e = e \quad (1.4.1.22)$$

```
> op(-2,L) = op(nops(L)-1, L);
```

$$d = d \quad (1.4.1.23)$$

NULL representa a sequência vazia, que é útil para inicializar sequências e deletar elementos:

```
> S := NULL;
```

$$S := \quad (1.4.1.24)$$

```
>   S := S,a,b;
```

$$S := a, b \quad (1.4.1.25)$$

```
> subsop(2=NULL, [a,b,c]);
```

$$[a, c] \quad (1.4.1.26)$$

Note que **NULL** é utilizado somente para entrada e nunca aparece na saída.

Manipulando estruturas de dados

Somas, produtos, sequências, funções, funções, listas e conjuntos são estruturas de dados muito similares.
Listas, funções e conjuntos podem ser manipuladas vias suas sequências, por exemplo,

op(*lista* **)** -> *sequência*, **[** *sequência* **]** -> *lista*

Criando somas, produtos e sequências

Para se ter eficiência:

- escreva explicitamente e comente
- use **add**, **mul**, **seq**
- use um loop **do** ***[com todo o código relacionado no mesmo grupo de execução]***. Por exemplo, suponha que $x_{n+1} = f(x_n)$, $x_1 = a$ e que queremos calcular $\sum_{i=1}^{n} x_i$, $\prod_{i=1}^{n} x_i$, e a seqüência x_i, $i = 1 .. n$ para $n = 5$:

```
> restart:
> # Começa o gpo de execuçao
  # Inicialização:
  S := 0:      # soma
```

```
P := 1:      # produto
Q := NULL:   # sequência
x := a:      # primeiro elemento
n := 5:      # número de elementos
# Loop principal:
to n do
   S := S + x;   # soma
   P := P * x;   # produto
   Q := Q , x;   # sequência
   x := f(x)     # próximo elemento
end do:
# Resultados:
'S' = S;
'P' = P;
'Q' = Q;
# Fim do grupo de execução
```

$$S = a + f(a) + f(f(a)) + f(f(f(a))) + f(f(f(f(a))))$$

$$P = a\, f(a)\, f(f(a))\, f(f(f(a)))\, f(f(f(f(a))))$$

$$Q = (a, f(a), f(f(a)), f(f(f(a))), f(f(f(f(a))))) \quad (1.4.2.1.1)$$

Juntando e estendendo listas

```
> L1 := [a,b,c]:  L2 := [d,e,f]:
> [op(L1),op(L2)];  # junta
```

$$[a, b, c, d, e, f] \quad (1.4.2.2.1)$$

```
> [op(L1),d];   # estende
```

$$[a, b, c, d] \quad (1.4.2.2.2)$$

Inserindo e deletando elementos de lista

```
> L := [a,b,c,d];
```

$$L := [a, b, c, d] \quad (1.4.2.3.1)$$

```
> [op(1..2,L), m, op(3..4,L)];  # insere depois
  do segundo elemento
```

$$[a, b, m, c, d] \quad (1.4.2.3.2)$$

```
> [op(1..2,L), op(4,L)];  # deleta o terceiro
  elemento
```

$$[a, b, d] \quad (1.4.2.3.3)$$

```
> subsop(3=NULL, L);  # deleta o terceiro
```

```
elemento
```

$$[a, b, d] \quad (1.4.2.3.4)$$

Somando e multiplicando elementos de listas ou sequências; operadores como funções

A maioria dos operadores pode ser utilizada como função (mapping or function), se protegidos por aspas agudas (backward quotes):

```
> `+`(a,b,c);
```

$$a + b + c \quad (1.4.2.4.1)$$

```
> L := [a,b,c]:
> `*`(op(L));  # op dá uma sequência de
  elementos
```

$$a\, b\, c \quad (1.4.2.4.2)$$

```
> `*`(a,b,c);
```

$$a\, b\, c \quad (1.4.2.4.3)$$

Note que

```
> mul(el, el=L);
```

$$a\, b\, c \quad (1.4.2.4.4)$$

```
> add(el, el=L);
```

$$a + b + c \quad (1.4.2.4.5)$$

```
> seq(el,el=L);
```

$$a, b, c \quad (1.4.2.4.6)$$

Comando `map`

```
> restart:
```

Aplica uma função a cada operando de uma expressão:

```
> map(f, [a,b,c]);
```

$$[f(a), f(b), f(c)] \quad (1.4.2.5.1)$$

```
> map(f, {a,b,c});
```

$$\{f(a), f(b), f(c)\} \quad (1.4.2.5.2)$$

```
> map(x -> x^2, x+y+z);
```

$$x^2 + y^2 + z^2 \quad (1.4.2.5.3)$$

```
> map(x -> x^2, (x+1)*y*z);
```

$$(x+1)^2\, y^2\, z^2 \quad (1.4.2.5.4)$$

```
> map(x -> x^2, {a,b,c});
```

$$\{a^2, b^2, c^2\} \quad (1.4.2.5.5)$$

```
> map(D,[x^2,x*y,z*x]);
```
$$[2\,\mathrm{D}(x)\,x,\ \mathrm{D}(x)\,y + x\,\mathrm{D}(y),\ \mathrm{D}(z)\,x + z\,\mathrm{D}(x)] \quad (1.4.2.5.6)$$

```
> f := (x,y) -> cos(x*y);
```
$$f := (x, y) \rightarrow \cos(x\,y) \quad (1.4.2.5.7)$$

```
> g:=(x,y) -> exp(x)/sin(y);
```
$$g := (x, y) \rightarrow \frac{e^x}{\sin(y)} \quad (1.4.2.5.8)$$

```
> map(D[2],[f,g]);
```
$$\left[(x, y) \rightarrow -\sin(x\,y)\,x,\ (x, y) \rightarrow -\frac{e^x \cos(y)}{\sin(y)^2}\right] \quad (1.4.2.5.9)$$

Observe que funções foram necessárias acima, pois

```
> h:=x*y^2:v:=y/x:
> map(D[1],[h,v]);
```
$$\left[\mathrm{D}_1(x)\,y^2 + 2\,x\,\mathrm{D}_1(y)\,y,\ \frac{\mathrm{D}_1(y)}{x} - \frac{y\,\mathrm{D}_1(x)}{x^2}\right] \quad (1.4.2.5.10)$$

```
> map(D,[x^2,x*y,5*x*z^2]);
```
$$[2\,\mathrm{D}(x)\,x,\ \mathrm{D}(x)\,y + x\,\mathrm{D}(y),\ 5\,\mathrm{D}(x)\,z^2 + 10\,x\,\mathrm{D}(z)\,z] \quad (1.4.2.5.11)$$

```
> map(simplify,[sin(x)^2+cos(x)^2, x*y/y]);
```
$$[1, x] \quad (1.4.2.5.12)$$

```
> map(evalb,[simplify(sin(x)^2+cos(x)^2=1),x*
y/y=x]);
```
$$[\mathit{true}, \mathit{true}] \quad (1.4.2.5.13)$$

2. Comandos Básicos de Programação

Iterações

Existem duas formas de executar iterações através do comando for:

for contador **from** valor_inicial **by** intervalo **to** valor_final **while** expressão_booleana
do comando_1;
    comando_2;
    comando_3;

...
od;

e

for variável **in** expressão **while** expressão_booleana
do comando_1;
comando_2;
comando_3;
...
od;
Em lugar de **od** é possível escrever **end do**.
Os comandos em negrito devem se escritos em letras minúsculas. A maior parte é opcional. Vejamos um exemplo da primeira forma. Os números pares podem ser gerados da seguinte forma.

```
> restart:
> for i to 5 do 2*i; od;
```

$$2$$
$$4$$
$$6$$
$$8$$
$$10 \quad (2.1.1)$$

Na ausência do comando **from**, o valor inicial do contador é 1. Um forma quase equivalente do mesmo comando é

```
> for i from 2 by 2 to 10 do i; od;
```

$$2$$
$$4$$
$$6$$
$$8$$
$$10 \quad (2.1.2)$$

Segue um exemplo da segunda forma de executar iterações. Seja L uma lista de funções.

```
> L:=[exp(x^2),x^3,ln(x)];
```

$$L := \left[e^{x^2}, x^3, \ln(x) \right] \quad (2.1.3)$$

Queremos calcular uma aproximação para integral definida de 1 a 2 dessas funções.

```
> for i in L do Int(i,x=1..2)=evalf(int(i,x=1..2));
  od;
```

$$\int_1^2 e^{x^2}\,dx = 14.98997601$$

$$\int_1^2 x^3\,dx = 3.750000000$$

$$\int_1^2 \ln(x)\,dx = 0.386294361 \qquad (2.1.4)$$

Note que o contador *i* no comando acima recebe funções como valor. Vejamos o valor final de *i*.

```
> i;
```

$$\ln(x) \qquad (2.1.5)$$

Considere o seguinte problema. Suponha que temos uma lista de funções na variável x.

```
> L := [ x^2, g(x), sin(x),  a*exp(x^2)];
```

$$L := \left[x^2, g(x), \sin(x), a\,e^{x^2}\right] \qquad (2.1.6)$$

Queremos construir a lista das derivadas. O próximo comando não resolve o problema.

```
> for i in L do diff(i,x); od;
```

$$2\,x$$

$$\frac{d}{dx}\,g(x)$$

$$\cos(x)$$

$$2\,a\,x\,e^{x^2} \qquad (2.1.7)$$

A solução através do uso do comando de iteração requer primeiramente a inicialização de uma lista nula.

```
> derivadas := [ ];
```

$$derivadas := [\ ] \qquad (2.1.8)$$

Agora vem a iteração.

```
> for i in L do
>    derivadas := [ op(derivadas), diff(i, x) ];
> od:
```

Note que terminamos o comando *for do od* com dois pontos para que nada seja mostrado na tela. Os comandos dentro da iteração podem terminar com dois pontos ou ponto e vírgula. Isso não tem efeito algum em termos de mostrar resultados na tela. Vejamos o resultado.

```
> derivadas;
```

$$\left[2\,x,\ \frac{\mathrm{d}}{\mathrm{d}x}\,g(x),\ \cos(x),\ 2\,a\,x\,\mathrm{e}^{x^2}\right] \tag{2.1.9}$$

O valor que o contador assume no final da iteração pode ser importante, como é o caso no seguinte problema. Qual é o maior número primo menor que 808?

```
for i from 808 by -1 while not isprime(i) do  od;
```

O valor do contador contém a informação que desejamos:

```
> i;
```

$$a\,\mathrm{e}^{x^2} \tag{2.1.10}$$

Exceto o *do od*, todos os outras partes são opcionais. Vejamos um exemplo do comando while. Suponha que *N* tem o valor 68.

```
> N := 68;
```

$$N := 68 \tag{2.1.11}$$

Queremos dividir *N* por 2 enquanto *N* for par.

```
> while type(N,even) do N := N/2 od;
```

$$N := 34$$

$$N := 17 \tag{2.1.12}$$

Podemos ter iterações dentro de iterações, porém os cálculos executados dentro de iterações encaixadas não são mostrados na tela.. Para que os resultados sejam exibidos, o valor da variável printlevel tem que ser maior ou igual a 2:

```
> printlevel := 2;
```

$$printlevel := 2 \tag{2.1.13}$$

```
> for i to 2 do
>   for j to 2 do
>     a[i,j] := P[i]+Q[j];
>   od
> od;
```

$$a_{1,\,1} := P_1 + Q_1$$

$$a_{1,\,2} := P_1 + Q_2$$

$$a_{2,\,1} := P_2 + Q_1$$

$$a_{2,\,2} := P_2 + Q_2 \tag{2.1.14}$$

```
> printlevel := 1;
```

$$printlevel := 1 \tag{2.1.15}$$

Existem três formas particulares do comando for que são seq, add e mul. A sintaxe desses comandos são iguais, de forma que vamos apenas descrever o comando add que executa um somatório. A sua sintaxe é uma das seguintes formas

$$add(f,\ i = a\,..b)$$

$$add(f, i=L)$$

onde *f* é uma expressão que geralmente depende do índice *i*. Na primeira forma, o índice *i* assume valores inteiros de *a* até *b*, onde *a* e *b* tem que ser numéricos. Na segunda forma o índice assume valores da expressão ou estrutura de dados *L*. Por exemplo

```
> restart:
> add( a[i]*x^i, i=0..5 );
```

$$a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 \quad \textbf{(2.1.16)}$$

```
> add( sin(i), i=[theta,omega,tau]);
```

$$\sin(\theta) + \sin(\omega) + \sin(\tau) \quad \textbf{(2.1.17)}$$

```
> seq( i^2, i=1..5 );
```

$$1, 4, 9, 16, 25 \quad \textbf{(2.1.18)}$$

```
> mul(x*i,i=1..5);
```

$$120\, x^5 \quad \textbf{(2.1.19)}$$

```
> L := [seq(i, i=1..6)];
```

$$L := [1, 2, 3, 4, 5, 6] \quad \textbf{(2.1.20)}$$

```
> add(i,i=L);
```

$$21 \quad \textbf{(2.1.21)}$$

```
> mul( x-i, i=L );
```

$$(x-1)\,(x-2)\,(x-3)\,(x-4)\,(x-5)\,(x-6) \quad \textbf{(2.1.22)}$$

Estruturas de Controle

Em certas situações desejamos executar um comando somente se uma certa condição for verdadeira. Para isto usamos o comando if da seguinte forma:

if *condição* **then** *comando1* elif *comando2* **fi**.

Por exemplo vamos verificar quais números de uma lista de números são divisíveis por 7 ou por 11:

```
> L := [82,28,21,447,921,34,165];
```

$$L := [82, 28, 21, 447, 921, 34, 165] \quad \textbf{(2.2.1)}$$

```
> for i in L do
>       if frac(i/7)=0 or frac(i/11)=0 then print(i)
  fi
> od;
```

28

$$\frac{21}{165} \quad \textbf{(2.2.2)}$$

O comando *print*(i) é executado somente quando a condição

$\text{frac}\left(\frac{i}{7}\right) = 0$ **or** $\text{frac}\left(\frac{i}{11}\right)$ for satisfeita. O comando frac retorna a parte fracionária de um número. Este comando faz parte da família **`trunc, round, ceil`** e **floor** que trunca, arredonda para o inteiro mais próximo, arredonda para cima e arredonda para baixo respectivamente.

Outro exemplo de uso do comando **if** é para definir funções contínuas por parte. Por exemplo, a função **step** pode ser definida da seguinte forma:

```
> unitstep := x -> if x<=0 then -1 else 1 fi;
```

$$unitstep := x \rightarrow \textbf{if } x \leq 0 \textbf{ then } -1 \textbf{ else } 1 \textbf{ end if} \quad \textbf{(2.2.3)}$$

Assim:

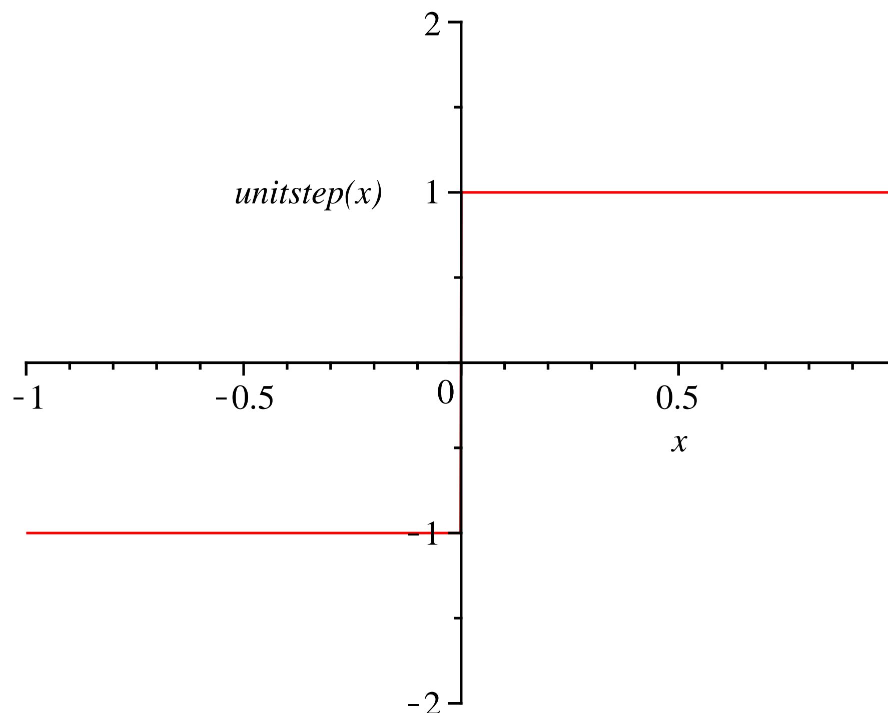
```
> unitstep(3);
```

$$1 \quad \textbf{(2.2.4)}$$

```
> unitstep(-2);
```

$$-1 \quad \textbf{(2.2.5)}$$

```
> plot('unitstep(x)',x=-1..1,`unitstep(x)`=-2..2);
```



A definição da função **unitstep** acima é muito rudimentar. Não podemos derivar, integrar ou obter a expansão em série de Taylor através dos comandos usuais do Maple. Ela não pode nem ser usada de maneira simbólica. Por exemplo, o próximo

comando retorna uma mensagem de erro:

```
> unitstep(x);
Error, (in unitstep) cannot determine if this expression is true or
false: x <= 0
```

Para fazer o gráfico, usamos as aspas direitas no primeiro argumento do comando **plot** para evitar que este erro fosse retornado. O desenvolvimento de procedimentos com características mais gerais compatíveis com outros comandos do Maple requerem o uso de procedimentos auxiliares que é um topicos abordados neste livro. Podemos melhorar a definição da função **unitstep** de forma que o comando unit *step*(x) retorne não avaliado caso a condição do comando **if** anterior não possa ser verificada:

```
> unitstep := x ->
>   if not type(x,'numeric')
>     then 'unitstep'(x)
>   else
>     if x<=0
>        then -1
>     else 1
>     fi;
>    fi;
```

unitstep := *x*→**if not** *type*(*x*, '*numeric*') **then** '*unitstep*'(*x*) **else** **(2.2.6)**

if $x \leq 0$ **then** -1 **else** 1 **end if**

end if

Agora teremos saídas simbólicas quando a entrada não é do tipo *numeric*:

```
> unitstep(a);
```

unitstep(*a*) **(2.2.7)**

É possível reescrever substituir os *else if* acima pelo comando *elif:*

```
> unitstep := x ->
>   if not type(x,'numeric')
>     then 'unitstep'(x)
>   elif x<=0
>        then -1
>     else 1
>   fi;
```

unitstep := *x*→**if not** *type*(*x*, '*numeric*') **then** '*unitstep*'(*x*) **elif** $x \leq 0$ **then** -1 **else** 1 **(2.2.8)**

end if

Porém, não foi dada nenhuma informação sobre como calcular a derivada da função

unitstep:

```
> diff(unitstep(x),x);
```

$$\frac{d}{dx}\,unitstep(x) \tag{2.2.9}$$

No caso de funções contínuas por partes, existe o comando **piecewise** que substitui os uso do comando **if** e cria automaticamente funções que podem ser diferenciadas, integradas e usados em um série de outros comandos do Maple. A função *step* pode ser definida como:

```
> unitstep := x -> piecewise(x<=0,-1,1);
```

$$unitstep := x \to piecewise(x \le 0, -1, 1) \tag{2.2.10}$$

Podemos tratá-la de maneira simbólica:

```
> unitstep(x);
```

$$\begin{cases} -1 & x \le 0 \\ 1 & otherwise \end{cases} \tag{2.2.11}$$

e diferenciá-la, entre outros cálculos:

```
> diff(unitstep(x),x);
```

$$\begin{cases} undefined & x = 0 \\ 0 & otherwise \end{cases} \tag{2.2.12}$$

Existem dois nomes especiais para serem usados dentro de iterações que são break e next. A variável break faz com que a iteração seja imediatamente interrompida. Por exemplo, no comando seguinte conseguimos interromper a iteração sem que o contador tenha um valor final.

```
> i:=0;
```

$$i := 0 \tag{2.2.13}$$

```
> do
>    i:=i+1;
>    if i=3 then break fi;
> od;
```

$$i := 1$$

$$i := 2$$

$$i := 3 \tag{2.2.14}$$

A variável next por sua vez não interrompe a iteração, mas quando avaliada faz com que uma volta seja pulada. Vamos imprimir os números ímpares da seguinte forma.

```
> for i to 10 do
>   if type(i,odd) then next fi;
>   print(i);
```

```
> od;
```

$$2$$
$$4$$
$$6$$
$$8$$
$$10 \quad \textbf{(2.2.15)}$$

Exercícios

1) Explique porque o comando não funciona e mostre como se soluciona o problema.

```
> for i to 10 do z := z+1 od;
```

$$z := z + 1$$

```
Error, too many levels of recursion
```

```
> restart:
```

```
> z:=zz;
```

$$z := zz \quad \textbf{(2.3.1)}$$

```
> for i to 10 do z := z+1 od;
```

$$z := zz + 1$$
$$z := zz + 2$$
$$z := zz + 3$$
$$z := zz + 4$$
$$z := zz + 5$$
$$z := zz + 6$$
$$z := zz + 7$$
$$z := zz + 8$$
$$z := zz + 9$$
$$z := zz + 10 \quad \textbf{(2.3.2)}$$

2) Sem executar o próximo comando diga quantas iterações ele vai gerar.

```
> for i to 10 do i := i+1 od:
```

3) Como se define uma iteração usando o comando *for do od* que executa a mesma tarefa que os seguintes comandos, onde a, b e n são números.
a) seq1:=seq(termo(a+i*n), i=0..(b-a)/n);
b) sum1:=sum('termo(i)', 'i'=a..b);
c) prod1:=product('termo(i)', 'i'=a..b);

d) list1:=[seq(termo(i), i=a..b)];
e) set1:={seq(termo(i), i=a..b)};
f) seq2:=seq(f(i),i=L);

4) Usando o comando *for do od*, gere as seguintes estruturas com 10 repetições.
a) [0, [0, [0, ... , [0]]]]]]]]]].
b) {{0}, {{0}}, {{{0}}}, ... }.
c) [tan(x), tan(tan(x)), tan(tan(tan(x))), ...].

5)
a) Gere uma lista L de números naturais randômicos com 100 elementos menores que 30.
b) Elimine os números repetidos.
c) Elimine os números repetidos sem alterar a ordem com eles aparecem na lista original.
d) Use um loop **for...in** para encontrar o maior número na lista. L Verifique seu resultado utilizando a função padrão de Maple **min.** Utilize a opção help se necessário.
e) Encontre o primeiro e o último número primo da lista L.
f) Encontre o maior e o menor número primo de L.

3. Introdução a Procedimentos

Sintaxe de procedimentos: proc() ... end proc

Procedimentos (procedures) são modos de empacotar partes de um programa. Os procedimentos em Maple são uma generalização dos mapeamentos (aplicações) definidos utilizando a sintaxe da "seta", e são usados em programas como se fossem funções. Eles podem ser reutilizados e controlados através de seus argumentos.
O termo técnico para a execução de um procedimento é calling. Idealmente, um procedimento irá interagir com o resto de um programa somente funcionalmente, tomando a entrada de um programa via seus argumentos e retornando a saída ao programa via seu valor de retorno. No entanto procedimentos podem também interagir via variáveis globais. Qualquer interação não-funcional é chamada side effect (efeito colateral).
A definição de um procedimento e sua execução são coisas completamente separadas, embora em Maple uma definição de procedimento possa ser seguida imediatamente por sua execução. Um procedimento de Maple é uma estrutura de dados que pode ser atribuída opcionalmente a um nome, ou seja, atribuindo-lhe uma variável.

A sintaxe de um procedimento é a seguinte:

```
proc(argumentos)
   local variáveis;
   global variáveis;
```

```
   corpo de comandos;
end proc
```

As palavras `local` e `global` são declarações que fornecem informações ao Maple. Variáveis locais têm atribuições que são locais ao procedimento. Por default, o claro retornado por um procedimento é o valor da última expressão avaliada no seu corpo. A sintaxe

```
(argumentos) -> comandos
```

que utilizamos para definir funções, é equivalente ao caso especial de um procedimento sem declarações:

```
proc(argumentos) comandos end proc
```

Por exemplo,

```
> f:=x->1/x;
```

$$f := x \rightarrow \frac{1}{x} \quad (3.1.1)$$

é equivalente a

```
> f:= proc (x) options operator, arrow; 1/x end proc;
```

$$f := x \rightarrow \frac{1}{x} \quad (3.1.2)$$

Exemplos

Já vimos procedimentos que definem funções. Vamos examinar agora como dar um nome e encapsular de forma eficiente uma sequência de comandos.

Vamos construir um procedimento, chamado **plotdif** que plota uma expressão f(x), junto com sua derivada f '(x), no intervalo $[a, b]$. Interativamente podemos computar a derivada da função utilizando diff e plotar as duas funções utilizando **plot** . Por exemplo:

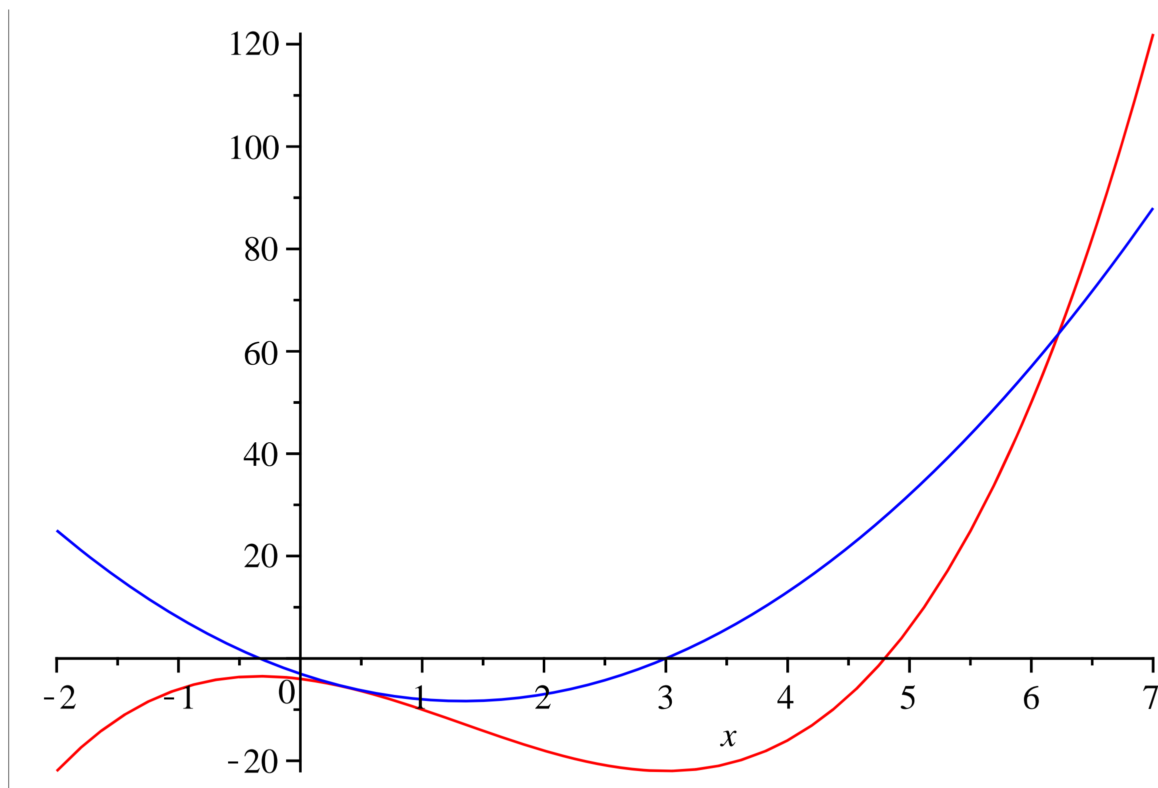
```
> y:=x^3-4*x^2-3*x-4;
```

$$y := x^3 - 4\,x^2 - 3\,x - 4 \quad (3.2.1)$$

```
> dy:=diff(y,x);
```

$$dy := 3\,x^2 - 8\,x - 3 \quad (3.2.2)$$

```
> plot([y,dy],x=-2..7,color=[red,blue]);
```



O seguinte procedimento combina a sequência de passos

1. Gráficos simultâneos

```
> plotdiff:=proc(y,x,a,b)
>     local dy;
>     dy:=diff(y,x);
>     plot([y,dy],x=a..b,color=[blue,black]);
> end;
```

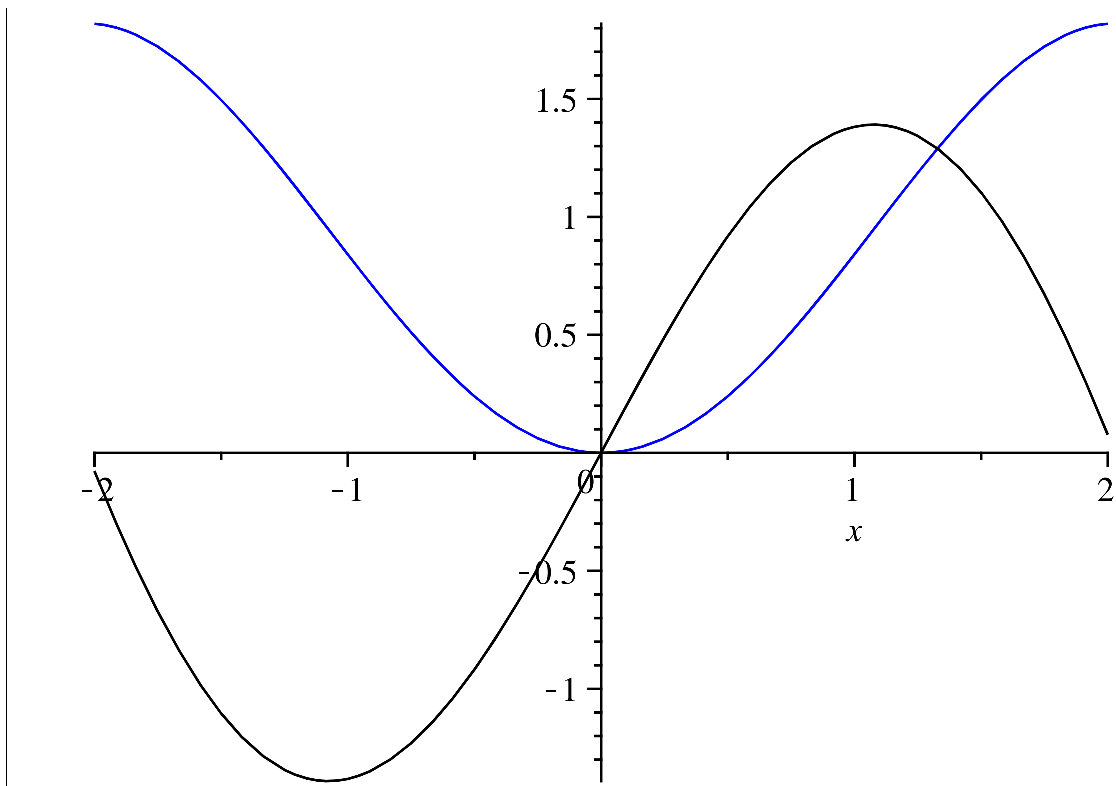
$plotdiff := \mathbf{proc}(y, x, a, b)$ **(3.2.3)**

local dy;

$dy := diff(y, x)$; $plot([y, dy], x = a..b, color = [blue, black])$

end proc

```
> plotdiff(x*sin(x),x,-2,2);
```



Este procedimento pode agora ser utilizado em outras rotinas. A declaração local assegura que dy é uma variável local, ou seja, este nome não terá qualquer atribuição fora do procedimento. Vários dos comando iterativos apresentados na seção anterior podem ser reescritos na forma de procedimentos.
Consideremos no próximo exemplo um procedimento que devolve o valor absoluto de um número, que vamos chamar de ABS (para distinguir de abs, que já está definido pelo Maple)

2.Valor absoluto de um número

```
> ABS:=proc(x)
>    if x<0 then
>      -x;
>    else
>       x;
>    fi;
> end;
```

$$ABS := \mathbf{proc}(x)\ \mathbf{if}\ x < 0\ \mathbf{then}\ -x\ \mathbf{else}\ x\ \mathbf{end\ if}\ \mathbf{end\ proc} \quad \textbf{(3.2.4)}$$

```
> ABS(-4.3);
```

$$4.3 \quad \textbf{(3.2.5)}$$

```
> ABS(3);
```

$$3 \quad \textbf{(3.2.6)}$$

procedimento ABS não pode lidar com uma entrada não-numérica:

```
> ABS(a);
Error, (in ABS) cannot determine if this expression is true or false: a
< 0
```

Como o sistema não sabe nada sobre a, não sabe o que fazer com ele. Em tais casos o

procedimento deveria ser capaz de retornar a entrada não avaliada. Para obter isso notemos o seguinte exemplo:

```
> 'ABS'(A);
```

$$ABS(A) \quad \textbf{(3.2.7)}$$

Estas aspas simples fazem com que a avaliação de ABS seja retardada. Portanto, podemos modificar o procedimento ABS para a seguinte forma;

```
> ABS:=proc(x)
>   if type(x,numeric) then
>      if x<0 then -x else x fi;
>    else
>        'ABS'(x);
>     fi
>   end;
```

$ABS := \textbf{proc}(x)$ **(3.2.8)**

if $type(x, numeric)$ **then if** $x < 0$ **then** $-x$ **else** x **end if else** $'ABS'(x)$ **end if**

end proc

```
> ABS(a);
```

$$ABS(a) \quad \textbf{(3.2.9)}$$

3. Raízes de um polinômio

Vamos construir um procedimento que plota raízes reais e complexas de um polinômio

```
> raizplot:=proc(p::polynom(constant,x))
>    local R, points;
>    R:=[fsolve(p,x,complex)];
>    points:=map(z->[Re(z),Im(z)],R);
>    plot(points,style=point,symbol=circle);
> end;
```

$raizplot := \textbf{proc}(p::(polynom(constant, x)))$ **(3.2.10)**

local $R, points$;

$R := [fsolve(p, x, complex)]$;

$points := map(z \rightarrow [\mathrm{Re}(z), \mathrm{Im}(z)], R)$;

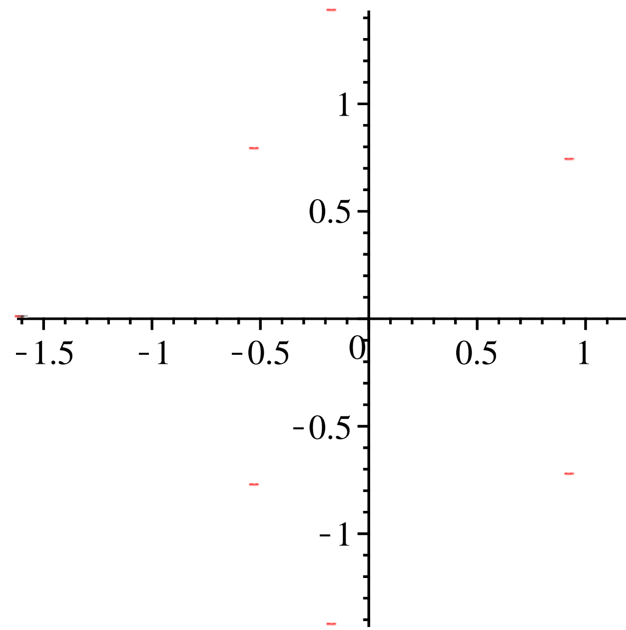
$plot(points, style = point, symbol = circle)$

end proc

```
> y:=x^8-3*x^4+4*x^3-x^2+x-5;
```

$$y := x^8 - 3x^4 + 4x^3 - x^2 + x - 5 \quad \textbf{(3.2.11)}$$

```
> raizplot(y);
```

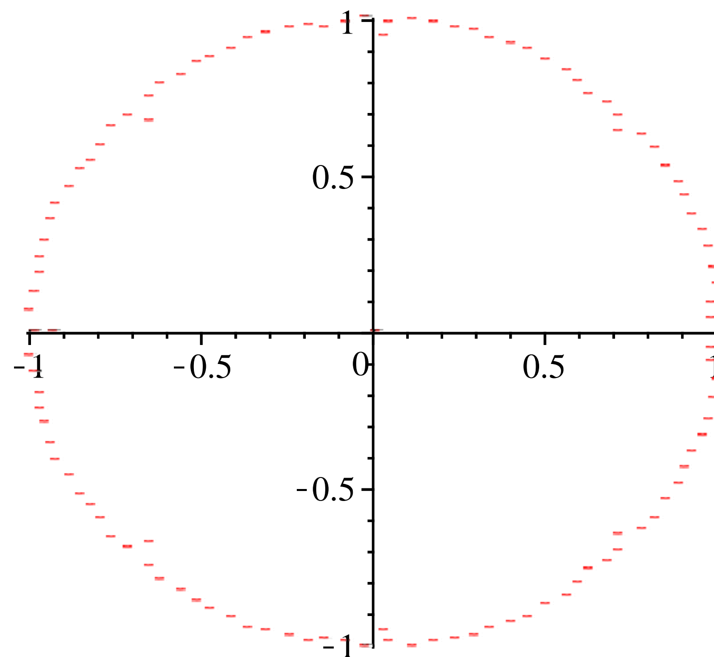


O comando randpoly gera um polinômio randômico:

```
> y:=randpoly(x,degree=120);
```

$$y := 87\,x^{115} - 56\,x^{76} - 62\,x^{25} + 97\,x^{10} - 73\,x^{2} \quad \textbf{(3.2.12)}$$

```
> raizplot(y);
```



Exercícios

1.Faça um procedimento de nome *clear* que limpa o valor de uma ou mais variáveis sem que haja necessidade de usar aspas direitas no argumento da função. Por exemplo:

```
a := 1; b:=2;
```

$$a := 1$$

$$b := 2$$

```
clear(a,b);
a; b;
```

$$a$$

$$b$$

2. Defina no Maple a função g abaixo, usando seta `->` e depois usando `g:=proc(...`

$$g(x) = \begin{cases} x^2 + x & x \leq 0 \\ \sin(x) & x < 3\pi \\ x^2 - 6\pi x + 9\pi^2 - x + 3\pi & 3\pi \leq x \end{cases}$$

3. Construa um procedimento que ordena uma lista lista numérica em ordem crescente.