Universidade da Madeira

DEPARTAMENTO DE
Matemática

Comparação de Sistemas de Programação

José Marcelino Caires Fernandes Temtem

Dissertação para a obtenção do grau de
Mestre em Matemática para o Ensino

Orientador: Professor Doutor F. Miguel Dionísio

Funchal – Madeira
Novembro de 2001

COMPARAÇÃO DE SISTEMAS

DE

PROGRAMAÇÃO


Derive 5


Maple 6


Mathematica 4.0

Funchal – Madeira
Novembro de 2001

COMPARING PROGRAMMING SYSTEMS

Summary:

This work compares the solutions presented by the systems *Derive5.0, Maple 6* and *Mathematica 4.0* for mathematical problems found in undergraduate and graduate studies. We tried to identify their differences as well as the points where everything works in a similar way.

The subjects compared in this work are: the numeric and symbolic calculus, the programming language and graphics capabilities.

We begin by presenting basic knowledge about every program. We handle numeric calculus and in particular, some functions from Number Theory.

We analyse several ways to manipulate lists and their elements and some areas of mathematical analysis such as equations, derivatives and integral calculus comprehending numerical and symbolic calculus. We also examine matrix and polynomial operations.

We analyse recursive, imperative and functional programming as well as rewriting rules. We give the user the most important constructions as well as the information about the way each type of programming works so that each person becomes able to solve the proposed problems.

The graphics studied in this work are those in one and two variables. These are the most common in schools and universities. We show how easily and quickly graphics are made. This is more evident when the graphics depend of two variables. We also show that animation, when possible, is very useful.

In this work we conclude that the system *Mathematica* is better concerning programming, *Maple* as a superior capacity to represent graphics and *Derive* allows us to have a simpler first contact and learn more easily the language.

COMPARAÇÃO DE SISTEMAS DE PROGRAMAÇÃO

Resumo:

Este trabalho compara as soluções disponibilizadas pelos sistemas *Derive 5.0*, *Maple 6* e *Mathematica 4.0* para problemas que encontramos no ensino secundário e também nos primeiros anos da universidade. Procuramos destacar os aspectos distintos entre cada um dos programas ao mesmo tempo que fazemos referência aos pontos em que tudo se passa de forma semelhante.

Esta dissertação aborda o cálculo numérico, o cálculo simbólico, a programação e os gráficos. Para cada um dos assuntos é estudada a forma como se podem resolver os problemas através dos três sistemas comparando-se estas soluções.

Inicialmente, é feita uma abordagem que permite ao utilizador adquirir os conhecimentos básicos acerca dos diversos programas. Tratamos de seguida de algumas questões relacionadas com o cálculo numérico e com algumas funções nomeadamente da Teoria dos Números.

Referimos listas e funções e são analisadas diversas formas de manipular listas e os seus elementos bem como algumas áreas da Análise Matemática das quais destacamos as equações, a derivação e a integração compreendendo cálculo numérico e cálculo simbólico. Examinamos um vasto conjunto de operações definidas sobre matrizes (representadas como listas de listas) e polinómios que abrangem as operações mais comuns de cada um dos campos.

Analisamos também a programação recursiva, a programação imperativa, a programação funcional e a programação por regras de reescrita. A abordagem aqui adoptada foi a de fornecer ao utilizador as construções chave mais importantes que cada paradigma de programação utiliza bem como as informações básicas acerca do funcionamento de cada uma delas de modo a permitir a resolução dos problemas propostos.

Por último os gráficos sobre os quais incidiu a nossa análise foram os de uma e de duas variáveis representados no referencial cartesiano, gráficos estes que são os mais utilizados quer ao nível do ensino superior quer ao nível do ensino secundário. A qualidade e a facilidade de obter rapidamente as representações dão outra dimensão ao estudo dos gráficos principalmente quando estamos a falar de gráficos a três dimensões. A ideia de animação gráfica é também aqui abordada sendo evidente os benefícios da utilização da mesma nos programas em que é possível efectuá-la.

Concluímos que na programação o *Mathematica* destaca-se em relação aos demais o mesmo se passando no *Maple* no respeitante à representação gráfica. O *Derive* permite que durante o contacto inicial seja mais fácil trabalhar e aprender a linguagem própria.

COMPARAÇÃO DE SISTEMAS DE PROGRAMAÇÃO

Palavras Chave:

- Comparação de Sistemas Computacionais
- Ensino de Matemática
- Cálculo Numérico e Simbólico
- Paradigmas de Programação
- Representação Gráfica de Funções

ÍNDICE

INTRODUÇÃO

Desenvolvido no âmbito da tese de mestrado em Matemática para o ensino da Universidade da Madeira (UMa), este trabalho teve como objectivo analisar o comportamento de três *Softwares* de matemática (*Derive*, *Maple* e *Mathematica*) quando confrontados com situações que vão desde os simples cálculos numéricos até à manipulação simbólica de expressões e programação.

Optámos por analisar cada programa individualmente em cada um dos assuntos em estudo de modo a que fossem mais notórias as diferenças ou semelhanças entre os diferentes sistemas em apreciação. Pensamos ser essa a forma que mais vantagens traz para o trabalho que desenvolvemos. Como não podia deixar de ser, no final de cada capítulo apresentámos as conclusões que achámos necessárias que se baseiam no tratamento individual a que cada programa foi sujeito.

Ao longo deste trabalho foram feitas breves explicações sobre cada um dos assuntos abordados tendo o cuidado de exemplificar sempre as situações referidas. Pensamos que os exemplos apresentados permitem verificar as vantagens e insuficiências de cada programa no tema em estudo. Os exemplos são apresentados exactamente da mesma forma que o utilizador os veria caso estivesse em frente a um computador com o programa a "correr". Como é óbvio não é possível, nem seria desejável referir todos os meios que cada programa possui para nos ajudar no tratamento de cada um dos temas uma vez que tal tornaria o trabalho muito "pesado" e de difícil leitura.

Esperamos que este trabalho, abordando alguns dos temas sobre os quais incidem os programas da área de Matemática não só dos primeiros anos da licenciatura mas, também do ensino secundário, contribua para construir na mente de muitos o desejo e a ambição de conhecerem um pouco mais sobre aquilo que lhes é ensinado. Foi nosso objectivo que os leitores ficassem mais elucidados acerca dos meios que cada programa dispõe e qual o mais capaz de os ajudar na construção do conhecimento pretendido.

O objectivo principal deste trabalho não foi dar noções matemáticas, apesar de em algumas situações pontuais as mesmas serem fornecidas. Partimos do princípio que as mesmas já são conhecidas e explicámos como realizar muito do trabalho que aprendemos a fazer com lápis e papel, através de cada um dos programas tentando dotar cada leitor de argumentos suficientes para concluir qual dos três programas está melhor equipado para realizar as tarefas pretendidas. Não sendo os exemplos cá apresentados suficientes sugerimos a realização de mais alguns, pois é muito mais eficaz quando somos nós que tomamos as rédeas do nosso conhecimento. Neste caso concreto, se tiver oportunidade de os tornar a executar no computador verá certamente que a motivação para ler e experimentar as capacidades que cada programa possui serão aumentadas de dia para dia.

AGRADECIMENTOS

Ao longo deste trabalho diversas foram as pessoas que contribuíram com o seu apoio para que este trabalho fosse elaborado. A todas elas os nossos agradecimentos e em especial às seguintes:

- ao meu Orientador Professor Doutor Francisco Miguel Dionísio pela atenção, apoio e sugestões sem as quais este trabalho não seria possível;
- a todo o Departamento de Matemática da Universidade da Madeira e em particular ao Professor Doutor José Luís da Silva pelo contributo relacionado com alguma da bibliografia e Professor Leonel Nóbrega pelo apoio relacionado com questões de informática;
- à Timberlake Consultants e em particular ao Sr. Luís Barros pela disponibilidade demonstrada na cedência de uma versão de experimentação do *Maple6*;
- a todos os meus familiares e amigos pela atenção e compreensão demonstrada;
- à Márcia Furtado pelas sugestões de cariz ortográfico e sobretudo pela atenção e disponibilidade.

1. - O PRIMEIRO CONTACTO

Em qualquer um dos programas é relativamente fácil começar a trabalhar. As potencialidades de qualquer um deles são grandes, como adiante veremos com mais pormenor.

Neste primeiro capítulo debruçamo-nos sobre as diferenças visuais que cada programa tem em relação aos restantes ao mesmo tempo que fazemos referência a alguns desses aspectos nomeadamente às diferentes barras.

Iniciamos uma sessão em cada programa e fornecemos as pistas fundamentais para que seja possível aos poucos e poucos, e com a ajuda do próprio programa, ir desenvolvendo o conhecimento acerca do mesmo, contribuindo para que o utilizador seja capaz de encontrar o trilho que o conduzirá a uma maior autonomia que certamente pretende construir.

Num primeiro contacto deparamo-nos com algumas diferenças já na forma como visualmente surge cada um dos programas no ecrã.

Ilustra-se de seguida o primeiro contacto com os vários sistemas, começando pelo *Derive*.

1.1. – ASPECTO INICIAL

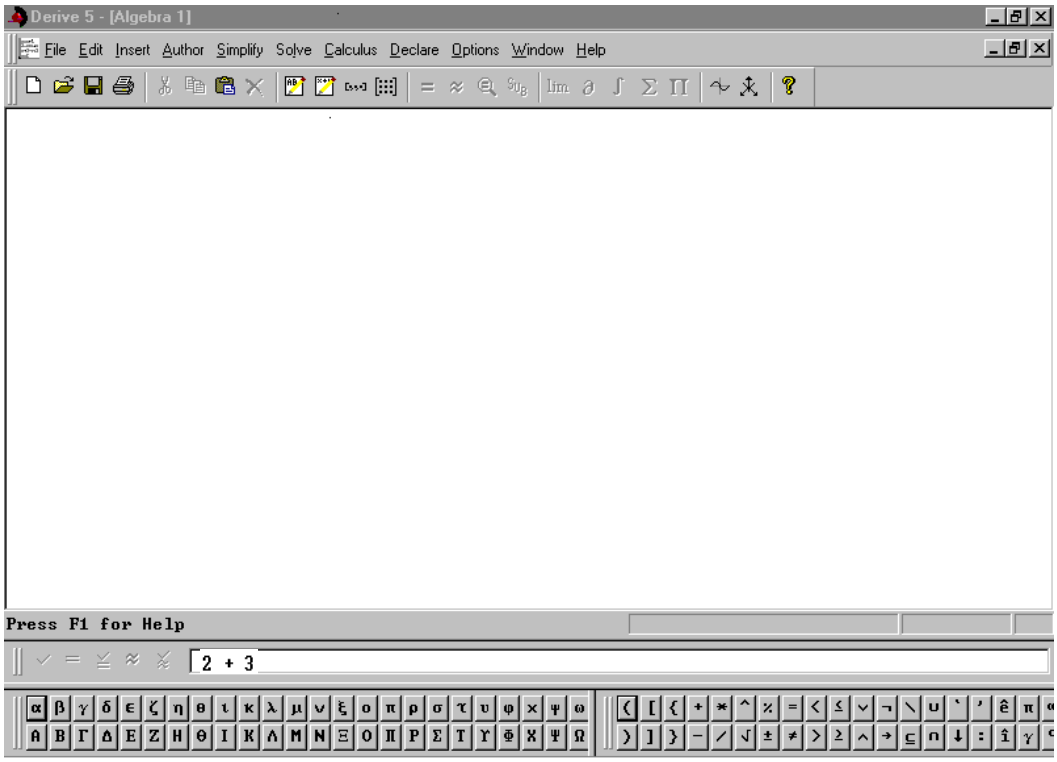
Derive:

Ao abrirmos o *Derive* versão 5.0 surge-nos uma janela constituída por uma parte superior em que aparece a identificação “Derive5 [Algebra1]” que nos permite verificar que entramos com sucesso no programa referido.

A barra de menus do *Derive* é constituída pelos menus *File*, *Edit*, *Author*, *Simplify*, *Solve*, *Calculus*, *Declare*, *Options*, *Window* e *Help*.

Imediatamente abaixo da barra de menus encontramos a barra de ferramentas que contém atalhos para operações tais como: Abrir, Guardar, Imprimir, Copiar e Colar.

De seguida ilustramos a forma como o programa surge no ecrã.



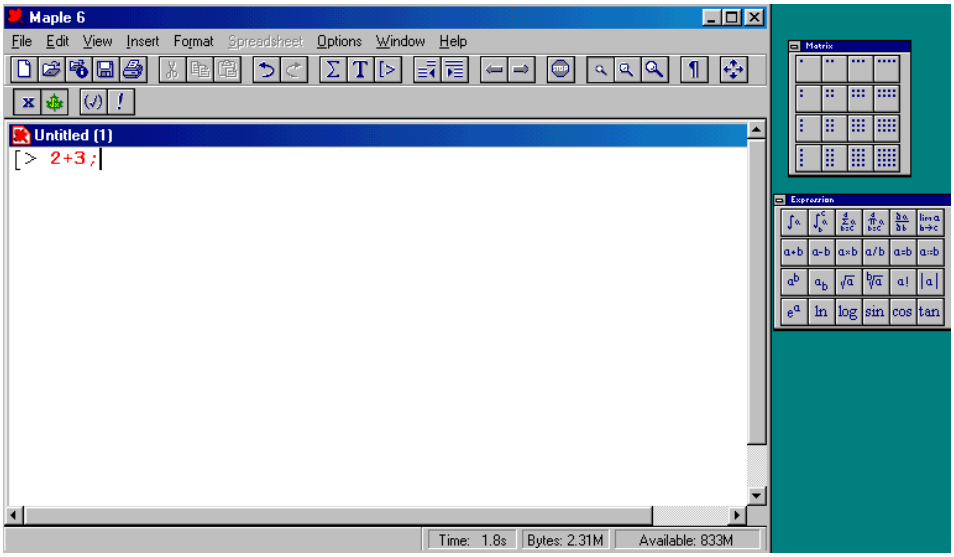
Na parte inferior da janela aberta pelo *Derive* encontra-se uma barra de estado que nos fornece informações do sistema e uma linha onde se introduzem as expressões que pretendemos avaliar. Se quisermos podemos também utilizar quer as letras gregas quer os símbolos que nos são apresentados para mais facilmente podermos editar as nossas expressões.

A zona de trabalho do programa aparece-nos na parte central da janela. É aí que podemos efectuar todo o nosso trabalho.

Maple:

A forma como nos surge o programa *Maple 6* no ecrã é semelhante à anterior. Aqui também, no topo da janela aberta surge uma indicação do programa com que iremos trabalhar.

A barra de menus do *Maple* é constituída pelos menus: *File*, *Edit*, *View*, *Insert*, *Format*, *Options*, *Window* e *Help*. A barra de ferramentas encontra-se imediatamente abaixo dispondo de atalhos para as operações elementares de Abrir, Guardar, Imprimir entre outras.



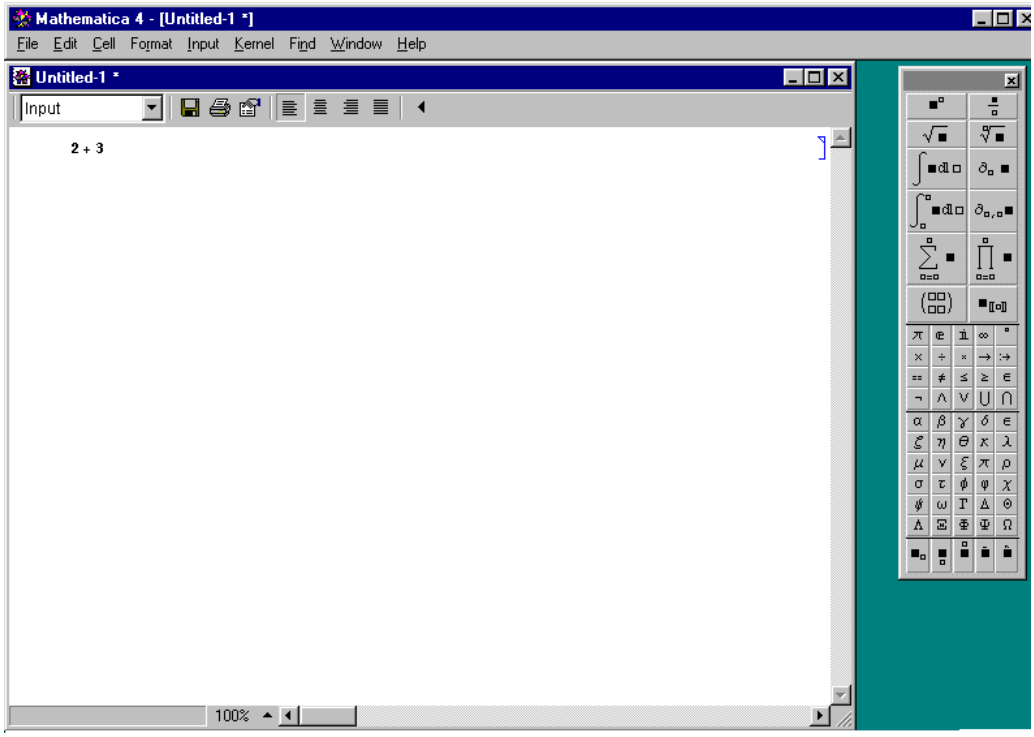
No *Maple* a zona de trabalho encontra-se na parte central do ecrã, sendo iniciada pelo símbolo "**[**>" que nos indica que a célula é uma célula de *Input*.

De referir também que estão disponíveis as *palettes* que permitem uma maior variedade de opções no que diz respeito à introdução dos *inputs*. Na figura acima as *palettes* são apresentadas do lado direito do programa propriamente dito. No entanto as mesmas podem ser movidas para qualquer outra zona do ecrã.

Nem sempre as *palettes* estão visíveis quando damos início a uma sessão em *Maple*. Quando assim acontece recorrendo ao menu *View* e ao submenu *Palettes* e escolhendo a opção mais conveniente para a situação em causa ou então *Show All Palettes* passamos a poder beneficiar das vantagens da utilização dos atalhos lá presentes. Ao longo deste trabalho sempre que utilizarmos as *palettes* e as mesmas não estiverem visíveis o procedimento anterior fará com que as mesmas surjam no ecrã.

Mathematica:

Quando iniciamos uma sessão com o *Mathematica* versão 4.0 surge-nos no ecrã uma janela que é constituída por uma barra de menus, contendo os menus *File*, *Edit*, *Cell*, *Format*, *Input*, *Kernel*, *Find*, *Window* e *Help*. Neste programa, a barra de ferramentas não está disponível no início da sessão mas, podemos ter acesso à mesma recorrendo ao menu *Format* e seleccionando o submenu *Show Toolbar*. Seguindo este procedimento passamos a dispor também de uma barra de ferramentas que contém atalhos para as operações de Guardar e Imprimir.



A zona em que podemos efectuar as operações pretendidas é a zona central da janela.

Através das *palettes* podemos utilizar o conjunto de símbolos que o *Mathematica* disponibiliza de modo que, mesmo nos *inputs* conseguimos empregar uma notação mais próxima daquela que é habitual em Matemática. Quando as mesmas não se encontrarem visíveis, recorrendo ao menu *File* e ao submenu *Palettes* e seleccionando a opção conveniente passamos a ter os meios para utilizar os benefícios que decorrem da utilização dos atalhos lá contidos.

1.2. - COMO COMEÇAR A TRABALHAR?

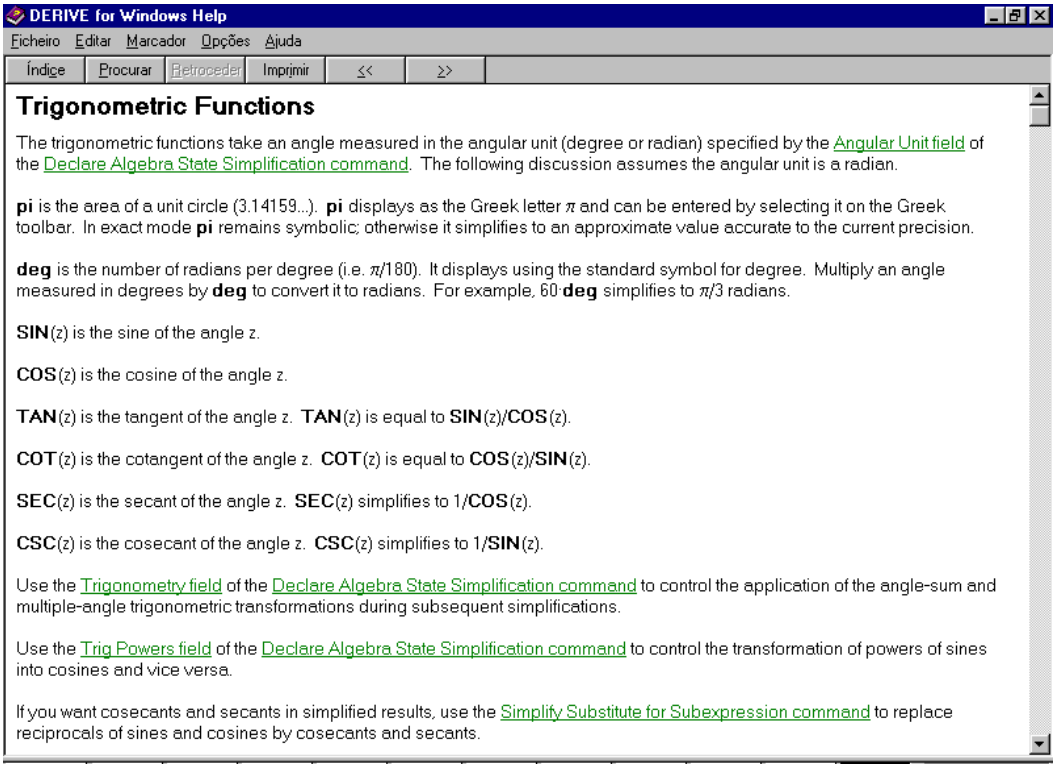
Em todos os programas existe um *Help* que facilita o primeiro contacto com a linguagem de qualquer um.

No que se segue faremos uma pequena análise da ajuda disponibilizada por cada um deles.

Derive:

No *Derive* consultando o menu *Help* temos acesso a todo um conjunto de tópicos de ajuda que depois de seleccionados nos dão as informações pretendidas. Aqui podemos procurar ajuda através de um índice ou através de conteúdos.

Se a título de exemplo quisermos saber alguma informação sobre trigonometria, nomeadamente como invocar funções trigonométricas, podemos optar por seleccionar no menu *Help* o submenu *Index* e assim mediante a escrita das letras **trig** temos acesso a um conjunto de palavras com estas iniciais. Seleccionando *Trigonometric Functions*, temos acesso, numa página de ajuda, a todas as funções trigonométricas que o *Derive* disponibiliza.



Tendo como ponto de partida esta página podemos através dos comandos << e >> ter acesso a outros possíveis tópicos de ajuda sobre os mais variados assuntos, bem como utilizar o facto da ajuda ser um hipertexto para encontrar um campo mais específico que eventualmente necessitemos.

Maple:

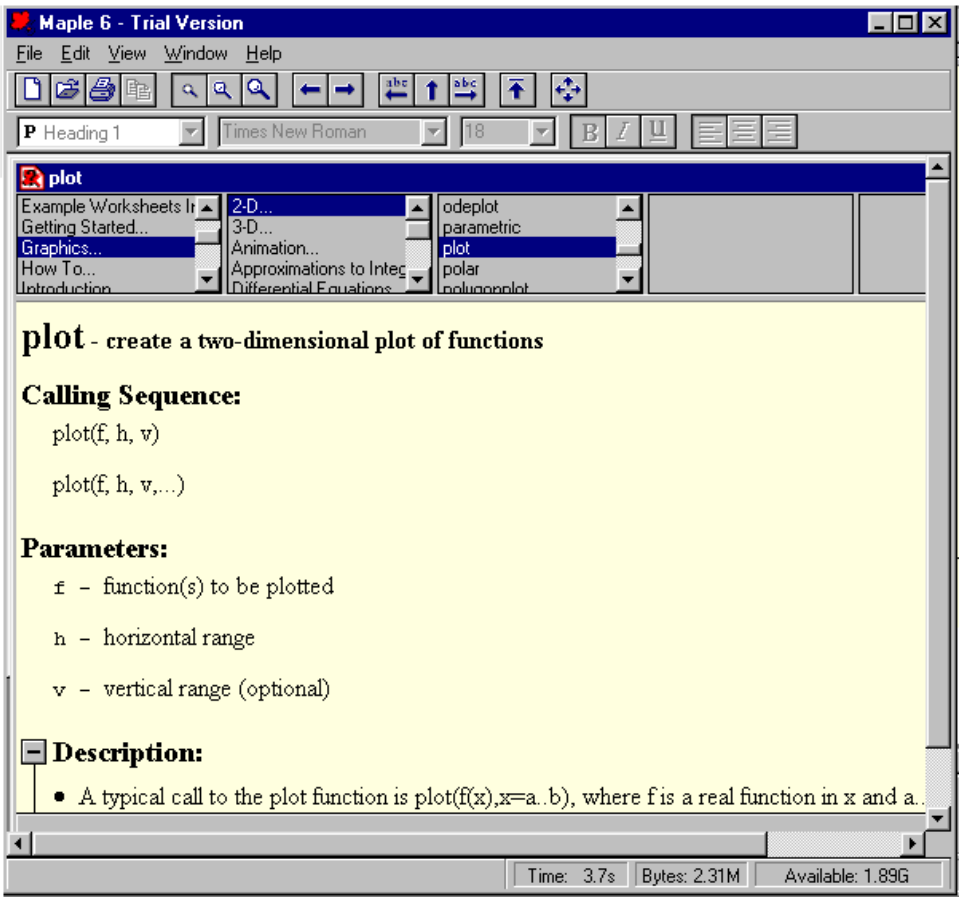
Clicando no menu *Help*, temos acesso a diferentes níveis de ajuda nos quais se inclui um sobre a forma de utilizar a ajuda disponibilizada (*Using Help*), onde são dadas as primeiras informações que tornam mais fácil e rápida a familiarização com as diferentes formas de auxílio que o sistema nos permite aceder.

Suponhamos que pretendemos fazer algumas operações no *Maple* mas que não sabemos exactamente como começar. Seleccionando o menu *Help* e o submenu *Introduction* somos transportados para uma janela que nos fornece as primeiras dicas sobre a forma de iniciar o nosso trabalho.

Clicando em *NewUser's Tour* são fornecidas mais algumas indicações sobre as características do *Maple*. Seleccionando a hiperligação da parte inferior da janela e analisando os tópicos então apresentados são-nos dados mais alguns esclarecimentos acerca do sistema com que estamos a trabalhar nos quais se incluem algumas funções predefinidas como sejam **limit**, **solve**, **expand**, **plot3d**, **sum**, **evalf** entre outras constatando também que qualquer declaração ou expressão em *Maple* que se pretenda avaliar tem de terminar com um ponto e vírgula "**;**". Aproveitamos esta oportunidade para referir que também podemos terminá-la com dois pontos "**:**" situação em que o *output* não é imprimido.

Outra forma de obter ajuda é usar o ponto de interrogação "**?**". Escrevendo numa célula de *input* **?plot** e mandando avaliá-la carregando no ENTER surge imediatamente no ecrã uma janela com toda a informação disponível sobre **plot** bem como os quase sempre elucidativos exemplos.

```
> ?plot
```

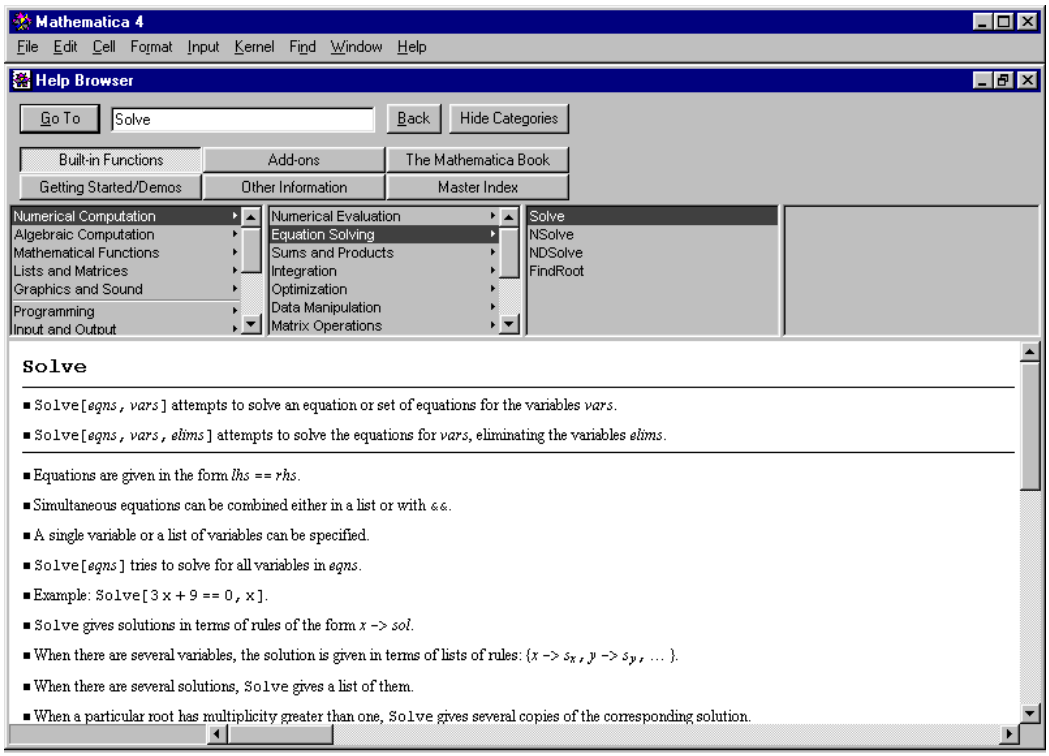


O facto da ajuda ser um hipertexto facilita a consulta de outras páginas de ajuda relacionadas com aquela em que nos encontramos.

Mathematica:

No *Mathematica* clicando no menu *Help* surge-nos no ecrã o *Help Browser* através do qual é possível ter acesso às funções predefinidas.

Suponhamos que pretendemos encontrar as soluções da equação $x^2+3x=0$ escolhendo no *Help Browser* sucessivamente *Numerical Computation->Equacion Solving -> Solve* surge-nos no ecrã a sintaxe do comando *Solve* juntamente com alguns exemplos e algumas informações adicionais. Na parte inferior da janela surgem sugestões e "≅ Further Examples".



Clicando em ≅ surgem-nos várias opções, opções essas que ao serem seleccionadas transformam a própria Janela de Ajuda num local de trabalho, em tudo semelhante a um *Notebook* e em que podemos mandar avaliar as expressões lá contidas tal como se estivéssemos a trabalhar sobre exemplos por nós elaborados. Podemos alterar os exemplos contidos e assim sem grande esforço construir um sem número de exemplos de modo a compreender rapidamente a linguagem utilizada pelo *Mathematica*. A familiarização com a linguagem fica mais fácil e, mesmo o mais inexperiente dos utilizadores fica imediatamente com uma ideia das potencialidades do *software* que está a utilizar.

Como a ajuda do *Mathematica* é um hipertexto podemos directamente através de um clique com o rato passar de uma página de ajuda para outra sem qualquer dificuldade.

Tal como no *Maple*, é possível copiar os exemplos existentes nas páginas de ajuda para uma nova página de trabalho e mandar também aí avaliá-los bem como utilizar o ponto de interrogação para obter informações sobre alguma função predefinida.

```
?Solve
Solve eqns, vars attempts to solve an
  equation or set of equations for the variables
  vars. Solve eqns, vars, elims attempts to solve the
  equations for vars, eliminating the variables elims.
```

1.3. - CONSTANTES

Derive:

Uma forma simples de nos referirmos às constantes π, *e* e *i* que designam a razão entre o perímetro e o diâmetro de uma circunferência, o número de Nepper e $\sqrt{-1}$ é utilizarmos os atalhos que nos surgem na parte mais inferior da janela, nomeadamente [π], [ê] e [î]. O símbolo ∞ é introduzido também através do atalho correspondente - [∞].

```
π
                                                    π
ê
                                                    ê
√(-1)
                                                    î
∞
                                                    ∞
```

Maple:

No que diz respeito às constantes **π** e **i** as mesmas representam-se por **Pi** e **I**, respectivamente. Não existe uma constante para designar o número de *Nepper* ***e***. Para obter esta constante temos de utilizar a função exponencial **exp(1)**. Para designar infinito podemos escrever **infinity**.

```
> Pi;
                                                    π
> sqrt(-1);
                                                    I
> infinity;
                                                    ∞
> exp(1);
                                                    e
```

A razão para este último *output* é o *Maple* trabalhar com valores exactos. O valor numérico é obtido fazendo **evalf(exp(1))**. Desta função (**evalf**) falaremos mais adiante.

Para inserir as constantes podemos ainda utilizar as *palettes*, mais precisamente os atalhos [π], [*i*], [e] e [∞] que estão presentes na *palette* dos símbolos. Refira-se que mesmo que utilizemos os atalhos, o que é registado na zona de trabalho, não são os

símbolos mas, as expressões tal e qual acima se mostra, à excepção da unidade imaginária, que é representada por *I*.

Mathematica:

Há semelhança daquilo que já vimos nos dois programas anteriores, o *Mathematica* também trabalha com os valores de π, *e*, *i* e ∞. A constante **π** pode ser introduzida na forma **Pi** e **Infinity** designa o infinito.

```
Pi
p

Infinity
¥

  └──
   -1
ä
```

No entanto, é mais simples e mais cómodo utilizar as *palettes*. Estas disponibilizam-nos as constantes e símbolos já referidos.

Utilizando os atalhos  ,  ,  e  introduzimos as nossas expressões na forma que é usual em Matemática. O *Mathematica*, ao contrário do *Maple*, aceita os símbolos e trabalha com eles na forma que foram introduzidos mesmo nos *inputs*.

```
p
p

ã
ã

¥
¥

ä
ä
```

1.4. - QUESTÕES DE LINGUAGEM

Em cada um dos programas existem maneiras diferentes para fazer um cálculo ou resolver um problema um pouco mais complicado. Existem regras que todos devemos conhecer para que a iniciação com qualquer um dos programas seja mais simples.

Derive:

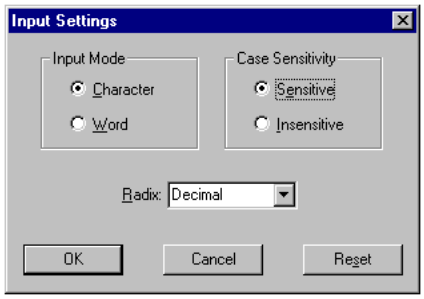
O *Derive* é um programa que não distingue as minúsculas das maiúsculas, isto é, ao pretendermos por exemplo calcular o **cos(π)** podemos escrever **Cos(π)**, **cos(π)** ou até mesmo **cOs(π).** No *Derive*, podemos referirmo-nos à constante **π** escrevendo **Pi** ou **pi**.

Repare-se contudo que ao escrevermos qualquer das expressões acima o *Derive* imediatamente as transforma em **COS(π).**

COS(π)

Passados alguns segundos já não sabemos como foi introduzida a função **cos**.

No entanto, é de salientar que existe a possibilidade através do menu *Declare* e do submenu *Input settings* seleccionar a opção *Case Sensitive*, passando então o *Derive* a distinguir as minúsculas das maiúsculas.



Depois de efectuar esta alteração é imprimido **Case *Mode* := Sensitive** na zona de trabalho.

Maple:

O *Maple* tem a particularidade de colocar os *inputs* e *outputs* de cor diferente. Assim os *inputs* aparecem a vermelho, enquanto que os *outputs* aparecem a azul. É importante também referir aqui que todos os comandos terminam forçosamente com "**;**" ou com "**:**" (ou seja terminam com um ponto e vírgula ou com dois pontos), usando-se este último quando pretendemos que o resultado não seja imprimido. Quando não colocamos o "**;**" e carregamos no ENTER ou no RETURN o *Maple* emite uma mensagem avisando que o comando está incompleto ou que falta o "**;**"(*semicolon*).

No *Maple* as funções predefinidas começam com letra minúscula.

Mathematica:

No *Mathematica* todas as funções predefinidas começam por maiúsculas e os argumentos das funções são colocados entre parênteses rectos "**[ ]**" pelo que ao escrevermos e mandarmos avaliar **cos[Pi]** não obtemos o resultado esperado.

```
In[1]:= Cos[Pi]
Out[1]= -1

In[2]:= cos[Pi]
General:spell1:
 Possible spelling error: new symbol name "cos" is
   similar to existing symbol "Cos".
Out[2]= cos[π]
```

Isto fica a dever-se ao facto de o *Mathematica* não reconhecer **cos** como sendo a função trigonométrica co-seno. No entanto, como podemos ver acima, o *Mathematica* consegue detectar que a função invocada é muito semelhante à que é do seu conhecimento e transmite essa mensagem ao utilizador para que este a possa corrigir se for caso disso. Se introduzíssemos novamente a mesma função o *Mathematica* deixaria de emitir o aviso, não que passasse a identificar **cos** com a função trigonométrica co-seno, mas porque admitiria que o utilizador já está consciente que **cos** não é o mesmo que **Cos**.

Notar que, ao contrário do *Maple*, não é necessário o "**;**" no fim de cada expressão. Aliás, no *Mathematica* a utilização do "**;**" no final de um comando faz com que não ocorra qualquer *output*. Mais a frente veremos que o "**;**" tem também a função de separar diversos comandos escritos em sequência.

1.5. - CONCLUSÃO

Os comandos que o *Derive* coloca de forma visível ao utilizador, nomeadamente na barra de ferramentas e principalmente na barra de atalhos, facilitam muito o primeiro contacto com o programa e tornam mais fácil que nos restantes programas, a efectuação de algumas operações já que a forma de as fazer é muito simples. Não há aqui a necessidade de procurar funções predefinidas para calcular por exemplo um integral. No caso concreto deste último exemplo, se formos ao *Mathematica* é também possível utilizar as *palettes* para introduzir alguns dos cálculos pretendidos já na forma matemática usual, como é o caso do cálculo de um integral. Assim o utilizador não tem a necessidade de procurar as funções que já estão definidas para fazer todos os cálculos. As *palettes* que o *Maple* disponibiliza permitem o ultrapassar de algumas questões relacionadas com a linguagem mas, não são tão eficazes como no *Mathematica* já que aqui as expressões não são inseridas na forma usual. Aparece, isso sim a estrutura da expressão que pretendemos construir usando as funções predefinidas que usaríamos se não dispuséssemos das *palettes*.

Inicialmente no *Mathematica* a utilização dos **[ ]** para introdução dos argumentos das funções revela-se pouco comum, mas uma vez que o procedimento tem praticamente o mesmo grau de dificuldade que os parênteses curvos, facilmente se ultrapassa esse inconveniente.

Por vezes o facto de, tanto o *Mathematica* como o *Maple* serem extremamente sensíveis em relação ao uso de maiúsculas e minúsculas traz alguns inconvenientes. As mensagens de aviso enviadas pelo *Mathematica* são muito úteis, principalmente para quem ainda não conhece bem a linguagem.

De realçar também que o uso do ponto de interrogação no *Mathematica* para obtenção de ajuda, dá-nos uma informação reduzida no próprio local de trabalho, enquanto que o mesmo procedimento no *Maple* tem como consequência a abertura da janela de ajuda relativa ao assunto pretendido.

Em resumo:

- O acesso a funções predefinidas é mais fácil no *Derive*, já que a barra de menus tem opções para muitas das operações que mais vulgarmente utilizamos.
- O *Mathematica* através das *palettes* permite que utilizemos a notação usada em matemática com maior eficácia que nos outros programas.
- O acesso à informação no *Mathematica* é mais cómodo que nos restantes programas podendo utilizar-se o ponto de interrogação para obter uma informação curta sobre o tópico desejado.

Portanto estes três aspectos são os mais importantes e influenciam a forma como encaramos cada um dos programas dando-nos uma primeira ideia dos meios disponibilizados. Se o *Derive* permite um mais simples contacto com as ferramentas que dispõe, o *Mathematica* também dispõe de argumentos válidos nomeadamente no respeitante à ajuda.

2. - CÁLCULO NUMÉRICO

Apesar de não ser essa a principal função de qualquer dos programas que aqui analisamos, em qualquer um deles se podem efectuar cálculos numéricos, quer estes sejam exactos, quer sejam aproximações. Analisaremos aqui como tirar partido das potencialidades deste *software* para atingir os fins para os quais normalmente se utiliza uma calculadora convencional.

Tendo em conta que os erros produzidos pelas aproximações vão progressivamente aumentando, trabalhar com um *software* que consegue escapar a aproximações é muito vantajoso.

Para além destas capacidades vemos aqui como utilizar o *Derive*, o *Maple* ou o *Mathematica* para calcular o máximo divisor comum ou o mínimo múltiplo comum de números inteiros e uma série de funções entre as quais poderemos destacar a função φ de Euler, o número de divisores e a função Mod, que estão tal como as anteriores intrinsecamente ligadas à Teoria dos Números.

Podemos observar ao longo deste capítulo a utilidade de qualquer um destes programas tem, não só pelo cálculo exacto que com eles é possível efectuar mas, também pelas funções predefinidas que cada um apresenta, a quem trabalha nestas áreas e necessita de fazer experimentação e cálculos com rapidez e eficiência.

2.1. - OPERAÇÕES ARITMÉTICAS

Derive:

Através do *Derive* podem ser feitos todos os cálculos que uma calculadora normal efectua. Trabalhando com valores exactos ou com aproximações, é possível chegar aos valores pretendidos sem grande esforço.

Vejamos alguns exemplos onde se ilustram algumas das capacidades do *Derive* ao nível do cálculo numérico.

Para introduzir a expressão basta simplesmente escrever a operação pretendida no teclado e carregar em ENTER. Contudo, existem alturas, nomeadamente quando seleccionamos com o rato uma expressão já anteriormente avaliada ou introduzida, em que não é suficiente introduzir a expressão e mandá-la avaliar, uma vez que nada acontece. Assim, à semelhança do que acontecia em versões anteriores do *Derive* em que era necessário recorrer a um atalho para esse efeito, também no *Derive* versão 5 podemos usar o atalho  , para contornar essa dificuldade. É de referir que também pode ser introduzida uma nova expressão através da utilização do rato, colocando o cursor na linha de introdução das expressões, processo este mais intuitivo que o anterior.

Seguindo um dos métodos anteriores para introduzir a expressão 3+2*3 e carregando no ENTER, o *Derive* copia a expressão para a zona de trabalho, expressão essa que pode ser avaliada através do atalho  existente na barra de ferramentas.

$$3 + 2 \cdot 3$$

$$9$$

A forma de efectuar as operações aritméticas é a normal, podendo usar-se o " " (espaço em branco) para nos referirmos à multiplicação. No decorrer deste trabalho não usaremos tal notação, uma vez que a mesma pode induzir em erro.

$$2 + \frac{1}{2}$$

$$\frac{5}{2}$$

$$\frac{5}{2} - \frac{3}{7}$$

$$\frac{29}{14}$$

De seguida tentamos fazer 2/3*8/5 e surgiu-nos no ecrã a expressão que se segue.

$$\frac{\frac{2}{3}\cdot 8}{5}$$

$$\frac{16}{15}$$

Para que a nossa expressão fique numa forma mais comum basta que se introduzam parênteses nas divisões e aí já temos o pretendido. Note-se contudo que o *Derive* interpretou correctamente o que lhe foi pedido, apenas não o escreveu da forma usual.

$$\frac{2}{3}\cdot\frac{8}{5}$$

$$\frac{16}{15}$$

É possível proceder à introdução de uma expressão já utilizada anteriormente sem que para isso seja necessário escrevê-la de novo. Para tal temos de seleccionar a expressão que queremos introduzir e carregar em F3. Temos ainda a hipótese de utilizar o F4 quando pretendemos que a expressão seleccionada seja inserida dentro de parênteses curvos.

No que diz respeito às precedências elas são as convencionais, podendo usar-se os parênteses curvos para ultrapassá-las.

$$2 + 3\cdot 2$$

$$8$$

$$2 + 3\cdot\left(4 - \frac{3}{2}\right)$$

$$\frac{19}{2}$$

$$2 + 3\cdot(4 - 1.5)$$

$$\frac{19}{2}$$

Neste último exemplo podemos verificar que, ao contrário do que acontece em outros programas, o *Derive*, ainda que na expressão ocorra 1,5, apresenta o resultado exacto (usando 1,5 = 3/2).

Maple:

Os cálculos mais simples que se podem fazer em *Maple* são os numéricos. O *Maple* pode trabalhar da mesma forma que uma calculadora normal. Para tal introduz-se a expressão usando a sintaxe normal, tendo o cuidado de terminá-la com “**;**”.

```
> 2+3;
                                  5

> 1+1/2;
                                  3
                                  -
                                  2

> 2+3*(5-6/2);
                                  8

> 2.8754/2;
                              1.437700000
```

Dos exemplos transcritos facilmente nos apercebemos que o *Maple* efectua cálculos exactos com números racionais, o que representa desde já uma vantagem em relação a uma calculadora normal que não possua cálculo simbólico.

No que diz respeito às operações aritméticas, estas fazem-se da forma usual:

```
> 2+5;
                                  7

> 4-1;
                                  3

> 4*2;
                                  8

> 4/2;
                                  2
```

Podemos utilizar o símbolo **%** para nos referirmos ao último valor calculado e **% %** para nos referirmos ao penúltimo valor calculado e assim sucessivamente.

```
> 2+3;
                                  5

> 3+5;
                                  8

> %;
                                  8

> %%%;
                                  5
```

Ao escrevermos **%** o *Maple* diz-nos que o último valor que calculou foi o 8 enquanto que **% % %** transmite ao *Maple* que queremos saber qual foi o resultado da

antepenúltima avaliação (o último foi o 8, o penúltimo foi o resultante de 3+5, e portanto o valor pretendido é o 2+3 que é 5).

Mathematica:

Veremos aqui alguns exemplos sobre como fazer alguns cálculos com o *Mathematica*.

O *Mathematica* é capaz de trabalhar com aproximações bem como com valores exactos. Depois de escrever uma expressão que queremos avaliar, basta carregar em ENTER ou em SHIFT+RETURN para que imediatamente tenhamos o resultado. Esta rapidez não se verifica quando mandamos avaliar a primeira expressão de uma sessão de *Mathematica* em que a operação leva alguns segundos. Tal facto fica a dever-se à activação do programa que efectua os cálculos que leva alguns segundos. Ao mandar avaliar uma segunda expressão o *output* surgirá muito mais rapidamente.

```
2+3
5

4+3/5
 23
 --
 5

2+3*6/7
 80
 --
 7

2.5/2
1.25

%*2
2.5
```

Usamos **%** para referirmo-nos ao último valor calculado, **%%** ou **%2** para o penúltimo valor calculado e assim sucessivamente.

A notação que o *Mathematica* utiliza para as operações aritméticas é a usual, salvaguardando a multiplicação para a qual se pode usar o “ ” (espaço em branco).

```
2+3
5

2*4
8
```

```
2 / 5
```

$$\frac{2}{5}$$

Ilustremos a utilização do espaço em branco entre dois números:

```
3 4
12
```

Aqui a utilização do “**;**” inibe a impressão do resultado, isto apesar do *Mathematica* efectuar o cálculo normalmente.

```
2 * 5 + 2;

%
12
```

2.2. - CÁLCULO EXACTO COM REAIS - APROXIMAÇÕES

Derive:

É possível efectuar todas as operações aritméticas com racionais e irracionais, sem recorrer a aproximações. Ilustremos aquilo a que nos referimos.

$$\frac{7}{3} + \frac{1}{5}$$

$$\frac{38}{15}$$

$$\frac{7}{5} \cdot \frac{8}{17}$$

$$\frac{56}{85}$$

$$\frac{\frac{3}{2}}{\frac{7}{11}}$$

$$\frac{33}{14}$$

$$\sqrt{2} + 2 \cdot \sqrt{3}$$

$$2 \cdot \sqrt{3} + \sqrt{2}$$

Se pretendermos uma aproximação podemos usar o atalho ≈ da barra de ferramentas. Se nada for dito em sentido contrário a aproximação que será obtida terá dez dígitos significativos. A utilização de SHIFT+RETURN também produz o mesmo efeito, com a vantagem de ser mais cómodo, ficando registado na zona de trabalho o *input* e o *output* resultante da aproximação.

$$\pi$$

$$3.141592653$$

No entanto podemos precisar de trabalhar com maior número de casas decimais.

O *Derive* oferece-nos a possibilidade de trabalharmos com valores aproximados, exactos ou mistos. Para seleccionarmos um destes tipos fazemos **Precision := Mode** onde **Mode** poderá ser **Approximate**, **Exact** ou **Mixed**. Inicialmente o *Derive* trabalha no modo exacto.

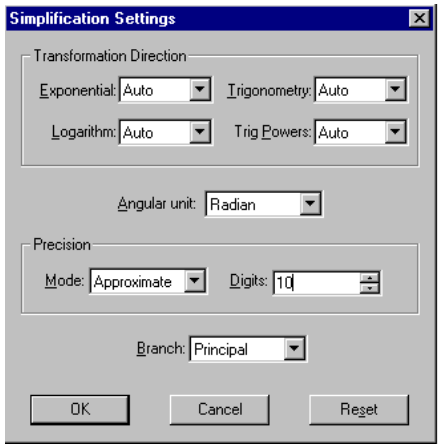
Precision := Approximate

Approximate

Se mandarmos avaliar $\sqrt{2}$ através do atalho = contido na barra de ferramentas o *Derive* fornece-nos um valor aproximado para a expressão avaliada.

```
√2
                                                    1.414213562
```

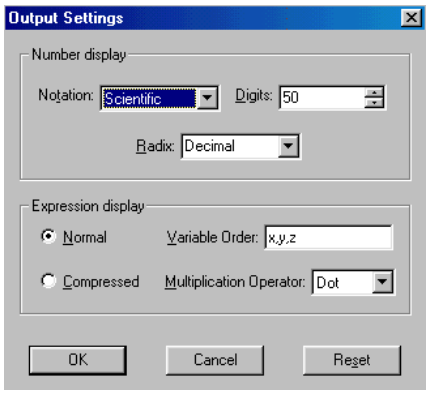
Podemos também conseguir o mesmo fim se utilizarmos o menu *Declare* e o submenu *Simplification Settings* e alterarmos a zona relativa ao *Mode*.



De maneira semelhante procedemos quando pretendemos aumentar ou diminuir o número de casas decimais com que iremos trabalhar. Para tal fazemos **PrecisionDigits : = número** onde **número** é um número inteiro positivo.

```
PrecisionDigits := 50
                                                    50

π
                  3.1415926535897932384626433832795028841971693993751
```

De modo análogo ao que acontece no caso anterior, este *output* só é obtido se mandarmos avaliar a expressão pretendida através da barra de ferramentas e dos respectivos atalhos = e ≈. Também se poderia alterar o número de algarismos significativos através do menu *Declare* e do submenu *Output Settings* (CTRL+ALT+O).



Logo que carregamos em OK, aparece na zona de trabalho **NotationDigits:=50** que nos indica que a partir daí as aproximações serão feitas com 50 algarismos significativos.

Maple:

O *Maple* trabalha com números racionais e dá-nos o resultado exacto, mesmo quando este não seja um inteiro ou uma dízima finita.

```
> 1/5+1/3;
```

$$\frac{8}{15}$$

O *Maple* permite que operemos com fracções com muita facilidade sem termos de nos preocupar com a redução ao mesmo denominador quando tal é necessário, com a vantagem de não cometer erros de cálculo.

No que diz respeito às restantes operações (multiplicação, subtracção e divisão) tudo se passa com normalidade sem se perder a exactidão dos resultados finais.

```
> 3/5*2/7;
```

$$\frac{6}{35}$$

```
> 3/7-2/5;
```

$$\frac{1}{35}$$

Caso seja pretendida uma aproximação o *Maple* está apto a fornecê-la. Para isso basta-nos usar a função **evalf**.

```
> evalf(Pi,50);
```

3.1415926535897932384626433832795028841971693993751

O primeiro argumento é o valor do qual queremos uma aproximação e o 50 significa que o *Maple* deverá dar-nos uma aproximação com 50 casas decimais. Podemos mandar avaliar a função anterior sem que esta tenha um segundo argumento, nesse caso o *Maple* dá-nos um valor aproximado com dez algarismos significativos.

```
> evalf(Pi);
```

3.141592654

Note-se que ao fazermos:

```
> 1.5*5;
```

7.5

o resultado obtido é um número decimal e não 15/2, isto porque para o *Maple* 1.5 não é o mesmo que 3/2. O *Maple* considera que uma dízima é sempre um valor aproximado.

Desde que numa determinada expressão exista uma dízima, o *Maple* apresentará também o resultado final em forma de dízima. Isto leva-nos a ter de usar sempre valores exactos, se é exactidão que pretendemos da parte do programa.

Mathematica:

O *Mathematica* tem a capacidade de operar de forma exacta com números racionais e irracionais.

```
1/2+1/3
```

$$\frac{5}{6}$$

```
Sqrt[2]
```

$$\sqrt{2}$$

Desta forma podemos utilizar o *Mathematica* para fazermos as operações aritméticas normais, mesmo que não estejamos a trabalhar apenas com inteiros, sem corrermos o risco de estarmos a cometer erros relacionados com as aproximações.

No *Mathematica* existe a função predefinida **N[m,n]**, que torna possível obter uma aproximação para um dado valor **m** com uma precisão de **n** dígitos. Quando não existe indicação acerca da precisão pretendida o *Mathematica* assume que a mesma é com 6 algarismos significativos.

```
N[Pi]
```

```
3.14159
```

```
N[Pi, 50]
```

```
3.1415926535897932384626433832795028841971693993751
```

Repare-se que o valor **Pi** pode também ser introduzido na forma usual, **π**, e que o *Mathematica* trabalha com o seu valor exacto.

2.3. - OPERAÇÕES COM INTEIROS

2.3.1. - MÍNIMO MÚLTIPLO COMUM E MÁXIMO DIVISOR COMUM

Derive:

O máximo divisor comum e o mínimo múltiplo comum são obtidos pelas funções **GCD** e **LCM** respectivamente.

```
GCD(24, 36)
                                        12

LCM(24, 36)
                                        72
```

Mesmo quando o número de argumentos destas funções é superior a dois continua a ser possível a sua aplicação.

```
GCD(42, 24, 15)
                                          3

LCM(4, 7, 8)
                                         56
```

Para escrever o máximo divisor comum de **a** e **b** como combinação linear de **a** e **b** aplicamos a função **EXTENDED_GCD** que nos fornecerá uma lista constituída pelo máximo divisor comum **d** e uma lista contendo os coeficientes **n** e **m** tais que **d = n × a + m × b.**

```
EXTENDED_GCD(168, 875)
                                  [7, [99, -19]]
```

Portanto **7 = 99 × 168 - 19 × 875**.

Não é possível prolongar as capacidades desta função para três ou mais inteiros. Se o tentarmos a função simplesmente ignora os restantes argumentos como se pode ver de seguida.

```
EXTENDED_GCD(168, 875, 5)
                                  [7, [99, -19]]

GCD(168, 875, 5)
                                          1
```

Maple:

As funções **igcd(a,b)** e **ilcm(a,b)** permitem calcular o máximo divisor comum e o mínimo múltiplo comum de dois ou mais inteiros.

```
> igcd(18,27);
                                    9

> ilcm(18,27);
                                   54
```

Da mesma forma calculamos o mínimo múltiplo comum e o máximo divisor comum de três ou mais inteiros.

```
> ilcm(-10,6,-8);
                                  120

> igcd(-10,6,-8);
                                    2
```

É também possível encontrar a forma de escrever o máximo divisor comum de dois inteiros como combinação linear desses mesmos dois números inteiros. Na forma seguinte os coeficientes são armazenados nas variáveis **s** e **t**, respectivamente.

```
> igcdex(45,12,'s','t');
                                    3

> s; t;
                                   -1
                                    4
```

Portanto neste caso $\mathbf{3 = -1 \times 45 + 4 \times 12}$.

Esta função tal como a equivalente em *Derive* não pode ser invocada para os casos em que temos três ou mais inteiros.

```
> igcdex(45,12,35,'a','b','c');
Error, (in igcdex) Illegal use of a formal parameter
```

Mathematica:

As funções **GCD** e **ExtendedGCD** referem-se ao máximo divisor comum de dois ou mais inteiros sendo que a última ainda tem a particularidade de nos dar a informação de como obter o máximo divisor comum como combinação linear dos dois ou mais números que constituem o argumento.

```
GCD[12, 84]
12
```

```
ExtendedGCD[72, 84]
{12, 1, 1}
```

O último *output* significa que **12 = -1 × 72 + 1 × 84**.

O mínimo múltiplo comum de dois números inteiros é obtido através do uso da função predefinida **LCM**.

```
LCM[72, 84]
504
```

Se estivermos a trabalhar com três ou mais inteiros continua a ser possível calcular o máximo divisor comum e o mínimo múltiplo comum.

```
GCD[36, 84, 30]
6
```

```
LCM[36, 84, 30]
1260
```

Ao contrário do que acontecia com o *Derive* e com o *Maple*, o *Mathematica* permite que apliquemos a função **ExtendedGCD** ainda que tenhamos mais do que dois inteiros.

```
ExtendedGCD[72, 24, 84, 45]
{3, {0, 12, 4, -1}}
```

```
ExtendedGCD[72, 84, 24]
{12, {-1, 1, 0}}
```

2.3.2. - FUNÇÕES DE E COM PRIMOS

Derive:

Quanto às funções **PRIME** e **NEXT_PRIME**, a primeira é um predicado que nos diz se um determinado número é ou não primo e a segunda dá-nos o primo seguinte ao número que introduzimos.

```
NEXT_PRIME(15)
                                                    17

PRIME(17)
                                                  true
```

Carregando o pacote *Number.MTH*, usando para tal o menu *File* e o submenu *Load* e escolhendo as opções *Math* e *Number* podemos ainda dispor das funções **NTH_PRIME** que nos diz qual o n-ésimo primo e **Euler_PHI**. Esta última diz-nos quantos dos números inteiros positivos inferiores a um dado número são primos com ele.

```
NTH_PRIME(3)
                                                     5

EULER_PHI(7)
                                                    6

EULER_PHI(10)
                                                    4
```

Maple:

No *Maple* também existem algumas funções que incidem sobre os primos. Podemos determinar se um número é ou não primo usando para tal a função **isprime**.

```
> isprime(143);
                                                false

> isprime(19);
                                                true
```

Sendo dado um inteiro podemos descobrir qual o primo imediatamente superior e qual o primo anterior.

```
> nextprime(143);
                                                149
```

```
> prevprime(143);
                                    139
```

É ainda possível utilizar uma função predefinida para extrair um dado primo através da introdução da posição em que este se encontra na lista dos primos.

```
> ithprime(5);
                                    11
```

A função **phi** disponível através do pacote *numtheory* fornece-nos o número de inteiros positivos que não excedem o argumento da função e que são primos com este.

```
> with(numtheory):
Warning, new definition for order

> phi(11);
                                    10

> phi(15);
                                     8
```

Mathematica:

No *Mathematica* está definido o predicado **PrimeQ** que nos diz se um dado número inteiro é ou não número primo.

```
PrimeQ [41]
True

PrimeQ [297]
False
```

Se quisermos saber, por exemplo, qual é o 6º primo podemos usar a função predefinida **Prime**. Imediatamente ficamos a saber que o 6º primo é o 13.

```
?Prime
Prime [n] gives the nth prime number.

Prime [6]
13
```

Tal como vimos para os dois programas anteriores, o *Mathematica* também possui uma função predefinida que permite descobrir quantos são os números menores que **n** que são primos com **n**, onde **n** é o argumento da função. Essa função é **EulerPhi**.

```
EulerPhi [17]
16

EulerPhi [20]
8
```

2.3.3. - DIVISORES

Derive:

Os divisores de um dado número podem ser calculados utilizando a função **DIVISORS**.

```
DIVISORS(24)
                                    [1, 2, 3, 4, 6, 8, 12, 24]
```

O número de divisores de um dado número é-nos fornecido pela função **DIVISOR_TAU**.

```
DIVISOR_TAU(24)
                                                8
```

Uma função da qual a anterior é um caso particular é **DIVISOR_SIGMA(k,n)** que dá-nos a soma das potências dos divisores de **n** quando o expoente é **k**.

```
DIVISOR_SIGMA(2, 24)
                                               850

                                    2    2
1 + 4 + 9 + 16 + 36 + 64 + 12  + 24
                                               850
```

Repare-se que se fizermos **k=0** obtemos o mesmo resultado que em **DIVISOR_TAU** já que nesse caso estamos a somar a unidade tantas vezes quantas vezes os divisores de **n** são.

```
DIVISOR_SIGMA(0, 24)
                                                8
```

Fazendo **k=1** obtemos a soma dos divisores de **n**.

```
DIVISOR_SIGMA(1, 24)
                                               60
```

Maple:

Carregando o pacote relativo à Teoria dos Números passamos a dispor de um conjunto de funções que nos permitem trabalhar com divisores. Se já o carregou durante a sessão em que se encontra, e é provável que já o tenha feito uma vez que precisamos dele aquando da utilização da função **phi** do tópico anterior**,** não precisa de tornar a carregá-lo.

Para obtermos os divisores positivos de um número inteiro aplicamos a função **divisors**.

```
> with(numtheory):
Warning, new definition for order

> divisors(20);
                          {1, 2, 4, 5, 10, 20}
```

Utilizando a função **tau** é possível ficarmos a conhecer quantos são os divisores de um dado inteiro.

```
> tau(20);
                                6
```

A soma dos divisores obtém-se pela utilização da função **sigma(n)** podendo também utilizar-se esta função para calcular a soma das potências de expoente **k** quando as bases são os divisores de **n**.

```
> sigma(20);
                               42

> sigma[0](20);
                                6

> sigma[2](20);
                              546
```

Mathematica:

Relativamente às situações ilustradas nos programas anteriores, o *Mathematica* dispõe de duas funções: **Divisors** e **DivisorSigma[k,n]**.

```
Divisors[12]
{1, 2, 3, 4, 6, 12}
```

Para obter o número de divisores utilizamos o facto já atrás referido de que, por cada divisor há que adicionar uma unidade. Sendo assim, considerando que a função **DivisorSigma[k,n]** nos dá a soma das potências de expoente **k** quando as bases são os divisores, fazendo **k = 0** obtemos o número de divisores.

```
DivisorSigma[0, 12]
6
```

A soma dos divisores obtém-se fazendo **k = 1**.

```
DivisorSigma[1, 12]
28
```

Se ainda for necessário podemos somar os quadrados dos divisores de **12** fazendo **DivisorSigma[2,12]**.

```
DivisorSigma[2,12]
210
```

Mais à frente veremos outras funções que nos permitiriam o acesso indirecto a algumas destas funções. Por exemplo, o número de divisores poderia ser dado pelo comprimento da lista dos divisores. Nesta altura ainda não nos referimos à forma de determinar o comprimento de uma lista faremo-lo no decorrer do capítulo 3.

2.3.4. - A FUNÇÃO MOD

Derive:

Podemos encontrar no *Derive* a função **MOD(m,n**) que nos fornece o resto não negativo da divisão inteira do primeiro argumento pelo segundo. A função **MODS** faz o mesmo, só que dá-nos um valor compreendido entre **–n/2** e **n/2** onde **n** é o segundo argumento da função.

```
MODS(7, 3)
                                                    1

MODS(17, 6)
                                                   -1

MOD(17, 6)
                                                    5
```

Maple:

A função **mod** no *Maple* está representada através das funções **mod**, **modp** e **mods**. Enquanto que a primeira pode utilizar-se na forma usual em que a função **mod** fica situada entre os argumentos, as restantes tem de ser usadas na forma mais comum sendo os parâmetros indicados entre parênteses curvos.

```
> 12 mod 7;
                                                    5

> modp(12,7);
                                                    5

> mods(12,7);
                                                   -2
```

A função **iquo(d,q,x)** dá-nos o quociente da divisão de **d** por **q** e dá a **x** o valor do resto da referida divisão. O mesmo se passa em relação à função **irem(d,q,y**) que dá--nos o resto da divisão de **d** por **q**, dando a **y** o valor do quociente da divisão.

```
> irem(23,4,'q');
                                                    3

> q;
                                                    5
```

```
> iquo(23,4,'r');
                                                    5

> r;
                                                    3

> irem(-23,4);
                                                   -3
```

Mathematica:

Através da aplicação das funções **Quotient** e **Mod** ficamos a saber o quociente e o resto da divisão inteira do primeiro argumento pelo segundo.

```
Mod[7,4]
1

Quotient[7,4]
4
```

Qualquer uma das funções anteriores pode ser invocada com três argumentos quando assim acontece o valor que obtemos será um pouco diferente do obtido anteriormente. Vejamos alguns exemplos para passarmos posteriormente à sua explicação.

```
Mod[27,5,2]
2

Mod[27,5,10]
12
```

Enquanto que a função **Mod[m,n]** dá-nos um valor compreendido entre **0** e **n-1** a mesma função com um terceiro argumento **d** irá dar um valor que estará entre **d** inclusive e **d+n** exclusive.

```
Quotient[27,5,2]
5

Quotient[27,5,10]
3
```

Deste feita a função **Quotient[m,n,d]** define-se à custa da anterior como sendo o valor **x** tal que **d ≤ m-n x < d+n**. Dito de uma forma mais intuitiva será o valor do divisor que feita a divisão permitirá encontrar um resto definido por **Mod[m,n,d]**.

2.3.5. - APROXIMAÇÕES A INTEIROS

Qualquer um dos três programas põe ao nosso dispor um conjunto de funções que nos permitem obter aproximações inteiras para determinados valores quer estes sejam dízimas, o resultado de uma divisão, de outra operação ou de uma função qualquer. Analisamos já de seguida algumas dessas funções. As funções aqui analisadas têm de comum o facto de o seu resultado ser sempre um número inteiro.

Derive:

Trabalhando com divisões e pretendendo aproximações o *Derive* permite-nos que escolhamos o modo como essa aproximação será feita. As funções que aqui analisamos estão definidas no pacote *Number.MTH* pelo que necessitamos de carregá-lo usando para tal o menu *File*, o submenu *Load* e escolher as opções *Math* e *Number*.

Optando por trabalhar com valores aproximados por excesso interessa conhecer a função **CEILING** que efectua aproximações para o menor inteiro superior à fracção introduzida. Por exemplo $\frac{7}{5} = 1{,}4$. Para aproximar este valor para o menor inteiro maior que ele utilizamos a função **CEILING**.

```
CEILING(7, 5)
                                                    2
```

Se pelo contrário pretendermos que a aproximação seja feita para o maior inteiro menor que uma dada fracção podemos utilizar a função **FLOOR**.

```
FLOOR(7, 5)
                                                    1
```

Quando o que nos interessa é o inteiro que mais próximo está da nossa fracção, a função **ROUND** serve perfeitamente. Esta aproxima a nossa fracção para o inteiro que lhe está mais próximo. Quando existem dois inteiros nessas condições a aproximação é feita para o número par mais próximo.

```
ROUND(7, 5)
                                                    1
```

Se não for com fracções que estivermos a trabalhar a função é invocada com um único argumento como se segue.

```
ROUND(7.5)
                                                    8

FLOOR(2.3)
                                                    2
```

Maple:

As aproximações a inteiros podem ser feitas de quatro formas diferentes. Pretendendo uma aproximação por excesso podemos utilizar a função **ceil** que retornará um inteiro nas mesmas condições que a função **CEILING** no *Derive*.

```
> ceil(7.2);
                                    8

> ceil(-9/2);
                                   -4
```

As aproximações por defeito podem ser feitas pela função **floor** que também actua da mesma forma que a sua congénere do programa anterior.

```
> floor(-7.5);
                                   -8

> floor(3/5);
                                    0
```

A função **trunc** permite obter também valores aproximados por excesso no caso do argumento representar um número negativo ou um valor aproximado por defeito caso se trate de um número positivo. Esta função simplesmente "esquece" as casas decimais.

```
> trunc(7.5);
                                    7

> trunc(-9/2);
                                   -4
```

Por último a função **round(x)** produz como *output* o inteiro que está mais próximo de **x**. Quando a igual distância a aproximação faz-se para o número par mais próximo o que também acontecia no *Derive*.

```
> round(3.2);
                                    3

> round(3.7);
                                    4

> round(3.5);
                                    4
```

As raízes quadradas aproximadas por um número inteiro podem ser obtidas através do uso das funções **isqrt(n)** e **iroot(n, 2),** podendo também usar-se esta última função para fazer raízes de outro índice, bastando para tal indicar o tipo de raíz que se pretende calcular através da modificação do segundo argumento da função para 3,4,…

```
> isqrt(25);
                                  5

> readlib(iroot);
                          proc(n, r) ... end

> iroot(65,3);
                                  4

> iroot(16,4);
                                  2
```

Observe-se que o resultado obtido através da função **isqrt(n)**, na maior parte das vezes, não corresponde ao valor que obteríamos se utilizássemos a função **sqrt(n)**. Tal só acontece quando o argumento **n** é um quadrado perfeito. O mesmo se passa com a função **iroot(m,3)** que só dará o valor da raiz cúbica real se **m** for um cubo perfeito, nos casos em que isso não acontece a função dá o valor inteiro mais próximo dessa mesma raiz.

Mathematica:

Quando estamos a trabalhar com aproximações, o *Mathematica* põe ao nosso dispor um conjunto de funções que nos permite manipular os resultados mais facilmente.

Através da função **Round** podemos obter o inteiro que está mais próximo de um dado valor. Esta função actua do mesmo modo que as funções anteriores com o mesmo nome já analisadas.

```
Round[2.75]
3

Round[5/2]
2
```

Também são possíveis as aproximações a inteiros por excesso e por defeito. A função **Floor** dá-nos o maior inteiro menor ou igual ao argumento e a função **Ceiling** fornece-nos o menor inteiro maior ou igual a um dado valor.

```
Floor[4/3]
1

Ceiling[4/3]
2
```

A parte inteira de um número pode ser dada pela função **IntegerPart** e a sua parte decimal é fornecida pela função **FractionalPart**. Naturalmente o *output* desta

última função nunca é um inteiro, de qualquer modo aqui fica também uma referência a esta função uma vez que está intrinsecamente ligada à função anterior.

```
IntegerPart(7.3)
```
$$7$$

```
FractionalPart(7.3)
```
$$0.3$$

```
IntegerPart(-5/3)
```
$$-1$$

```
FractionalPart(-5/3)
```
$$-\frac{2}{3}$$

Como se pode ver os resultados obtidos pela função **IntegerPart** são os mesmos que obteríamos se utilizássemos a função **trunc** em *Maple*.

2.4. - FUNÇÕES MATEMÁTICAS PREDEFINIDAS

Derive:

No *Derive* estão predefinidas as funções mais vulgarmente usadas tais como as funções trigonométricas (**Sin**, **Cos**, **Tan**, **Cotg**, **Sinh**, **Cosh**, **Tanh**, …, **Arcsin**, **Arccos**, **Arctan**, …), a função exponencial (**Exp**), o logaritmo natural ou de Nepper (**Ln**) e o de base 10 (**Log[10]**) e a raiz quadrada (**Sqrt**).

Vejamos então como calcular o co-seno de **π** (apesar de no ecrã surgir **π**, a expressão introduzida foi **COS(Pi)**). Se quisermos podemos usar as letras gregas existentes na parte inferior e em particular π para introduzirmos **π** na forma usual.

```
COS(π)
```

$$-1$$

Calculemos por exemplo o valor do cubo da função exponencial.

```
ê^3
```

$$\hat{e}^3$$

A função do logaritmo natural é obtida através da função **Ln(n)**, enquanto que o logaritmo de uma base diferente é obtido através de **LOG(n,b)** onde **b** é a base pretendida.

```
LN(ê)
```

$$1$$

```
LOG(1000, 10)
```

$$3$$

A função **ABS** fornece-nos o valor absoluto ou módulo de um dado valor. Introduzindo a expressão **ABS(3.5)**, o *Derive* transforma-a imediatamente, surgindo no ecrã o *input* e o correspondente *output* depois de avaliado.

```
|3.5|
```

$$\frac{7}{2}$$

$$\left|-\frac{11}{5}\right|$$

$$\frac{11}{5}$$

Maple:

No *Maple* estão definidas as funções matemáticas mais usadas, nomeadamente as funções trigonométricas, a função exponencial (**exp**), a função logaritmo(**ln**) e a raiz quadrada (**sqrt**).

```
> cos(Pi);
```

$$-1$$

```
> ln(exp(3));
```

$$3$$

Ao pretendermos utilizar a função logaritmo de base 10, temos de previamente mandar o *Maple* ler essa função através do comando **readlib(log10)**, como se ilustra abaixo. A impressão no ecrã de **proc(x) ... end** poderia ser evitada se usássemos "**:**" em vez de "**;**" (isto é, se substituíssemos o ponto e vírgula pelos dois pontos).

```
> readlib(log10);
```

$$\mathbf{proc}(x)\ \ldots\ \mathbf{end}$$

```
> log[10](10000);
```

$$\frac{\ln(10000)}{\ln(10)}$$

Mandemos simplificar a expressão anterior para confirmar o resultado obtido.

```
> simplify(ln(10000)/ln(10));
```

$$4$$

A função **sqrt** dá-nos a raiz quadrada de um dado número.

```
> sqrt(144);
```

$$12$$

```
> sqrt(125);
```

$$5\sqrt{5}$$

Neste último caso, uma vez que não pedimos um valor aproximado, o *Maple* apresenta o resultado mais simples que conseguiu determinar que corresponde à raiz quadrada exacta de 125.

O valor absoluto ou módulo é dado pela função **abs**.

```
> abs(2.5);
```

$$2.5$$

```
> abs(-3/11);
```

$$\frac{3}{11}$$

Mathematica:

No *Mathematica* estão definidas as funções matemáticas de uso mais comum, como as funções trigonométricas, a função logaritmo (**Log**), a função exponencial (**E**), a raiz quadrada (**Sqrt**),...

Apresentamos de seguida alguns exemplos sobre as funções que tornam possível trabalhar com os conceitos mais frequentemente utilizados quando trabalhamos com funções matemáticas.

```
Cos[Pi]
-1
```

Como atrás vimos, a função exponencial de base *e* (*e =2.71828...)* no *Mathematica* é definida pela função **E**, enquanto que o logaritmo Natural é dado pela função **Log**. Caso pretendamos calcular o logaritmo de uma base diferente basta fazer **Log[b,n]** onde **b** indica a base.

```
?E
E is the exponential constant e
  base of natural logarithms, with numerical
  value approximately equal to 2.71828.
```

```
?Log
Log[z] gives the natural logarithm of z.  Log[b, z] gives the logarithm to
  base b.  Log[b, z] gives the logarithm to base b.
```

```
Log[10, 10000]
4
```

Quanto à função da raiz quadrada ela é dada pela função **Sqrt[n]**.

```
Sqrt[2]
√2
```

Outra forma de referirmo-nos à raiz quadrada é utilizarmos as *palettes* e introduzirmos a raiz na notação matemática usual, utilizando para tal os atalhos  ou  .

```
√3
√3
```

Para raízes de índice diferente de dois procedemos de modo semelhante.

```
∜5
5^(1/4)
```

O valor absoluto de um dado valor pode ser obtido através da utilização da função **Abs**.

```
Abs[-3/8]
```

$$\frac{3}{8}$$

```
Abs[-2.7]
```

```
2.7
```

2.5. – CONCLUSÃO

No tratamento numérico de dados, todos os programas comportam-se de maneira muito semelhante não existindo grandes diferenças.

Qualquer dos programas está apto a efectuar operações com números reais manipulando-os de forma exacta.

Ao pretender obter aproximações no *Maple* e no *Mathematica* é necessário utilizar uma função predefinida enquanto que no *Derive* basta utilizar o atalho  ou mandar avaliar as expressões introduzidas carregando em SHIFT+ENTER.

O comportamento do *Derive* difere dos demais programas quando num determinado cálculo surgem dízimas. Para o *Derive* uma dízima é tratada como um valor exacto enquanto que nos restantes é considerada como uma aproximação. Tal facto tem consequências ao nível dos *outputs* obtidos quando pelo menos um dos números com que estamos a operar é uma dízima, no *Derive* obtemos um valor exacto no *Maple* e no *Mathematica* uma aproximação.

Quando as aproximações pretendidas são para inteiros, todos os programas têm funções predefinidas para as fazer. O *Maple* nesse campo disponibiliza funções que permitem aproximar directamente raízes de qualquer índice para um inteiro.

Apenas o *Mathematica* consegue encontrar uma combinação linear para o máximo divisor comum de três ou mais inteiros ficando-se os restantes programas pelos dois inteiros.

Em todos os programas é possível encontrar um predicado que permite verificar se um dado número é primo bem com saber qual o primo que ocupa uma posição conhecida na lista dos primos. No *Maple* é ainda possível saber qual o primo anterior e o primo seguinte a um dado número.

A função φ de Euler está definida em todos os programas possuindo todos eles funções para nos fornecer os divisores de um dado número. Ainda sobre divisores conseguimos, utilizando funções predefinidas, calcular o número de divisores e a soma de qualquer potência dos divisores de um dado inteiro. Refira-se aqui que o *Mathematica* para efectuar estas operações tem menos funções predefinidas mas, consegue alcançar os mesmos objectivos que os outros programas.

A utilização da função **mod** é permitida em todos os programas havendo a hipótese de optar por outras funções com ela relacionadas por exemplo **mods** (*Derive* ou *Maple*) ou acrescentar um terceiro argumento à função **Mod** (*Mathematica*) de modo a que esta forneça um inteiro mais apropriado para a situação concreta em estudo.

Em resumo:

- O *Derive* considera as dízimas como valores exactos enquanto que o *Maple* e o *Mathematica* tratam-nas como aproximações.
- O *Maple* ao nível das aproximações para inteiros dispõe de mais funções que os outros programas.
- Mesmo com menos funções predefinidas para trabalhar na área por nós escolhida dos divisores, o *Mathematica* consegue realizar as mesmas tarefas que os restantes programas.

Parece-nos importante realçar que o tratamento que o *Derive* faz das dízimas não coincide com o dos demais o que influência o *output* obtido quando pelo menos uma das expressões a calcular envolve uma dízima. Nas restantes áreas aqui abordadas o comportamento é semelhante.

3. - LISTAS E FUNÇÕES

Começaremos nesta fase a aprofundar um pouco mais os conhecimentos que temos de cada um dos programas. Neste capítulo iniciaremos a manipulação de expressões e exemplificaremos mais algumas vantagens que temos em trabalhar com qualquer um dos programas.

Não deixaremos para trás os cálculos numéricos, mas passaremos a trabalhar com diferentes tipos de expressões em cada um dos programas sem que tenham as expressões forçosamente de ser números. Exploraremos as ferramentas que quer o *Derive*, quer o *Maple*, quer o *Mathematica* põem ao dispor do utilizador para que este possa mais fácil e rapidamente atingir os seus objectivos.

Este capítulo é muito importante pois iniciamos aqui a manipulação de expressões e de estruturas mais complexas que os números que, até este momento, têm sido o objecto do nosso estudo.

Abordamos nesta altura a sintaxe própria que cada programa possui para criarmos as nossas funções e/ou programas. Neste trabalho resolveremos algumas situações ou problemas utilizando estruturas distintas de simples números, daí tratarmos de por em evidência as bases que se utilizarão na sua composição bem como os diversos meios que temos para intervir nessas mesmas estruturas. Enfim, estudamos aqui algumas das funções que cada sistema disponibiliza para permitir que através das suas características seja mais fácil resolver de forma eficiente os problemas com os quais nos deparamos no nosso trabalho.

3.1. - ATRIBUIÇÃO DE NOMES A EXPRESSÕES

Derive:

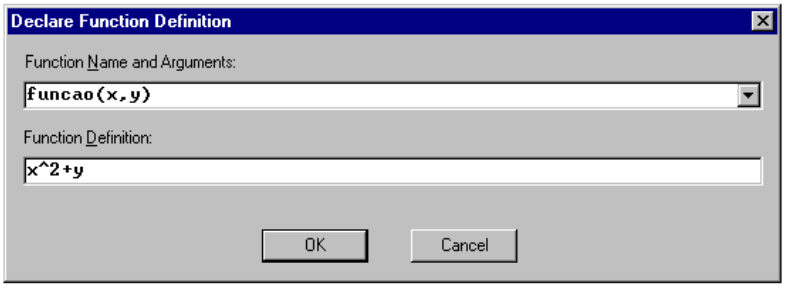
Considerando que no decorrer da resolução de um dado problema surgem-nos por vezes valores que importa guardar é de interesse compreender como tal se pode fazer utilizando as capacidades do *Derive*. Um dos métodos possíveis é usar a sintaxe **variável:=expressão**.

```
x := 3
                                                    3

x + 2
                                                    5
```

Não podemos fazer atribuições a todo o tipo de variáveis, uma vez que algumas delas estão reservadas para o próprio sistema. Assim, ao tentarmos fazer uma atribuição a **Pi,** obtemos um erro de sintaxe uma vez que, a expressão **Pi** representa a constante **π**.

Podemos também utilizar o *Derive* para construirmos nós mesmos as funções com que queremos trabalhar.

Podemos definir uma função através do menu *Declare* e do submenu *Function Definition*. Depois temos de introduzir o nome da função juntamente com os argumentos e a expressão que a define como de seguida se exemplifica.



Outra forma seria usar a sintaxe **função(x,y):=expressão**.

Definamos por exemplo uma função que recebendo dois números fornece-nos o produto dos dois. Aqui não necessitamos de avaliar a expressão que define a função.

```
multiplicacao(a, b) := a·b


multiplicacao(5, 7)
                                                   35
```

Apesar da simplicidade do exemplo anterior, podemos sempre utilizar a definição de funções em situações mais complexas.

```
RQMULT(a, b) := √(a·b)

RQMULT(2, 18)
                                                    6

RQMULT(3, 27)
                                                    9
```

Maple:

O *Maple* permite que sejam dados nomes a expressões a fim de facilitar o nosso trabalho. Para tal usa-se a seguinte sintaxe: **nome := expressão**. Podemos dar um nome a qualquer expressão *Maple*.

```
> x:=2;
```

$$x := 2$$

A partir deste momento e se nada for dito em contrário, a expressão **x** guarda o valor **2**.

```
> x+3;
```

$$5$$

Tentemos fazer nova atribuição:

```
> equacao:=y=y-5;
```

$$equacao := y = y - 5$$

Os nomes em *Maple* podem conter qualquer carácter e “underscores”, mas não podem começar por um número.

```
> equação:=y=x+2;
Syntax error, control character unexpected
```

O erro surgido deve-se ao uso indevido do “ç”.

No entanto, nem todos os nomes estão disponíveis para variáveis. O *Maple* tem predefinidos alguns nomes que por esse facto estão reservados. Tentando atribuir a um nome que já está reservado ou predefinido, o *Maple* emite uma mensagem de erro, informando que o nome está protegido.

```
> Pi:=3.1415;
Error, attempting to assign to `Pi` which is protected
```

Não ficam por aqui as possibilidades abertas por este programa. Podemos também tirar partido da possibilidade de sermos nós próprios a definir as funções que pretendemos utilizar.

Definamos uma função que recebendo um número calcula o valor do seu cubo.

```
> f:=x->x^3;
```

$$f := x \to x^3$$

Passamos assim a dispor de mais uma função com a qual poderemos trabalhar como se a mesma estivesse predefinida. Assim, é possível por exemplo calcular a imagem de um dado número por meio desta função.

```
> f(2);
```

$$8$$

```
> f(2.5);
```

$$15.625$$

O procedimento anterior poderá parecer pouco útil, mas o mesmo ganha relevância quanto maior for o trabalho a realizar e mais complexa for a função.

Vejamos ainda um exemplo que nos mostra o meio de definir uma função que possua mais do que uma variável.

```
> quadif:=(x,y)->(x-y)^2;
```

$$quadif := (x, y) \to (x - y)^2$$

```
> quadif(4,5);
```

$$1$$

```
> quadif(5,-2);
```

$$49$$

Mathematica:

No *Mathematica*, podemos atribuir valores a variáveis. A atribuição pode ser feita através de uma atribuição imediata ou através de uma atribuição diferida. Na primeira, usamos o sinal de igualdade (**nome = expressão**), enquanto que na segunda usamos **:=** (**nome := expressão)**. Vejamos em primeiro lugar como se processa a atribuição imediata.

```
b= 3
3

a= b+ 1
4
```

Portanto o valor de **a** passou a ser **4** enquanto que em **b** ficou armazenado o valor **3**.

Na atribuição diferida, aparentemente tudo se passa de maneira semelhante. A avaliação da atribuição não é feita no momento. A mesma só é efectuada quando a variável ocorrer numa expressão que se pretende avaliar.

```
c := b+1

c
4

?c
Global`c
c := b+1

?a
Global`a
a = 4
```

Portanto à variável **c** fica associado o valor de **b+1**. Repare-se que à variável **a** ficou associado o valor **4**.

Apesar do que atrás foi dito ainda ficamos com a ideia que ambos os tipos de atribuição produzem o mesmo efeito nas variáveis em causa. Ora, na realidade tal não é assim.

Continuemos ainda com o exemplo anterior e façamos uma nova atribuição a **b**.

```
b = 7
7
```

Vejamos agora os valores guardados em **a** e **c** respectivamente.

```
a
4

c
8
```

Detectamos aqui a diferença entre os dois tipos de atribuição. Enquanto que a variável **a** não foi alterada, a variável **c,** pelo facto de ter sido definida através de uma atribuição diferida, foi modificada, isto porque a **c** está associado o valor de **b+1**. Por consequência ao alterar o valor de **b** estamos também a fazer o mesmo na variável **c**.

Consoante a situação em que nos encontramos, importa pois utilizar a atribuição que mais vantagens nos dá. Refira-se ainda que normalmente é preferível utilizar a atribuição imediata quando se trata apenas de armazenar valores em variáveis. A atribuição diferida é particularmente útil quando pretendemos definir funções por atribuição paramétrica da qual falaremos no capítulo 5.

Apesar do grande número de nomes que podemos dar às variáveis ou funções por nós definidas, existem alguns nomes que o *Mathematica* reserva para si, como sejam as funções predefinidas e as constantes. Caso tentemos fazer uma atribuição que não seja permitida, imediatamente surge-nos um erro que nos indica que o símbolo ou expressão em causa está protegido.

```
Pi = 3.1415
Set::wrsym : Symbol p is Protected.
3.1415
```

Veremos agora como definir funções por abstracção funcional. A definição de uma função em *Mathematica* obedece à construção:

Function[**w**,**corpo**]

onde **w** é um nome ou uma lista de nomes que representa os parâmetros da função e **corpo** é uma expressão *Mathematica* que traduz os cálculos que a função deve efectuar.

```
quadcub = Function [ x^2 + x^3 ]
Function [ x^2 + x^3 ]
```

Definida a função que soma o quadrado de um número com o seu cubo, podemos utilizá-la para calcular as imagens de alguns objectos.

```
quadcub [ 1 ]
2
```

```
quadcub [ 2 ]
12
```

Definamos ainda mais uma função que dados dois números calcula o produto da soma dos dois com a sua diferença, trata-se pois de uma função de dois argumentos.

```
prod = Function [ { x, y } , (x + y) (x - y) ]
Function [ { x, y } , (x + y) (x - y) ]
```

```
prod [ 3, 1 ]
4
```

```
prod [ 2, 3 ]
- 5
```

Repare-se que neste caso, os argumentos da função que definimos são colocados entre chavetas constituindo uma estrutura a que se dá o nome de lista da qual falaremos ainda neste capítulo.

3.2. - APAGAR VARIÁVEIS

Derive:

Como vimos, fazendo uma atribuição a uma variável esta passa a denotar um determinado valor como de seguida se mostra.

```
x := 3
```

3

Mandando avaliar **x** automaticamente obtemos o valor lá guardado.

```
x
```

3

Para guardar um novo valor basta usar a mesma sintaxe. Desta forma a anterior atribuição é substituída pela nova.

```
x := 8
```

8

Ao pretender voltar ao estado inicial em que a variável **x** não guarda nenhum valor em particular, temos de atribuir a **x** o valor de **x** como de seguida se exemplifica.

```
x := x
```

x

Mandando avaliar **x** podemos verificar que na realidade, a esta variável não está associado qualquer valor.

```
x
```

x

Maple:

É possível alterar o valor previamente guardado numa variável através de uma nova atribuição.

```
> x:=4;
```

$x := 4$

```
> x;
```

4

```
> x:=5;
```

$x := 5$

```
> x;
```

$$5$$

Se pretendermos fazer com que a expressão **x** volte ao seu estado original em que não possuía qualquer valor associado, podemos fazer como se segue.

```
> x:='x';
```

$$x := x$$

```
> x;
```

$$x$$

Outra forma de consegui-lo é utilizar o comando **restart**. Desta forma todas as variáveis são apagadas. Estamos a colocar-nos exactamente na situação em que nos encontrávamos quando iniciámos a sessão em *Maple*.

```
> restart;
```

Mathematica:

Após executar a atribuição

```
x = 2
```
2

a variável **x**, se nada for dito em contrário, ficará até ao final da sessão com o valor **2**.

Suponhamos que utilizando a variável **x** pretendemos guardar o valor **7**. Tal pode obviamente ser feito através de uma nova atribuição que automaticamente substitui todas as atribuições anteriores.

```
x = 7
```
7

```
x
```
7

Mas se o procedimento anterior resolve o caso em que queremos alterar o valor guardado, não resolve o problema de fazer com que a variável **x** não guarde qualquer valor. Quando pretendemos que assim seja temos de usar a função **Clear**.

```
Clear[x]
```

```
x
```
x

3.3. - TIPOS

Derive:

Uma lista é uma sequência ordenada de elementos. Uma das aplicações para as listas é a definição de vectores. Para introduzir uma lista utilizamos o menu *Author* e o submenu *Vector command* no qual é pedida a dimensão da lista em questão, procedendo-se depois à introdução dos seus elementos. Para o mesmo fim, podemos utilizar o atalho  , existente na barra de ferramentas.

Alternativamente podemos simplesmente escrever uma lista utilizando os parênteses rectos e introduzir os elementos separados por vírgulas.

```
[4, 5, 2, -1, 2]
                                        [4, 5, 2, -1, 2]
```

A aplicação das listas para definir vectores está bem vincada na definição de listas através de uma expressão geradora. Ora, nem sempre sendo prático definir uma lista elemento a elemento o *Derive* através da função **VECTOR**, permite que definamos uma lista utilizando uma expressão para gerar os respectivos elementos.

```
VECTOR(x^2, x, 3)
                                        [1, 4, 9]
```

O comando **VECTOR** pode ser invocado com quatro e até com cinco argumentos. Nos casos em que usamos quatro argumentos, para além da expressão geradora dos elementos da lista e da variável, surge também o valor inicial e o valor final da variável. O quinto argumento quando usado define a forma de progredir, o comprimento do passo a dar.

```
VECTOR(2·n, n, 2, 4)
                                        [4, 6, 8]

VECTOR(√n, n, 3, 7, 2)
                                        [√3, √5, √7]
```

Há ainda a possibilidade de utilizarmos a função **VECTOR** apenas com três argumentos, mas em que o terceiro argumento é ele próprio uma lista. Nesse caso **VECTOR** dá-nos o valor que a expressão introduzida no primeiro argumento assume quando a variável percorre os elementos expressos na lista.

```
VECTOR(√n, n, [1, 4, 9, 16, 25, 36])
                                        [1, 2, 3, 4, 5, 6]
```

Para nos referirmos a um conjunto no *Derive* colocamos os elementos separados por vírgulas dentro de chavetas.

Tratando-se de conjuntos, a ordem pela qual os elementos são introduzidos nem sempre é respeitada quando mandamos avaliar a expressão introduzida.

```
{1, 2, 3, 4, 5, -2}
                                        {-2, 1, 2, 3, 4, 5}
```

Quando são introduzidos elementos repetidos num conjunto o *Derive* apenas considera uma cópia desse elemento, desprezando portanto as restantes. Apesar da ordem não ser relevante, o *Derive* organiza sistematicamente os conjuntos por nós definidos. Neste caso concreto os elementos foram colocados por ordem crescente.

```
{0, 1, -1, 2, -2, 3, 4, 5, 2, 2, 5}
                                    {-2, -1, 0, 1, 2, 3, 4, 5}
```

Podem usar-se as reticências para abreviar a forma de definir um conjunto.

```
{1, ..., 9}
                                        {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Pode ainda definir-se a diferença entre dois quaisquer valores consecutivos de um conjunto pela colocação de dois valores antes das reticências.

```
{1, 3, ..., 9}
                                            {1, 3, 5, 7, 9}

{1, -2, ..., -10}
                                        {-8, -5, -2, 1}
```

Maple:

A estrutura de dados base do *Maple* são as sequências de expressões que não são mais do que um grupo de expressões *Maple* separadas por vírgulas.

```
> 1,2,3,4;
                                            1, 2, 3, 4

> x,y,z,w;
                                            x, y, z, w
```

Como se pode constatar este método de introduzir uma sequência é pouco prático quando pretendemos trabalhar com sequências maiores. Nesses casos podemos utilizar a função **seq** que dada uma expressão gera os elementos partindo de um valor inicial até um valor final.

```
> seq(i,i=1..15);
                1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

> seq(3*i,i=4..9);
                            12, 15, 18, 21, 24, 27
```

As sequências de expressões não são nem listas nem conjuntos. São uma estrutura de dados distinta do *Maple* e têm as suas próprias propriedades.

Por exemplo, elas preservam a ordem e a repetição dos seus elementos. Os itens introduzidos permanecem na ordem introduzida e caso seja introduzido um elemento duas vezes, ambas as cópias permanecerão na sequência de expressões. As sequências prolongam as capacidades de muitas das operações básicas do *Maple.*

No *Maple* criamos uma lista colocando, um número qualquer de objectos *Maple* separados por vírgulas, dentro de parênteses rectos.

O *Maple*, tal como acontece com as sequências de expressões, preserva a ordem e a repetição numa lista.

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

$$[a, b, c, d, e]$$

Um conjunto (*set*) é uma sequência de expressões dentro de chavetas.

```
> conjunto:={1,0,-1,1,3,3};
```

$$conjunto := \{0, -1, 1, 3\}$$

Não são preservadas nem a ordem nem a repetição em conjuntos.

Tal como vimos para as sequências, podemos gerar também uma lista ou um conjunto tirando partido da função **seq**. Para isso basta-nos introduzir essa mesma função dentro de chavetas ou dentro de parênteses rectos quer pretendamos obter um conjunto quer pretendamos uma lista.

```
> {seq(i^2+3,i=0..3)};
```

$$\{3, 4, 7, 12\}$$

```
> [seq(i^2,i=-3..3)];
```

$$[9, 4, 1, 0, 1, 4, 9]$$

Os *arrays* são uma extensão da estrutura de dados lista. Assim, a ideia subjacente é que um *array* é uma lista em que, a cada item se associa um número inteiro (que até pode ser zero) – o seu índice – que representa a sua posição na lista.

Podemos definir ou alterar os elementos de um *array* sem ter de defini-lo completamente.

```
> vect:=array(1..3);
```

$$vect := \mathrm{array}(1 .. 3, [\ ])$$

```
> vect[1]:=5;vect[2]:=7;vect[3]:=9;
```

$$vect_1 := 5$$

$$vect_2 := 7$$

$$vect_3 := 9$$

Ou, fazendo tudo de uma só vez.

```
> dobro:=array(1..3,[2,4,6]);
```

$$dobro := [2, 4, 6]$$

Para ver o conteúdo de um *array* é necessário usar o comando **print**.

```
> print(vect);
```

$$[5, 7, 9]$$

Definamos agora um *array* com dois índices (uma matriz).

```
> soma:=array(1..2,1..2);
```

$$soma := \mathrm{array}(1 .. 2, 1 .. 2, [\ ])$$

Comecemos por atribuir valores à primeira e depois à segunda linha do *array*.

```
> soma[1,1]:=2;soma[1,2]:=2;soma[2,1]:=3;soma[2,2]:=4;
```

$$soma_{1,1} := 2$$

$$soma_{1,2} := 2$$

$$soma_{2,1} := 3$$

$$soma_{2,2} := 4$$

```
> print(soma);
```

$$\begin{bmatrix} 2 & 2 \\ 3 & 4 \end{bmatrix}$$

É possível gerar os elementos de um *array* através de uma expressão. Em primeiro lugar vejamos um exemplo para depois tecermos algumas considerações sobre o mesmo.

```
> v:=array(1..6):

> for i from 1 to 6 do v[i]:=3*i od:

> print(v);
```

$$[3, 6, 9, 12, 15, 18]$$

O que foi pedido ao *Maple* para fazer foi que percorresse os índices do *array* desde a sua primeira entrada até a última e que a todas elas fizesse corresponder o seu triplo. Para consegui-lo utilizámos a construção **for** da qual falaremos no capítulo 5 relativo à programação.

Uma *table* é uma extensão do conceito de *array*. Enquanto que num *array* tínhamos forçosamente um inteiro como índice, aqui já não tem de o ser.

```
> num_de_alunos:=table([sétimo=20,oitavo=30,nono=25]);
num_de_alunos := table([
    nono = 25
    oitavo = 30
    sétimo = 20
    ])
```

Mathematica:

No *Mathematica* as expressões são formadas a partir das expressões mais simples, como sejam as constantes e as variáveis.

Uma sequência de expressões no *Mathematica* é um conjunto de expressões separadas por vírgulas. Por si só as sequências não têm aplicações (se mandarmos avaliar um conjunto de expressões separadas por vírgulas surgirá um erro), mas são elas a base para todas as estruturas mais complexas presentes neste programa, nomeadamente no tipo lista.

Uma lista em *Mathematica* pode ser definida pela enumeração de todos os seus elementos, dentro de chavetas e separados por vírgulas. Pode também optar-se por definir uma lista usando a função predefinida **List** e colocando os elementos como sendo os argumentos desta função.

```
List[a, b, c]
{a, b, c}
```

Torna-se evidente que a forma atrás referida para definir uma lista nem sempre é a mais indicada, principalmente quando a lista a formar é maior. Quando assim é, podemos tirar partido de funções predefinidas como **Range**, **Table** ou **Array**.

A função **Range[imin,imax,di]** gera uma lista começando em **imin** e terminando em **imax**, progredindo **di** unidades de cada vez.

```
Range[2, 15, 2]
{2, 4, 6, 8, 10, 12, 14}
```

A diferença fundamental entre usar uma *table* e um *array* é que a primeira usa para gerar os elementos uma expressão enquanto que a segunda usa uma função (*function)*. Vejamos como construir uma lista com os cinco primeiros quadrados perfeitos usando alternativamente uma *Table* e um *Array*.

```
Table[i^2, {i, 5}]
{1, 4, 9, 16, 25}

Table[i^2, {i, 5}]
{1, 4, 9, 16, 25}

Array[Function[x, x^2], 5]
{1, 4, 9, 16, 25}
```

Neste último caso poderíamos ainda definir a função fora do *Array*, procedimento que torna mais claro a definição do *Array*.

```
f = Function[x, x^2]
Array[f, 5]
{1, 4, 9, 16, 25
```

Neste caso não seria tão simples se pretendêssemos fazer o mesmo utilizando a função **Range**, uma vez que não é constante a diferença entre dois elementos consecutivos da lista.

Uma possível solução seria fazer **Range[5]*Range[5]** e obteríamos o pretendido. Quando a função **Range** é invocada com um único argumento, o *Mathematica* assume que o valor mínimo é **1** e que o progresso se faz unidade a unidade até atingir o valor do único argumento que é considerado como sendo o valor **imax**. Quando invocada com dois argumentos, o primeiro é o valor de **imin** e o segundo o valor de **imax**, fazendo o progresso também de unidade a unidade.

Se quiséssemos construir a lista dos números maiores que **10** e menores que **49** que se iniciasse em **11** e que progredisse de três em três, facilmente aplicaríamos a função **Range** para obter o pretendido.

```
Range[11, 48, 3]
{11, 14, 17, 20, 23, 26, 29, 32, 35, 38, 41, 44, 47
```

3.4. - ACESSO AOS ELEMENTOS DE UMA ESTRUTURA

Derive:

Na base de uma lista estão naturalmente os elementos que a compõem. Dada a funcionalidade de nos referirmos a cada um deles o *Derive* oferece a possibilidade de utilizarmos algumas funções para obtermos os vários elementos de uma lista ou conjunto.

A função **FIRST**, como o próprio nome indica, dá-nos o primeiro elemento de uma lista. Nos conjuntos esta função não se aplica de forma satisfatória uma vez que não obtemos o elemento pretendido, isto é, não é retornado qualquer elemento. Ao invés disso o *Derive* apenas nos diz que pretendemos o primeiro elemento e por aí se fica.

```
FIRST([7, 4, -1])

                                                    7


FIRST({7, -3, 0, -1})

                                          {-3, -1, 0, 7}
                                                        1
```

Pretendendo obter outro elemento que não o primeiro podemos utilizar a função **sub**. Introduzindo da maneira usual a expressão **[17,19,23,29]sub2**, conseguimos aceder ao segundo elemento da lista.

```
[7, 0, 1, 9]
            2

                                                    0
```

No *Derive*, as matrizes são representadas como listas de listas, por esse facto podemos penetrar na estrutura "matriz" utilizando duas vezes o comando **sub** (ou seja, fazendo **[[1,2,3],[4,5,6],[7,8,9]] sub3 sub1**, partindo do princípio que queremos o elemento situado na terceira linha e na primeira coluna).

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}_{3,1}$$

$$7$$

Introduzindo **[[1,2,3],[4,5,6],[7,8,9]] sub [3,1]** iremos obter uma matriz composta pela terceira e pela primeira linha por esta ordem.

```
[ 1  2  3 ]
[ 4  5  6 ]
[ 7  8  9 ]
[3, 1]
```

$$\begin{bmatrix} 7 & 8 & 9 \\ 1 & 2 & 3 \end{bmatrix}$$

Com a mesma finalidade podemos ainda utilizar a função **ELEMENT** que dado uma lista e uma posição fornece-nos o elemento que ocupa essa posição.

```
ELEMENT([7, 14, 21, 28], 3)
```

21

Se estivermos no campo das matrizes continuamos a dispor desta função havendo que referir a linha e a coluna onde o elemento que pretendemos se encontra.

```
ELEMENT([ 1  3  5 ], 2, 3)
        [ 0  2 -1 ]
        [ 7  4  3 ]
```

−1

Se pretendermos obter uma submatriz temos de especificar as linhas pretendidas.

```
ELEMENT([ 1  3  5 ], [2, 3])
        [ 0  2 -1 ]
        [ 7  4  3 ]
```

$$\begin{bmatrix} 0 & 2 & -1 \\ 7 & 4 & 3 \end{bmatrix}$$

Maple:

A preservação da ordem numa lista torna possível a extracção de um elemento particular de uma lista.

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

$$lista := [a, b, c, d, e]$$

```
> [a,b,c,d,e][4];
```

$$d$$

Podemos aceder aos elementos de um conjunto de forma semelhante à anterior.

```
> conjunto:={1,0,-1,1,3,3};
```

$$conjunto := \{0, -1, 1, 3\}$$

```
> conjunto[2];
```

$$-1$$

Podemos seleccionar um único elemento de um *array* ou de um conjunto (*set*) usando a mesma notação aplicada às listas.

```
> vect:=array(1..3,[5,7,9]);
```

$$vect := [5, 7, 9]$$

```
> vect[2];
```

$$7$$

Podemos seleccionar um elemento especificando a linha e a coluna pretendida.

```
> soma:=array(1..2,1..2);
```

$$soma := \text{array}(1 .. 2, 1 .. 2, [\ ])$$

```
> soma[1,1]:=2;soma[1,2]:=2;soma[2,1]:=3;soma[2,2]:=4;
```

$$soma_{1,1} := 2$$

$$soma_{1,2} := 2$$

$$soma_{2,1} := 3$$

$$soma_{2,2} := 4$$

```
> soma[2,1];
```

$$3$$

```
> alfab:=array(1..3,1..2,[ [a,b],[c,d],[e,f]]);
```

$$alfab := \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}$$

Nas *tables* tendo em conta que os índices não são necessariamente inteiros, o modo de seleccionarmos um elemento é análogo.

```
> num_de_alunos:=table([sétimo=20,oitavo=30,nono=25]);
```

$$num\_de\_alunos := \text{table}([$$
$$nono = 25$$
$$oitavo = 30$$
$$sétimo = 20$$
$$])$$

```
> num_de_alunos[oitavo];
```

$$30$$

Se em vez de um único elemento quisermos seleccionar pela posição que os elementos se encontram na lista um conjunto de elementos podemos fazer **S[i..j]** ou **op[i..j,S]** onde estamos a obter os elementos da lista ou conjunto **S** que vão desde a posição **i** até à posição **j** inclusive. Se utilizarmos números inteiros negativos para nos referirmos às posições estamos a referir-nos ao último quando escrevemos –1 e ao escrevermos –2 estamos a dirigir-nos ao penúltimo elemento e assim por aí adiante.

```
> lista:=[a,b,c,d,e,f,g,h,i]:
> lista[3..6];
```

$$[c, d, e, f]$$

```
> lista[-2..-1];
```

$$[h, i]$$

```
> op(2..4,lista);
```

$$b, c, d$$

Mathematica:

O *Mathematica* permite-nos ainda ter acesso aos diferentes elementos de uma lista quer através das funções predefinidas **Last**, **First**, quer através de **[[ ]]** (selecção de um elemento).

```
, 5, 7, 11, 13, 17
7

, 7 , 1, 17 , 2, 1
17
```

Podemos também utilizar a selecção de um elemento para definir uma lista ou alterar uma já existente.

```
nomes = , beatriz, carla
, beatriz, carla

nomes ana
joana

?nomes
Global`nomes
nomes = , joana, carla
```

3.5. - INSERÇÃO DE ELEMENTOS

Derive:

A inserção de elementos faz-se no *Derive* através da função **INSERT** e indicando o elemento que pretendemos introduzir bem como a posição em que o mesmo deve ser inserido. Vejamos como introduzir o elemento **11** na quinta posição.

```
INSERT(11, [2, 3, 5, 7, 13], 5)
                                    [2, 3, 5, 7, 11, 13]
```

Nos conjuntos, uma vez que a ordem não é relevante, podemos inserir um elemento construindo um outro conjunto em que o único elemento é o que pretendemos inserir e depois utilizar a função **UNION**.

```
{1, 8, 4} ∪ {3}
                                    {1, 3, 4, 8}
```

Maple:

No *Maple*, existindo algumas diferenças às quais já nos referimos no que diz respeito a listas e conjuntos, também a forma de inserir um elemento difere conforme estejamos a trabalhar com umas ou com outros.

A inserção de um elemento numa lista é feita por **[op(L),x]**. Note-se que o elemento é colocado no fim da lista **L**.

```
> [op([1,2,3,4,5]),13];
                                    [1, 2, 3, 4, 5, 13]
```

Por seu lado se estivermos a trabalhar com conjuntos, o operador **union** resolve o problema já referido. De modo análogo ao que já vimos no *Derive*.

```
> {1,1,2,3,5,8,13}union{21};
                                    {1, 2, 3, 5, 8, 13, 21}
```

Mathematica:

Pelo emprego das funções **Append** e **Prepend** acrescentamos um elemento no fim e no início de uma lista respectivamente. Estas duas funções permitem que construamos passo a passo uma lista partindo de uma já existente.

```
Append [illegible] , 4
[illegible], 3, 4
```

```
Prepend [2, 3], a]
[a, 2, 3
```

Se tal for conveniente podemos chamar a lista que pretendemos **listaex** fazendo para tal uma atribuição imediata.

```
listaex = Append [2, 3], 4]
[2, 3, 4
```

Também tem toda a utilidade existir uma função que insira numa lista um dado elemento sem que a posição seja forçosamente a primeira ou a última. Tal é possível recorrendo à função predefinida **Insert** que coloca o segundo argumento da função na posição definida pelo terceiro argumento na lista constante no primeiro argumento.

```
listaex = Insert [listaex, bb, 2]
[2, bb, 2, 3, 4
```

3.6. - ELIMINAÇÃO DE ELEMENTOS

Derive:

A eliminação de um elemento faz-se recorrendo à função **DELETE**.

```
DELETE([2, 3, 5, 7, 9, 13], 5)
                                    [2, 3, 5, 7, 13]
```

Nos casos em que o elemento a apagar consta da primeira posição podemos sempre usar a função **REST**.

```
REST([2, 3, 5, 7, 9, 13])
                                    [3, 5, 7, 9, 13]
```

Para eliminar elementos de um conjunto utilizamos o atalho  .

```
{1, 8, 4} \ {8}
                                        {1, 4}
```

Maple:

Uma forma de conseguir extrair um elemento de um conjunto ou de uma lista é através da utilização da função **remove**. Esta função retira da lista ou conjunto todos os objectos que verificam uma dada condição.

```
> remove(isprime,[1,2,3,4,5,6,7,8]);
                                    [1,4,6,8]

> remove(has,[1,2,3,4,5,6],4);
                                    [1,2,3,5,6]
```

A função **has(f,x)** é um predicado de comparação sintáctica que dá *true* nos casos em que **f** contém a subexpressão **x**.

Temos ainda a possibilidade de apagar o elemento que está numa posição conhecida de uma lista.

Fazendo **subop(i=NULL,L)** estamos a apagar o elemento da lista **L** que está na posição **i**.

```
> subsop(3=NULL,[7,-8,7,4,-11]);
                                    [7,-8,4,-11]
```

De forma semelhante usando a função predefinida **minus** podemos retirar um dado elemento de um conjunto.

```
> {3,2,-4,0}minus{-4};
                                        {0, 2, 3}
```

Mathematica:

Há sempre a possibilidade de apagar ou retirar um dado (ou um conjunto de) elemento(s) de uma lista.

A função **Delete[lista,n]** apaga o elemento que está na posição **n** da **lista**. Se o número **n** for negativo a posição é contada a partir do fim.

```
?listaex
Global`listaex
listaex = {a, b, 2, 3, 4}

Delete[listaex, 3]
{a, b, 3, 4}
```

A função **Rest** fornece-nos a lista retirando o seu primeiro elemento e a função **Drop** permite retirar da lista um certo número de elementos.

```
Rest[listaex]
{b, 2, 3, 4}

Drop[listaex, 1]
{b, 2, 3, 4}

Drop[listaex, 2]
{2, 3, 4}
```

Também nesta última função se o segundo argumento for negativo, então a ordem é contada do fim pelo que os valores retirados são os últimos. Deste comentário conclui-se que a função **Rest** pode ser substituída pela função **Drop** desde que nesta última o segundo argumento seja **–1**.

3.7. - SUBSTITUIÇÃO DE ELEMENTOS

Derive:

Ao definir uma lista corremos o risco, como acontece em quase tudo o que fazemos, de nos enganarmos nomeadamente na introdução de um dado elemento.

Quando tal acontece é possível substituir unicamente o elemento em questão.

A função **REPLACE** permite que substituamos o elemento que está numa determinada posição.

```
primos := [2, 3, 5, 7, 9, 13]
                                    [2, 3, 5, 7, 9, 13]

REPLACE(11, primos, 5)
                                  [2, 3, 5, 7, 11, 13]
```

Se não fosse indicada a posição do elemento (no caso anterior 5) o *Derive* assumia que o elemento a substituir era o primeiro.

Maple:

Após definir uma lista podemos ainda alterar os seus elementos sem ter de definir novamente a lista. Para tal, basta conhecermos qual a posição do elemento a substituir bem como o novo elemento. Sendo estes dois do nosso conhecimento a utilização da função **subsop** resolve com facilidade o problema da substituição dos dois elementos.

Vejamos como alterar o terceiro elemento da lista **[3,5,-2,5,4]** por **7**.

```
> subsop(3=7,[3,5,-2,5,4]);
                                    [3, 5, 7, 5, 4]
```

Mathematica:

Pretendendo modificar um elemento de uma lista podemos utilizar a função **ReplacePart**. Esta função recebendo uma lista, um elemento e uma posição, substitui o elemento que está na posição referida pelo elemento pretendido. No exemplo que se segue substituiremos o nome “beatriz” que se encontra na segunda posição pelo nome “joana”.

```
ReplacePart[{ana, beatriz, carla}, joana, 2]
{ana, joana, carla}
```

3.8. - SELECÇÃO DE ELEMENTOS

Derive:

Uma função que também é importante é a função **SELECT** que selecciona os elementos que verificam um determinado predicado.

```
SELECT(k > 7, k, [2, 7, 8, 11, 0, -9])
                                        [8, 11]

SELECT(PRIME(k), k, 1, 35)
                          [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
```

Em ambos os casos a função deu-nos os elementos da lista que estavam em conformidade com os predicados.

Maple:

A selecção de um elemento de uma lista pode ser efectuada através da função **select**. Assim sendo conhecida a lista, basta-nos referir apenas o tipo de objectos que pretendemos seleccionar.

```
> select(isprime,[1,2,3,4,5,6,7,8,9,10,11,12,13]);
                                        [2, 3, 5, 7, 11, 13]
```

Mathematica:

A função **Take** possibilita que retiremos da lista um certo número de elementos. Tal como acontece com algumas das funções analisadas instantes atrás, se o segundo argumento for negativo, então a ordem é contada a partir do fim.

```
Take [2, 3, 5, 7, 11, 13, 17 , 3]
{2, 3, 5}

Take [2, 3, 5, 7, 11, 13, 17 , -3]
{11, 13, 17}
```

Usando uma função predefinida podemos também seleccionar os elementos de uma dada lista que verificam um determinado predicado. Enquanto que as funções anteriores seleccionavam os elementos pela posição em que estes se encontravam na lista, a função **Select** selecciona-os de acordo com as propriedades dos seus elementos.

```
Select [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 , EvenQ]
{2, 4, 6, 8, 10}

Select [Table[Prime[k], {k, 1, 25}] , PrimeQ]
{2, 3, 5, 7, 11, 13, 17, 19, 23}
```

3.9. - COMPRIMENTO DE UMA ESTRUTURA

Derive:

Quando trabalhamos com listas é útil conhecermos quantos elementos tem essa estrutura. Através do uso da função **DIM** ficamos a saber com quantos elementos estamos a trabalhar.

```
DIM([2, 3, -7, 11, 4])
```

5

Se estivermos na área das matrizes podemos utilizar a mesma função para indagar acerca das dimensões da matriz. Vejamos como obter o número de linhas da matriz **[[a,b,c],[d,e,f]]**.

$$\mathrm{DIM}\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

2

Para saber o número de colunas basta aplicar a mesma função ao primeiro elemento da matriz.

$$\mathrm{DIM}\left(\mathrm{ELEMENT}\left(\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}, 1\right)\right)$$

3

Portanto o número de colunas neste caso concreto é de três.

Maple:

No *Maple* a determinação do número de elementos de um conjunto ou lista faz-se recorrendo à função **nops**.

```
> conjunto:={1,0,-1,1,3,3};
```

$$conjunto := \{0, -1, 1, 3\}$$

```
> nops(conjunto);
```

4

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

5

Refira-se que no *Maple* existe ainda a função **length**. Esta função, caso o argumento seja um inteiro, fornece-nos o número de dígitos do mesmo. Nos casos em estamos a trabalhar com *strings* dá-nos o número de caracteres utilizados. Nos outros casos o **length** de cada expressão é calculado recursivamente e adicionado com o número de palavras usado para representar a expressão.

```
> length(457);
                                3
> length(marcia);
                                6
```

Mathematica:

Quando estamos a trabalhar com listas é útil conhecermos o número de elementos que as mesmas contêm. Tal necessidade é evidente quando, na elaboração de um programa se pretende começar a partir do último elemento. Ora, conhecendo o comprimento da lista é possível utilizarmos esse número, que tem forçosamente de ser um inteiro, para através do uso da selecção de componente, nos referirmos a esse último elemento.

No *Mathematica* a função que nos dá o comprimento de uma lista é a função **Length**.

```
Length[{1, 2, 3}]
3

Length[{{1, 2}, 7, 4, {5, 6}, 7}]
4
```

O último *input* significa que a nossa lista tem 4 elementos onde como sabemos dois deles são ainda listas. É claro que se fosse necessário, poderíamos ainda saber qual o comprimento de cada um dos elementos da lista inicial, utilizando para tal uma outra função – **Map**. Desta função falaremos em mais pormenor um pouco mais à frente. No entanto, podemos desde já dizer que este funcional faz com que uma determinada função seja aplicada a cada um dos elementos de uma dada lista.

```
Map[Length, {{1, 2}, 7, 4, {5, 6}, 7}]
{2, 0, 0, 3}
```

3.10. - JUNÇÃO DE DUAS ESTRUTURAS

Derive:

Podemos juntar duas ou mais listas recorrendo à função **APPEND**.

```
APPEND([1, 2], [5, -1], [0, 1])
                                    [1, 2, 5, -1, 0, 1]

APPEND [1  2  0]
       [3  2  4]
       [7  5  6]
       [4  1  5]
                          [1, 2, 0, 3, 2, 4, 7, 5, 6, 4, 1, 5]
```

Temos a mesma possibilidade se estivermos a trabalhar com conjuntos utilizando a função **UNION** ou o atalho ∪ presente na parte inferior do *Derive*. Introduzimos a expressão **{1, 2} UNION {1, 3}** e mandamo-la avaliar.

```
{1, 2} ∪ {1, 3}
                                              {1, 2, 3}

{1, 2, 0, -1} ∪ {0, 2, 3}
                                         {-1, 0, 1, 2, 3}
```

Maple:

A reunião de dois conjuntos definidos em *Maple* pode fazer-se através da função predefinida **union**. Esta função pode utilizar-se colocando a função entre os conjuntos que pretendemos reunir (que não têm de ser apenas dois) ou então utilizando a notação usual de uma função, onde os conjuntos são indicados como os argumentos da função **union**. No último caso a função deve ser invocada entre dois acentos graves **`union`**.

```
> {2,-3,7,11}union{4,-5,0};
                                    {0, 2, -3, 4, -5, 7, 11}

> `union`({a,b,c},{a,d});
                                         {a, b, c, d}
```

Mathematica:

As estruturas representadas em *Mathematica* como listas podem ser reunidas numa só lista pelo uso da função ***Join***. Esta função que pode ser invocada com mais do

que dois argumentos cria uma nova lista constituída pela junção de duas ou mais listas. Ainda que ocorram elementos repetidos, estes são sempre representados na nova lista.

```
Join [illegible], 7
[illegible], 3, 2, 4, 7

Join [illegible]
[illegible], 5, 7, [illegible], 1, 8
```

É possível ainda retirar as chavetas interiores se utilizarmos a função **Flatten**.

```
Flatten [illegible]
[illegible], 5, 7, 4, 1, 8
```

Embora o *Mathematica* não disponibilize directamente conjuntos, que têm de ser representados por listas, existem diversas operações específicas de conjuntos (representados por listas) nomeadamente a união, intersecção, complemento, ...

```
Union [illegible], 5, -2
[illegible], 1, 2, 3, 5

Intersection [illegible], 5, -2
[illegible]

Complement [illegible], c, d, e, f , [illegible], [illegible]
[illegible]
```

3.11. - FUNÇÃO PERTENCE

Derive:

Conhecido um elemento e uma lista ou conjunto podemos descobrir se esse elemento faz parte da lista ou conjunto se utilizarmos a função **MEMBER?**. Esta função retorna **true** nos casos em que o elemento faz parte da lista ou conjunto e **false** no caso contrário. Vejamos alguns exemplos onde aplicámos a referida função.

```
MEMBER?(2, [8, 2, 3, 4])
                                                    true

MEMBER?(5, [1, 4, 3, 4])
                                                    false

MEMBER?(7, {7, 4, 3, 4})
                                                    true

MEMBER?(5, {11, 7, 4, 3, -4})
                                                    false
```

Maple:

Por meio do comando **member** é possível saber se um dado elemento faz ou não parte de um conjunto ou de uma lista.

```
> member(4,[4,5,7]);
                                                    true

> member(a,[3,5,9,7]);
                                                    false
```

A função pode ainda ser invocada com um terceiro argumento (uma expressão) no qual fica guardado a primeira posição em que o elemento ocorre.

```
> member(w,[x,y,w,u],'k');
                                                    true


> k;
                                                    3
```

Nos conjuntos tudo se passa de maneira semelhante.

```
> member(3,{1,3,4,5});
                                                    true
```

```
> member(1,{2,-3,-5,-1});
                                                    false
```

Mathematica:

A função correspondente em *Mathematica* é a função **MemberQ**. Esta função aplica-se com uma lista e um elemento como argumentos e por esta ordem. Assim o predicado **MemberQ** dar-nos-á **True** nos casos em que o elemento pertença à lista constante no primeiro argumento.

```
MemberQ[{a,b,c}, a]
True

MemberQ[{1,3,5}, 2]
False
```

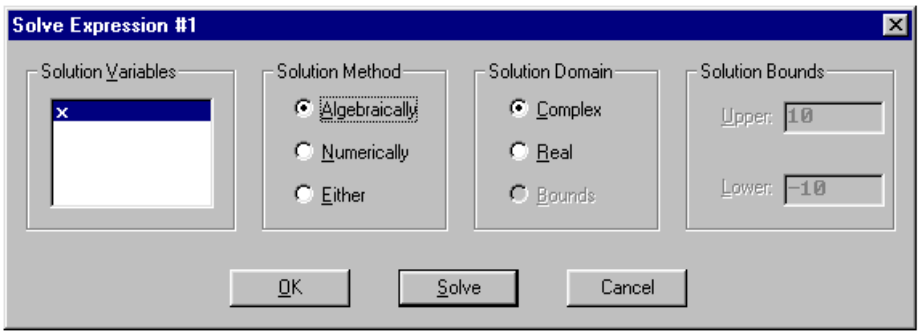
3.12. – ANÁLISE MATEMÁTICA

3.12.1. - EQUAÇÕES

Derive:

A resolução de equações no *Derive* pode ser feita recorrendo ao menu *Solve* e ao submenu *Algebraically*. Podemos tirar partido do atalho  se introduzirmos a expressão em primeiro lugar e o utilizarmos de seguida. Nessa situação surge uma janela na qual podemos especificar o tipo de resolução que pretendemos para cada caso.

$$x^2 + 7 \cdot x + 10 = 0$$



Carregando em OK e usando o atalho  obtemos as soluções para a nossa equação.

$$\text{SOLVE}(x^2 + 7 \cdot x + 10 = 0,\ x)$$

$$x = -5 \vee x = -2$$

$$\text{SOLVE}(x^3 + 2 \cdot x^2 - 29 \cdot x + 42 = 0,\ x)$$

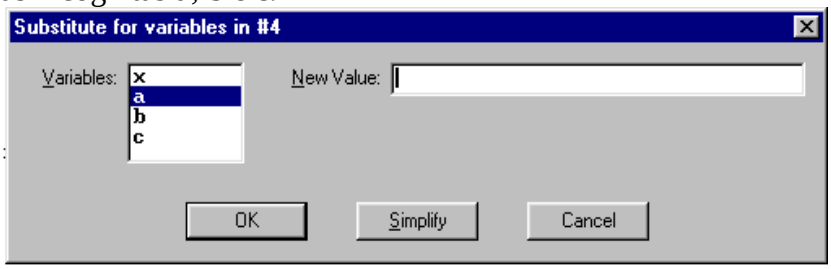
$$x = -7 \vee x = 3 \vee x = 2$$

Mas, não ficamos por aqui no que respeita à resolução de equações algébricas, o *Derive* também é capaz de resolver simbolicamente uma equação arbitrária. Neste caso pedimos para resolver a equação de segundo grau.

$$\text{SOLVE}(a \cdot x^2 + b \cdot x + c = 0,\ x)$$

$$x = \frac{\sqrt{(b^2 - 4 \cdot a \cdot c)} - b}{2 \cdot a} \vee x = -\frac{\sqrt{(b^2 - 4 \cdot a \cdot c)} + b}{2 \cdot a}$$

Agora, mediante este último *output*, que não é mais do que a fórmula resolvente das equações de segundo grau, e sem usar a função **SOLVE**, podemos resolver qualquer equação da forma $\mathbf{ax^2 + bx + c = 0}$. Para tal basta que substituamos os valores de **a**, **b** e **c** na solução geral. Tal pode ser feito utilizando o atalho  . Aí temos de referir quais os valores para as incógnitas **a**, **b** e **c**.



Imediatamente o sistema substitui nas expressões os novos valores que depois de mandado avaliar dará as soluções da nossa equação. Neste caso concreto **a=1**, **b=-8** e **c=15**.

$$x = \frac{\sqrt{((-8)^2 - 4 \cdot 1 \cdot 15)} - -8}{2 \cdot 1} \vee x = -\frac{\sqrt{((-8)^2 - 4 \cdot 1 \cdot 15)} + -8}{2 \cdot 1}$$

$$x = 5 \vee x = 3$$

Mesmo quando as raízes são complexas o *Derive* é capaz de nos dar as soluções pretendidas.

$$x^2 + 2 = 0$$

$$\text{SOLVE}(x^2 + 2 = 0,\ x)$$

$$x = -\sqrt{2} \cdot \hat{\imath} \vee x = \sqrt{2} \cdot \hat{\imath}$$

Para resolver um sistema de equações lineares podemos utilizar o menu *Solve* e o submenu *System*. Neste primeiro exemplo procuramos a solução de um sistema possível e determinado.

$$\text{SOLVE}\left(\left[x + \frac{3}{4} \cdot y + 3 \cdot z = 0,\ 3 \cdot x + 4 \cdot y - z = 0,\ \frac{5}{2} \cdot x + 2 \cdot x - \frac{1}{2} \cdot z = 4\right],\ [x, y, z]\right)$$

$$\left[x = \frac{204}{233}\quad y = -\frac{160}{233}\quad z = -\frac{28}{233}\right]$$

Portanto a solução do sistema é a obtida.

Tratando-se de sistemas indeterminados não conseguimos chegar a uma única solução mas a um conjunto delas. Vejamos aquilo a que nos referimos.

```
SOLVE([x + 2·y = 4, 3·x + 6·y = 12], [x, y])
                                        [x + 2·y = 4]

SOLVE([x + 2·y = 4, 3·x + 6·y = 12], [x])
                                        [x = 4 - 2·y]
```

Neste caso as soluções são todos os pares de números da forma (**4-2y,y**).

Se o sistema for impossível, não existem quaisquer soluções e portanto a lista das soluções é vazia.

```
SOLVE([x + 2·y = 4, 3·x + 6·y = 11], [x, y])
                                              []
```

Vamos agora analisar qual a resposta que o *Derive* dá quando confrontado com equações que envolvem outras funções como por exemplo as funções trigonométricas.

```
SOLVE(SIN(x) = 0, x)
                                   x = -π ∨ x = π ∨ x = 0
```

Esta equação tem uma infinidade de soluções que são todas aquelas que se podem escrever na forma $\mathbf{x = k\pi}$, com $\mathbf{k} \in Z$. Assim as soluções obtidas estão correctas mas, o programa não consegue determiná-las todas.

Analisemos ainda mais um caso.

```
SOLVE(COS(x) = SIN(EXP(x)), x)
```

$$\hat{e}^x - x = \frac{5\cdot\pi}{2} \vee \hat{e}^x - x = -\frac{3\cdot\pi}{2} \vee \hat{e}^x - x = \frac{\pi}{2} \vee \hat{e}^x + x = \frac{5\cdot\pi}{2}$$

$$\vee \hat{e}^x + x = -\frac{3\cdot\pi}{2} \vee \hat{e}^x + x = \frac{\pi}{2}$$

Aqui também não obtivemos uma solução satisfatória.

Maple:

A resolução de equações algébricas é possível no *Maple* recorrendo à função predefinida **solve**. Através do uso desta função conseguimos ultrapassar em muitos casos o trabalho monótono dos cálculos.

A função **solve** recebe como argumentos um conjunto de uma ou mais equações e tenta resolvê-las em ordem a um conjunto de incógnitas especificadas pelo segundo argumento da função.

```
> solve({x^2-9*x+14=0},{x});
                                        {x = 2}, {x = 7}
```

Vejamos agora como o *Maple* lida simbolicamente com equações. Para isso vamos procurar a solução para uma equação de segundo grau arbitrária.

```
> solve({a*x^2+b*x+c=0},{x});
```

$$\{x = \frac{1}{2}\frac{-b+\sqrt{b^2-4ac}}{a}\}, \{x = \frac{1}{2}\frac{-b-\sqrt{b^2-4ac}}{a}\}$$

Como se pode ver, o *Maple* retorna cada uma das possíveis soluções dentro de chavetas, o mesmo será dizer que cada solução forma um conjunto.

Como se pode constatar conseguimos encontrar as soluções para uma equação mesmo que as mesmas sejam complexas.

```
> solve(x^2+5=0,{x});
```

$$\{x = I\sqrt{5}\}, \{x = -I\sqrt{5}\}$$

Podemos também usar o comando **solve** para resolver sistemas de equações.

```
> solve({x+2*y=3,y+1/x=1},{x,y});
```

$$\{x = -1, y = 2\}, \{x = 2, y = \frac{1}{2}\}$$

Nem sempre são necessárias as chavetas envolvendo as equações ou as variáveis, mas usando-as obriga o *Maple* a fornecer a solução como um conjunto o que traz por vezes algumas vantagens.

```
> eqns:={x+2*y=3,y+1/x=1};
```

$$eqns := \{x + 2y = 3, y + \frac{1}{x} = 1\}$$

```
> soln:=solve(eqns,{x,y});
```

$$soln := \{x = -1, y = 2\}, \{x = 2, y = \frac{1}{2}\}$$

Obtivemos portanto duas soluções. Podemos verificar se as soluções realmente são estas, substituindo na equação original.

```
> subs(soln[1],eqns);
```

$$\{1 = 1, 3 = 3\}$$

```
> subs(soln[2],eqns);
```

$$\{1 = 1, 3 = 3\}$$

Portanto as soluções verificam as condições exigidas pelo sistema. Estes dois pares de soluções encontrados são as únicas soluções para o sistema. Trata-se de um sistema possível e determinado. Quando o sistema a resolver é indeterminado, o *Maple*

fornece uma expressão algébrica que nos permite obter uma das incógnitas à custa da outra.

```
> solve({1/2*x+3*y=7,4*x+24*y=56},{x,y});
```

$$\{y = y, x = -6y + 14\}$$

Sendo o sistema a resolver impossível, o *Maple* simplesmente não retorna qualquer *output* o que dá a ideia que não conseguiu concluir o que lhe foi pedido.

```
> solve({3*x+y=0,3/2*x+1/2*y=7},{x,y});
```

No caso anterior não obtivemos qualquer *output*.

Vamos analisar agora o comportamento do *Maple* quando confrontado com equações envolvendo funções trigonométricas.

```
> solve(sin(x)=1,{x});
```

$$\{x = \frac{1}{2}\pi\}$$

Tal como acontecia com o *Derive* obtivemos uma solução mas, como é do nosso conhecimento as soluções são todas aquelas que se podem escrever na forma $x = \frac{1}{2}\pi + 2k\pi,\ k \in \mathrm{Z}$.

Vejamos ainda uma última situação.

```
> solve(cos(x)=sin(exp(x)),{x});
```

$$\{x = \mathrm{RootOf}\left(\_Z - \ln\left(\frac{1}{2}\pi - \_Z\right)\right)\}$$

Sem ter obtido uma solução concreta podemos sempre partir para o cálculo numérico e pedir ao programa que faça uma aproximação do valor surgido.

```
> evalf(solve(cos(x)=sin(exp(x)),{x}));
```

$$\{x = .2660265834\}$$

Mathematica:

Utilizando o *Mathematica* é possível encontrar de forma eficiente as soluções de muitas das equações que nos surgem.

Assim, encontrar uma solução para uma equação do segundo grau não representa qualquer dificuldade para o *Mathematica*.

Antes de entrarmos num exemplo concreto, convém lembrar que ao fazermos **x=3** estamos a efectuar uma atribuição imediata, ou seja, estamos a fazer com que a variável **x** passe a guardar o valor **3**. Assim para designar o teste de igualdade temos de usar " = = ".

```
Solve[2x^3+3*x^2-3*x-2==0, x]
```

$$\left\{x \to 2,\ x \to -\frac{1}{2},\ x \to -1\right\}$$

```
Solve[a*x^2+b*x+c==0, x]
```

$$\left\{x \to \frac{-b-\sqrt{b^2-4ac}}{2a},\ x \to \frac{-b+\sqrt{b^2-4ac}}{2a}\right\}$$

Mesmo quando as soluções são complexas conseguimos encontrá-las.

```
Solve[x^2+7==0, x]
```

$$\left\{x \to -i\sqrt{7},\ x \to i\sqrt{7}\right\}$$

De forma muito semelhante procedemos ao pretender resolver um sistema de equações lineares.

```
Solve[x+y==8, -2x+5y==17, {x, y}]
```

$$\left\{\left\{x \to -\frac{28}{23},\ y \to \frac{67}{23}\right\}\right\}$$

Quando o sistema a resolver é indeterminado o programa, não conseguindo determinar uma única solução, emite uma mensagem de erro onde diz que não lhe é possível dar uma solução em função de todas as variáveis pretendidas.

```
Solve[2x+6y==5, 15x+36y==30, {x, y}]
Solve::svars : Equations may not
   give solutions for all "solve" variables.
```

$$\left\{\left\{x \to 2-\frac{12y}{5}\right\}\right\}$$

Tendo em conta o erro podemos então mandar resolver o sistema apenas em ordem a uma das variáveis. No entanto este procedimento não acrescenta muito ao que já sabíamos.

```
Solve[2x+6y==5, 15x+36y==30, x]
```

$$\left\{\left\{x \to -\frac{2}{5}(5+6y)\right\}\right\}$$

Quando o sistema é impossível a solução apresentada é a lista vazia o que indica de forma clara que não existem quaisquer soluções.

```
Solve[x+2y==7, 3x+6y==3, {x, y}]
```

$$\{\}$$

É claro que nem sempre conseguimos obter todas soluções que pretendemos. No caso que se segue o *Mathematica* fornece-nos apenas duas soluções quando sabemos que existem uma infinidade delas.

```
Solve[Cos[x]^2==1, x]
```

```
Solve::ifun : Inverse functions are being used
   by Solve, so some solutions may not be found.
```

$$\left\{\left\{x \to -\frac{\pi}{2}\right\}, \left\{x \to \frac{\pi}{2}\right\}\right\}$$

Analisemos mais um caso em que o *Mathematica* se vê impossibilitado de dar-nos uma solução.

Solve [redacted]

```
Solve::tdep :
 The equations appear to involve the variables to be
   solved for in an essentially non-algebraic way.
```

Solve [redacted]

Quando tal acontece podemos ainda recorrer à função **FindRoot** que tenta encontrar uma solução aproximada para a equação pretendida. Para aplicar esta função é necessário introduzir o valor do qual se pretende a solução mais aproximada.

FindRoot [redacted]

$\{x \to 0.90762\}$

Desta forma obtivemos a solução aproximada para a equação introduzida que mais próximo do **1** está.

3.12.2. - DERIVAÇÃO

Derive:

A derivada de uma função pode obter-se rapidamente utilizando o operador **DIF**. Neste, o primeiro argumento representa a expressão, o segundo argumento a variável e o terceiro argumento que pode ser omitido, representa a ordem da derivada pretendida.

Nos casos que apresentamos de seguida introduzimos as expressões **DIF(SIN(x),x)** e **DIF(LOG(x),x,2)** já que queríamos calcular a derivada da função seno e a segunda derivada da função logaritmo. Carregamos em ENTER e mandamos avaliar as derivadas carregando em =.

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

$$COS(x)$$

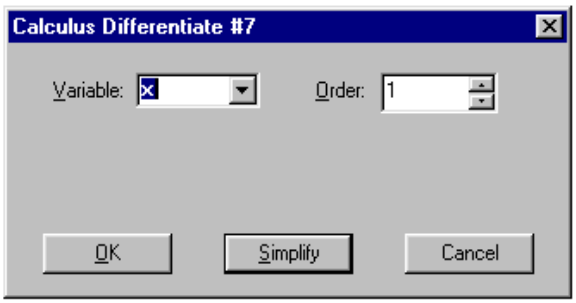
$$\left(\frac{d}{dx}\right)^2 LOG(x)$$

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

Com a mesma finalidade podemos recorrer ao menu *calculus* e ao submenu *differentiate* ou de forma mais abreviada utilizando o atalho ∂.

Nestes casos temos de introduzir a expressão da qual se pretende calcular a derivada e depois utilizar o atalho referido. Na zona de trabalho surge uma janela a partir de onde podemos especificar qual a variável e a ordem da derivada pretendida.

TAN(x)



Depois tudo se processa como na situação anterior.

```
TAN(x)
```

$$\frac{d}{dx}\,\mathrm{TAN}(x)$$

$$\frac{1}{\mathrm{COS}(x)^2}$$

Maple:

Dada uma função **f** de um argumento, **D(f)** determina a sua derivada. O resultado da aplicação do operador **D** é uma função com um único argumento.

```
> D(cos);
```

$$-\sin$$

A composição de duas funções f e g é denotada por f@g.

```
> D(cos@g);
```

$$(-\sin)@g\,D(g)$$

A utilização do operador **D** continua a ser possível quando as funções têm mais do que um argumento. Nesses casos surge a necessidade de referir qual dos argumentos deve ser tratado como variável. Aí o que a função **D** faz é calcular a derivada parcial em ordem ao argumento que foi considerado como variável.

Comecemos por definir a função **f**.

```
> f:=(x,y)->sin(x)* sin(y);
```

$$f := (x,y) \to \sin(x)\,\sin(y)$$

Consideremos em primeiro lugar que a nossa variável é o **x** (primeiro argumento da função **f**).

```
> D[1](f);
```

$$(x,y) \to \cos(x)\,\sin(y)$$

O resultado da aplicação do operador **D** a **f** é uma função definida pelo último *output*. Se a variável fosse o segundo argumento da função o resultado já não seria o mesmo. Vejamos:

```
> D[2](f);
```

$$(x,y) \to \sin(x)\,\cos(y)$$

A derivação de expressões em ordem a uma variável faz-se pela função **diff**.

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

$$-1-\cot(x)^2$$

```
> diff(x^2*sin(1/x),x);
```

$$2x\sin\left(\frac{1}{x}\right)-\cos\left(\frac{1}{x}\right)$$

O operador **diff** pode ser invocado com mais do que dois argumentos. Ao introduzirmos por exemplo **diff(f(x),x,y)** estamos numa situação inteiramente análoga à que teríamos fazendo **diff(diff(f(x),x),y)**.

```
> diff((1+2*y^3)*x^2,x,y);
```

$$12y^2x$$

```
> diff(f(x,y),x,y);
```

$$\frac{\partial^2}{\partial y\,\partial x}f(x,y)$$

Existe ainda o operador **Diff** com "D" maiúsculo que nos fornece um *output* semelhante ao que é usual para designar as derivadas parciais.

```
> Diff(x^3,x);
```

$$\frac{\partial}{\partial x}x^3$$

Repare-se que o operador **Diff** mantém a expressão introduzida sem efectuar o cálculo da derivada especificada. Trata-se de uma função que se destina na grande maioria dos casos a uma melhor apresentação.

A utilização de **implicitdiff(f,y,x)** fornece-nos a derivada de uma função definida por uma equação.

```
> f:=exp(y)+y=x+5*z;
```

$$f:=e^y+y=x+5z$$

```
> implicitdiff(f,y,x);
```

$$\frac{1}{e^y+1}$$

```
> implicitdiff(f,y,z);
```

$$\frac{5}{e^y+1}$$

Mathematica:

O operador **Derivative** determina a função derivada de uma função.
Vejamos como obter a primeira derivada da função seno.

```
Derivative[1][Sin]
Cos[#1]&
```

O número **1** inserido no primeiro parênteses recto quer dizer que estamos interessados em calcular a primeira derivada. Caso o nosso objectivo fosse a segunda derivada bastaria colocar o número **2**, onde no caso anterior colocámos o **1**.
Podemos também representar a derivada de uma função na forma usual.

```
Log''[x]
-1/x^2
```

Podemos obter a expressão da derivada de uma dada função utilizando o operador **D**.

```
D[Tan[x], x]
Sec[x]^2
```

```
D[Cos[x^2 + 2 x], x]
-4 x^2 Cos[x^2] - 4 Sin[x^2]
```

Realce-se aqui que o resultado de aplicar o operador **Derivative** é uma função e que o operador **D** dá-nos a expressão da função derivada. Daí que calcular a derivada de uma função num ponto terá que ter em conta esse facto. Enquanto que utilizando o operador **Derivative** ou **‘** basta aplicar a função obtida no ponto três, usando o operador **D** temos que substituir **x** pelo valor **3**. No fim como *output* obtemos o mesmo resultado.

```
Log'[3]
1/3
```

```
D[Log[x], x] /. x → 3
1/3
```

As derivadas parciais em ordem a **x** podem ser obtidas através de δ_x. A utilização deste símbolo faz-se recorrendo às *palettes*, mais concretamente ao atalho  .

```
∂x x^3 * Cos[x * y]
3 x^2 Cos[x y]
```

3.12.3. - INTEGRAÇÃO

Derive:

A determinação de um integral de uma dada função faz-se recorrendo ao operador **INT**. **INT(u,x)** permite a obtenção do integral indefinido da expressão **u** na variável **x**.

$$\int \mathrm{SIN}(x)\,dx$$

$$-\mathrm{COS}(x)$$

Nos casos em pretendemos calcular o integral definido temos de adicionar aos argumentos do operador outros dois que representam o limite inferior e limite superior. Para calcular o integral da função exponencial no intervalo [0,10], introduzimos a expressão **INT(EXP(x), x, 0, 10)**. Tal como no caso anterior, o *Derive* ao passar a expressão para a zona de trabalho modifica-a para a forma usual.

$$\int_{0}^{10} \mathrm{EXP}(x)\,dx$$

$$\hat{e}^{10} - 1$$

Podemos também optar por calcular um integral quer seja definido ou indefinido através do atalho  existente na barra de ferramentas. Para tal temos de em primeiro lugar introduzir a expressão a integrar e depois utilizar o atalho.

Introduzindo o *input* seguinte, e usando o referido atalho

$$\mathrm{LN}(x)$$

surge uma janela na qual especificamos as características do integral a calcular.

Depois carregamos em OK e mandamos avaliar a expressão introduzida como nos casos anteriores.

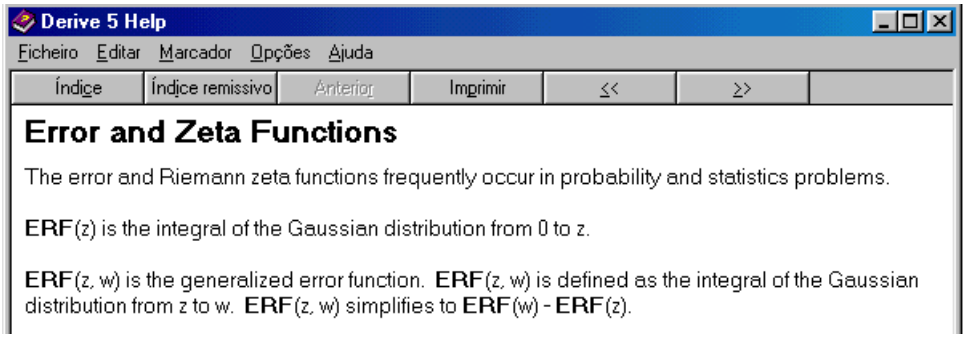
$$\int \mathrm{LN}(x)\ dx$$

$$x \cdot \mathrm{LN}(x) - x$$

$$\int \mathrm{EXP}(-x^2)\ dx$$

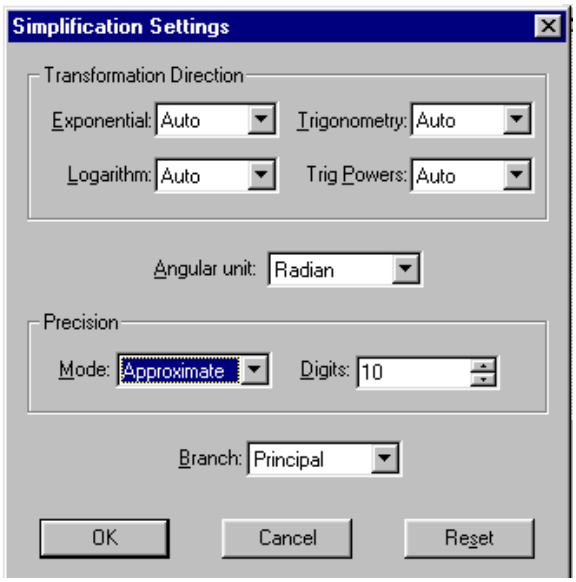
$$\frac{\sqrt{\pi} \cdot \mathrm{ERF}(x)}{2}$$

Consultando o *Help* ficamos a saber que **ERF(X)** é o integral da distribuição gaussiana entre **0** e **x**.



Podemos obter valores aproximados para um dado integral quando não é possível ao *Derive* dar-nos um valor exacto.

Recordamos aqui que para obter uma aproximação para uma dada expressão utilizamos o menu *Declare*, o submenu *Simplification Settings* e alteramos o campo da precisão para *Approximate*.



Na zona de trabalho surge a indicação da alteração feita.

```
Precision := Approximate
```

Mandando avaliar o integral pretendido obteremos uma aproximação.

$$\int_{-\infty}^{\infty} EXP(-x^2)\,dx$$

$$1.772453850$$

Voltemos a trabalhar com valores exactos.

```
Precision := Exact
```

$$Exact$$

Podemos calcular integrais duplos recorrendo ao duplo uso dos métodos já analisados um dos quais poderá ser **INT(INT(f,x), y)**. No exemplo que se segue mandamos calcular o integral duplo em ordem a **x** da função **x**. É evidente que neste caso podemos mandar integrar em função de variáveis distintas.

$$\int\int x\,dx\,dx$$

$$\frac{x^3}{6}$$

$$\int\int x \cdot SIN(y)\,dx\,dy$$

$$-\frac{x^2 \cdot COS(y)}{2}$$

Outro método para fazer o mesmo trabalho é usar a função já analisada **DIF** em que a ordem do integral pretendido é o simétrico da ordem da derivada.

$$\left[\frac{d}{dx}\right]^{-2} x$$

$$\frac{x^3}{6}$$

Maple:

Da mesma forma que o operador **Diff** permitia uma visualização semelhante à que é habitual, também o operador **Int** tem essa função **Int(f,x)** e **Int(f,x=a..b)** são duas possibilidades para representar com o símbolo de integral $\int$ respectivamente o integral indefinido e o integral definido entre **a** e **b**.

Para utilizarmos o operador anterior na forma apresentada temos de carregar o pacote correspondente (*student*) sem o que não será possível visualizar os integrais definidos anteriormente. Podemos também optar por utilizar o operador na forma **student[Int]**.

```
> with(student):
Warning, new definition for D
> Int(x^2,x=0..5);
```

$$\int_0^5 x^2\,dx$$

Podemos obrigar o *Maple* a avaliar a expressão mediante a utilização da função **value**.

```
> value(Int(x^2,x=0..5));
```

$$\frac{125}{3}$$

A aplicação do operador **int(f,x)** revela-se de maior utilidade já que é de mais fácil aplicação quando pretendemos calcular o integral de uma dada função.

```
> int(sin(x),x);
```

$$-\cos(x)$$

```
> int(sin(x),x=0..Pi);
```

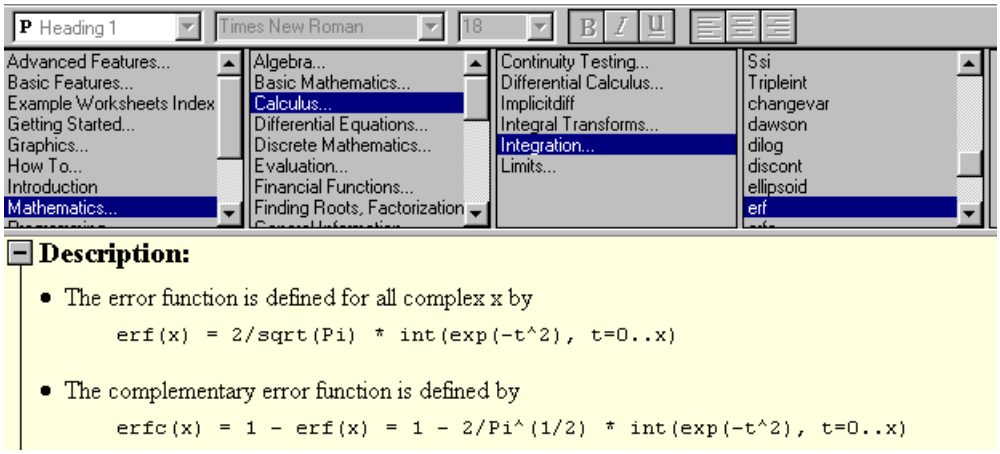
$$2$$

```
> int(exp(-x^2),x);
```

$$\frac{1}{2}\sqrt{\pi}\,\mathrm{erf}(x)$$

Consultando a ajuda disponibilizada pelo *Maple* tomamos conhecimento do que representa a função anterior.

```
> ?erf
```



A nível da visualização o *Maple* disponibiliza-nos a integração por partes através do pacote *student* e do operador **intparts**.

```
> intparts(Int(x^3*ln(x),x),ln(x));
```

$$\frac{1}{4}\ln(x)\,x^4 - \int \frac{1}{4}x^3\,dx$$

Este operador é invocado na forma **intparts(f,u)** onde **f** é uma expressão da forma **Int(u*dv,x)** e **u** é o factor do integral que se pretende que seja derivado.

Nos casos em que um integral definido não é avaliado podemos optar por fazer uma integração numérica.

```
> evalf(int(exp(-x^2),x=0..1));
```

.7468241330

Podemos utilizar **infinity** e **- infinity** para definir o intervalo de integração.

```
> evalf(int(exp(-x^2),x=-infinity..infinity));
```

1.772453851

Existe um segundo argumento da função anterior que indica o número de dígitos com que pretendemos a aproximação.

```
> evalf(int(exp(-x^2),x=-infinity..infinity),5);
```

1.7725

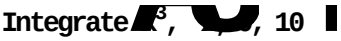
Mathematica:

O operador da integração no *Mathematica* é representado na forma usual ou então através de **Integrate**. Usando as *palettes* podemos utilizar a notação usual quer estejamos a falar de integrais definidos ou indefinidos.

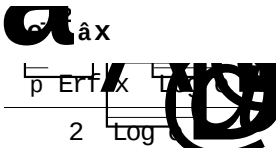
```
2500
```

Utilizando o operador **Integrate** especificamos no segundo argumento, constituído por uma lista, a variável juntamente com os limites inferior e superior. Optando por colocar no segundo argumento apenas a variável estamos a definir um integral indefinido.

```
Sin
```



```
2500
```



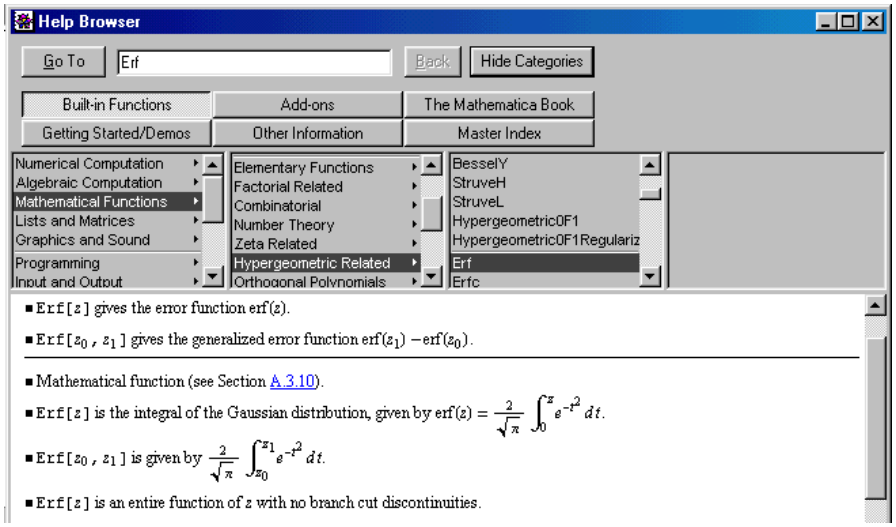
```
?Erf
Erf
```



```
 gives the error function erf
```

```
 gives
  the generalized error function erf
```


Consultando o *Help Browser* podemos obter mais algumas informações sobre esta função.



Podemos obter uma aproximação para um integral através de **Nintegrate**.

```
NIntegrate Exp[-x^2], 1, 1
1.49365
```

```
NIntegrate Exp[-x^2], ¥, ¥
1.77245
```

Podemos ainda determinar o resultado de um integral duplo utilizando o operador **Integrate** na forma **Integrate[f,{x, xmin,xmax},{y,ymin,ymax}]** que nos dará $\int_{xmin}^{xmax} dx \int_{ymin}^{ymax} f\ dy$.

```
x^2 â y â x
```

$$\frac{x^3 y^2}{6}$$

```
Integrate[x^2, x, y]
```

$$\frac{x^3 y^2}{6}$$

3.12.4. - SOMATÓRIOS E SÉRIES

Derive:

A determinação de um somatório pode ser feita através da função **Sum**.

No caso seguinte introduzimos a expressão **Sum(i,i,1,100)**. Automaticamente o *Derive* "transforma" o nosso *input* e regista no local de trabalho o que de seguida se mostra.

$$\sum_{i=1}^{100} i$$

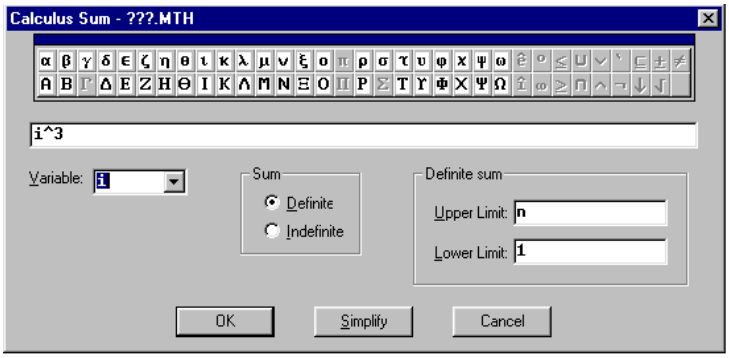
$$5050$$

Se invocarmos a função **SUM** com um só argumento, ou seja, com uma lista, esta retorna a soma dos elementos da mesma. Para exemplificar o que nos referimos introduzimos a expressão **SUM([7,2,-3])**.

$$\Sigma([7, 2, -3])$$

$$6$$

Podemos ainda calcular um somatório recorrendo ao atalho existente na barra de ferramentas. Nesse caso surge no ecrã uma janela na qual temos de especificar o somatório que pretendemos calcular.



$$\sum_{i=1}^{n} i^3$$

$$\frac{n^2 \cdot (n+1)^2}{4}$$

O cálculo de uma série pode ser feita nos mesmos moldes.

$$\sum_{i=0}^{\infty} \frac{1}{i!}$$

$$\hat{e}$$

Não se tratando de uma série convergente o *Derive* dá como *output* ∞ ou - ∞ conforme o caso, isto nos casos em que a série "converge" para infinito.

$$\sum_{i=0}^{\infty} i$$

$$\infty$$

$$\sum_{i=0}^{\infty} -i$$

$$-\infty$$

Analisando o comportamento do *Derive* quando confrontado com uma série que depende da paridade do limite superior não nos é apresentado um *output* satisfatório.

$$\sum_{i=1}^{\infty} (-1)^i$$

$$\frac{\mathrm{SIN}(\infty)}{2} - \frac{1}{2}$$

Considerando alguns casos concretos podemos verificar que a série na realidade é divergente.

$$\sum_{i=1}^{120} (-1)^i$$

$$0$$

$$\sum_{i=1}^{123} (-1)^i$$

$$-1$$

Maple:

A determinação de uma soma de um conjunto de números faz-se no *Maple* recorrendo à função **add**.

```
> add(i,i=1..7);
```

$$28$$

```
> add(i,i=[10,7,3,1,4]);
```

$$25$$

Existe ainda a função **sum** que é utilizada para a soma simbólica. Devemos utilizar esta função quando pretendemos obter uma fórmula para o somatório definido ou indefinido. Os argumentos da função devem ser usados entre dois apóstrofes (‘’).

```
> sum('k^2', 'k'=0..4);
```

$$30$$

```
> sum('k^2','k');
```

$$\frac{1}{3}k^3-\frac{1}{2}k^2+\frac{1}{6}k$$

O *output* anterior forneceu-nos uma expressão válida para calcular a soma de quadrados perfeitos. Assim, para obtermos a soma dos cinco primeiros quadrados perfeitos (considerando que o zero também é um quadrado perfeito) e utilizando o resultado que obtivemos basta-nos substituir o **k** por cinco.

```
> subs(k=5,1/3*k^3-1/2*k^2+1/6*k);
```

$$30$$

```
> 0+1+4+9+16;
```

$$30$$

O *Maple* consegue ainda calcular a soma de séries que sejam convergentes.

```
> sum('1/k!','k'=0..infinity);
```

$$e$$

Nos casos em que a série “converge” para o infinito o resultado é ∞ ou - ∞.

```
> sum('3*k','k'=0..infinity);
```

$$\infty$$

```
> sum('-3*k','k'=0..infinity);
```

$$-\infty$$

No caso seguinte o resultado obtido é incorrecto uma vez que a série considerada é da forma –1, 1, -1, 1, -1, ...

```
> sum('(-1)^k','k'=1..infinity);
```

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

Como podemos verificar de seguida, quando o limite superior é par o somatório é zero e quando este é ímpar o seu valor é –1. Como desconhecemos a paridade de ∞, seria mais correcto se o utilizador fosse informado de que não é possível conhecer o valor que a referida série assume.

```
> sum('(-1)^k','k'=1..337);
                                    -1

> sum('(-1)^k','k'=1..4000);
                                    0
```

Mathematica:

O somatório de um número finito ou infinito de termos é obtido através da função **Sum**. Esta função é invocada com dois argumentos. O primeiro é a expressão a avaliar enquanto que o segundo é uma lista que nos indica a variável e os valores que a mesma deve assumir (**imin**, **imax** e **di**).

Alternativamente podemos utilizar a notação usual se recorrermos às *palettes*. Aí usando o atalho  podemos introduzir o *input* na forma mais comum.

```
Sum [i, 8]
```

66

```
i=3
```

135

```
Sum [i, 10, 2]
```

28

No último *input* mandámos adicionar os valores compreendidos entre 4 e 10, mas avançando de dois em dois valores. Neste caso o *output* é a soma de 4, 6, 8 e 10.

Tratando-se de uma série convergente, é ainda possível calcular o valor da mesma.

```
i=3
```

135

```
i=0 i!
```

ã

Caso a série não seja convergente, o *Mathematica* vê-se impossibilitado de nos dar um valor.

```
i=0
Sum::div : Sum does not converge.
i=0
```

Obtemos a mesma informação no seguinte exemplo em que a série não "converge" para o infinito.

```
Sum[1/i, {i, 1, ∞}]
Sum::div : Sum does not converge.
  ∞
  ∑ 1/i
 i=1
```

Portanto existe aqui uma diferença em relação aos programas anteriores. Enquanto estes forneciam um *output* que não era correcto o *Mathematica* tem o mérito de transmitir ao utilizador que a série que está a considerar não converge.

Aqui também se mandarmos avaliar esta série com um limite superior par ou ímpar obtemos os resultados correctos, tal como acontecia nos programas anteriores.

```
Sum[(-1)^i, {i, 1, 125}]
-1
```

```
Sum[(-1)^i, {i, 1, 432}]
0
```

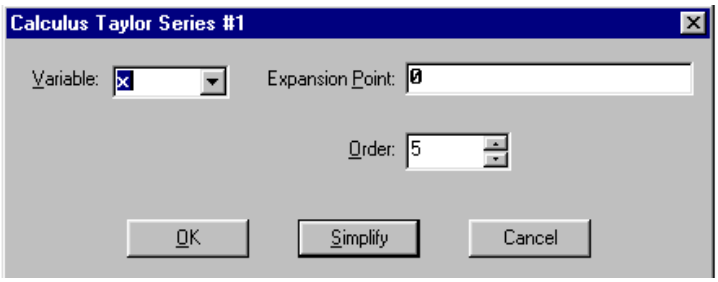
Para obter a soma dos valores constantes de uma lista temos de utilizar duas funções predefinidas, neste caso concreto **Apply** e **Plus** sendo que a aplicação conjunta destas duas funções o que fará será dizer ao *Mathematica* que deve proceder como se a lista fosse uma soma e portanto, tem de somar as respectivas entradas. Mais à frente e mais especificamente quando abordarmos a programação funcional tornar-se-á mais explícita o modo como actua o funcional **Apply**.

```
Apply[Plus, {2, 3, 7, -3, 4}]
18
```

3.12.5. – EXPANSÃO EM SÉRIE

Derive:

Introduzindo a expressão da função da qual queremos obter uma expansão em série podemos utilizar o menu *Calculus* e o submenu *Taylor Series*. Surge na zona de trabalho uma janela em que indicamos qual a variável, o ponto em torno do qual será feita a expansão e a ordem.




```
SIN(x)
TAYLOR(SIN(x), x, 0, 5)
```

$$\frac{x^5}{120} - \frac{x^3}{6} + x$$

A expansão em Série de Taylor pode ser obtida ainda pela utilização da função predefinida **TAYLOR(u,x,a,n)**, constatação facilmente verificável pela forma como o *Derive* transcreve para a zona de trabalho a série pedida anteriormente. Aqui na nossa expressão o **n** representa a ordem da aproximação que não é mais que o número de derivadas parciais usadas para truncar a Série de Taylor que aproxima a expansão. Uma vez que algumas destas derivadas podem ser zero, o número de termos e o grau do polinómio resultante poderá ser menor que a ordem da aproximação.

```
EXP(x)
TAYLOR(EXP(x), x, 0, 5)
```

$$\frac{x^5}{120} + \frac{x^4}{24} + \frac{x^3}{6} + \frac{x^2}{2} + x + 1$$

```
TAYLOR(EXP(x), x, 2, 5)
```

$$\frac{\hat{e}^2 \cdot (x^5 - 5 \cdot x^4 + 20 \cdot x^3 - 20 \cdot x^2 + 40 \cdot x + 8)}{120}$$

Quando não é possível obter um desenvolvimento em Série de Taylor o sistema vê-se impossibilitado de dar uma resposta que nos satisfaça.

TAYLOR(1/x + y + x^3, x, 0, 5)

?

Maple:

A função predefinida **series** determina uma expansão em série em torno de um ponto para uma expressão numa variável. Esta função pode ser invocada com dois ou três argumentos sendo o primeiro a expressão da qual se pretende obter o desenvolvimento em série. O segundo argumento é uma equação do tipo **x=a** ou **x**, caso em que o *Maple* toma **x=0**.

Quando utilizado, o terceiro argumento representa a ordem até à qual pretendemos que o desenvolvimento seja efectuado.

```
> series(f(x),x=0);
```

$$f(0)+D(f)(0)\,x+\frac{1}{2}(D^{(2)})(f)(0)\,x^2+\frac{1}{6}(D^{(3)})(f)(0)\,x^3+\frac{1}{24}(D^{(4)})(f)(0)\,x^4+\frac{1}{120}(D^{(5)})(f)(0)\,x^5+O(x^6)$$

```
> series(f(x),x=a,3);
```

$$f(a)+D(f)(a)\,(x-a)+\frac{1}{2}(D^{(2)})(f)(a)\,(x-a)^2+O((x-a)^3)$$

```
> series (sin(x),x=0);
```

$$x-\frac{1}{6}x^3+\frac{1}{120}x^5+O(x^6)$$

A função **taylor** é um caso particular da função **series** e como tal aplica-se da mesma forma.

```
> taylor(exp(x),x,4);
```

$$1+x+\frac{1}{2}x^2+\frac{1}{6}x^3+O(x^4)$$

```
> taylor(1/x,x=1,3);
```

$$1-(x-1)+(x-1)^2+O((x-1)^3)$$

```
> taylor(1/x+y+x^3,x,3);
Error, does not have a taylor expansion, try series()
```

```
> series(1/x+y+x^3,x,3);
```

$$x^{-1} + y + O(x^3)$$

Mathematica:

Mediante a utilização da função predefinida **series** obtemos o desenvolvimento em série de uma função. **Series[f,{x,x$_0$,n}]** gera uma expansão em série em torno de **x$_0$** e até o termo de ordem **n**.

```
Series[f[x], {x, a, 5}]
```

$$f[a] + f'[a](x-a) + \frac{1}{2}f''[a](x-a)^2 + \frac{1}{6}f^{(3)}[a](x-a)^3 + \frac{1}{24}f^{(4)}[a](x-a)^4 + \frac{1}{120}f^{(5)}[a](x-a)^5 + O[x-a]^6$$

```
Series[f[x], {x, a, 3}]
```

$$f[a] + f'[a](x-a) + \frac{1}{2}f''[a](x-a)^2 + \frac{1}{6}f^{(3)}[a](x-a)^3 + O[x-a]^4$$

```
Series[1/x+y+x^3, {x, 0, 5}]
```

$$\frac{1}{x} + y + x^3 + 0[x]^6$$

```
Series[Cos[x], {x, 0, 4}]
```

$$1 - \frac{x^2}{2} + \frac{x^4}{24} + 0[x]^5$$

A determinação dos coeficientes de uma série é possível se conhecermos a ordem do termo e utilizarmos a função predefinida **SeriesCoefficient**.

```
SeriesCoefficient[%, 4]
```

$$\frac{1}{24}$$

Podemos converter uma série num polinómio utilizando **Normal[Series]**.

```
Series[Exp[x], {x, 0, 4}]
```

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + 0[x]^5$$

```
Normal[%]
```

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24}$$

3.12.6. - PRODUTÓRIOS

Derive:

O cálculo de um produtório também pode ser feito no *Derive* através do atalho Π ou então utilizando a função **PRODUCT**. Se quisermos calcular o produto de um conjunto de números podemos optar por aplicar a função referida com um único argumento – a lista constituída por esse conjunto de números. Se introduzirmos a expressão **PRODUCT([2,3,5,7])** e mandarmos avaliá-la teremos como *output* o produto dos quatro números que constituem a lista.

```
Π([2, 3, 5, 7])
```

$$210$$

De forma em tudo semelhante à aplicada para os somatórios podemos calcular o produto de uma expressão quando uma determinada variável varia entre dois dados valores.

$$\prod_{k=1}^{5} 2 \cdot k$$

$$3840$$

$$\prod_{k=1}^{n} k$$

$$n!$$

Há também a possibilidade de impor que a variável apenas assuma os valores constantes de um determinada lista.

```
    2
Π(k , k, [1, 2, 4, 7])
```

$$3136$$

Maple:

O produto de um conjunto de números pode ser fornecido pelo uso da função **mul**.

```
> mul(i,i=1..4);
```

$$24$$

```
> mul(i,i=[2,4,8,16]);
```

$$1024$$

Ao pretendermos obter a fórmula para um produto definido ou indefinido temos de recorrer à função **product**.

```
> product(k,k=1..5);
```

$$120$$

```
> product(k,k=1..n);
```

$$\Gamma(n+1)$$

A função **Γ** (função gama) representa uma extensão da função factorial aos complexos, sendo definida como o integral da função $\Gamma(z)=e^{-t}\,t^{z-1}$ quando **t** varia entre **0** e **infinito**. No caso concreto dos inteiros não negativos tem-se que **Γ(n+1)=n!**.

Mathematica:

O produtório, tal como acontece com os somatórios, pode ser introduzido na forma usual havendo para isso que recorrer às *palettes* mais concretamente ao atalho  .

Também se pode calcular um produtório através da função **Product** cuja aplicação se faz de forma análoga à função **Sum**.

Product [redacted}**, n** ▌

n!

[redacted]

i=1

n! 2

Se pretendermos podemos também calcular o produto de uma expressão quando a variável assume valores que não são consecutivos. Para o conseguir torna-se necessário invocar a função **Product** com o segundo argumento constituído por uma lista de comprimento **4** onde o último elemento representa o comprimento dos passos a dar. Vejamos um exemplo em que mandámos calcular o produtório quando a variável assume valores separados por **2** unidades neste caso concreto **1,3,5,7** e **9**.

Product [redacted}**, 10, 2** ▌

945

3.12.7. - LIMITES

Derive:

O cálculo de limites no *Derive* pode fazer-se recorrendo ao menu *calculus* e ao submenu *Limit*. Aí especificamos as características do limite que pretendemos calcular. Outra forma será recorrer ao atalho  da barra de ferramentas.

$$\lim_{x \to 2} \frac{3 \cdot x^2 + 2 \cdot x - 16}{x^2 - x - 2}$$

$$\frac{14}{3}$$

O cálculo dos limites laterais à esquerda e à direita fazem-se da mesma forma havendo apenas de alterar o campo que nos indica o lado pelo qual pretendemos que sejam feitas as aproximações.

$$\lim_{x \to 0+} \frac{9}{x}$$

$$\infty$$

$$\lim_{x \to 0-} \frac{9}{x}$$

$$-\infty$$

$$\lim_{x \to 0} \frac{9}{x}$$

$$\pm\infty$$

Caso queiramos calcular um limite utilizando a função predefinida, tal é possível através da função LIM. A título de exemplo para calcular o último limite quando **x** tende para zero por valores à direita introduzir-se-ia a expressão **LIM(9/x, x, 0, 1)**. Aqui o 1 significa que nos estamos a aproximar por valores à direita do zero. Se pretendêssemos o limite à esquerda em vez de **1** teríamos de colocar **–1**.

Maple:

Podemos determinar o valor do limite de uma função recorrendo à função **limit**.

```
> limit((x^2-3*x+2)/(3*x^3+5*x^2-6*x-2),x=1);
```

$$\frac{-1}{13}$$

```
> limit((x-1)/sqrt(x^2-1),x=1);
```

$$0$$

Pode ainda calcular-se o limite de uma dada função à medida que se aproxima de um dado valor por valores à direita ou à esquerda. Tal é feito utilizando um terceiro argumento na função **limit**.

```
> limit(cos(x)/(x^2-4),x=-2,left);
```

$$-\infty$$

```
> limit(cos(x)/(x^2-4),x=-2,right);
```

$$\infty$$

Não existe portanto limite.

```
> limit(cos(x)/(x^2-4),x=-2);
```

undefined

Mathematica:

O cálculo de limites de funções faz-se recorrendo à função **Limit**.

```
Limit[(x+3)/(x^3+4), x->¥]
```

```
0
```

```
Limit[Sqrt[x^2-a^2], x->¥]
```

```
0
```

No que diz respeito à determinação do limite à direita e à esquerda este faz-se colocando um terceiro argumento (**-1** ou **+1**) que no caso de ser **+1** indica que nos estamos a aproximar do ponto por valores à esquerda e no caso de ser **–1** indica que vamos nos aproximando do ponto por valores à direita.

```
Limit[1/x^2, x->0, Direction->-1]
```

```
¥
```

```
Limit[1/x^2, x->0, Direction->+1]
```

```
-¥
```

Querem estes resultados dizer que o limite da função quando **x** tende para **0** por valores à direita é **+ ∞** e que o limite da função quando **x** tende para **0** por valores à esquerda é **- ∞**. Não existe portanto limite nestas condições. Quando muito poderemos dizer que a função tende para infinito sem sinal. Reparemos ainda como é fácil confundir infinito sem sinal (∞) com + ∞.

```
Limit[1/x^2, x->0]
```

```
¥
```

3.13. – CONCLUSÃO

Qualquer que seja o programa com que estejamos a trabalhar é possível utilizar variáveis para armazenar valores, sendo também possível guardar em variáveis expressões que não têm de ser numéricas. Quando não dispomos de uma função predefinida que realize aquilo que pretendemos, podemos recorrer à definição de funções e, utilizando a sintaxe correcta, definir a função desejada. Neste campo o *Mathematica* possui mais argumentos que os restantes programas pois a atribuição paramétrica diferida é um poderoso meio de definir funções como será visto mais adiante.

Em relação às estruturas disponibilizadas, todos os programas dispõem do tipo lista. Para além deste o *Derive* e o *Maple* ainda possuem conjuntos.

Depois de definida uma lista é possível aceder a qualquer um dos seus elementos utilizando as funções predefinidas em cada programa. O mesmo se passa em relação aos conjuntos nos programas em que estes estão definidos.

A inserção de elementos é mais eficiente no *Derive* e no *Mathematica* já que aí o elemento pode ser inserido em qualquer posição da lista enquanto que no *Maple* essa posição tem de ser a última. No *Maple* não nos foi possível encontrar uma forma de juntar duas listas apesar de o mesmo se efectuar facilmente quando estamos a trabalhar com conjuntos. De referir que apesar do *Mathematica* não disponibilizar directamente conjuntos é possível fazer a reunião de duas listas que eventualmente representem conjuntos. No *Mathematica*, para além dos meios que o *Derive* e o *Maple* possuem que incidem fundamentalmente na capacidade de seleccionar elementos que verificam um determinado predicado, podem escolher-se elementos através da posição que os elementos ocupam na lista.

Em todos os sistemas é possível verificar se um dado elemento está presente numa lista (ou conjunto tratando-se do *Derive* ou do *Maple*).

Quer estejamos a trabalhar com equações algébricas concretas ou arbitrárias, os programas que aqui analisamos simplificam o nosso trabalho permitindo obter as soluções exactas de maneira muito similar. Em relação aos sistemas de equações poderemos afirmar o mesmo quando estes são possíveis. Quanto aos sistemas impossíveis o *Maple* simplesmente omite o *output* o que, quanto a nós, dá a ideia que este foi incapaz de concluir que não havia solução.

Quando confrontados com equações não algébricas, nomeadamente envolvendo funções trigonométricas, nenhum dos programas consegue dar a totalidade das soluções.

Existem também alguns casos nomeadamente quando as equações envolvem funções trigonométricas e a função exponencial em que só é possível obter soluções para as mesmas através de métodos numéricos (aproximações).

Utilizando o *Derive*, o *Maple* ou o *Mathematica* conseguimos obter valores para somatórios e séries. Merece aqui especial referência o facto de apenas o *Mathematica* conseguir detectar que a série $\sum_{i=1}^{\infty}(-1)^i$ é divergente. Perante a mesma série obtivemos uma resposta incorrecta por parte do *Maple* enquanto que o *Derive* deu-nos $\frac{\text{sen}\,\infty}{2}-\frac{1}{2}$. Passando para somatórios em todos os programas obtemos valores correctos.

Refira-se ainda que nos casos em que uma série tende para ± ∞ no *Derive* e no *Maple* obtivemos ∞ ou - ∞ o que não acontece no *Mathematica*. Este apenas informa que a série é divergente.

Na expansão em série todos os programas, mediante especificação da ordem e do ponto em torno do qual se pretende o desenvolvimento, conseguem obter o desenvolvimento em Série de Taylor. Para os casos mais gerais, em que não existe desenvolvimento em Série de Taylor, temos de recorrer ao *Maple* ou ao *Mathematica*.

Nos produtórios os três programas equivalem-se sendo possível encontrar valores para os produtórios considerados mesmo quando o limite superior é um valor arbitrário.

O modo de calcular os limites é muito parecido de um programa para o outro. No *Derive* e no *Mathematica* a forma de indicar cada um dos limites laterais é semelhante mas oposta. Por exemplo ao indicarmos -1 no *Derive* estamos a calcular o limite à esquerda enquanto que a mesma indicação no *Mathematica* impõe o cálculo do limite à direita. Nestes casos o *Maple* é mais explícito pois basta escrevermos *left* ou *right*.

Em resumo:

- Em qualquer dos programas é possível utilizar variáveis para guardar expressões bem como definir funções. No *Mathematica* podem fazer-se atribuições e definir funções através da atribuição diferida.
- O *Derive* e o *Maple* disponibilizam mais estruturas que o *Mathematica* nomeadamente conjuntos.
- A inserção de elementos é mais fácil no *Derive* e no *Mathematica*.
- No *Maple* não nos foi possível encontrar um meio directo de juntar duas listas.
- O *Mathematica* consegue determinar em mais casos que os outros programas em análise quando uma série é divergente. Quando uma série tende para ± ∞ o *Mathematica* dá uma mensagem de erro enquanto que o *Derive* e o *Maple* dão -∞ ou ∞ conforme o caso.
- A forma de indicar os limites laterais é mais intuitiva no *Maple*.

Neste capítulo temos que focar que o *Mathematica* possui mais meios para se efectuarem atribuições e definir funções sendo possível detectar em mais casos que nos restantes programas, quando estamos na presença de uma série divergente.

4. - VECTORES MATRIZES E POLINÓMIOS

O tipo lista está presente em todos os programas sendo utilizado na grande parte das situações que ultrapassam a mera manipulação de expressões.

Assim as listas utilizam-se para representar aquelas situações mais complexas como sejam vectores e matrizes, podendo utilizar-se listas dentro de outras listas, o que dá a este tipo uma grande versatilidade.

Atrás já vimos todo um conjunto de funções que estão aptas a receber como argumentos as listas, facilitando a representação das mais diversas situações.

Este capítulo pode dividir-se em duas partes. Na primeira fazemos um pequeno estudo de como podemos utilizar o tipo lista para definir as operações elementares do cálculo vectorial e matricial e na segunda abordamos os polinómios.

Começamos por definir as operações usuais de soma de dois vectores (representados por listas), produto de um escalar por um vector e produto interno de dois vectores até atingirmos o caso mais geral do cálculo matricial no qual recorremos à representação de listas de listas para representar convenientemente o tipo matriz. Avançando vamos tornar mais explícito a forma de operar com matrizes e aplicar o cálculo matricial à resolução de um sistema de equações possível e determinado. Teremos também oportunidade de calcular determinantes bem como valores e vectores próprios.

Nos polinómios veremos algumas funções que sobre eles estão definidas, umas mais elementares que outras procurando mostrar as vantagens da utilização destes três programas. Entre outras falaremos aqui de máximo divisor comum e mínimo múltiplo comum de dois polinómios, resto e quociente da divisão de dois polinómios e interpolação polinomial sem nos esquecermos da factorização e expansão de polinómios.

No decorrer deste capítulo daremos continuidade ao cálculo simbólico já que não serão apenas os números que serão objecto do nosso estudo, consideraremos também situações arbitrárias, situações em que será possível verificar o poder de qualquer um dos *softwares* em análise.

4.1. - CÁLCULO VECTORIAL

Derive:

Os vectores em *Derive* representam-se através de listas. Assim utilizando as listas, conseguimos efectuar a multiplicação de um vector por um escalar, calcular a soma e subtracção de dois vectores, bem como calcular o produto interno de dois vectores.

```
[1, 2, 4] + [1, -1, -5]
                                                    [2, 1, -1]
```

É claro que não é possível calcular a soma de dois vectores que não tenham a mesma dimensão. Ao mandarmos avaliar uma situação em os vectores não têm o mesmo comprimento a expressão fica por avaliar.

```
[1, 2, 4] + [1, -1, -5, 21]
                                          [1, 2, 4] + [1, -1, -5, 21]
```

A multiplicação por um escalar faz-se da maneira usual, ou seja, multiplicando o escalar por cada um dos elementos do vector.

```
5 · [1, 2, 4]
                                                    [5, 10, 20]
```

É ainda possível calcular o produto interno de dois vectores.

```
[1, 5, -1] · [1, 2, 3]
                                                          8

[a, b, c] · [d, e, f]
                                                  a·d + b·e + c·f
```

Maple:

Do mesmo modo, também em *Maple* os vectores são representados por listas.

A soma e subtracção de dois vectores consegue fazer-se facilmente. Para efectuar uma soma ou subtracção procede-se de modo semelhante ao que utilizaríamos para trabalhar com números inteiros.

```
> [3,7,-1]+[1,2,0];
                                                    [4, 9, -1]
```

A soma ou subtracção de dois vectores que não tenham a mesma dimensão não é permitida.

```
> [1,2,3]+[1,3,5,7,8];
Error, adding lists of different length
```

A multiplicação de um vector por um escalar é feita da forma comum. Assim, o resultado da multiplicação de um escalar por um vector é um vector em que cada um dos seus elementos vem multiplicado pelo referido escalar.

```
> 4*[3,7,-1];
```

$$[12, 28, -4]$$

O produto interno de dois vectores pode ser calculado através da função **multiply**.

```
> multiply([7,2,0,1],[1,2,3,4]);
```

$$15$$

```
> multiply([a,b,c],[d,e,f]);
```

$$a\,d+b\,e+c\,f$$

Mathematica:

No *Mathematica*, tal como nos dois casos anteriores, um vector é representado por uma lista de elementos.

É possível multiplicar um vector por um escalar, somar ou subtrair dois vectores e fazer o produto interno de dois vectores.

A multiplicação de um escalar faz-se como é usual multiplicando o escalar por cada um dos elementos do vector.

```
2* , 5, - 1
, 10, - 2
```

O valor da soma ou subtracção de dois vectores é igual a um vector cujas entradas são as somas ou subtracções dos elementos da mesma posição em cada um dos vectores.

```
, 5, - 1 + , 4, 6
, 9, 5
```

Caso as listas que representam os vectores não tenham a mesma dimensão não é possível efectuar a sua soma.

```
, 3 + 2, 8, 7
Thread::tdlen :
 Objects of unequal length in , 3 + - 2, 8, 7  cannot be combined.
, 3 + 2, 8, 7
```

O produto interno define-se como a soma do produto do primeiro elemento do primeiro vector pelo primeiro elemento do segundo vector com o produto do segundo elemento do primeiro vector pelo segundo elemento do segundo vector, ...

```
[redacted], 3, 4 . [redacted], 3, 5
```

```
34
```

```
[redacted], c . [redacted], f
```

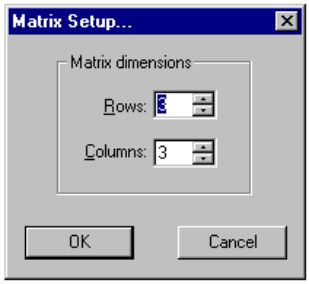
```
a d + b e + c f
```

4.2. - CÁLCULO MATRICIAL

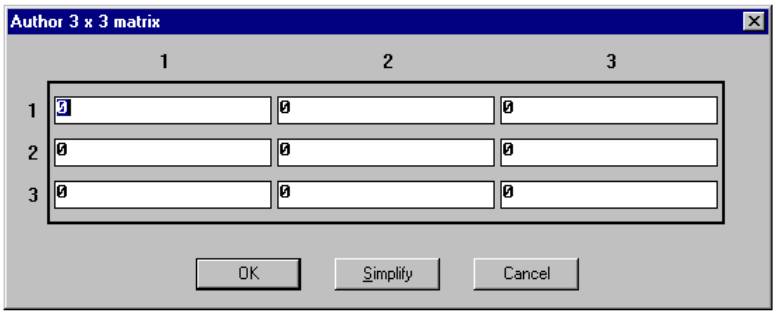
4.2.1. - CONSTRUÇÃO DE UMA MATRIZ

Derive:

Para editar uma matriz existem várias possibilidades. Uma delas é recorrer ao atalho  existente na barra de ferramentas. Imediatamente surge no ecrã uma janela que nos pede as dimensões da matriz.



Posteriormente é só introduzir os elementos que pretendemos.



Outra forma será recorrer ao menu *Author* e ao submenu *Matrix* ou ao submenu *Expression* havendo neste último caso que introduzir a matriz na forma de uma lista constituída por listas (lista de listas).

Neste caso a nossa matriz foi **[[1,2,3],[4,5,6],[7,8,9]]**.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Maple:

Uma matriz é representada como um *array* com dois argumentos.

Ao escrevermos **array(1..3,1..4)** estamos a criar uma matriz arbitrária de **3** linhas por **4** colunas.

```
> array(1..3,1..4);
```

$$\begin{bmatrix} ?_{1,1} & ?_{1,2} & ?_{1,3} & ?_{1,4} \\ ?_{2,1} & ?_{2,2} & ?_{2,3} & ?_{2,4} \\ ?_{3,1} & ?_{3,2} & ?_{3,3} & ?_{3,4} \end{bmatrix}$$

Recorrendo ao pacote *linalg* podemos utilizar outras funções para definir matrizes.

A função **matrix** permite que vejamos a matriz na forma que é usual. No entanto, não é esta a forma mais simples de obter uma representação de uma matriz. A grande vantagem é a visualização imediata da matriz enquanto tal. Vejamos alguns aspectos desta função, começando por definir uma matriz de dimensão 3 por 2.

```
> with(linalg):
> matrix(3,2,[3,2,4,-1,3,0]);
```

$$\begin{bmatrix} 3 & 2 \\ 4 & -1 \\ 3 & 0 \end{bmatrix}$$

A lista anterior representa os elementos que fazem parte da matriz, enquanto que o primeiro e o segundo argumento referem-se à dimensão da matriz. Existem outras formas de introduzir matrizes, utilizando ainda a função anterior.

```
> matrix([[3,2],[4,-1],[3,0]]);
```

$$\begin{bmatrix} 3 & 2 \\ 4 & -1 \\ 3 & 0 \end{bmatrix}$$

O que aqui pedimos é que apresente sob a forma de matriz a lista de listas introduzida.

Se pretendermos introduzir uma matriz cujos elementos são gerados através de uma função temos de a criar e só depois definir a dimensão da matriz.

```
> f:=i->i^2;
```

$$f := i \to i^2$$

```
> A:=matrix(2,2,f);
```

$$A := \begin{bmatrix} 1 & 1 \\ 4 & 4 \end{bmatrix}$$

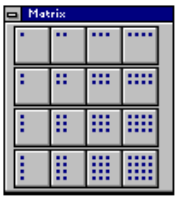
```
> g:=(i,j)->i-j;
```

$$g := (i,j) \rightarrow i - j$$

```
> matrix(3,3,g);
```

$$\begin{bmatrix} 0 & -1 & -2 \\ 1 & 0 & -1 \\ 2 & 1 & 0 \end{bmatrix}$$

Uma outra forma de definir uma matriz é tirar partido das *palettes*, mais precisamente de



em que especificamos a dimensão da matriz com que pretendemos trabalhar. Neste caso, a dimensão introduzida foi 3 por 4. Na zona de trabalho surge uma expressão utilizando a função **matrix** em que as entradas não estão definidas.

```
> Matrix([[%?, %?, %?, %?], [%?, %?, %?, %?], [%?, %?,
  %?, %?]]);
```

Utilizando a tecla → (TAB) facilmente conseguimos definir a matriz pretendida. Mandando avaliar tudo se passa como atrás foi referido.

```
> Matrix([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]);
```

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

Mathematica:

Uma matriz é representada como uma lista de listas.

[illegible], 3 , [illegible], 6 representa uma matriz que tem duas linhas e três colunas.

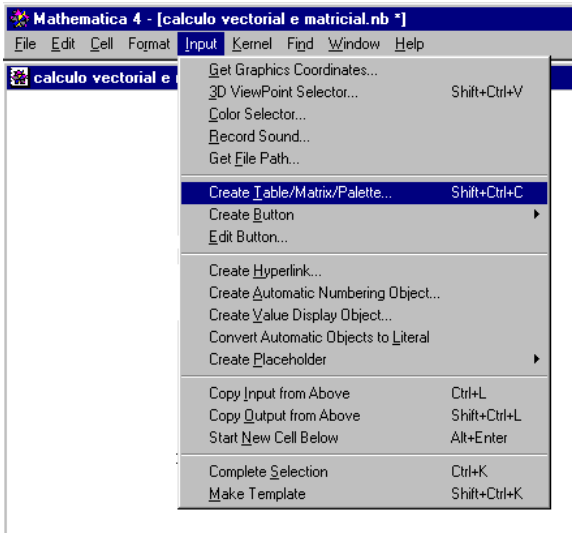
Para uma fácil visualização daquilo a que nos referimos podemos representar a matriz na forma matricial, recorrendo para isso à função **MatrixForm**.

```
MatrixForm [illegible]3 , [illegible], 6 ]
1 2 3
4 5 6
```

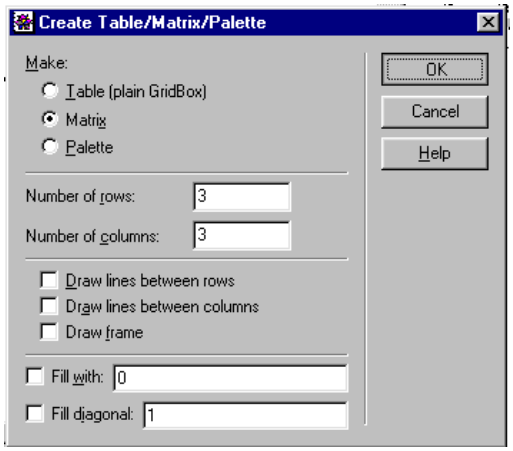
Se estivermos a trabalhar com matrizes de dimensão 2 por 2 podemos ainda usar as *palettes* para efectuar todo o trabalho de edição de matrizes, ganhando-se assim clareza uma vez que podemos introduzir as matrizes já na forma com que é comum trabalharmos.

Para tal recorremos às *palettes*, mais concretamente ao atalho  .

Estando a trabalhar com matrizes que nem sempre são da dimensão atrás referida, continuamos a dispor desta oportunidade, se para tal recorrermos ao menu *Input* e ao submenu *Create Table/Matrix/Palette* especificando posteriormente o número de linhas e de colunas da matriz que pretendemos.



No que se segue resolvemos definir uma matriz com três linhas e três colunas.




```
{{2, 4, 7},
 {-5, 1, 8},
 {4, 5, 0}}
```

4.2.2. - OPERAÇÕES COM MATRIZES

Derive:

Facilmente podemos determinar a soma, a subtracção e a multiplicação de duas matrizes desde que estas operações sejam permitidas pelas dimensões das matrizes em questão.

$$\begin{bmatrix} 1 & 7 \\ 4 & 5 \end{bmatrix} + \begin{bmatrix} -1 & 4 \\ 2 & 2 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 11 \\ 6 & 7 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 7 \\ 4 & 5 \end{bmatrix} - \begin{bmatrix} -1 & 4 \\ 2 & 2 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 3 \\ 2 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 7 \\ 4 & 5 \end{bmatrix} \cdot \begin{bmatrix} -1 & 4 \\ 2 & 2 \end{bmatrix}$$

$$\begin{bmatrix} 13 & 18 \\ 6 & 26 \end{bmatrix}$$

Recorde-se que para somar e subtrair matrizes é necessário que estas tenham o mesmo número de linhas e de colunas, e que para multiplicar duas matrizes basta que o número de colunas da primeira matriz seja igual ao número de linhas da segunda matriz.

$$\begin{bmatrix} 5 & 4 & 1 \\ 7 & 2 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 \\ 2 & 1 \\ -1 & 9 \end{bmatrix}$$

$$\begin{bmatrix} 12 & 33 \\ 11 & 30 \end{bmatrix}$$

Maple:

Definidas as matrizes, podemos efectuar as operações elementares com as mesmas.

```
> matriz1:=[[1,4,3],[2,3,1],[3,2,-1]]:

> matriz2:=[[-1,4,1],[3,3,2],[4,2,1]]:
```

```
> matrix(matriz1);
```

$$\begin{bmatrix} 1 & 4 & 3 \\ 2 & 3 & 1 \\ 3 & 2 & -1 \end{bmatrix}$$

```
> matrix(matriz2);
```

$$\begin{bmatrix} -1 & 4 & 1 \\ 3 & 3 & 2 \\ 4 & 2 & 1 \end{bmatrix}$$

```
> matriz1+matriz2;
```

$$[[0, 8, 4], [5, 6, 3], [7, 4, 0]]$$

```
> matriz1-matriz2;
```

$$[[2, 0, 2], [-1, 0, -1], [-1, 0, -2]]$$

Se quisermos podemos obter o resultado na forma de matriz recorrendo à função **matrix**.

```
> matrix(%);
```

$$\begin{bmatrix} 2 & 0 & 2 \\ -1 & 0 & -1 \\ -1 & 0 & -2 \end{bmatrix}$$

Poderíamos também ter optado por fazer todos os cálculos utilizando a função **matrix** o que nos traria algumas vantagens ao nível da visualização. Contudo, para obtermos a avaliação das expressões pretendidas teríamos de recorrer à função **evalm**.

```
> matrix(matriz1)-matrix(matriz2);
```

$$\begin{bmatrix} 1 & 4 & 3 \\ 2 & 3 & 1 \\ 3 & 2 & -1 \end{bmatrix} - \begin{bmatrix} -1 & 4 & 1 \\ 3 & 3 & 2 \\ 4 & 2 & 1 \end{bmatrix}$$

```
> evalm(%);
```

$$\begin{bmatrix} 2 & 0 & 2 \\ -1 & 0 & -1 \\ -1 & 0 & -2 \end{bmatrix}$$

Para a multiplicação de matrizes temos de usar uma função particular, essa função é **multiply**.

```
> multiply(matriz1,matriz2);
```

$$\begin{bmatrix} 23 & 22 & 12 \\ 11 & 19 & 9 \\ -1 & 16 & 6 \end{bmatrix}$$

Mathematica:

Podemos efectuar as operações usuais sobre matrizes tais como a soma, a subtracção e a multiplicação de matrizes bem como a multiplicação de um escalar por uma matriz.

As matrizes em *Mathematica* são representadas por listas de listas. Assim, a soma e subtracção de duas matrizes pode ser feita como apresentamos de seguida.

```
{{1, 5, 4}, {7, 8, 3}} + {{4, -1, 1}, {4, 3, 1}}
{{5, 4, 5}, {11, 11, 4}}
```

Outra forma que é equivalente à anterior é utilizar as *palettes* e introduzir a matriz já na forma usual. Reparemos que o *output* é exactamente igual ao anterior.

```
{{1, 5, 4}, {7, 8, 3}} + {{4, -1, 1}, {4, 3, 1}}
{{5, 4, 5}, {11, 11, 4}}
```

Podemos mandar o *Mathematica* escrever o *output* em forma de matriz.

```
MatrixForm[%]
5   8   4
11  11  4
```

```
{{1, 5, 4}, {7, 8, 3}} - {{4, -1, 1}, {4, 3, 1}}
{{-3, 6, 3}, {3, 5, 2}}
```

```
3* {{1, 5, 4}, {7, 8, 3}}
{{3, 15, 12}, {21, 24, 9}}
```

Mais uma vez aqui lembramos o cuidado a ter quando pretendemos efectuar a multiplicação de duas matrizes. É necessário garantir que o número de colunas da primeira matriz seja igual ao número de linhas da segunda matriz.

```
{{1, 2, 4}, {5, 0, 1}} . {{1, 2, 3}, {0, 1, -1}}
{{1, 4, -1}, {5, 5, -5}, {2, 3, -1}}
```

```
MatrixForm[%]
1  4  -1
5  5  -5
2  3  -1
```

4.2.3. - POTÊNCIAS DE MATRIZES

Derive:

O cálculo da potência de uma matriz é efectuado usando a notação normalmente usada quando pretendemos calcular a potência de qualquer número. No exemplo que se segue pretendendo calcular o quadrado da matriz em questão bastou fazer **[[1,2],[3,4]]^2**.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^2$$

$$\begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix}$$

A mesma notação utilizamos ao pretender calcular a inversa de uma matriz.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^{-1}$$

$$\begin{bmatrix} -2 & 1 \\ \frac{3}{2} & -\frac{1}{2} \end{bmatrix}$$

Maple:

O cálculo de uma potência de uma matriz faz-se da maneira usual.

```
> [[1,-3,3],[0,1,-2],[7,1,4]]^2;
```

$$[[1, -3, 3], [0, 1, -2], [7, 1, 4]]^2$$

Também aqui é necessário recorrer à função **evalm** para obtermos o resultado pretendido.

```
> evalm(%);
```

$$\begin{bmatrix} 22 & -3 & 21 \\ -14 & -1 & -10 \\ 35 & -16 & 35 \end{bmatrix}$$

A inversa de uma matriz quadrada pode ser obtida através da função **inverse**.
Em primeiro lugar temos de carregar o pacote **linalg**.

```
> with(linalg):
Warning, new definition for norm
Warning, new definition for trace
```

```
> [[1,-3,3],[0,1,-2],[7,1,4]];
```

```
> inverse(%);
```

$$\begin{bmatrix} \frac{2}{9} & \frac{5}{9} & \frac{1}{9} \\ \frac{-14}{27} & \frac{-17}{27} & \frac{2}{27} \\ \frac{-7}{27} & \frac{-22}{27} & \frac{1}{27} \end{bmatrix}$$

```
> multiply(%,%%);
```

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

Mathematica:

Para calcular a potência de uma matriz torna-se necessário recorrer à função predefinida **MatrixPower**. Para não estarmos posteriormente a pedir que o *output* seja apresentado em forma de matriz, pedimos já directamente ao *Mathematica* que o faça colocando **//MatrixForm** no fim do *input*.

```
MatrixPower[{{1, 0, 2}, {0, -3, 2}, {2, 5, 0}}, 2] //MatrixForm
```

```
5    10    2
0   -11   -8
3    10   -11
```

Repare-se que se tentarmos utilizar o "^" para cálculo da potência de uma matriz não obtemos o pretendido. Ao invés disso o *Mathematica* aplica a função potenciação a cada um dos elementos da matriz. Assim ao fazermos **A^2** onde **A** é uma matriz, estamos a pedir ao *Mathematica* que nos forneça uma matriz em que cada entrada é o quadrado da entrada inicial.

```
{{5, 3}, {3, 4}} //MatrixForm
```

```
25   4
9    16
```

Utilizando a função **Inverse** podemos obter a matriz inversa de uma dada matriz quadrada (número de linhas igual ao número de colunas).

```
Inverse[{{1, 2, -1}, {0, 1, 2}, {3, 0, 1}}] // MatrixForm
```

$$\begin{pmatrix} \frac{1}{20} & \frac{1}{4} & -\frac{1}{10} \\ \frac{7}{20} & -\frac{1}{4} & \frac{3}{10} \\ -\frac{3}{20} & -\frac{1}{4} & \frac{1}{2} \end{pmatrix}$$

Podemos obter a matriz identidade multiplicando a matriz original pela sua inversa.

```
{{1/20, 1/4, -1/10}, {7/20, -1/4, 3/10}, {-3/20, -1/4, 1/2}} . {{1, 2, -1}, {0, 1, 2}, {3, 0, 1}} // MatrixForm
```

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

Outra forma de obter a matriz identidade é utilizar a função **IdentityMatrix**.

```
IdentityMatrix[3] // MatrixForm
```

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

4.2.4. - RESOLUÇÃO MATRICIAL DE SISTEMAS

Um dos problemas que pode ser solucionado através da utilização do cálculo matricial é a resolução de um sistema de equações lineares.

O sistema:

$$\begin{cases} 2x-3y+5z=7 \\ 7x+y+7z=5 \\ -x+y-z=-4 \end{cases}$$

pode ser traduzido na forma matricial pela igualdade:

$$\begin{pmatrix} 2 & -3 & 5 \\ 7 & 1 & 7 \\ -1 & 1 & -1 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \end{pmatrix}=\begin{pmatrix} 7 \\ 5 \\ 4 \end{pmatrix}.$$

Logo, multiplicando à esquerda pela inversa da matriz dos coeficientes ambos os membros da igualdade anterior, chegamos à solução.

Vejamos então como resolveríamos o sistema anterior em cada um dos programas.

Derive:

Com o objectivo de resolver o sistema anterior, comecemos por determinar a inversa da matriz dos coeficientes.

$$\begin{bmatrix} 2 & -3 & 5 \\ 7 & 1 & 7 \\ -1 & 1 & -1 \end{bmatrix}^{-1}$$

$$\begin{bmatrix} -\frac{1}{3} & \frac{1}{12} & -\frac{13}{12} \\ 0 & \frac{1}{8} & \frac{7}{8} \\ \frac{1}{3} & \frac{1}{24} & \frac{23}{24} \end{bmatrix}$$

Determinemos então a solução para o nosso sistema.

$$\begin{bmatrix} -\frac{1}{3} & \frac{1}{12} & -\frac{13}{12} \\ 0 & \frac{1}{8} & \frac{7}{8} \\ \frac{1}{3} & \frac{1}{24} & \frac{23}{24} \end{bmatrix} \cdot \begin{bmatrix} 7 \\ 5 \\ 4 \end{bmatrix}$$

$$\begin{bmatrix} -\frac{25}{4} \\ \frac{33}{8} \\ \frac{51}{8} \end{bmatrix}$$

O método anterior tem a limitação de poder aplicar-se apenas quando as equações são linearmente independentes. Uma condição necessária e suficiente para que o sistema de equações seja linearmente independente é que a matriz dos coeficientes tenha determinante diferente de zero. Do cálculo de determinantes falaremos já de seguida.

Maple:

Resolvamos também o sistema anterior através do cálculo matricial.

Recorde-se que para multiplicar e calcular a inversa de uma matriz temos de carregar o pacote **linalg** ou então aplicar os comandos **linalg[multiply]** e **linalg[inverse]**.

Optámos por definir a matriz dos coeficientes à parte para uma maior clareza.

```
> with(linalg):

> A:=[[2,-3,5],[7,1,7],[-1,1,-1]]:

> matrix(A);
```

$$\begin{bmatrix} 2 & -3 & 5 \\ 7 & 1 & 7 \\ -1 & 1 & -1 \end{bmatrix}$$

Calculemos então a solução para o nosso sistema.

```
> multiply(inverse(A),[[7],[5],[4]]);
```

$$\begin{bmatrix} \frac{-25}{4} \\ \frac{33}{8} \\ \frac{51}{8} \end{bmatrix}$$

Mathematica:

Vamos resolver no *Mathematica* ainda o mesmo sistema de três equações com três incógnitas usando as matrizes.

Para resolver o sistema basta determinar a inversa da matriz dos coeficientes e multiplicar à esquerda ambos os membros da igualdade anterior. Daí que a solução do sistema anterior é:

```
Inverse[{{2, 3, 5}, {1, 1, 7}, {1, 1, -1}}].{{7}, {5}, {4}}
```

$$\left\{\left\{-\frac{25}{4}\right\}, \left\{\frac{33}{8}\right\}, \left\{\frac{51}{8}\right\}\right\}.$$

Podemos confirmar o resultado através da utilização da função **Solve**.

```
Solve[{2x+3y+5z == 7, 7x+y+7z == 5, -x+y-z == 4}, {x, y, z}]
```

$$\left\{\left\{x \to -\frac{25}{4}, y \to \frac{33}{8}, z \to \frac{51}{8}\right\}\right\}$$

Quer o *Maple* quer o *Derive* disponibilizam funções equivalentes ao **Solve**, nomeadamente **solve** e **SOLVE** respectivamente.

4.2.5. - DETERMINANTES

Derive:

O cálculo de um determinante pode fazer-se tirando partido da função **DET**.

$$\mathrm{DET}\begin{bmatrix} 2 & -3 & 5 \\ 7 & 1 & 7 \\ -1 & 1 & -1 \end{bmatrix}$$

$$24$$

Podemos aplicar a função **DET** a uma matriz arbitrária e assim aceder à forma de calcular um determinante de uma matriz com três linhas e três colunas.

$$\mathrm{DET}\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

$$a\cdot(e\cdot i - f\cdot h) + b\cdot(f\cdot g - d\cdot i) + c\cdot(d\cdot h - e\cdot g)$$

$$\mathrm{DET}\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix}$$

$$a\cdot(f\cdot(k\cdot p - l\cdot o) + g\cdot(l\cdot n - j\cdot p) + h\cdot(j\cdot o - k\cdot n)) - b\cdot(e\cdot(k\cdot p - l\cdot o) + g\cdot(l\cdot m - i\cdot p) + h\cdot(i\cdot o - k\cdot m)) + c\cdot(e\cdot(j\cdot p - l\cdot n) + f\cdot(l\cdot m - i\cdot p) + h\cdot(i\cdot n - j\cdot m)) - d\cdot(e\cdot(j\cdot o - k\cdot n) + f\cdot(k\cdot m - i\cdot o) + g\cdot(i\cdot n - j\cdot m))$$

Maple:

O cálculo do determinante de uma matriz é feito através da função **det**.

```
> det([[1,-3,3],[0,1,-2],[7,1,4]]);
```

$$27$$

Simbolicamente podemos também calcular o determinante de uma matriz arbitrária de dimensão 3 ou 4.

```
> A:=[[a,b,c],[d,e,f],[g,h,i]];
```

$$A := [[a, b, c], [d, e, f], [g, h, i]]$$

```
> det(A);
```

$$a\,e\,i - a\,f\,h - d\,b\,i + d\,c\,h + g\,b\,f - g\,c\,e$$

```
> B := [[a,b,c,d],[e,f,g,h],[i,j,k,l],[m,n,o,p]];
```

$$B := [[a,b,c,d],[e,f,g,h],[i,j,k,l],[m,n,o,p]]$$

```
> det(B);
```

$$afkp - aflo - ajgp + ajho + angl - anhk - ebkp + eblo + ejcp - ejdo - encl + endk + ibgp - ibho - ifcp + ifdo + inch - indg - mbgl + mbhk + mfcl - mfdk - mjch + mjdg$$

Mathematica:

O determinante de uma matriz pode obter-se através do comando **Det**.

```
-20
```

Facilmente podemos ter acesso à forma com se calcula o determinante de uma matriz de três linhas por três colunas ou até mesmo com quatro linhas e quatro colunas.

```
-ceg+bfg+cdh-afh-bdi+aei
```

```
dgjm-chjm-dfkm+bhkm+cflm-bglm-dgin+chin+
 dekn-ahkn-celn+agln+dfio-bhio-dejo+ahjo+
 belo-aflo-cfip+bgip+cejp-agjp-bekp+afkp
```

4.2.6. - TRAÇO DE UMA MATRIZ

Derive:

O traço de uma matriz quadrada sendo definido como a soma dos elementos da diagonal principal, também pode ser calculado directamente recorrendo a uma função predefinida.

$$\text{TRACE}\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix}$$

$$15$$

Maple:

Para calcularmos o traço de uma matriz quadrada temos de carregar o pacote *linalg*.

```
> with(linalg):
Warning, new definition for norm
Warning, new definition for trace
```

O cálculo do traço de uma matriz faz-se recorrendo à função **trace**.

```
> trace( array([[1,2],[1,4]]) );
```

$$5$$

```
> A := [[1,2,3],[4,5,6],[7,8,9]];
```

$$A := [[1, 2, 3], [4, 5, 6], [7, 8, 9]]$$

```
> trace(A);
```

$$15$$

Mathematica:

O traço de uma matriz quadrada pode ser obtido no *Mathematica* através da função **Tr**.

```
Tr[{{a, b, c, d}, {e, f, g, h}, {i, j, k, l}, {m, n, o, p}}]
```

```
a + f + k + p
```

```
Tr[{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}]
```

```
15
```

4.2.7. - TRANSPOSTA DE UMA MATRIZ

Derive:

A transposta de uma matriz pode ser obtida através de ` (acento grave).

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}`$$

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

Maple:

O cálculo da transposta de uma matriz ou de um vector faz-se pela função **transpose**.

```
> A := matrix(3,3, [1,2,3,4,5,6,7,8,9]);
```

$$A := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
> transpose(A);
```

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

Mathematica:

Podemos obter a transposta de uma matriz utilizando a função predefinida **Transpose**.

Note-se que a transposta de uma matriz está definida, em qualquer um dos programas mesmo que as matrizes não sejam quadradas.

4.2.8. - VECTORES E VALORES PRÓPRIOS DE UMA MATRIZ

Derive:

Pretendendo obter os valores e vectores próprios através do *Derive* podemos em primeiro lugar obter o polinómio característico através do uso da função **CHARPOLY**.

$$\text{CHARPOLY}\left(\begin{bmatrix} 1 & 2 & 0 \\ 2 & -2 & 0 \\ 0 & 0 & 3 \end{bmatrix}, x\right)$$

$$(3 - x)\cdot(x^2 + x - 6)$$

Os valores próprios, que são os zeros do polinómio característico, podem ser obtidos directamente pela função **EIGENVALUES**.

$$\text{EIGENVALUES}\left(\begin{bmatrix} 1 & 2 & 0 \\ 2 & -2 & 0 \\ 0 & 0 & 3 \end{bmatrix}, x\right)$$

$$[x = 2,\ x = 3,\ x = -3]$$

Carregando o ficheiro *Vector.MTH* através do menu *File* e do submenu *Load*, seleccionando *Math* e o ficheiro *Vector* é possível dispor de mais algumas funções para trabalhar com vectores e com matrizes. Entre essas funções está a função **EXACT_EIGENVECTOR** que nos permite calcular os vectores próprios associados a um determinado valor próprio.

$$\text{EXACT\_EIGENVECTOR}\left(\begin{bmatrix} 1 & 2 & 0 \\ 2 & -2 & 0 \\ 0 & 0 & 3 \end{bmatrix}, 3\right)$$

$$[[0,\ 0,\ @1]]$$

$$\text{EXACT\_EIGENVECTOR}\left(\begin{bmatrix} 1 & 2 & 0 \\ 2 & -2 & 0 \\ 0 & 0 & 3 \end{bmatrix}, -3\right)$$

$$[[@2,\ -2\cdot @2,\ 0]]$$

$$\text{EXACT\_EIGENVECTOR}\left(\begin{bmatrix} 1 & 2 & 0 \\ 2 & -2 & 0 \\ 0 & 0 & 3 \end{bmatrix}, 2\right)$$

$$\left[\left[@3,\ \frac{@3}{2},\ 0\right]\right]$$

Nos resultados anteriores @1, @2 e @3 representam valores arbitrários.

Maple:

Carregando o pacote *linalg*, a que já nos referimos anteriormente, podemos calcular a expressão do polinómio característico de uma matriz quadrada.

```
> with(linalg):
Warning, new definition for norm
Warning, new definition for trace

> A:=[[0,4,0],[2,-2,0],[0,0,3]]);
```

$$A := [[0, 4, 0], [2, -2, 0], [0, 0, 3]]$$

```
> matrix(A);
```

$$\begin{bmatrix} 0 & 4 & 0 \\ 2 & -2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

```
> charmat(A,lambda);
```

$$\begin{bmatrix} \lambda & -4 & 0 \\ -2 & \lambda+2 & 0 \\ 0 & 0 & \lambda-3 \end{bmatrix}$$

```
> charpoly(A,lambda);
```

$$\lambda^3 - \lambda^2 - 14\,\lambda + 24$$

Os valores próprios podem ser obtidos utilizando a função **eigenvals**.

```
> eigenvals(A);
```

$$-4, 2, 3$$

A função **eigenvectors** dá-nos os vectores próprios associados a cada um dos valores próprios.

```
> eigenvectors(A);
```

$$[3, 1, \{[0, 0, 1]\}], [-4, 1, \{[-1, 1, 0]\}], [2, 1, \{[2, 1, 0]\}]$$

O *output* anterior diz-nos que o valor próprio 3 tem multiplicidade 1 e o vector próprio a si associado é (0, 0, 1), o valor próprio -4 tem multiplicidade 1 e o vector próprio a si associado é (-1, 1, 0) e por último que o valor próprio 2, também com multiplicidade 1 tem como vector próprio associado o (2, 1, 0).

Refira-se que as funções anteriores podem ser utilizadas sem carregar todo o pacote *linalg* se usarmos, em vez da forma abreviada a que estamos habituados, as funções **linalg[eigenvals], linalg[eigenvectors], linalg[charpoly]** e **linalg[charmat]**. O procedimento anterior carrega apenas a função especificada.

Mathematica:

Não dispondo de uma função predefinida para obter o polinómio característico, o *Mathematica* consegue fornecê-lo se utilizarmos a própria definição de polinómio característico.

```
A= {{1, 1, 1}, {1, 2, 0}, {1, 0, 6}}
```

```
Det[A - x IdentityMatrix[3]]
```

$$12 - 20\,x + 9\,x^2 - x^3$$

A obtenção dos valores próprios é feita pela utilização da função predefinida **Eigenvalues.**

```
Eigenvalues[A]
```

$$\{1, 2, 6\}$$

Da mesma forma se obtêm os vectores próprios.

```
Eigenvectors[A]
```

$$\{\{-2, 0, 1\}, \{-1, -4, 1\}, \{5, 1, 5\}\}$$

O primeiro vector é o que está associado ao primeiro valor próprio, o segundo vector é o que está associado ao segundo valor próprio e assim sucessivamente.

Se for do nosso interesse obter simultaneamente os valores próprios e vectores próprios, tal pode ser conseguido através da função **Eigensystem** que nos dará uma lista de listas em que a primeira é constituída pelos valores próprios e a segunda é constituída pelos vectores próprios associados a cada um dos valores próprios da primeira lista .

```
Eigensystem[A]
```

$$\{\{1, 2, 6\}, \{\{-2, 0, 1\}, \{-1, -4, 1\}, \{5, 1, 5\}\}\}$$

4.3. - POLINÓMIOS

4.3.1. - COEFICIENTES DE UM POLINÓMIO

Derive:

Podemos aceder aos coeficientes de um dado polinómio se recorrermos ao menu *File* e ao submenu *Load*, escolhendo *Math* e optando por *Misc*. Desta forma estamos a tornar possível a utilização da função **POLY_COEFF**.

A função **POLY_COEFF** diz-nos qual o coeficiente do termo do polinómio de grau igual ao seu terceiro argumento.

```
POLY_COEFF(x^2 - 4·x - 1, x, 2)

                                                    1

POLY_COEFF(x^2 - 4·x - 1, x, 1)

                                                   -4
```

Ou seja, o termo de grau **2** tem coeficiente igual a **1** e o termo de grau **1** tem coeficiente igual a **– 4**. Vejamos ainda um último exemplo um pouco mais rebuscado.

```
POLY_COEFF((x + 2)·x^2 - x^2·(1 + n), x, 2)

                                                  1 - n
```

Maple:

A função correspondente em *Maple* é **coeff**.

Para que possamos determinar os coeficientes de um dado polinómio temos de utilizar a função **collect**. Esta função simplesmente junta os coeficientes dos termos que têm o mesmo grau no polinómio.

```
> collect(3*x+y*x,x);
                                                 (3 + y) x

> collect((x^4-3*x+1)*(x^2+5*x)+1+2*x,x);
                                        x^6 + 5 x^5 - 3 x^3 - 14 x^2 + 7 x + 1
```

Se não usarmos a função **collect** a função **coeff** poderá não funcionar correctamente como se pode ver no último dos exemplos que de seguida apresentamos.

```
> coeff(3*x^2+2*x+5*x+1,x^2);
                                                    3
```

```
> coeff(3*x^2+2*x+5*x+1,x);
```

$$7$$

```
> polino:=(x+2)*x^2-x^2*(1+n);
```

$$polino := (x+2)\,x^2 - x^2\,(1+n)$$

```
> coeff(polino,x^2);
```

$$-1-n$$

```
> coeff(collect(polino,x),x^2);
```

$$1-n$$

Mathematica:

No *Mathematica* temos a possibilidade de conhecer os coeficientes de um determinado polinómio numa dada variável. No exemplo seguinte a variável considerada é **x**.

```
CoefficientList[x^4 + 2 x^3 y + 5 x y^2 + 13 y^4, x]
{13 y^4, 5 y^2, 0, 2 y, 1}
```

O último *output* indica-nos que o coeficiente do termo de grau zero é **13 y^4**, o coeficiente do termo de grau um é **2 x^3**, que o coeficiente do termo de grau dois é **0** e assim sucessivamente.

Ainda a este propósito refira-se que em *Mathematica* também está definida a função **Collect**, mas não sentimos qualquer necessidade de a utilizar aquando da determinação do coeficiente de um dado grau.

Vejamos aqui o exemplo atrás apresentado para justificar a utilização da função **collect** no *Maple*.

```
(1 + 2 x) (-x^2 + x^2 (1 + n))
-(1 + n) (-x^2 + x^2 (1 + x))
```

```
CoefficientList[%, x]
{0, 0, 1 - n, 1}
```

Conseguimos aqui obter o resultado correcto sem ter de recorrer à função anterior.

4.3.2. - GRAU DE UM POLINÓMIO

Derive:

De forma análoga à utilizada quando determinamos os coeficientes, também para determinar o grau de um polinómio temos de recorrer ao menu *File* e ao *submenu Load*, escolhendo *Math* e optando por *Misc*.

O grau de um polinómio numa variável pode ser determinado através da utilização da função **POLY_DEGREE.**

```
POLY_DEGREE(x^2 - 4·x - 1, x)

                                                    2

POLY_DEGREE(5·x^2·y^3 - 4·x·y^4 - 1, x)

                                                    2

POLY_DEGREE(5·x^2·y^3 - 4·x·y^4 - 1, y)

                                                    4
```

Maple:

O grau de um polinómio pode ser determinado pela função predefinida **degree**.

```
> degree(3*x^3+2*x^5 -2.5*x);

                                                    5

> degree(3*x^2*y+2*x^2*y^3-x);

                                                    5

> degree(x^5+3*x^3-x^5+y^4);

                                                    4
```

Mathematica:

Não conseguimos encontrar uma função predefinida que nos calculasse o grau de um polinómio mas, através de algumas das funções que já analisámos podemos obter o grau de um polinómio se considerarmos que na lista dos coeficientes aparecem todos os coeficientes incluído o de grau zero. Isto mostra que o comprimento desta lista excede sempre em uma unidade o grau do polinómio. Assim sendo basta-nos calcular o comprimento da referida lista se dela retirarmos um dos elementos. Neste caso e por questões de simplicidade optámos por retirar o primeiro elemento da referida lista. Obtivemos desta forma uma expressão composta por três funções predefinidas que nos fornece o grau de um polinómio.

```
Length Rest CoefficientList x^3+2* x, x
```
```
3
```

```
Length Rest CoefficientList x^15+2* x^3+1, x
```
```
15
```

Contudo é de salientar que caso o polinómio tenha mais do que uma variável esta composição de funções nem sempre nos dará o grau correcto para o polinómio já que o grau de um monómio é definido como a soma dos expoentes das suas partes literais. No exemplo que se segue o grau do polinómio é 17 que não corresponde ao resultado obtido pela função. Repare-se que ao darmos esta definição e, em particular, ao utilizarmos a função **CoefficientList** estamos a dizer que o polinómio deve ser considerado na variável **x** e não em **x** e **y** simultaneamente.

```
Length Rest CoefficientList y+x^15* y^2+2* x^3+1, x
```
```
15
```

4.3.3. - ORDENAÇÃO DE UM POLINÓMIO

Derive:

Conseguir que um polinómio fique ordenado de acordo com os graus dos seus termos é um problema de fácil resolução nem havendo o utilizador de preocupar-se em utilizar uma função predefinida para ultrapassar essa dificuldade.

Assim sendo, o objectivo de conseguir um polinómio nas condições já referidas, pode ser atingido bastando para tal introduzir a expressão do referido polinómio e depois simplesmente carregar no atalho =.

$$2\cdot x^2 + 4\cdot x^5 + 3\cdot x + 4\cdot x^{11}$$

$$4\cdot x^{11} + 4\cdot x^5 + 2\cdot x^2 + 3\cdot x$$

Maple:

Podemos obter um polinómio ordenado por ordem decrescente dos expoentes dos seus termos através da função **sort**. Antes porém, vejamos que quando introduzimos um polinómio sem utilizar o comando anterior os seus monómios permanecem pela ordem introduzida.

```
> 3*x^5+4*x^2-x^7+1;
```

$$3x^5+4x^2-x^7+1$$

Pela utilização da função **sort** conseguimos que o polinómio fique ordenado por ordem decrescente dos graus de cada um dos monómios que o compõem.

```
> sort(3*x^5+4*x^2-x^7+1);
```

$$-x^7+3x^5+4x^2+1$$

Reparemos contudo que, nomeadamente nos casos em que precisamos de recorrer a uma atribuição para melhor e mais facilmente efectuarmos o nosso trabalho, o comando **sort** automaticamente faz com que o polinómio inicial fique também alterado, não havendo nestes casos a necessidade de fazer uma nova atribuição.

```
> pol_inicial:=3*x^5+4*x^7-3.5*x^3+4*x^6+2;
```

$$pol\_inicial := 3x^5+4x^7-3.5x^3+4x^6+2$$

```
> sort(pol_inicial);
```

$$4x^7+4x^6+3x^5-3.5x^3+2$$

```
> pol_inicial;
```

$$4x^7+4x^6+3x^5-3.5x^3+2$$

A ordem do polinómio que tínhamos inicialmente foi modificada, passando também ele a ter os seus termos ordenados de forma decrescente pelo grau dos mesmos.

De referir também que, quando um polinómio apresenta mais que uma variável a questão da ordenação é feita através da soma dos expoentes da parte literal de cada termo. O *Maple* dá-nos também a possibilidade de quando o polinómio tem mais do que uma variável, ordenarmos os termos usando como critério de ordenação primeiro uma variável e só considerando a(s) outra(s) depois. Tal é possível introduzindo na função um 2º argumento constituído por uma lista onde se indicam as variáveis pela ordem pretendida e um 3º (**'plex'**) que indica que a ordem pela qual os termos devem ser ordenados deverá ser em primeiro lugar os graus do primeiro elemento da lista.

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

$$pol\_segundo := 2y^4+3x^3y^3+x^2$$

```
> sort(pol_segundo);
```

$$3x^3y^3+2y^4+x^2$$

```
> sort(pol_segundo,[x,y],'plex');
```

$$3x^3y^3+x^2+2y^4$$

```
> sort(pol_segundo,[y,x],'plex');
```

$$2y^4+3y^3x^3+x^2$$

Mathematica:

No *Mathematica* ao contrário do que acontece no *Maple*, a utilização da função predefinida **Sort** não provoca qualquer efeito no que diz respeito à ordenação de polinómios. O *Mathematica* simplifica e ordena qualquer expressão introduzida e em particular os polinómios como se pode ver no *output* do próximo exemplo.

```
3* x^7+ 3* x^8+ 4* x+ 2* x^2
```

$$4x+2x^2+3x^7+3x^8$$

Desta forma sempre que introduzimos um polinómio ele imediatamente surge ordenado pela ordem crescente dos graus dos seus termos.

4.3.4. - EXPANSÃO

Derive:

Quando estamos a trabalhar com polinómios há casos em que é útil obtermos uma expressão para o polinómio que temos sem que este tenha por exemplo parênteses. Através da função **EXPAND** podemos resolver esse problema.

Esta função desenvolve o mais possível o polinómio, aplicando nomeadamente a propriedade distributiva e reduzindo os termos semelhantes. Nem sempre é necessário utilizar a função **EXPAND**, como de seguida se mostra, mas casos há em que tal é necessário. Um desses casos é o segundo exemplo. Aí se não utilizarmos esta função, nada se altera.

$$(x^2 + 2\cdot x - 1)\cdot(x + 1) + 3 - 13\cdot x$$

$$x^3 + 3\cdot x^2 - 12\cdot x + 2$$

$$\mathrm{EXPAND}((x^2 + 2\cdot x - 1)\cdot(x + 1) + 3 - 13\cdot x)$$

$$x^3 + 3\cdot x^2 - 12\cdot x + 2$$

$$\mathrm{EXPAND}((x + 1)\cdot(x - 3)^2)$$

$$x^3 - 5\cdot x^2 + 3\cdot x + 9$$

Maple:

De modo semelhante ao anterior também se pode desenvolver um dado polinómio no *Maple* se recorrermos também à função predefinida **expand**.

```
> expand((x-1)*(x+3)*(x+2));
```

$$x^3+4x^2+x-6$$

A função **expand** pode ser invocada com dois ou mais argumentos. Tal é feito quando não queremos que determinadas subexpressões sejam desenvolvidas.

```
> expand((x+1)*(y+z),x+1);
```

$$(x+1)y+(x+1)z$$

```
> expand((x+1)*(y+z),x+2);
```

$$xy+xz+y+z$$

Mathematica:

Podemos mandar desenvolver um polinómio através da função **Expand**.

```
Expand [redacted]
3 + x^2 + 3 x^3 + x^5
```

```
Expand [redacted]
2 + x - 4 x^2 - 2 x^3 + 2 x^4 + x^5
```

Utilizando a função **Expand** com dois argumentos fica o segundo reservado para a expressão que se pretende desenvolvida. Todos os factores que são iguais ao segundo argumento são desenvolvidos enquanto que os restantes permanecem tal e qual no início.

```
Expand [redacted]
1 + 3 x + 3 x^2 + x^3 + [redacted]
```

Portanto o *Mathematica* desenvolveu o mais possível a expressão sem infringir a regra atrás referida de simplificar apenas aquelas que continham a subexpressão **1+x**.

4.3.5. - FACTORIZAÇÃO

Derive:

Podemos também recorrer à função **FACTOR** quando pretendemos obter um polinómio factorizado.

$$\text{FACTOR}\left(x^4 + \frac{11\cdot x^3}{2} - 3\cdot x^2 - \frac{29\cdot x}{2} + 11,\ x\right)$$

$$\frac{(x+2)\cdot(x-1)^2\cdot(2\cdot x+11)}{2}$$

É evidente que nem sempre obtemos o resultado pretendido, uma vez que o *Derive* nem sempre consegue obter uma factorização. De seguida tentamos factorizar o polinómio $\mathbf{x^3+3x^2-12x+2}$ a que já nos referimos anteriormente.

$$\text{FACTOR}(x^3 + 3\cdot x^2 - 12\cdot x + 2,\ x)$$

$$x^3 + 3\cdot x^2 - 12\cdot x + 2$$

O *Derive* não conseguiu encontrar uma factorização para o polinómio. Como veremos adiante nenhum dos programas aqui analisados conseguiu encontrar uma factorização para este polinómio.

Maple:

Utilizando a função **factor** podemos factorizar um dado polinómio:

```
> factor(x^3+4x^2+x-6);
Syntax error, missing operator or `;` .
```

O erro ocorrido deve-se ao facto de o *Maple* não considerar **4x** como sendo o produto de **4** por **x**. Corrijamos o erro para prosseguirmos colocando a operação de multiplicação entre o número e a variável.

```
> factor(x^3+4*x^2+x-6);
```

$$(x-1)(x+3)(x+2)$$

No entanto, repare-se que a função **factor** nem sempre factoriza o polinómio que nós introduzimos.

```
> factor(x^3+3*x^2-12*x+2);
```

$$x^3+3x^2-12x+2$$

Neste caso obtivemos como *output* um polinómio exactamente igual ao introduzido no argumento da função **factor**, uma vez que o *Maple* não conseguiu encontrar uma factorização.

Mathematica:

A utilização da função **Factor** para encontrar uma factorização para um polinómio constitui uma boa escolha se estivermos a trabalhar com o *Mathematica*.

```
Factor[1 + x^2 - 3 x^3 + x^5]
(1 + x)^2 (1 - x)^2 (1 - x + x^2)
```

A função **Factor** nem sempre consegue obter uma factorização para um polinómio, como no seguinte exemplo:

```
Factor[x^3 + 3 x^2 - 12 x + 2]
2 - 12 x + 3 x^2 + x^3
```

.

4.3.6. - RESTO E QUOCIENTE

Derive:

Podemos utilizar o *Derive* para calcular o resto e o quociente da divisão de dois polinómios, através das funções **REMAINDER** e **QUOCIENT** respectivamente, funções estas a que já nos referimos anteriormente, quando analisámos as operações sobre inteiros.

```
QUOTIENT(x^3 + 3·x^2 - 12·x + 2, x^2 + 2·x - 1)
                                              x + 1

REMAINDER(x^3 + 3·x^2 - 12·x + 2, x^2 + 2·x - 1)
                                          3 - 13·x
```

Maple:

Conseguimos facilmente determinar o quociente e o resto da divisão de dois polinómios utilizando as funções **quo** e **rem**.

```
> polino:=x^6+5*x^5-3*x^3-14*x^2+7*x+1;
```

$$polino := x^6 + 5x^5 - 3x^3 - 14x^2 + 7x + 1$$

```
> quo(polino,x^2+5*x,x);
```

$$x^4 - 3x + 1$$

```
> rem(polino,x^2+5*x,x);
```

$$1 + 2x$$

Verifiquemos que **$(x^4-3x+1)*(x^2+5x)+2x+1$** dá realmente o polinómio que anteriormente designamos por **polino**.

```
> expand((x^4-3*x+1)*(x^2+5*x)+1+2*x);
```

$$x^6 + 5x^5 - 3x^3 - 14x^2 + 7x + 1$$

Mathematica:

Tal como a factorização, a divisão de polinómios também está contemplada no que a funções predefinidas diz respeito. Existem funções que fornecem o quociente e o resto da divisão de dois polinómios.

```
PolynomialQuotient[x^2, x^2 + 2x, x]
```

$$-\frac{1}{4} + \frac{x}{2}$$

```
PolynomialRemainder[x^2, x^2 + 2x, x]
```

$$\frac{1}{4}$$

Podemos efectuar a multiplicação do quociente pelo divisor e somar com o resto para assim verificarmos os resultados obtidos.

```
Expand[(1 - 4 x) (2 + 2 x) + 1/4]
```

$$x^2$$

4.3.7. - MÁXIMO DIVISOR COMUM DE DOIS POLINÓMIOS

Derive:

Calcular o máximo divisor comum de dois polinómios é uma tarefa bastante simplificada requerendo a utilização da função **POLY_GCD**.

```
POLY_GCD(x^2 - 5·x + 6, x^3 - 5·x^2 + 3·x + 9)
                                          x - 3
```

Maple:

A utilização da função **gcd** permite-nos obter o máximo divisor comum de dois polinómios.

```
> gcd(x+2,x^2+4*x+4);
```

$$x+2$$

A função ainda pode ser invocada com mais dois argumentos nos quais são guardados os polinómios tais que o seu produto pelo polinómio que representa o máximo divisor comum dá os polinómios iniciais.

```
> gcd(x^2+3*x+2,x^2+4*x+3,'s','t');
```

$$x+1$$

```
> s,t;
```

$$x+2, x+3$$

```
> expand((x+2)*(x+1));
```

$$x^2+3x+2$$

Para concluir refira-se que a obtenção do máximo divisor comum de dois polinómios como combinação linear dos respectivos polinómios pode fazer-se recorrendo à função **gcdex**.

```
> gcdex(x^2+3*x+2,x^2+4*x+3,x,'s','t');
```

$$x+1$$

```
> s,t;
```

$$-1, 1$$

```
> -1*(x^2+3*x+2)+(x^2+4*x+3);
```

$$x+1$$

Mathematica:

Usando apenas funções predefinidas podemos calcular o máximo divisor comum de dois polinómios exactamente da mesma forma como determinaríamos se estivéssemos a trabalhar com inteiros aplicando a função **PolynomialGCD**. Refira-se que esta função pode utilizar-se para determinar o máximo divisor comum de dois números, mas aí estamos implicitamente a calcular o máximo divisor comum de dois polinómios de grau zero. Tanto no *Maple* como no *Derive* podemos fazer o mesmo, isto é, utilizar as respectivas funções predefinidas para o cálculo do máximo divisor comum de polinómios no tratamento de inteiros.

```
PolynomialGCD[4 + x - 4 x^2 - 2 x^3 + 2 x^4 + x^5, 6 - x - 4 x^2 - x^3]
```

$$-2 + x + x^2$$

4.3.8. - MÍNIMO MÚLTIPLO COMUM DE DOIS POLINÓMIOS

Derive:

Não nos foi possível encontrar uma função predefinida que calculasse directamente o mínimo múltiplo comum de dois polinómios. Uma forma de utilizar o *Derive* para obtê-lo é recordar que o produto do máximo divisor comum pelo mínimo múltiplo comum de dois polinómios é igual ao produto dos dois polinómios. Assim, para conseguirmos obter o mínimo múltiplo comum temos de multiplicar os dois polinómios e depois dividir pelo máximo divisor comum.

$$\frac{(x^2 - 5\cdot x + 6)\cdot(x^3 - 5\cdot x^2 + 3\cdot x + 9)}{\text{POLY\_GCD}(x^2 - 5\cdot x + 6,\ x^3 - 5\cdot x^2 + 3\cdot x + 9)}$$

$$(x^2 - 2\cdot x - 3)\cdot(x^2 - 5\cdot x + 6)$$

$$\text{EXPAND}((x^2 - 2\cdot x - 3)\cdot(x^2 - 5\cdot x + 6))$$

$$x^4 - 7\cdot x^3 + 13\cdot x^2 + 3\cdot x - 18$$

Maple:

A determinação do mínimo múltiplo comum de dois polinómios, tal como acontece com o máximo divisor comum, é muito semelhante à forma que já ilustrámos atrás quando nos referimos ao cálculo do mínimo múltiplo comum e máximo divisor comum de inteiros.

```
> lcm(x^2+3*x+2,x^2+4*x+3);
```

$$x^3+6x^2+11x+6$$

Mathematica:

Tal como acontecia para o máximo divisor comum de dois polinómios, aqui também há a necessidade de colocar a designação *Polynomial* antes de **LCM**.

```
PolynomialLCM[2 + x - 4 x^2 - 2 x^3 + 2 x^4 + x^5, 6 - x - 4 x^2 - x^3]
 3 - x - 4 x^2 - x^3 (1 - x + x^2 + x^3)
```

Apesar do resultado ser apresentado de forma factorizada, se assim o desejássemos, poderíamos utilizar a função **Expand** e assim obter um polinómio desenvolvido.

```
Expand[%]
```

$$-6 - 5x + 11x^2 + 10x^3 - 4x^4 - 5x^5 - x^6$$

4.3.9. - INTERPOLAÇÃO POLINOMIAL

Derive:

O *Derive* possibilita-nos ainda que encontremos um polinómio interpolador, utilizando para tal a função **POLY_INTERPOLATE**, onde o primeiro argumento é uma matriz onde constam os pontos (cada linha representa um ponto) e o segundo a variável. Note-se que para podermos tirar partido desta função como em casos anteriores temos de recorrer ao menu *File* e ao submenu *Load*, escolhendo *Math* e optar por *Misc*.

$$\text{POLY\_INTERPOLATE}\left(\begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}, x\right)$$

$$x$$

$$\text{POLY\_INTERPOLATE}\left(\begin{bmatrix} 2 & 4 \\ 1 & 1 \\ 5 & 25 \end{bmatrix}, x\right)$$

$$x^2$$

$$\text{POLY\_INTERPOLATE}\left(\begin{bmatrix} 1 & -4 \\ 3 & -4 \\ 4 & -1 \end{bmatrix}, x\right)$$

$$x^2 - 4 \cdot x - 1$$

Maple:

A construção de um polinómio interpolador pode fazer-se pela utilização da função **interp** em que têm de ser fornecidas duas listas correspondentes respectivamente aos valores da variável independente e aos valores da variável dependente bem como a variável em que queremos o nosso polinómio definido. Vejamos no primeiro caso um possível polinómio que passa nos pontos de coordenadas (1,4) e (3,12).

```
> interp([1,3],[4,12],x);
```

$$4x$$

Vejamos ainda mais alguns exemplos.

```
> interp([2,3,5],[8,25,119],x);
```

$$10x^2 - 33x + 34$$

```
> interp([2,3,5,1],[8,25,119,3],x);
```

$$x^3 - 2x + 4$$

Mathematica:

No *Mathematica* podemos, conhecidos os pontos, definir um polinómio interpolador para os mesmos. A função **InterpolatingPolynomial** é invocada com dois argumentos, onde o primeiro é uma lista e o segundo é a variável em que queremos definir o nosso polinómio.

A lista que constitui o primeiro argumento pode ainda ser constituída por outras listas que nesse caso representam os pontos **(x, p(x))**.

Se a lista for uma lista de inteiros ou reais, então o primeiro elemento da mesma é a imagem de 1, o segundo elemento é a imagem do **2** e por aí adiante.

Neste primeiro exemplo, pretendemos um polinómio que passa pelos pontos **(1,2)** e **(2,4)**. Para obtermos o pretendido, basta fazermos:

```
InterpolatingPolynomial[
2+2 (1+x)

Simplify[
2x
```

Deste primeiro exemplo retiramos também a utilidade que tem a função **Simplify**, que procura simplificar as expressões que constituem o seu argumento.

Suponhamos agora que pretendemos interpolar os pontos **(2,4)**, **(3,9)** e **(7,49)** então faríamos:

```
InterpolatingPolynomial[
4+ (2+x)
Simplify[
x^2
```

Apesar de nos casos anteriores termos obtido o que seria de prever, tal nem sempre acontece, pois o polinómio interpolador não é único.

```
InterpolatingPolynomial[
11+ (-3+
11 (-3+x)+ x^2

InterpolatingPolynomial[
11+ (2+x)
Simplify[
-7+5x+x^3
```

4.3.10. - PREDICADOS DE E PARA POLINÓMIOS

Maple:

Embora, através da função **rem** facilmente consigamos verificar se um polinómio é divisível por outro, tal constatação é ainda possível usando o predicado **divide** que dará **true** nos casos em que o primeiro argumento é divisível pelo segundo e **false** caso tal não aconteça. Este predicado, tal como outros ganhará particular relevância quando passarmos ao campo da programação.

```
> divide(x^2+5*x+6,x+2);
```

true

```
> divide(x^2+5*x+6,x-3);
```

false

Mathematica:

A função **PolynomialQ** permite-nos saber se uma dada expressão é ou não um polinómio na variável que surge no segundo argumento.

```
PolynomialQ[x^2+4x+3, x]
True

PolynomialQ[x^4+3x^2, x]
False
```

4.3.11. - VARIÁVEIS DE UM POLINÓMIO

Derive:

No *Derive* podemos através da função **VARIABLES** saber quais as expressões que são consideradas variáveis. Repare-se que as variáveis nem sempre correspondem às diferentes partes literais que surgem numa dada expressão. Um desses casos é quando alguma das expressões foi alvo de uma atribuição.

VARIABLES($2 \cdot x^2 + 3 \cdot x + y$)

[x, y]

x := 3

3

VARIABLES($2 \cdot x^2 + 3 \cdot x + y$)

[y]

Mathematica:

Embora tal seja quase sempre evidente, se quisermos podemos saber quais as variáveis com que estamos a trabalhar como se exemplifica de seguida.

```
Variables[x^3+y^4+4z]
{x, y, z}
```

```
Variables[x^3+y^2-y^2]
{x}
```

Tendo alguma letra sido alvo de uma atribuição já não é uma variável.

```
x=3;
Variables[x^3+y^2+2y^2]
{y}
```

No caso anterior, o **x** não é uma variável. Para que volte a sê-lo basta fazer **Clear[x]** ou **Remove[x]**.

4.4. - CONCLUSÃO

O cálculo vectorial que desenvolvemos em cada programa permitiu concluir que as operações processam-se da mesma forma equiparando-se as capacidades de cada um deles.

Em relação ao tratamento das matrizes existem atalhos disponíveis para as introduzir em qualquer dos programas em análise. No entanto, os atalhos que o *Derive* e o *Mathematica* possuem são mais eficientes uma vez que nestes dois a matriz é introduzida na forma usual enquanto que no *Maple* ainda que utilizemos o atalho presente nas *palettes* aparece-nos na zona de trabalho uma lista de listas na qual temos de preencher as entradas, procedimento este que não é tão eficaz se considerarmos o aspecto visual. No *Maple* existe uma função predefinida que permite que uma lista de listas seja mostrada na forma de matriz (desde que a sua representação seja uma matriz) o mesmo acontecendo com o *Mathematica*. No *Derive* desde que a lista de listas represente uma matriz é com essa forma que é mostrada na zona de trabalho.

As operações aritméticas de soma e subtracção efectuam-se da forma comum desde que as dimensões das respectivas matrizes o permitam. Quanto à multiplicação existem funções predefinidas destinadas a essa operação.

No cálculo de potências de uma matriz a notação usual é utilizada em *Derive* e em *Maple* havendo neste último que utilizar uma função especial para conseguir obter o resultado final. No *Mathematica* é necessário utilizar uma função predefinida – neste caso **MatrixPower**.

Nos três programas é possível resolver um sistema através do cálculo matricial e calcular o determinante de uma matriz quadrada de maneira semelhante.

Tanto no *Maple* como no *Derive* existem funções predefinidas que permitem obter directamente o polinómio característico de uma matriz o que não acontece no *Mathematica*. Para conseguir obtê-lo temos de aplicar a definição do mesmo. O *Maple* é o único no qual é possível visualizar a matriz característica.

Existem também funções predefinidas que calculam os valores e vectores próprios de uma matriz.

Na determinação dos coeficientes de um determinado grau o *Maple* não consegue, através do uso exclusivo da função **coeff**, dar uma resposta satisfatória em todos os casos sendo necessário recorrer ainda à função **collect**. Nos outros programas não sentimos necessidade de fazer o mesmo uma vez que, as funções **POLY_COEFF** e **CoefficientList** equivalentes à já referida função **coeff** resolveram bem as situações por nós introduzidas.

Ao contrário do que acontece nos outros programas, o *Mathematica* não dispõe de uma função predefinida que calcule o grau de um polinómio.

O *Derive* e o *Mathematica* ordenam automaticamente os polinómios por ordem decrescente e crescente dos graus dos seus termos, respectivamente. No *Maple* o mesmo é conseguido utilizando a função **sort**.

No que concerne à expansão e à factorização, os três programas comportam-se da mesma forma o mesmo acontecendo em relação ao resto e ao quociente da divisão de dois polinómios.

O máximo divisor comum de dois polinómios pode ser determinado directamente em qualquer programa sendo que no *Maple* temos a possibilidade de escrevê-lo como combinação linear dos polinómios em questão. No *Derive* não nos foi possível encontrar uma função predefinida que nos fornecesse o mínimo múltiplo

comum de dois polinómios. Para solucionar este problema definimo-lo à custa do máximo divisor comum dos polinómios e do seu produto.

No que diz respeito a predicados relacionados com polinómios eles diferem de programa para programa não tendo sido possível encontrar algum definido em *Derive*.

No *Maple* também não nos foi possível encontrar uma forma de obter directamente as variáveis de um polinómio.

Por último a interpolação polinomial faz-se de maneira muito igual em todos os programas diferindo apenas no nome que cada função apresenta consoante o programa em que estamos.

Em suma, existem alguns aspectos em que algum dos programas se evidencia mas, essa mais valia não é significativa se analisarmos o programa de uma forma mais geral em relação ao tratamento ao qual foram sujeitos os polinómios.

Em resumo:

- A introdução de matrizes na forma usual é possível no *Derive* e no *Mathematica*.
- No *Maple* a determinação dos coeficientes de um polinómio exige, em algumas situações, o recurso a duas funções predefinidas.
- O *Mathematica* não dispõe de uma função predefinida para calcular o grau de um polinómio.
- No *Derive* não encontramos forma de calcular directamente o mínimo múltiplo comum de dois polinómios.
- No *Maple* é possível escrever o máximo divisor comum de dois polinómios como combinação linear dos mesmos.

Não existem diferenças significativas entre os três programas sendo aquelas que aqui apresentamos apenas diferenças pontuais que não são suficientes para destacar qualquer dos programas em relação aos outros.

5. - PROGRAMAÇÃO

Apesar de serem muitas as funções predefinidas que qualquer um dos programas nos apresenta, permitindo que resolvamos com grande facilidade alguns dos nossos problemas, existem sempre casos em que nos seria útil ter uma função mais específica ou mais particular para efectuar determinado tipo de operação.

Por mais funções que existam nunca será possível que um programador defina funções que sejam auto-suficientes para todos os utilizadores do programa. Deste modo, estes *softwares* ficaríam sempre incompletos se não fosse dada a possibilidade ao utilizador de construir ele próprio as funções que mais lhe convêm.

A programação surge como um meio de cada utilizador modelar o programa às suas necessidades criando aquelas funções que dada a sua especificidade não fazem parte da linguagem original do programa.

Todos os programas põem ao nosso dispor uma linguagem que permite a criação de novas funções através de uma correcta utilização da linguagem própria. Desta forma é possível "modelar" o próprio programa, dando-lhe o cunho pessoal de quem o utiliza e sobretudo permitindo o desenvolvimento de novas aplicações.

A programação assume nestes casos uma importância relevante pois permite que o utilizador se envolva de forma mais profunda com a linguagem utilizada pelo próprio programa e fique a conhecer melhor as suas potencialidades.

Iremos neste capítulo ilustrar a forma de elaborar alguns programas quer no *Derive*, quer no *Maple*, quer no *Mathematica*. Os exemplos aqui apresentados são do conhecimento geral, existindo em muitos dos casos funções predefinidas que já efectuam as operações descritas. No entanto, servem estes como uma ilustração de técnicas e potencialidades que permitem construir novos programas para os que de futuro se podem construir. Não se pretende também que se ponha de lado a utilização das funções predefinidas quando elas existem, pelo contrário, a programação destina-se a aumentar as potencialidades das funções já existentes.

Refira-se ainda que ao construirmos um programa que faça o mesmo que uma função predefinida já existente estamos a criar uma função que certamente não será tão eficiente como aquela que já existe e a eficiência é um objectivo que devemos alcançar. Contudo não é nem será esse o objectivo deste capítulo.

5.1. - EXPRESSÕES BOOLEANAS

Derive:

As expressões de valor verdadeiro ou falso a que damos o nome de expressões booleanas são bastante importantes para conseguirmos realizar algumas tarefas.

Assim a este respeito começamos por afirmar que as constantes booleanas são denotadas por **true** e **false** (verdadeiro e falso, respectivamente).

Para além das constantes existem também alguns conectivos com os quais podemos construir expressões booleanas mais complexas.

Assim a negação de **p** é obtida por **NOT(p)** que é inscrita na zona de trabalho como **¬p**.

```
¬ true
                                                    false

¬ false
                                                    true
```

Como se pode verificar o operador lógico **NOT** é um operador unário, isto é, utiliza apenas um argumento. O mesmo já não se passa com os que analisaremos já de seguida.

A conjunção de duas condições ou expressões booleanas pode ser introduzida através da função **AND**. A forma que surge no *input* é **p∧q** que nos dará **true** se **p** e **q** forem ambas verdadeiras e **false** nos restantes casos.

```
true ∧ false
                                                    false

2 < 3 ∧ 0 < π
                                                    true
```

A disjunção de duas condições cuja função em *Derive* é dada por **OR** dará **false** apenas nos casos em que ambas as condições forem falsas. O *Derive* define ainda a disjunção exclusiva pela função **XOR** que nos fornece **false** nos casos em que ambas as condições são verdadeiras ou falsas e nos dá **true** quando apenas uma das condições é verdadeira.

```
false ∨ true
                                                    true

2.1 > 3.1 ∨ false
                                                    false

false XOR false
                                                    false
```

```
true XOR true
                                                    false
```

A implicação p→q em *Derive* é definida com base na disjunção e negação de condições. Assim, **p IMP q** é o mesmo que **NOT p OR q**. Se **p** for **true** e **q** for **false**, **p IMP q** é **false**. **p IMP q** é **true** em todos os outros casos.

```
true → false
                                                    false

false → true
                                                    true
```

Por último a equivalência de duas condições **p** e **q** é dada por **p IFF q** iniciais das palavras *if and only if* (se e só se) que dá **true** nos casos em que **p** e **q** têm o mesmo valor lógico e **false** nos outros casos.

```
false IFF true
                                                    false

false IFF false
                                                    true
```

A ordem de avaliação dos conectivos booleanos é em primeiro lugar as negações, seguido das conjunções, disjunções e disjunções exclusivas, implicações e equivalências por esta ordem.

Podemos utilizar os parênteses para ultrapassar as precedências referidas anteriormente.

De qualquer forma, **NOT p OR q AND r** é equivalente a **(NOT p) OR (q AND r)**.

Vejamos um pequeno resumo daquilo que acabamos de dizer através de uma outra função que o *Derive* nos disponibiliza.

```
TRUTH_TABLE(p, q, p ∧ q, p ∨ q, p XOR q, p → q, p IFF q)
```

p	q	p ∧ q	p ∨ q	p XOR q	p → q	p IFF q
true	true	true	true	false	true	true
true	false	false	true	true	false	false
false	true	false	true	true	true	false
false	false	false	false	false	true	true

Maple:

Os valores lógicos em *Maple* são fornecidos pelos nomes **true** e **false**. Para aquelas expressões cuja veracidade é desconhecida retornará **FAIL**.

Na base de todas as expressões booleanas estão os operadores lógicos **and**, **or** e **not**. Assim, a implicação, por exemplo é definida à custa da negação e da disjunção.

Avaliemos algumas expressões envolvendo operadores lógicos.

```
> not false;
```

true

```
> not (true);
```

false

```
> true and true;
```

true

```
> true and false;
```

false

```
> true or false;
```

true

```
> 2>3 or 0>1;
```

false

Existem algumas expressões que por vezes pretendemos transformar numa expressão booleana mas, que à partida, o *Maple* não avalia. Fazendo uso da função **evalb(x)** podemos "obrigar" o sistema a fornecer-nos um valor lógico para o seu argumento.

```
> x=x;
```

$x = x$

```
> evalb(x=x);
```

true

```
> evalb(x=y);
```

false

Para simplificar uma expressão booleana podemos, depois de carregar o pacote **logic** fazendo **with(logic);** usar a função **bsimp(b)**.

```
> with(logic);
```

[*bequal, bsimp, canon, convert/MOD2, convert/frominert, convert/toinert, distrib, dual, environ, randbool, satisfy, tautology*]

```
> bsimp(not(not(b)));
```

b

Mathematica:

Aquilo a que já nos referimos no que diz respeito aos conectivos booleanos continua a ser válido. Assim não voltaremos a explicar o comportamento de cada uma das funções aqui analisadas uma vez que isso já foi feito anteriormente no *Derive*. No *Mathematica* alteram-se apenas os nomes, nomeadamente que no diz respeito às iniciais que aqui são maiúsculas, pelos quais as funções são definidas.

Assim a negação, a conjunção, a disjunção e a disjunção exclusiva estão definidas em *Mathematica* nos mesmos moldes que no *Derive*.

A negação é definida pela função **Not** que também pode ser substituída pelo símbolo que é comum utilizarmos para representar a negação. **¬p** representa a negação de **p**.

```
Not[True]
False

¬True
False

¬2 > 3
True
```

A conjunção aplica-se através da função **And** enquanto que a disjunção e a disjunção exclusiva são obtidas pelas funções **Or** e **Xor,** respectivamente.

```
And[True, True]
True

And[True, False]
False

Or[True, False]
True

Or[False, False]
False

Xor[True, True]
False

2 < 3 ∧ 3 < 7
True

True ∨ False
True
```

5.2. - PRELIMINARES

Vamos recordar aqui a definição de funções e referir a expressão condicional **if** *condição* **then** *exp1* **else** *exp2*, construção básica que convém ter presente quando se programa. Esta poderia em português enunciar-se na forma "se... então... caso contrário... "

Na expressão condicional cada um dos programas começa por analisar a condição. Se esta for verdadeira passa para a *exp1*. Caso isso não aconteça salta para a *exp2*. Pode ainda definir-se a expressão condicional sem que se defina a acção a realizar caso a condição não seja verdadeira. Nesse caso quando a condição não é verdadeira nada é feito.

Aproveitamos a oportunidade para referir que no *Mathematica* e no *Derive*, ao contrário do que acontece com o *Maple*, não se utilizam as palavras *then* e *else* em vez disso *condição, exp1* e *exp2* são apresentadas separadas por vírgulas. No *Maple* ao construirmos um **if** dentro de outro **if** pode utilizar-se a expressão **elif** que traz algumas vantagens nomeadamente em termos de visualização e consequente compreensão de cada um dos programas por nós elaborados.

Adiante abordaremos a sintaxe que cada programa utiliza na expressão condicional.

Derive:

Para que se possa dar início à construção de um programa é necessário conhecer algumas regras fundamentais sem o que não será possível cumprir com sucesso os objectivos previstos.

Antes de mais podemos dizer que a base para iniciarmos a programação já foi introduzida quando ilustrámos o modo de definir funções.

É exactamente com base nesse método que vamos elaborar os nossos primeiros programas em *Derive*.

Para iniciarmos vamos verificar como construir um programa que dado o raio calcula o perímetro dessa circunferência.

```
perimetro(r) := 2·π·r
```

Aqui não temos necessidade da mandar avaliar a expressão introduzida. A partir do momento em que carregamos em ENTER para que a expressão seja registada na zona de trabalho, o *Derive* já reconhece a função.

Para determinar o valor do perímetro de uma circunferência de raio 7 temos de mandar avaliar a expressão **perimetro(7)** por meio do atalho **=**.

```
perimetro(7)

                                    14·π
```

Mandando avaliar a mesma expressão carregando em SHIFT+ENTER obtemos um valor aproximado para o perímetro pretendido.

```
perimetro(7)
                                                    43.98229715
```

A função perímetro passa a desfrutar das mesmas características que as funções disponibilizadas pelo próprio sistema.

Como nem todas as situações são tão lineares como o caso anterior surge a necessidade de utilizar algumas construções mais complexas que permitem ultrapassar mais algumas barreiras.

Nesta altura ilustramos apenas uma dessas construções que normalmente se designa por expressão condicional. Introduzimos a expressão **condicional(x):=IF(x>2,x+7,x-1)** que surge no local de trabalho como de seguida apresentamos.

```
condicional(x) :=
  If x > 2
     x + 7
     x - 1
```

Passamos a dispor com a expressão condicional de um processo que permite fazer uma escolha das operações a efectuar mediante a verificação ou não de uma dada condição.

```
condicional(5)
                                                          12

condicional(0)
                                                          -1
```

Quando as situações que estamos a abordar requerem que sejam avaliadas sequências de expressões, o *Derive* utiliza a sintaxe **PROG(***corpo***)**. As regras de cálculo traduzidas em *corpo*, e que lá são introduzidas separadas por vírgulas, são sequencialmente avaliadas retornando a função **PROG** o último valor avaliado.

No exemplo seguinte definimos o programa da seguinte forma: **exp(n):=PROG(n^2, IF(n=2, n+2, 3*n), n^3)**.

```
exp(n) :=
  Prog
    n^2
    If n = 2
       n + 2
       3·n
    n^3

exp(2)
                                                          8

exp(5)
                                                         125
```

Mais adiante e à medida que tal se revelar necessário ilustraremos outras construções que são em muitas situações bastante úteis.

Maple:

Ilustramos aqui a forma de definir uma nova função ou procedimento.

Um procedimento em *Maple* é dado pela sintaxe **proc(***arg***) (***corpo***) end** onde *corpo* serão as regras que pretendemos que o programa execute e *arg* os argumentos do mesmo.

Assim a definição de um programa a que chamamos **exemplo** que dado um inteiro calcula a sua raiz quadrada poderá ser:

```
> exemplo:=proc(x)
> sqrt(x);
> end;
```

$$exemplo := \mathbf{proc}(x)\ \mathrm{sqrt}(x)\ \mathbf{end\ proc}$$

```
> exemplo(4);
```

$$2$$

```
> exemplo(5);
```

$$\sqrt{5}$$

Podemos suprimir o *output* surgido quando mandamos avaliar o *procedure* terminando o mesmo com dois pontos (“**:**”). É ainda possível criar e invocar um procedimento sem que este tenha argumentos.

A expressão condicional é dada pela sintaxe **if** *condição* **then** *exp1* **else** *exp2* **fi**. A aplicação da expressão condicional foi feita em **exemplo2**. Neste caso definimos um **if** dentro de outro **if**. Apesar de não o termos feito quer no *Derive* quer no *Mathematica* tal também pode ser feito como aliás veremos mais à frente e em particular na definição de funções recursivamente (sucessão de Fibonacci).

```
> exemplo2:=proc(n)
> if n<2 then
> n
> else
> if n=2 then
> -3
> else
> -n
> fi
> fi
> end:

> exemplo2(2);
```

$$-3$$

```
> exemplo2(-2.5);
```

$$-2.5$$

```
> exemplo2(6);
```

$$-6$$

Mathematica:

Recordemos que definimos uma função em *Mathematica* através da palavra **Function**. Mais precisamente define-se uma função fazendo **Function[***w***,** *corpo***]** onde *w* é um nome ou uma lista de nomes (os parâmetros da função) e *corpo* é uma expressão (ou sequência de expressões) *Mathematica* que traduz a regra do cálculo da função.

Uma função que dado um número real calcula o seu cubo poderia definir-se assim:

```
cubo = Function[x, x^3]
Function[x, x^3]
```

Poderíamos omitir o *output* surgido colocando ponto e vírgula (“**;**”) no final da definição da função.

Trabalhamos agora com a função criada como se ela fosse predefinida.

```
cubo[1]
1
cubo[3]
27
```

A expressão condicional em *Mathematica* é dada pela sintaxe **If[***condição1***,** *exp1***,** *exp2***]** querendo transmitir que caso se verifique a *condição1* deverá ser efectuada a *exp1* enquanto que se a *condição* não for verdadeira deverá ser efectuada a *exp2*. É uma forma um pouco diferente da já analisada no *Maple* já que omite as palavras *then* e *else* apesar das mesmas se adequarem perfeitamente a este caso.

Para definir a função $f(x)=\begin{cases} x^2, \text{ se } x>0 \\ x^3, \text{ se } x\leq 0 \end{cases}$ faríamos:

```
f = Function[x, If[x > 0, x^2, x^3]]
```

```
f[3]
9
```

```
f[-2]
-8
```

5.3. - PROGRAMAÇÃO RECURSIVA

A programação recursiva consiste em definir uma função à custa dela própria. Tal pode fazer-se através da invocação da função dentro da função.

A construção de uma função recursiva utiliza a expressão condicional na qual têm, obrigatoriamente, de estar bem definidas quer a base quer o passo de recursão.

Enquanto que o passo garante que a função progrida no sentido da conclusão da tarefa pedida, a base representa o culminar de um conjunto de etapas que a função percorreu.

Derive:

Já de seguida exemplificaremos a construção de um programa recursivo em *Derive*.

Definamos um procedimento que calcule o factorial de um dado inteiro positivo.

Como é do nosso conhecimento podemos definir o factorial de um dado inteiro na forma seguinte: n! = n * (n-1)! onde 0! é igual a 1. Estamos claramente a construir a função factorial à custa dela própria. Por essa via definimos da seguinte forma a nossa função: **factorial(n) := PROG(IF(n = 0, 1, n*factorial(n - 1)))** .

```
factorial(n) :=
  Prog
    If n = 0
       1
       n·factorial(n - 1)
```

```
factorial(1)
                                                    1
```

```
factorial(5)
                                                  120
```

Outra situação muito conhecida e tipicamente recursiva é a sucessão de Fibonacci onde cada termo, à excepção dos dois primeiros, é definido como a soma dos dois termos imediatamente anteriores. Uma vez que o *Derive* modifica a expressão por nós introduzida, aqui fica a forma como a obtivemos:

fibonacci(n) :=PROG(IF(n=0,0,IF(n=1, 1, fibonacci(n -1)+fibonacci(n-2)))) .

```
fibonacci(n) :=
  Prog
    If n = 0
       0
       If n = 1
          1
          fibonacci(n - 1) + fibonacci(n - 2)
```

```
fibonacci(0)
                                                    0

fibonacci(1)
                                                    1

fibonacci(8)
                                                   21
```

Nem só sobre números inteiros se faz recursão. Esta pode fazer-se também sobre listas. Vamos construir uma função que nos dá **true** nos casos em que um dado elemento pertence a uma lista e **false** nos outros casos.

Nesta situação utilizamos o facto de que um dado elemento só está numa lista se está na primeira posição ou se está na lista quando dela retiramos o primeiro elemento. A expressão introduzida foi:

Pertence(w, x) := PROG(IF(DIM(w) = 0, false, IF(ELEMENT(w, 1) = x, true, pertence(DELETE_ELEMENT(w, 1), x)))) .

```
pertence(w, x) :=
  Prog
    If DIM(w) = 0
       false
       If ELEMENT(w, 1) = x
          true
          pertence(DELETE_ELEMENT(w, 1), x)

pertence([2, 3, 4, 5], 0)
                                                  false

pertence([2, 3, 4, 5], 2)
                                                  true

pertence([], 5)
                                                  false

pertence([2, 3, 4, 5], 3)
                                                  true
```

Tivemos de usar a função **DIM** uma vez que o *Derive* não disponibiliza a comparação de duas listas. Assim não foi possível saber se a lista era igual à lista vazia. Para contornar essa dificuldade optámos por utilizar o facto do comprimento de uma lista vazia ser zero.

Maple:

Ao construirmos uma função de forma recursiva, estamos a criar uma função cuja definição depende dela própria, isto é, a função é invocada dentro da função. É este o paradigma de programação que é mais simples, não querendo de modo algum isto dizer que seja o mais eficaz.

Vamos definir a função factorial de uma forma recursiva utilizando um raciocínio em tudo semelhante ao já exposto para o *Derive*.

Chamemos à função que iremos definir "fatorial". Repare-se que não poderíamos utilizar a palavra "factorial" uma vez que esse é o nome da função predefinida para o próprio factorial e como tal está protegida.

```
> fatorial:=proc(n)
> if n=0 then
> 1
> else
> n*fatorial(n-1)
> fi
> end:
```

Ilustremos agora a aplicação da função anterior.

```
> fatorial(0);
                                    1
> fatorial(3);
                                    6
> fatorial(7);
                                  5040
```

Por agora não nos preocupamos com o facto de **n** ter forçosamente de ser um inteiro não negativo. Estamos a partir do princípio que o utilizador apenas fornece como argumentos da função elementos dos quais faz sentido calcular o factorial.

A sucessão de Fibonacci é definida por f(0) = 0, f(1) = 1 e f(n) = f (n-1) + f(n-2) para $n \geq 2$.

```
> fibonacci:=proc(n)
> if n=0 then
> 0
> else
> if n=1 then
> 1
> else
> fibonacci(n-1)+fibonacci(n-2)
> fi
> fi
> end:

> fibonacci(3);
                                    2
> fibonacci(7);
                                   13
```

Vejamos agora como definir de forma recursiva a função **pertence** que nos dará **true** caso um dado elemento esteja numa determinada lista.

```
> pertence:=proc(w,x)
> if w=[] then
> false
> else
> if w[1]=x then
> true
> else
> pertence(subsop(1=NULL,w),x);
> fi
> fi
> end:

> pertence([1,2,3,4,5,6],3);
                                                    true

> pertence([1,2,3,4,5,6],7);
                                                    false

> pertence([],2);
                                                    false

> pertence([1,2,3],1);
                                                    true
```

Mathematica:

A programação recursiva pode fazer-se utilizando o *Mathematica* tirando partido de algumas das muitas funções predefinidas que nos ajudam a programar.

Como já dissemos anteriormente ao programar de uma forma recursiva estamos a definir uma determinada função à custa dela própria.

Definamos de forma recursiva a função factorial.

```
factorial = Function [n, If [n == 0, 1, n* factorial [n - 1]]]
General::spell1 :
 Possible spelling error: new symbol name "factorial" is
   similar to existing symbol "Factorial".

factorial [2]
2

factorial [6]
720
```

Note-se que a utilização de "= =" em **n = = 0** refere-se ao teste de igualdade.

A sucessão de Fibonacci pode também ser definida de forma recursiva.

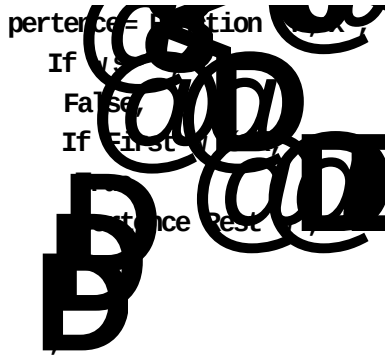
```
1
```

```
21
```

Na anterior função tivemos de recorrer à composição de dois **If**'s, ou seja, um **If** dentro de outro **If** à semelhança do que já havíamos feito atrás.

Sendo o domínio das duas funções anteriores os inteiros não negativos podemos passar a uma situação envolvendo o tipo lista.

Definamos então mais uma vez a função **pertence** que recebendo uma lista e um elemento nos dirá se esse elemento consta da lista.



```
True
```

```
False
```

5.4. - PROGRAMAÇÃO IMPERATIVA

Sem dúvida que é a programação imperativa a forma mais comum de programar. Esta caracteriza-se por efectuar-se através de uma alteração de um estado inicial até um estado final através da passagem por um conjunto de estados intermédios que possibilitam a aproximação ao objectivo final.

Ciclos:

Neste tipo de programação utilizam-se as atribuições, as expressões condicionais e os ciclos para o que se recorre às construções **While, For** e **Loop**.

Em poucas palavras podemos dizer que os ciclos permitem que se efectuem um determinado número de vezes um conjunto de operações. A "chave" de um ciclo é repetir, repetir, repetir, ...

Vamos neste momento analisar algumas das diferenças existentes entre os três programas no que diz respeito aos ciclos existentes para depois passarmos à sua aplicação concreta.

Um **Loop** é constituído por um conjunto de comandos que constituem a acção que pretendemos realizada. Para que este termine temos de completá-lo com uma expressão do tipo **Return(n)** ou **Exit**. Nos casos que vamos analisar sentimos necessidade de introduzir a função **RETURN** num **IF**. Assim a condição do **IF** funciona com o mesmo papel que desempenha a guarda de um **While**. Aqui, enquanto que essa condição for falsa a sequência de comandos é efectuada sucessivamente até que a condição passe a ser verdadeira, altura em que é aplicado o comando **Return** o que conclui o programa.

Quanto ao **While** este é constituído por duas partes, a guarda e os comandos a executar. Enquanto que a condição que define a guarda for verdadeira o programa deverá efectuar os comandos. Quando a guarda for falsa então o **While** terminará. A diferença fundamental entre um **While** e um **Loop** é que este último tem necessidade de recorrer a uma outra construção auxiliar – um **If –** para efectuar o seu trabalho enquanto que um **While** é "independente" de qualquer outra construção.

No respeitante ao **For**, o que acontece é a existência de uma variável que percorre um determinado conjunto de valores. O **For** tem início no primeiro elemento desse conjunto de valores e termina quando é atingido o último valor desse conjunto referido. Veremos já de seguida as diferenças sintácticas que cada uma destas construções apresenta.

Temos de garantir que ao definirmos um programa imperativo este terá que terminar principalmente quando estamos a construir ciclos. Se isso não acontecer o programa não chegará ao fim.

Variáveis Locais e Globais:

As variáveis que se encontram num programa podem ser de dois tipos: globais ou locais.

As variáveis globais são aquelas que estão disponíveis quer dentro dos programas quer fora destes.

As variáveis locais são nomes ou expressões que utilizamos dentro de uma função ou programa. A grande vantagem é que a alteração do valor de uma variável local não tem consequências fora da função ou programa onde está definida.

Em *Mathematica* usamos a expressão **Module[{a, b},corpo]** para definirmos as variáveis locais, neste caso **a** e **b**. Em *Maple* utilizamos **local a b** para tornar explícita a mesma situação. De referir ainda que no caso do *Maple*, se não for indicado explicitamente se uma variável é local ou global, o programa decide se uma dada variável é local ou global emitindo uma mensagem informando da decisão. Se o *Maple* informar que **c** é local podemos impor que a mesma seja global escrevendo **global c** tal como faríamos para as variáveis locais. No *Mathematica* todas as variáveis que não são declaradas dentro de um *Module* são globais.

Derive:

A programação imperativa faz-se em *Derive* recorrendo à construção **LOOP**. Nesta função as expressões que constituem os argumentos são repetidas continuamente até que surja um comando de **EXIT** ou **RETURN**, comandos estes que terminam com o **LOOP**. Quando surge a função **RETURN(n)** imediatamente a execução do programa em questão termina, retornando o programa nesse caso o argumento **n**. A função **EXIT**, quando surge num **LOOP** ou **PROG**, faz com que a execução do mesmo termine e a avaliação continue com a execução da próxima expressão. Ou seja, a diferença fundamental é que **RETURN** faz com que a função por nós definida termine imediatamente, enquanto que **EXIT** faz com que acabe apenas o **PROG** ou **LOOP** no qual se encontra inserida a função passando à avaliação da expressão seguinte.

Vejamos como definimos a função **factorial** de um dado inteiro não negativo recorrendo a um programa de tipo imperativo. Em primeiro lugar fica aqui registado a forma com a função foi introduzida:

fact(n) := PROG(i := 1, r := 1, LOOP(IF(i > n, RETURN r), r := r * i, i := i + 1)) .

```
fact(n) :=
  Prog
    i := 1
    r := 1
    Loop
      If i > n
        RETURN r
      r := r·i
      i := i + 1
```

```
fact(0)
```

1

```
fact(3)
```

6

A função **pertence** também pode ser definida usando um **LOOP**. Para isso iremos utilizar a função **DIM** que fornece-nos o comprimento da lista e a função **ELEMENT** que dá-nos o elemento que se encontra numa certa posição de uma lista. A função foi introduzida como se segue:

pertence(w, x) := PROG(b := false, i := 1, LOOP(IF(b = true ∨ i > DIM(w), RETURN b, IF(ELEMENT(w, i) = x, b := true, i := i + 1)))) .

```
pertence(w, x) :=
  Prog
    b := false
    i := 1
    Loop
      If b = true ∨ i > DIM(w)
        RETURN b
        If ELEMENT(w, i) = x
          b := true
          i := i + 1
```

```
pertence([3, 7, -1, 6], 3)
                                                    true

pertence([], 2)
                                                    false

pertence([3, 7, -1, 6], 5)
                                                    false

pertence([3, 7, -1, 6], -1)
                                                    true
```

Maple:

No *Maple* veremos dois tipos de ciclos um **while** e um **for**. A primeira construção faz-se recorrendo à sintaxe **while** *guarda* **do** *expr* **od;** . Enquanto que a **guarda** for verdadeira as expressões de **expr** são sempre avaliadas.

O outro tipo de ciclo corresponde à construção **for** *i* **from** *n* **to** *m* **do** *expressões* **od;**.

Iniciando-se em **n**, o conjunto de expressões é avaliado, até que a variável **i** atinja o valor **m**. Vejamos como definir a função factorial usando um **while**.

```
> fact:=proc(n)
> local i, r;
> r:=1;
> i:=1;
> while i<=n do
> r:=r*i;
> i:=i+1
> od;
> r
> end:
```

```
> fact(0);
```

1

```
> fact(5);
```

120

Da mesma forma poderíamos definir a função factorial usando um **for**, ganhando-se simplicidade.

```
> fact1:=proc(n)
> local i, r;
> r:=1;
> for i from 1 to n
   do
> r:=r*i
> od;
> r
> end:
```

```
> fact1(3);
```

6

A função **pertence** pode definir-se recorrendo à programação imperativa. Para tal basta criar uma variável **i** que percorra todas as posições da lista.

```
> pertence:=proc(w,x)
> local i, n, b ;
> i:=1;
> n:=nops(w);
> b:=false;
> while i<=n and b=false do
> if w[i]=x then
> b:=true
> fi;
> i:=i+1
> od;
> b
> end:
```

```
> pertence([1,2,3,4,5],1);
```

true

```
> pertence([1,2,3,4,5],7);
```

false

```
> pertence([1,2,3,4,5],3);
```

true

Este tipo de progresso que é feito de elemento para elemento de uma lista é tão comum que o *Maple* dispõe de um ciclo especial para tratar destes casos.

```
> pertence1:=proc(w,x)
> local i;
> for i in w do
> if x=i then RETURN(true)
> fi;
> od;
> false;
> end:

> pertence1([1,2,3,4,5],7);
```

false

```
> pertence1([1,2,3,4,5],1);
```

true

Se nada for dito em contrário um *procedure* retorna o último valor calculado. Usando a função **RETURN** deixa de ser assim. O que surge como *output* do programa é o argumento desta função.

Mathematica:

Em *Mathematica* tal como acontece no *Maple* definimos ciclos recorrendo às funções **While** e **For** que diferem entre si pela forma como são escritas, mais precisamente pelo facto de em *Mathematica* a primeira letra ser maiúscula.

Tratando-se de um **While** a sintaxe é **While[***condição,expressão1***]**. Enquanto que a **condição** for verdadeira o programa executará a expressão ou conjunto de expressões que designamos por **expressão1**.

Uma outra forma de definir um ciclo poderá ser através da função **For**.

```
? For
For start, test, incr, body
  executes start, then repeatedly evaluates
  body and incr until test fails to give True.
```

Portanto, iniciando no valor correspondente ao primeiro argumento enquanto que a condição presente no seu segundo argumento for verdadeira o *Mathematica* executará a acção indicada pelo último argumento avançando a variável de acordo com a forma definida pelo terceiro argumento.

```
For i = 1, i < 4, i = i + 1, Print i
```
```
1
2
3
```

Usamos propositadamente o comando **Print** pois se não o fizéssemos apenas nos seria possível constatar o resultado final.

```
fact = Function n, Module
    i = 1;
    r = 1;
    While i ≤ n,
        r = r * i;
        i = i + 1
    ,
    r
]
```

```
fact 0
```
```
1
```

```
fact 4
```
```
24
```

Usando um **For** a função **fact** surge um pouco mais condensada, contendo todos os elementos presentes no **While** anterior. Neste caso não nos parece que ganhemos muito optando por esta função ao invés da anterior.

```
fact1 = Function n, Module
    r = 1
    For i = 1, i ≤ n, i = i + 1, r = r * i
]
```

```
fact1 4
```
```
24
```

```
fact1 2
```
```
2
```

A construção **For** é mais versátil que no *Maple* uma vez que permite que definamos através de uma condição a regra de paragem. Quer isto dizer que a condição não tem necessariamente de ser do tipo **i ≤ n**. Poderá ser por exemplo **b = = False** o que faz com que cresça imenso o nosso leque de escolhas para a condição tornando mais simples o nosso trabalho.

Por último vejamos novamente a já nossa conhecida função **pertence** definida agora de forma imperativa.

```
pertence = Function[x, Module[{n, i, b},
    n = Length[x];
    i = 1;
    b = False;
    While[i <= n && b == False,
     If[x[[i]] == x[[n]], b = True];
     i = i + 1
    ];
    b]]
```

```
pertence[{1, 2, 3, 4, 5, 6}, 1]
```
True

```
pertence[{1, 2, 3, 4, 5, 6}, 6]
```
True

```
pertence[{1, 2, 3, 4, 5, 6}, 8]
```
False

```
pertence[{}, 1]
```
False

5.5. - PROGRAMAÇÃO FUNCIONAL

Na programação funcional a função desejada é definida recorrendo à eventual definição de outras funções por abstracção funcional e à aplicação dessas funções e de outras funções predefinidas, eventualmente sobre listas ou mesmo outras funções.

Ao programarmos funcionalmente não utilizaremos a composição sequencial, comandos iterativos **While** e a memorização de valores em variáveis através de atribuições. As funções são utilizadas como argumento de outras funções (funcionais ou operadores). Para o conseguirmos recorremos aos funcionais **Apply** e **Map**.

Nesta secção optamos por nos referirmos em primeiro lugar ao *Mathematica* uma vez que consideramos que este apresenta melhores argumentos para abordar a programação funcional.

Mathematica:

O funcional **Apply** substitui a cabeça da expressão apresentada no segundo argumento pela função que consta no primeiro. Esta substituição é feita desde que a expressão seja não atómica.

Vejamos um exemplo da aplicação do funcional **Apply**.

Recorde-se que a lista **{1,2,3,4}** é representada em *Mathematica* por **List[1,2,3,4]**.

```
FullForm [ , 3, 4 ]
List [ , 2, 3, 4 ]
```

Ao pretender obter a soma dos elementos que compõem a lista bastar-nos-ia substituir a cabeça **List** por **Plus**. Conseguimos fazê-lo facilmente aplicando o funcional **Apply**.

```
Apply [ , , 3, 4 ]
10
```

Ao pretender obter a definição de uma função que nos dê o factorial de um dado número inteiro não negativo teremos de utilizar um raciocínio semelhante ao anterior utilizando a multiplicação que em *Mathematica* se designa por **Times**.

Para calcular **n!** temos de construir uma lista com os números 1, 2, 3, ..., n. Ora podemos consegui-la utilizando a função **Table**.

```
Table [ , , ]
, 3, 4
```

Assim somos levados à seguinte definição de **fact**.

```
fact = Function [ , Apply [ , Table [ , , ] ] ]

fact [ ]
1
```

```
fact 5
120
```

Quanto ao funcional **Map** este faz com que uma dada função seja aplicada a todos os elementos de uma lista.

```
Map [redacted], 11
[redacted]
```

Posto isto podemos passar à definição funcional da função **pertence**.

A ideia subjacente a esta função é percorrer a lista e comparar um a um todos os elementos da lista com o elemento que queremos encontrar. Temos portanto de definir uma função que compare cada elemento da lista com o elemento que queremos pesquisar.

Depois obteremos uma lista composta por elementos que serão **True** ou **False**. A função **pertence** deverá dar **True** nos casos em que existe pelo menos um **True** na lista anterior. Daqui surge como imediata a aplicação da função **Or**.

Assim a definição para a função **pertence** poderá ser a seguinte.

```
pertence(Function [redacted]
  Apply(Or, Map(Function(y, y=x), [redacted]))

pertence([redacted],3,4,3)
True

pertence([redacted],3,4,0)
False

pertence([redacted],3,4,4)
True

pertence([redacted],3,4,7)
False
```

Derive:

No *Derive* podemos utilizar a função **MAP_LIST** para distribuir a aplicação de uma função pelos elementos de listas e conjuntos.

Nos casos em que **u** é uma expressão envolvendo a variável **x** e **c** é um conjunto ou uma lista, **MAP_LIST(u,x,c)** avalia **u(x)** quando **x** é igual a cada um dos elementos de **c** e retorna um conjunto ou uma lista.

Podemos ainda aplicar a função **MAP_LIST** com mais do que três argumentos. Nesses casos é possível indicar onde deve começar a aplicação da função, onde deve terminar e qual o tamanho dos passos a dar.

Na nossa definição de factorial basta-nos aplicar a definição de **n!** como sendo o produto dos números inteiros positivos menores ou iguais ao **n**. Como já vimos anteriormente podemos obter o produto de um conjunto de números através da função

PRODUCT. Tendo em conta esse facto podemos definir a função factorial como **fact(n) :=PRODUCT(x, x, 1, n)**. Carregando em ENTER a nossa função fica registada na nossa zona de trabalho como se segue.

$$fact(n) := \prod_{x=1}^{n} x$$

fact(0)

1

fact(2)

2

fact(4)

24

Na definição da função pertence já sentimos a necessidade de utilizar a função **MAP_LIST** à qual já fizemos referência. Neste caso particular a função anterior tem por objectivo verificar se alguma das entradas da lista coincide com o elemento introduzido. Na definição da função **pertence** criamos uma função auxiliar de nome **ou_lista** que permite o cálculo da disjunção dos elementos de uma lista composta pelas constantes booleanas **true** ou **false**. Tal foi feito mediante a utilização de uma expressão condicional em que nos casos em que o elemento da lista era **true** era transformado em **1** e **0** nos restantes casos. Depois, somando os valores, e tendo em conta que a disjunção de condições é verdadeira desde que exista pelo menos uma das condições que seja verdadeira, bastou impor que caso a soma fosse maior que **0** a função **ou_lista** desse **true**. A definição para a função encontrada foi a seguinte:

ou_lista(w) := IF(Σ (IF (y, 1, 0), y, w) > 0, true, false).

```
ou_lista(w) :=
  If Σ(IF(y, 1, 0), y, w) > 0
     true
     false
```

ou_lista([false, true, false])

true

ou_lista([false, false, false])

false

ou_lista([true, true, true])

true

Para concluirmos a definição da nossa função temos de garantir que o argumento que é fornecido à função anterior lá chega nas condições pretendidas. Conseguimos isso pela comparação de cada entrada da lista original com o elemento do qual pretendemos saber se está ou não na lista para o que utilizamos o funcional **MAP_LIST**.

```
pertence(w, x) := ou_lista(MAP_LIST(y = x, y, w))
```

```
pertence([2, 7, -5, 4], 2)
                                                    true

pertence([2, 7, -5, 4], 3)
                                                    false

pertence([2, 7, -5, 4], 4)
                                                    true

pertence([2, 7, -5, 4], -5)
                                                    true

pertence([2, 7, -5, 4], -3)
                                                    false

pertence([], -5)
                                                    false
```

Maple:

Para conseguir definir funcionalmente a função factorial sentimos necessidade de definir localmente uma função a que chamamos **sequencia** cuja finalidade é a de simplesmente nos gerar o conjunto vazio quando o argumento da nossa função for zero. Isto fica a dever-se ao facto de **seq(j, j =1 .. 0)** não ser avaliado.

Caso não nos interessasse o caso do factorial de zero a nossa definição não necessitaria da função **sequencia**.

```
> fact:=proc(n)
> local sequencia,i;
> sequencia:=proc(m)
> local j;
> if m>0 then
> {seq(j,j=1..m)}
> else
> {}
> fi
> end;
> mul(i,i=sequencia(n))
> end:
```

Vejamos que realmente nestas condições a função está de acordo com o que pretendíamos.

```
> fact(0);
                                    1

> fact(3);
                                    6

> fact(5);
                                   120
```

Para conseguir definir funcionalmente a função **pertence** seguimos aqui um raciocínio semelhante ao utilizado no *Derive*. Tal como lá, utilizamos uma função **f** que permite a comparação de cada elemento da lista com aquele que pretendemos saber se faz parte desta. Ao invocarmos a função **pertence(w,x)** onde **w** é a lista e **x** é o elemento, caso a função **f** encontre em **w** algum elemento que coincida com **x** este é transformado em **1** e nos restantes casos é alterado para **0**. Posteriormente esses uns e zeros são adicionados permitindo que se obtenha **true** nos casos em que existia um (ou mais) elemento(s) iguais a **x**. Este é um método muito parecido com aquele que usamos para o *Derive*.

```
> pertence:=proc(w,x)
> local f,i;
> f:=(n,x)->if n=x then 1
> else
> 0
> fi;
> if add(i,i=map(f,w,x))=0 then false
> else
> true
> fi
> end:

> pertence([1,2,3,4],3);
                                  true

> pertence([1,2,3,4],1);
                                  true

> pertence([1,2,3,4],4);
                                  true

> pertence([1,2,3,4],5);
                                  false

> pertence([],2);
                                  false
```

5.6. - PROGRAMAÇÃO POR REGRAS DE REESCRITA

A ideia subjacente à programação por regras de reescrita é a de substituir uma expressão por outra. Quer no *Maple* quer no *Derive* não nos foi possível encontrar uma forma de definir programas através deste paradigma. No *Maple* está disponível a função **type** que nos permite controlar o tipo de argumentos que um procedimento aí definido pode receber. Uma vez que esse assunto, no que ao *Mathematica* diz respeito, se encontra inserido nesta secção achamos oportuno referir aqui esse aspecto no *Maple*, chamando a atenção para o facto de que isso não é programação por regras de reescrita.

Mathematica:

A definição de funções por atribuição paramétrica diferida é uma forma alternativa à definição de funções por abstracção funcional.

O paradigma da reescrita baseia-se na existência de um conjunto de expressões e numa interpretação que é feita destas sendo elas reescritas por aplicação das regras de reescrita. O *Mathematica* dispõe de uma base de regras de reescrita disponíveis, a qual é formada pelas regras predefinidas que estabelecem o modo como são avaliadas as funções predefinidas no *Mathematica*, e pelas regras de reescrita introduzidas pelo utilizador na sessão em causa (e ainda não apagadas).

Ao criarmos uma regra **substituir[le,ld]** iremos substituir as ocorrências da expressão **le** pela expressão **ld**. A avaliação de uma expressão consiste em aplicar-lhe o conjunto de regras de reescrita que a ela dizem respeito.

Antes de continuarmos convém esclarecer o conceito de molde universal. O molde universal é uma "estrutura" a que todas as expressões *Mathematica* se adequam. Assim ao escrevermos **f[_] := 4** estamos a definir uma função **f** que, seja qual for o argumento, nos fornecerá 4.

```
f[2]
f[x]
4
f[Sin[x]]
4
```

Os moldes são utilizados não só para nos referirmos a todas as expressões *Mathematica* mas também de um modo mais restrito a algumas classes de expressões, aquilo a que se dá o nome de forma de expressão.

Exemplificando aquilo a que nos referimos podemos afirmar que à forma de expressão _^_ se ajustarão todas as expressões que sejam escritas como potência de duas expressões *Mathematica*. A forma de expressão ajusta-se a **x^y** e a **x^3**. Note-se no entanto, que a forma anterior não se ajusta a **3^5** em virtude de o *Mathematica* simplificar em primeiro lugar a expressão ficando **243** que não é a potência de duas expressões mas apenas um *Integer* (inteiro).

O predicado **MatchQ** poderá ajudar-nos a compreender o funcionamento das formas de expressão.

```
MatchQ[x^3, _^_]
True

MatchQ[x^3, _^3]
True

MatchQ[2^3, _^_]
False
```

Aquilo que conhecemos neste momento não é ainda suficiente para podermos utilizar as regras de reescrita para definirmos a nossa função factorial.

Uma primeira ideia seria definir a função **fact** do seguinte modo:

```
fact[n_] := n!
```

No entanto vejamos o que acontece se tentarmos calcular **fact[3]**.

```
fact[3]
n!
```

O *Mathematica* até ao momento não sabe, porque nada lhe foi dito nesse sentido, que o **_** é o **n**. Para conseguir essa identificação é necessário utilizar as etiquetas.

Para tal usamos a construção **etiqueta molde** ou **etiqueta: formaExpressão**. Ao escrevermos **f[x_]:=x^2** estamos a dizer que o argumento da função **f** deve ser substituído pelo valor do seu quadrado.

Utilizando a função **Clear[f]**, anulam-se todas as regras de reescrita existentes associadas a **f**.

```
Clear[fact]
```

Portanto uma possível solução para a definição do factorial poderá ser a seguinte.

```
fact[n_] := If[n == 0, 1, n * fact[n - 1]]

fact[3]
6
```

Podemos definir várias regras de reescrita para uma mesma função sendo essas regras aplicadas de acordo com algumas prioridades. São elas:

1º São aplicadas as regras definidas pelo utilizador e só depois as predefinidas.

2º Uma regra mais específica (isto é, se se ajusta a menos expressões) tem prioridade sobre uma regra mais geral.

3º Em caso de igual "especificidade" as regras são aplicadas pela ordem com que foram introduzidas.

Ao utilizarmos **?f** o sistema dá-nos toda a informação que dispõe sobre a função **f** e pela sua ordem de especificidade.

Introduzimos as seguintes regras:

```
f[x_^3]

f[x_]

f[x_^2]
```

e mandamos avaliar uma expressão. Vejamos como o programa se comportou.

```
f[2]
8
```

Como pudemos verificar o *Mathematica* aplicou a segunda regra já que é a regra menos específica que encontra em primeiro lugar. Repare-se que o facto de o sistema não ter aplicado a primeira das regras introduzidas fica a dever-se ao facto de **x** não se ajustar a **2** que foi a expressão introduzida.

```
MatchQ[2,x^3]
False

MatchQ[2,x_]
True

MatchQ[2,2]
True

?f
Global`f
f[x_^3]

f[x_]

f[x_^2]
```

Utilizando os critérios de aplicação das regras de reescrita associadas a uma dada função podemos definir ainda a função factorial como se segue.

```
Clear[fact]
fact[0]=1
fact[n_]:=n fact[n-1]

fact[1]
1

fact[5]
120
```

Uma das vantagens que a definição de uma função por atribuição paramétrica diferida tem em relação à definição por abstracção funcional é o facto desta permitir que definamos o tipo de argumentos que uma determinada função deverá receber.

No caso da função factorial quando a definimos estamos obviamente a pensar que o utilizador irá fornecer-lhe como argumento apenas números inteiros não negativos. No entanto, neste momento nada impede que tentemos calcular o factorial de um número negativo. O *Mathematica* nesse caso irá dar um erro.

```
fact[-5]
$RecursionLimit::reclim : Recursion depth of 256 exceeded.
-6138134392365655825508810874366172877050571718444065641332212848526685056306730021611034997236265663323354781024509631575854710487579525737339335755352323118836222724997168027188257408365700078302615464069748900163986639038663033000884545868629346915288283849530353075839810397946448096421006946736742334516591518116735107375418958758787476292270575496920123238242671828216186744175975288686515121212
```

Assim, para evitar estas situações é possível estabelecer à partida que a função factorial só deverá ser avaliada quando o argumento for um inteiro não negativo. Para consegui-lo precisamos de moldes mais restritos que o molde universal. Os moldes **_Real**, **_Integer** e **_List** são alguns exemplos desses moldes.

```
MatchQ[3, _Integer]
True

MatchQ[3, _Real]
False
```

O *Mathematica* não se fica por aqui no que diz respeito ao condicionamento das formas dos argumentos. Assim é possível acrescentar um predicado a uma forma a construções do tipo **formaExpressão?predicado**.

Nestes casos esta construção só se ajusta a uma expressão **exp** se a forma se ajustar a ela e o resultado de aplicar o predicado a **exp** retornar **True**.

Podemos complementar uma condição com uma forma através de construções do tipo: **formaExpressão/;condição**.

A condição normalmente envolve as etiquetas e tem de ser satisfeita pelas partes às quais se referem as etiquetas.

Usando este processo podemos definir em primeiro lugar um predicado a que chamamos **naturalQ0** que nos dará **True** nos casos em que o argumento seja um inteiro positivo ou o zero.

```
naturalQ0[n_Integer /; n >= 0] := True
naturalQ0[_] := False
```

Portanto a expressão **n** tem de ser um inteiro e ser maior ou igual a zero.

Exemplifiquemos o comportamento desta função.

```
naturalQ0[5]
```
```
True
```

```
naturalQ0[0]
```
```
True
```

```
naturalQ0[-3]
```
```
False
```

```
naturalQ0[1/5]
```
```
False
```

Então a função **fact** pode ser definida utilizando o predicado anterior.

```
fact1[n_/; naturalQ0[n]] := fact1[n-1]
fact1[0] = 1
```

```
fact1[6]
```
```
720
```

```
fact1[-5]
```
```
fact1[-5]
```

```
fact1[2.]
```
```
fact1[2.]
```

Uma vez que os moldes que usamos para definir a função **fact1** não se ajustam aos argumentos introduzidos o *Mathematica* deixa a expressão por avaliar.

Poderíamos definir factorial como se segue.

```
fact2[n_/; IntegerQ[n] && n ≥ 0] := n fact2[n-1]
fact2[0] = 1
```

```
fact2[2]
```
```
2
```

```
fact2[-3]
```
```
fact2[-3]
```

```
fact2[1/2]
```
```
1
```

Quanto à função **pertence** pode aplicar-se um raciocínio inteiramente semelhante.

```
pertence[x_, {}] := False;
pertence[x_, list_List] := x == First[list] || pertence[x, Rest[list]]
```

```
pertence[1, {2, 3, 4}]
```
```
False
```

```
pertence [illegible]
pertence [illegible]

pertence [illegible] 5 , 2.]
True

pertence [illegible] 2., 5, 6 , 2.]
True

pertence [illegible] , 2.]
False

pertence [illegible]
pertence [illegible]
```

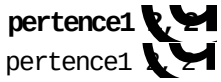
Se a lista em questão tivesse de ser de inteiros como seria?

Para nos referirmos a sequências de expressões temos os moldes **_ _**, **_ _h** que representam os moldes que são satisfeitos por uma sequência ou mais de expressões, sendo que no segundo caso estas têm de ter cabeça **h**. Os moldes **_ _ _** e **_ _ _h** são aqueles que são satisfeitos por sequências de zero ou mais expressões sendo que no último caso as expressões têm de ter cabeça **h**.

Vamos aplicar os moldes de sequências de expressões para definirmos a função **pertence1** que recebendo uma lista de reais e um real nos dirá se o elemento está na lista.

```
pertence1 [illegible]
 pertence1 [illegible] x Real [illegible] First [illegible] True, pertence1 Rest [illegible]
pertence1 [illegible] , [illegible] elemento não é real"
pertence1 _ _Real [illegible] A lista não é lista de reais"
```

```
pertence1 [illegible] 2.]
False

pertence1 [illegible]
O elemento não é real

pertence1 [illegible] , 2.]
A lista não é lista de reais

pertence1 [illegible] 2., 3., 4., 5. , 3.]
True

pertence1 [illegible] 2., 3., 4., 5. , 5.]
True

pertence1 [illegible] 2., 3., 4., 5. , 1.]
True

pertence1 [illegible] 2., 3., 4., 5. , 7.]
False
```



Maple:

Os procedimentos que temos vindo a construir podem ser modificados de modo a que o seu domínio seja alterado. Modificando a parte inicial dos procedimentos é possível fazer com que estes apenas possam receber como argumentos os objectos que o construtor do programa pensou. Evitam-se assim alguns possíveis erros que surgem com alguma naturalidade e frequência quando, por exemplo, o utilizador introduz um nome onde era suposto introduzir um número inteiro.

O *Maple* disponibiliza-nos alguns tipos aos quais podemos ter acesso se fizermos **type?**.

Salientamos aqui alguns desses tipos: *****, ^, **array**, **list**, **positive**, **set**, **table**, **boolean**, **integer**, **nonnegint**, **negint**, **NONNEGATIVE**, **posint**, **negative**, **prime**, **matrix** e **anything**.

```
> type(a+b,`+`);
                                true

> type(-3,nonnegint);
                                false

> type([1,2,3,4,4],list);
                                true

> type(2+2,integer);
                                true
```

A função factorial pode ser modificada por forma a que receba como argumento apenas números inteiros não negativos. Já acima vimos que no *Maple* existe o predicado **nonnegint** que nos permite identificar imediatamente se o argumento está nas condições em que ao elaborarmos a função estávamos a pensar. Uma outra forma de nos referirmos ao tipo das expressões é escrevermos **exp::tipo**. Utilizamos essa forma para definirmos a função **fact**.

```
> fact:=proc(n::nonnegint)
> if n=0 then
> 1 else
> n*fact(n-1)
> fi
> end:

> fact(3);
                                6
```

```
> fact(3.5);
Error, fact expects its 1st argument, n, to be of type
nonnegint, but received 3.5

> fact(-5);
Error, fact expects its 1st argument, n, to be of type
nonnegint, but received -5
```

Como podemos constatar, o programa não efectuou os nossos dois últimos pedidos já que conseguiu detectar que o argumento que introduzimos não é do tipo pretendido.

Na função **pertence** podemos fazer algo de semelhante admitindo que o elemento que procuramos é um inteiro. Aqui o que fizemos foi seguir o programa que já tínhamos elaborado na programação imperativa e definir o tipo dos argumentos que a função deve receber.

```
> pertence:=proc(w::list,x::integer)
> local i, n, b ;
> i:=1;
> n:=nops(w);
> b:=false;
> while i<=n and b=false do
> if w[i]=x then
> b:=true
> fi;
> i:=i+1
> od;
> b
> end:

> pertence([1,2,3,4],5);
                          false

> pertence([1,2,3,4],2);
                           true

> pertence([1,2,3,4],5.3);
Error, pertence expects its 2nd argument, x, to be of type
integer, but received 5.3

> pertence({1,2,3,4},5);
Error, pertence expects its 1st argument, w, to be of type
list, but received {1, 2, 3, 4}
```

Como se pode verificar a nossa função é agora mais selectiva em relação ao tipo de argumento que recebe.

5.7. – CONCLUSÃO

As constantes booleanas **true** e **false** estão presentes em todos os programas bem como os conectivos booleanos sendo que no *Maple* não se encontra definida a disjunção exclusiva. De referir também que no *Derive* através da função **TRUTH_TABLE** podemos construir as tabelas de verdade com muita facilidade.

A programação recursiva faz-se de forma muito semelhante nos programas em análise uma vez que todos possuem a expressão condicional. Não existem aqui grandes diferenças a realçar.

Na programação imperativa começamos a notar algumas diferenças, nomeadamente em relação às construções disponibilizadas para efectuar ciclos. O *Derive* possui a função **Loop** enquanto que o *Maple* e o *Mathematica* permitem a utilização das funções **While** e **For**. Aqui parece-nos que tanto um **While** como um **For** apresentam vantagem em relação à construção **Loop**, isto porque a mesma precisa para se poder efectuar de outras construções auxiliares, nomeadamente a expressão condicional. A construção **For** em *Mathematica* é mais versátil já que, enquanto que em *Maple* a construção se resume a percorrer através de uma variável de progresso um conjunto de valores desde um valor inicial até um valor final, aqui (em *Mathematica*) podemos directamente definir o comprimento do passo bem como definir uma guarda que não tem forçosamente que ser o atingir de um valor final para uma variável do progresso.

De notar também que a construção **For** em *Maple* apresenta uma variante que se revela útil quando estamos a trabalhar com listas permitindo atravessar a lista de forma mais simples.

Na programação funcional deparamo-nos com algumas dificuldades para conseguir definir a função pertence no *Derive* e no *Maple* devido ao facto de lá não podermos aplicar a disjunção às entradas de uma lista.

Para além da função **Map**, que está definida em todos os programas (no *Derive* **MAP_List**), no *Mathematica* a função **Apply** revela-se de grande utilidade pois possibilita que se apliquem operações a listas de uma forma eficiente e original.

Por último na programação por regras de reescrita não há muito para comparar em virtude de apenas no *Mathematica* podermos falar deste tipo de programação.

Podemos afirmar que a mesma abre novos horizontes e permite que se seleccionem os argumentos que as nossas funções recebem. Este último aspecto está disponível no *Maple*. Também aqui é possível impor que um procedimento apenas receba determinado tipo de parâmetros.

Em resumo:

- No *Derive* a obtenção de tabelas de verdade é bastante simples.
- A programação recursiva é idêntica em todos os programas.
- A programação imperativa é mais eficaz no *Maple* e no *Mathematica*. No *Derive* a função **Loop** não tem argumentos suficientes para igualar as correspondentes funções dos restantes programas.
- A programação funcional é mais imaginativa e eficiente no *Mathematica* podendo utilizar-se a função **Apply** de modo a que certas funções se apliquem directamente a listas.
- A programação por regras de reescrita só é possível no *Mathematica*.

Ao nível da programação as diferenças fundamentais encontram-se na programação imperativa, na programação funcional e na programação por regras de reescrita. O *Mathematica* apresenta-se mais bem equipado que os restantes programas. Isto é evidente sobretudo nos dois últimos paradigmas.

6. - GRÁFICOS

A representação gráfica de uma função permite que em poucos segundos tenhamos uma ideia geral sobre o comportamento dessa função. Sem mais esforço é possível descortinar imediatamente muita informação só pelo simples facto de olhar para a representação gráfica de uma função.

De facto, muito cedo aprendemos a representar com lápis e papel algumas funções o que nos permitia descobrir algumas particularidades de certas famílias de funções. O que veremos nesta abordagem começará exactamente por uma passagem rápida pelos aspectos básicos da elaboração de uma representação gráfica. Depois, à medida que formos avançando vamos exemplificar a mais valia que cada um dos programas acarreta.

Só por si, a rapidez com que conseguimos uma representação gráfica permite que estudemos um grande número de funções em poucos minutos, ultrapassando dessa forma os cálculos indispensáveis para um estudo desta natureza realizado “à mão”.

Ao passarmos para gráficos a três dimensões poderemos visualizar superfícies que doutra forma seriam quase impossíveis.

Enfim a representação gráfica vem permitir que se abram novas portas cujo conteúdo normalmente é fértil em ideias.

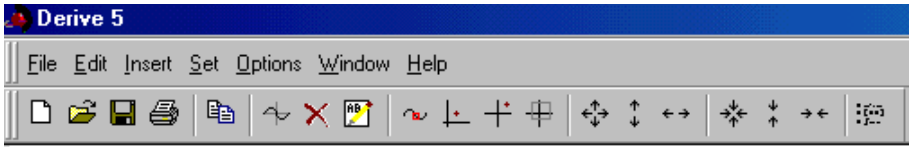
6.1. – FUNÇÕES DE UMA VARIÁVEL

Derive:

Para obter uma representação gráfica de uma determinada função, utilizando o *Derive* temos de introduzir a expressão da função que pretendemos representar e depois, estando a referida expressão seleccionada, utilizar o atalho  .

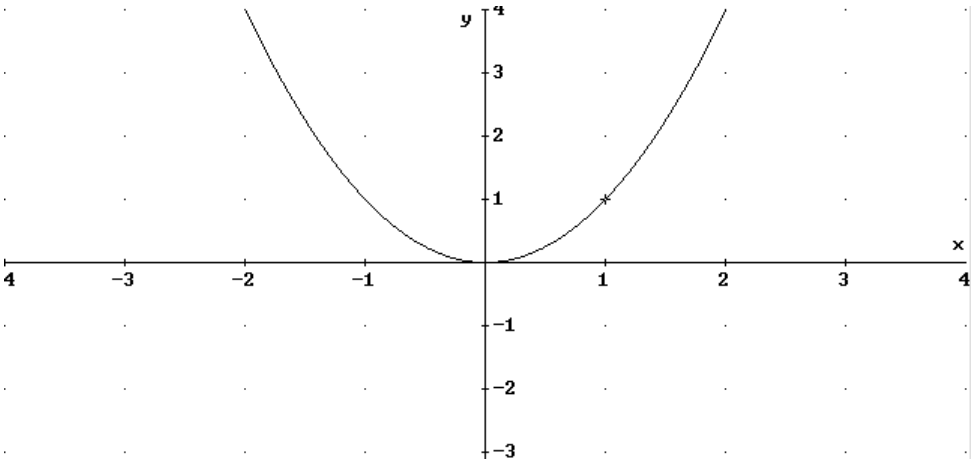
Neste caso vamos representar graficamente a função definida pela expressão $\mathbf{x^2}$.

Em consequência do procedimento descrito anteriormente somos transportados para uma janela diferente. Aí passamos a dispor de uma nova barra de ferramentas com atalhos distintos.

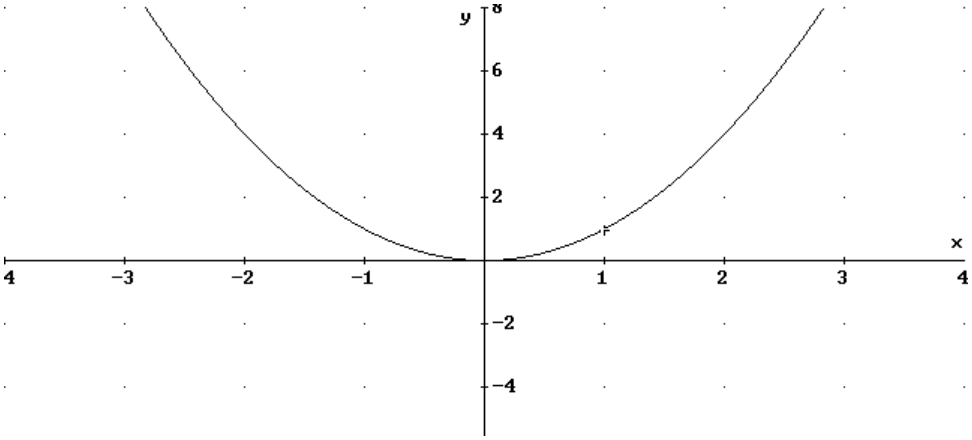


Para obter o gráfico temos de utilizar o atalho  .

Surge-nos então no ecrã a representação gráfica da função definida pela expressão $\mathbf{x^2}$.



Utilizando os diferentes atalhos podemos modificar o aspecto com que a nossa função surge no ecrã. Assim, por exemplo  faz com que a escala do eixo vertical se altere passando o utilizador a conseguir alcançar valores das ordenadas mais distantes da origem.

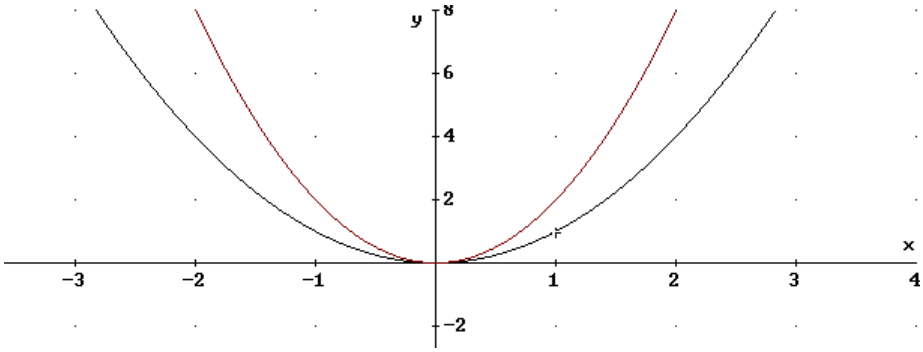


Facilmente compreendemos o efeito que cada um dos atalhos tem sobre a representação gráfica de uma função. Para completar a informação visual que cada atalho nos dá basta experimentá-lo algumas vezes. Assim, com grande facilidade compreendemos o papel que cada um deles tem.

Para regressar à janela de álgebra onde introduzimos a expressão $\mathbf{x^2}$ basta-nos utilizar o atalho mais à direita  .

Introduzindo uma nova expressão $\mathbf{2x^2}$ e seguindo o caminho já efectuado para $\mathbf{x^2}$ somos conduzidos à janela do gráfico anterior que ainda contém o gráfico anterior.

Para obtermos o gráfico da nossa nova função utilizamos mais uma vez o atalho  e imediatamente a nossa função é representada no eixo cartesiano.



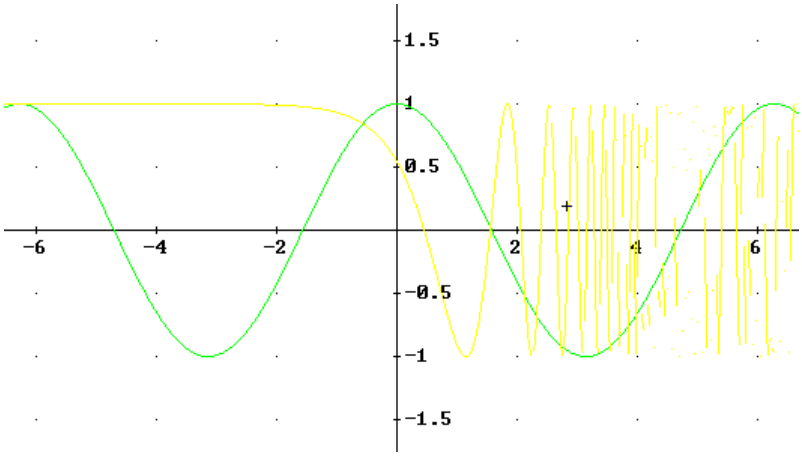
Podemos também optar por seleccionar diversas expressões ao mesmo tempo que o *Derive* representa-as imediatamente sobre o mesmo referencial.

Antes de representarmos duas funções simultaneamente convém limpar as representações já existentes. Para tal utilizamos o atalho  duas vezes, uma vez que são duas as representações gráficas que de momento estão representados.

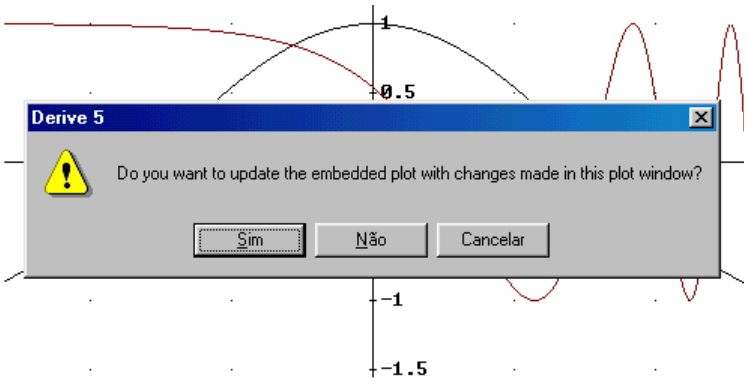
Introduzimos agora as duas expressões das funções a representar e seleccionamos ambas ao mesmo tempo.

$$\cos(x)$$

$$\cos(e^{x})$$



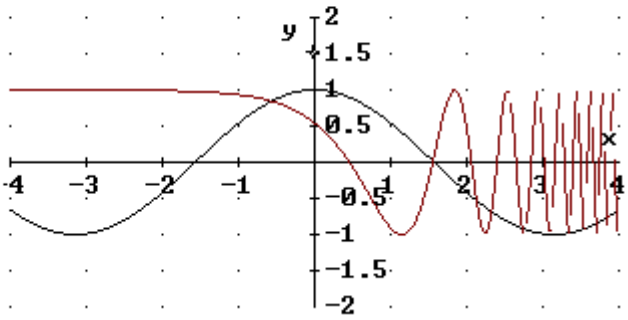
Se pretendermos inserir o gráfico 2D na nossa zona de trabalho temos de seleccionar as funções cujo gráfico queremos visualizar e ir através do menu *Insert* e do submenu *2D Plot object*. Seguindo este caminho, temos de representar as funções pretendidas na janela aberta por esse modo. Desta forma e, após ter colocado o gráfico com o aspecto desejado, para o que são muito úteis os atalhos que controlam a escala nos dois eixos, basta fechar a janela então criada clicando em ✕ e responder afirmativamente à pergunta que o *Derive* formula.



Isto faz com que passemos a ter o gráfico, que anteriormente estava representado numa janela de gráfico de duas dimensões, na nossa zona de trabalho.

$$\cos(x)$$

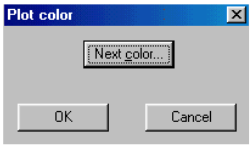
$$\cos(e^{x})$$



Seleccionando a imagem que representa as nossas funções podemos modificar o seu tamanho e clicando duas vezes, como é usual, podemos alterar o aspecto das funções representadas. Para que as alterações sejam aplicadas basta que ao fechar essa janela digamos que as alterações são para ser actualizadas como anteriormente já exemplificámos.

Tal como já tivemos oportunidade de verificar, o *Derive* representa as diferentes funções com cores distintas sendo possível ao utilizador alterar a cor que o programa automaticamente selecciona. Caso a cor não seja do agrado do utilizador carregando em  as cores são alteradas.

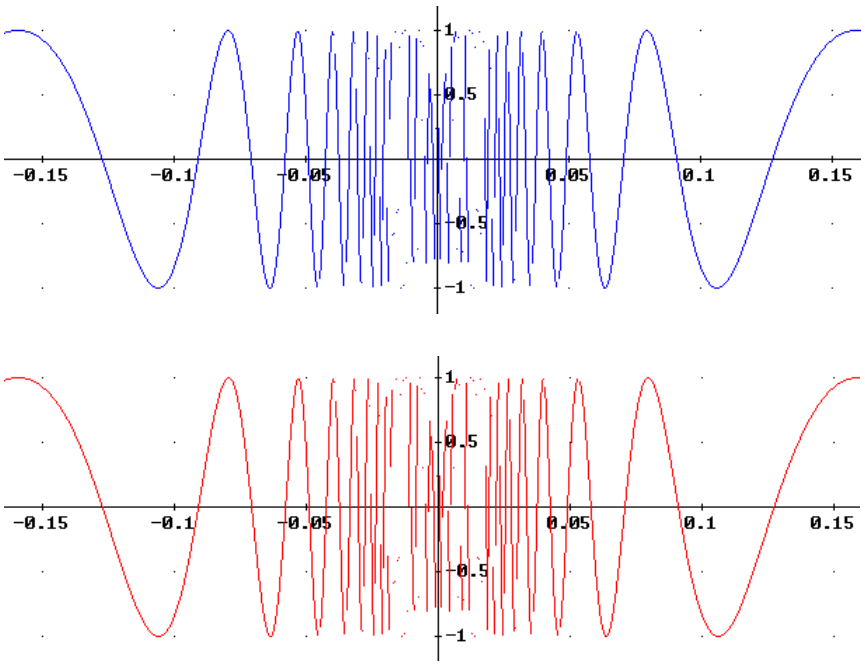
Uma forma de sermos nós a escolher a cor da representação gráfica é usarmos o menu *Options*, o submenu *Display* e optarmos por *Plot color*.



Depois basta carregar em *Next Color* e seleccionar a cor desejada para a próxima função a ser representada.

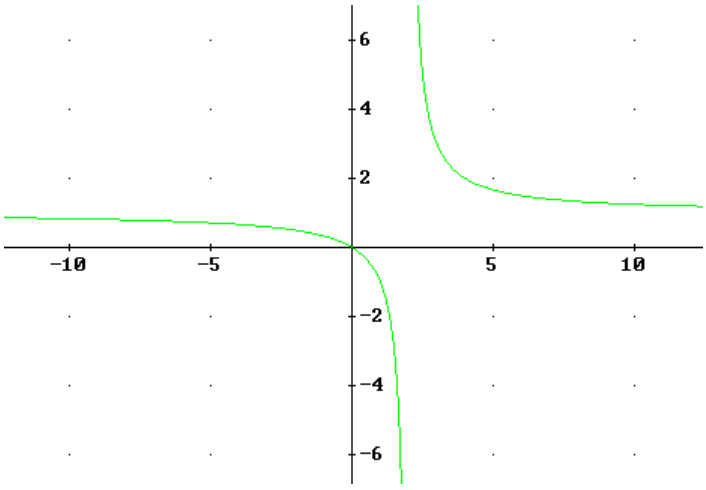
Neste caso optamos por representar a nossa função a azul e a vermelho sucessivamente.

$$\cos\left[\frac{1}{x}\right]$$



Vejamos ainda mais uma função. Reparemos mais uma vez que é o utilizador que por sua iniciativa tem de escolher a janela que melhor representa a função que pretende estudar.

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



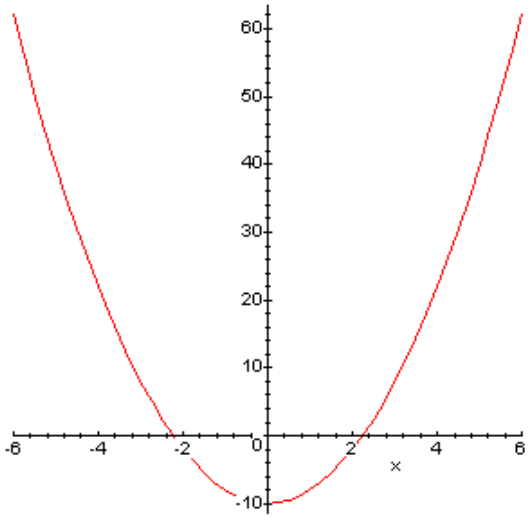
Maple:

Podemos obter uma representação gráfica para uma função definida por uma expressão se utilizarmos a função **plot**.

A representação gráfica permite obter um conhecimento mais profundo acerca da função com que estamos a trabalhar e um avançar mais rápido do trabalho ou estudo que pretendemos realizar.

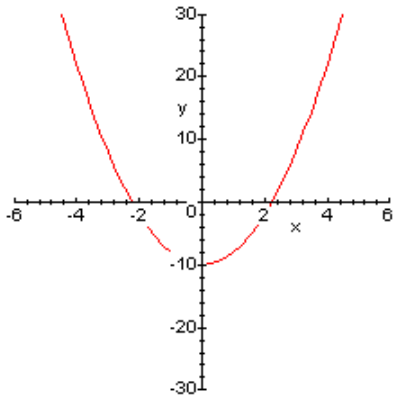
Vejamos então como obter uma representação gráfica de $\mathbf{2x^2-10}$ para valores de **x** entre **–6** e **6**.

```
> plot(2*x^2-10,x=-6..6);
```



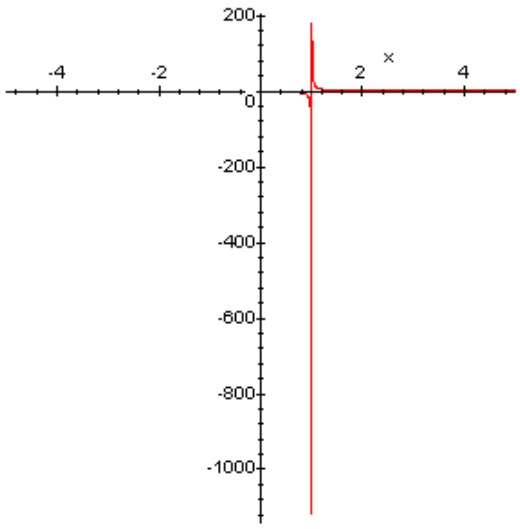
O *Maple* automaticamente dimensiona as ordenadas de modo que surja a representação gráfica dos pontos correspondentes ao domínio especificado. Nem sempre será assim. Se o desejarmos podemos também definir o alcance do eixo dos **y**.

```
> plot(2*x^2-10,x=-6..6,y=-30..30);
```



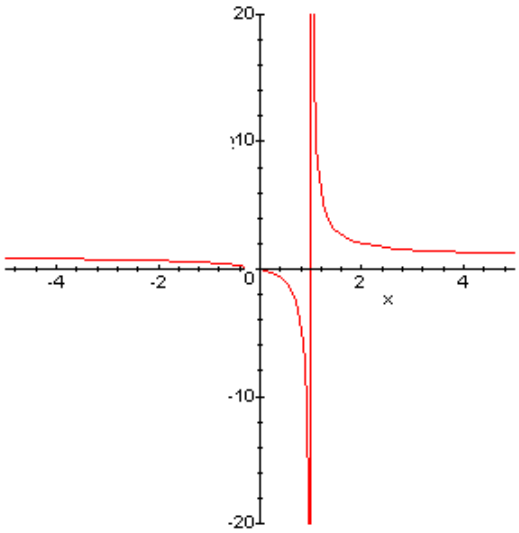
Isto é particularmente útil quando estamos a trabalhar com certo tipo de gráficos, designadamente aqueles que têm assimptotas verticais.

```
> plot(x/(x-1),x=-5..5);
```



Evidentemente não é esta a melhor informação que podemos extrair do gráfico. Analisemos a diferença quando impomos uma escala para o eixo dos **y**.

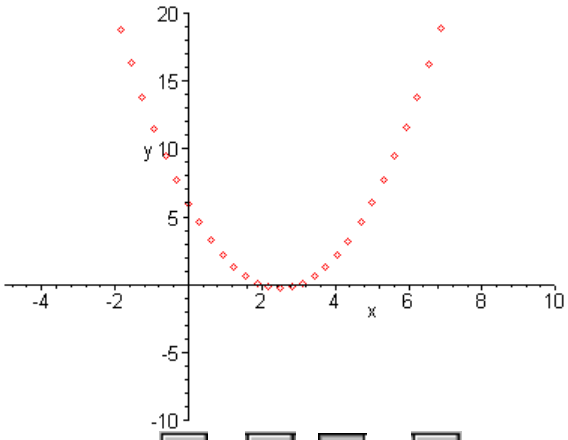
```
> plot(x/(x-1),x=-5..5,y=-20..20);
```



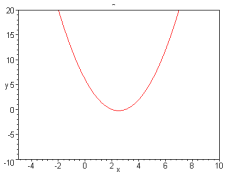
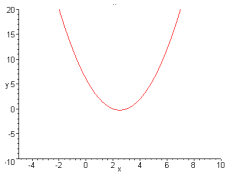
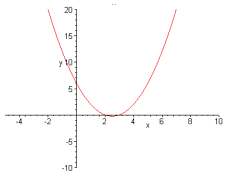
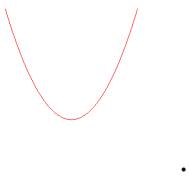
Parece evidente que ganhamos clareza.

Seleccionando o gráfico, a barra de ferramentas inferior é alterada surgindo as seguintes opções.

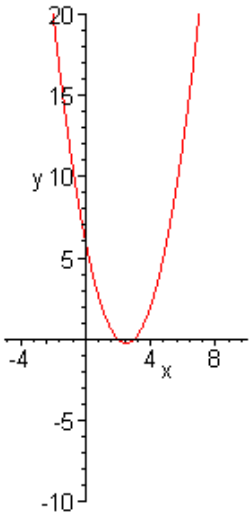
Com estes atalhos e através da nova barra de menus podemos trabalhar sobre as representações gráficas obtidas escolhendo a melhor visualização para cada uma das situações com que estamos a trabalhar. Por exemplo o atalho  quando utilizado faz com a função fique representada por pontos como de seguida mostramos.



Os atalhos  ,  ,  e  permitem um enquadramento diferente do habitual para a função. O primeiro insere a função dentro de um rectângulo no qual os eixos são representados no lado esquerdo e em baixo da função, o segundo representa a função com os eixos totalmente à esquerda e abaixo da função. Quanto ao terceiro, representa os eixos na posição habitual e o último retira os eixos. Aplicando estes atalhos obtivemos as seguintes representações (note-se que reduzimos as imagens):

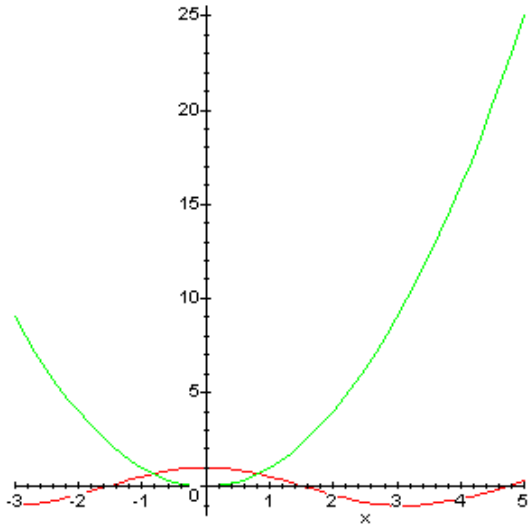





Por último o atalho  , impõe a mesma escala no eixo vertical e horizontal. Aplicamos este atalho partindo da penúltima representação (a que tem os eixos).

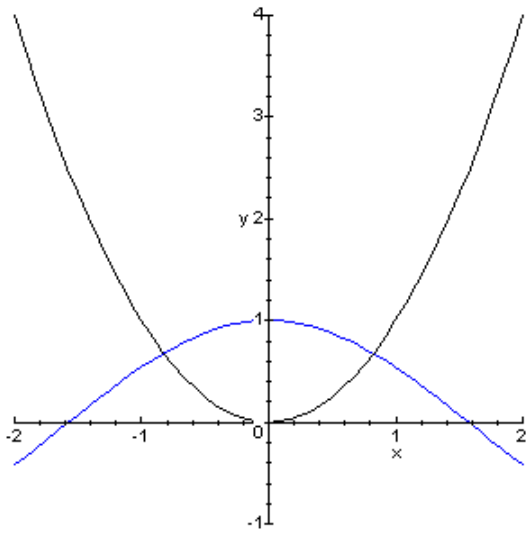


Podemos representar mais do que uma expressão de uma só vez. Para tal temos de construir uma lista onde incluímos as expressões dos gráficos a construir. O *Maple* representa cada gráfico por uma cor diferente, tendo o utilizador a possibilidade de decidir qual a cor a utilizar para cada gráfico.

```
> plot([cos(x),x^2],x=-3..5);
```




```
> plot([cos(x),x^2],x=-2..2,y=-1..4,color=[blue,black]);
```



Para além das cores referidas acima estão ainda disponíveis mais algumas cores.

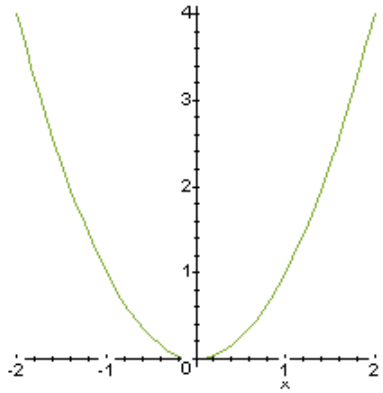
Consultando a ajuda do *Maple* podemos verificar que existem as cores que de seguida se mostram.

```
> ?color
```

```
aquamarine black  blue       navy   coral cyan
brown      gold   green      gray   grey  khaki
magenta    maroon orange     pink   plum  red
sienna     tan    turquoise violet wheat white
yellow
```

Podemos ser nós próprios a criar as nossas cores utilizando a sintaxe: **macro(cor=COLOR(RGB,0.5,0.7,0.2))**. Nesta definição **R**, **G** e **B** são as iniciais de vermelho (*red*), verde (*green*) e azul (*blue*). À frente no *Mathematica* refere-se outra vez esta convenção para as cores.

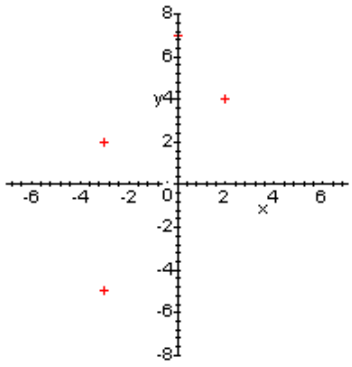
```
> macro(cor=COLOR(RGB,0.5,0.7,0.2));

> plot(x^2,x=-2..2,color=cor);
```



Temos também a possibilidade de representar pontos num gráfico cartesiano.

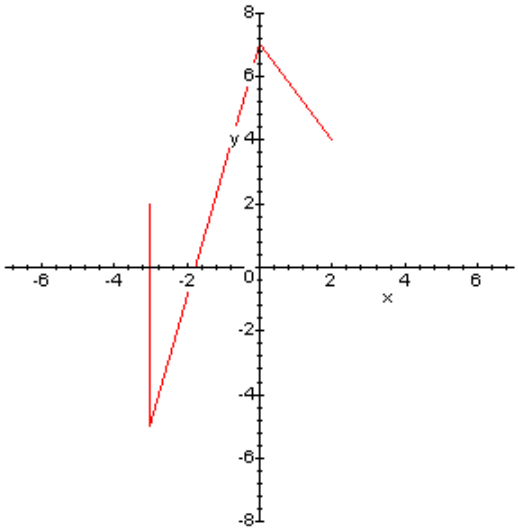
A representação faz-se através da mesma função sendo o primeiro argumento constituído por uma lista de listas denotando cada lista os pontos a serem representados.

```
> plot([[2,4],[0,7],[-3,-5],[-3,2]],x=-7..7,y=-8..8,
  style=point);
```



A junção dos pontos pode obter-se se alterarmos o estilo para *line*.

```
> plot([[2,4],[0,7],[-3,-5],[-3,2]],x=-7..7,y=-8..8,
  style=line);
```

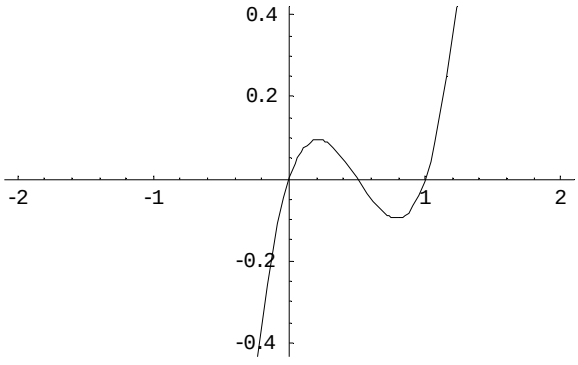


Mathematica:

No *Mathematica* a representação gráfica de uma função obtém-se através da função **Plot**.

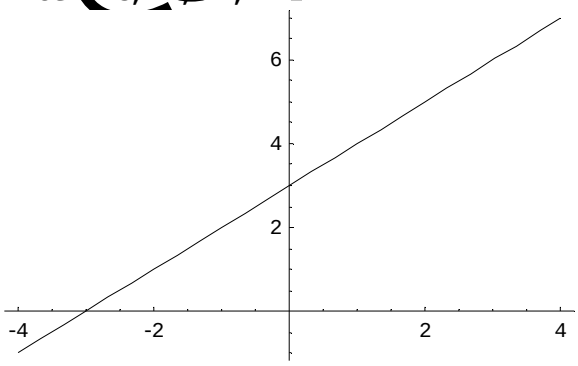
Para representar uma função no *Mathematica* temos de introduzir a expressão da função e especificar qual o domínio em que queremos representá-la. Vejamos alguns exemplos.

```
Plot[x^3-3x^2+x, {x, -2, 2}]
```



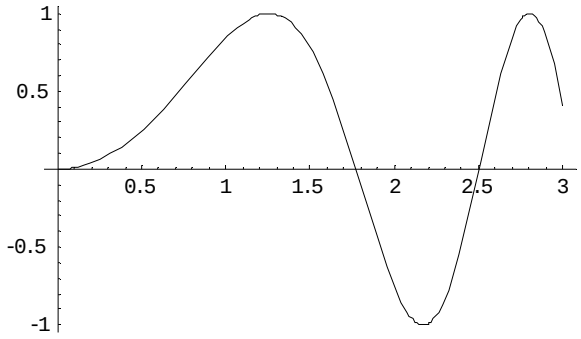

```
...Graphics...
```

```
Plot [illegible], 4, 4 ]
```



...Graphics ...

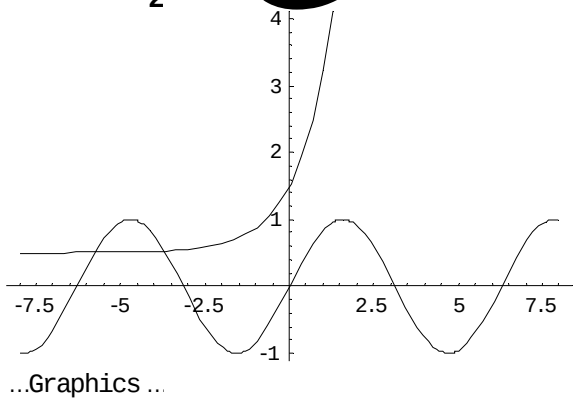
```
Plot [illegible], 3 ]
```



...Graphics ...

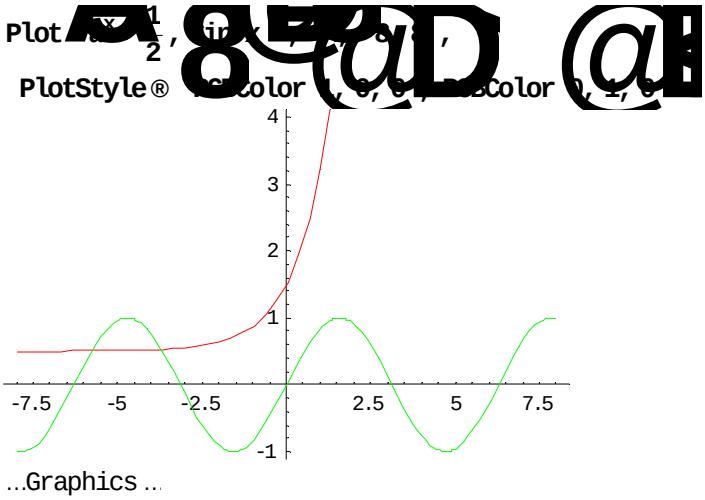
Utilizando as listas, podemos representar num mesmo referencial mais do que uma função. Para isso temos de colocar como primeiro argumento da função **Plot** uma lista constituída pelas expressões das duas funções que queremos considerar.

```
Plot [illegible], Sin [illegible], 8, 8 ]
```



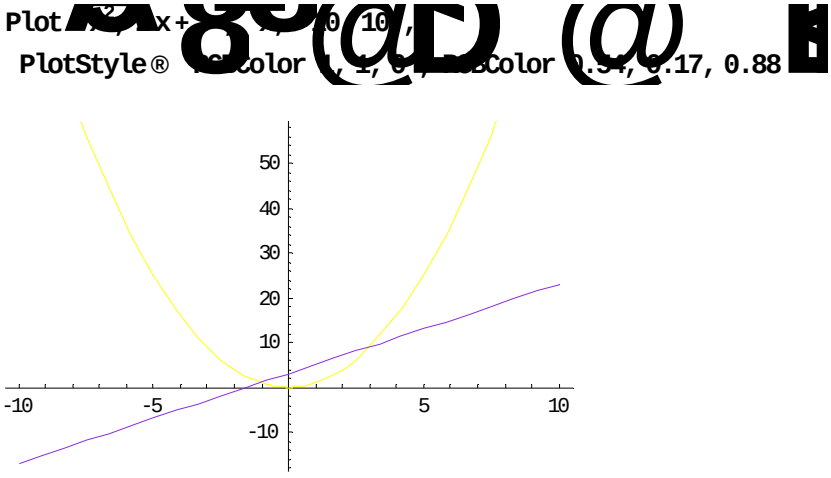
...Graphics ...

Como podemos observar, as funções são representadas na mesma cor, neste caso a negro. Se quisermos é possível representá-las com uma cor diferente. Para obter tal resultado final temos de definir nós próprios a tonalidade pretendida mediante a inclusão de um terceiro argumento.



...Graphics...

Aqui **RGB** são as iniciais de *Red*, *Green* e *Blue* (vermelho, verde e azul) e [1,0,0] significa que a única cor a considerar é o vermelho. Qualquer um dos argumentos da função **RGBColor** será um número compreendido entre zero e um e serão estes que definirão a tonalidade da cor com que surgirá a função que a ele se refere.



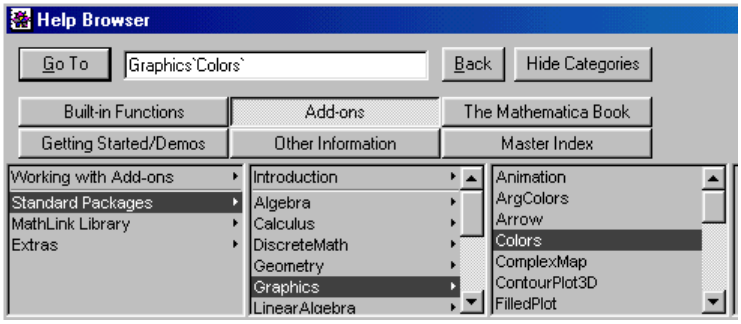
...Graphics...

O *Mathematica* disponibiliza-nos um vasto conjunto de cores a que podemos ter acesso usando a função **RGBColor**. Aqui ficam algumas delas:

AliceBlue	AlizarinCrimson
Antique	Apricot
Aquamarine	AureolineYellow
Azure	Banana
Beige	Bisque
Black	BlanchedAlmond
Blue	BlueViolet
Brick	Brown

.

Poderá consultar as restantes através do pacote **Graphics`Colors`**.



Carregando o pacote referido, o *Mathematica* informa-nos acerca dos parâmetros a usar na função **RGBColor** para obter cada uma das cores.

Comecemos por carregar o pacote.

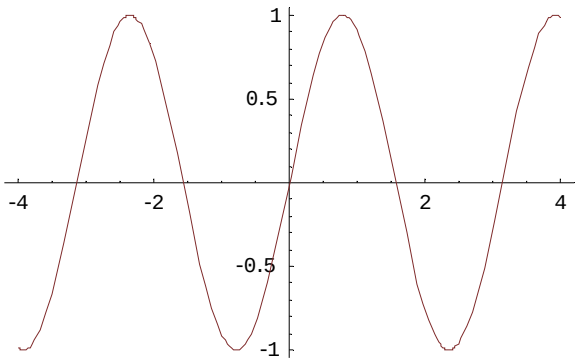
```
<< Graphics`Colors`
```

Para sabermos como obter a cor castanha basta-nos escrever **Brown** e mandar avaliar esta expressão e assim obtemos os parâmetros que pretendíamos.

```
Brown
RGBColor 0.5, 0.164693, 0.164693
```

Verifiquemos que com os parâmetros indicados, realmente obtemos um gráfico representado com a cor referida.

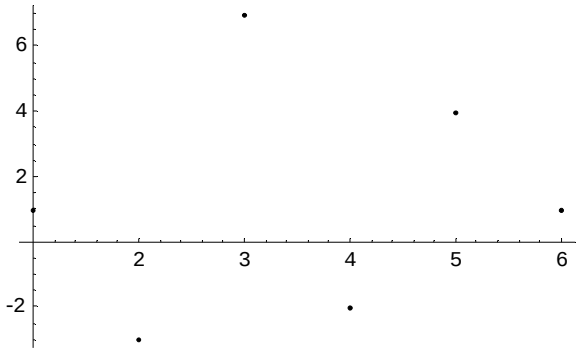
```
Plot Sin[x], {x, -4, 4}, PlotStyle → RGBColor[0.5, 0.164693, 0.164693]
```




```
-Graphics-
```

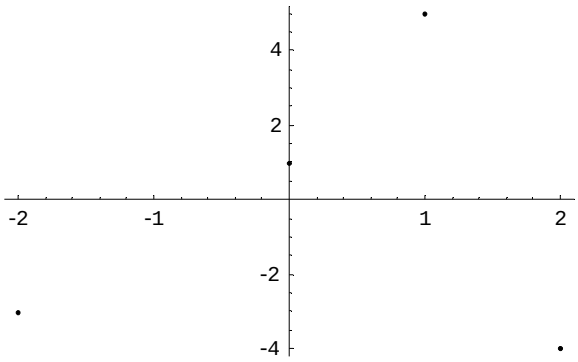
Para representar um conjunto de pontos utilizamos a função **ListPlot** cujo argumento pode ser uma lista ou uma lista de listas. No primeiro caso os elementos da lista são as imagens de 1, 2, 3, ... enquanto que no segundo cada uma das listas mais pequenas constitui um par ordenado constituído pelo valor da abcissa e da ordenada de cada ponto.

```
ListPlot[{1, 3, 7, -2, 4, 1}]
```



...Graphics...

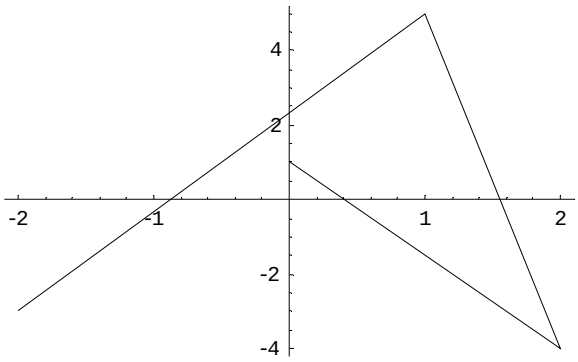
```
ListPlot[{{-2, 3}, {-1, 1}, {1, 4}, {2, -2}}]
```



...Graphics...

A junção dos pontos é conseguida do seguinte modo:

```
ListPlot[{{-2, 3}, {-1, 1}, {1, 4}, {2, -2}}, PlotJoined → True]
```



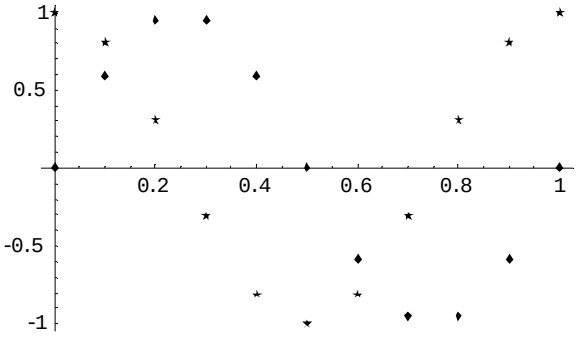
...Graphics...

Indo um pouco mais além podemos referir que é possível representar mais que um conjunto de valores num mesmo referencial mas, para tal, é necessário carregar o pacote **MultipleListPlot**.

```
<< Graphics`MultipleListPlot`
```

Vamos em primeiro lugar construir duas listas com os valores que pretendemos representar para o que usaremos a função **Table**. Optamos por terminar cada uma das expressões *Mathematica* que definimos com "**;**" para que não sejam imprimidas as coordenadas dos pontos que representamos já de seguida.

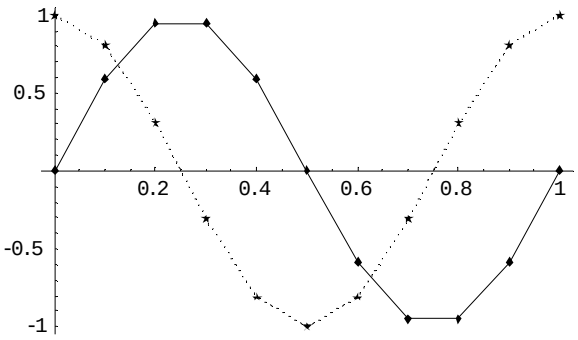
```
lista1 = Table [ ... , 1, 0.1 ];
lista2 = Table [ ... , 1, 0.1 ];

MultipleListPlot [lista1, lista2]
```




```
...Graphics...
```

Observando com atenção conseguimos distinguir quais os pontos pertencentes a cada uma das listas dado que os pontos são representados por símbolos diferentes. No entanto, mais fácil é dizermos ao *Mathematica* para juntar os dois conjuntos de pontos da mesma forma que fizemos no caso anterior.

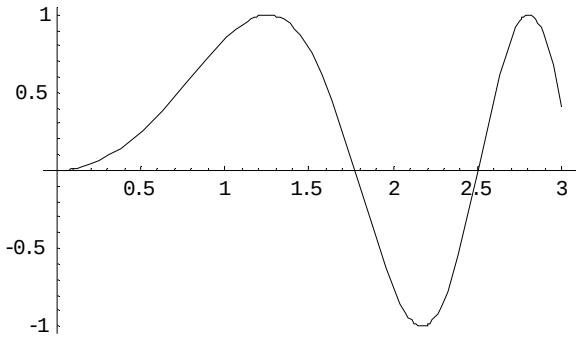
```
MultipleListPlot [lista1, lista2, PlotJoined → True]
```




```
...Graphics...
```

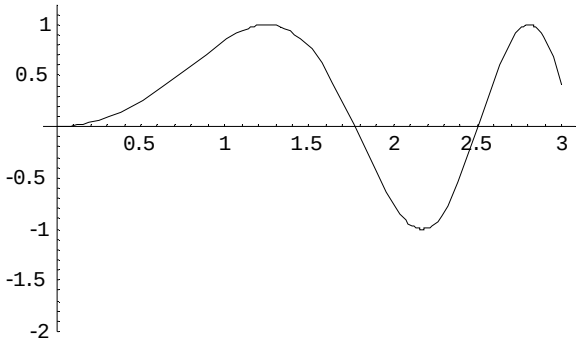
Quando elaboramos um gráfico não temos que especificar para que valores das ordenadas queremos a nossa representação. Neste caso o programa automaticamente se encarrega de definir o valor máximo e o valor mínimo para o eixo vertical. Contudo, essa é uma facilidade que, sendo prática em grande parte dos casos, dos quais são exemplo os que até aqui analisámos, torna mais lento o descortinar de algumas propriedades ou características que em alguns casos importa realçar. Para ultrapassar esta por vezes inconveniente característica, podemos especificar a opção **PlotRange** que nos permite controlar a porção de gráfico que pretendemos visualizar quer no eixo das abcissas quer no eixo das ordenadas.

```
Plot [illegible], 3
```



...Graphics...

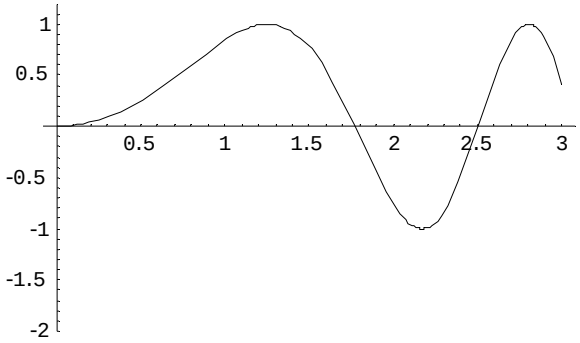
```
Plot [illegible], 3 , PlotRange ® [illegible], 1.2
```



...Graphics...

Para voltar a desenhar um gráfico podemos utilizar **Show.**

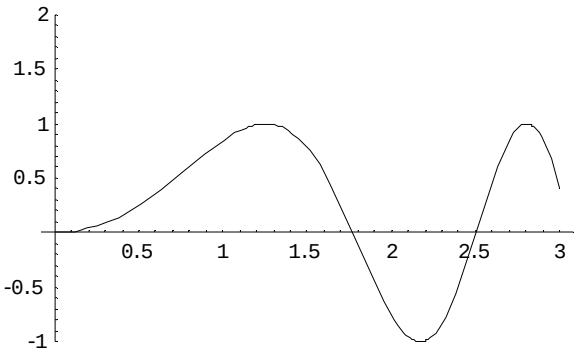
```
Show [illegible]
```



...Graphics...

Ao efectuar este procedimento podemos efectuar algumas alterações.

```
Show[%, PlotRange -> {-1, 2}]
```




```
-Graphics-
```

6.2. - FUNÇÕES DE DUAS VARIÁVEIS

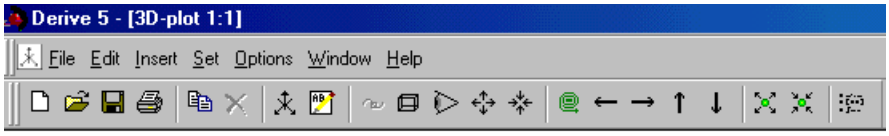
Derive:

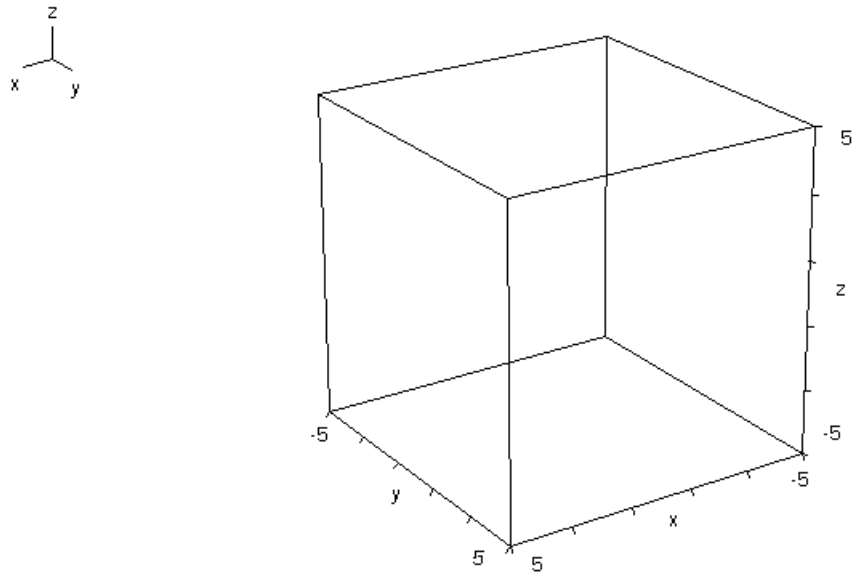
Para representar uma função que depende de duas variáveis seguimos um trajecto em tudo semelhante ao que vimos para as funções de uma variável.

Em primeiro lugar ilustramos como se pode passar da representação de um gráfico a duas dimensões para um gráfico a três dimensões.

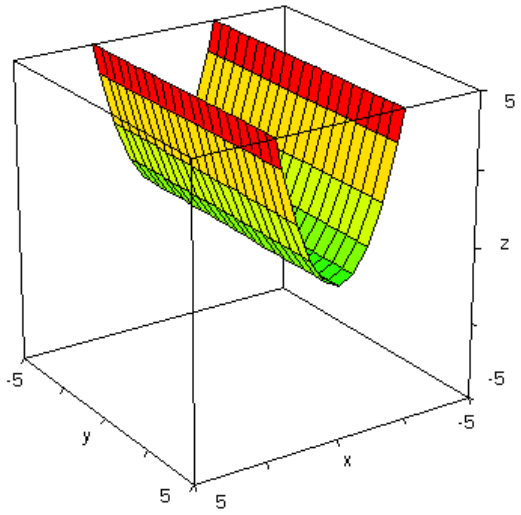
Usaremos a função $\mathbf{x^2}$.

Utilizando o atalho  somos levados para uma janela 3D. Aí encontramos representado um cubo no qual será representada a nossa função. Também neste caso, à semelhança do que acontecia nos gráficos de duas dimensões, as nossas barras de menus e de ferramentas são alteradas.

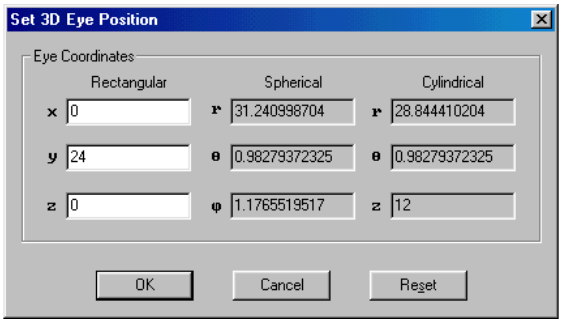




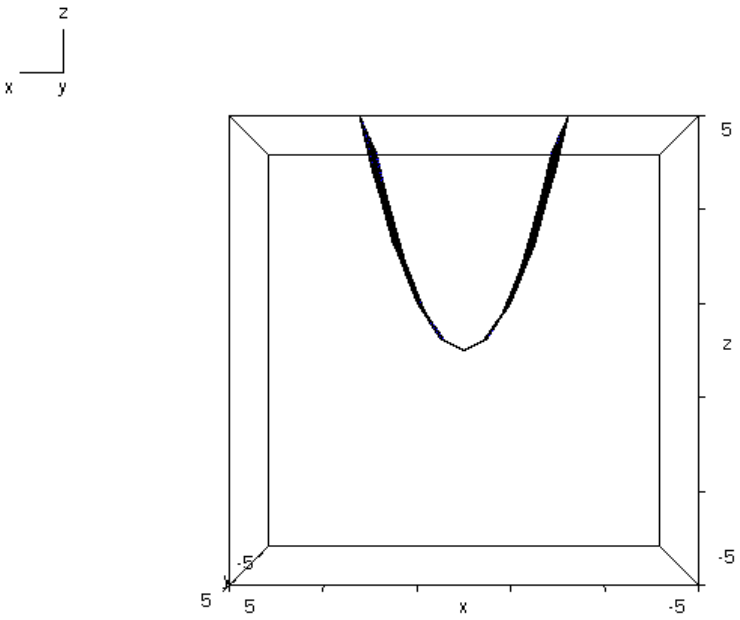
Usando o atalho  obtemos o gráfico da nossa função.



Podemos modificar a posição de onde estamos a ver a superfície de tal modo que o eixo dos **y** fique mesmo à nossa frente. Usando o atalho  e alterando as coordenadas como de seguida mostra a janela

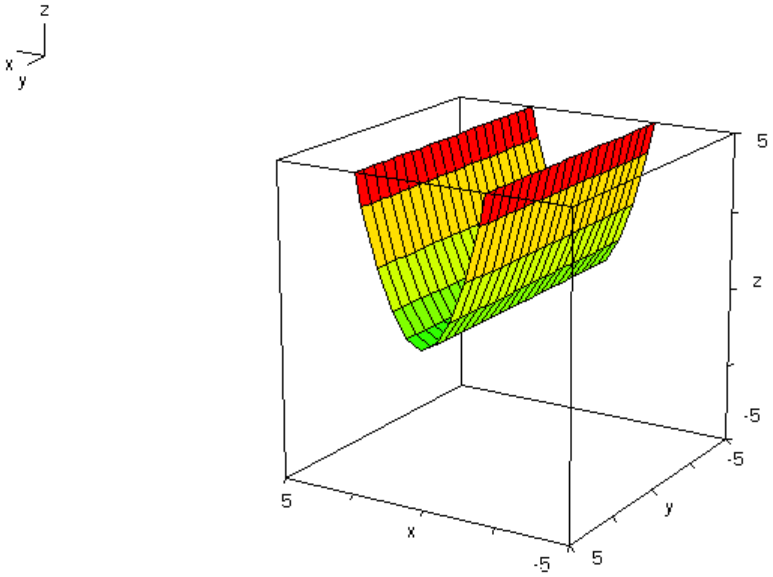


obtemos uma imagem que se assemelha muito a um gráfico de duas dimensões, isto porque, deste ponto de vista, não nos é possível ver a variação das ordenadas.



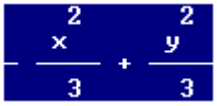
Vemos que podemos imaginar que as representações de funções de duas variáveis podem ser entendidas como uma generalização das de uma variável.

Mediante a utilização dos atalhos  ,  ,  e  é possível ao utilizador colocar a figura na posição que considera mais adequada sem qualquer dificuldade.



Representemos mais uma função.

Para voltarmos à nossa zona de trabalho temos de utilizar o atalho  .

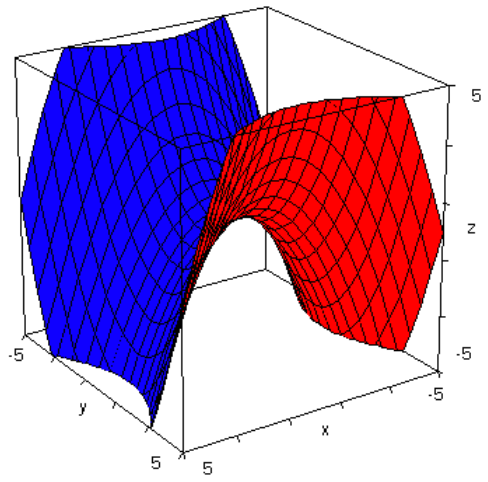


Seleccionando a expressão e voltando à janela do gráfico 3D encontramos ainda o gráfico anterior lá representado.

Sendo possível representar mais que uma função de duas variáveis simultaneamente achamos que será mais útil representar cada função individualmente.

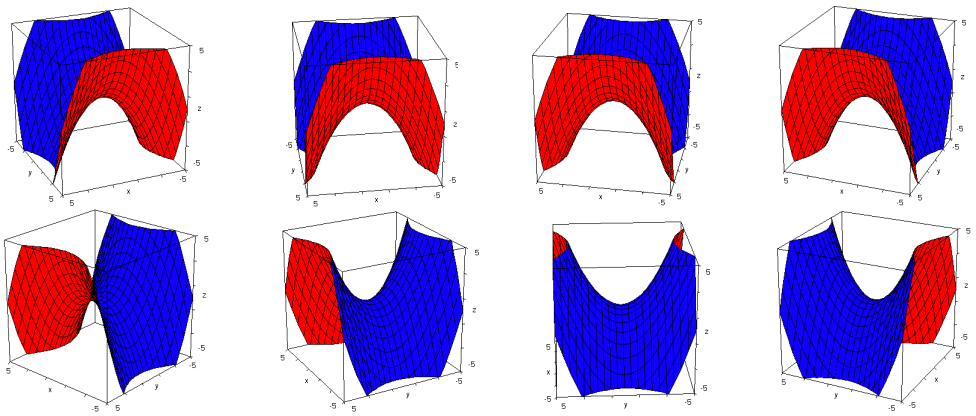
Apagando a representação gráfica utilizando  podemos representá-la como anteriormente.

Embora tenhamos alterado a cor da imagem seguinte deixamos esse tópico um pouco mais para a frente, uma vez que o mesmo não é relevante para o que se segue.

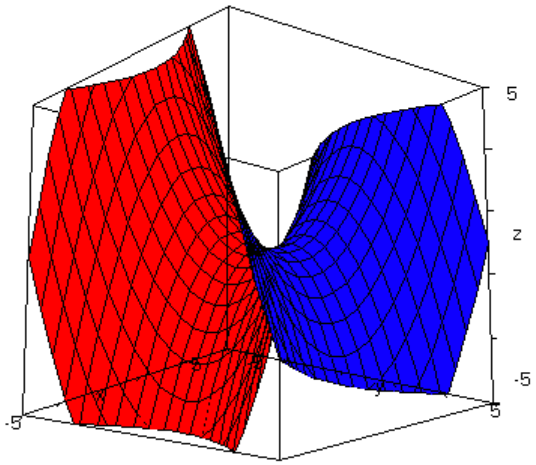


Refira-se ainda que, caso exista mais que uma função representada, para conseguir que as mesmas sejam apagadas, é necessário recorrer ao menu *Edit* e ao submenu *Delete All Plots.*

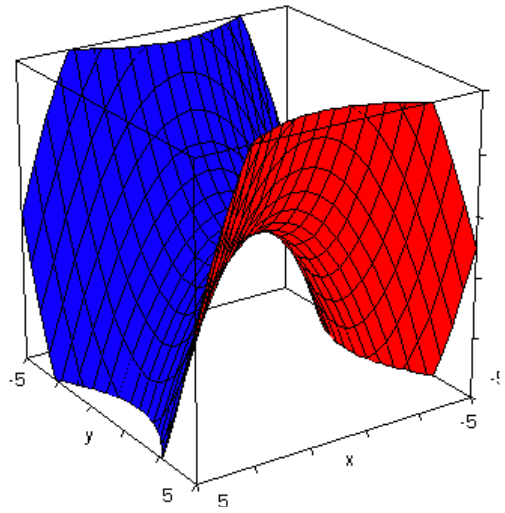
Podemos ainda complementar as nossas alterações fazendo-as de uma forma sistemática usando  o que faz com a que figura passe a efectuar um movimento continuo de rotação. Enquanto a figura roda em torno do eixo dos **z** é possível incliná-la usando os atalhos  e  anteriormente referidos. Na ilustração seguinte, que representa uma sequência de imagens tirada desse movimento de rotação, procedeu-se a uma redução das imagens.



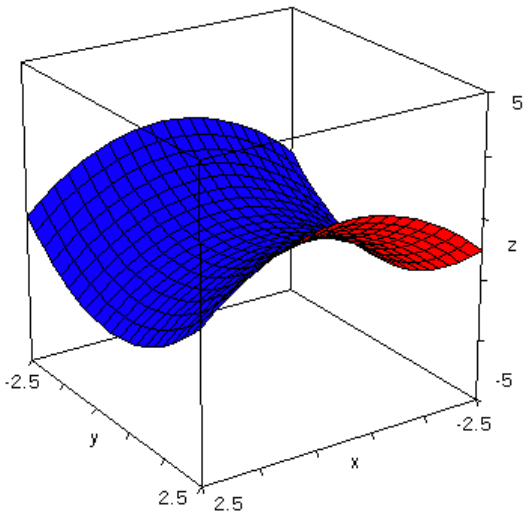
Por outro lado também é possível aumentar e diminuir o tamanho do gráfico usando  e  respectivamente, efeito que também é conseguido carregando em PgDn ou PgUp. Utilizamos o atalho  várias vezes para passar da imagem da esquerda para a imagem da direita.



Para voltarmos à imagem com tamanho inicial basta-nos, mediante a utilização de  , seleccionar RESET e carregar em OK.



Os atalhos  e  destinam-se a modificar simultaneamente a escala das ordenas e das abcissas. Usamos o atalho  para obter a imagem seguinte a partir da anterior.



Também aqui para voltar à figura inicial utilizamos um atalho, desta vez  e fazemos RESET e OK.

Vamos agora representar um cone.

A expressão geral de um cone é dada por $\frac{x^2}{a^2}+\frac{y^2}{b^2}-\frac{z^2}{c^2}=0$. Para representarmos o cone temos de resolver a equação anterior em ordem a **z**.

Utilizando o *Derive* essa resolução é bastante fácil.

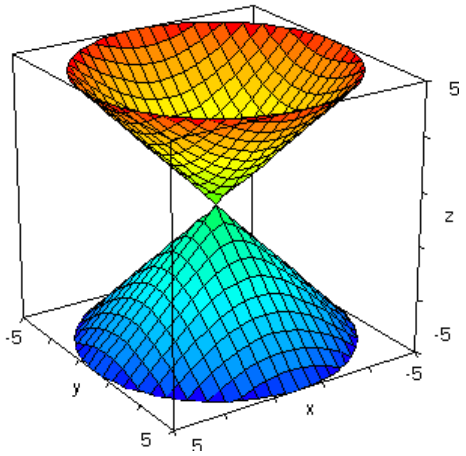
$$\text{SOLVE}\left(\frac{x^2}{a^2}+\frac{y^2}{b^2}-\frac{z^2}{c^2}=0,\ z\right)$$

$$z=-\frac{c\cdot\sqrt{b^2\cdot x^2+a^2\cdot y^2}}{a\cdot b}\ \vee\ z=\frac{c\cdot\sqrt{b^2\cdot x^2+a^2\cdot y^2}}{a\cdot b}$$

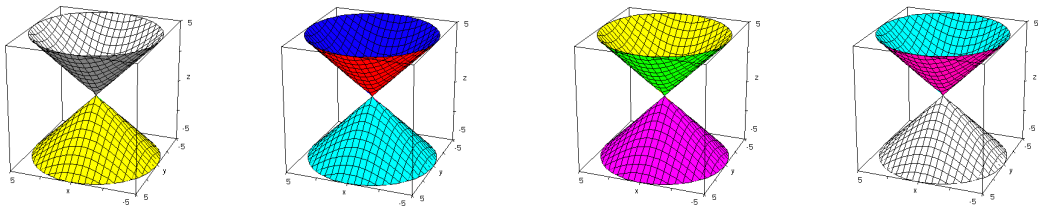
Vamos representar um cone em que **a=b=c=2**. Assim obtemos que o valor da cota de cada ponto do gráfico é dado pela seguinte expressão.

$$z=-\frac{2\cdot\sqrt{2^2\cdot x^2+2^2\cdot y^2}}{2\cdot 2}\ \vee\ z=\frac{2\cdot\sqrt{2^2\cdot x^2+2^2\cdot y^2}}{2\cdot 2}$$

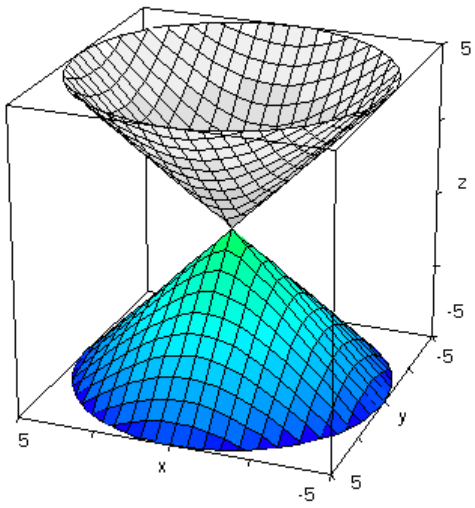
Na obtenção do *output* anterior utilizamos a capacidade que o *Derive* tem para substituir variáveis por valores concretos. Para o efeito foi utilizado o atalho  . Mandando simplificar a expressão anterior obtemos que  . Representando graficamente esta disjunção de condições obtemos a seguinte representação.



Seleccionando a opção *Change Colors* através do menu *Options* conseguimos que sempre que mandamos representar uma expressão esta seja apresentada de uma cor distinta da anterior. Mandamos representar graficamente a expressão do cone algumas vezes e obtivemos estes resultados.

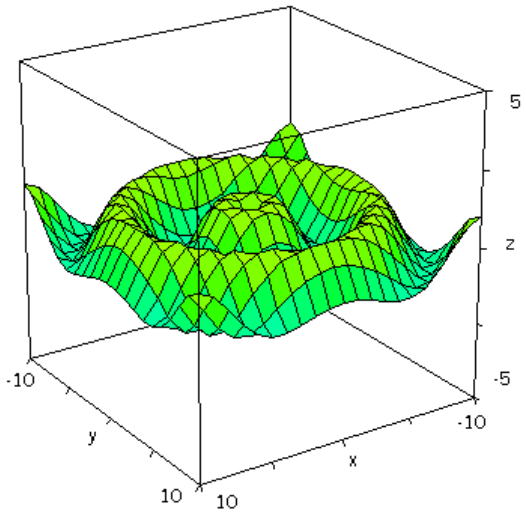


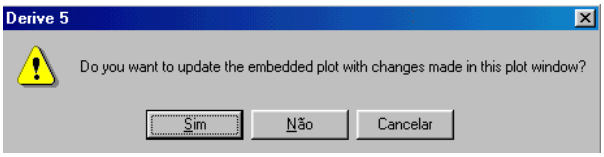
Clicando uma vez em cima do gráfico para que este fique seleccionado e depois clicando mais duas vezes somos transportados para uma janela onde, seleccionando *Plot Color*, podemos escolher outros tipos de coloração. Neste caso optámos por *Gray Scale*.



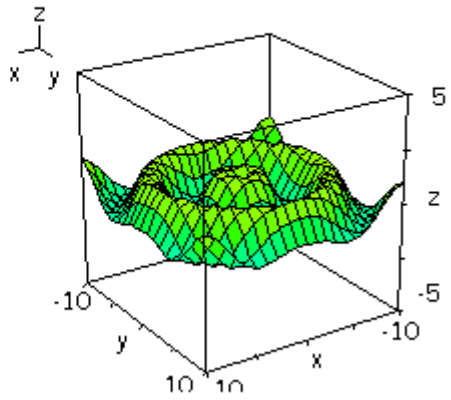
Repare-se que apenas se alterou a parte superior da imagem mas, recordemos também a forma como obtivemos a representação. Esta foi obtida através de um disjunção de duas expressões pelo que, ao seleccionarmos a parte superior, o *Derive* subentende que é apenas essa a parte que desejamos modificar. Para alterar a parte inferior da imagem teríamos de a ter seleccionado.

Para inserir um gráfico na zona de trabalho seleccionamos a expressão da função e através do menu *Insert* e do submenu *3D-Plot Object* elaboramos o gráfico seguindo o percurso já exemplificado e depois fechamos a janela carregando em ✕ respondendo afirmativamente à questão formulada. Neste caso pretendemos inserir a representação gráfica da função $z = Sin\left(\sqrt{x^2 + y^2}\right)$.





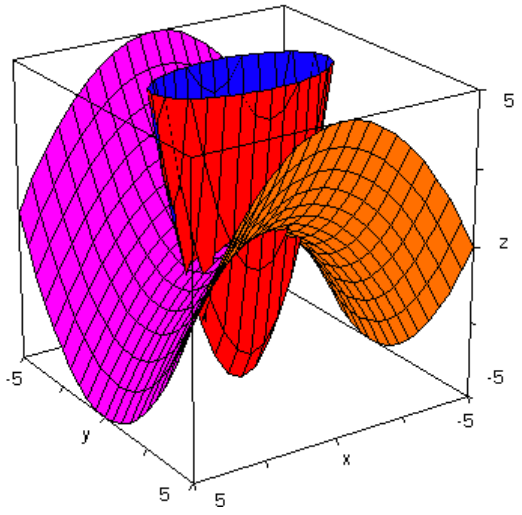
Em resultado do procedimento anterior obtemos o gráfico pretendido na nossa zona de trabalho.



A terminar esta parte relativa às funções de duas varáveis, apresentamos aqui a representação simultânea de duas funções de duas variáveis obtida pela selecção das respectivas expressões introduzidas no nosso local de trabalho.

$$1 \cdot x^2 + 4 \cdot y^2 - 5$$

$$-\frac{x^2}{5} + \frac{y^2}{5}$$

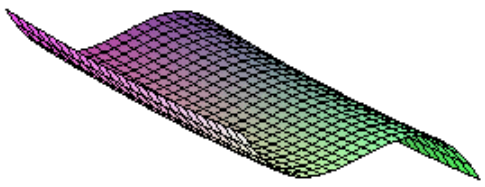


Maple:

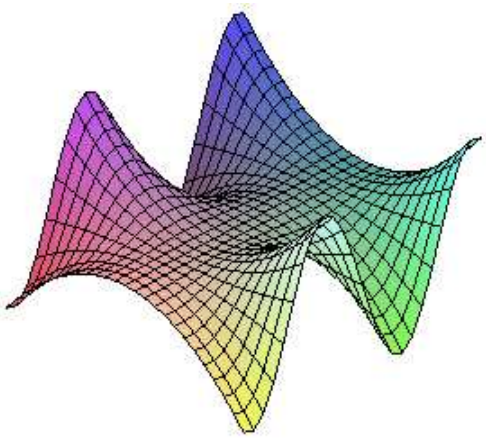
A função que é utilizada na representação gráfica de funções de duas variáveis é **plot3d**.

Para que consigamos obter uma representação de uma superfície temos de especificar quais os valores que devem ser considerados para o domínio da função. Por essa razão temos de especificar os intervalos nos quais queremos a nossa representação feita. Comecemos esta secção com alguns exemplos simples para depois passarmos a situações um pouco mais complicadas algumas das quais já analisadas através do *Derive*.

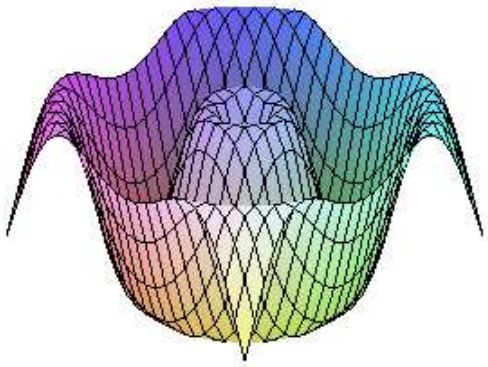
```
> plot3d(5*x^3+5*y^2+3,x=-15..15,y=-15..15);
```



```
> plot3d(sin(x)*y^2,x=-5..5,y=-5..5);
```

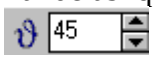
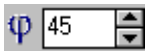


```
> plot3d(sin(sqrt(x^2+y^2)),x=-7..7,y=-7..7);
```



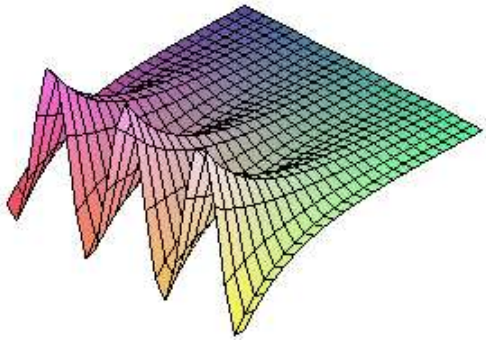
Se seleccionarmos o gráfico de qualquer uma das representações gráficas teremos oportunidade de verificar que a barra de ferramentas bem como a barra de menus modificam-se para que estejam mais aptas a trabalhar com estas representações.

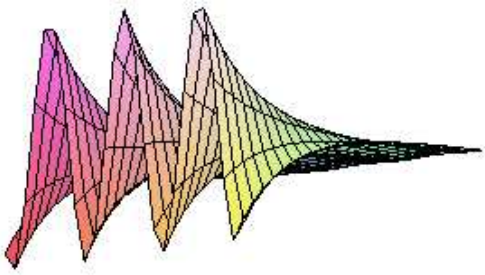
Recorrendo aos atalhos disponibilizados pelo *Maple* conseguimos visualizar a nossa superfície de várias formas.

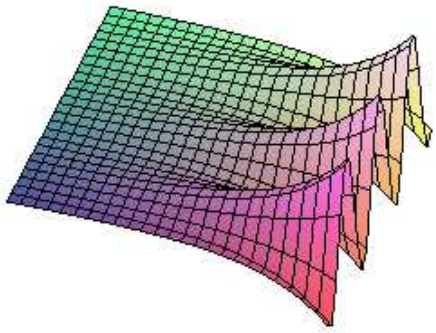
Comecemos por ilustrar uma delas que consiste na capacidade de rodar a imagem. Fazendo uso de  em  ou  ou então simplesmente escrevendo os valores correspondentes à rotação a efectuar nos eixos vertical e horizontal conseguimos observar a imagem de diferentes pontos de vista.

Exemplifiquemos essa situação com uma outra função.

```
> plot3d(cos(y)*exp(0.25*x),x=-10..10,y=-10..10);
```

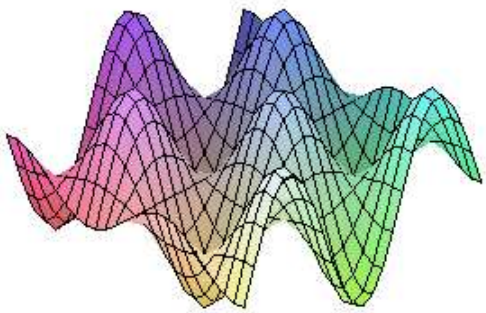


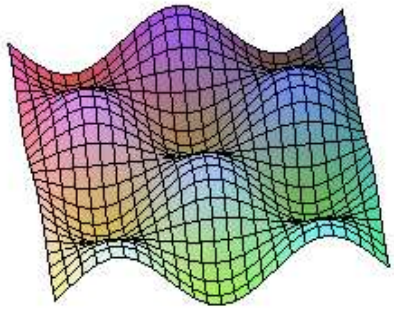




Outra forma de conseguirmos alterar o nosso ponto de vista é utilizando o rato. Desta forma, só temos de clicar com o rato mantendo premido o botão e arrastar a superfície para a posição desejada. Este é um processo bastante fácil de pôr em prática uma vez que a liberdade de movimentos que nos é concedida é quase total. Podemos rodar a nossa representação gráfica em qualquer sentido e partindo de qualquer posição. É um processo bastante prático e muito eficaz.

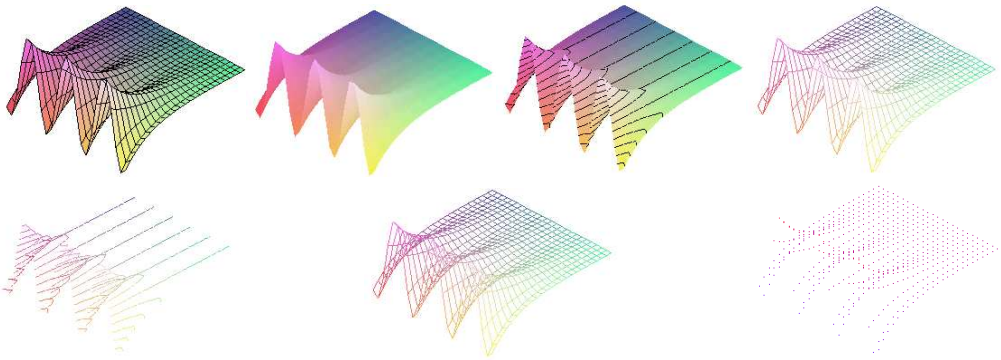
```
> plot3d(cos(x)*sin(y),x=-5..5,y=-5..5);
```



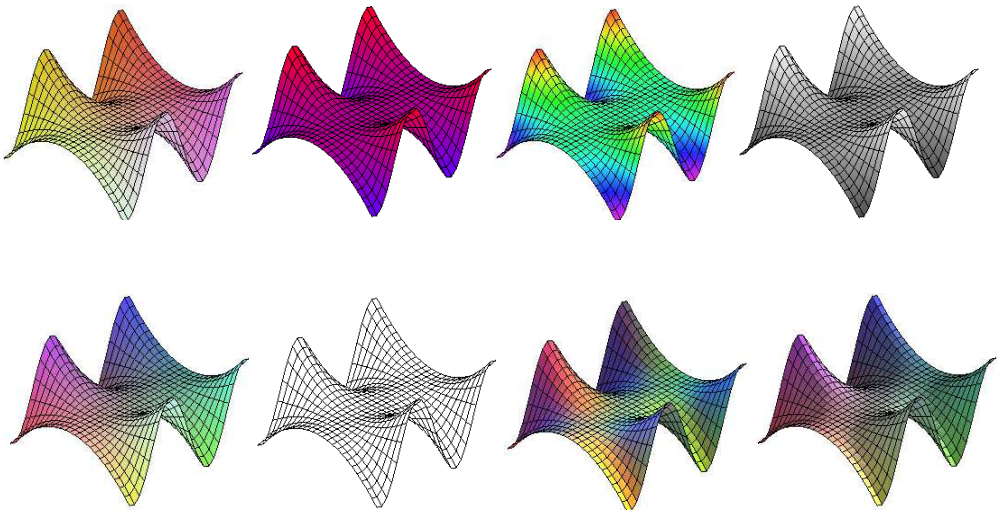


Para regressar à imagem inicial basta voltar a avaliar a expressão que lhe deu origem. Neste caso a simplicidade também se mantém.

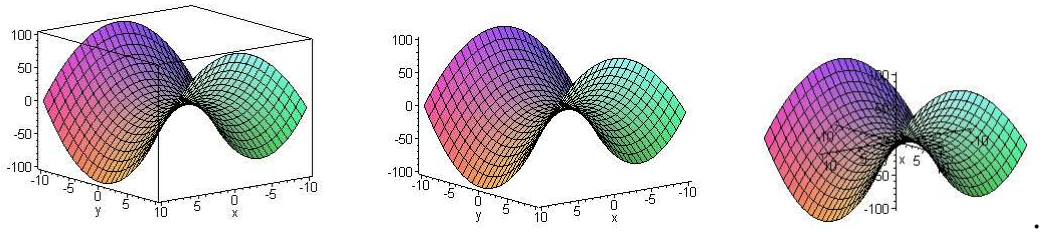
No respeitante ao aspecto com que a representação surge existem alguns atalhos, nomeadamente  ,  ,  ,  ,  ,  e  cuja aplicação foi feita sequencialmente à mesma função e em que o resultado final é mostrado de seguida. Estes modos de representação de funções também podem ser obtidos através do menu *Style*.



Seguindo o menu *Color* estão disponíveis alguns submenus cuja utilização se traduz por uma mudança no colorido da representação. De seguida apresentamos alguns desses aspectos.



Também é possível inserir eixos nas diferentes representações. Até ao momento todas as representações gráficas tem sido elaboradas sem eixos mas tal não tem de ser sempre assim. Recorrendo ao menu *Axes* ou aos atalhos  ,  ,  e  é possível alterar esta situação. O último atalho destina-se à representação sem eixos enquanto os restantes produzem o seguinte efeito:

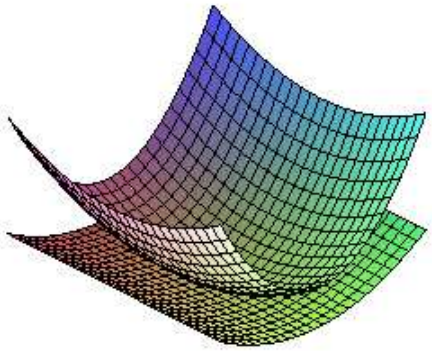


A expressão *Maple* utilizada para gerar a superfície anterior foi **plot3d(-x^2+y^2,x=-10..10,y=-10..10);**.

Ao pretender representar mais que uma função de uma só vez há que carregar o pacote **plots** e depois utilizar a função **display3d**. Para uma mais eficiente compreensão recorremos a duas atribuições. O mesmo poderia fazer-se sem a elas recorrer.

```
> with(plots):
Warning, the name changecoords has been redefined

> p:=plot3d(x^2,x=-5..5,y=-5..5):

> q:=plot3d(3*x^2+y^2,x=-5..5,y=-5..5):
```

```
> display3d([p,q]);
```

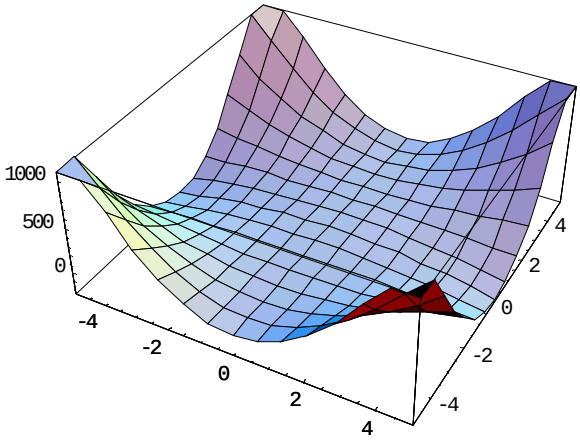


Mathematica:

A representação gráfica de funções de duas variáveis, em *Mathematica*, faz-se através da função **Plot3D**.

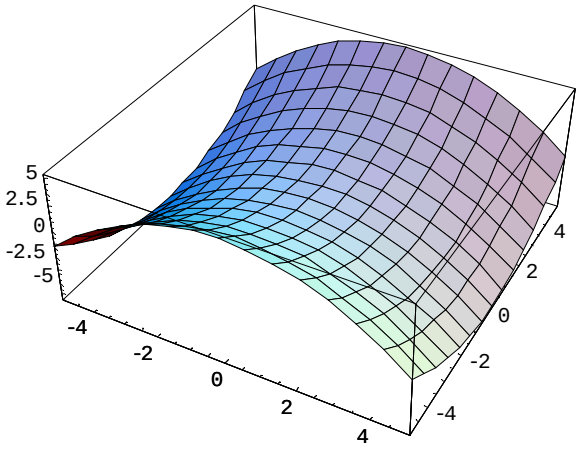
Mediante a especificação da expressão da função a representar e dos valores para o domínio das duas variáveis em questão podemos obter uma imagem para a nossa expressão.

```
Plot3D [illegible]y^2- 8*Abs[illegible]5, 5 , [illegible]5, 5 ]
```



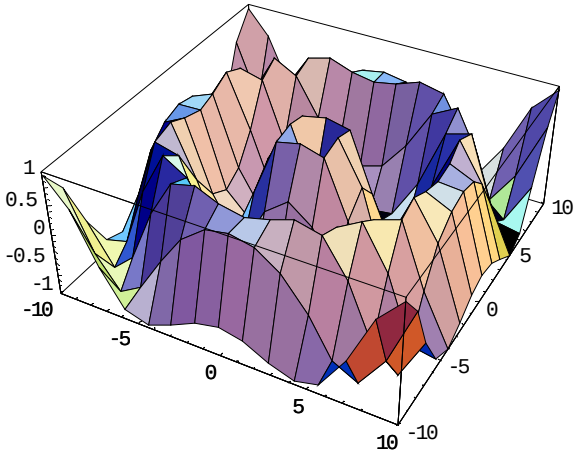
...SurfaceGraphics ...

```
Plot3D[x/4 - y^2/5, {x, -5, 5}, {y, -5, 5}]
```



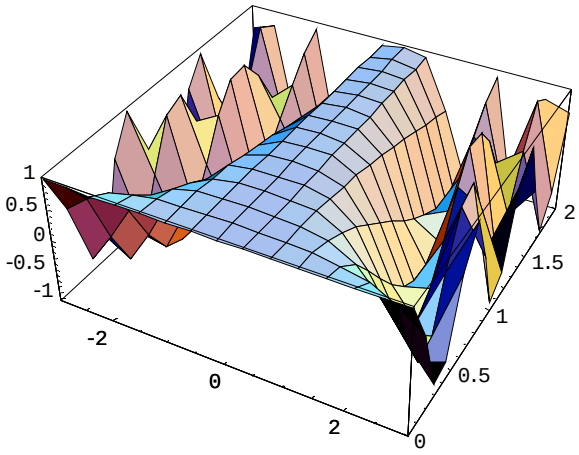
...SurfaceGraphics...

```
Plot3D[Sin[x^2 + y^2], {x, -10, 10}, {y, -10, 10}]
```



...SurfaceGraphics...

```
Plot3D[Cos[x^2 - y], {x, -3, 3}, {y, 0, 2}]
```



...SurfaceGraphics...

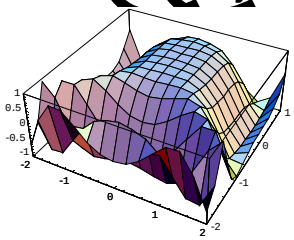
Para nos ajudar no estudo deste tipo de funções o *Mathematica* oferece-nos algumas funções que nos permitem um conhecimento mais completo acerca das mesmas. Através da opção **ViewPoint** podemos especificar o ponto do espaço a partir do qual os objectos representados serão vistos.

Deste modo, aqui ficam algumas coordenadas para **ViewPoint** consoante a posição pretendida:

{0, -2, 0}→ directamente à frente
{0, -2, 2}→ de frente e de cima
{0, -2, -2}→ de frente e de baixo
{-2, -2, 0}→ do canto do lado esquerdo
{2, -2, 0}→ do canto do lado direito
{0, 0, 2}→ directamente de cima
{1.3, -2.4, 2}→ ponto predefinido.

Representaremos a partir deste momento os gráficos num formato mais pequeno, uma vez que o espaço que os mesmos ocupam com a sua dimensão normal é considerável. Sem qualquer indicação obtivemos a seguinte representação.

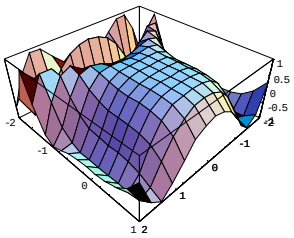
```
Plot3D[Cos[x^2 + y^3], {x, -2, 2}, {y, -2, 1}]
```



```
...SurfaceGraphics...
```

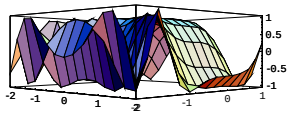
Vejamos agora a mesma representação de outros pontos de vista.

```
Show[%, ViewPoint → {1.2, 1.2, 1.2}]
```



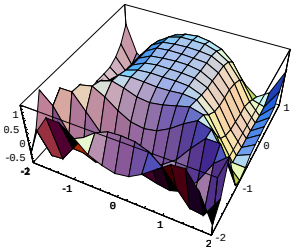
```
...SurfaceGraphics...
```

```
Show[%, ViewPoint → {2, 2, 0}]
```



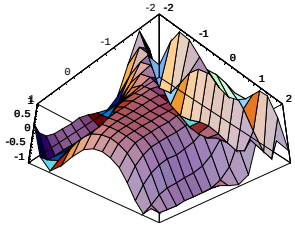
```
...SurfaceGraphics...
```

```
Show [ ..., ViewPoint → ..., 2, 2 ]
```



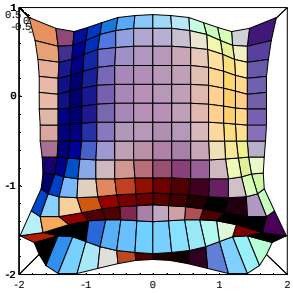
```
...SurfaceGraphics...
```

```
Show [ ..., ViewPoint → ..., - 2, - 2 ]
```



```
...SurfaceGraphics...
```

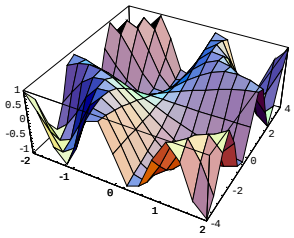
```
Show [ ..., ViewPoint → ..., 2 ]
```



```
...SurfaceGraphics...
```

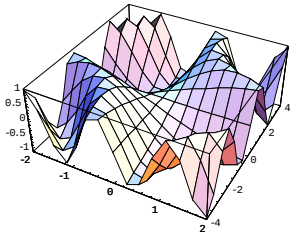
Considerando a seguinte função:

```
Plot3D [ ... ], ..., 2, 2 ], ..., 4, 4 ]
```



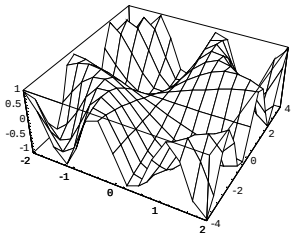
podemos, nós próprios definir o nível de luminosidade com que queremos ver a imagem se procedermos do seguinte modo:

```
Plot3D Sin[x y], {x, -2, 2}, {y, -4, 4}, AmbientLight -> GrayLevel[0.25]
```



...SurfaceGraphics...

```
Plot3D Sin[x y], {x, -2, 2}, {y, -4, 4}, AmbientLight -> GrayLevel[1]
```

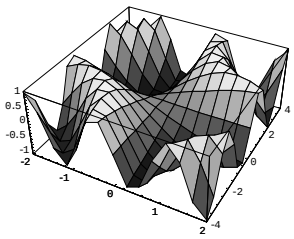


...SurfaceGraphics...

O argumento de **GrayLevel** tem de ser um número compreendido entre 0 e 1 inclusivé, sendo que o valor predefinido é o próprio zero. Assim, à medida que vamos subindo o argumento da função, esta vai tornando-se cada vez mais clara.

Para além da forma anterior é-nos ainda possível pedir ao programa que não ilumine a nossa imagem para o que juntamos a função **Plot3D** e a opção **Lighting→False** que especifica que neste caso não deve ser usado simulador de iluminação na imagem.

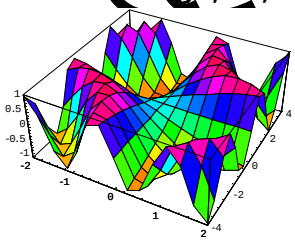
```
Plot3D Sin[x y], {x, -2, 2}, {y, -4, 4}, Lighting → False
```



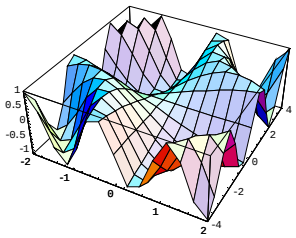
...SurfaceGraphics...

Temos ainda a possibilidade de alterar a forma como a imagem é colorida usando a opção **ColorOutput→tipo_de_cor** onde **tipo_de_cor** poderá ser **Hue**, **CMYKColor**, **RGBColor** ou **GrayLevel**.

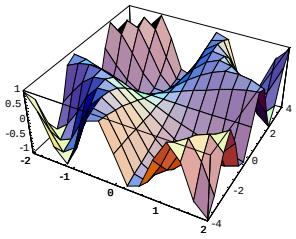
```
Plot3D Sin[x y], {x, -2, 2}, {y, -4, 4}, ColorFunction → Hue
```



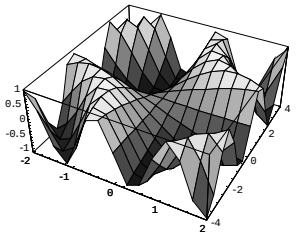
```
...SurfaceGraphics...
Plot3D Sin[x y], {x, -2, 2}, {y, -4, 4}, ColorOutput → CMYKColor
```



```
...SurfaceGraphics...
Plot3D Sin[x y], {x, -2, 2}, {y, -4, 4}, Lighting → True, ColorOutput → RGBColor
```



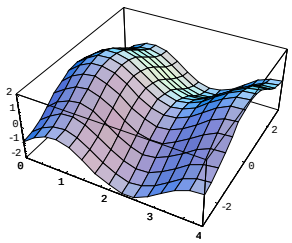
```
...SurfaceGraphics...
Plot3D Sin[x y], {x, -2, 2}, {y, -4, 4}, Lighting → False,
 ColorOutput → GrayLevel
```



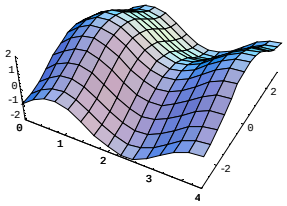
```
...SurfaceGraphics...
```

Tal como acontecia com os programas anteriores o *Mathematica* permite que introduzamos mais algumas opções no respeitante à forma na qual a nossa imagem fica contextualizada. Até aqui a nossa imagem tem vindo sempre inserida numa "caixa" com a forma de um paralelepípedo em perspectiva. Ora, é possível retirar essa caixa fazendo **Boxed→False**.

```
Plot3D Sin[x y], {x, 0, 4}, {y, -3, 3}
```



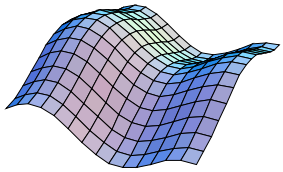
```
...SurfaceGraphics...
Plot3D Sin[x y], {x, 0, 4}, {y, -3, 3}, Boxed → False
```



```
...SurfaceGraphics...
```

Como podemos verificar os eixos permaneceram representados. Os mesmos podem contudo ser retirados fazendo **Axes→False**.

```
Plot3D Sin[x y], {x, 0, 4}, {y, -3, 3}, Boxed → False, Axes → False
```

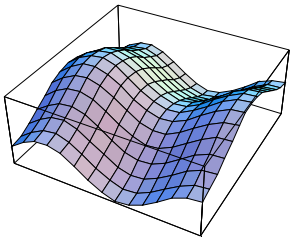


```
...SurfaceGraphics...
```

Passamos a dispor de uma representação gráfica isolada, isto é, sem qualquer entrave à observação directa da função. Perdemos alguma informação e ganhamos um pouco mais de visibilidade.

Também é possível retirar os eixos sem retirar a caixa como se pode ver já de seguida.

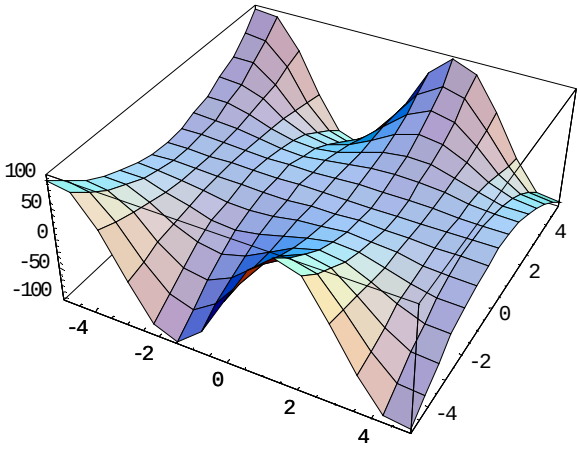
```
Plot3D Sin[x y], {x, 0, 4}, {y, -3, 3}, Axes → False
```



```
...SurfaceGraphics...
```

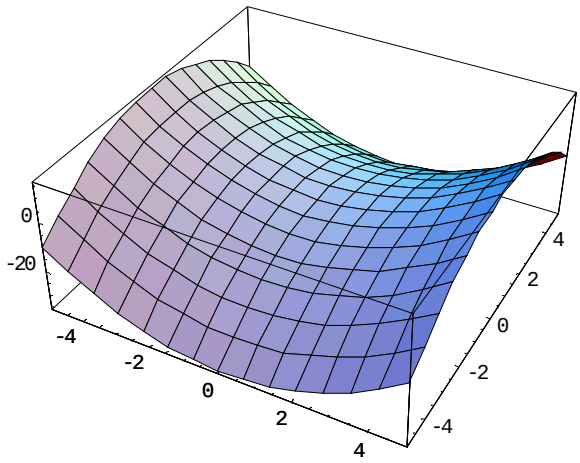
Para representar simultaneamente mais do que uma função de duas variáveis temos de utilizar a função **Show**. Aqui por questões de simplicidade fizemos duas atribuições.

```
p = Plot3D Sin[x^2 + y^2], {x, -5, 5}, {y, -5, 5}
```



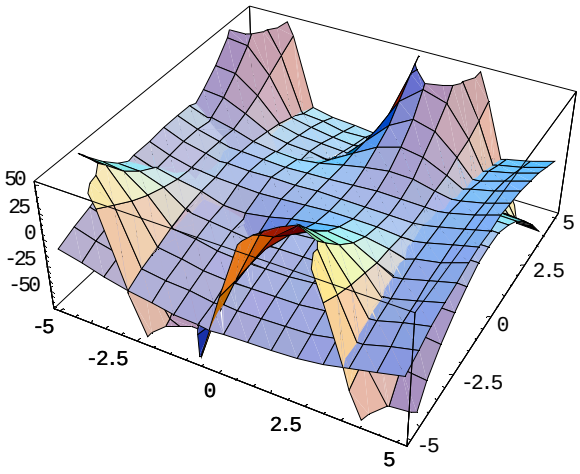
...SurfaceGraphics...

```
q = Plot3D[x^2 + y^2 - 10, {x, -5, 5}, {y, -5, 5}]
```



...SurfaceGraphics...

```
Show[p, q]
```



...Graphics3D...

6.3. - ANIMAÇÃO DE GRÁFICOS

Podemos tirar partido das capacidades do *software* com que aqui temos trabalhado para apresentar gráficos que não estão estáticos, ou seja, gráficos em que é possível dar a noção de movimento à medida que se altera um dos parâmetros da função representada. Já aqui demos a noção de movimento quando falámos de gráficos de funções de duas variáveis mas, nessa altura o único movimento que era feito relacionava-se com o ponto do espaço de onde estávamos a observar a imagem. Tratava-se pois sempre da mesma figura que, vista de pontos diferentes nos mostrava novos pormenores da figura e escondia outros parecendo em alguns aspectos diferente. Neste caso não é isso que se passa. A figura que temos vai efectivamente mudando estando o observador sempre colocado na mesma posição. Tal é feito através da geração de um conjunto de representações gráficas que depois são mostradas rapidamente tal e qual acontece num filme. Assim o nosso gráfico consegue claramente dar a ideia que se está a movimentar.

Naturalmente dada a limitação do papel não é possível transmitir aqui a noção de movimento, no entanto tal como já vimos para o caso dos gráficos de funções de duas variáveis, vamos apresentar aqui sequências de imagens procurando transmitir aquilo que efectivamente presenciaríamos caso executássemos os *inputs* aqui apresentados. Sugerimos a experimentação dos mesmos para complementar a ideia que aqui tentamos deixar.

Maple:

Vejamos como gerar um gráfico animado de uma função de uma variável.

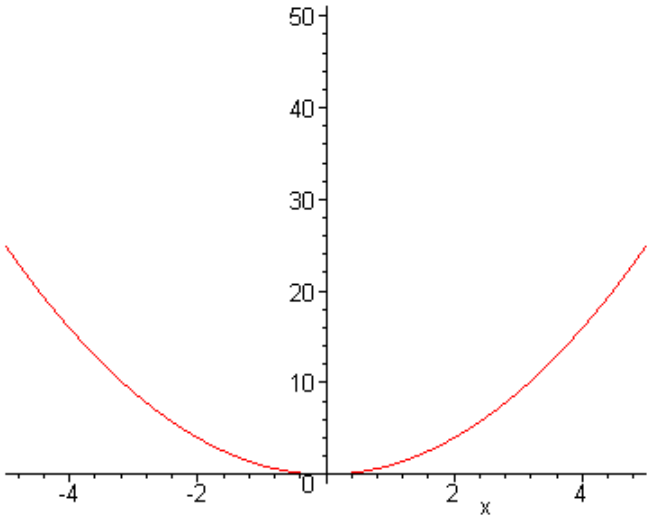
Este processo acarreta algumas vantagens pois permite que rapidamente consigamos analisar o efeito que a variação de um dado parâmetro produz no gráfico da função.

Em primeiro lugar temos de carregar o pacote correspondente.

```
> with(plots):
Warning, the name changecoords has been redefined
```

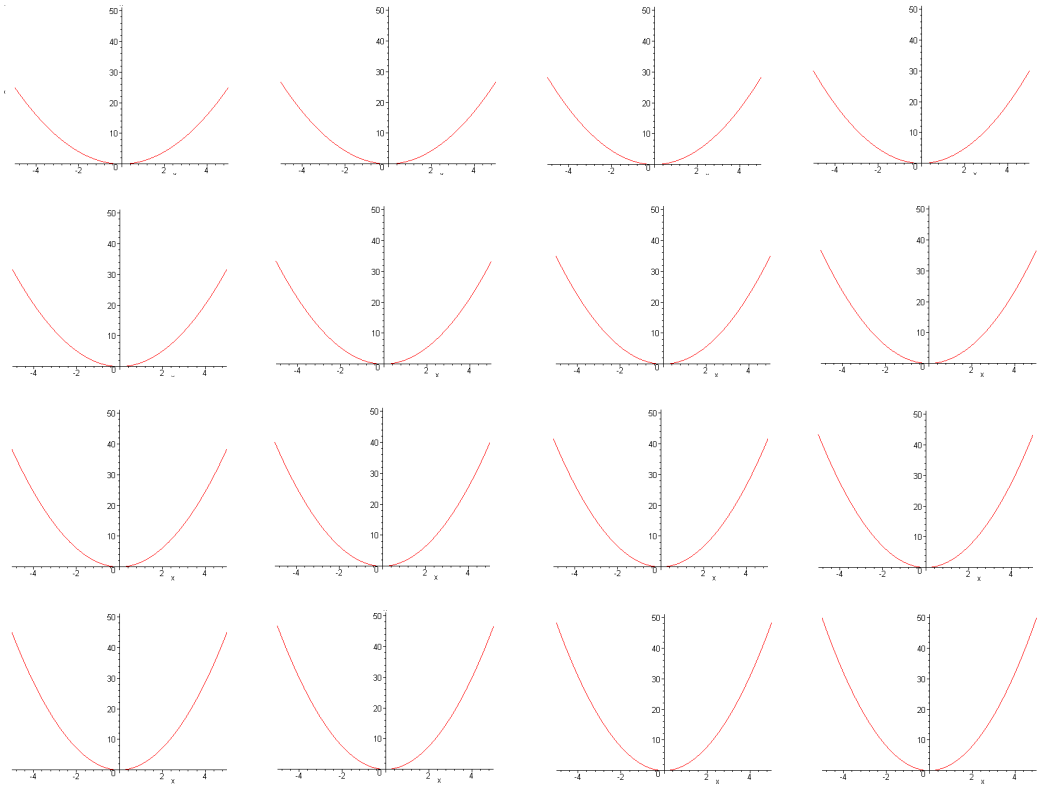
Depois temos de utilizar uma das funções aí definidas, neste caso **animated**. Vejamos como poderíamos estudar o efeito que a alteração do parâmetro **a** tem na função $\mathbf{a\ x^2}$. Mandando avaliar a próxima expressão surge no ecrã a seguinte imagem, que corresponde à primeira das imagens que o *Maple* elaborou.

```
> animate(a*x^2,x=-5..5,a=1..2);
```



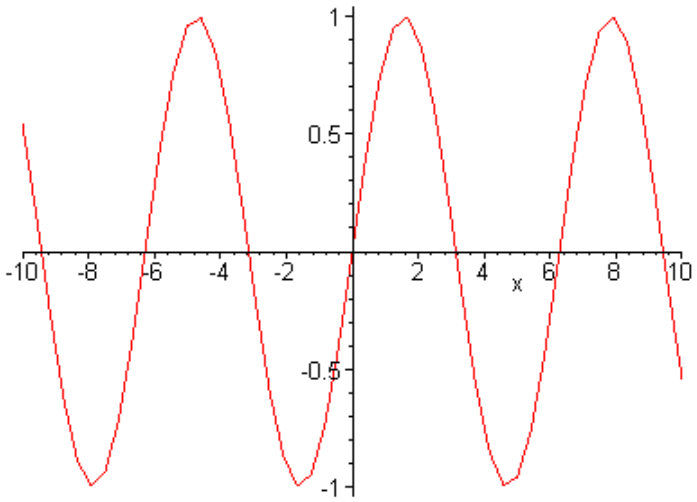
Para ver o movimento que a alteração do valor de **a** provoca, temos de seleccionar o gráfico o que faz com que a barra de menus se altere e que surja um conjunto de atalhos que será muito útil para o que se pretende fazer.

De seguida temos de utilizar o atalho  e imediatamente o *Maple* mostra-nos a sequência de dezasseis imagens que gerou dando-nos a noção de movimento.



Se pretendermos que o *Maple* elabore mais representações podemos acrescentar **frames=número_de_representações _pretendidas**.

```
> animate(sin(x+t),x=-10..10,t=0..3,frames=50);
```

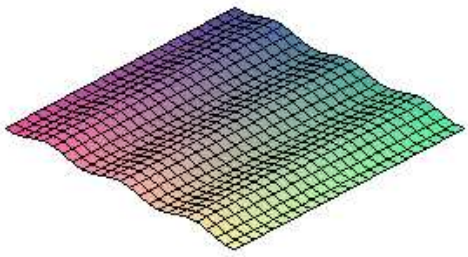


Claro que aqui não nos é possível verificar a melhoria da animação mas, comparando com a situação em que apenas tínhamos dezasseis imagens existem melhorias substanciais não existindo tantos saltos dando a ideia que o movimento é mais contínuo.

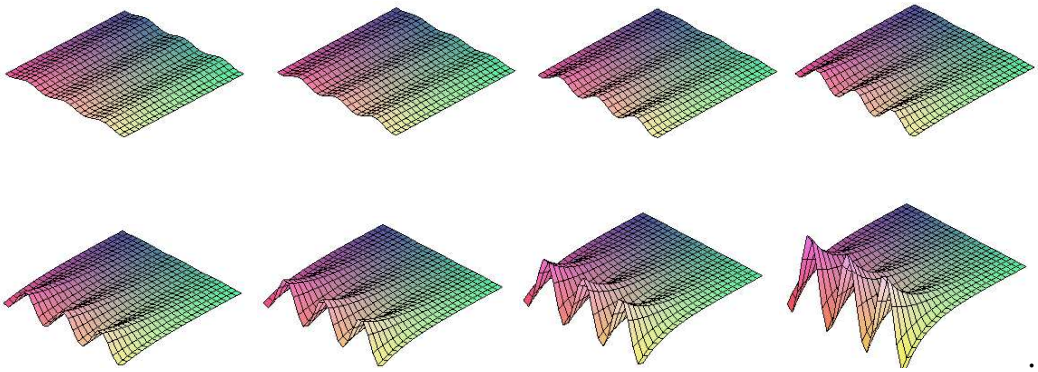
Utilizando o atalho  as imagens que foram entretanto construídas são passadas de forma contínua "saltando" da última imagem para a primeira e repetindo sucessivamente as imagens. Podemos pará-las recorrendo a  , situação em que estamos a dizer que a sequência de imagens só deve ser passada uma vez ou então através de  onde efectivamente paramos mesmo a sequência. Quanto aos atalhos  e  destinam-se a controlar a direcção na qual são mostradas as imagens. É ainda possível controlar a velocidade da animação recorrendo aos atalhos  e  e até mesmo passar uma imagem de cada vez através de  . Todo um conjunto de atalhos que são muito intuitivos quer pela imagem que mostram quer pelo conhecimento que temos dos mesmos da nossa vida quotidiana.

Nas representações de funções de duas variáveis tudo se passa de forma análoga, utilizando-se desta forma a função **animate3d**.

```
> animate3d(cos(y)*exp(t*x),x=-10..10,y=-10..10,t=0..0.3);
```



As oito imagens geradas pelo programa são as seguintes:



Também nesta situação é possível dizer ao *Maple* quantas imagens pretendemos acrescentando por exemplo **frames=16** supondo que queremos o dobro das imagens.

```
> animate3d(cos(y)*exp(t*x),x=-10..10,y=-10..10,t=0..0.3,
  frames=16);
```

A representação que então surge no ecrã, que aqui não foi colocada, é exactamente igual à que obteríamos sem colocar a última opção e que já foi mostrada anteriormente. A qualidade do movimento é que será superior.

De realçar ainda aqui o facto de que, enquanto decorre a animação, é possível ao utilizador modificar, com o auxílio do rato, a posição da imagem tal e qual fizemos quando abordámos a representação gráfica de funções de duas variáveis. Este processo permite que com muita facilidade possamos ver rapidamente e de diversos pontos do espaço a sequência de imagens gerada.

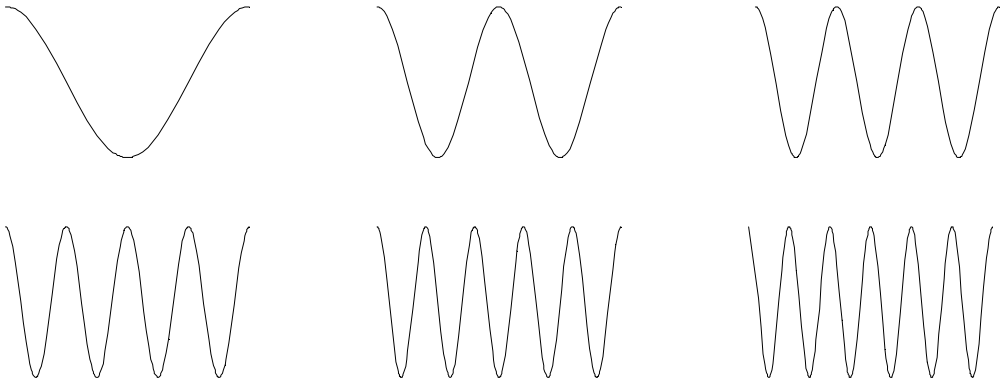
Mathematica:

Estudar o comportamento de certas famílias de funções torna-se mais fácil e mais rápido se pudermos utilizar ferramentas que permitam uma melhor visualização das mudanças operadas em cada uma das funções quando se altera um determinado parâmetro. Tal como vimos atrás, também através do *Mathematica* podemos aceder facilmente a um conjunto de representações que depois o programa irá passar em rápida sequência dando a ideia que o nosso gráfico tem vida própria como se de um filme se tratasse. Para conseguir fazer tudo isto carregamos o pacote **Graphics`Animation`**.

```
<< Graphics`Animation`
```

Convém aqui recordar a função **PlotRange** que limita os valores das ordenadas, impedindo que seja o *Mathematica* a decidir directamente a janela mais conveniente para apresentar o gráfico. Assim ao pretendermos fazer uma animação devemos utilizar esta função pois é do nosso interesse que os valores máximos e mínimos das abcissas e ordenadas se mantenham constantes ao longo de todas as imagens representadas. Caso assim não seja, serão visíveis distorções que dificultam e por vezes impossibilitam o estudo correcto de alguns aspectos das funções analisadas. Não quer isto dizer que seja sempre assim, o exemplo que apresentamos de seguida não necessita da função **PlotRange**, isso fica a dever-se ao facto de neste caso concreto os valores máximos e mínimos das funções representadas durante este processo ser sempre **1** e **–1**, daí que a janela que o *Mathematica* nos dá serve perfeitamente.

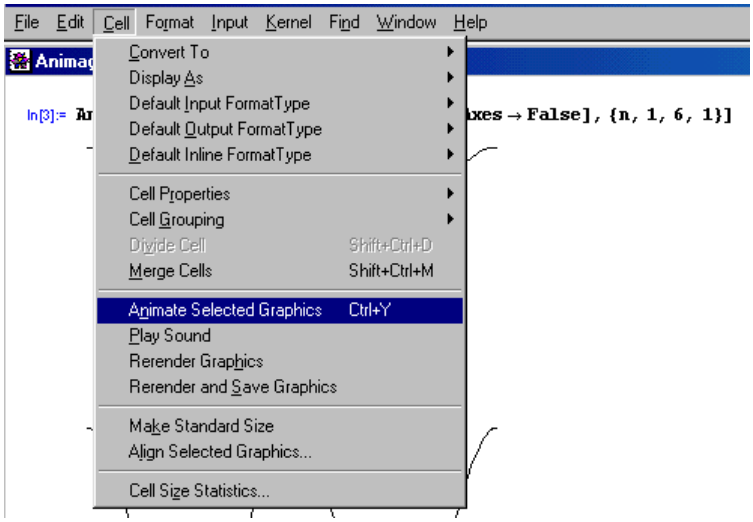
```
Animate Plot Cos x, , 2* Pi , Axes® False , , 6, 1
```



Note-se que as representações gráficas não surgem no local de trabalho do *Mathematica* como apresentamos em cima mas na vertical como aliás é usual. Aqui o que fizemos foi dar-lhes outra disposição e reduzi-los.

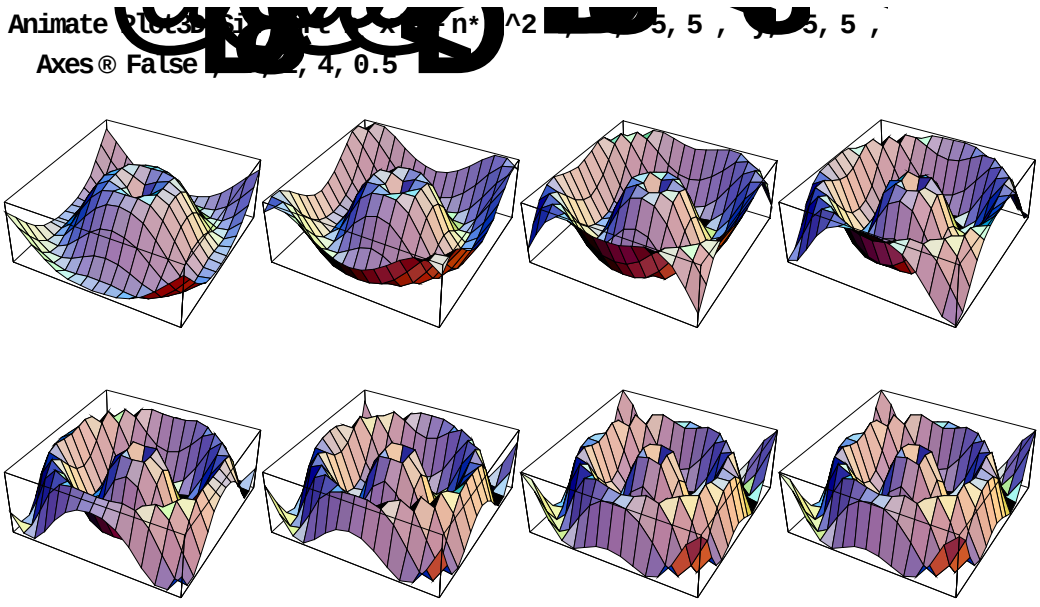
São os gráficos anteriores que são passados sucessivamente dando a ideia de que estamos perante uma movimentação do gráfico. Ainda antes de continuar com a explicação da forma como se pode efectivamente ver a animação, já que até agora apenas foi mostrado como gerar o conjunto de gráficos, convém esclarecer que ao escrevermos **{n,1,6,1}** estamos a dizer ao programa que os valores que **n** deve assumir são os que vão desde um até seis e avançando uma unidade de cada vez. Se não especificássemos o comprimento do passo a dar seriam representadas vinte e quatro imagens o que resultaria numa animação mais eficaz.

Para animar o conjunto de imagens que obtivemos através da função **Animate** temos de seleccionar o conjunto dos gráficos. Tal é feito facilmente usando as marcas azuis à direita que separam os vários *inputs* e *outputs*. Estando os gráficos seleccionados recorremos ao menu *Cell* e ao submenu *Animate Selected Graphics*.



Seguindo o processo exemplificado, numa das imagens que normalmente é a primeira do topo começa a passar a sequência de imagens seleccionadas. As imagens vão repetindo-se sucessivamente até que o utilizador efectue qualquer movimento na sessão em causa. Como podemos ver na figura é possível dispensar a utilização do menu e submenu respectivo, bastando para isso seleccionar as imagens e depois fazer uso do atalho **Ctrl+Y**.

Para as funções de duas variáveis tudo se passa de modo inteiramente semelhante.



6.4. - CONCLUSÃO

Não estudámos as diversas possibilidades de produzir gráficos usando outras possibilidades (sistemas de coordenadas, por exemplo) pelo que a comparação que foi feita é apenas sobre a qualidade gráfica de gráficos simples que, de qualquer forma, são os que nos interessam nesta tese.

A representação de funções quer de uma quer de duas variáveis no *Derive* faz-se de um modo diferente que nos restantes programas. No *Derive* existe uma janela especial para fazer as representações das funções não ficando as mesmas à priori representadas na zona de trabalho, embora tal também se possa fazer. No *Maple* e no *Mathematica* as representações são apresentadas imediatamente após a avaliação das respectivas funções predefinidas – **plot** ou **plot3d** conforme o caso.

Relativamente aos gráficos das funções de uma variável há que referir que os atalhos que o *Derive* e o *Maple* apresentam facilitam a representação das funções na forma pretendida sendo o primeiro o que mais vantagens apresenta neste aspecto.

Quanto aos gráficos de funções de duas variáveis o *Derive* e o *Maple* apresentam-se melhor equipados que o *Mathematica*. Isto é particularmente evidente quando depois de representada uma dada função queremos vê-la de um outro ponto de vista. Aí no *Mathematica* é necessário mandar avaliar novamente a expressão e indicar o ponto do espaço de onde a superfície deve ser vista. No *Derive* e no *Maple* é muito mais fácil pois existem atalhos que permitem que modifiquemos a imagem como desejamos sendo possível fazer a imagem rodar em qualquer sentido. No *Maple* há ainda a possibilidade de utilizar o rato para o mesmo efeito, procedimento este que consideramos muito eficaz.

A animação de gráficos pode ser feita no *Maple* e no *Mathematica* sendo que no primeiro as imagens que são geradas pelo programa são ocultadas do utilizador até que seja feita uma utilização das mesmas. No *Mathematica* as imagens são todas apresentadas em sequência. Para facilitar o tratamento da animação o *Maple* disponibiliza alguns atalhos que permitem controlar de uma forma mais eficaz que no M*athematica* a animação produzida.

Em resumo:

- As representações gráficas no *Derive* são feitas numa janela à parte.
- Os atalhos disponibilizados pelo *Maple* e em especial pelo *Derive* tornam mais fácil a representação gráfica de funções de uma variável.
- Nas representações gráficas de duas variáveis o *Derive* e o *Maple* estão melhor apetrechados que o *Mathematica*.
- A animação de gráficos é possível no *Maple* e no *Mathematica* sendo que no primeiro existem atalhos que permitem controlar a animação de forma mais eficaz que no segundo.

Salientamos que os três últimos aspectos são os mais relevantes. O primeiro ponto apresenta-se como uma diferença que na realidade produz os mesmos efeitos. O *Maple* é o programa em que é mais fácil obter e manipular uma representação gráfica seguindo-se o *Derive*. Neste último não é possível obter uma animação de gráficos.

CONCLUSÃO

Ao longo deste trabalho foram feitas conclusões que destacaram o comportamento dos programas em cada um dos temas em estudo. As mesmas permitem-nos concluir que em muitos casos não existem grandes diferenças. Existe um programa que tem mais uma ou outra função predefinida num determinado assunto mas, o mesmo programa tem noutro aspecto menos uma ou duas funções que os restantes.

Analisando com cuidado as conclusões facilmente se constrói a ideia que os programas equivalem-se em muitas situações. De qualquer forma faremos ainda aqui mais algumas considerações salientando aquelas situações em que cada um dos programas se destaca.

No âmbito do contacto inicial parece-nos que o *Derive* apresenta vantagem em relação aos restantes pois a sua barra de menus e atalhos é mais completa que nos dois outros programas. Tendo em conta que da primeira vez que utilizamos qualquer dos programas não conhecemos a linguagem própria de cada um este é um aspecto importante.

Em relação à programação o *Mathematica* destaca-se claramente do *Derive* e do *Maple*. Esta diferença é mais acentuada sobretudo em relação à programação funcional e programação por regras de reescrita. Os argumentos que o *Mathematica* apresenta neste domínio permitem-lhe olhar de cima para os restantes programas. Comparando o *Maple* e o *Derive* podemos dizer que ao nível da programação imperativa as funções que o primeiro apresenta são mais eficientes que aquelas que são disponibilizadas pelo segundo.

A representação gráfica de funções está mais desenvolvida no *Maple* e no *Derive* que no *Mathematica*. O *Derive* destaca-se nas funções de uma variável enquanto que o *Maple* evidencia-se nas funções de duas variáveis. A qualidade das representações é superior no *Maple* e no *Derive*.

Na animação de gráficos o *Maple* continua a superiorizar-se em relação ao *Mathematica* pois neste (no *Maple*) é mais fácil manipular a animação. No *Derive* não é possível fazer animação de gráficos.

Existem ainda outras diferenças de pormenor que referimos já de seguida.

No *Derive* as dízimas são valores exactos enquanto que no *Maple* e no *Mathematica* estas são sempre aproximações.

No *Mathematica* a atribuição paramétrica diferida permite guardar expressões em variáveis de uma forma distinta da dos restantes programas. A definição de funções bem como guardar expressões em variáveis é possível em qualquer programa.

No *Mathematica* não está disponível a estrutura conjunto mas, estão disponíveis muitas das operações que incidem sobre conjuntos sendo estes representados por listas.

Quer o *Derive*, quer o *Maple*, quer o *Mathematica* possuem diversos meios para trabalhar com polinómios não se destacando qualquer um deles, o mesmo acontecendo em relação à área do cálculo vectorial e matricial.

Em suma:

As diferenças mais acentuadas relacionam-se com as representações gráficas e com a programação. No primeiro caso são melhores o *Maple* e o *Derive*, no segundo destaca-se o *Mathematica*.

A escolha de qualquer um destes programas em detrimento dos outros passará fundamentalmente por saber de antemão o objectivo que se pretende alcançar com este *software*.

Certamente foi possível notar ao longo deste trabalho que existem áreas em que um dos programas é mais eficaz que os outros. Cada leitor/utilizador deverá analisar cuidadosamente as capacidades de cada programa em cada um dos campos em que trabalha e só depois decidir.

Pensamos ter conseguido atingir o objectivo fundamental que era comparar estes três programas. A forma como o trabalho foi organizado quanto a nós permitiu que o leitor decidisse por si só o programa que está mais capaz de satisfazer aquilo que cada um procurava.

Esperamos que de algum modo tenhamos contribuído para tornar mais real o desejo de que a Matemática esteja cada vez mais ao alcance de todos.

APÊNDICE

Por não ser esse o objectivo da tese, não se reservou nenhum capítulo especial para tratarmos a edição de texto. Sobre este assunto podemos referir que o *Derive*, o *Maple* e o *Mathematica* permitem intercalar texto com células de *input* e *output*. Embora em matéria de edição de texto qualquer um dos sistemas fique muito atrás de programas que foram especialmente criados e desenvolvidos para esse efeito, temos que, dentro de alguns dos sistemas em análise já é possível realizar algumas formatações e alterações no respeitante ao tratamento de textos. Vamos então analisar separadamente alguns aspectos relacionados com a edição de texto em cada um dos três sistemas em análise.

Derive:

No *Derive* a situação não é tão vantajosa como nos restantes sistemas pois as "ferramentas" disponibilizadas não são tão versáteis nomeadamente no respeitante às opções para formatação do tipo de letra. Quer se trate de um título ou do desenvolvimento de um determinado assunto, a letra é apresentada sempre do mesmo modo ficando o título ou as frases sempre alinhadas à esquerda. Para a introdução de uma "célula" de texto é necessário recorrer ao menu *Insert* e ao submenu *Text Object* ou ao atalho equivalente (F5) e depois introduzir o texto pretendido. De seguida mostramos o aspecto do ambiente de trabalho quando utilizamos simultaneamente células de *input* e *output* e células de texto.

Resolução da equação quadrática

Utilizando o Derive é possível obter imediatamente as soluções de uma equação de segundo grau qualquer.
Já de seguida iremos analisar um desses exemplos tendo nesse caso oportunidade de notar a economia de tempo e de cálculos que esse método permite. Analisemos pois as soluções da equação x^2+5x+6=0.

#1: $\text{SOLVE}(x^2 + 5 \cdot x + 6 = 0,\ x)$

#2: $x = -3 \vee x = -2$

Contudo os casos numéricos não se ficam por aqui. Mesmo quando estamos perante uma equação que apenas possuí raízes não reais o Derive dá-nos as soluções rapidamente.
Utilizando o Derive é possível obter imediatamente as soluções de uma equação de segundo grau qualquer.

#3: $\text{SOLVE}(x^2 + 3 \cdot x + 17 = 0,\ x)$

#4: $x = -\frac{3}{2} - \frac{\sqrt{59} \cdot i}{2} \vee x = -\frac{3}{2} + \frac{\sqrt{59} \cdot i}{2}$

É ainda possível descobrirmos qual a fórmula resolvente para as equações do segundo grau. Para isso temos de pedir ao Derive que resolva uma equação arbitrária.

#5: SOLVE(a·x^2 + b·x + c = 0, x)

#6: $x = \frac{\sqrt{(b^2 - 4 \cdot a \cdot c)} - b}{2 \cdot a} \vee x = -\frac{\sqrt{(b^2 - 4 \cdot a \cdot c)} + b}{2 \cdot a}$

#7: SOLVE(a·x^2 + b·x = 0, x)

#8: $x = -\frac{b}{a} \vee x = 0$

Maple:

No *Maple* também é possível editar texto. Para isso recorremos ao atalho T presente na barra de ferramentas ou deslocamos o cursor para o lado esquerdo do símbolo > que surge após mandarmos avaliar cada expressão. Ao fazermos isto, no topo da janela surgem opções que nos permitem definir as características do texto a criar. Se inicialmente está seleccionada a opção *Normal* podemos sempre alterá-la por exemplo para *Text Output* ou *Title* entre muitas outras. Para além do aspecto anterior podemos ainda definir o tipo e o tamanho da letra bem como destacar parte ou a totalidade do texto colocando-o a negro, a sublinhado ou a itálico. Podemos formatar a forma como o texto é colocado na zona de trabalho seleccionando os atalhos correspondentes ao alinhamento à direita, ao centro, à esquerda e justificado tal e qual procederíamos se estivéssemos no *Word*. Ao intercalar "células" de *input* com células de *output* o *Maple* apresenta a vantagem da cor distinguir claramente os *inputs* e *outputs* do texto.

O exemplo anterior no *Maple* ficaria como se segue. Na imagem mantivemos o topo da janela de modo a que o leitor possa mais facilmente visualizar as opções de que dispõe para formatação do texto.

Resolução da equação quadrática

Utilizando o *Maple 6* é possível obter imediatamente as soluções de uma equação de segundo grau qualquer. Já de seguida iremos analisar um desses exemplos tendo nesse caso oportunidade de notar a economia de tempo e de cálculos que esse método permite. analisemos pois as soluções da equação x^2 +5x+6=0.

```
> solve(x^2+5*x+6=0,x);
```

-2, -3

Contudo os casos numéricos não se ficam por aqui. mesmo quando estamos perante uma equação que apenas possui raízes não reais o *Maple* dá-nos as soluções rapidamente.

```
> solve(x^2+3*x+17=0);
```

$$-\frac{3}{2}+\frac{1}{2}I\sqrt{59}, -\frac{3}{2}-\frac{1}{2}I\sqrt{59}$$

É ainda possível descobrirmos qual a fórmula resolvente para as equações do segundo grau. Para isso temos de pedir ao *Maple* que resolva uma equação do segundo grau arbitrária.

```
> solve(a*x^2+b*x+c=0,x);
```

$$\frac{1}{2}\frac{-b+\sqrt{b^2-4ac}}{a}, \frac{1}{2}\frac{-b-\sqrt{b^2-4ac}}{a}$$

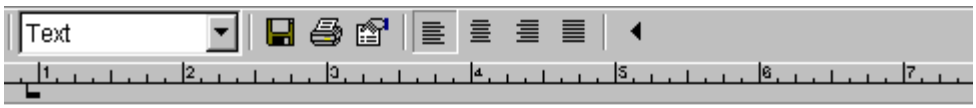
```
> solve(a*x^2+b*x=0,x);
```

$$0, -\frac{b}{a}$$

Mathematica:

O *Mathematica* também dispõe da possibilidade de escolher a cor com que as letras são apresentadas quer sejam um *input* quer sejam um texto. Através do menu *Format* e do submenu *Show Toolbar* surge-nos no topo da janela uma barra de ferramentas, à qual já fizemos referência no inicio deste trabalho, e que permite que definamos a célula que estamos a criar como sendo de texto (*Text*), um título (*Title*), um subtítulo (*Subtitle*) entre outras. Para além disso também na barra de ferramentas surgem os usuais atalhos para alinhar o texto à direita, à esquerda, ao centro e justificado. Ainda dentro do menu *Format*, o *Mathematica* põe ao nosso dispor outros argumentos como sejam a régua que surge mediante a selecção do submenu *Show Ruler*, o tipo de letra (submenu *Font*), o tamanho da letra (submenu *Size*) e a formatação da letra para negrito, itálico ou sublinhado (submenu *Face*). Para além destes aspectos e que aqui salientamos existem ainda outros dentro do mesmo menu que fazem com tenhamos uma grande liberdade de acção quando pretendemos escrever artigos sem abandonar o *Mathematica*.

Vejamos também aqui o exemplo já atrás referido.



Resolução da equação quadrática

Utilizando o *Mathematica* é possível obter imediatamente as soluções de uma equação de segundo grau qualquer. Já de seguida iremos analisar um desses exemplos tendo nesse caso oportunidade de notar a economia de tempo e de cálculos que essa forma permite. Analisemos pois as soluções da equação $x^2+5x+6=0$.

```
Solve[x^2 + 5 x + 6 == 0, x]
```

```
{{x → -3}, {x → -2}}
```

Contudo os casos numéricos não se ficam por aqui. Mesmo quando estamos perante uma equação que apenas possui raízes não reais o *Mathematica* dá-nos as soluções rapidamente.

```
Solve[x^2 + 3 x + 17 == 0, x]
```

$$\left\{\left\{x \to \frac{1}{2}\left(-3 - i\sqrt{59}\right)\right\}, \left\{x \to \frac{1}{2}\left(-3 + i\sqrt{59}\right)\right\}\right\}$$

É ainda possível descobrirmos qual a fórmula resolvente para as equações do segundo grau. Para isso temos de pedir ao *Mathematica* que resolva uma equação arbitrária.

```
Solve[a x^2 + b x + c == 0, x]
```

$$\left\{\left\{x \to \frac{-b - \sqrt{b^2 - 4\,a\,c}}{2\,a}\right\}, \left\{x \to \frac{-b + \sqrt{b^2 - 4\,a\,c}}{2\,a}\right\}\right\}$$

```
Solve[a x^2 + b x == 0, x]
```

$$\left\{\{x \to 0\}, \left\{x \to -\frac{b}{a}\right\}\right\}$$

BIBLIOGRAFIA

CARMO, J.; A. SERNADAS; C. SERNADAS; F. M. DIONÍSIO & C. CALEIRO – *Introdução à Programação em Mathematica*, IST Press, 1999

HEAL, K. M.; M. L. HANSEAN & K. M. RICKARD – *Maple V Learning Guide*, Springer 1996

KUTZLER, BERNHARD & VLASTA KOKOL_VOLJC – *Introduction to Derive 5*, 1ª edição 2000

MONAGAN, M. B.; K. O. GEDDES; K. M. HEAL; G. LABAHN & S.M.VORKOETTER – *Maple V Programing Guide*, Springer, 1996

WOLFRAM, STEPHEN – *The Mathematica Book*, Wolfram Media, 3ª edição 1996.

Site da Internet:

http://www.mapleapps.com/App_Center_Tour/frame_main.html