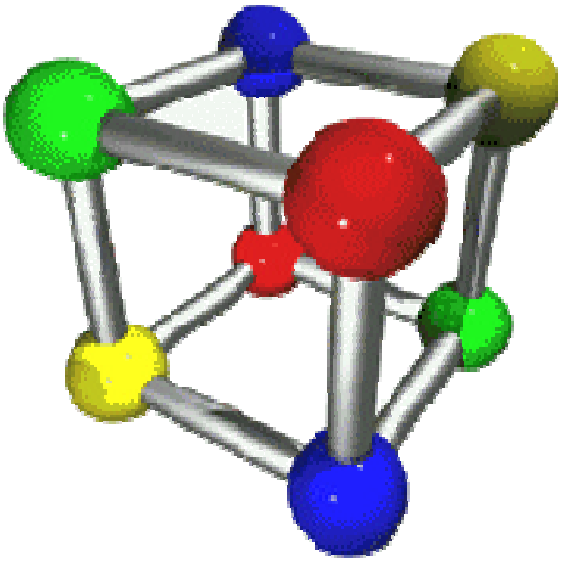
MuPAD Light



Tutorial

Ricardo Miranda Martins

rmiranda@vicosa.ufv.br

ÍNDICE

INTRODUÇÃO **4**
Computação Numérica Vs. Computação Simbólica, O que é o MuPAD, Objetivos do Tutorial.

INTERFACE, ENTRADA E SAÍDA DE COMANDOS **6**
Interface, Comandos e Funções, Obtendo ajuda.

ARITMÉTICA **9**
Operações e operadores elementares, Respostas usando ponto flutuante, Constantes Matemáticas, O Infinito, Precisão das respostas, Funções básicas para aritmética.

NÚMEROS COMPLEXOS **16**
A unidade imaginária, Operações com os complexos: potência, divisão e inverso de um número complexo.

ÁLGEBRA BÁSICA **17**
Definindo um polinômio, Encontrando raízes, Fatorando e Expandindo polinômios, Divisão de polinômios, resolvendo equações, Somatórios e produtórios, Binomial.

ESTRUTURAS DE DADOS **23**
Sequências, Operações com dados, Listas, Conjuntos e Operações, Tabelas.

FUNÇÕES **29**
Definindo funções, Funções por partes, Composição de funções.

GRÁFICOS **32**
Gráficos de funções de uma variável, Gráficos de funções de três variáveis, Plotando várias curvas, Plotando uma curva usando suas equações paramétricas, Plotando superfícies usando suas coordenadas, Gráficos de funções implícitas, Gráficos usando coordenadas polares, esféricas e cilíndricas.

CÁLCULO DIFERENCIAL E INTEGRAL **40**
Limites, Derivadas, Integrais.

ÁLGEBRA LINEAR **45**
Matrizes, Operações com matrizes, Matriz Inversa, Determinante de uma matriz, Polinômio característico, Autovalores e autovetores.

SISTEMAS LINEARES E OTIMIZAÇÃO **51**
Sistemas de equações lineares, Equações polinomiais, Raízes de polinômios. Problemas de Programação Linear: O método Simplex.

TEORIA DOS NÚMEROS **55**
Divisores, Fatoração, MDC e MMC, Funções número-teóricas, Números de Fibonnacci, Representação g-ádica, Congruências Lineares.

BASES DE GRÖBNER **60**
S-polinômio, Base de Gröbner para um ideal.

PROGRAMAÇÃO **61**
Imprimindo objetos na tela, Laços For e While, Função if, Criando procedimentos.

BIBLIOGRAFIA **68**
Bibliografia.

INTRODUÇÃO

Dentre as ferramentas computacionais que podem ser utilizadas para se resolver um determinado problema, existem aquelas puramente numéricas, que utilizam algoritmos bem conhecidos para encontrar soluções de equações, e existem aquelas algébricas. A principal diferença entre elas é a exatidão da resposta: na computação numérica os dados (números) são armazenados como números reais, e como a capacidade de memória dos computadores é limitada, os arredondamentos acabam afetando a precisão da resposta. Já na computação simbólica, como os dados são armazenados como frações e manipulados algebricamente, a precisão da resposta é total. Outra vantagem da computação simbólica é a possibilidade do uso de "fórmulas fechadas", ou seja, a resolução de problemas literais.

O MuPAD (Multi-Processing Algebra Data Tool) é um sistema de computação algébrica (C.A.S. – Computer Algebra System) interativo, desenvolvido à partir de 1990 na Universidade de Paderborn (Alemanha) com todos os recursos dos principais softwares comerciais nesta área, como o Mathematica e o Maple. A principal vantagem do MuPAD sobre estes softwares é a possibilidade de serem definidos novos tipos de dados (criando estruturas algébricas, como grupos ou anéis, por exemplo) e a de se adicionarem programas em C++ ao núcleo do sistema.

O MuPAD pode ser copiado no site http://www.mupad.de, que é o site do fabricante. Existem versões para Windows, Linux e MAC. Para Linux, a versão completa é livre, e para Windows e MAC existem versões de demonstração (MuPAD Light) com uso restrito de memória, restrição esta que pode ser removida com o registro (gratuito) do programa. Outra restrição da versão Light é a impossibilidade de editar uma linha de comando já interpretada. Assim, é necessário copiar-colar os comandos para alterá-los. Ainda, só é possível salvar o arquivo de trabalho em formato texto. A versão em que este tutorial se baseia é a Light 2.5.2.

Este tutorial cobrirá os aspectos básicos do MuPAD para os mais variados fins, como a resolução de problemas de Álgebras Abstrata e Linear, Otimização e Cálculo

Diferencial e Integral, além de comandos usados para programar procedimentos e laços de programação. Ao invés de sintaxes formais, os comandos são apresentados com exemplos comentados para propiciar um uso imediato das funções.

Este tutorial foi planejado com o objetivo de ser material didático de um curso sobre o MuPAD para os alunos do Departamento de Matemática da Universidade Federal de Viçosa. Assim, a ordem de apresentação dos tópicos segue a ordem na qual as disciplinas relacionadas são cursadas pelos alunos de Matemática na UFV. Ainda assim é possível estudar este tutorial alterando a ordem das seções, pois pouca ligação existe entre seções que cobrem áreas distintas da Matemática. O mesmo motivo explica o caráter básico da abordagem dada aos conceitos matemática e computacionais.

Este trabalho é parte de um projeto de Iniciação Científica do Programa Institucional de Bolsas de Iniciação Cietífica – PIBIC/UFV, financiado pelo CNPq.

Ricardo Miranda Martins
Viçosa, 2004

INTERFACE, ENTRADA E SAÍDA DE COMANDOS

Ao ser iniciado, o MuPAD fica pronto para receber instruções. Sua janela é bem simples, com um espaço para digitação de comandos e apresentação de resultados e alguns poucos botões e menus. Abaixo segue uma descrição dos menus e botões na interface do MuPAD.

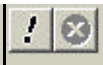
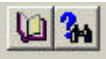
File	
Save as	*Salva o conteúdo atual da planilha*
Print	*Imprime a planilha atual*
Print Preview	*Visualiza a impressão*
Print Setup	*Configura a impressão*
Exit	*Fecha o MuPAD*

Edit	
Undo	*Desfaz o último comando*
Cut	*Recorta a seleção atual*
Copy	*Copia a seleção atual*
Paste	*Cola o conteúdo da área de transferência*
Delete All	*Apaga toda a planilha atual*

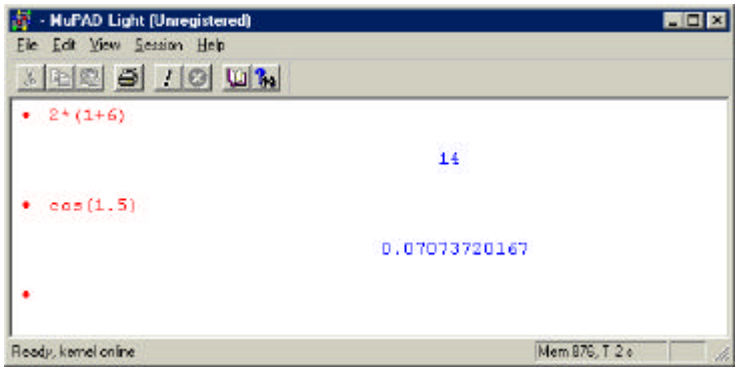
View	
Toolbar	*Mostra/Oculta os ícones*
Status bar	*Mostra/Oculta a barra de status*
Options	*Opções de fonte e caminho de bibliotecas*

Session	
Evaluate	*Executa linha de comando atual*
Pretty Print	*Alterna o modo de apresentar os resultados*
Text width	*Configura o tamanho do texto*
Stop Kernel	*Força a parada do comando atual*

Help	
Browse Manual	*Procura algo o sistema de ajuda*
Open tutorial	*Abre o tutorial interno*
Help topics	*Mostra os tópicos da ajuda*
Register	*Registra o MuPAD Light*
About ...	*Informações sobre o MuPAD*

			
Recortar Copiar Colar	Imprimir	Executar linha Parar computação	Iniciar Tutorial interno Procurar na ajuda

A entrada de comandos é bem intuitiva, e as operações aritméticas são realizadas do modo usual. O modo de se usar uma função é bem parecido com o que é feito manualmente. Cuidado! O MuPAD diferencia maiúsculas de minúsculas, e a maioria de suas funções são em letras minúsculas. Assim, **cos(x)** é diferente de Cos(x) ou de cos(X). Veja um exemplo:



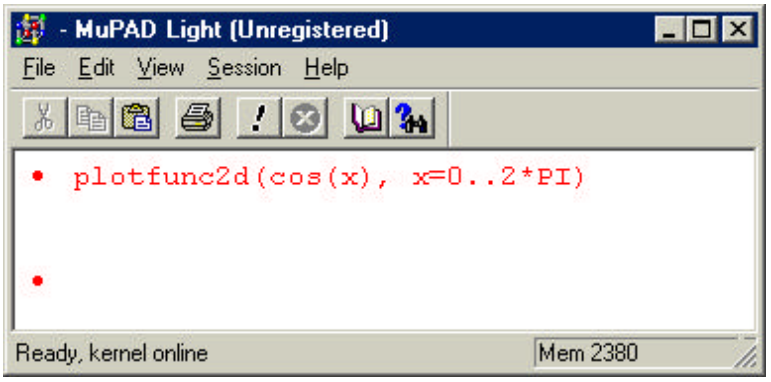
Na interface do MuPAD, os textos em vermelho denotam entrada de comandos, e em azul, a resposta.

Na figura acima, o primeiro comando é um comando aritmético normal. O símbolo * é usado para a multiplicação. Para divisão, usamos / e para potência, ^, por exemplo, para calcularmos 23 devemos digitar no MuPAD: 2^3 [ENTER].

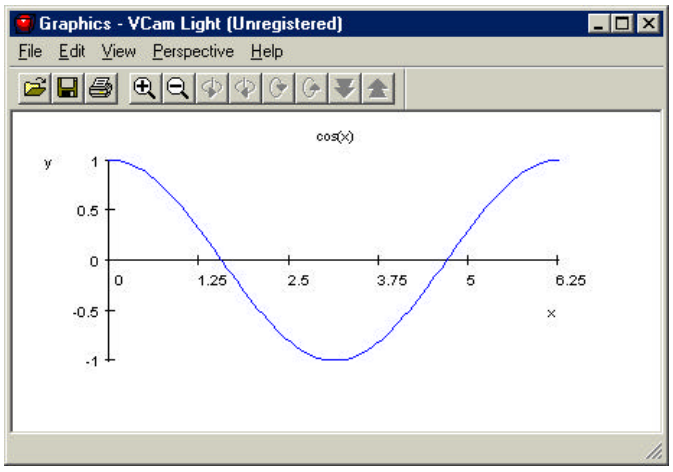
Na segunda linha da figura, o MuPAD é instruído a calcular o valor da função cosseno no ponto 1,5 (os valores devem ser passados em radianos e o separador de decimais é o ponto, e não a vírgula).

A maioria dos comandos no MuPAD segue a sintaxe (sintaxe é o "modo" de usar o comando) da função cosseno, ou seja para calcularmos o valor da função **f** no ponto **x**, devemos digitar **f(x)**. Veja um exemplo:

Se quisermos que o MuPAD faça o gráfico da função **cos(x)**, devemos usar a função **plotfunc2d**, que, como o nome já diz, "plota" (desenha) funções em duas dimensões. A sintaxe da função **plotfunc2d** é a seguinte: **plotfunc2d(f(x), g(x), x=a..b)**, onde **f** e **g** são funções e **a..b** é o intervalo que queremos do gráfico. No MuPAD, ficaria assim:



O resultado aparecerá numa nova janela:



O sistema de ajuda do MuPAD é bem completo, assim, se tivermos qualquer dúvida, podemos usar os seguintes comandos:

F2: Ativa o sistema de ajuda

>> **info(comando):** Mostra informações sobre "comando"

>> **?comando:** Mostra ajuda sobre "comando"

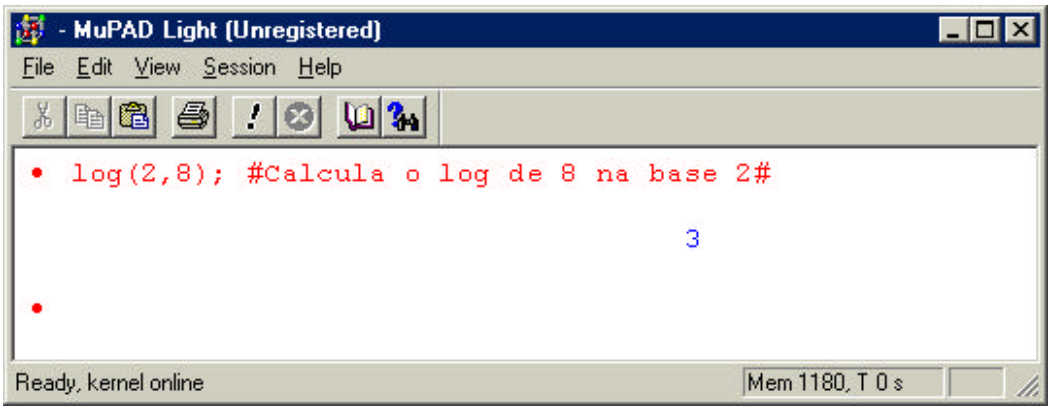
ARITMÉTICA

O MuPAD tem vários recursos para se trabalhar somente com números.

Os comandos abaixo estarão todos comentados. Os comentários começam e terminam com um sinal #. No MuPAD, todo o texto escrito entre #'s não é executado. É possível entrar vários comandos numa única linha, bastante separa-los com um ponto-e-vírgula (;), e a ordem da resposta será a mesma ordem da entrada dos comandos. Os comandos podem ainda ser separados por dois pontos (:), porém, seu resultado não será mostrado. Neste tutorial, o prompt do MuPAD (a bolinha vermelha) será substituída pelo sinal ">>" para simplificar a notação. Então, a linha:

```
>> log(2,8); #Calcula o log de 8 na base 2#
                                    3
```

ficaria no MuPAD:



Agora, as operações aritméticas básicas:

```
>> 2+3; #Soma#
                                    5
>> 2*3; #Multiplicação#
                                    6
```

```
>> 6/2; #Divisão#
                                3
```

```
>> sqrt(9); #Raiz quadrada de 9#
                                3
```

É possível ainda realizar as operações aritméticas utilizando funções internas do MuPAD ao invés dos operadores usuais. Para isso, existem as funções **_plus**, **_subtract**, **_mult**, **_divide** e **_power** (para adição, subtração, multiplicação, divisão e potenciação, respectivamente). O uso de todas elas é semelhante:

```
>> _plus(1,10);
                               11
```

```
>> _power(2,10)=2^10; #Potenciação, de duas formas #
                          1024 = 1024
```

Agora um exemplo mais complicado do uso da função **_plus** para produzir um somatório:

```
>> _plus(i^2 $ i=1..10); #Calcula a soma dos quadrados dos
inteiros, de 1 até 10#
                              338350
```

Algumas vezes, quando uma divisão não for exata, a resposta será dada na forma de fração. O mesmo acontece para extração de raiz não exata e em alguns outros casos. Para forçá-lo a exprimir a resposta como um número real, o comando é **float**. Por exemplo:

```
>> 3/2
                               3/2
```

```
>> float(3/2)
                                1.5

>> sqrt(6)
                                 1/2
                                6

>> float(sqrt(6))
                            2.449489743
```

Para evitar o uso do comando **float**, é possível digitar o número com casas decimais. Por exemplo:

```
>> sqrt(6.0)
                            2.449489743
```

Para calcular a raiz cúbica de um número, deve-se usa-la na forma de fração, ou seja, a raiz cúbica de 2 é o mesmo que $2^{1/3}$. Ou seja:

```
>> 2.0^(1/3)
                            1.25992105
```

O valor das famosas constantes matemáticas Pi e "e" (Número de Euler) aparece no MuPAD com uma precisão tão grande quanto se queira (ou quanto a memória do computador permitir). O uso destas constantes é como usual, ou seja:

```
>> float(PI)
                            3.141592654
>> float(E)
                            2.718281828
```

O valor do Número de Euler pode ser obtido pela função exponencial aplicada no ponto x=1:

```
>> float( exp(1) )
                        2.718281828

>> cos(PI)
                            -1

>> float(PI^2+2*E); #Sem o float, o MuPAD simplesmente reescreve esta expressão#
                        15.30616806
```

Caso seja necessário usar o resultado do último cálculo, é possível usar o símbolo %. Seu uso é simples, como no exemplo abaixo:

```
>> PI*E^2; # O MuPAD não irá retornar o valor desta expressão, ou seja, precisaremos usar a função float#
                                2
                          PI exp(1)

>> float(%); # Usando a função float sem reescrever toda a expressão anterior#
                        23.21340436
```

Outra “constante” existente no MuPAD é a **infinity**. Ela representa “o infinito”, ou seja, um número muito grande. É possível operar com **infinity**:

```
>> 10+infinity;
                          infinity
```

```
>> 1-infinity
                              -infinity

>>  infinity/infinity; # Cuidado! Isso não vale 1!#
                              undefined
```

Em alguns cálculos envolvendo o infinito o MuPAD realiza um processo de limite antes de mostrar o resultado:

```
>> 1/infinity
                                  0
```

A precisão dos resultados pode ser facilmente alterada. É necessário, para isso, alterar o valor de uma variável. As variáveis, no MuPAD são definidas da mesma forma que na maioria das linguagens de programação. A sintaxe é **VARIÁVEL:=VALOR**. No MuPAD, a variável **DIGITS** guarda a informação sobre quantos dígitos devem ser considerados no cálculo. O padrão é 10, mas pode ser alterado. Por exemplo:

```
>> float(PI)
                              3.141592654

>> DIGITS:=200
                                  200

>> float(PI); #Agora teremos Pi com 200 dígitos!#
3.1415926535897932384626433832795028841971693993751058209749445923078164062862089986280348253421170679821480865132823066470938446095505822317253594081284811174502841027019385211055596446229489549303820
```

Existem funções específicas para lidar com aritmética. Abaixo seguem algumas delas:

```
>> abs(-2); # Retorna o módulo de um número#
                              2

>> ceil(1.51); #Arredonda o número para o próximo inteiro#
                              2

>> floor(1.51); #Arredonda para o inteiro anterior#
                              1

>> round(1.51); #Arredondamento padrão, para o inteiro mais
próximo#
                              2

>> trunc(8.374637); #Retorna a parte inteira de um número#
                              8

>> fact(4)=4!; #Função Fatorial, que pode ser usada de duas
maneiras#
                           24=24

>> ifactor(338272); # Fatora em primos um dado inteiro#
                          5       2
                        2  11  31

>> isprime(1001); isprime(3847); #Verifica se um número é
primo#
                          false
```

```
>> 23 mod 5; #Retorna o resto da divisão do primeiro parâmetro pelo segundo#
                                3
```

```
>> 23 div 5; #Retorna o quociente na divisão do primeiro parâmetro pelo segundo#
                                4
```

```
>> sign(-7); sign(7); #Função sinal: retorna –1 se o número for negativo e 1 se for positivo#
                              -1, 1
```

Um outro sinal importante é o de diferente. No MuPAD, o sinal de diferente é o <> (menor ou maior, ou seja, diferente!). O comando abaixo pergunta se 2 é diferente de 3. A resposta, claro, é Verdade.

```
>> is(2 <> 3)
                              TRUE
```

NÚMEROS COMPLEXOS

A unidade imaginária i, no MuPAD, é representada como I (i maiúsculo). Um número complexo, ou seja, um número do tipo a+bi, onde i= -1. No MuPAD eles são definidos da mesma forma, e é possível operá-los e explorar suas propriedades.da mesma forma que os números reais.

```
>> x:=2*I+1: #Define o número complexo x#
>> y:=3*I-5: #Define o número complexo y#

>> x+y; x*y; #Algumas operações#
                          - 4 + 5 I
                          - 11 - 7 I

>> x^3; #Potenciação de um número complexo#
                          - 11 - 2 I

>> x/y; #Divisão entre complexos#
                       1/34 - 13/34 I

>> abs(x); #Retorna a norma de x#
                             1/2
                             5

>> Re(x); #Parte real de x#; Im(x); #Parte imaginária de x#
                             1
                             2

>> 1/x; #Inverso de x#
                         1/5 - 2/5 I
```

ÁLGEBRA BÁSICA

Como um Sistema de Computação Algébrica, o MuPAD mostra sua força quando tratamos de problemas onde é necessária a manipulação algébrica de expressões. Ao contrário de calculadoras (não-programáveis como as famosas HP48/49), o MuPAD pode reconhecer e operar objetos tais como polinômios, funções, matrizes, entre outros.

Inicialmente lidaremos com polinômios em uma variável.

Os polinômios são digitados da forma natural, ou seja, com x denotando a indeterminada (variável) e os coeficientes numéricos. Ao trabalhar com polinômios, é sempre bom armazenar os polinômios em variáveis, ou seja:

```
>> p:=x^2+4*x+2
```

$$4\,x + x^2 + 2$$

```
>> solve(p); #Encontra as raízes do polinômio p#
```

$$\{[x = 2^{1/2} - 2],\ [x = -2^{1/2} - 2]\}$$

```
>> numeric::polyroots(p); #Usa métodos numéricos para
encontrar aproximações das raízes do polinômio p#
                [-0.5857864376, -3.414213562]

>> p+1
```

$$4\,x + x^2 + 3$$

```
>> q:=x^3+4
```

$$x^3 + 4$$

Abaixo seguem alguns comandos úteis para trabalhar com polinômios.

```
>> factor(p+2); #Fatora polinômios#
                                2
                             (x+2)

>> factor(q-5)
                                  2
                      (x - 1) (x + x  + 1)

>> expand(p*q); # Expande um polinômio#
                        2      3      4    5
              16 x + 4 x  + 2 x  + 4 x  + x  + 8
```

Para dividir dois polinômios, o comando é **divide(f,g)**, e ele retorna o quociente e o resto da divisão do polinômio f pelo g.

```
>> divide(p,q)
                                  2
                         0, 4 x + x  + 2
```

O que deu errado? Nada. Veja que o grau de p é maior que o grau de q. Assim, o quociente da divisão é 0 e o resto é o próprio p. Se invertermos:

```
>> divide(q,p)
                         x - 4, 14x + 12
```

Para obter somente o quociente e/ou o resto de uma divisão, podemos fazer como abaixo:

```
>> divide(q,p, Quo); #Retorna o quociente da divisão#
                              x - 4

>> divide(q,p, Rem); #Retorna o resto (Remainder) da divisão#
                            14 x + 12
```

Ao trabalhar com equações polinomiais, existem dois comandos bem úteis:

```
>> a:=p=3; #A variável a recebe a equação p=3#
                                 2
                        4 x + x  + 2 = 3

>> lhs(a); #Left-Hand Side, retorna o lado esquerdo da
equação a#
                                 2
                          4 x + x  + 2

>> rhs(a); #Right-Hand Side, retorna o lado direito da
equação a#
                                3
```

Para resolvermos uma equação polinomial, o comando é novamente o **solve**:

```
>> solve(p = 30)
                       1/2                  1/2
           {[x = - 4 2    - 2], [x = 4 2    - 2]}
```

Cuidado! Devido ao espaço entre o “4” e o “2”, a resposta acima deve ser lida como “$x=-4\sqrt{2}-2$ ou $x=4\sqrt{2}-2$”.

Agora duas funções muito úteis para se trabalhar com quociente de polinômios:

```
>> t:=p/q: #A variável t recebe p/q, uma fração. O sinal dois-pontos no final, ao invés da vírgula, faz com que este comando não dê resposta#

>> numer(t); # Extrai o numerador de uma fração#
                                2
                          4 x + x  + 2

>> denom(t); # Extrai o denominador de uma fração#
                                3
                               x + 4
```

Algumas expressões são simplificadas de uma forma bem inteligente pelo MuPAD:

```
>> a:=cos(x)^2+sin(x)^2: # “sin” (sine) é a função seno#

>> simplify(a)
                                1
>> b:=(x^2-1)/(x+1)-(x-1):

>>simplify(b);
                                0
```

A decomposição em Frações Parciais é um dos recursos do MuPAD. Usando Frações Parciais, é possível simplificar equações racionais. Por exemplo:

```
>> p:=4*x^2/(x^3-2*x-1):
```

```
>> partfrac(p)
                                  4          4
                                ----- + ----------
                                x + 1     2
                                        x  - x - 1
```

Cálculos envolvendo somatórios e produtórios também são possível no MuPAD, com o uso dos comandos **sum** e **product**. Veja abaixo:

```
>> sum(x, x=1..10); #Soma x, com x indo de 1 até 10, ou seja:
1+2+3+4+...+9+10#
                                  55
```

```
>> product(x, x=1..10); #Quase o mesmo acima, porém, com a
multiplicação: 1*2*3*4*...*9*10.#
                               3628800
```

Ainda é possível usar expressões mais complicadas nos somatórios e produtórios:

```
>> p:=x->x^2+2*x+3; #Define uma função polinomial#
                          x -> x^2 + 2*x + 3
```

```
>> sum(p(x),x=1..5); #Faz a soma p(1)+p(2)+...+p(5)#
                                  100
```

```
>> sum(1/n^2, n=1..infinity); #Um resultado interessante:
qual será a soma dos inversos dos quadrados dos naturais?#
                                    2
                                  PI
                                  ---
                                   6
```

```
>> soma:=sum(x^y, y=1..10); #Cria uma soma#
          2    3    4    5    6    7    8    9    10
      x + x  + x  + x  + x  + x  + x  + x  + x  + x

>> product(soma, x=1..5);
                  3092789258581550616000 0
```

O resultado acima equivale a definir uma função polinomial s(x) a partir do polinômio soma e fazer a conta s(1)*s(2)*...*s(5).

Outro exemplo importante, usando a função **binomial**, usada para calcular binomiais: binomial(n,i)=n!/[(n-i)!*i!].

```
>> sum( binomial(15,n), n=0..15)
                            32768
>> 2^15;
                            32768
```

Coincidência? Não, já que $\binom{n}{p}=2^n$!

ESTRUTURAS DE DADOS

Estruturas são formas de se agrupar dados que tem uma certa estrutura e ordem. Para manipular estes dados, existem operações especiais e regras especiais.

As principais estrutura de dados são as sequências, que no MuPAD podem ser criadas muito facilmente. A partir delas, podemos criar listas e conjuntos O comando abaixo cria uma sequência e a armazena na variável seq.

```
>> seq:=a,b,c,d
                                a,b,c,d
```

Para criar sequências longas, é melhor usar indexadores ao invés de letras para diferenciar as variáveis. Isto é feito com o símbolo $, como abaixo:

```
>> seq:=a[i] $ i=1..10
a[1], a[2], a[3], a[4], a[5], a[6], a[7], a[8], a[9], a[10]
```

Podemos extrair elementos de uma sequência pelo sua posição, ou seja:

```
>> seq[2]; # Mostra o segundo elemento da sequência seq#
                              a[2]
```

Ainda, é possível alterar o valor de um elemento dentro de uma seqüência:

```
>> seq[2]:=5
                                5

>> seq
  a[1], 5, a[3], a[4], a[5], a[6], a[7], a[8], a[9], a[10]
```

Repare que o segundo elemento agora não é mais o a[2], e sim o número 5.

Vamos definir agora uma seqüência numérica para explorar alguns dos comandos do MuPAD para se lidar com seqüência.

```
>> numeros:=0,1,2,3,4,5,6,7,8,9
                    0,1,2,3,4,5,6,7,8,9
```

A seqüência acima poderia ter sido criada mais facilmente com o comando **numeros:=i $i=0..9**.

Podemos extrair os extremos de uma sequência com comandos bem intuitivos:

```
>> max(numeros); min(numeros)
                              9
                              0
```

Para obtermos a soma dos elementos de uma sequência, o comando está abaixo. Repare que o índice de soma (i) abaixo é diferente do índice que usamos para criar a sequência numeros.

```
>> sum( numeros[i], i=1..10)
                             45
```

Uma lista é uma seqüência ordenada de objetos delimitada por colchetes. Algumas das operações com seqüências listadas acima podem ser usadas para listas. Veremos mais algumas agora:

```
>> L:=[ 1,6,45,3,5,6,8,x]
                    [ 1,6,45,3,5,6,8,x]
```

```
>> nops(L); #Mostra o número de elementos na lista L#
                              8

>> contains(L, 4); #Verifica se o número 4 aparece na lista
lista L; senão, retorna 0#
                              0

>> contains(L, 6); #Caso o segundo argumento apareça na
lista, ele retorna a posição da primeira ocorrência#
                              2
```

Para adicionar elementos a uma lista, o comando é **append**. Para concatenar (unir) duas listas, poder usar o operador **.**, como abaixo:

```
>> M:=[a,b,c]
                           [a,b,c]

>> append(M,d)
                          [a,b,c,d]

>> L.M
            [1, 6, 45, 3, 5, 6, 8, x, a, b, c]
```

É possível ordenar uma lista de acordo com a magnitude dos objetos. Por exemplo, vamos ordenar a lista L.M:

```
>> sort(L.M)
            [a, b, c, x, 1, 3, 5, 6, 6, 8, 45]
```

Pode-se usar o comando **zip** para operar com duas listas. Sua sintaxe é a seguinte: **zip(A,B,comando)**, onde A e B são objetos e comando é um comando que use dois argumentos (este comando será usado para "unir" A e B).

```
>> A:=[1,2,3]: B:=[x,y,z]:

>> zip(A,B,_plus); #Estamos unindo A e B aditivamente#
                    [x + 1, y + 2, z + 3]

>> zip(A,B,_mult); #Estamos unindo A e B multiplicativamente#
                        [x, 2 y, 3 z]

>> zip(A,B,_power); #Estamos unindo A e B por potências#
                                y   z
                             [1, 2 , 3 ]
```

Conjuntos, que são sequências não-ordenadas de objetos, podem ser criados da mesma forma que listas e sequências. As operações mais comuns com conjuntos (união, interseção e subtração) são realizadas da forma usual.

Cuidado! A ordem apresentada na tela quando se define um conjunto é uma ordem aleatória, criada a partir de regras internas do MuPAD.

```
>> A:={1,2,3}: B:={4,5,6}: C:={1,4}: #Estamos criando os
conjuntos A, B e C#

>> A union B; #União do conjunto A com o B#
                        {1,2,3,4,5,6}

>> A minus B; #Conjunto A menos o conjunto B#
                           {1,2,3}
```

```
>> A minus C; #Conjunto A menos o conjunto C#
                              {2,3}

>> (A intersect C) union B; #Primeiro o MuPAD calcula a
interseção do conjunto A com o C, e faz a união deste
resultado com o conjunto C; repare na diferença entre o
resultado deste comando e do próximo #
                            {1,4,5,6}

>> A intersect (C union B); #Calcula a interseção do conjunto
A com a união dos conjuntos C e B#
                              {1}
```

Uma outra forma de agrupar dados é em tabelas. As tabelas são como matrizes ou vetores, apesar destes objetos serem distintos para o MuPAD.

```
>> A:=array(1..2,1..2); #Cria uma tabela vazia com 2 linhas e
2 colunas#
                    +-                    -+
                    |  ?[1, 1], ?[1, 2]   |
                    |                     |
                    |  ?[2, 1], ?[2, 2]   |
                    +-                    -+
```

```
>> B:=array(1..3, 1..2, [ [1,2], [3,4],[5,6]])
                          +-      -+
                          |  1, 2  |
                          |  3, 4  |
                          |  5, 6  |
                          +-      -+
```

Os elementos de uma tabela podem ser modificados da mesma forma que os de uma lista, porém, precisamos de especificar a localização do elemento, fazendo uma atribuição como abaixo:

```
>> B[1,2]:=0; # O valor na linha 1 e coluna 2 será 0#
                               0
```

```
>> B; #Conferindo se o comando acima funcionou #
                          +-      -+
                          |  1, 0  |
                          |        |
                          |  3, 4  |
                          |        |
                          |  5, 6  |
                          +-      -+
```

FUNÇÕES

Existem duas maneiras de se definir uma função no MuPAD. A mais direta (e simples) é usando o operador ->, como abaixo:

```
>> f:=x->2*x: #Define a função f, que multiplica seu argumento por 2#

>> g:=(x,y)->x^2+y^2: #Define a função g, que, dados dois números, retorna a soma de seus quadrados#
```

É possível criar uma função usando uma linguagem de programação semelhante ao Pascal. Algumas funções mais complexas devem ser criadas desta maneira. Abaixo, um exemplo (use Shift+Enter para pular a linha sem executar o comando):

```
>> g:=proc(x,y) begin
>>   if x > y then return(x)
>>     else return(y)
>>   end_if
>> end_proc
```

Na primeira linha do comando acima, **g** é definida como sendo um procedimento onde serão usadas duas variáveis. Nas linhas 2 e 3, o objetivo da função é descrito: se o primeiro parâmetro for maior que o primeiro, retorna-o; senão, ou seja, se o segundo for o maior, retorna esse. Na linha 4 o comando **if** é terminado e na 5ª linha o comando **proc** é finalizado. Assim, dados dois números, a função **g** mostra o maior deles.

```
>> g(100,4)
                                100

>> g(4,100)
```

```
                                100
```

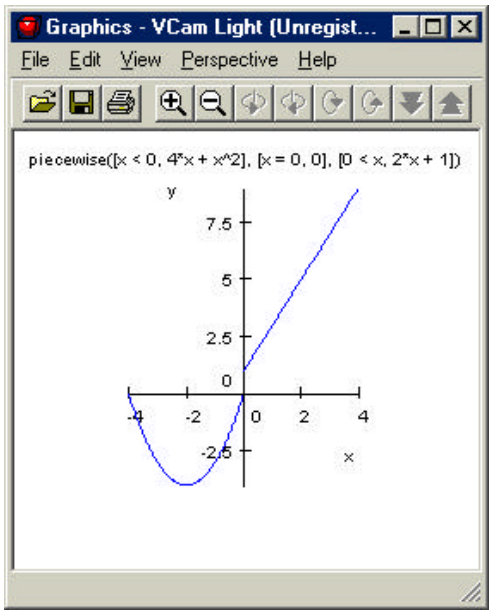
Para calcular o valor da função em um ponto, a sintaxe é idêntica à maneira usada manualmente:

```
>> f(4)
                                8

>> f(3.14)
                              6.28

>> g(1,2)
                                5

>> g(1,f(5)); #Aqui usamos uma composição de funções
equivalente a calcularmos g(1,10), já que f(5)=10#
                               101
```



No MuPAD, se for preciso definir uma função por partes, deve-se usar a função **piecewise**:

```
>> P:=x->piecewise( [x<0,
x^2+4*x], [x=0, 0], [x>0, 2*x+1])
x -> piecewise([x<0, x^2 + 4*x],
[x =0, 0], [0 < x, 2*x + 1])
```

Ao lado, um gráfico da função acima definida. Os comandos para produzir gráficos como este serão apresentados na próxima seção.

Para calcular composição de funções, pode-se proceder de duas maneiras:

```
>> h:=f@f: #Definimos a função h como sendo f composta com f#
>> h(1); #Agora calculando h(1)=f o f(1)=f(f(1))=f(2)=4#
                                4

>> h:=f@@8: #Definindo h como a composta de f com ela mesma 8
vezes#
>> h(1)
                               256
```

A composição de funções pode ser usada juntamente com outras operações.

```
>> u:=g*cos@f
           ((x, y) -> x^2 + y^2)*cos@(x -> 2*x)

>> u(1,2)
                           5 sin(2)
```

GRÁFICOS

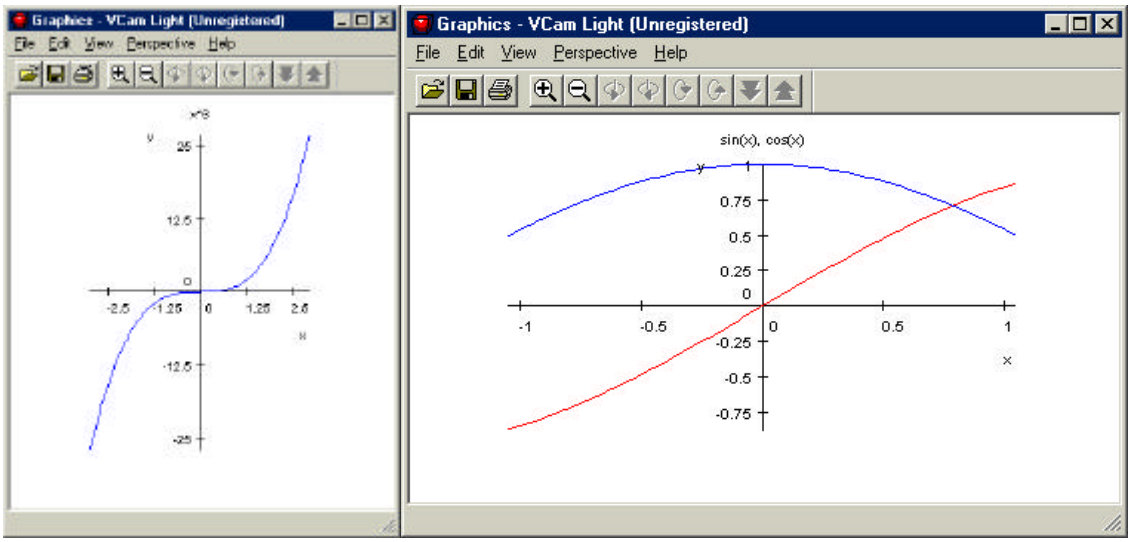
O MuPAD possui ferramentas variadas para renderizar (desenhar) objetos matemáticos em duas ou três dimensões. Existe uma biblioteca para lidar com os casos mais complexos, porém existem comandos padrão para gráficos de funções, talvez os mais desenhados.

Os comandos para traçar gráficos de funções (ou equações) são basicamente 4: **plotfunc2d**, **plotfunc3d**, **plot2d** e **plot3d**. Os gráficos geralmente aparecem em uma janela separada, para facilitar a visualização.

```
>> plotfunc2d( x^3, x=-3..3); #Gráfico da função y=x^3, no
intervalo de x=-3 até x=3.#
```

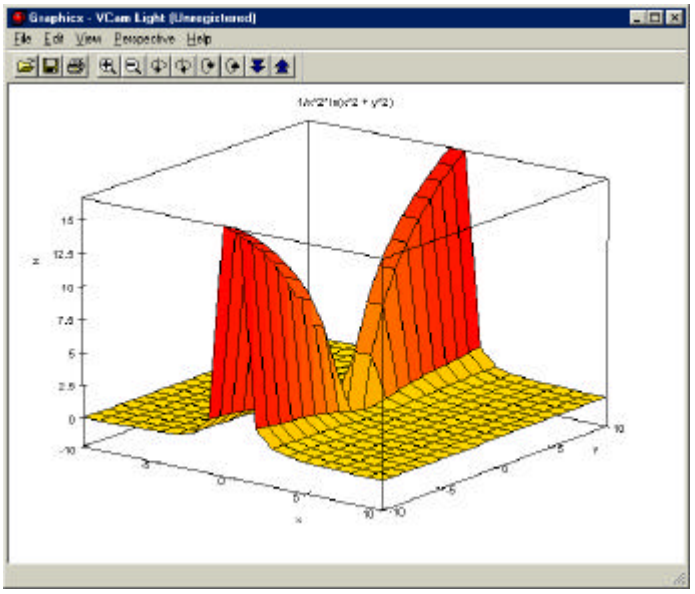
É possível plotar vários gráficos juntos:

```
>> plotfunc2d( sin(x), cos(x), tan(x), x=-PI, PI); #Gráficos
de seno e cosseno, numa mesma janela#
```

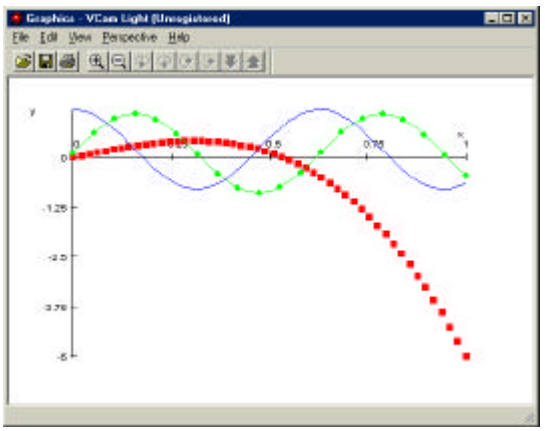


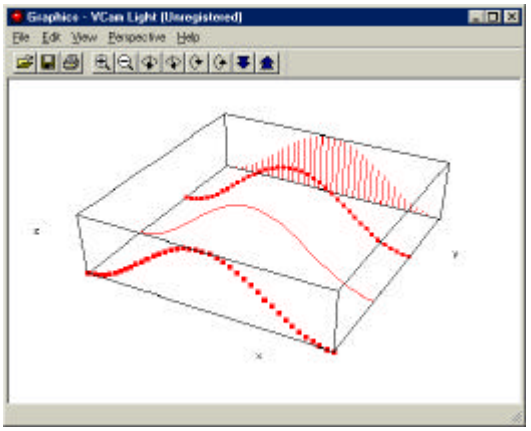
E se a funções tiver duas variáveis..

```
>> plotfunc3d(ln(x^2+y^2)/(x^2), x=-10..10,  y=-10..10)
```



As funções **plotfunc2d/3d** na realidade executam as "verdadeiras" funções de plotagem, **plot2d/plot3d**, passando os parâmetros padrão. Por isso, estas duas funções mostradas só devem ser usadas para gráficos de funções que não necessitem de parâmetros especiais. Porém, usando estes parâmetros (que podem ser vistos digitando **?plot2d** ou **?plot3d**), é possível conseguir gráficos bem interessantes, como os gráficos abaixo:





Para usar a função **plot2d**, é preciso de uma das duas coisas: uma lista de objetos geométricos (como pontos ou polígonos), ou uma parametrização do tipo u -> [x(u), y(u)] da curva que se deseja plotar. Uma função do tipo y=f(x) pode ser parametrizada facilmente pela função x-> [x, f(x)]. O problema com as funções **plot2d/plot3d** é que elas não conseguem lidar com descontinuidades. Abaixo seguem alguns exemplos (comentários em itálico e entre colchetes):

```
>> ponto1:=point(1, 0,  [Este ponto terá coordenadas (1,0)]
Color=RGB::Blue): [O (1,0) será Azul]

>> ponto2:=point(0, 1, Color=RGB::Red): [O ponto (0,1) será vermelho]

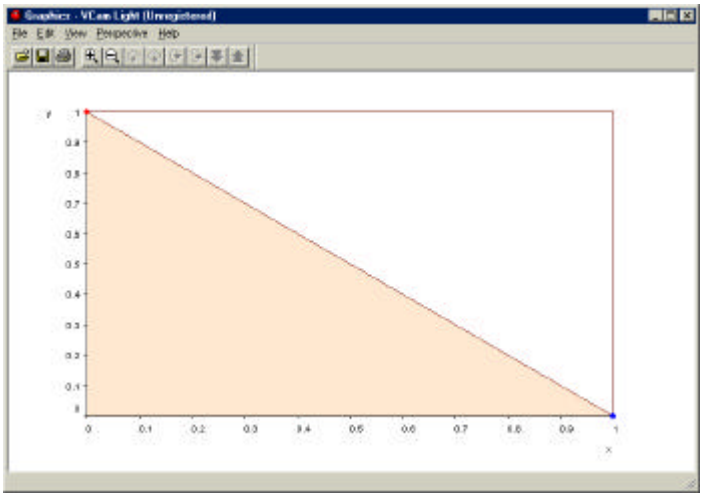
>> contorno:=polygon( [Início da criação de um polígono]
point(1,1), point(0,1), point(1,0), [Vértices do polígono]
Color=RGB::Brown, [Cor do polígono: Marron]
Closed=TRUE [O polígono será fechado]
):

>> triangulo:=polygon(
point(0,0), point(0,1), point(1,0), Closed=TRUE,
Filled=TRUE, [O polígono será preenchido]
Color=RGB::Antique ):

>> Figura:=[Mode=List, [a "Figura" será descrita como uma lista]
[ponto1, ponto2, contorno, triangulo] [Objetos na figura]
]:

>> plot2d(
BackGround=RGB::White,    [Cor de fundo: Branco]
PointWidth=50, [largura dos pontos: 50 pixels]
PointStyle=FilledCircles, [Estilo dos pontos: Círculos preenchidos]
```

```
Figura [Lista dos objetos]
)
```



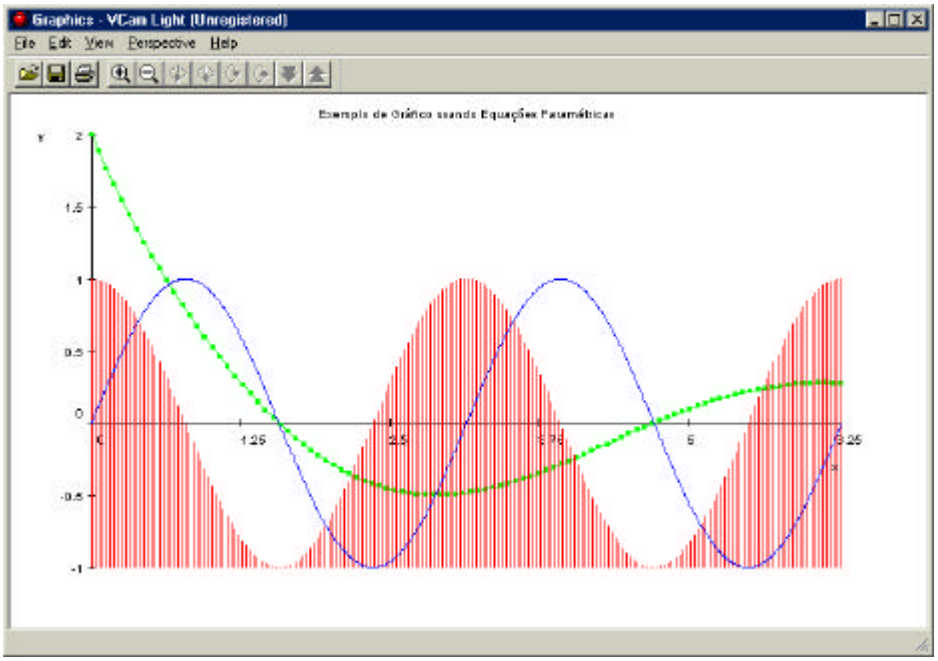
Agora um exemplo do uso de equações paramétricas:

```
>> grafico1:=[Mode=Curve, [Modo de Eqs. Paramétricas]
[u, 2/(u+1)*cos(u)], [Parametrização do gráfico]
u=[0,2*PI], [Intervalo]
Style=[LinesPoints] [Estilo do gráfico: Linhas e Pontos]
];

>> grafico2:=[Mode=Curve, [u, cos(2*u)], u=[0,2*PI],
Style=[Impulses], [Estilo do gráfico: "Impulsos" (Rachura na área abaixo do
gráfico)]
Grid=[200] [definição do gráfico, quanto mais alto mais bem definido, padrão=100]
];

>> grafico3:=[Mode=Curve, [u, sin(2*u)], u=[0,2*PI],
Style=[Lines], [Estilo do gráfico: Somente linhas]
Smoothness=[10] [Suavidade do gráfico: O padrão é 0]
];
```

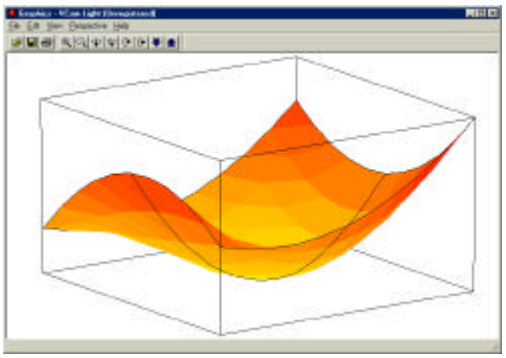
```
>> plot2d(grafico1,grafico2,grafico3, Title="Exemplo de
Gráfico usando Equações Paramétricas")
```
[Plota todos os gráficos, adicionando um título]



É possível desenhar uma superfície no MuPAD entrando com informações sobre as coordenadas de alguns de seus pontos com o comando **matrixplot**, que pertence à biblioteca **plot**. Para tal, basta definir uma matriz tal que o elemento na posição aij é o valor de z para (x,y)=(i,j), ou seja, se o primeiro elemento é 3, então o ponto (1,1,3) pertence à superfície. Abaixo segue um exemplo:

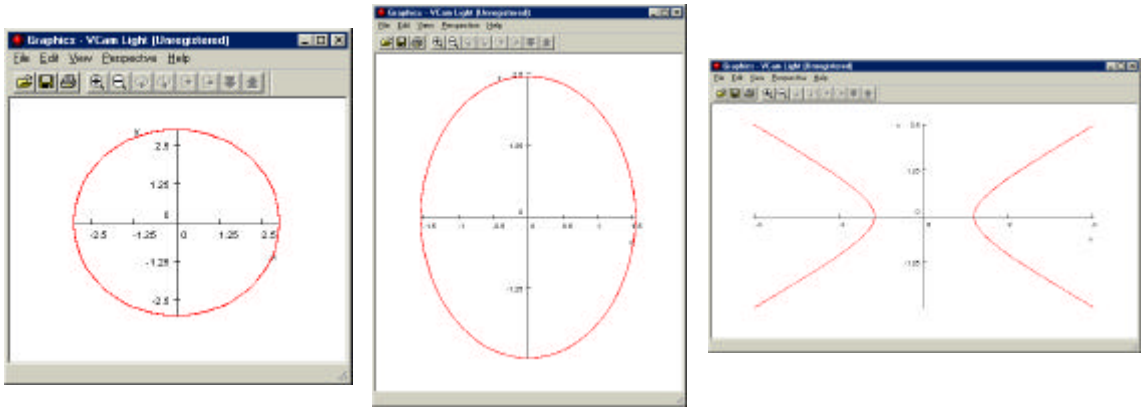
```
>> S:=matrix([
            [3,4,2],
            [3,1,2],
            [5,3,4]
           ]): #Cria a
matriz de coordenadas#

>> plot(plot::matrixplot(S));
```



A plotagem de gráficos de equações pode ser feita com o comando **implicit**, da biblioteca plot, que é voltado para trabalhar com funções implícitas. Se for usada uma expressão ao invés uma equação, o MuPAD transforma esta expressão em uma equação, igualando-a a zero. Os pontos pertencentes à estes gráficos são resolvidos usando métodos numéricos, portanto, o valor da variável **DIGITS** pode incluenciar a qualidade dos desenhos.

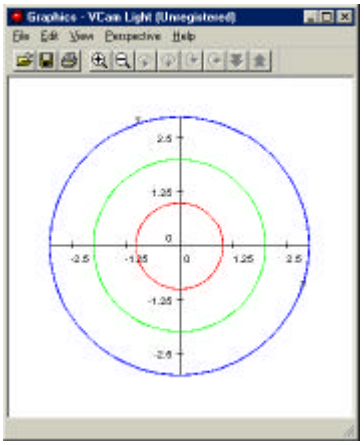
```
>> plot( plot::implicit( x^2+y^2=9, x=-4..4, y=-4..4))
>> plot( plot::implicit( 5*x^2+2*y^2-12, x=-4..4, y=-4..4))
>> plot( plot::implicit( 5*x^2-12*y^2-7, x=-4..4, y=-4..4))
```

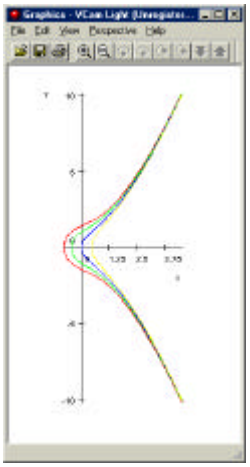


Os recursos do implicit podem ser usados para plotar uma família de equações (funções), usando a opção *Contours*. Seu uso é simples: **plot::implicit(f, x=a..b, y=c..d, Contours=[c_1, c_2, ..., c_n])** plota as equações f=c_1, f=c_2, ..., f=c_n juntas. Por exemplo, abaixo estão plotados os gráficos de $x^2+y^2=1$, $x^2+y^2=4$ e $x^2+y^2=9$ e de uma família de equações elípticas.

```
>> plot( plot::implicit( x^2+y^2, x=-10..10, y=-10..10,
Contours=[1,4,9]))

>> plot( plot::implicit( x^3+2*x+2-y^2, x=-10..10, y=-10..10,
Contours=[0,1,2,3]))
```

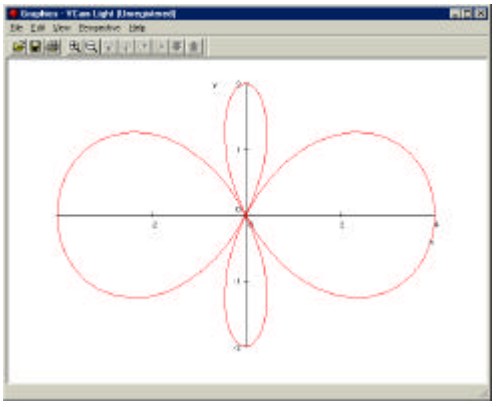


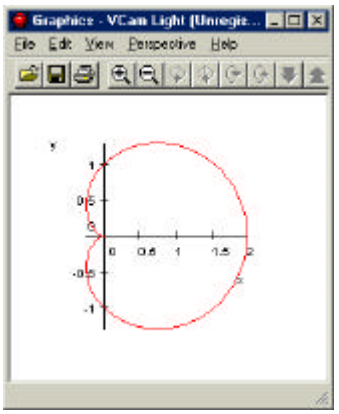


Pode-se ainda plotar gráficos usando outros tipos de coordenadas, como coordenadas polares, esféricas ou cilíndricas.

Para coordenadas polares, o comando (com a sintaxe) é o seguinte: **polar([r,t], t=[a,b])**, onde r é uma f(t), como nos exemplos abaixo:

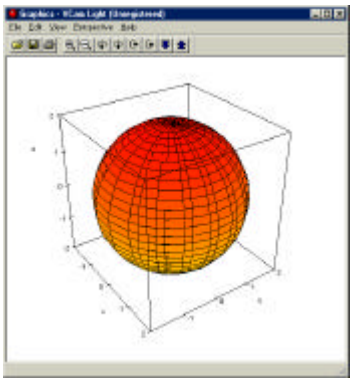
```
>> plot( plot::polar( [1+3*cos(2*t), t], t=[0, 2*PI]))
>> plot( plot::polar( [1+cos(t), t], t=[0, 2*PI]))
```



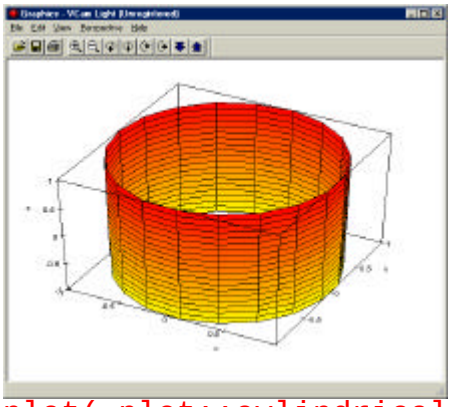


As coordenadas esféricas são definidas a partir das coordenadas retangulares pela transformação (x,y,z,)->r(*cos*Ö*sin*è, *sin*Ö*sin*è, *cos*è), sendo Ö e è os ângulos que o vetor (x,y,z) faz ccom os eixos x e y, respectivamente. Para gráficos usando coordenadas esféricas, o comando é **spherical([r, Ö, è], u=a..b, v=c..d))**, com r, Ö e è funções de u e v.

Para coordenadas cilíndricas, o comando é o **cylindrical** e a sintaxe é a mesma do anterior. Seguem exemplos de gráficos plotados nestes sistemas de coordenadas:



```
>> plot( plot::spherical( [2, u, v], u=0..3*PI, v=0..PI) )
```



```
>> plot( plot::cylindrical( [1, u, v], u = -PI..PI,  v = -1..1))
```

Cálculo Diferencial e Integral

O MuPAD calcula limites, derivadas e integrais, podendo ser uma ferramenta útil para trabalhar conceitos de Cálculo.

Para calcular um limite no MuPAD, a sintaxe é **limit(funcao, x=ponto, direcao)**, onde "direção" é Left para limite a esquerda, Right para limite a direita, e se omitida, supõe-se o limite bidirecional. Por exemplo:

```
>> f:=x->1/(x-1): #Uma função descontínua em x=1#

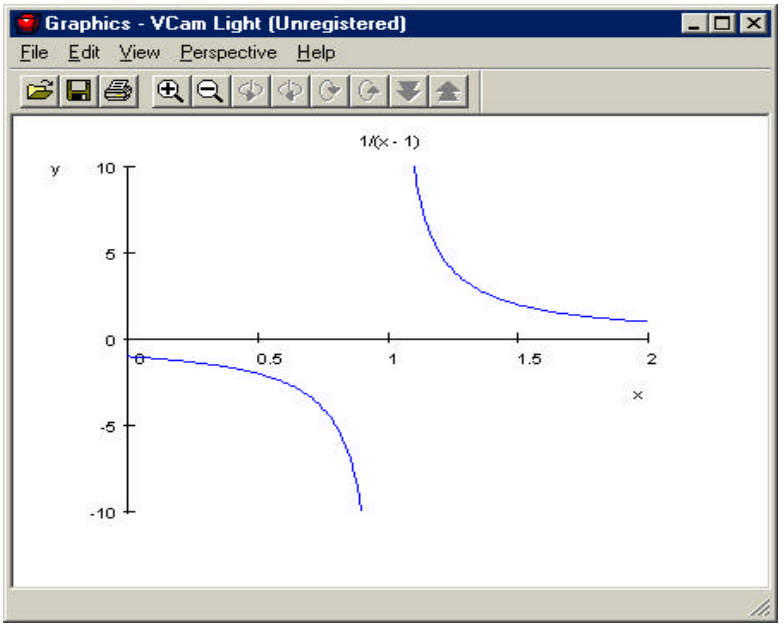
>> limit( f(x), x=1, Left); #O limite tendendo à esquerda de
x=1#
                          -infinity

>> limit( f(x), x=1, Right); #O limite tendendo à direita de
x=1#
                           infinity

>> limit( f(x), x=1); #Como os dois limites acima são
diferentes, o limite bilateral não existe!#
                          undefined
```

Para conferir as respostas graficamente, é possível plotar o gráfico da função acima definida:

```
>> funcplot2d( f(x), x=0..2 );
```



A derivada de uma função pode ser calculada algebricamente no MuPAD, com o uso do comando **diff**, como na maioria dos outros softwares matemáticos:

```
>> f:=x->x^2*sin(x):

>> diff( f(x) , x ); # Deriva a função f em relação à
variável x#
                                  2
                    2 x sin(x) + x  cos(x)
```

Ou então de uma maneira idêntica à notação de Newton para derivadas:

```
>> f'(x)
                                  2
                    2 x sin(x) + x  cos(x)
```

```
>> f''(x)
                                        2
                2 sin(x) + 4 x cos(x) - x  sin(x)
```

Com a derivada é possível determinar retas tangentes a gráficos de funções, já que o valor da derivada de uma função em um ponto x_0 determina a inclinação da reta tangente à curva naquele ponto. Veja abaixo:

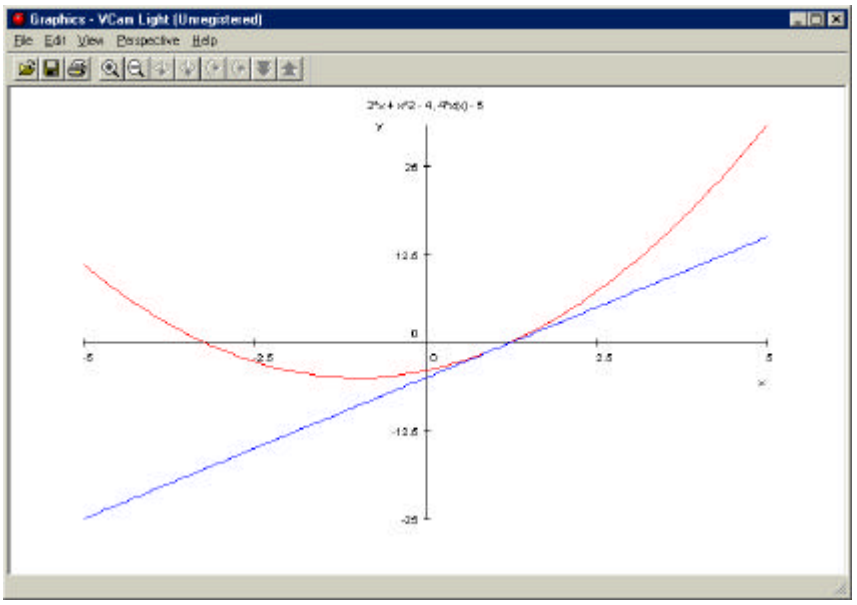
```
>> g:=x->x^2+2*x-4: #Uma função polinomial#

>> ponto:=1: #O ponto pelo qual será traçada a tangente#

>> inclinacao:=g'(ponto): #Definimos inclinacao como sendo a
derivada no ponto#

>> reta_tangente:= inclinacao*(x-ponto)+f(ponto); #Aqui a
equacao da reta tangente é criada#
                            4 x - 5

>> plotfunc2d( g(x), reta_tangente(x), x=-5..5); #Plotando os
dois gráficos juntos#
```



A integração é bem semelhante à derivação (a função f é a mesma de cima, **$f(x)=x^2\sin(x)$**)

```
>> int( f'(x), x)
                                        2
                  2 sin(x) + sin(x) (x  - 2)
```

A resposta está correta, apesar de diferente da esperada. Uma simplificação resolve o problema:

```
>> factor(%)
                                     2
                                sin(x) x
```

A integração numérica é feita adicionando um intervalo de integração ao lado da variável de integração:

```
>> int(ln(x), x=1..4)
                           4 ln(4) - 3
```

Assim, fica sendo possível calcular áreas sob gráficos, como segue:

```
>> h:=x->ln(x): #Define uma função h(x)=ln(x)#

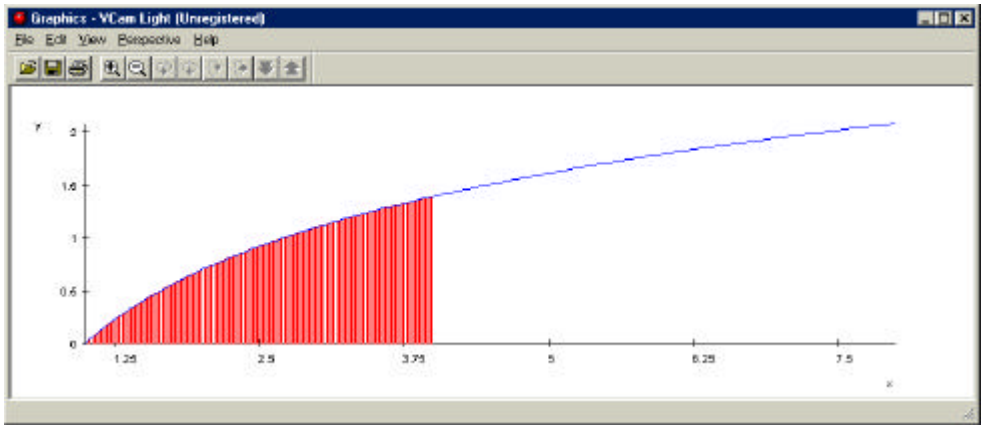
>> inferior:=1: #Limite inferior#

>> superior:=4.0: #Limite superior#
```

```
>> grafico:=[Mode=Curve, [u, h(u)], u=[inferior,superior],
Style=[Impulses], Grid=[200]]: #O gráfico da função,
rachurado#

>> linha:=[Mode=Curve, [u, h(u)], u=[inferior,2*superior],
Grid=[200]]: #O traço do gráfico#

>> plot2d(grafico,linha,RealValuesOnly = TRUE );
```




```
>> int( h(x), x=inferior..superior); #A área da parte
rachurada acima#
                              2.545177444
```

É possível calcular expansões em séries no MuPAD com o comando **taylor**. A sintaxe é **taylor(Função, x=centro_do_intervalo, maior_potência)**. Veja abaixo:

```
>> taylor(sin(x), x=0, 10)

                    3     5     7       9
                   x     x     x       x          11
               x - -- + --- - ---- + ------ + O(x  )
                   6    120   5040   362880
```

ÁLGEBRA LINEAR

Uma matriz é uma tabela de números (ou objetos). Apesar de matrizes e vetores poderem ser criados como arrays, desta forma é preciso definir as operações de adição e multiplicação destes. Para evitar isto, o MuPAD existe um comando especial para criar matrizes. Funções mais avançadas para tratar matrizes podem ser acionados com na biblioteca linalg, específica para tratar de Álgebra Linear.

Para criar uma matriz, o comando é **matrix(m,n)**, onde m é o número de linhas da matriz e n é o número de colunas. Se as entradas da matriz não forem especificadas, o MuPAD cria uma matriz de zeros.

```
>> A:=matrix(3,3)
```

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

```
>> B:=matrix(2,2,[ [1,2], [3,4]]); #Criando uma matriz especificando as entradas#
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

```
>> B:=matrix(3,3,8,Diagonal); #Cria uma matriz diagonal, com
os elementos da diagonal iguais a 8, ou seja, 8*Id#
                        +-         -+
                        |  8, 0, 0  |
                        |           |
                        |  0, 8, 0  |
                        |           |
                        |  0, 0, 8  |
                        +-         -+
```

Para operar com matrizes, os comandos são os mesmo usados para se trabalhar com números.

```
>>  C:=matrix(2,2, [[6,7], [9,9]])
                          +-      -+
                          |  6, 7  |
                          |        |
                          |  9, 9  |
                          +-      -+

>> B+C, B*C, B*C+B^2; #Algumas operações com matrizes#
          +-        -+  +-        -+  +-        -+
          |   7,  9  |  |  24, 25  |  |  31, 35  |
          |          |, |          |, |          |
          |  12, 13  |  |  54, 57  |  |  69, 79  |
          +-        -+  +-        -+  +-        -+
```

A inversa de uma matriz pode ser obtida do modo mais natural possível:

```
>> inversa:=1/B
```

$$\begin{pmatrix} -2 & 1 \\ 3/2 & -1/2 \end{pmatrix}$$

```
>> inversa*B; #A multiplicação de uma matriz pela sua inversa
resulta numa matriz identidade#
```

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Se a matriz não possuir inversa, uma mensagem de erro (FAIL) aparece.

Para concatenar duas matrizes, o comando é o mesmo para listas, A.B.

Para usar outras funções (mais ligadas às propriedades das matrizes e menos às suas características numéricas) é necessário usar o pacote linalg. Para usá-lo, a sintaxe é **linalg::comando(Matriz)**, como no exemplo abaixo:

```
>> linalg::tr(B); #Retorna o traço (soma dos elementos da
diagonal principal) da matriz B#
```

$$5$$

Abaixo, mais alguns comandos no pacote linalg.

```
>> linalg::det(B); linalg::det(2*B);
                              -2
                              -8

>> linalg::det(a*B)
                                 2
                             -2 a
```

Acima é possível perceber uma propriedade do determinante de uma matriz: ao multiplicarmos uma matriz 2x2 por uma constante *a*, o determinante desta fica multiplicado pela constante elevada ao quadrado.

Seguem mais alguns comandos para trabalhar com matrizes:

```
>> linalg::transpose(C)
                          +-      -+
                          |  6, 9  |
                          |        |
                          |  7, 9  |
                          +-      -+

>> linalg::randomMatrix(2,2,Dom::Integer); #Gera uma matriz
2x2 aleatória com entradas Inteiras; outros tipos de
entradas: Real, Rational, Complex#
                         +-       -+
                         |  4, -7  |
                         |         |
                         |  6, 16  |
                         +-       -+
```

```
>>  linalg::randomMatrix(3,3,Dom::Integer,  -20..20);  #Uma
matriz aleatória com entradas inteiras entre –20 e 20#
```

$$\begin{pmatrix} -10, & 9 \\ -14, & 11 \end{pmatrix}$$

A seguir, um importante teorema da Álgebra Linear será exemplificado com o uso do MuPAD. É o Teorema de Cayley-Hamilton, que nos diz que uma matriz é raiz de seu polinômio característico.

```
>> M:=matrix(2,2,[ [5,7], [4,1] ]): #Cria uma matrix 2x2#

>> Id:=matrix(2,2,[ [1,0], [0,1] ]): #Cria a identidade 2x2#

>> p:=linalg::charpoly(M,x); #Calcula o polinômio caracte-
rístico da matrix M#
```

$$x^2 - 6\,x - 23$$

Agora use uma expressão para ver o que acontece quando “substituímos” a matriz M no polinômio p (considere “23” como 23*Id):

```
>> M^2 - 6*M – 23*Id
```

$$\begin{pmatrix} 0, & 0 \\ 0, & 0 \end{pmatrix}$$

O resultado é uma matriz nula, como previsto pelo teorema.

Para se obter os autovalores e autovetores de uma matriz, existem dois comandos:

```
>> linalg::eigenvalues(M)
                                    1/2     1/2
                        {3 - 4 2   , 4 2    + 3}

>> linalg::eigenvectors(M)
-- --                 -- +-                -+ -- --
|  |                  |  |     1/2          |  |  |
|  |         1/2      |  |  - 2    + 1/2    |  |  |
|  |  3 - 4 2   , 1,  |  |                  |  |  |,
|  |                  |  |        1         |  |  |
-- --                 -- +-                -+ -- --

   --                 -- +-              -+ -- -- --
   |                  |  |   1/2          |  |  |  |
   |     1/2          |  |  2    + 1/2    |  |  |  |
   |  4 2    + 3, 1,  |  |                |  |  |  |
   |                  |  |       1        |  |  |  |
   --                 -- +-              -+ -- -- --
```

A resposta acima está no formato [[autovalor, multiplicidade], [autovetor associado]]. Assim, temos que o autovalor $3-4*2^{1/2}$ tem multiplicidade 1 e seu autovetor associado é $(-2^{1/2}+1/2, 1)$, enquanto o autovetor associado ao autovalor $3+4*2^{1/2}$ é o $(2^{1/2}+1/2, 1)$.

SISTEMAS LINEARES E OTIMIZAÇÃO

Uma das mais importantes aplicações da Álgebra Linear aparece na resolução de sistemas de equações lineares e problemas de otimização (programação matemática). O MuPAD tem recursos para lidar com estes dois problemas de uma forma muito prática e rápida.

Por exemplo, imagine que queremos resolver o sistema abaixo, ou seja, encontrar os valores de x, y e z que satisfaçam todas as equações:

$$\begin{aligned} x + y + z &= 10 \\ 2x + y + z &= 3 \\ x - 2y + 4z &= 11 \end{aligned}$$

No MuPAD, o procedimento seria o seguinte:

```
>> equacoes:={2*x+y+z=3, x+y+z=10, x-2*y+4*z=11}; #Faz a
variável equacoes receber o sistema#
   {x + y + z = 10, 2 x + y + z = 3, x - 2 y + 4 z = 11}

>> solucao:=solve(equacoes); #A variável solucao recebe a
solução do sistema#
              {[x = -7, y = 25/3, z = 26/3]}

>> x, y, z
                          x, y, z
```

Apesar do sistema ter sido resolvido, a variável x não assumiu nenhum valor. Para atribuir às variáveis envolvidas seu valor encontrado, precisamos usar um outro comando:

```
>> assign(op(solucao))
                [x = -7, y = 25/3, z = 26/3]

>> x,y,z
                          -7, 25/3, 26/3
```

O comando **solve** é universal para se obter "soluções". O MuPAD analisa o objeto que se deseja "resolver" e decide qual método utilizar. Assim, ele é usado para resolver quase tudo no MuPAD. Abaixo um exemplo de uso do MuPAD para encontrar raízes de um polinômio:

```
>> solve(18*x^2-27*x+26*x^3+9*x^4+x^5-27 , x); # Resolve o
polinômio em relação à variável x, ou seja, encontra as
raízes do polinômio#
                          {-3, -1, 1}
```

Para alguns polinômios de grau maior que 5, é impossível exprimir suas raízes em termos de radicais, então o MuPAD mostrará uma resposta parecida com o caso abaixo:

```
>> solve(x^6+x^2+x^3+x,x)
                                  2     3     4
       {-1, 0} union RootOf(X2  - X2  + X2  + 1, X2)
```

O que a resposta acima diz é o seguinte: as raízes de x^6+2^x+x^3+x são –1, 0, além das raízes do polinômio x^2-x^3+x^4+1. Mas quais são estas raízes? Elas podem ser obtidas (numericamente) usando o comando float:

```
>> float(%)
    {-1.0, 0.0} union {- 0.3518079518 + 0.7203417364 I,
- 0.3518079518 - 0.7203417364 I, 0.8518079518 - 0.911292162
                I, 0.8518079518 + 0.911292162 I}
```

Para resolução de problemas que procuram o valor máximo assumido por uma função cujas variáveis devem satisfazer um sistema de inequações lineares, o MuPAD possui implementado o algoritmo Simplex, que é o método mais rápido e eficaz para resolver este tipo de problema. Por exemplo, considere o seguinte problema:

Maximizar Z = 4x + 3y
Sujeito a: 3x + 2y 6
2x + 2y 4
x - y -1
x, y 0

No problema acima, deve-se obter o valor máximo da função Z(x,y)=4x+3y, dado que as variáveis x e y devem obedecer às restrições dadas, além de serem positivas.

Este problema ficaria assim no MuPAD:

```
>> rest:=[ {3*x+2*y<=6, 2*x+2*y>=4, y-x<=1, x>=0, y>=0},
4*x+3*y ]; # Armazenando as equações na variável rest#
 [{0 <=x, 0<=y, y - x <=1, 4<=2x + 2y, 3x + 2y<=6}, 4x + 3y]

>> linopt::maximize(rest); # Agora o MuPAD resolve o
problema, analisando o conteúdo da variável rest#
             [OPTIMAL, {x = 4/5, y = 9/5}, 43/5]
```

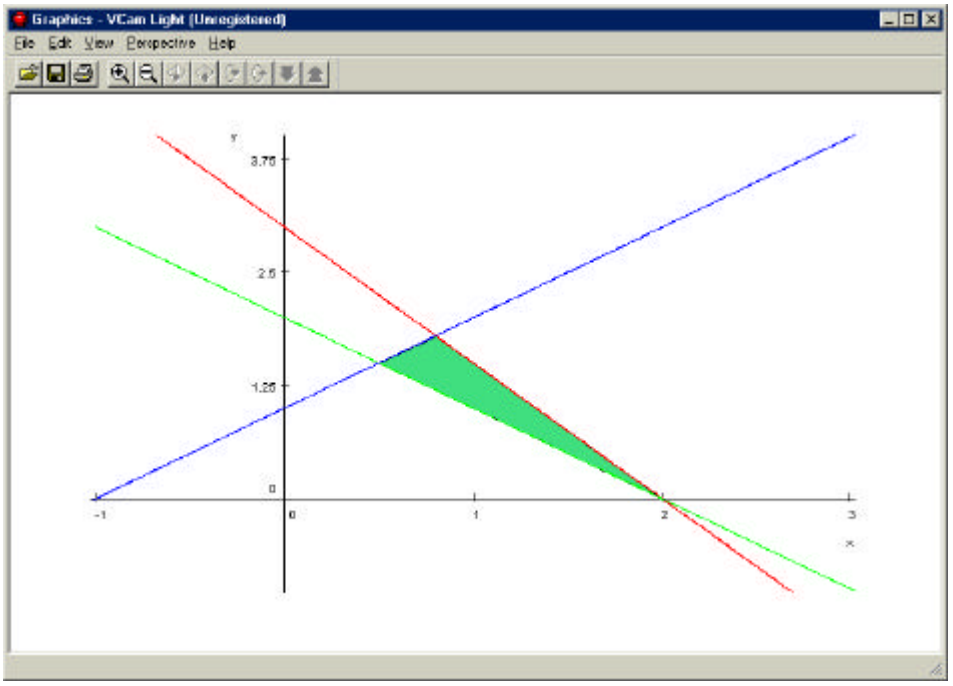
Acima, a resposta nos diz que o MuPAD encontrou uma solução ótima no ponto (4/5, 9/5), e este ponto, quando substituído na função objetivo, resulta num valor máximo de 43/5 para o problema.

Se traçarmos os gráficos das equações acima, teremos a figura abaixo, onde está rachurado de verde a região dos pontos viáveis, ou seja, pontos que são satisfazem o sistema de inequações. Para isso foram usados os comandos abaixo:

```
>> eqs:=plot::implicit( [3*x+2*y-6,2*x+2*y-4, y-x-1], x=-
1..4, y=-1..4): #Cria uma imagem com as equações#

>> regiao:=plot::Polygon( plot::Point(1/2,3/2),
plot::Point(2,0), plot::Point(4/5, 9/5), Closed=TRUE,
Filled=TRUE, Color=[0.254, 0.874, 0.5]): #Cria a região
rachurada#

>> plot(regiao, eqs); #Plota tudo#
```



A biblioteca linopt, específica para lidar com Otimização Linear (inclusive Programação Linear Inteira, usando o Algoritmo de Lang-Doing), possui vários recursos interessantes. Para maiores detalhes, digite **?linopt**.

TEORIA DOS NÚMEROS

O MuPAD possui ferramentas para lidar com problemas da Teoria dos Números, que, em sua maioria, pertencem à biblioteca **numlib**.

Para usar as funções relacionadas com Teoria dos Números, é preciso fazer uma chamada à biblioteca **numlib**. Abaixo, seguem algumas das funções deste pacote. O restante pode ser acionado com o comando **?numlib**.

Para aproximar um número real por frações contínuas não é necessário o uso da biblioteca numlib (existe o comando **contfrac**, que tem a mesma função). Porém, aqui ela será usada.

```
>> numlib::contfrac(1.325325,4); #Transforma o número real 1,325 em uma fração contínua com precisão de 4 casas decimais#
```

$$1 + \cfrac{1}{3 + \cfrac{1}{13 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \ldots}}}}}$$

Para transformarmos uma fração não-inteira em um número real, podemos usar o comando **decimal**:

```
>> numlib::decimal(1/7)
                    0, [1, 4, 2, 8, 5, 7]
```

O resultado diz que 1/7 é uma dízima periódica, em que a parte periódica é o número 142857.

Abaixo estão algumas funções relacionadas com divisores:

```
>> numlib::divisors(496); #Retorna os divisores de um
inteiro; no caso, 496#
                  [1, 2, 4, 8, 16, 31, 62, 124, 248, 496]

>> numlib::numdivisors(496); #O número de divisores de 496#
                                    10

>> numlib::tau(496); #Esta função faz o mesmo que a anterior#
                                    10

>> numlib::primedivisors(496); #Retorna os divisores primos
de 496#
                                  [2, 31]

>> numlib::numprimedivisors(496); #Retorna o número de
divisores primos de 496#
                                    2
```

Para fatorar um inteiro, a função **ifactor** já foi mencionada. Ela usa algoritmos convencionais para fatoração. Para encontrar um fator de um inteiro usando o método das curvas elípticas (mais eficaz, em alguns casos), existe a função **ecm**.

```
>> a:=2^(40^2)+1: #A variável “a” é um número muito grande#

>> numlib::ecm(a); #A função “ecm” encontra um fator de “a” em menos de 1 segundo#
                                364801
```

Para testar a primalidade de números muito grandes, o método das curvas elípticas é bem mais eficaz que outros métodos. Em alguns casos, o comando **isprime** (que usa o teste de Miller-Rabin) pode retornar que um dado número composto é primo. Isto se deve ao fato dos testes de primalidade serem probabilisticos, ou seja, se for retornado que o número não é primo, é por que o MuPAD encontrou um fator e neste caso o teste é 100% confiável. Porém, no caso de nenhum fator ser encontrado (talvez por falha de processamento, ou pouca memória), o número é considerado primo.

O comando **proveprime** (da biblioteca numlib) usa o método das curvas elípticas para testar a primalidade, portanto, sempre retorna uma resposta verdadeira. O problema é que ele roda bem mais lentamente que o isprime.

```
>> a:=2^(2^13)+1: #Um número grande#

>> numlib::proveprime(a);
                                FALSE
```

As funções abaixo são para cálculo de máximo/mínimo divisor comum:

```
>> igcd(4,3); #Retorna o Máximo Divisor Comum (Greatest Commom Divisor) de dois inteiros #
                                  1
>> lcm(4,3); #Retorna o Mínimo Múltiplo Comum (Least Commom Multiple)#
                                  12
```

A função abaixo é uma implementação do Algoritmo Extendido de Euclides, que, dados n inteiros, combina-os linearmente numa expressão equivalente ao seu MDC. Por exemplo:

```
>> A:= 15,69,21,40: #Uma sequência de números#
```

```
>> b:=numlib::igcdmult(A); #Determina o MDC da sequência A e os coeficientes dos elementos de A para a combinação#
                                [1, 117, -26, 0, 1]
```

A resposta acima nos diz que o MDC dos elementos do conjunto A é 1, e pode ser obtido fazendo a conta: 117*15 + (-26)*69 + (0)*21 + (1)*40 = 1.

A função **Phi** de Euler, que, aplicada no inteiro n, retorna a quantidade de primos com n menores que ele.

```
>> numlib::phi(1548); #A função Phi de Euler para o inteiro 1548#
                                504
```

```
>> numlib::phi(457); #Se aplicada num número primo p, Phi(p)=p-1#
                                456
```

A função **sigma**, que retorna a soma dos divisores de um dado inteiro (inclusive 1 e o próprio número):

```
>> numlib::sigma(45)
                                78
```

```
>> numlib::sigma(496)-496
                              496
```

Acima, uma curiosidade: A soma dos divisores de 496 (excluíndo-se o 496) é igual ao próprio 496. Por isso dizemos que 496 é um número perfeito. Outros números perfeitos são 6 e 28.

Para cálculo de números de Fibonnacci:

```
>> numlib::fibonacci(141)
                  131151201344081895336534324866
```

É sempre possível escrever um dado número racional em sua forma expandida na base 10, por exemplo, 1234 = 4*10^0 + 3*10^1 + 2*10^2 + 1*10^3. Esta seria a representação g-ádica de 1234, com g=10. Para representar um inteiro em sua forma g ádica, existe um comando no MuPAD, que retorna uma lista com os coeficientes a serem usados, da menor para a maior potência:

```
>> numlib::g_adic(300,10); numlib::g_adic(156,7);
                            [0, 0, 3]
                            [2, 1, 3]
```

O resultado acima nos diz que $300=0*10^0+0*10^1+3*10^2$ e que $156=2*7^0 + 1*7 + 3*7^2$.

Um resultado importante para os inteiros também pode ser usado no MuPAD: o Algoritmo Chinês do Resto, que é usado para resolver sistemas como o abaixo:

$$x \equiv 3 \pmod 7$$
$$x \equiv 2 \pmod 4$$
$$x \equiv 1 \pmod 3$$

No MuPAD:

```
>> numlib::ichrem( [3,2,1], [7,4,3]); #Encontrando a solução
do sistema acima#
                              10
```

BASES DE GRÖBNER

O MuPAD vem com o pacote groebner, com algumas funções para lidar com anéis de polinômios em várias indeterminadas sobre um corpo, em particular com a possibilidade de se calcular as bases de Gröbner para um dado ideal. As ordens monomiais lexicográfica, lexicográfica graduada e lexicográfica reversa graduada podem ser usadas.

O S-polinômio, usado no algoritmo de Buchberger, pode ser encontrado com o comando abaixo:

```
>> groebner::spoly(x^2*y+3*y,x*y^3+2*x)
                                   3      2
                                3 y  - 2 x
```

Ainda é possível especificar qual ordem se deseja usar (LexOrder, DegreeOrder, DegInvLexOrder – esta última é a padrão), com uma terceira opção:

```
>> groebner::spoly(x^2*y+3*y,x*y^3+2*x, LexOrder)
                                   3      2
                                3 y  - 2 x
```

Agora a função mais importante do pacote, que calcula uma base de Gröbner reduzida para o ideal gerado pelos polinômios na lista especificada: **gbasis**.

```
>> p:=3*xyz^2+z+x*y:

>> q:=x^2*y+2*y^2*z:

>> groebner::gbasis( [p,q],LexOrder);
                      3    2        2    2                  2
              [2 x y  - x  y + 6 y  xyz , z + x y + 3 xyz ]
```

PROGRAMAÇÃO

Nesta seção serão abordadas as funções básicas para programação no MuPAD. Sua linguagem de programação é bem semelhante ao Pascal, inclusive as sintaxes.

Para imprimir um objeto na tela pode-se usar o comando **print**. Sua sintaxe é a seguinte: **print(Opções, Objeto)**. As opções podem ser **Unquoted** (para exibir o texto fora de aspas) e **NoNL**, para suprimir uma linha adicional no final do texto.

```
>> print(4^3)
                                64

>>print("4^3"); #As aspas evitam a execução da operação#
                              "4^3"

>>print(Unquoted, "4^3"); #Tirando as aspas da exibição#
                               4^3

>> print("Texto com espaço"); #Para usar espaço, é necessário
usar aspas#
                      "Texto com espaço"

>> print(Unquoted, "Texto com espaço\tTabulação"); #Um \t
entre palavras insere 8 espaços entre estas#
              Texto com espaço        Tabulação

>> print(Unquoted, "Nova\nlinha"); #O código \n insere uma
linha nova#
                               Nova
                              Linha
```

Para misturar expressões com o texto, existe o comando **expr2text**, que realiza esta conversão. Assim, é possível usá-lo, juntamente com o **print**, para produzir saídas num formato bem legível. Para concatenar expressões, novamente, o operador é o **.**:

```
>> print(NoNL, Unquoted, "Valor da tangente de Pi/5:
".expr2text(float(tan(PI/5))))
Valor da tangente de Pi/5: 0.726542528
```

Em um algoritmo, algumas operações precisam ser realizadas várias vezes. Para isso, existem os laços (ou loops). Um destes laços é o **for**, que é usada principalmente para executar tarefas em que o número de passos é bem conhecido. Seu uso é simples e pode ser feito de duas maneiras:

for i **from** <inicio> **to** <final> [**step** <tamanho do passo>] **do**
<operações>
end_for
_for(i, <inicio>, <final>, <tamanho do passo>, <operações>**)**

O que ele faz é o seguinte: Para a variável i indo do valor <inicio> até o valor <final>, faça <operações> e termine o laço. O tamanho do passo é o valor do incremento que será dado à variável i em cada execução do laço. Se for necessária uma contagem regressiva (caso em que o valor inicial da variável i é maior que o valor final), ao invés de *to*, deve-se usar *downto*. Veja alguns exemplos:

```
>> soma:=0: #A variável soma recebe o valor 0#
>> for i from 1 to 10 do soma:=soma+i; print(soma) end_for
                                  1
                                  3
                                  6
                                 10
                                 15
```

Acima, a variável **soma** recebe seu valor atual somado ao valor da variável **i**, a cada vez que o laço é executado, e isto é feito 5 vezes, ou seja, **i** vai de 1 até 5. A cada vez, o valor de **soma** é impresso.

Abaixo, a variável soma inicia valendo 0 e é incrementada pelo seu valor mais o valor da variável i, que assume os valores "de 10 até 1, indo de 2 em 2", ou seja, {10, 8, 6, 4, 2}.

```
>> soma:=0:
>> for i from 10 downto 1 step 2 do soma:=soma+i; print(soma)
end_for
                                10
                                18
                                24
                                28
                                30
```

Agora um exemplo mais complexo: o cálculo da soma de todos os números pares de 3 algarismos (ou seja, de 100 até 998).

```
>> soma:=0: for i from 100 to 999 step 2 do soma:=soma+i
end_for
                              247050
```

O próximo exemplo usa o Método de Newton para o cálculo de raizes quadradas. Suponha que se queira calcular a raiz quadrada de um número *a*. Então, a partir de um valor r_0, a raiz quadrada de *a* é o limite da sequência r_k, onde $r_k=(r_{k-1}+a/r_{k-1})/2$, k=1, 2, 3, ... Vamos programar isso no MuPAD e encontrar algumas raízes, começando pela raíz de 117.5. Assim, a=117.5, e r_0=5 (um valor qualquer, de preferência próximo da raiz desejada).

```
>> a:=117.5:
>> r[0]:=5:
>> for k from 1 to 5 do r[k]:=(r[k-1]+a/r[k-1])/2 end_for;
>> sqrt(a)
                  10.839741694339693952 59
                  10.839741694339399709 49
```

O primeiro número acima é o valor dado pelo laço for, e o segundo é o valor dado pela função **sqrt**. Note que com apenas 5 passos, o erro obtido foi menor que 10^{-12}! Se o valor inicial r_0 for mais próximo da raiz real, é possível obter um resultado ainda mais preciso.

É possível usar o laço **for** com objetos. No exemplo abaixo, após definida uma lista com 3 funções, para cada uma delas, será calculada a integral definida de x=1 até 10:

```
>> Funcoes:=[sin(x), x^2, log(2,x)]:
>> for i in Funcoes do print(float(int(i, x=1..10))) end_for
                        1.379373835
                           333.0
                        20.23502558
```

Outro comando semelhante ao **for** é o **while**. Porém, o **while** executa algumas operações até que alguma condição seja satisfeita. Por exemplo:

```
>> i := 1:
>> s := 0:
>> while i < 10 do
>> s := s + i;
>> i := i + 1;
>> end_while
>> soma;
                              45
```

Acima, enquanto o valor de i for menor que 10, a variável s será acrescida de seu valor atual somado ao valor de i, e a variável i será incrementada em uma unidade. Assim, o que a função acima faz é soma os inteiros de 1 até 9. Se tivéssemos usado .. *while i<=10* .. o resultado seria 55, pois o 10 também seria somado.

Se for preciso executar alguma operação baseado no valor de uma variável e/ou condição, é necessário usar o comando **if**. Sua sintaxe é: **if** <condição> **then** <operação> **elif** <condição2> **then** <operação2> ... **else** <operaçãoN> **end_if**. Abaixo, um exemplo:

```
>> if float(sqrt(2))<float(PI)
>>  then
>>     print(Unquoted, "Pi é maior que raiz de dois")
>>  else
>>     print(Unquoted, "Raiz de 2 é maior do que Pi")
>> end_if
                Pi é maior que raiz de dois
```

Acima, é realizado o teste: ($2 < \pi$) . Caso seja verdade, então é retornado “Pi é maior que raiz de dois”. Caso contrário, “Raiz de 2 é maior do que Pi”.

É possível usar o comando **if** juntamente com o **for**. Abaixo segue um exemplo, onde é calculada a soma de todos os números primos menores que 1000.

```
>> soma:=0:
>> for i from 1 to 999 do
>>   if isprime(i)=TRUE
>>     then soma:=soma+i
>>   end_if
>> end_for

>> soma
                              76127
```

Alguns algoritmos precisam de recorrer sempre a alguns comandos. Para isso, existem os procedimentos, que são semelhantes a sub-programas possíveis de serem criados pelo usuário. Eles já foram mencionados neste tutorial, na seção sobre Funções. Agora serão descritas outras maneiras de se criar um procedimento, usando as funções **for** e **if**.

A sintaxe para se criar um procedimento é Nome:=**proc(**parametros**)** **begin** <instrucoes> **end_proc**. Por exemplo, para criar um procedimento que visa somar todos os primos menores ou iguais que um dado número n, pode-se proceder como abaixo:

```
>> somap:=proc(n)
>> begin
>> soma:=0:
>>   for i from 1 to n do
>>     if isprime(i)=TRUE
>>       then soma:=soma+i
>>     end_if
>>   end_for:
>> print(soma)
>> end_proc
```

Para usar o procedimento **somap** é fácil:

```
>> somap(10)
                                17

>> somap(100)
                               1060
```

Já o procedimento abaixo cria uma rotina para se calcular a integral e a derivada de uma dada função:

```
>> fi:=proc(f)
>> begin
>>   print(Unquoted,"Derivada= ".expr2text(diff(f,x)));
>>   print(Unquoted,"Integral= ".expr2text(int(f,x)));
>> end_proc

>> fi(ln(x))
                        Derivada= 1/x
                    Integral= x*ln(x) - x

>> fi(x^2)
                        Derivada= 2*x
                      Integral= 1/3*x^3
```

BIBLIOGRAFIA

[1] OEVEL, W., WEHMEIER, GERHARD, J., The MuPAD Tutorial, Paderborn, 2002.

[2] MOURA, A. O., SANTOS, L. M. R., Introdução ao Maple, Viçosa, 2001.

[3] BRANDÃO, L. O., WATANABE, C. J., Uma Introdução ao MuPAD, São Paulo, 1997.

[4] VAZ, C. L. D., Aprendendo MuPAD, Labmac, UFPA, 2001.