Ц 840д
К-81

Учебно-
методические
пособия
Учебно-научного
центра ОИЯИ
Дубна

УНЦ-2000-8

R.Kragler*

Mathematica Tutorial Course

*FH Ravensburg-Wengarten, University of Applied Sciences, Germany

2000

Ц 840д
К-81

154095

R.Kragler*

Mathematica Tutorial Course

* FH Ravensburg-Weingarten, University of Applied Sciences,Germany

БИБЛИОТЕКА
ОИЯИ

Краглер Р.
Вводный курс по системе *Математика*

В настоящее издание включены три лекции, прочитанные по системе *Математика* в УНЦ ОИЯИ.

В первой лекции дан обзор основных возможностей системы *Математика.* Подробно обсуждается применение системы для численных и символьных вычислений, графики и программирование на языке системы. Особое внимание уделено новым возможностям *Математики 3.0* таким, как создание интерактивных документов.

Во второй лекции основное внимание уделено стилям программирования в системе *Математика.* Демонстрируются возможности языка высокого уровня системы и его использование в различных стилях программирования. Изложение иллюстрируется многочисленными примерами.

Третья лекция содержит введение в пользовательский интерфейс системы *Математика*. Приведены примеры генерации кнопок и палитр, вращающихся иконок, гиперссылок и т.п.

Лекции прочитаны в Учебно-научном центре ОИЯИ профессором Р.Краглером в рамках цикла лекций «Современные проблемы естествознания».

R.Kragler
***Mathematica* Tutorial Course**

Included are three lectures on the *Mathematica* system given at the University Centre of JINR.

The first lecture reviews the main features of *Mathematica*. Treated in detail are the application of the system for numerical and symbolic calculations, graphics, and programming in the language of the system. Special attention is paid to the new capabilities of *Mathematica* 3.0 like creating interactive documents.

The second lecture deals mostly with the styles of programming in *Mathematica.* The capabilities of the high-level language of the system and its application in different styles of programming are shown. The narration is amply illustrated with examples.

The third lecture is an introduction to the user interface of *Mathematica*. Given are examples of generating buttons, palettes, rotating icons, hyperlinks, etc.

The lectures were given by Prof. R. Kragler at the University Centre of JINR within the lecture cycle «Modern Problems of Natural Sciences».

© Объединенный институт ядерных исследований. Дубна, 2000

University Center
Joint Institute of Nuclear Research / Dubna

Mathematica Seminars
October 27-29, 1998

Survey of Mathematica **Features :**

Numerics, Symbolic Computation and Graphics

Prof. Dr. Robert Kragler
FH Ravensburg-Weingarten /University of Applied Sciences
kragler@fh-weingarten.de

Abstract

The lecture will give a survey on ***Mathematica Version 3.0*** and it will be demonstrated how *Mathematica* notebooks can be used as convenient means for exchange of scientific information in the field of Mathematics, Physics and Engineering.
The salient features of *Mathematica* - i.e. *numerical* and *symbolic calculations*, *graphics* and the high-level *programming language* - will be discussed in this lecture in detail online and illustrated by computer animations taken from different areas of science and engineering. Special emphasis is laid upon the new features of *Mathematica* version 3.0 such as *typesetting* and the interactivity of *Mathematica* documents. All examples are particularly chosen to demonstrate how the effectiveness of science resp. engineering education can be improved using CAS tools. Finally, an outlook on further feasible developments of CAS is given.

Some general remarks on CAS

Motivation

In the past logarithmic tables, slide rulers and collections of mathematical formulae (such as the famous "Table for Integrals, Series and Products" of Gradshteyn & Ryzhik) had been indispensible tools for physicists and engineers. At least the first two requisites had been replaced by electronic pocket calculators which in turn is nowadays replaced by more powerful **PCs** and *portable laptop computers*. The computing power available today on one's desk required only one decade ago the usage of *mainframe computers*. There are only few real hard, number-crunching problems in science which cannot be accessed without a *super computer.*

It is generally accepted that one has to resort to a computer in order to do **numeric calculations.**

We are all used to **computer graphics**. The synthetic images generated by computers - (just think of Steven Spielberg's movie *Jurassic Park* with the dinosaurs) - are astonishingly realistic so that the buzz word **virtual reality** is used. However, even the famous domain of *chess game* was recently conquered by computers; on May 16, 1997 the renouned Russian chess word-champion *G. Kasparov* was beaten by Deep Blue, an IBM super computer. This example for **artificial intelligence** demonstrates that customized hard- and software combined with computer power commerically available can effectively be used for solving intricate problems.

Perhaps less spectacular and thus less know to the public is the fact that even **symbolic calculations**, such as manipulation of formulae which usally required *paper and pencil,* are meanwhile done by computers, and as will be shown later faster and more reliable than humans who tend to make errors when performing tedious rewriting and simplifications of terms. Today, the term **computer algebra** is generally used to characterize machine processing of symbolic expressions in mathematics on a computer.

Reasons using CAS

The very reason resorting to CAS is always the fact that calculations necessary for the solution of a problem turn out to be so voluminous that they can no longer be done with paper and pencil. This may even lead to a point where the *quantity* of necessary computations turns over into a *qualitative* distinction : i.e., although a mathematical problem may in principle be solvable by a finite number of steps the amount of steps, however, makes practical execution impossible.

Due to availability of CAS the *qualitative change in mathematics* takes place in several ways : at first CAS will be applied to such problems which could be solved in a conventional way but require a considerable amount of computations.

The following example of the *calculation of integrals with rational function* may illustrate this :

$$\int \frac{x^5}{(x^4+x^2+1)^2}\,dx \quad \text{// FullSimplify}$$

$$\frac{1 - x^2 + \frac{2\,(1+x^2+x^4)\left(\text{ArcTan}\left[\frac{-1+x^2}{\sqrt{3}\,(1+x^2)}\right]-\text{ArcTan}\left[\frac{2+x^2}{\sqrt{3}\,x^2}\right]\right)}{\sqrt{3}}}{6\,(1+x^2+x^4)}$$

Doing these kind of calculations with paper and pencil one is tempted to minimize the effort of computations in using experience with similar problems. As regards to CAS *heuristics* becomes unimportant with respect to the algorithm. In the case of integration the so-called *Risch algorithm* (1969) is the essential new insight to make this automatism possible at all. Algorithms are developed with the aim to solve these kind of problems on a computer by means of CAS.

There is another class of problems for which an algorithm is known, however *for all cases of practical interest the computational expenditures are so high* that it can no longer be done with conventional means. Often there is an interdependence between the development of algorithms and their implementation in CAS.

Possible Applications of CAS

If complicated problems in science and technique have to be solved it goes without saying for scientists and engineers to use computers for *numerical calculations* and in this context it is generally tolerated that there may occur rounding errors due to the way a computer (as a finite machine) works which might doctor the result.

However, in the area of *modelling and simulation of physical phenomena*, before doing an actual computation, some kind of manipulation of equations or transformations into suitable coordinate systems are necessary which were always done by hand. Yet, with the progress in symbolic computations, i.e. powerful CAS available, there is no longer any need to do these tedious, error-prone calculations manually. Instead they can be done much faster and more reliable by computers now.

Obviously, in the field of *science* and *engineering* there are practically no limits for CAS applications. According to the wide range using CAS these systems are successfully applied in following *areas* :

- System Control
- Analysis and Synthesis of Electrical Networks
- Robotics
- Hydro- and Aerodynamics
- Thermodynamics
- Elastomechanics
- Non-linear Dynamics
- Process Data Processing etc.

For *scientists and engineers, students and practitioners* usage of CAS can be a convenient supplement supporting their work on very different levels :

on lowest level when tedious, error-susceptible basic operations have to be performed such as symbolic differentiation and integration, solving of equations etc. ;

on higher level when e.g. systems of differential equations in the context of physical modeling have to be investigated;

on highest level to study the behavior of approximations of partial differential equations as occur in aerodynamics.

In the *process of teaching*, i.e. in ***scientific education***, a well-planed assignment of CAS helps to increase the *attractivity of lectures*. Using computer algebra tools combined with new multimedia techniques will help to improve substantially the content of various lectures, especially from *didactics* point of view. *Visualization* and *animation of mathematical objects* may contribute in a substantial way to improve the acceptance of (weary) lectures. This is possible now due to powerful hardware such as laptop computers along with coloured LCD panels or data projectors available at affordable costs.

Why just *Mathematica* ?

Features of *Mathematica*

Mathematica, ***Maple V*** and ***Macsyma*** are the most wide spread *universal* CA systems in academia and industry which cover a wide spectrum of mathematics.
At our polytechnic both systems are used :
Maple V is mainly used for *research* and *teaching* in natural sciences, in engineering and mathematics. The reason for this limited usage is the extraordinary large extent of approx. 2.500 mathematical functions, the high level of abstraction and the fact that only recently a graphical front end version for Maple V was developed.
Mathematica is not only used in the areas of research and teaching but will be applied in many other fields e.g. in banking etc. Not only the comfortable user front end but also the excellent graphics features contributed to this success.

The preceding, general remarks are to be illustrated in more details : ***Mathematica*** is a CA program for *symbolic manipulation* of formulae mainly. Apart from *numerical* and *symbolic* calculations this computer algebra system is supplemented by the following *salient features* :

- Graphic Interactive User Interface (Front End)
- Comfortable 2D/3D Graphics and Animation
- High-level Programming Language for Developing new Procedures
- Extensive and Extendible Program Library for Mathematical Procedures

In comparison to other CAS an *outstanding feature* of *Mathematica* is its execellent graphics and the kind of worksheets called **Notebooks**. What are the features making *Mathematica* notebooks so remarkable ?!
Mathematica **Notebooks** are *interactive* electronic documents which may contain :
Mathematica **code** (to be executed), **text** (with most DTP features supported such different fonts, colour, mathematical formulae etc.), **graphics** and **animation,** (even in real-time using the add-on program *MathLive*), **sound** which is encoded in PostScript code too, as is graphics.
According to these features there is an interesting aspect of *Mathematica* notebooks :
Today students have at least access to PCs in a computer lab or even own a PC privately on which they can run *Mathematica* resp. *MathReader*. This fact makes *Mathematica* notebooks interesting for the development of *multimedia lecture units*, say in physics or mathematics, or even *student courses for self-learning* which are written like conventional lecture notes by the professor and distributed to the students who will download the files from a central server.
Besides the possibilities in *didactics* the CA system *Mathematica* - due to its ability of symbolic manipulation of formulae and various numerical procedures - incorporates a great potential in solving mathematical problems in very distinct fields of technology and science.

Particularly the following characterization applies to the new *Mathematica* version 3.0 :

Mathematica is not only a Computer Algebra System
Mathematica is rather a System for Technical Computing

Mathematica Notebooks for Exchange of Scientific Information

Mathematica **notebooks** are ordinary *Mathematica* expressions. Therefore they can immediately be *manipulated* by the user without any special setup etc. thus the results given in *Mathematica* notebooks

can easily be checked or altered. *Mathematica* notebooks are organised in terms of *hierarchical cells* the layout of which can be pre-defined in so-called *StyleSheets* which are again *Mathematica* notebooks. Notebooks can even be created by the *Mathematica* kernel as output. There are no limitations as regards to the appearance of *Mathematica* notebooks. Due to the hierarchical cell concept notebooks can be read like books wherein there are table of content, sections and subsections etc. which are accessible after opening the corresponding pages. However, the *advantage* of the electronic document in comparison to conventional books is the *interactivity*, i.e. probing and manipulating the *Mathematica* code incorporated in the notebook. (The very first pioneering book which contained all chapters in terms of *Mathematica* notebooks on a CD-ROM disk was "Exploring Mathematics with *Mathematica*" by Th. Gray and J. Glynn published by Addison-Wesley Publ. Company already in 1991.
But the *real important* new feature in *Mathematica* version 3.0 is the *portability* of *Mathematica* notebooks. *Mathematica* notebooks contain all information in terms of ASCII code. They are *platform independent.* In earlier versions it was always a big problem to convert *Mathematica* notebooks which had been generated, say, under Windows operating system to Unix workstations (OS Solaris), NeXT (OS NeXTSTEP) or Macintosh (Mac OS 7) computers. Especially international characters such as German umlauts (or kyrillic letters) and other special symbols were a tedious problem to cope with in the past because there is no general accepted standard for character encoding. In *Mathematica* V 3.0 this problem is definitely solved. There are *code translation tables* for all *Mathematica* implementations which are provided for a large number of hardware platforms.

Thus *Mathematica* notebooks may be used as mean for *exchange of scientific information* and could become a kind of standard for interactive electronic documents. On a wider scale *Mathematica* notebooks can be provided to academia via Web. There is already the built-in capability to save *Mathematica* notebooks in HTML format so that they be used as WWW pages. In this context Wolfram Research made in its newsletter MATHwire from April '98 two interesting announcements :

> Based on *Mathematica*'s highly successful notebook paradigm, Wolfram Research has developed **Publicon** - a comprehensive *solution for interactive technical publishing.* Allowing the creation of professional-quality technical documents for on-screen, web, and printed use, a free beta version of Publicon can be downloaded from <http://www.publicom.com/> .

HyperDemo

> *Mathematica* becomes the framework for Web Typesetting Technology. The key ideas forming the core of the new **MathML** (*Math*ematical *M*arkup *L*anguage) standard recently ratified by W3C (i.e. the WWW Consortium) come directly from Wolfram Research's typesetting technology. MathML is designed to allow mathematical expressions to be transmitted over the web, preserving the structure needed to do computations with them in *Mathematica.* Information on MathML can be downloades from the web at <http://www.wolfram.com/news/mathml/ > .

In order to read *Mathematica* notebooks it is even not necessary to have *Mathematica* installed on a computer. It suffices to run the browser *MathReader* freely available from Wolfram Research. On the premises that (voluminous) graphics and sound cells are deleted (which can easily be reconstructed with the help of running *Mathematica*) from *Mathematica* notebooks they shrink to reasonable size and may be attached to email messages and be sent via Internet to colleagues all over the world. Thus, worldwide scientific cooperation by means of exchange of *Mathematica* notebooks works in practice already today.

Mathematica's Capabilities in Technical Computing

A closer look to the capabilities of *Mathematica* reveals that it is not only a simple CAS *but* a

System for Creating *Interactive* Documents (*Mathematica* notebooks)

Notebooks are complete interactive document combining text, tables, graphics, sound, calculations, and other elements in terms of cells

Every *Mathematica* notebook is *Mathematica* expression which can therefore be manipulated by the kernel, see e.g. NotebookE.nb

```
dir = subDir;
openNotebookButton["NotebookE.nb", 1, dir]
```

***Mathematica* can be used just like a calculator : type in some questions, and *Mathematica* returns the answers.**

Mathematica can do basic calculations

```
Log[100!] // N
363.739
```

```
∫√x √(1 + x) dx
```

$$\sqrt{1+x}\left(\frac{\sqrt{x}}{4}+\frac{x^{3/2}}{2}\right)-\frac{\text{ArcSinh}[\sqrt{x}]}{4}$$

```
(sole = Solve[x^3 + 2 x^2 + 2 x - 1 == 0, x]) // ColumnForm
```

$$\left\{x\to-\frac{2}{3}-\frac{2}{3}\left(\frac{2}{47+3\sqrt{249}}\right)^{1/3}+\frac{1}{3}\left(\frac{1}{2}\left(47+3\sqrt{249}\right)\right)^{1/3}\right\}$$
$$\left\{x\to-\frac{2}{3}+\frac{1}{3}\left(1+I\sqrt{3}\right)\left(\frac{2}{47+3\sqrt{249}}\right)^{1/3}-\frac{1}{6}\left(1-I\sqrt{3}\right)\left(\frac{1}{2}\left(47+3\sqrt{249}\right)\right)^{1/3}\right\}$$
$$\left\{x\to-\frac{2}{3}+\frac{1}{3}\left(1-I\sqrt{3}\right)\left(\frac{2}{47+3\sqrt{249}}\right)^{1/3}-\frac{1}{6}\left(1+I\sqrt{3}\right)\left(\frac{1}{2}\left(47+3\sqrt{249}\right)\right)^{1/3}\right\}$$

Mathematica can work with formulae of arbitrary length - hence solving problems that are untractable by hand.

```
( x^121 + y^121 ) // Factor
Simplify[%]
```

$$(x+y)\,(x^{10}-x^9y+x^8y^2-x^7y^3+x^6y^4-x^5y^5+x^4y^6-x^3y^7+x^2y^8-xy^9+y^{10})\,(x^{110}-x^{99}y^{11}+x^{88}y^{22}-x^{77}y^{33}+x^{66}y^{44}-x^{55}y^{55}+x^{44}y^{66}-x^{33}y^{77}+x^{22}y^{88}-x^{11}y^{99}+y^{110})$$

$$x^{121}+y^{121}$$

For basic operations one may resort to palettes such as `BasicCalculations.nb` in order to simplify the marked expression in the cell below with `TrigReduce`.

```
dir = $TopDirectory <> "\\SystemFiles\\FrontEnd\\Palettes"
openNotebookButton["BasicCalculations.nb", 1, dir]
C:\PROGRAMME\WOLFRAM RESEARCH\MATHEMATICA\3.0\SystemFiles\FrontEnd\
  Palettes
```

```
3 Sin[α] - 4 Sin[α]^3
```

```
Sin[3 α]
```

Instrument for Extensive Numerical Calculations

Mathematica, can do power computing; one can perform _number-crunching numerical calculations_

Numbers of *arbitrary size*

```
500!
1220136825991110068701238785423046926253574342803192842192413588385845373153881997605496447\
 5022032818630136164771482035841633787220781772004807852051593292854779075719393306037729\
 6085908627042917454788242491272634430567017327076946106280231045264421887878946575477714\
 9863494367781037644274033827365397471386477878495438489595537537990423241061271326984327\
 7457155463099772027810145610811883737095310163563244329870295638966289116589747695720879\
 2692887128178007026517450776841071962439039432253642260523494585012991857150124870696156\
 8141625359056693423813008856249246891564126775654481886506593847951775360894005745238940\
 3357984763639449053130623237490664450488246650759467358620746379251842004593696929810222\
 6397195259719094521782333175693458150855233282076282002340262690789834245171200620771464\
 0979456116127629145951237229913340169552363850942885592018727433795173014586357570828355\
 7801587354327688886801203998823847021514676054454076635359841744304801289383138968816394\
 8746965881750450692636533817505547812864000000000000000000000000000000000000000000000000\
 0000000000000000000000000000000000000000000000000000000000000000000000000
```

and arbitrary *precision* (such as π with 100 digits) can easily be dealt with.

```
N[π, 100]
3.141592653589793238462643383279502884197169399375105820974944592307816406\
 286208998628034825342117068
```

Mathematica uses an state-of-the-art *algorithm for factoring integers.*

```
p = 2^105 - 1
f = FactorInteger[p]
40564819207303340847894502572031
```

```
{{7, 2}, {31, 1}, {71, 1}, {127, 1}, {151, 1}, {337, 1}, {29191, 1},
 {106681, 1}, {122921, 1}, {152041, 1}}
```

With the user-defined function `FactorisationHold` the explicit evaluation of the power products $(p_i)^{e_i}$ is held so that the following neat representation is obtained

```
FactorisationHold[args : {{_, _}..}] :=
      Apply[Times, Apply[Power, # // HoldForm, {2}], {1}]& [args]
FactorisationHold[f]
```

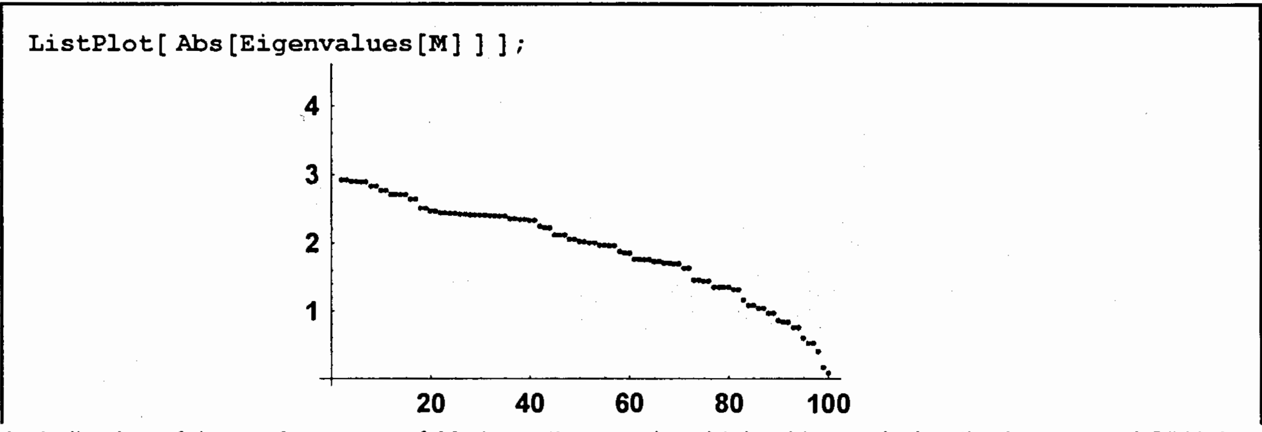
```
7^2 31^1 71^1 127^1 151^1 337^1 29191^1 106681^1 122921^1 152041^1
```

Mathematica can solve numerically a *nonlinear differential equation*. The answer is an *interpolating* function which implicitly represents the whole solution and is displayed as parametric plot.

```
sol =
 NDSolve[ { x''[t] + x[t]^3 == Sin[t] Cos[t], x[0] == x'[0] == 0}, x, {t, 0, 50}];

ParametricPlot[{ x[t], x'[t] } /. sol , {t, 0, 50},
             Compiled → False, PlotStyle → {Red, th}];
```

Mathematica easily calculates the *eigenvalues of a (100 × 100) matrix* with random numbers

```
M = Table[ Random[], {100}, {100}];
```



```
ListPlot[ Abs[Eigenvalues[M] ] ];
```


An indication of the *performance* of *Mathematica* may be obtained by analysing the increase of CPU time for the evaluation of *eigenvalues* of these random matrices with size that increases from n = 50 to 500.

PerformanceE.nb

```
dir = subDir;
openNotebookButton["PerformanceE.nb", 1, dir]
```

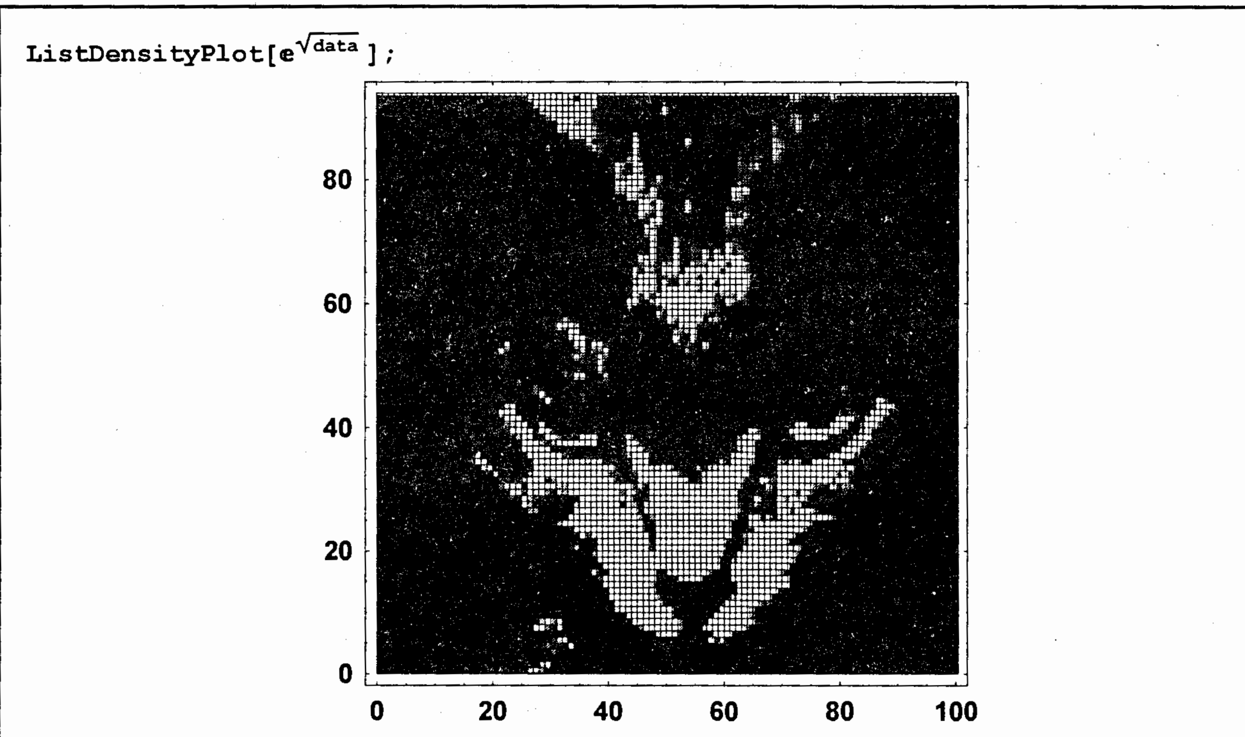
***Mathematica* is able to import data of arbitrary format and process the data with suitable procedures.**

In this example *data* are read from the file `image.dat` as input. (Note : the data file has to be in the working directory)

```
data = ReadList["image.dat", Number, RecordLists → True];
Dimensions[data]
{94, 100}
```

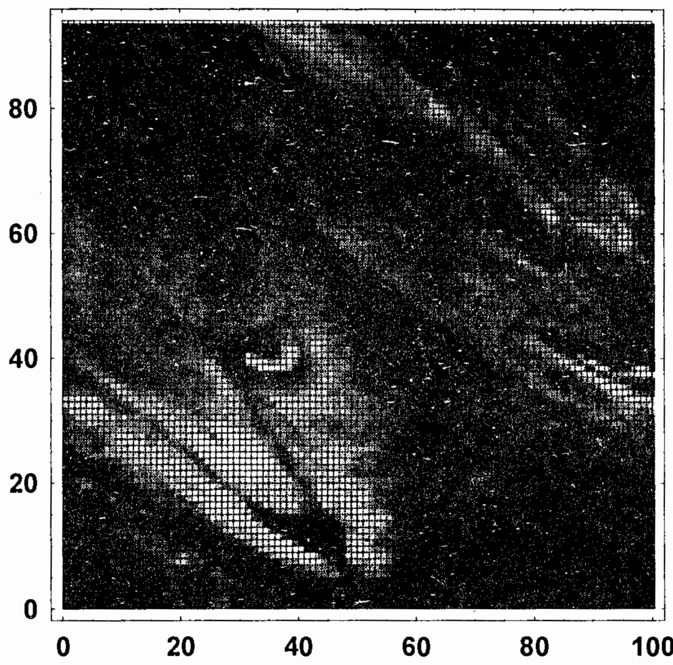
and rendered as a *density plot* with an arbitrary function $e^{\sqrt{x}}$ applied to the data given

```
ListDensityPlot[e^√data];
```



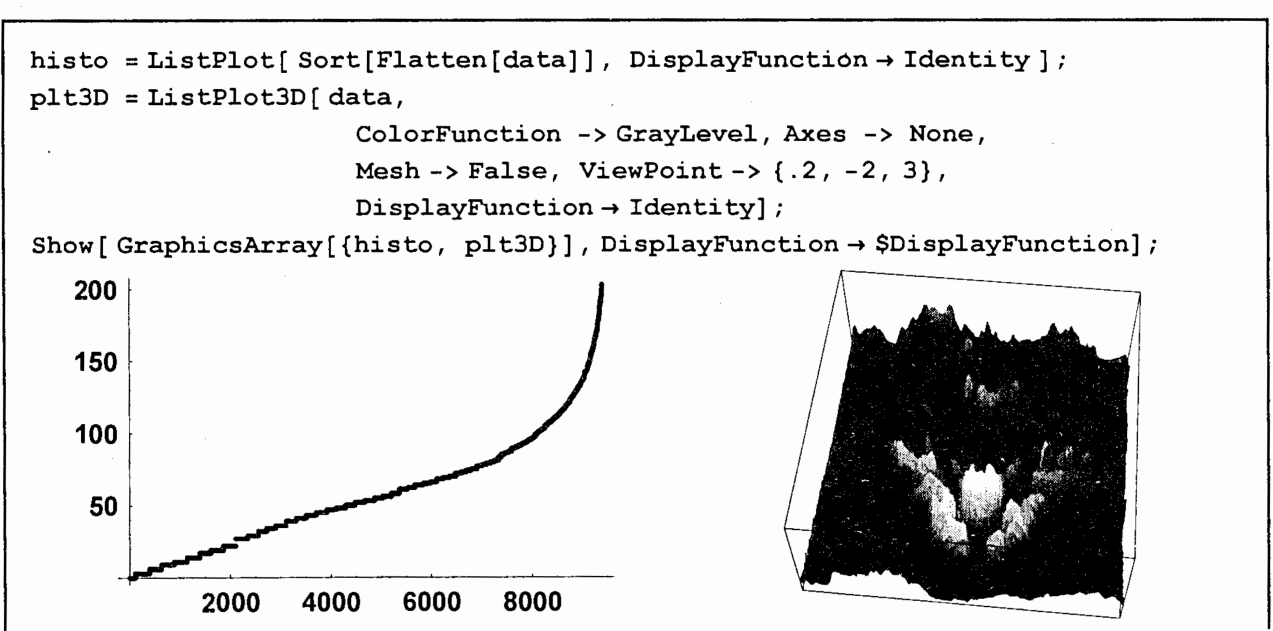
Then, data are shifted to the right in a *cylic* manner. Successive shift of pixels shows the well-known phenomenon of *Poincaré recurrence.*

```
mIR := MapIndexed[RotateRight, #]&
ListDensityPlot[ Nest[mIR, data, 99] ];
```



The histogram, i.e. the *distribution of gray levels* in the pixel data, and a *3D representation* based on the same data is shown below.

```
histo = ListPlot[ Sort[Flatten[data]], DisplayFunction → Identity ];
plt3D = ListPlot3D[ data,
                    ColorFunction -> GrayLevel, Axes -> None,
                    Mesh -> False, ViewPoint -> {.2, -2, 3},
                    DisplayFunction → Identity];
Show[ GraphicsArray[{histo, plt3D}], DisplayFunction → $DisplayFunction];
```



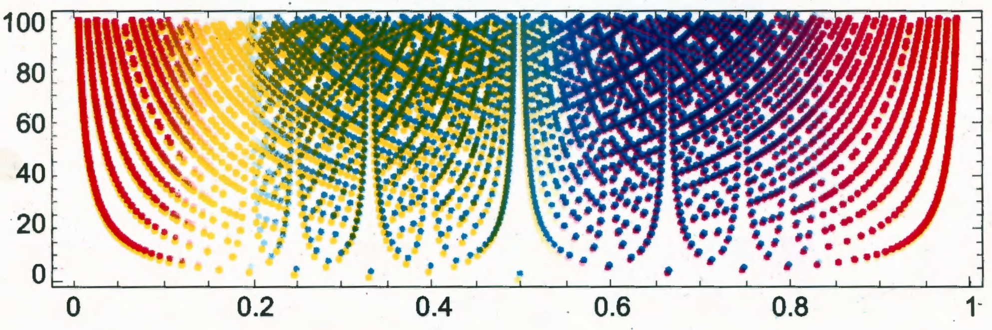
It is quite easy to create stunning visual images with *Mathematica*.

Mathematica includes *graphics primitives* from which one can build up 2D resp. 3D graphics of any complexity. Here a list of point primitives is generated.

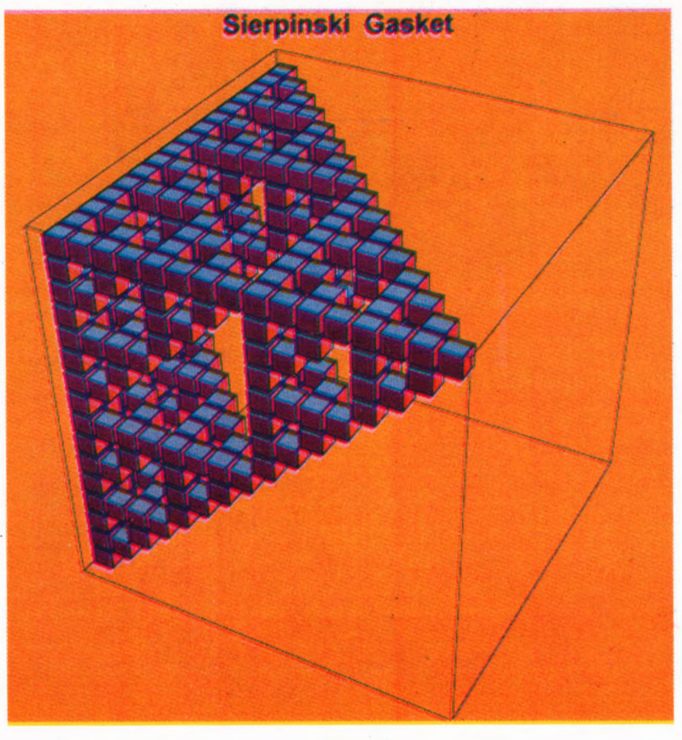
```
gr =
  Flatten[Table[{Point[{p/q, Denominator[p/q]}], Hue[p/q]}, {q, 100}, {p, q - 1}]];
```

The graphics corresponding to this list of point primitives is shown below

```
Show[Graphics[gr, Frame → True, AspectRatio → .3 ]];
```

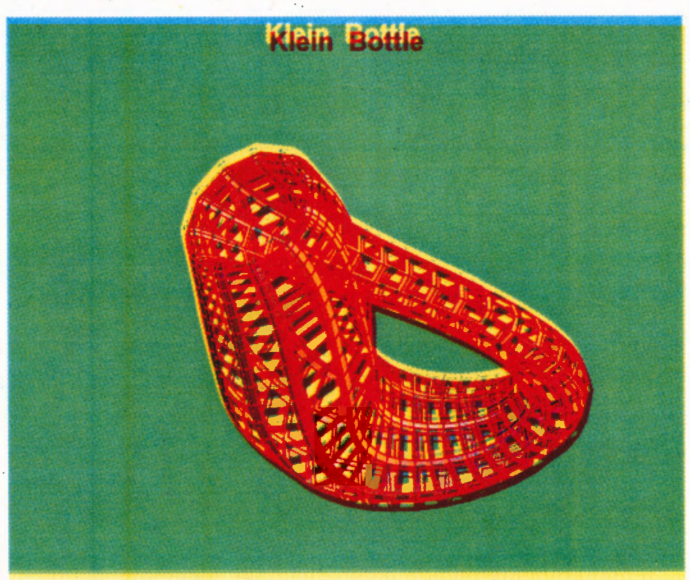


This is a generalization in 3D of the so-called *Sierpinski gasket*, which is obtained if cuboids are substituted in 3D space instead of points in the 2D plane .



```
g = Table[ If[ Mod[ Multinomial[x, y, z], 2] == 1,
            Cuboid[1.2 {x, y, -z}], {} ],
        {x, 0, 15}, {y, 0, 15}, {z, 0, 15}] // Flatten ;
sierpinski = Show[Graphics3D[g],
                Background → RGBColor[1., 0.8, 0.4],
                PlotLabel → "Sierpinski  Gasket"];
```

The so-called *Klein Bottle* (a generalization of the Möbius strip in 2D) is a 3-dimensional object with *non-orientable* surface, i.e. the object has a closed surface but one cannot distinguish between interior and external faces, there is only one single surface.



The object is defined as a *parametric surface* which is obtained by decorating a droplet-like space curve with a tube.

```
Needs["Graphics`ParametricPlot3D`"]; v =. ;
Needs["Graphics`Shapes`"];
bx = 6 Cos[u] (1 + Sin[u]);
by = 16 Sin[u];
rad = 4 (1 - Cos[u]/2);
X = If[π < u ≤ 2 π, bx + rad Cos[v + π], bx + rad Cos[u] Cos[v]];
Y = If[π < u ≤ 2 π, by, by + rad Sin[u] Cos[v]];
Z = rad Sin[v];
```

```
klein = ParametricPlot3D[{X, Y, Z}, {u, 0, 2 π}, {v, 0, 2 π},
                    PlotPoints → {48, 12},
                    Axes → False, Boxed → False,
                    PlotLabel → "Klein  Bottle",
                    ViewPoint → {1.4, -2.6, -1.7},
                    DisplayFunction → Identity ] ;

perfKlein = PerforatePolygons[ klein, 0.7];
Show[ perfKlein, Background → RGBColor[0.2, 1., 1.],
      DisplayFunction → $DisplayFunction] ;
```

- **Real time animation (*DynamicVisualizer* or *Conix 3DExplorer*)**

With the *DynamicVisualizer* or *Conix 3D Explorer*, new *Mathematica* application packages available, it possible to display and manipulate 3D objects in ***real-time***.

```
Needs["DynamicVisualizer`"];
```

```
Visualize[{ }];     (*Brings DynamicVisualizer window to the front *)
```

In order to display a single *Mathematica* graphics using the graphics viewpoint the command `VisualizerShow[3dObject]` has to be invoked. When the left mouse button is pressed the object is selected. Moving the mouse causes the object to rotate. Holding down the CTRL resp. SHIFT key together with the mouse button pressed and move the mouse then the object is zoomed or draged.

```
g = Table[ If[ Mod[ Multinomial[x, y, z], 2] == 1,
                Cuboid[1.2 {x, y, -z}], {} ],
           {x, 0, 15}, {y, 0, 15}, {z, 0, 15}] // Flatten ;
sierpinski = Show[Graphics3D[g],
                  Background → RGBColor[1., 0.8, 0.4],
                  PlotLabel → "Sierpinsk Gasket",
                  DisplayFunction → Identity];
```

```
VisualizerShow[sierpinski];
```

After closing the link between *Mathematica* and the *DynamicVisualizer* it is possible to launch this application again.

```
ClearVisualizer[ ];        (* Resetting DynamicVisualizer *)
QuitVisualizer[ ];         (* Closing DynamicVisualizer   *)
```

- **Animations**

Mathematica produces animated movies which give a better understanding of movements such as a *cam-follower* or a *gear mechanism* shown in `Animations.nb`

- **Sound**

Mathematica treats sound on the same footing as graphics. Substituting the command `Plot` by `Play` it is then possible to make even complicated *sound patterns* audible which is demonstrated by `SoundDemo.nb`.

High-Level Programming Language

> ***Mathematica* is an unprecedentedly flexible and intuitive programming language.**

Examples of the *Mathematica* programming language are given in the following notebook `ProgrammingDemo.nb`

In comparison to other computer algebra systems such as **Maple V**, **Macsyma**, **Axion**, **Reduce** due to the *functional programming* style of ***Mathematica*** it is often possible to write an elegant concise (single-line) program code whereas other CA programs require for the same task several lines at least. In order to show the efficiency of *Mathematica* programming the following test programs to given problems have been collected for comparison of `ProgrammingEfficiency.nb`. (See also : News-Group *sci.math.symbolic*).

Expert System

> ***Mathematica*** **incorporates knowledge from the standard mathematical handbooks, and uses its own advanced algorithms to go even beyond.**

Mathematica knows about all the hundreds of ***special functions*** in pure mathematics and mathematical physics.

Here is the *Legendre function of the 2nd kind* $Q_n(z)$

```
Q3(x)                                    (* LegendreQ[3,x] *)
```

$$\frac{2}{3}-\frac{5x^2}{2}-\frac{3}{4}x\left(1-\frac{5x^2}{3}\right)\mathrm{Log}\left[\frac{1+x}{1-x}\right]$$

Mathematica can *evaluate* special functions such as the *even Mathieu function* $ce_n(z,q)$ with any parameters *to any precision.*

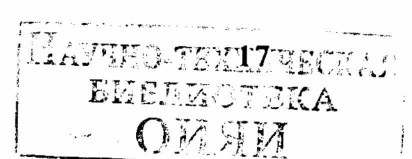
```
MathieuC[1 + i, 2 i, 3] // N[#, 30]&
```

3.92513113741251986434976461 68+ 1.89882391154334724110527479714*i*

Mathematica is now able to do vastly more ***integrals*** than were ever before possible for either humans or computers. For example, *Mathematica* claims to cover all the integrals in Gradshteyn and Ryzhik "*Tables of Integrals, Series and Products*" ,

for example the *indefinite integral*

$$\int \sqrt{x}\ \mathrm{ArcTan}[x]\,dx$$



$$\frac{2}{3}\tan^{-1}(x)\,x^{3/2} - \frac{4\sqrt{x}}{3} + \frac{1}{3}\sqrt{2}\tan^{-1}\left(\frac{2\sqrt{x}-\sqrt{2}}{\sqrt{2}}\right) + \frac{1}{3}\sqrt{2}\tan^{-1}\left(\frac{2\sqrt{x}+\sqrt{2}}{\sqrt{2}}\right) - \frac{\log\left(-x+\sqrt{2}\sqrt{x}-1\right)}{3\sqrt{2}} + \frac{\log\left(x+\sqrt{2}\sqrt{x}+1\right)}{3\sqrt{2}}$$

or the following *definite integrals*

$$\int_0^\infty \mathrm{Log}[x]\, e^{-x^3}\, dx$$

$$-\frac{1}{54}\Gamma\left(\frac{1}{3}\right)\left(6\gamma + \sqrt{3}\,\pi + 9\log(3)\right)$$

$$\int_0^\infty \mathrm{Sin}[x^2]\, e^{-x}\, dx$$

$$-\frac{-\sqrt{\pi}\cos(\frac{1}{4}) + \sqrt{2}\;{}_1F_2(1;\frac{3}{4},\frac{5}{4};-\frac{1}{64}) - \sqrt{\pi}\sin(\frac{1}{4})}{2\sqrt{2}}$$

Mathematica can also evaluate *finite and infinite sums and products.*

$$\sum_{k=1}^{n}\frac{1}{k^6}$$

$$\frac{\pi^6}{945} - \frac{1}{120}\psi^{(5)}(n+1)$$

$$\sum_{n=0}^{\infty}\frac{x^n}{n!^4}$$

$${}_0F_3(;1,1,1;x)$$

$$\prod_{i=1}^{6}\left(1 + \mathrm{Nest}[x^{\#}\&, x, i]\right)$$

$$(1+x^x)\,(1+x^{x^x})\,(1+x^{x^{x^x}})\,(1+x^{x^{x^{x^x}}})\,(1+x^{x^{x^{x^{x^x}}}})\,(1+x^{x^{x^{x^{x^{x^x}}}}})$$

Mathematica can solve a wide range of *ordinary* and *partial differential equations.*

```
DSolve[y''[x] + y'[x] + x y[x] == 0, y[x], x]
```

$$\left\{\left\{y(x) \to e^{-x/2}\left(\mathrm{Bi}\left(\sqrt[3]{-1}\left(x-\frac{1}{4}\right)\right)c_1 + \mathrm{Ai}\left(\sqrt[3]{-1}\left(x-\frac{1}{4}\right)\right)c_2\right)\right\}\right\}$$

Mathematica's *algorithms* can generate a huge range of mathematical results.

$$\prod_{n=1}^{5} \Gamma\left(\frac{2n}{5}\right) \text{ // FullSimplify}$$

$$\frac{12\pi^2}{25\sqrt{5}}$$

$$\log(2) < \zeta(3) < \sqrt{2}$$

```
True
```

$$\text{TrigReduce}(\cos^4(x))$$

$$\frac{1}{8}(3 + 4\,\text{Cos}[2\,x] + \text{Cos}[4\,x])$$

An example from Number Theory is to find the 10^9th *prime number* by using a mixture of algorithms and built-in tables.

```
Prime[10^9]
```

```
22801763489
```

Embedded System (*MathLink* Applications)

- **The modular architecture of *Mathematica* makes is it quite easy to use *Mathematica* itself as a highly powerful software component.**

The communication is established by the addon feature *MathLink*. MathLinkApplications.nb

Examples of *Mathematica* Notebooks

Files in Directory " . \CD_Dubna98\Lecture"

The built-in *Mathematica* command **FileNames** lists all files according to given file specification in a prescribed directory and in subsequent subdirectories downwards to level n.

```
FileNames["*.*", actDir, 2] // ColumnForm
```

whereas the user-defined procedure **filesInDirectory** provides the list of all *Mathematica* notebooks (file extention ***.nb***) in a given directory

```
filesInDirectory[subDir ] // ColumnForm

4ColorTheorem.nb
Add2.nb
AirFoil.nb
Animations.nb
CellularAutomata.nb
IFS.nb
Mandelbrot.nb
MathLinkApplications.nb
MovingPendulum.nb
NotebookE.nb
PerformanceE.nb
ProgrammingDemo.nb
ProgrammingEfficiency.nb
Projectile.nb
SoundDemo.nb
Swing2_3.nb
Tunneling.nb
```

The procedure openNotebookButton creates an array of buttoms which can be used to start the corresponding *Mathematica* notebooks in a directory tree structure.

```
openNotebookButton["*.nb", 1, subDir]
```

4 ColorTheorem . nb	Add2 . nb
AirFoil . nb	Animations . nb
CellularAutomata . nb	IFS . nb
Mandelbrot . nb	MathLinkApplications . nb
MovingPendulum . nb	NotebookE . nb
PerformanceE . nb	ProgrammingDemo . nb
ProgrammingEfficiency . nb	Projectile . nb
SoundDemo . nb	Swing2_ 3. nb

Calculus : Integral over Rational Function $\frac{1}{(x^2+a\,x+b)^n}$

Whilst for the integral with rational integrand $\frac{1}{(x^2+a\,x+b)^n}$ with some computational exertion

Definitions and Settings

```
Clear[a, b, n]; Remove[a, b, n];

fe[expr_] := ExpandAll[FullSimplify[expr]]
```

$$\mathbb{J}[n_][x_] := \int \frac{1}{(x^2+a\,x+b)^n}\,dx\ ;$$

the following *recursion* can be verified

$$\mathbb{J}[n][x] == \frac{1}{(n-1)\,(4\,b-a^2)}\left(\frac{2\,x+a}{(x^2+a\,x+b)^{n-1}} + (4\,n-6)\,\mathbb{J}[n-1][x]\right);$$

by differentiation of both sides of the equation above

$$\left(\frac{\partial \mathbb{J}[n][x]}{\partial x} == \frac{\partial \frac{\frac{a+2x}{(x^2+a\,x+b)^{n-1}}+(4\,n-6)\ \mathbb{J}[n-1][x]}{(n-1)\ (4\,b-a^2)}}{\partial x}\right) \quad //\,fe$$

```
True
```

Mathematica provides a *closed* solution in terms of a hypergeometric function $_2F_1$**(a,b,c,z)** which is a nearly unsolvable task calculating with paper and pencil .

```
J[n][x] // FullSimplify
```

$$\frac{1}{n-1}\left(2^{-n-1}\left(-a-2\,x+\sqrt{a^2-4\,b}\right)(b+x\,(a+x))^{-n}\left(\frac{a+2\,x}{\sqrt{a^2-4\,b}}+1\right)^n {}_2F_1\left(1-n, n; 2-n; \frac{1}{2}-\frac{a+2\,x}{2\,\sqrt{a^2-4\,b}}\right)\right)$$

Calculus : Conformal Mapping and Joukowski Transformation AirFoil.nb

Mechanics : Projectile with Air Resistance Projectile.nb

Oscillations : 2 & 3 Masses Swinger Swing2_3.nb

Oscillations :Moving Pendulum MovingPendulum.nb

Quantum Physics : Tunneling Tunneling.nb

Geometry : Polyhedron Explorer PolyhedronExplorer.nb

Chemistry : Periodic Table PeriodicTable.nb

Outlook on the Development of CAS

The computer algebra systems existing today show that it is possible to perform extensive *symbolic calculations* even on a PC and obtain interesting results. The hardware expenditure start on the lower end with a PC and goes up to a super computer for especially involved problems. The future of CAS is strongly influenced by two developments:

Firstly, with the tendency of declining hardware costs the spreading of these systeme will increase on all levels, particularly in highschools and colleges and on students' level. In some years computer algebra will be included in most curricula.

(For example in Austria the usage of intelligent pocket calculators TI 89 and Ti 92 with the built-in computer algebra system ***Derive*** is promoted from the ministry of education.)

The usage of CAS as regards to differentiation and integration, solution of equations, problems of linear algebra and other tasks will soon be taken for granted as is today the usage of a pocket calculator for evaluting square roots.

Secondly, incorporating new tasks the size of existing CAS will strongly increase so that *software engineering* will become more and more important.

The interesting **question** however is, *if* and *how* *mathematics itself will be changed* by means of automatic processing of symbolic mathematical expressions, or which new areas will originate from the availability of CAS. At first glance this question might be surprising. Yet, this is exactly the case since the 50's with respect to *numerical mathematics* after the availability of computers. Without numerical computers areas such as *optimization theory, nonlinear dynamics* or *soliton theory* are unconceivable. The availability of CAS will initiate further fundamental changes of mathematics similarly to the impact which computers had on numerics.

Examples for this statement are given below

Four Colour Theorem

The proof of the Four-Colour-Theorem, which occurs in the context of cartography, was only given in 1976 by *K. Appel* and *W. Haken* by means of a computer, which was used for "book keeping", to verify certain assumptions necessary for the proof. Because these evaluations with a computer had been quite tedious - doing the same by paper and pencil would go on for eternity - it follows that one may prove this result with the help of another computer only. Therefore some purist mathematicians think that this kind of computer proof is not real proof. Obviously, the Four-Colour-Theorem is an example for an odd aspect of mathematics: that even a simple question easily understood can lead to an incredibly complicated proof. `4 ColorTheorem.nb`

Fractals and Iterated Function Systems (IFS)

Another area are fractals; discovered by the mathematician *B. Mandelbrot* in 1980 and named after him the most famous fractals are the *Mandelbrot set* and the *Julia sets* `Mandelbrot.nb` associated with . The Mandelbrot set which arises from iteration of a harmless quadratic expression such as $z^2+c \to z$ in the complex plane is one of the *most complicated objects in*

mathematics! But only a computer program with a few lines of code is required to generate these incredibly complicated sets. There is perhaps no other proof more impressive for the *complexity* hidden in simple rules as the variety of structures reflected in the Mandelbrot set. The complexity of these sets and thus the *coexistence of order and chaos* cannot be studied without the help of a computer.

And only gradually there emerges a new concept of so-called *iterated function systems* (*IFS*) `IFS.nb` in mathematics.

Cellular Automata

Another area is the so-called *Game of Life* which was suggested in 1970 by the mathematician *J.H.Conway*. This mathematical game which is also denoted as *Cellular Automaton*, clearly demonstrates that a set of simple rules can give rise to a complex world. `CellularAutomata.nb`

The assumption is to create a cellular game (on a triangular, square or hexagonal lattice) which is based on rules as simple as possible and nevertheless should be unpredictable. This set of rule should be complete so that the game - once started - would go on by itself. Growth and change occurs discontinuously; one step leads unavoidable to the next one. The result is a small universe based on logics in which everything is pre-determined, and yet there exists *no* possibility for the observer to predict the destiny of future generations in the game - besides let the game go on by itself.

According to *Stephen Wolfram* "cellular automata are examples of mathematical systems built from many identical components to form one unit. All components are simple, together they are able to show complex behaviour. Investigating them in detail one can develop on the one hand *specific models* for certain systems, on the other hand one hopes to find generally valid principles which are applicable to a large number of complex systems".

Cellular automata are mathematical odds and interesting as games. Besides that they become increasingly important as simple models of physical phenomena. They range from hydrodynamic turbulences to the spreading of virus infection or forest fires. The *idea* is to describe a physical system in such a compact way that one can recognize the essential properties and predict future behaviour. Traditionally, mathematicians and physicists use partial differential equations for the description of physical systems; they describe how certain variables change in time. Solving these differential equations means to predict the behaviour of the system. However, many of these systems of partial differential equations are only solvable with great difficulties or not at all. Wolfram developed in 1984 a method for the approximate solution of partial differential equations which resorts to cellular automata instead of traditional approximative methods of solution. Wolfram claims that this approach is more adequate to *digital* computers than various methods solving partial differential equations on a computer by approximation. In a certain sense this methodical ansatz changes the way of constructing a theory. One changes, so to speak, the recipe in order to use in an optimal way the resources of digital computers. The appearance of massive parallel computers - where interconnected processors enable to partition the

computation such that each processor will perform the same operations simultaneously on different parts of the same data - makes the concept suggested by Wolfram even more attractive. By the way, the involvement of Wolfram with cellular automata finally led - after the predecessor system SMP - to the development of the current CAS *Mathematica*.

Conclusion

In the general context to incorporate **computer algebra systems** (and in particular *Mathematica*) in the teaching of science courses subsequent *positive aspects* can be summarized :

- Save time when solving mathematical problems
- make parameter dependence of algebraic solutions transparent
- visualize 3D objects and dynamic processes
- demonstrate numerical effects of methods by freely chosen accuracy
- extend existing program libraries by new algorithms

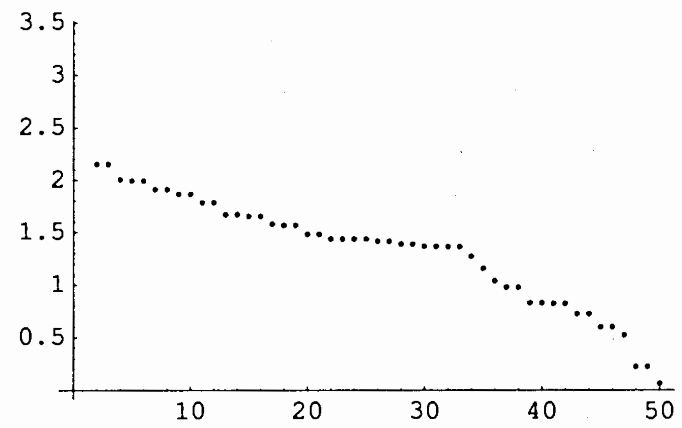
The implications which result from applying these techniques in lectures are formulated in terms of *theses* :

- it is possible to deal with *more involved, more realistic problems* in the lecture hall (ad hoc)
- excellent visualization of mathematical issues using the graphics capabilities of CAS is provided
- there results an early familiarity of students with CAS techniques to be useful in industry later
- as regards to knowledge transfer in the process of lecturing the emphasis on <u>TECHNIQUES</u> achieving a solution *moves* towards <u>DISCUSSION</u> of the properties of the solution obtained

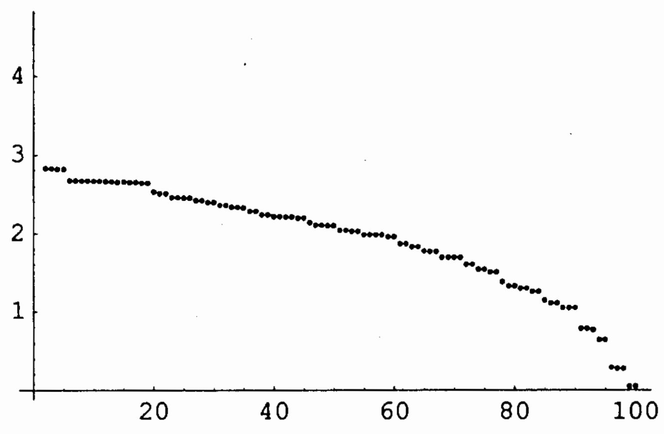
Mathematica Performance

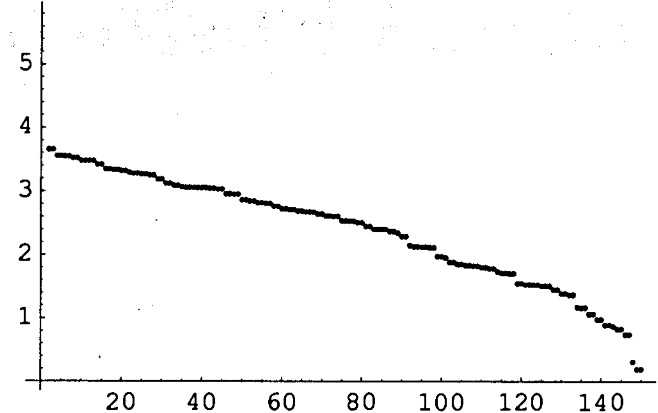
Calculation of eigenvalues for n × n random matrices of increasing size.

```
Clear[z]; z = {};
k = {50, 100, 150, 200, 300, 400, 500};

Do[ n = k[[j]];
      M = Table[ Random[], {n}, {n} ];
      zeit = ( ListPlot[ Abs[ Eigenvalues[M] ] ] // Timing  );
      z = AppendTo[z, zeit[[1, 1]]  ];
         Print[ zeit[[1, 1]] ] ,
   {j, 1, Length[k] } ];
    z
```

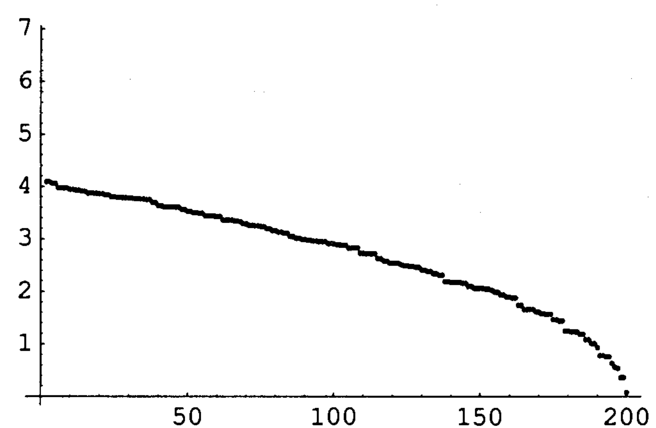


0.99

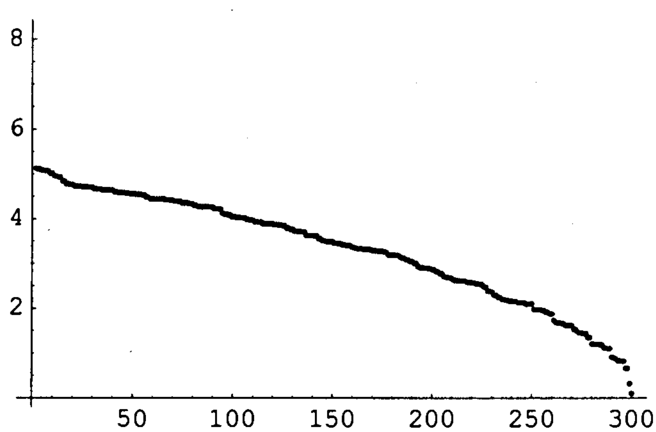


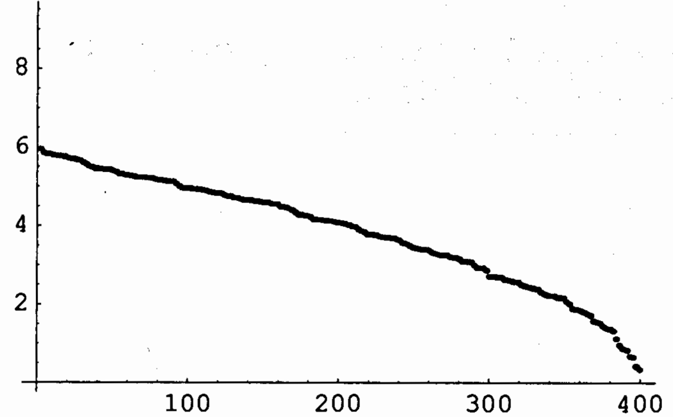


3.18

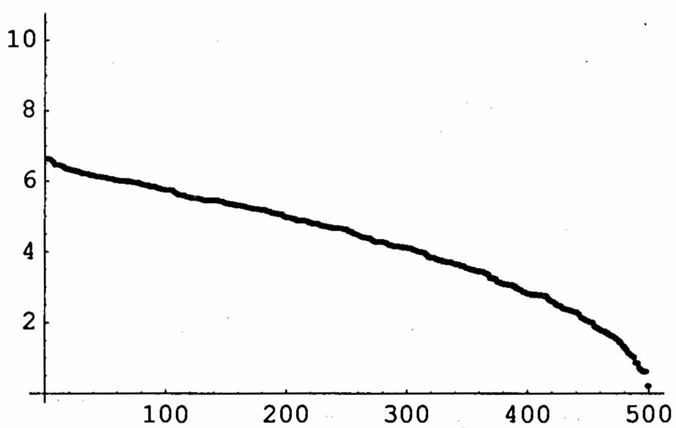


11.26






```
129.46
```



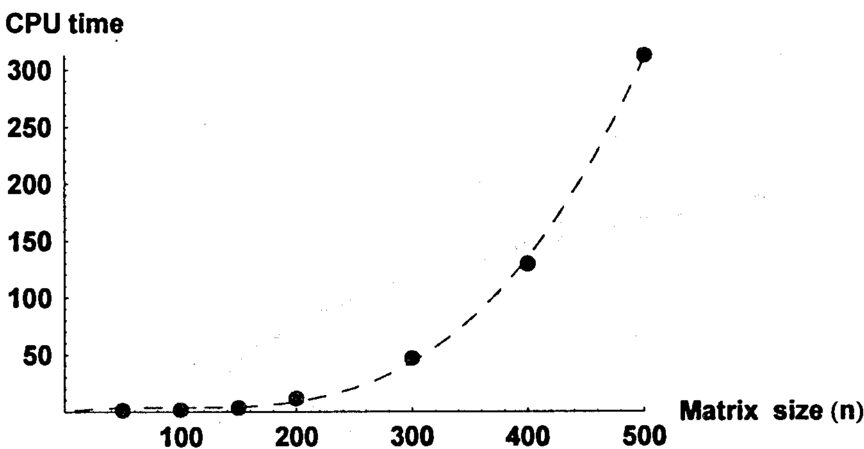

```
312.75

{0.99, 1.43, 3.18, 11.26, 46.96, 129.46, 312.75}
```

```
              (*  z = {0.99,1.43,3.18,11.26,46.96,129.46,312.75}; *)
xf = Transpose[{k, z}];


 lp =  ListPlot[ xf,
              PlotRange → {{0, Max[k]}, {0, Max[z]}},
              PlotStyle -> {Red, AbsolutePointSize[6]},
              DisplayFunction → Identity ];

F1[x_] = Fit[ xf, Table[x^m, {m, 1, 3}] , x]        (* Polynom-Fit *)

fp = Plot[ F1[x], {x, 1, 500}, ,
              PlotRange → All ,
              PlotStyle → {Blue, Dashing[{0.03}]},
              DisplayFunction → Identity ];

Show[ lp, fp, AxesLabel -> {"  Matrix  size (n)", "CPU time"},
           DisplayFunction → $DisplayFunction ];
```

Performance of *Mathematica* 3.0 found as regards to the calculation of eigenvalues for n × n random matrices.

$$\begin{pmatrix} 50 & 0.99 \\ 100 & 1.43 \\ 150 & 3.18 \\ 200 & 11.26 \\ 300 & 46.96 \\ 400 & 129.46 \\ 500 & 312.75 \end{pmatrix}$$

$4.62023 \times 10^{-6} x^3 - 0.00130389 x^2 + 0.11898 x$



Mathematica Notebooks

Chaos and Fractals

Prof. Dr. Robert Kragler
FH Ravensburg-Weingarten / *University of Applied Sciences*
kragler@rz.fh-weingarten.de

***Mandelbrot* and *Julia* Sets**

Initializations

Visualization of Complex Functions

Mathematica Journal Vol 3 # 1, "Tricks of the Trade" p. 18

■ Usage of Compile

The evaluation of a function f[x_1,x_2,...] is essentially accelerated if *Mathematica* can assume that the variables x_1,x_2, ...can be dealt with as machine numbers instead of checking each time whether it is a number, a list, an algebraic object or some other type of expression. The usage of

Compile[{x_1,x_2,...},expression] (1)

creates a CompiledFunction object which contains a sequence of simple instructions (similar to machine code) for the evaluation of the *compiled* function and thus can be executed faster than an ordinary *Mathematica* function.

It is assumed for the evaluation of a compiled function that all parameters x_1,x_2,... which occur are numbers or logical variables. For large expressions compilation may speed up execution up to a factor of 20. Using the command Compile is always reasonable if a given numeric or logical expression is evaluated many times. However, although compilation makes the execution of numerical functions more efficient it is recommended to use built-in *Mathematica* functions whenever this is possible because they are generally faster than every compiled user-defined procedure in *Mathematica* . Built-in functions such as Integrate but also Plot and Plot3D make use of the default option Compiled→True for any function inserted. With Compiled→False compilation is deactivated.

Iterative Maps

Here are two examples which demonstrate the usage of `Compile` in order to speed up evaluation. These are *iterative mappings* of type

$$z_{n+1} = (z_n)^2 + c \qquad\qquad z_{n+1} = (z_n)^2 + z_0 \tag{2}$$

■ *Julia* Sets

The ***Julia set*** associated with a complex number c is obtained by *iteration* of the mapping

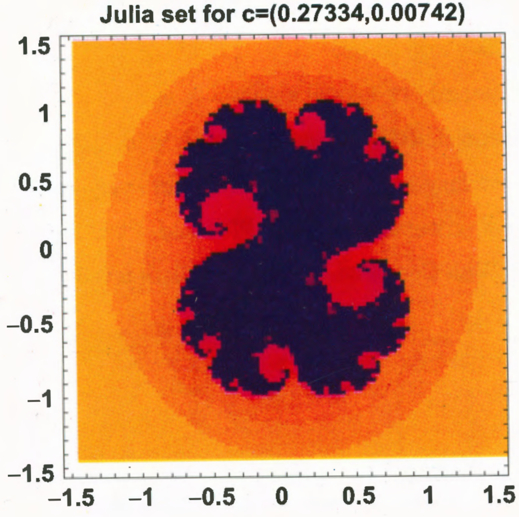
$$z = z^2 + c \tag{3}$$

It is the number of all those points which will not tend to ∞ during interation.

```
juliaC = Compile[{x, y, lim, cx, cy},
          Module[{z, c, n = 0},
                 z = x + ⅈ y; c = cx + ⅈ cy;
                While[ Abs[z] < 2.0 && n ≤ lim, z = z² + c; ++n];
                 n] ];
                          n5 :=
 NumberForm[ #, {6, 5}]& ; juliaSet[ cx_, cy_, col_] :=
 DensityPlot[ -juliaC[x, y, 50, cx, cy], {x, -1.5, 1.5}, {y, -1.5, 1.5},
             PlotPoints → 100, Mesh → False,
             ColorFunction → col, PlotLabel →
   StringForm["Julia set for c=(``,``)", cx // n5, cy // n5] ];
```

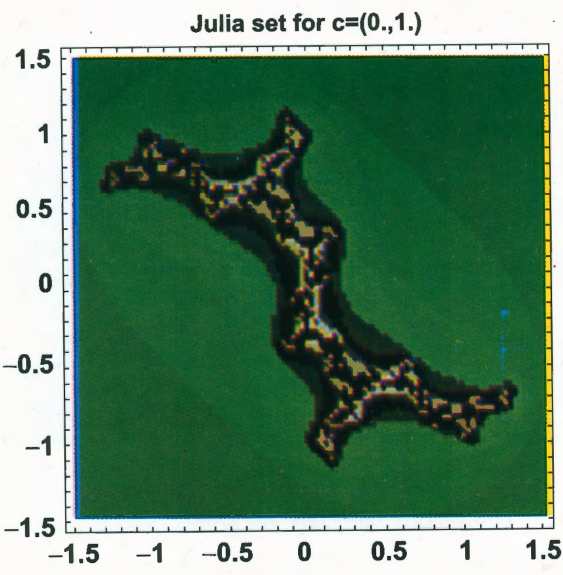
The following Julia set is obtained for a value of `c = 0.27334+0.00742ⅈ` :

```
                    colJ1[z_] := Hue[z/2.1 - 1/3];
juliaSet[0.27334, 0.00742, colJ1];
```

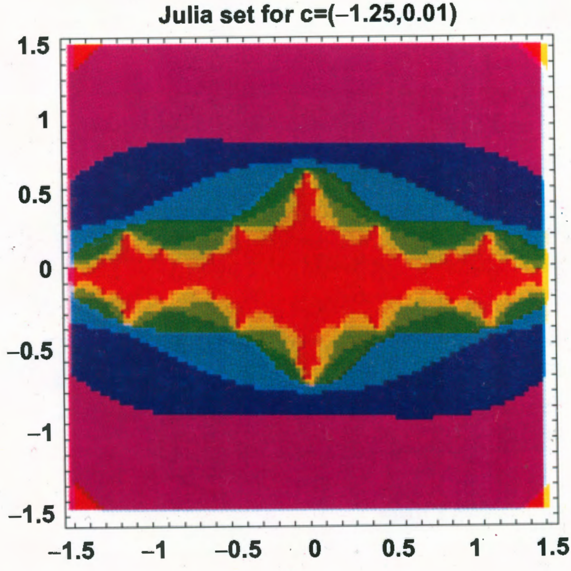


The appearance of the fractal *Julia sets* strongly depends on the location of the fixpoint ***c*** with respect to the so-called *Mandelbrot set*. Interesting structures will be obtained for those values of ***c*** which are located in the vicinity of the Mandelbrot set. For values of ***c*** from outside of the *Mandelbrot set* the corresponding *Julia sets* are continuous. For values of ***c*** outside the *Mandelbrot set* the corresponding *Julia set* disintegrates into island-like structures. The more the distance of ***c*** from the *Mandelbrot set* the more the associated *Julia set* dissolves which is called *Fatou dust*.

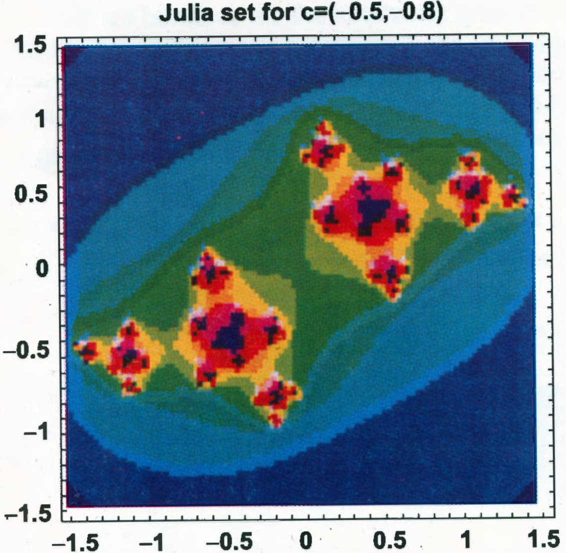
```
colJ2[z_] := Hue[.45, z, Mod[1/(z^5 + .01), 1] ];
juliaSet[0, 1, colJ2];
```




```
colJ3[z_] := Hue[Mod[z^2, 1], .8, .8 ];
juliaSet[-1.25, 0.01, colJ3];
```




```
colJ4[z_] := Hue[z + .7];
juliaSet[-0.5, -0.8, colJ4];
```



■ *Mandelbrot* Set

The *Mandelbrot set* is the set of points `c` = `z` not escaping to ∞ under the iteration of the mapping

$$z = z^2 + z_0 \tag{4}$$

Points *within* the Mandelbrot set lead to continuous *Julia sets*, for points outside the *Mandelbrot set* the corresponding *Julia sets* disintegrate into mosaic-like structures. The following modification of the procedure `juliaC` may be used to create the *Mandelbrot set* too.

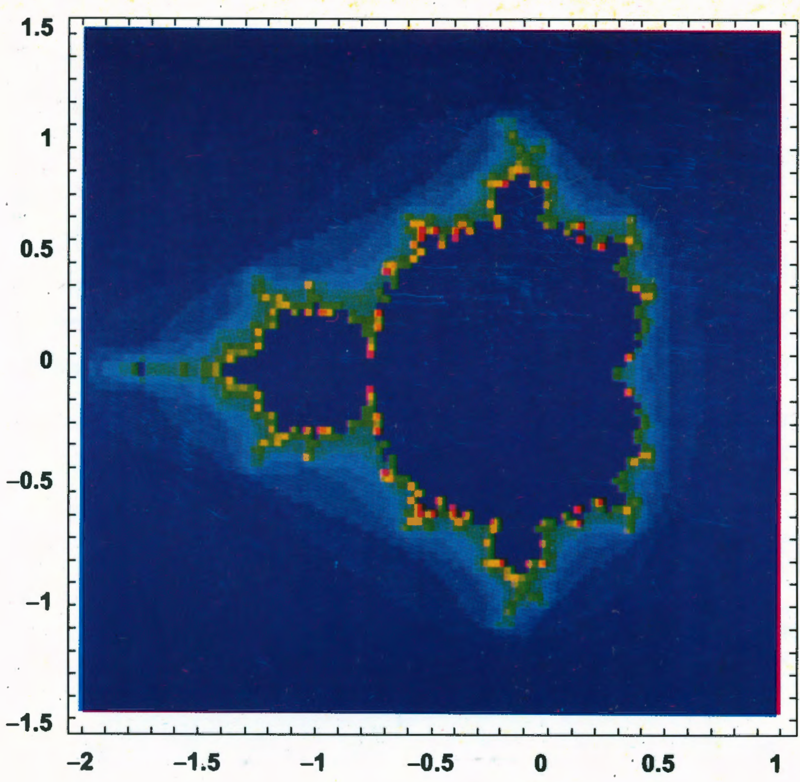
```
mandelbrotC = Compile[{x, y, lim},
                    Module[{z0, z, n = 0},
                              z0 = x + I y; z = z0;
                      While[Abs[z] < 2.0 && n ≤ lim, z = z^2 + z0; ++n];
                              n] ];
```

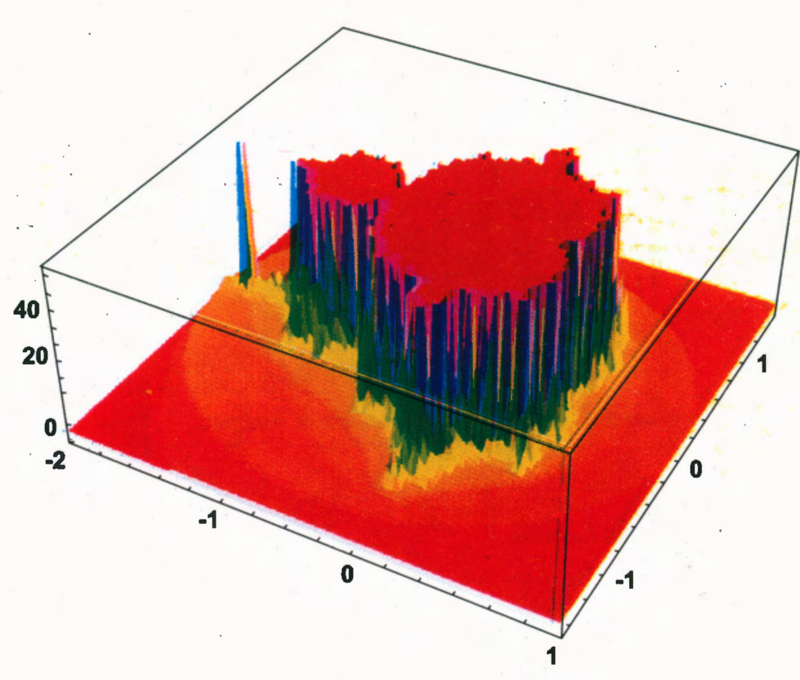
First, the *Mandelbrot set* is displayed as a density plot, moreover, the visualization of the *Mandelbrot set* as a 3D surface by means of the procedure `Plot3D[mandelbrot,...]` is possible too.

```
m1 = DensityPlot[-mandelbrotC[x, y, 50], {x, -2., 1.}, {y, -1.5, 1.5},
                    ColorFunction → (Hue[.7 + #1]&),
                     PlotPoints → 100, Mesh → False,
                     DisplayFunction → Identity];
m2 = Plot3D[ mandelbrotC[x, y, 50], {x, -2., 1.}, {y, -1.5, 1.5},
             ColorFunction → Hue, PlotPoints → 100, Mesh → False,
             DisplayFunction → Identity];
```

```
Show[ GraphicsArray[{{m1}, {m2}}],
      PlotLabel → "Mandelbrot  set z = z²+z0",
      DisplayFunction → $DisplayFunction ] ;
```

Mandelbrot set $z = z^2 + z_0$





MathLink Applications

Mathematica as a Software Component

Mathematica has a *modular architecture* that makes it easy to use as a highly powerful software component.

Here is some input and output in the *standard notebook front end* to *Mathematica*.

$$\int \sqrt{\text{Log[x]}}\, dx$$

$$-\frac{1}{2}\sqrt{\pi}\ \text{Erfi}\left[\sqrt{\text{Log[x]}}\right] + x\sqrt{\text{Log[x]}}$$

You can also access the *Mathematica* kernel directly from a *raw terminal*.

```
Integrate[Sqrt[Log[x]], x]

   1
-(-) Sqrt[Pi] Erfi[Sqrt[Log[x]]] + x Sqrt[Log[x]]
   2
```

MathLink provides a general program-level interface between *Mathematica* and external programs.

Here is *C code* for sending an expression $\int \sqrt{\text{Log[x]}}\, dx$ from an external program to *Mathematica*.

```
/* Integrate[Sqrt[Log[x]], x] */

MLPutFunction ( stdlink, "EvaluatePacket", 1);
MLPutFunction ( stdlink, "Integrate", 2);
MLPutFunction ( stdlink, "Sqrt", 1);
MLPutFunction ( stdlink, "Log", 1);
MLPutSymbol ( stdlink, "x", 1);
MLPutSymbol ( stdlink, "x");
MLEndPacket ( stdlink);
```

This installs a compiled *external C program* that does *bitwise operations on integers*.

```
link = Install["bitops"];
```

This executes the external code for the BitAnd function.

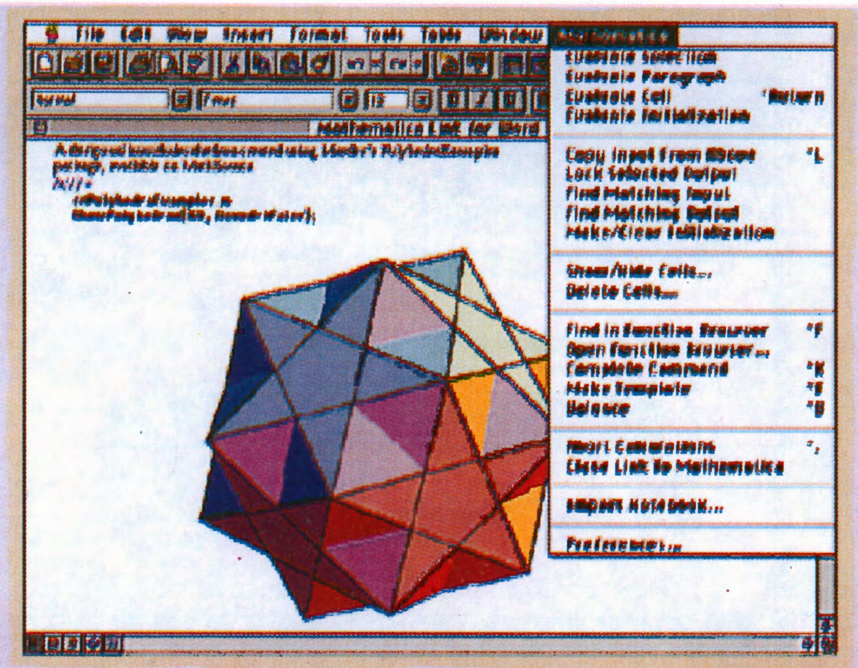
```
BitAnd[22222, 33333]
516
```

This uninstalls the external program.

```
Uninstall[link];
```

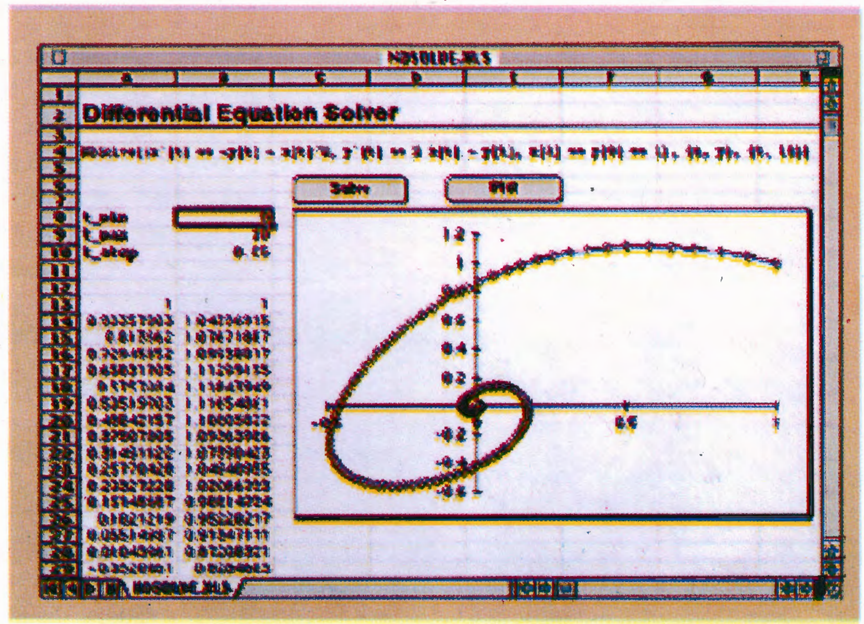
MathLink can be used to access the Mathematica kernel from external programs.

MS Word front end to *Mathematica*.



Netscape on the server of Wolfram Research.

MS Excel linked to *Mathematica.*



Under MS Windows, you can click this button to start a simple example "..\AddOns*MathLink*\DevelopersKits\Windows\UnsupportedGoddies\ **vbfe**" of a *Visual Basic* front end to *Mathematica.*

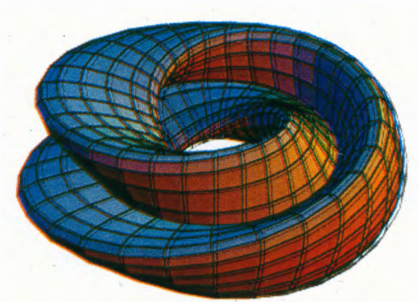
MathLink can be used to access other programs from within the _Mathematica_ kernel.

MathLink allows you to set up *templates to specify how external programs* should be called. This defines a *link* to a *C subroutine library.*

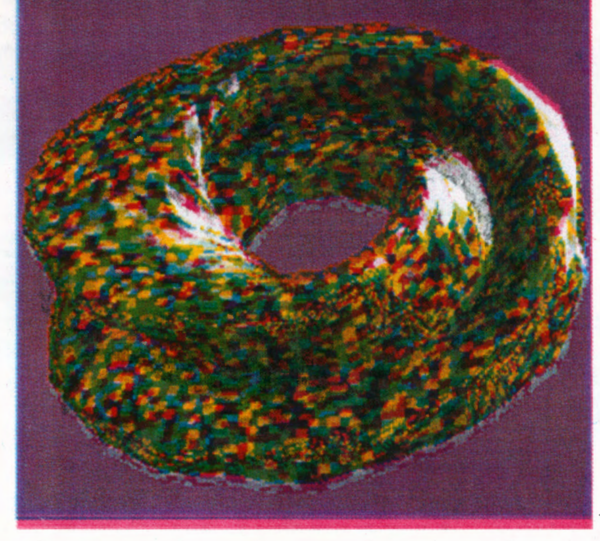
```
:Begin:
:Function:          anneal
:Pattern:           TSPTour[r:{{_, _}..}]
:Arguments:         {First[Transpose[r]], Last[Transpose[r]],
                     Length[r], Range[Length[r]]}
:ArgumentTypes:     {RealList, RealList, Integer, IntegerList}
:ReturnType:        Manual
:End:
```

Here is a 3D graphic generated ***within*** *Mathematica.*

```
ParametricPlot3D[
 {(2 + Cos[u/2] Sin[v] - Sin[u/2] Sin[2 v]) Cos[u],
  (2 + Cos[u/2] Sin[v] - Sin[u/2] Sin[2 v]) Sin[u],
  Sin[u/2] Sin[v] + Cos[u/2] Sin[2 v]},
 {v, 0, 2 Pi}, {u, 0, 2 Pi}, PlotPoints -> 30, Boxed -> False, Axes -> None];
```



This image was generated by sending a description of the graphic from *Mathematica* via *MathLink* connection to an ***external*** *photorealistic renderer.*



MathLink can be used to control the *Mathematica* front end from *within* the kernel.

This tells the front end to bring up the *color selector dialog box*.

```
FrontEndTokenExecute["ColorSelectorDialog"]
```

MathLink can be used to communicate between _Mathematica_ kernels — on one computer or several.

On most computer systems (typically excluding Macintosh) this launches a subsidiary *Mathematica* kernel on your computer.

```
link = LinkLaunch["MathKernel -mathlink"];
```

This reads data from the subsidiary *Mathematica* kernel.

```
LinkRead[link]
InputNamePacket[In[1]:= ]
```

This writes a command to the subsidiary kernel.

```
LinkWrite[link, Unevaluated[$SessionID]]
```

This reads back the `$SessionID` from the subsidiary kernel.

```
LinkRead[link]
ReturnPacket[20514827392884103963]
```

The `$SessionID` in your main kernel will be different.

```
$SessionID
20514820511569745766
```

This closes down the subsidiary kernel.

```
LinkClose[link];
```

University Center
Joint Institute of Nuclear Research / Dubna

Mathematica Seminars
October 27-29, 1998

Mathematica Programming

Prof. Dr. Robert Kragler
FH Ravensburg-Weingarten /University of Applied Sciences
kragler@fh-weingarten.de

Abstract :

The purpose of this tutorial is to demonstrate the versability and flexibility of *Mathematica*'s high-level programming language which could be likewise used as *testbed* for various programming paradigms.

In the *first part* some general remarks on different *Mathematica* notions such as **prefix**, **postfix** and **infix** notation and some new typesetting features of the current *Mathematica* version 3.0 will be discussed.

- Prefix Notation
- Postfix Notation
- Infix Notation
- New Typesetting Features

The *second part* deals in detail with various programming styles wich can be realized using the *Mathematica* language

- Procedural Programming
- Recursive Programming
- Functional Programming
  (with the concept of pure and higher-level functions such as
  `Apply, Map, Nest, Fold, FixedPoint, Thread, Inner, Outer` etc.)
- Rule-Based Programming
- Logic (or Declarative) Programming
- Abstract Data Types
- Object-Oriented Programming
- Modularisation and *Mathematica* Packages

The different programming styles resp. concepts will be illustrated by numerous examples.

- *Initializations*

General Features of *Mathematica*

- *Mathematica* is a Computer Algebra System and (especially since version 3.0) at the same time it is a System for Technical Computing

Mathematica is a CAS for *symbolic manipulation* of formulae mainly. Apart from *numerical* and *symbolic* calculations and comfortable *2D/3D graphics* and *animation* this system is supplemented by the following *salient features* :

- Graphic Interactive User Interface i.e. the Front End and the Notebook
- High-level Programming Language for Developing new Procedures

In comparison to other CAS an outstanding feature of *Mathematica* is its high-level programming language which, supplemented by the *Mathematica* notebook environment, provides execellent means for developing and testing algorithms. Hence, because the *Mathematica* engine, i.e. the kernel, is an *interpreter* (and not a compiler) system it lends itself for *rapid prototyping*. Auxilliary to that the *Mathematica* programming language is very close to the way of thinking in mathematics and physics; thus it is quite easy to rewrite formulae into *Mathematica* expressions in a straightforward manner. Due to the concise *Mathematica* encoding programs may shrink by a factor of 10 to 20 in comparsion to traditional programming languages such as FORTRAN or C .

Mathematica Notebooks : *Interactive* Documents

Mathematica Notebooks are *interactive* electronic documents which may contain : *Mathematica* **code** (to be executed), **text** (with most DTP features supported such as different fonts, colour, mathematical notation etc.), **graphics** and **animation** (even in real-time using the add-on program *MathLive*), **sound** which is encoded in PostScript code the same way as graphics and other elements in terms of cells.

Every *Mathematica* notebook is a *Mathematica* expression which can therefore be generated or manipulated by the kernel. `NotebookE . nb`

In more detail : ***Mathematica* notebooks** are ordinary *Mathematica* expressions. Therefore they can immediately be *manipulated* by the user without any special setup etc. thus the results given in *Mathematica* notebooks can easily be checked or altered. *Mathematica* notebooks are organised in terms of *hierarchical cells* the layout of which can be pre-defined in so-called *StyleSheets* which are again *Mathematica* notebooks. Notebooks can even be created by the *Mathematica* kernel as output. There are no limitations as regards to the appearance of *Mathematica* notebooks. Due to the hierarchical cell concept notebooks can be read like books wherein there are table of content, sections and subsections etc. which are accessible after opening the corresponding pages. However, the *advantage* of the electronic document in comparison to conventional books is the *interactivity*, i.e. probing and manipulating the *Mathematica* code incorporated in the notebook. (The very first pioneering book which contained all chapters in terms of *Mathematica* notebooks on a CD-ROM disk was "Exploring Mathematics with *Mathematica*" by Th. Gray and J. Glynn published by Addison-Wesley Publ. Company already in 1991.)

But the *real important* new feature in *Mathematica* version 3.0 is the *portability* of *Mathematica* notebooks. *Mathematica* notebooks contain all information in terms of ASCII code. They are *platform independent*. In earlier versions it was always a big problem to convert *Mathematica* notebooks which had been generated, say, under Windows operating system to Unix workstations (OS Solaris), NeXT (OS NeXTSTEP) or Macintosh (Mac OS 7) computers. Especially international characters such as German umlauts (or kyrillic letters) and other special symbols were a tedious problem to cope with in the past

because there is no general accepted standard for character encoding. In *Mathematica* V 3.0 this problem is definitely solved. There are *code translation tables* for all *Mathematica* implementations which are provided for a large number of hardware platforms.

Thus *Mathematica* notebooks may be used as a mean for *exchange of scientific information* and could become a kind of standard for interactive electronic documents. On a wider scale *Mathematica* notebooks can be provided to academia via Web. There is already the built-in capability to save *Mathematica* notebooks in *HTML format* so that they can be used as WWW pages.

Recent developements from Wolfram Research are *Publicon* and *MathML* :

Publicon - a comprehensive *solution for interactive technical publishing* which allows the creation of professional-quality technical documents for on-screen, Web, and printed use. A free beta version of *Publicon* can be downloaded from <http://www.publicom.com/> .

MathML (*Math*ematical *M*arkup *L*anguage) - a new standard recently ratified by W3C (i.e. the WWW Consortium) becomes the framework for Web typesetting technology. *MathML* is designed to allow mathematical expressions to be transmitted over the Web, preserving the structure needed to do computations with them in *Mathematica*. Information on *MathML* can be downloaded from the Web at <http://www.wolfram.com/news/mathml/ > .

Mathematica Programming Language

Mathematica is an unprecedentedly *flexible* and *intuitive* programming language. *Examples* of this feature are given in the following notebook : `ProgrammingDemo.nb`

In contrast to other CAS because of the *functional programming* style of ***Mathematica*** it is often possible to write an elegant concise one-liner program whereas other CAS require for the same task at least several lines. In order to show the *efficiency* of *Mathematica* programming the following test programs to given problems have been collected for comparison in : `ProgrammingEfficiency.nb`

The different *programming styles* supported by *Mathematica* will be discussed in depth in the following notebook which serves as a tutorial : `ProgrammingStyles.nb`

Conclusion

In conclusion the *Mathematica* high-level programming language is a powerful interpreter language most suitable for development and testing of algorithms. Thus it is an ideal tool for rapid prototyping. It comprises all programming styles and features known from traditional programming languages. There is an extensive range of more than 1000 commands and procedures provided by *Mathematica*. Hence very concise code can be written as a consequence of which the size of *Mathematica* programs may shrink to 1/10 or more compared to other traditional programming languages such as FORTRAN or C etc. A salient feature is that the *Mathematica* programming language is very close to the way of thinking in science and technology. Thus, the conversion of mathematical formulae into *Mathematica* expressions is in most cases straightforward and is faciliated even more by the notation of *Mathematica* version 3.0.

The features of *Mathematica* can be summarized as follows :

- high-level interpreter language
- most suitable for development and testing of algorithms

- comprehensive range of commands
- concise programm code
- programming language follows thinking in science
- straightforward conversion of formulae into *Mathematica* code

Mathematica Notebooks

Mathematica Programming Language

Prof. Dr. Robert Kragler
FH Ravensburg-Weingarten */ University of Applied Sciences*
kragler@rz.fh-weingarten.de

Programming Demo

- **Indexed Variables**

Copyright Wolfram Research Inc.

Unifying Concept of the *Mathematica* Programming Language

Mathematica includes *advanced programming concepts* from modern computer science - as well as adding new ideas of its own. *Mathematica* incorporates a whole range of programming paradigms - thus one can write every program in its most natural way resp. style.

Mathematica is built on the powerful *unifying concept* that everything can be represented as a symbolic expression.

- **All symbolic expressions are built up from combinations of the form** `head[arg1, arg2 ...]`

A list of elements

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

An algebraic expression

```
Plus[Power[x, 2], Sqrt[x]]
```
$\sqrt{x} + x^2$

An equation

```
Equal[x, Sin[x]]
x == Sin[x]
```

A logic expression

```
And[p, Not[q]]
p && !q
```

A command

```
AddTo[Part[m, 1], a]
m[[1]] += a
```

Graphics

```
Graphics[{Hue[0], Circle[{1, 0}, 2], Circle[{-1, 0}, 2]}] // Show ;
```

Abstract mathematical notation

```
Tilde[ CirclePlus[a, b], Subscript[c, Infinity]]
a⊕b ~ c∞
```

A button

```
ButtonBox["Press here"] // DisplayForm
Press here
```

A cell containing text

A cell containing text is represented as

```
Cell["A cell containing text is represented as", "Text"]
```

- **The uniformity of symbolic expressions makes it easy to add to *Mathematica* any construct wanted**

A chemical compound HNO_3

```
Remove[Chemical]
```

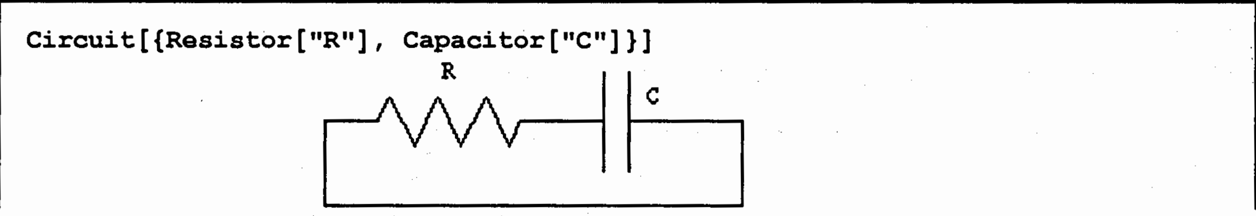
```
Chemical[arg : {{_Symbol, _Integer}..}] :=
    Module[{chem, chemFormula = {} },
       Do[ element = arg[[i, 1]]; n = arg[[i, 2]];
          Switch[ ToString[element], "Hydrogen", chem = "H",
                                    "Oxygen",   chem = "O",
                                    "Nitrogen", chem = "N"];
          If[n == 1, chem, chem = Subscript[chem, n] ];
          chemFormula = AppendTo[chemFormula, chem],
          {i, 1, Length[arg]} ];
          (Times @@ chemFormula) // Print
          ]
```

```
Chemical[{{Hydrogen, 1}, {Nitrogen, 1}, {Oxygen, 3}}]
```

$H N O_3$

An electric circuit

```
Circuit[{Resistor["R"], Capacitor["C"]}]
```



- **All operations in *Mathematica* are ultimately transformations of symbolic expressions. *Mathematica* has a uniquely powerful pattern matcher for applying transformation rules.**

The notation `/.` tells *Mathematica* to apply the simple rule `b →1+x`.

```
{a, b, c, d} /. b → 1 + x
{a, 1 + x, c, d}
```

x_ and y_ each stand for any expression, so the pattern `x_+ y_` stands for a sum of terms.

```
{a + b, c + d, a + c} /. x_ + y_ -> x^2 + y^2
{a^2 + b^2, c^2 + d^2, a^2 + c^2}
```

```
{a + b, c + d, a + c} /. a + x_ -> x^3
{b^3, c + d, c^3}
```

- ***Mathematica* uses patterns to generalize the notion of functions.**

This is an ordinary function definition for `f(x)` to be used for any `x`.

```
f[x_] := 2/x
```

Here is a special case that *overrides* the general definition.

```
f[0] := e
```

Here is an example of the use of **f**.

```
f[6] + f[a + b] + f[0]
```

$\frac{1}{3} + \frac{2}{a + b} + e$

This clears the definitions given for `f`.

```
Clear[f]
```

- **An important feature of using patterns is that they allow "functions" to take arguments in any structure.**

This defines a value for `g` with an argument that is a *list* of two elements.

```
g[{x_, y_}] := x + y
```

```
g[{4, a b}]
```

4 + a b

```
Clear[g]
```

This specifies the value for a function `area` when given a `Circle` object as an argument.

```
area[Circle[{_, _}, r_]] := π r^2
```

```
area[Circle[{2, 3}, u]]
```

πu^2

This implements a *logic reduction rule*. $p \wedge q \vee p \quad p$

```
reduce[p_ && q_ || p_] := p
```

```
logic = A ∧ (¬ A) ∨ A
logic  // reduce
A && ! A || A
```

```
A
```

Mathematica Programming Styles

▪ Procedural Programming

```
z = a;
Do[ Print[z *= z + i], {i, 3}]
```

```
a (1 + a)

a (1 + a) (2 + a (1 + a))

a (1 + a) (2 + a (1 + a)) (3 + a (1 + a) (2 + a (1 + a)))
```

```
Clear[z]
```

▪ Functional Programming

```
NestList[f, x, 4]
{x, f[x], f[f[x]], f[f[f[x]]], f[f[f[f[x]]]]}
```

The expression $(1 + \#)^2$ & is a *pure (anonymous) function*, where # acts as a place holder to be replaced during execution by the actual variable.

```
NestList[(1 + #)^2 &, x, 3]
{x, (1 + x)^2, (1 + (1 + x)^2)^2, (1 + (1 + (1 + x)^2)^2)^2}
```

▪ Rule-based Programming

```
p[x_ + y_] := p[x] + p[y]
```

```
p[a + b + c + d]
p[a] + p[b] + p[c] + p[d]
```

_ stands for any simple expression ; __ for *any* sequence of expressions.

```
s[{x__, a_, y__}, a_] := {a, x, x, y, y}
```

```
s[{1, 2, 3, 4, 5, 6}, 4]
{4, 1, 2, 3, 1, 2, 3, 5, 6, 5, 6}
```

```
Clear[p, s]
```

Object-Oriented Programming

Here are 3 different definitions to be associated with the *object* **h** .

```
h /: f_[h[x_]]    := fh[f, x]
```

```
h /: p[h[x_], x_] := hp[x]
```

```
h /: h[x_] + h[y_] := hplus[x, y]
```

This connects the definitions with the object **h**

```
f[h[r]] + p[h[x], x] + h[a] + h[b]
fh[f, r] + hp[x] + hplus[a, b]
```

```
Clear[h]
```

List-based Programming

Many operations of *Mathematica* are automatically *threaded* over lists.

```
1 + {a, b, c}^2
{1 + a^2, 1 + b^2, 1 + c^2}
```

```
lst = Table[i^j, {i, 4}, {j, i}]
{{1}, {2, 4}, {3, 9, 27}, {4, 16, 64, 256}}
```

This command *flattens* sublists.

```
lstf = Flatten[lst]
{1, 2, 4, 3, 9, 27, 4, 16, 64, 256}
```

This *partitions* into sublists with 2 elements each.

```
Partition[lstf, 2]
{{1, 2}, {4, 3}, {9, 27}, {4, 16}, {64, 256}}
```

String-based Programming

```
StringReplace["aababbaaabaabababa", {"aa" → "💡💡💡", "ba" → "⌚"}]
💡💡💡⌚b⌚💡💡💡⌚a⌚⌚⌚
```

Mixed Programming Paradigms

Many of *Mathematica's* powerful functions mix different programming paradigms.

```
Position[{1, 2, 3, 4, 5} / 2, _Integer]
{{2}, {4}}
```

```
MapIndexed[Power, {a, b, c, d}]
{{a}, {b^2}, {c^3}, {d^4}}
```

```
FixedPointList[If[EvenQ[#1], #1/2, #1]&, 10^5]
{100000, 50000, 25000, 12500, 6250, 3125, 3125}
```

```
ReplaceList[{a, b, c, d, e}, {x__, y__} -> {{x}, {y}}]
{{{a}, {b, c, d, e}}, {{a, b}, {c, d, e}}, {{a, b, c}, {d, e}},
 {{a, b, c, d}, {e}}}
```

Mathematica gives the flexibility to write programs in many different styles. Here a 12 different definitions of the `Factorial` function are given.

```
f1[n_] := Factorial[n]
```

```
f2[n_] := n!
```

```
f3[n_] := Gamma[n + 1]
```

```
f4[n_] := n f4[n - 1];
f4[1] = 1;
```

```
f5[n_] := Product[i, {i, n}]
```

```
f6[n_] := Module[{t = 1},
                 Do[t = t*i, {i, n}];
                 t]
```

```
f7[n_] := Module[{t = 1, i},
                 For[i = 1, i <= n, i++, t *= i];
                 t]
```

```
f8[n_] := Apply[Times, Range[n]]
```

```
f9[n_] := Fold[Times, 1, Range[n]]
```

```
f10[n_] := If[n == 1, 1, n f10[n - 1] ]
```

```
f11[n_] := Fold[ #2[#1]&, 1, Array[Function[t, # t]&, n]]
```

```
f12      := If[#1 == 1, 1, #1 f12[#1 - 1]]&
```

In order to test the *efficiency of the 12 different factorial function definitions* the *total CPU time* for the factorials { `1000!, 1500!, 2000!, 3000!` } is measured in seconds and sorted with respect to time.

```
fac = {f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12} ;
val = {1000, 1500, 2000, 3000};
$RecursionLimit = 4000; nfac = Length[fac];
```

```
time = Table[ (Log[fac[[i]] /@ val ]) // N // Timing , {i, 1, nfac}];
```

Thus one obtains for example

```
{ time[[5]], FullDefinition[fac[[5]] // Evaluate] }
{{1.65 Second , {5912.13 , 9474.41 , 13206.5 , 21024. }},  f5[n_] := ∏_{i=1}^n i}
```

As regards to efficiency of the factorial function definitions the following *ranking* is found :

```
orderedTimes =
     Table[ { time[[i]][[1, 1]], FullDefinition[fac[[i]] // Evaluate] },
          {i, nfac - 1}] ⋃ {{ time[[12]][[1, 1]], FullDefinition[f12]}} ;

orderedTimes // ColumnForm
{0.06, f10[n_] := If[n == 1, 1, n f10[n - 1]] }
{0.16, f2[n_] := n!}
{0.17, f1[n_] := n!}
{0.17, f3[n_] := Gamma [n + 1] }
{1.37, f8[n_] := Times @@Range [n] }
{1.48, f9[n_] := Fold [Times , 1, Range [n]] }
{1.65, f5[n_] := ∏_{i=1}^n i}
{2.2 , f6[n_] := Module [{t = 1}, Do[t = t i, {i, n}]; t] }
{2.41, f4[1] = 1              }

       f4[n_] := n f4[n - 1]
{2.69, f12 := If[#1 == 1, 1, #1 f12[#1 - 1]]&}
{3.24, f7[n_] := Module [{t = 1, i}, For[i = 1, i ≤ n, i++, t *= i]; t] }
{3.35, f11[n_] := Fold [#2[#1]&, 1, Array [Function [t, #1 t]&, n]] }
```

Writing Programs in *Mathematica*

Mathematica's high-level programming constructs enable to built sophisticated programs more quickly than with traditional programming languages.

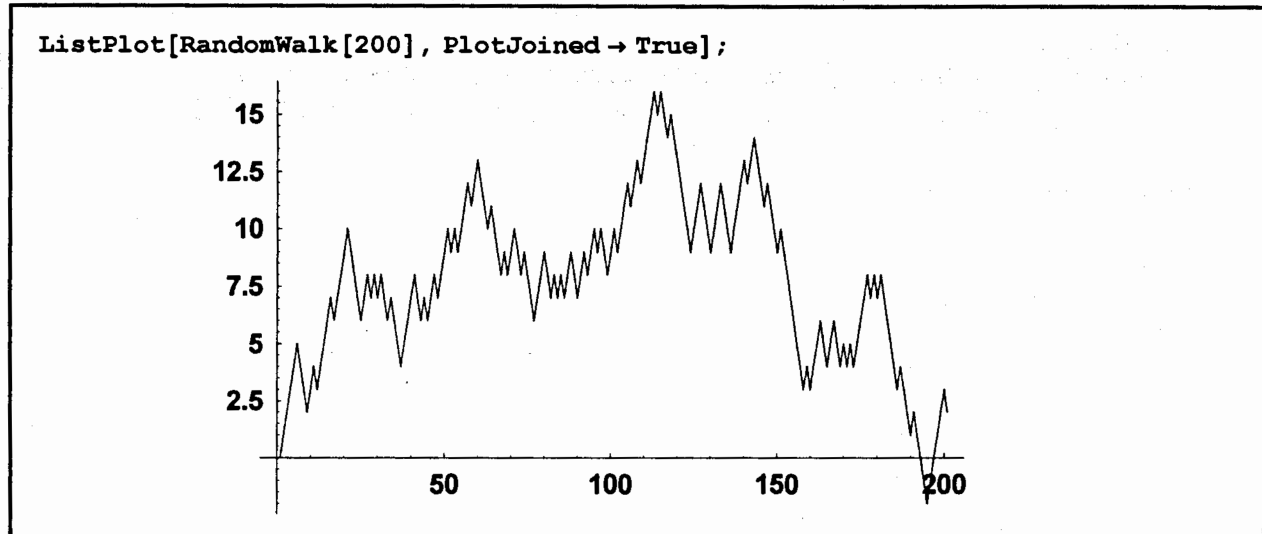
- **Even single-line *Mathematica* programs can perform complex operations.**

This program creates a *one-dimensional* random walk

```
RandomWalk[n_] := NestList[(# + (-1)^Random[Integer])&, 0, n]
```

Here is a plot of the first 200 steps.

```
ListPlot[RandomWalk[200], PlotJoined → True];
```

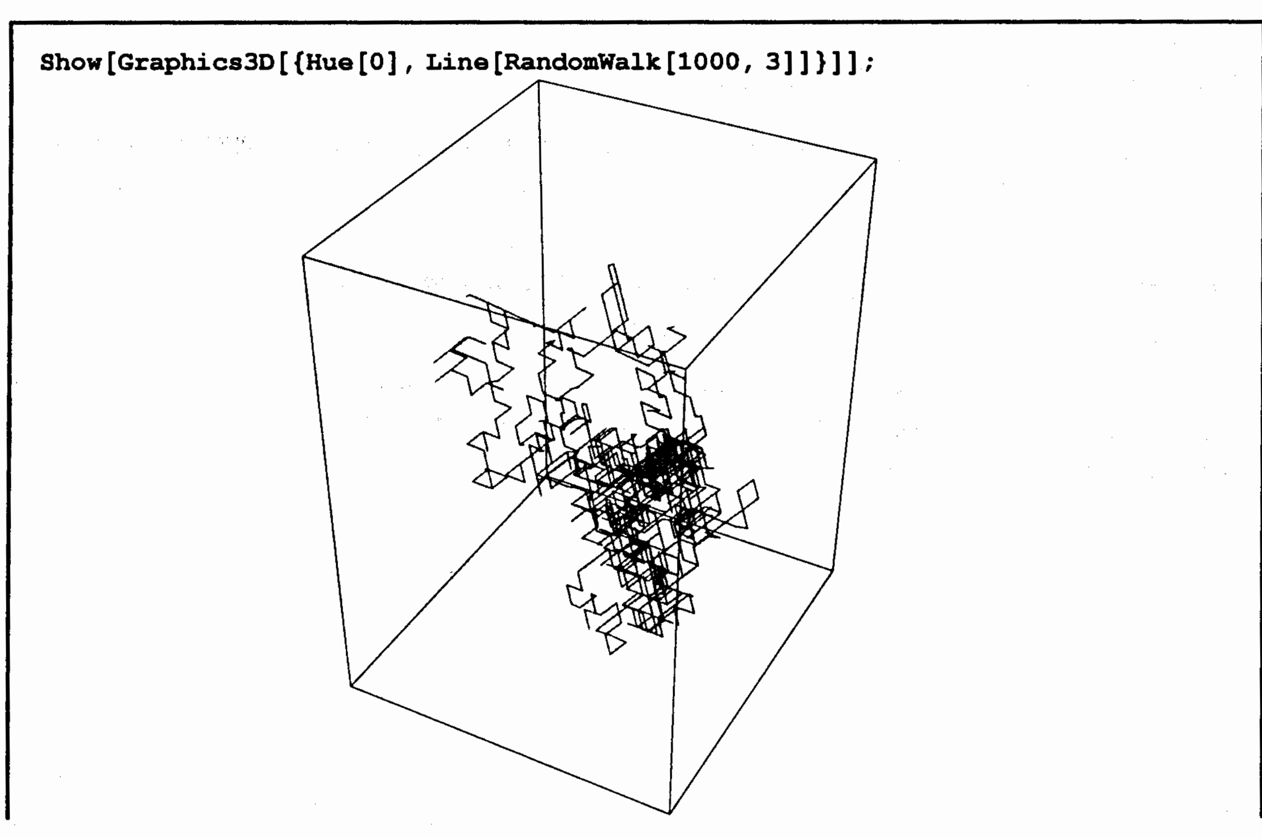


The directness of *Mathematica* programs makes a generalization quite easy. The subsequent program produces a *random walk in d dimensions*.

```
RandomWalk[n_, d_] :=
 NestList[ (# + (-1)^Table[Random[Integer], {d}] )&, Table[0, {d}], n]
```

Here is the *three-dimensional* random walk.

```
Show[Graphics3D[{Hue[0], Line[RandomWalk[1000, 3]]}]];
```



- ***Mathematica* is a uniquely scalable language - suitable for programs of any size**

Cellular Automaton (*Game of Life*)

Here is a direct program for one step in the **Life**.cellular automaton.

```
LifeStep[a_List] :=
 MapThread[If[(#1 == 1 && #2 == 4) || #2 == 3, 1, 0]&,
   {a, Sum[RotateLeft[a, {i, j}], {i, -1, 1}, {j, -1, 1}]}, 2]
```

Here is an alternative highly optimized program acting on cell lists.

```
LifeStep[list_] :=
 With[{u = Split[Sort[Flatten[Outer[Plus, list, N9, 1], 1]]]},
   Union[Cases[u, {x_, _, _} -> x],
      Intersection[Cases[u, {x_, _, _, _} -> x], list]]]

N9 = Flatten[Array[List, {3, 3}, -1], 1];
```

Mathematica makes it easy to build up programs from *components*. In this example the parts for the simulation of a *cellular automaton* are put together.

```
Clear[CAStep]
```

```
CenterList[n_Integer] := ReplacePart[Table[0, {n}], 1, Ceiling[n/2]]

ElementaryRule [num_Integer] := IntegerDigits[num, 2, 8]

CAStep[rule_List, a_List] :=
                rule[[ 8 - (RotateLeft[a] + 2 (a + 2 RotateRight[a])) ]]
```

Mathematica has a *compiler* for optimizing programs that work with lists and numbers. Here is a definition of **CAStep** which resorts to Compile.

```
CAStep = Compile[{{rule, _Integer, 1}, {a, _Integer, 1}},
            rule[[ 8 - (RotateLeft[a] + 2 (a + 2 RotateRight[a])) ]]];
```

```
CAEvolveList[rule_List, init_List, t_Integer] :=
                              NestList[CAStep[rule, #]&, init, t]
```

```
CAGraphics[history_List] := Graphics[Raster[1 - Reverse[history]],
                                      AspectRatio -> Automatic]
```

This example demonstrates how the program runs.

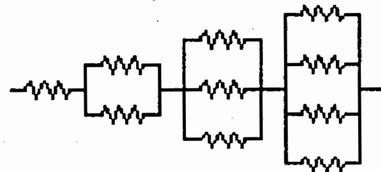
```
Show[ CAGraphics[
                    CAEvolveList[
  ElementaryRule[30], CenterList[201], 150]   ] ];
```

Mathematica programs are often a direct translation of material in textbooks. Hereare definitions for the impedance of an electric circuit.

Impedance of a Circuit

```
Impedance[Resistor[r_], ω_]  :=   r
Impedance[Capacitor[c_], ω_] :=  1/(ⅉ ω c)
Impedance[Inductor[l_], ω_]  :=  ⅉ ω l

Impedance[SeriesElement[e_], ω_]    :=
         Apply[Plus, Map[Impedance[#, ω]&, e]]
Impedance[ParallelElement[e_], ω_] :=
         1/Apply[Plus, 1/Map[Impedance[#, ω]&, e]]
```

The definitions above are then used for the following calculation of a circuit.



```
Remove[Rn];
```

```
                                    Rn := Subscript[R, n]
Impedance[SeriesElement[
          Table[ ParallelElement[
            Table[SeriesElement[{Resistor[Rn]}], {n}]
                                   ], {n, 1, 4}]
                 ], ω]
```

$$R_1 + \frac{R_2}{2} + \frac{R_3}{3} + \frac{R_4}{4}$$

- **In *Mathematica* algorithms can be expressed clearly.**

Golden Ratio $\frac{(1+\sqrt{5})}{2}$

```
GoldenRatio
GoldenRatio
```

Both programs approximate the *Golden Ratio* up to **k** digits.

```
ϕ1[k_] := 1 + FixedPoint[N[1/(1 + #1), k]&, 1]
```

```
ϕ2[k_] := FixedPoint[N[√(1 + #1), k]&, 1]
```

```
{ϕ1[50], ϕ2[50], N[GoldenRatio, 50]}
{1.6180339887498948482045868343656381177203091798058    ,
 1.6180339887498948482045868343656381177203091798057    ,
 1.6180339887498948482045868343656381177203091798058    }
```

- ***Mathematica* programs allow a unique combination of mathematical and computational notation.**

Subsequent definitions correspond to a recently discovered approximation for *prime numbers*.

```
Ω[n_] := Apply[Plus, Map[Last, FactorInteger[n]]]

μ[n_] := MoebiusMu[n]
```

$$P[x_] := -\sum_{k=1}^{\lfloor Log[2,x] \rfloor} \mu[k] \sum_{n=2}^{\lfloor x^{1/k} \rfloor} \mu[n]\, \Omega[n] \left\lfloor \frac{x^{1/k}}{n} \right\rfloor /;\ x > 0$$

Here is a comparison of the approximation **P[n]** with the built-in function **PrimePi[n]**, which determines the *number of primes* ≤ **x**.

```
{ P[100000], PrimePi[100000] }
{9592 , 9592 }
```

Mathematica Notebooks

Mathematica Programming Language

Prof. Dr. Robert Kragler
FH Ravensburg-Weingarten */ University of Applied Sciences*
kragler@rz.fh-weingarten.de

Programming Efficiency

Create a Sequence of 0's and 1's

- Problem: All sequences of Length 3
  Result wanted : { [0,0,0],[1,0,0],[0,1,0],[1,1,0],[0,0,1],[1,0,1],[0,1,1],[1,1,1] }

For sequences consisting of digits **0** and **1** there are 2^n possible arrangements. Solutions for *Mathematica*, Maple, Macsyma, Axiom and Reduce are :

- **Mathematica**

```
sequences[n_] := Distribute[Table[{0, 1}, {n}], List]
 sequences[3]
 sequences[4]
{{0, 0, 0}, {0, 0, 1}, {0, 1, 0}, {0, 1, 1}, {1, 0, 0}, {1, 0, 1},
 {1, 1, 0}, {1, 1, 1}}
```

```
{{0, 0, 0, 0}, {0, 0, 0, 1}, {0, 0, 1, 0}, {0, 0, 1, 1}, {0, 1, 0, 0},
 {0, 1, 0, 1}, {0, 1, 1, 0}, {0, 1, 1, 1}, {1, 0, 0, 0}, {1, 0, 0, 1},
 {1, 0, 1, 0}, {1, 0, 1, 1}, {1, 1, 0, 0}, {1, 1, 0, 1}, {1, 1, 1, 0}, {1, 1, 1, 1}}
```

- **Maple**

```
sequences := n -> if (n = 0) then []
                  elif (n = 1) then [[0],[1]]
                  else map(x->([op(x),0],[op(x),1]),sequences(n-1))
                   fi
```

- **Macsyma**

```
sequences(n):=
  if n=0 then []
  else if n=1 then [[0],[1]]
   else apply('append,map(lambda([l],[endcons(0,l),
                              endcons(1,l)]),
                 sequences(n-1)))$
```

- **Axiom**

```
sequences n ==
  one? n => [[0], [1]]
  concat [[concat(0, s), concat(1, s)]
                  for s in sequences(n-1)]
```

- **Reduce**

```
procedure sequences n;
if n = 1 then {{0},{1}}
else for each s in sequences(n - 1) conc {0 . s,1 . s};
```

List all Elements larger than all Preceding Elements

- Problem : maxima[{ 3, 2, 6, 4, 8, 1}]
  Result wanted : { 3, 6, 8 }

- **Mathematica**

```
maxima[x_] := Union[Accumulate[Max, x]]
maxima[{3, 2, 6, 4, 8, 1}]
{3, 6, 8}
```

- **Maple**

```
maxima := proc(l)
local mx, i;
mx := op(1, l)-1;
[ seq( proc(m, v) if ( eval(m) < v ) then m := v;
       v else NULL fi end('mx',i), i=l) ]
end;
```

- **Macsyma**

```
maxima(l):= maxima2(l,minf)$

maxima2(l,x):=
        if l=[] then []
        else if first(l)>x
                then cons(first(l),maxima2(rest(l),first(l)))
        else maxima2(rest(l),x)$
```

- **Axiom**

```
maxima x == [x.i for i in 1..#x |
          every?(a +-> a < x.i, first(x, (i-1)::NNI))]
```

- **Reduce**

```
procedure maxima u;
first u . maximal(rest u,first u);

procedure maximal(u,v);
if u = {} then {}
else if first u > v then first u . maximal(rest u,first u) else
maximal(rest u,v);
```

Encode Sequences in Lists (Run-length Encoding)

▪ Problem :	runEncode[{a,a,b,c,c,c,a}]
Result wanted :	{ {a, 2}, {b, 1}, {c, 3}, {a, 1} }

Determine from a given set the run-lengths of equal characters following each other.

- **Mathematica**

```
runEncode[x_List] :=
 ({#1, 1}&) /@x //. {u___, {v_, r_}, {v_, s_}, w___} → {u, {v, r + s}, w}
```

```
runEncode[{a, a, b, c, c, c, a}]
{{a, 2}, {b, 1}, {c, 3}, {a, 1}}
```

- **Maple**

```
 run_encode := l -> if (l = []) then [] else [encode_block(l,1,1)] fi:
encode_block := (l,i,c)-> if (i >= nops(l)) then [l[i],c]
                          elif (l[i] = l[i+1]) then
encode_block(l,i+1,c+1);
                          else [l[i],c],encode_block(l,i+1,1);
                          fi:
```

- **Macsyma**

```
 elem_count(x,l):=
      if l=[] then 1
      else if first(l)=x then 1+elem_count(x,rest(l))
      else 1$

 run_encode(l):=
      if l=[] then []
      else block([c: elem_count(first(l),rest(l))],
             cons([first(l),c],run_encode(rest(l,c))))$
```

- **Axiom**

```
lcount(a,x) == (empty? x => 1; a = first x => 1 + lcount(a, rest x); 1)
runEncode x ==
  empty? x => [];
  cons([a := first x, n := lcount(a, rest x)],
                              runEncode rest(x, n))
```

- **Reduce**

```
procedure runencode lst;
if lst = {} then {} else runencode1(first lst,rest lst,1);

procedure runencode1(u,v,n);
if v = {} then {{u,n}}
else if u = first v then runencode1(u,rest v,n + 1) else {u,n} .
runencode v;
```

Mathematica Notebooks

Mathematica Programming Language

Prof. Dr. Robert Kragler
***FH Ravensburg-Weingarten** / University of Applied Sciences*
kragler@rz.fh-weingarten.de

Programming Styles

- ***Initializations***

General Remarks on the *Mathematica* Programming Language

The success of *Mathematica* relies to some extend on the fact that in adddition to the three columns *Symbolic* and *Numerical Calculation* and *Graphics* there is a powerful *high-level Programming Language* available which comprises very different *programming concepts*.

An occasional *Mathematica* user may do his calculations with *Mathematica* in a dialogue mode where already in this case he is offered the possibility of processing results of previous calculations and nesting *Mathematica* expressions. With these features, however, the possibilites provided by *Mathematica* are far not exhausted.

Mathematica is more than a handy pocket calculator. Auxilliary to more than 1000 commands implemented in the kernel *Mathematica* comprises a *complete high-level programming language* (comparable to C or FORTRAN etc.) allowing

- to access all built-in functions of *Mathematica*
- to extend the functionality of *Mathematica* by additional user-written procedures if required

Different *Mathematica* Notations

A special feature of the *Mathematica* programming language is the great number of synonymous notations and the possibility resulting from this feature, to employ the *syntax* as means to emphasize or especially point out certain aspects or sequentialization etc. If suitable *postfix* , *prefix* or *infix* and *bracket* notations may be applied exclusively or in combination.

f[*x*, *y*]		standardform für f(x,y)
Prefix[f[*x*]]	f @ x	standard *Prefix* notation for f(x)
Postfix[f[*x*]]	x // f	standard *Postfix* notation for f(x)
Infix[f[*x*, *y*, ...]	x~f~y~f~...	standard *Infix* notation for f(x,y)
Prefix[f[*x*], h]		Prefix form h x
Postfix[f[*x*], h]		Postfix form x h
Infix[f[*x*, *y*, ...], h]		Infix form x h y h ...
PrecedenceForm[*expr*, *n*]		*expr* to be parenthesized with precedence *n*

As will be shown below the **Postfix** operator (//) may be used to emphasize the *sequential execution* of serveral operators from left to right ; the usage of the **Prefix** operator (@) points out the sequence in which functions are applied , whereas the **parenthesis** notation common in other programming languages is more suitable for functions of several arguments or for augmenting nesting structures. The **Infix** operator (~) may be used as a (binary) operator acting between only *two* operands resp. parameters as is the case in other notations too. According to the need even *within* a *Mathematica* statement the *mix of different notations* is possible. This offers syntax possibilities similar to natural languages.

■ Examples to model mathematical notation

When there is an output form involving operators the question arises of whether the arguments of some of them should be parenthesized. In general this depends on the ranking order (*precedence*) of the particular operator. With `PrecedenceForm[...]` one can specifiy the precedence level to assign to each operator a number between 1 and 1000. The higher the precedence level for an operator, the less it needs to be parenthesized. Using `Infix[exp,h,precedence,grouping]` it is possible to affect the parenthesizing of the output form. The precedence is determined by integer numbers; for `OutputForm` some precedence levels are given as follows

Connection	*precedence level*	*example*
Dot	210	`x.y.z`
Power	190	`x^y`
Times	150	`x*y*z`
Plus	140	`x + y + z`
Equal	130	`x == y`
Set	60	`x = y`

Possible *specifications* for grouping (*associativity*) of input forms with *same* precedence level are

NonAssociative	not associative	always brackets
None	always associative	(never bracket)
Left	left associative	e.g. `(a/b)/c`
Right	right associative	e.g. `a^(b^c)`

Prefix operator

```
? Prefix
```

```
Prefix[f[expr]] prints with f[expr] given in default prefix form:
  f @ expr. Prefix[f[expr], h] prints as hexpr.
```

An equivalent short for **Prefix[...]** is the operator @ (*function application*)

```
Prefix[ f[x + y] ]
f@(x + y)
```

```
f @ x + y
f @ (x + y)
y + f[x]
```

```
f[x + y]
```

```
Prefix[ f[x], "ℒ∘"]
Prefix[ f[2 + n], "ℒ∘"]
ℒ∘ x
```

```
ℒ∘ (2 + n)
```

Postfix operator

```
? Postfix
```

```
Postfix[f[expr]] prints with f[expr] given in default postfix form:
  expr // f. Postfix[f[expr], h] prints as exprh.
```

```
Postfix[ f[2 + n], "∘ℋ"]
2 + n ∘ℋ
```

Because the precedence level is with 160 larger than for Plus (+) one should expect a result which differs from the previous one.

```
Postfix[ f[2 + n], "∘ℋ", 160]
```

Infix operator

```
? Infix
```

```
Infix[f[e1, e2, ... ]] prints with f[e1, e2, ... ] given in default
  infix form: e1 ~ f ~ e2 ~ f ~ e3 ... . Infix[expr, h] prints with
  arguments separated by h: e1 h e2 h e3 ... .
```

```
Infix[ f[x, y]]
x ~f~ y
```

```
{a, b, c} ~Join~ {d, e} ~Join~ {f, g, h}
{a, b, c, d, e, f, g, h}
```

```
Infix[ f[x, y], "⊕"]
x⊕y
```

```
Format[ vectorProduct[a_, b_] ] := Infix[{a, b}, "x"]
   (* Format[vectorProduct[a_,b_] ]:=Infix[{a,b},"x",160,None] *)
vectorProduct[a, b]
a×b
```

```
Infix[f[a + b, c + d], " ⊕ ", 135, None]
Infix[f[a + b, c + d], " ⊕ ", 145, None]
a + b Z c + d
```

```
(a + b) Z (c + d)
```

```
dot[a_, b_] := Infix[dot[a, b], "°", 120, None]
dot[dot[a, b], dot[c, d]]
a°b°c°d
```

```
dot[a_, b_] := Infix[dot[a, b], "°", 120, NonAssociative]
dot[dot[a, b], dot[c, d]]
(a°b)°(c°d)
```

```
dot[a_, b_] := Infix[dot[a, b], "°", 120, Left]
dot[dot[a, b], dot[c, d]]
```

```
a°b°(c°d)
```

Precedence

```
? PrecedenceForm
```

```
PrecedenceForm[expr, prec] prints with expr parenthesized as it
  would be if it contained an operator with precedence prec.
```

The following output of a list will be decorated with the `Infix` operator " ⇔ "

```
s = Infix[{a, b, c, d}, "⇔"]
a⇔b⇔c⇔d
```

Because of its default the `Infix` operator has a precedence level higher than `Power[...]`, i.e. exponentiation. Hence, squaring the expression s defined above with `Infix` operator " ⇔ " one obtains the result.

```
s^n
(a⇔b⇔c⇔d)^n
```

Due to the fact that *precedence level* for **Add** (+) is 140 there will be different results depending on whether the precedence level is higher or lower than 140

```
a + PrecedenceForm[c d, 120]
a + (c d)
```

```
a + PrecedenceForm[c d, 200]   (* should be a + c d *)
a + (c d)
```

- **Example** $|\sin(\frac{\pi}{4})|$

The following variations show the same content but in different notation

sequential execution in the fashion of UNIX pipes

```
(-π/4) // Sin // Abs // N
0.707107
```

functional (operational) way of viewing with final clause "...and then all numeric"

```
Abs @ Sin @ (-π/4) // N
0.707107
```

procedural way of viewing specifying the number of digits wanted of the numeric result

```
N[Abs[Sin[-π/4]], 20]
0.70710678118654752440
```

mixed form with less parentheses

```
N[Abs @ Sin[-π/4], 20]
0.70710678118654752440
```

mixed form with final clause "...with absolute value"

```
N[Sin[-π/4] // N, 20]
0.70710678118654752440
```

Overview of *Mathematica* Programming Styles

Mathematica as a powerful and (at the same time) elegant programming language support essentially all programming styles known from other high-level programming languages. In an first approach there will be only an overview of the variety of programming styles to be realized with *Mathematica;* in a second step there will be a discussion of technical details.

Algorithms from *procedural programming* languages like FORTRAN, C, Pascal, Modula-2 etc. can be transscripted into *Mathematica* code in a straightforward manner. However, this is not always the most elegant way to do so. Because of the possibilites which are offered in *functional* and *rule-based* programming languages mathematical algorithms are in most cases more compact and more efficient.

The *essential programming styles* supported by *Mathematica* are

- procedural programming (FORTRAN, Pascal, C like)
- recursive programming (C, Lisp like)
- functional programming (APL, Lisp like)
- rule-base programming (Prolog, C++ like)

A simple example - the *factorial* function - serves as illustration for the different programming styles :

Factorial in Procedural Programming Style

Procedural

Procedural programming in the style of FORTRAN, Pascal or C language relies on *structural elements* such as blocks, conditions, loops and control structures (as well as iteration and recursion).

```
facFortran[n_] := Block[{s = 1},
                        Do[ s *= i, {i, n}]; s]
```

```
facFortran[5000] // N // Timing
{3.46 Second, 4.22857792660554 × 10^16325}
```

Factorial in Recursive Programming Style

Recursive

Modern programming languages like C (and recently FORTRAN 90 too) allow *recursive* function calls. Hence, some mathematical problems can be solved in a more concise way if formulated in a recursive fashion.

```
facC[n_] := If[ n == 1, 1, n facC[n - 1] ]
$RecursionLimit = $IterationLimit = 10000;
```

```
facC[5000] // N // Timing
{3.4 Second, 4.22857792660554 × 10^16325}
```

Factorial in Functional Programming Style

Functional

Functional programming à la APL or Lisp make use of the concept of *functional* and *structural operators*

```
facAPL[n_Integer] := Times @@ Range[n]
```

```
facAPL[5000] // N // Timing
{2.31 Second, 4.22857792660554 × 10^16325}
```

Factorial in Rule-based Programming Style

RuleBased

Rule-based programming in the style of Prolog or C++ makes use of *pattern matching* and *transformation rules*. This programming style is most suitable when there are only few rules to be formulated in order to transform instructions from one form into another one. Each rule is related to a model and acts as an operator on the mapping of the model. Most rules can be constructed from a direct conversion of the corresponding mathematical formulae. Rules can be named so that they can be activated if needed or implemented as a global rule base to be always available

```
facProlog[n_] := n facProlog[n - 1]
facProlog[0] := 1
```

```
? facProlog
```

```
Global`facProlog

facProlog[0] := 1

facProlog[n_] := n * facProlog[n - 1]
```

```
facProlog[5000] // N // Timing
{3.13 Second, 4.22857792660554 × 10^16325}
```

Programming Styles in Detail

Obviously *Mathematica* as a programming language does not tie down the user neither to a single programming style nor to a certain notation. The essential advantage in comparison to traditional programming languages is however that *Mathematica* is an interactive *interpreter* system.
(Only for some simple expressions it is possible to create compiled functions which speeds up evaluation. A built-in *compiler* in the *Mathematica* kernel translates expressions into pseudocode which is used by a subsequent pseudocode interpreter to evaluate the compile functions.)
This allows an experimental proceeding to call individual functions separately and to test them without the necessity of writing a complete program. Thus, the user can build up complicated programs based on simple functions which in turn make use of fundamental operations of *Mathematica*. Many programs of that kind are available as *Mathematica* packages for very different applications to be loaded into the *Mathematica* kernel if needed.

Below follows a list of *different programming styles* :

Procedural Programming

First of all the *procedural programming* (as known from Pascal or C) should be dealed with. In this programming style functions may call each other by value or by reference. The usual *constructs* available are e.g. *loops*, *conditionals*, *local variables*, *arguments with default values* and *recursive calls*. This programming style is well-known. And the most important control structures are **Do**, **While** and **For** as well as conditionals such as **If**, **Which** and **Switch**.

In order to define *local variables* within functions or procedures one uses the mechanism of **Module** (with lexical scoping) or **Block** (dynamic scoping)

```
c = b^2;
```

The local value for ***b*** in the block is used *throughout* the evaluation of ***b+c***

```
Block[{b = a}, b + c]
a + a^2
```

Here only the ***b*** that appears *explicitly* in ***b+c*** is treated as a local variable

```
Module[{b = a}, b + c]
a + b^2
```

- **Example : Sign function sign(x)**

```
signum[x_] := If[x ≤ 0, 0, 1]
signum[10]
1
```

```
Plot[signum[i], {i, -5, 5},
     PlotStyle → {{RGBColor[1, 0, 0], AbsoluteThickness[2]}},
     AxesLabel → {"x", "Sign (x)"}, AspectRatio → 0.4,
     Background → GrayLevel[0.75]];
```


In order to point out the *intricacy* of this programming style consider the task to *represent numbers with respect to the base 256*. The realization in *procedural style* could be as follows

```
makeList1[num_Integer] :=
      Module[{res = {}, rem = num},
             While[rem ≠ 0, PrependTo[res, Mod[rem, 256]];
             rem = Quotient[rem, 256]];
             res ]
```

```
coeff1 = makeList1[111111111111]
{25, 222, 189, 1, 199}
```

To test the correctness of the forgoing procedure terms of the form $c_i\, 256^i$ are summed up to yield the decimal number

```
i1 = Length[ coeff1 ];
s = Sum[coeff1[[i]] HoldForm[256]^(i1-i), {i, 1, i1}]
199 + 256 + 189 256^2 + 222 256^3 + 25 256^4
```

$$s = \sum_{i=1}^{i1} \text{coeff1}[\![i]\!]\ \text{HoldForm}[256]^{(i1-i)}$$

$$199 + 256 + 189\ 256^2 + 222\ 256^3 + 25\ 256^4$$

```
s // ReleaseHold            (* or ReleaseHold //@ s  *)
111111111111
```

This naive *procedural* formulation is very unsatisfying because 2 local variables *rem* and *res* are used and the flow of the program is controlled by a *While*-loop. Using a *recursive* algorithm however the program can be written in a more concise and elegant way.

Recursive Programming

All modern programming languages support *recursive function calls*. Using this concept one can solve more concisely those mathematical problems which are described recursively. Thus one avoids an explicit control of the flow of the program.

A solution of the example above is achieved in a recursive way with the help of two definitions

```
makeList2[0] = {};
makeList2[num_Integer] :=
 Append[makeList2[ Quotient[num, 256] ], Mod[num, 256]]
```

```
makeList2[111111111111]
{25, 222, 189, 1, 199}
```

In general a recursive program is *faster* although the book-keeping of the recursion requires some additional computational effort. What is in favor of a *recursive* program style is its clearness and the ease to read.

Efficiency

In the case of simple problems with *recursive* solutions mostly *iterative* solutions will exist too which are more efficient. For complicated problems, however, it is preferable using a short recursive algorithm instead of a long iterative one.

In algorithms with recursive calls to be used more than once the *storage* of intermediate results (e.g. `f[n_] = f[n] = expression`) avoids multiple computations. As prominent example serve the *Fibonacci numbers*

Fibonacci Numbers

A version without intermediate storage require more computation

```
fib1[0] = 1; fib1[1] = 1;
fib1[n_] := fib1[n - 1] + fib1[n - 2]
fib1[20] // Timing
{2.2 Second, 10946}
```

Storing intermediate results makes the calculation more efficient

```
fib2[0] = 1; fib2[1] = 1;
fib2[n_] := fib2[n] = fib2[n - 1] + fib2[n - 2]
```

```
fib2[20] // Timing
{0. Second, 10946}
```

```
$RecursionLimit = 1000;
fib2[200] // Timing
{0.17 Second, 453973694165307953197296969697410619233826}
```

In the present example, however, an *iterative* algorithm is much faster which starts with the calculation at f[2] in increasing order and calculates from the two preceding value the next one

```
fib3[0] = 1; fib3[1] = 1;
fib3[n_] :=
   Module[{i, fn = 1, fn1 = 1, fn2},
         Do[ fn2 = fn1; fn1 = fn;
             fn = fn1 + fn2, {i, 2, n}];
         fn ]
```

```
fib3[200] // Timing
{0.06 Second, 453973694165307953197296969697410619233826}
```

Functional Programming

An *interactive* programming environment as provided by *Mathematica* is most suitable to do a step-by-step of the results. *Procedural programming* is unsuitable for this because there occur mostly local variables and control structures which make it difficult to insert pieces of program code.

However, the technique of *functional programming* will ease a stepwise construction of a program. As building block one uses functions, their results are handed over to other functions in terms of arguments or parameters. The entire program consists of expressions in terms of nested function calls. Control structures in functional programs may often be substituted by iterative functions such as **Sum** etc. In this context an important role is taken by so-called anonymous or ***pure functions*** which are applied on lists together with *Mathematica* functions such as **Map** (**f /@ expr**), **Apply** (**f @@ expr**)

▪ Example : Approximation of Euler number ⅇ

The following functional program approximates the *Euler number* ⅇ up to 50 decimals

```
N50 = N[#, 50]&;
```

```
Sum[1 / i!, {i, 0, 60}] // N50 // Timing
{0.05 Second, 2.7182818284590452353602874713526624977572470937000}
```

As regards to efficiency this procedure is comparable with the C type program variant with a **For** loop

```
r = 1;
For[i = 1, i ≤ 60, i++,
    For[j = 1; s = 1, j ≤ i, j++, s *= j];
    r += 1/s];
r // N50
2.7182818284590452353602874713526624977572470937000
```

▪ Example : Newton's Approximation for $\sqrt{3}$

Even *Newton's approximation* of the square root is more transparent as *fixed point procedure* using a pure function

```
a = 3;
FixedPoint[1/2 (#1 + a/#1)&, 1.] // N50
1.732050807568877
```

$$\texttt{FixedPoint}\left[\frac{1}{2}\left(\#1+\frac{a}{\#1}\right)\&,\ 1.\right]$$

rather than a `While` loop in the style of a Pascal program

```
x = 1.; y = 2.;
While[! (x == y), y = x; x = 1/2 (x + a/x)];
x // N50
1.732050807568877
```

Concept of pure (or anonymous) functions

Sometimes it is rather favorable to operate with user-specified functions on elements of lists. This can be done with the command **Map** or with its shortcut /@ e.g.

```
Clear[x, f, g, a, b, c, d]
```

```
h[x_] := f[x] + g[x]
h /@ {a, b, c, d}
{f[a] + g[a], f[b] + g[b], f[c] + g[c], f[d] + g[d]}
```

However, it is too involved defining a function (in this case h[x]) which is only needed to map it with **Map** resp. /@ . For functional operators such as **Map** or **Nest** one always has to specifiy a function which is applied on some operands (i.e. variables or lists). In the example above the name of a function **h** was used in order to define the function. Yet, it is too much trouble to define a function only for using it once. *Mathematica* allows the construction of so-called *anonymous or pure functions*. These objects have no explicit function name and are primarily used to act on the arguments of other *Mathematica* functions.

The following *pure function* will give the same result as above

```
Function[x, f[x] + g[x]] /@ {a, b, c, d}
{f[a] + g[a], f[b] + g[b], f[c] + g[c], f[d] + g[d]}
```

In general the *syntax* of a *pure function* is :

Function[body][arg] **Function[body][arg1, arg2,...]** **(body)& [arg]** **(body)& [arg1, arg2,...]**	Result of the calculation of **arg1,...** is substituted for all occurrences of **#1,...** in **body** , the resulting expression will be returned after evaluation

The shortcut notation for pure functions uses # as placeholder for the formal parameters (resp. **#1**, **#2**,... if there is more than one parameter), whereas the name of these formal parameters is irrelevant. Only the function definition has to be specified and terminated with the character **&** .

```
(f[#] + g[#]&) /@ {a, b, c, d}
{f[a] + g[a], f[b] + g[b], f[c] + g[c], f[d] + g[d]}
```

The notation **Function[x,f[x]+g[x]]** and **(f[#]+g[#])&** are equivalent. The second formulation strips down the function completely to its *functionality* and hence justifies the name *pure function.* Obviously, pure functions are very close to mathematical notation of *operators*.

Rule-based Programming

Even procedural programming only makes use of a part of the possibilities the *Mathematica* language offers. To go beyond this, *rules* can be defined which can be applied automatically or manually. In this context even the built-in functions of *Mathematica* can be defined or extended in a newly.

Programming with *definitions* and *transformation rules* distinguishes *Mathematica.* In this way it is quite easy to implement into *Mathematica* straightforward the mathematical knowledge from tables of formulae etc. For this kind of problems *rule-based programming* is the most powerful programming style. Whenever in some cases only a few number of rules have to be applied this programming style is the most suitable one. If, however, the set of all rules should be used for all possible cases then functional resp. procedural programming is more favorable.

- **Examples of Rule-based Programming**

Manual substitution rule

```
x^2 /. x → a
a^2
```

where " /. " has the meaning of

```
?/.
expr /. rules applies a rule or list of rules in an attempt to
  transform each subpart of an expression expr.
```

Factorial function

```
fak[n_] := n fak[n - 1]
fak[0] = 1;
fak[x_?Negative] = "Not defined!!";
```

Thus *Mathematica* contains the following set of rules for the calculation of the factorial function :

```
?fak
Global`fak

fak[0] = 1

fak[(x_)?Negative] = "Not defined!!"

fak[n_] := n * fak[n - 1]
```

```
fak[50]
```

```
30414093201713378043612608166064768844377641568960512000000000000
```

```
fak[-1]
Not defined!!
```

■ Definition of a linear function

```
q[x_] + q[y_] ^:= q[x + y]      (* ^:= is used for UpSetDelayed *)
```

```
?q
```

```
Global`q

q[x_] + q[y_] ^:= q[x + y]
```

```
q[a] + q[b] + q[c]
q[a + b + c]
```

■ Rules for differentiating algebraic functions

Symbolic calculation will be illustated in the subsequent example by the most famous *recursive* definition, namely by differential calculus.

```
Clear[𝒟]

𝒟[c_?NumberQ, x_]    := 0
𝒟[x_, x_]           := 1
𝒟[f_^n_?NumberQ, x_] := n f^(n-1) 𝒟[f, x]
𝒟[c_ f_, x_]         := c 𝒟[f, x] /; FreeQ[c, x]
𝒟[f_ + g_, x_]       := 𝒟[f, x] + 𝒟[g, x]
𝒟[f_ g_, x_]         := 𝒟[f, x] g + f 𝒟[g, x]
𝒟[f_/g_, x_]         := (𝒟[f, x] g - f 𝒟[g, x])/g^2
𝒟[1/g_, x_]          := -𝒟[g, x]/g^2
```

Now the *set of rules for differentiation* 𝒟[..., x] is specified as follows

```
?𝒟
```

```
Global`𝒟
```

```
𝒟[(c_)?NumberQ, x_] := 0

𝒟[x_, x_] := 1

𝒟[(g_)^(-1), x_] := -(𝒟[g, x]/g^2)

𝒟[(f_)^(n_)?NumberQ, x_] := n*f^(n - 1)*𝒟[f, x]

𝒟[(c_)*(f_), x_] := c*𝒟[f, x] /; FreeQ[c, x]

𝒟[(f_) + (g_), x_] := 𝒟[f, x] + 𝒟[g, x]

𝒟[(f_)/(g_), x_] := (𝒟[f, x]*g - f*𝒟[g, x])/g^2

𝒟[(f_)*(g_), x_] := 𝒟[f, x]*g + f*𝒟[g, x]
```

with

```
? NumberQ
```

```
NumberQ[expr] gives True if expr is a number, and False otherwise.
```

```
? FreeQ
```

```
FreeQ[expr, form] yields True if no subexpression in expr matches
  form, and yields False otherwise. FreeQ[expr, form, levelspec]
  tests only those parts of expr on levels specified by levelspec.
```

Application of simple differentiation rules on algebraic functions using as a shortcut notation the following pure function "operator"

```
𝒟x := 𝒟[#, x]&
```

$\mathcal{D}_x$ /@ $\left\{1,\ x,\ x^4,\ (x+2\,x^3)^4,\ \frac{x}{x-1}\right\}$

$\left\{0,\ 1,\ 4\,x^3,\ 4\,(1+6\,x^2)\,(x+2\,x^3)^3,\ -\frac{1}{(-1+x)^2}\right\}$

Definition of the functions **f (x)** and **g (x)**

$f[x_] := \frac{x^3}{2};\qquad g[x_] := 1+x$

```
𝒟x /@ { f[x], g[x], f[x] - g[x] }
```

$\left\{\frac{3\,x^2}{2},\ 1,\ -1+\frac{3\,x^2}{2}\right\}$

$\text{Trace}[\,\mathcal{D}[f[x^2],\ x],\ \mathcal{D}\,]$

```
{𝒟[x^6/2, x], 1/2 𝒟[x^6, x], {𝒟[x^6, x], 6 x^(6-1) 𝒟[x, x], {𝒟[x, x], 1}}}
```

```
𝒟x /@ { f[x] g[x], f[x]/g[x], 1/f[x], x/g[x] } // Simplify
```

$$\left\{\frac{1}{2} x^2 (3+4 x), \frac{x^2 (3+2 x)}{2 (1+x)^2}, -\frac{6}{x^4}, \frac{1}{(1+x)^2}\right\}$$

■ Laplace Transformation Rules

The following example demonstrate how the *essential properties* of the Laplace transformation can be specifies with only few transformation rules

```
Clear[a, b, c, n, s, t]
```

```
ℒ[c_, t_, s_]          := c/s /; FreeQ[c, t]
ℒ[a_ + b_, t_, s_]     := ℒ[a, t, s] + ℒ[b, t, s]
ℒ[c_ a_, t_, s_]       := c ℒ[a, t, s] /; FreeQ[c, t]
ℒ[t_^n_., t_, s_]      := n!/s^(n+1) /; FreeQ[n, t] && n > 0
ℒ[a_ t_^n_., t_, s_]   := (-1)^n ∂{s,n} ℒ[a, t, s] /; FreeQ[n, t] && n > 0
```

```
ℒ[a_/t_, t_, s_] := Module[{τ}, ∫_s^∞ ℒ[a, t, τ] dτ]
ℒ[a_. ⅇ^(b_. + c_. t_), t_, s_] := ℒ[a ⅇ^b, t, s - c] /; FreeQ[{b, c}, t]
```

```
ℒ[t^3 ⅇ^(-3 t), t, s]
```

$$\frac{6}{(3+s)^4}$$

The package "**Calculus`LaplaceTransform`**" a more complete rule base for the Laplace-transformation is contained which increases the time for evaluation however.

■ Definition of a user-specified logarithmic function log

```
log[1] = 0;
log[E] = 1;
log[x_ y_] := log[x] + log[y]
log[x_^n_] := n log[x]
```

```
{log[2 x^3], log[x^(2 y)], log[x^(y^2)]}
{log[2] + 3 log[x], 2 y log[x], y^2 log[x]}
```

The built-in function Log however cannot handle the case Log[x^y]

```
Log[x^y]
Log[x^y]
```

but can be *extended* by the following rule

```
Unprotect[Log, Power];
Clear[Log];
Log[x_^y_] ^:= y Log[x]
Protect[Log, Power];
```

```
{Log[x^(2 y)], Log[x^(y^2)]}
{2 y Log[x], y^2 Log[x]}
```

Programming with Binding Propagation

Transformation rules are always directed : rules which make use of a given pattern are applied to expressions. Sometimes, however, it is more suitable to work with bindings which are formulated as equations and are solved with `Solve[eqns, vars, elims]`. Thus, no special direction is distinguished; equations may be solved arbitrarily with respect to variables or parameters wanted.

The following examples will be studied:

Example : Solution of a system of equations

$$\text{eqn}_1 = \left\{ v_a == \frac{v_0\, r_3}{r_1 + r_2},\ v_a == v_i,\ g == \frac{v_0}{v_i} \right\};$$

The system of equations will be resolved with respect to g

```
Solve[eqn₁, g, {r₁, r₂}]
```

$$\left\{\left\{g \to \frac{v_0}{v_i}\right\}\right\}$$

Solution with respect to r_1 and in addition elimination of g

```
Solve[eqn₁, r₁]
Simplify[Solve[eqn₁, r₁, g]]
```

$$\{\{r_1 \to -r_2 + g\, r_3\}\}$$

$$\left\{\left\{r_1 \to -r_2 + \frac{r_3\, v_0}{v_i}\right\}\right\}$$

Example : Inclined Throw

```
Clear[x₀, y₀, z₀, v₀, g, α]
```

$$\text{eqns}_2 = \left\{ x == x_0 + v_0 \operatorname{Cos}[\alpha]\, t,\ y == y_0 + v_0 \operatorname{Sin}[\alpha]\, t - \frac{g\, t^2}{2},\ z == z_0 \right\};$$

Eliminating time t resultes in the parabolic trajectory.

```
Solve[eqns₂, y, t] // ExpandAll
```

$$\left\{\left\{y \to y_0 - \frac{g\, x^2 \operatorname{Sec}[\alpha]^2}{2\, v_0^2} + \frac{g\, x\, x_0 \operatorname{Sec}[\alpha]^2}{v_0^2} - \frac{g\, x_0^2 \operatorname{Sec}[\alpha]^2}{2\, v_0^2} + x \operatorname{Tan}[\alpha] - x_0 \operatorname{Tan}[\alpha]\right\}\right\}$$

Perhaps, the question is posed what are the initial conditions to be chosen so that the mass is flying after *1sec* in x- and y-direction *1 m*

```
Solve[eqns₂ /. {x → 1, y → 1, z → 0, t → 1}, {x₀, y₀, z₀}] // ExpandAll
```

```
{{x0 → 1 - v0 Cos[α], y0 → 1 + g/2 - v0 Sin[α], z0 → 0}}
```

Or, for given initial location (e.g. $x_0 = z_0 = 0$, $y_0 = 1$) and initial velocity ($v_0 = 1$), the shooting angle is wanted which is associated with the maximum range of throw. In this case the parabolic trajectory has to be investigated after elimination of time t.

```
initialCond = {x0 → 0, y0 → 1, z0 → 0, v0 → 1};
res = Solve[eqns2 /. initialCond, y, t]
{{y → 1/2 Sec[α]^2 (1 - g x^2 + Cos[2 α] + x Sin[2 α])}}
```

Then, one determines the x-value belonging to y=0 which is the end of the parabola.

```
sol = Solve[y == 0 /. res[[1]], x] // Simplify
{{x → (Sin[2 α] - Sqrt[4 g + 4 g Cos[2 α] + Sin[2 α]^2])/(2 g)},
 {x → (Sin[2 α] + Sqrt[4 g + 4 g Cos[2 α] + Sin[2 α]^2])/(2 g)}}
```

Now, the (relevant) solution should be maximized with respect to the angle α.

```
FindMinimum[ -x /. sol[[2]] /. g → 9.81, {α, 1}]
{-0.462887, {α → 0.21676}}
```

Logic Programming

Logic programming or *declarative programming* tries to write down declarations that express certain properties of the desired results without specifying the flow of control. *Pattern matching* and *backtracking* are used to solve an instance of the problem.

Logic programming languages such as Prolog operate with transformation rules too. However, in contrast to *Mathematica* they are able to process several rules *in parallel*. The advantage of logic programming is based on the fact that no particular path of solution has to specified for a calculation. However, one pays for this kind of proceeding by computation time which is hard to predict or even un-predictable. *Mathematica* dimishes this disadvantage by testing the rules known one after the other and applying the first (!) one which works. Often *Mathematica* tries several possibilities in analogy to logic programming systems.

Mathematica provides a general mechanism in order to specifiy limitations for patterns. With "/;" (which reads "... whenever" or "...provided that") a condition is put at the end of a pattern to signify that it applies only when the specified condition is `True`.

Example : Pattern verification with restriction

In the subsequent definition of a function the condition never gives the value `True` so that one can see in detail how all possibilities will be checked.

```
Remove[f];
f[x__, y__, z__] := dummy /; Print[{x}, {y}, {z}]
```

```
f[1, 2, 3, 4, 5]
{1}{2}{3, 4, 5}
{1}{2, 3}{4, 5}
{1, 2}{3}{4, 5}
{1}{2, 3, 4}{5}
{1, 2}{3, 4}{5}
{1, 2, 3}{4}{5}
f[1, 2, 3, 4, 5]
```

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

Example : Reversion of a list

The following example programmed in typical PROLOG style reverses a list. The auxiliary function `rev[leftList,rightList]` acts on two stacks (lists) It removes the first element from `leftList` and inserts it into `rightList`. Wrapping with `Trace[expr,form,opts]` shows how this code works. It uses the method we would use to reverse the order of a card deck: taking one after another and placing it onto a second pile.

```
reverse[l_List] := rev[l, {}]
rev[{}, r_List] := r
rev[{s_, t___}, {r___}] := rev[{t}, {s, r}]
```

```
liste = {a, b, c, d, e};
Trace[ reverse[liste], rev[_List, _List], TraceAbove → True ] // ColumnForm
reverse[{a, b, c, d, e}]
rev[{a, b, c, d, e}, {}]
rev[{b, c, d, e}, {a}]
rev[{c, d, e}, {b, a}]
rev[{d, e}, {c, b, a}]
rev[{e}, {d, c, b, a}]
rev[{}, {e, d, c, b, a}]
{e, d, c, b, a}
```

Backtracking can be implemented with side conditions. The pattern matcher generates all possible cases. The side conditions can then be used to commit a certain case.

Example : Sorting of a list

An *inversion* in a list is a pair of adjacent elements such that the first one is larger than the second one. A sorted list is characterized by not having any inversions of adjacent elements. To sort a list, one simply reverses any inversions found. Thus, no particular order of doing this must be specified.

```
sort[{a___, i_, j_, z___}] := sort[{a, j, i, z}] /; i > j
sort[l_List] := l
```

The pattern matcher generates all possible pairs of adjacent elements **i** and **j**. Whenever they are out of order, they will be reversed.

```
Trace[sort[{5, 1, 3, 2}], sort, TraceAbove → True] // ColumnForm
sort[{5, 1, 3, 2}]
sort[{1, 5, 3, 2}]
sort[{1, 3, 5, 2}]
sort[{1, 3, 2, 5}]
sort[{1, 2, 3, 5}]
{1, 2, 3, 5}
```

Abstract Data Types

The following discussion follows closely chapter 2 in Roman Maeder's book "The *Mathematica* Programmer" .

Abstract data types are both a theoretically well-defined concept and a useful tool for program development. Following the principles of abstract data type design, one arrives at a clear separation of *specification* and *implementation.*

Abstract data types are defined through *type names*, *function names,* and *equations*. These can be realized in *Mathematica* very easily. The equations become *rewrite rules*. The interactive nature of *Mathematica* makes it well suited for rapid prototyping and testing of designs.

A *data type* can be defined roughly as set of *values* and *access methods*. One can then decide on *data representations* and *functions* with which data are manipulated and follow a usually self-imposed discipline in using only these functions in the application program.

One arrives at such guidelines through the study of the theoretical foundation of the design of abstract data types. Usually, such theoretical foundations are of theoretical interest only since they are either non-constructive to begin with or cannot be made into executable programs on a real computer. However, the specification of an abstract data type can be made executable under certain conditions. This is particularly easy in *Mathematica*, since it contains an interactive symbolic programming language with rewrite rules. This should be remembered during the following rather theoretical definitions of an abstract data type. They will be turned into running *Mathematica* programs soon.

Definition of Abstract Data Types

An ***algebra*** is a *mathematical structure* that consists of a *set of elements* and a number of *operations* on these elements. The *natural number* with the usual arithmetic operations are an example. Since one often uses more than one type of data this notation needs to be expanded into many-sorted algebras. Hence, the task of defining the operations then becomes a bit tricky, since one has to specify the types of all operands.

An abstract data type is specified in terms of three parts :

- A set $\mathcal{S}$ of *sort names*
- A set Σ of *function names* (or operator symbols)
- A set $\mathcal{E}$ of *equations*

The triplet $<\mathcal{S}, \Sigma, \mathcal{E}>$ is called a *specification*. The sorts can be thought of as the names of the *types* to be modelled. Usually there is more than one; auxiliary types, such as bool (standing for Boolean values) may be used. *Sort names* are printed as boldface lowercase characters.
In the first example natural number (non-negative integers) are modelled, with *Boolean* values as an auxiliary type. Therefore the set $\mathcal{S}$ of *sort names* consists of two elements

$$\mathcal{S} = \{\mathbf{int}, \mathbf{bool}\} \tag{1}$$

The set Σ of *function names* are broken down according to the number and types of arguments and the return type. This is described by a sequence of elements from $\mathcal{S}$, the last one being the return type, the preceeding ones describing the argument type in the proper order.

Formally, the set Σ consists of sets $\Sigma_{w,s}$ with $s \in \mathcal{S}$, $w \in \mathcal{S}^*$ (This says that w is a *sequence of elements* of $\mathcal{S}$, and s is a single element of $\mathcal{S}$, the *return type*.) With λ denoting the empty sequence the elements of $\Sigma_{\lambda,s}$ are called *constants* since they are functions without arguments.

In the present example of natural number the set Σ of *function names* comprises the follwoing definitions

$$\begin{aligned} \Sigma_{\text{int}} &= \{z\} \\ \Sigma_{\text{bool}} &= \{f, t\} \end{aligned}$$

$$\begin{aligned}\Sigma_{\text{int,int}} &= \{s\}\\ \Sigma_{\text{int,bool}} &= \{\mathbf{isz}\}\\ \Sigma_{\text{bool, bool}} &= \{\mathbf{not}\}\\ \Sigma_{\text{int, int, int}} &= \{\mathbf{add},\ \mathbf{mult}\}\end{aligned}$$

The notation points out that **z** is a constant of type **int** and that **add** is a function of two arguments of type **int**, returning a result of type int. Σ ist called the signature. Note, that up to now these function names has not given any meaning yet, (even though one can think of **add** as the addition function of integers).

There is nothing in the definitions so far that forces one to use the addition function as meaning for thee function named **add.** In order to implement the operations in Σ one can give a number of equations that must be satisfied by the functions chosen. The set $\mathcal{E}$ of *equations* below enforces the usual meaning for the operations. *Variables* are printed in italics with types obvious from the functions in which they appear.

$$\begin{aligned}\mathbf{isz}(\mathbf{z}) &= \mathbf{t}\\ \mathbf{isz}(\mathbf{s}(n)) &= \mathbf{f}\\ \mathbf{not}(\mathbf{t}) &= \mathbf{f}\\ \mathbf{not}(\mathbf{f}) &= \mathbf{t}\\ \mathbf{add}(n,\mathbf{z}) &= n\\ \mathbf{add}(n,\mathbf{s}(m)) &= \mathbf{s}(\mathbf{add}(n,m))\\ \mathbf{mult}(n,\mathbf{z}) &= \mathbf{z}\\ \mathbf{mult}(n,\mathbf{s}(m)) &= \mathbf{add}(\mathbf{mult}(n,m),n)\end{aligned} \tag{3}$$

Now, the Σ-algebras can be defined: a **Σ-algebra** is a mathematical structure that "fits" the signature Σ. It consists of *sets* for the sorts from S and of *mappings* corresponding to the operator symbols from Σ.

Now the present example can be endowed with its intendes mening, using the non-negative integers as carrier set of sort **int**. To describe the functions the *Mathematica* notation can be used, e.g.

$$\begin{aligned}\mathcal{B}_{\text{int}} &= \{0,1,2,...\}\\ \mathcal{B}_{\text{bool}} &= \{\text{False, True}\}\\ z_B &= \mathbf{0\&}\\ f_B &= \mathbf{False\&}\\ t_B &= \mathbf{True\&}\\ s_B &= \mathbf{Function[n,n+1]}\\ \mathbf{isz}_B &= \mathbf{Function[n,n==0]}\\ \mathbf{not}_B &= \mathbf{Not}\\ \mathbf{add}_B &= \mathbf{Plus}\\ \mathbf{mult}_B &= \mathbf{Times}\end{aligned} \tag{4}$$

It is customary to leave out the empty argument sequence for constants. Instead of writing
`(False&) []` one simply writes `False` etc.

If the equations are sufficiently well behaved this algebra can be modelled in *Mathematica* with the equations to be thought as reductions, always replacing the left side by the simpler right side. In this way there is one *special term* in each class of terms identified by the equations. Any other term is transformed into normal form by the applicable reduction rules. In the present example the equations can be simply turned into *rewrite rules* which will be sufficient for reducing all equivalent terms into normal form. The first example shows the corresponding code in *Mathematica.*

Example 1 : Data type *natural numbers*

The model for *natural numbers* consists of the *constant zero* and a *successor function*. This suffices to represent all natural numbers. Another kind of objects represents the Boolean values to express the predicates (i.e. functions returning a Boolean value).

The initial algebra for the specification of natural numbers is

Sort names $\mathcal{S}$

```
int;
bool;
```

Function names Σ

```
{z};              (* → int              constant zero *)
{t, f};           (* →  bool            Boolean true, false *)
{s};              (* int → int          Successor  *)
{isz};            (* int → bool         Predicate "is sequence of zero *)
{not};            (* bool → bool        Negation*)
{add, mult};       (* int int → int      Addition / Multiplication  *)
```

Equations $\mathcal{E}$ turned into rewrite rules

```
isz[z] := t         (* Predicate function applied to z gives zero *)
isz[s[n_]] := f
not[w] := f
not[f] := t
add[n_, z] := n
add[n_, s[m_]] := s[add[n, m]]
mult[n_, z] := z
mult[n_, s[m_]] := add[mult[n, m], n]
```

Test

Here a few sample computations. The integer n is represented as n-fold nesting of s applied to z

```
add[s[z], s[s[z]]]          (* Addition 1 + 2 = 3  *)
s[s[s[z]]]
```

A trace shows the computations going on.

```
Trace[ add[s[z], s[s[z]]], add, TraceForward → True ] // ColumnForm
add[s[z], s[s[z]]]
s[add[s[z], s[z]]]
{add[s[z], s[z]], s[add[s[z], z]], {add[s[z], z], s[z]}, s[s[z]]}
s[s[s[z]]]
```

The product of the integers 3 and 4 is

```
mult[s[s[z]], s[s[s[z]]]]      (* multiplication 2  * 3= 6 *)
```

```
s[s[s[s[s[s[z]]]]]]
```

```
Trace[ mult[s[s[z]], s[s[s[z]]]], mult,  TraceForward→True ] // ColumnForm
mult[s[s[z]], s[s[s[z]]]]
add[mult[s[s[z]], s[s[z]]], s[s[z]]]
{mult[s[s[z]], s[s[z]]], add[mult[s[s[z]], s[z]], s[s[z]]], {mult[s[s[z]], s[z]
s[s[s[s[s[s[z]]]]]]
```

```
not[isz[s[z]]]       (* Test of 1 ≠ 0  *)
t
```

One does not want to perform arithmetic with this abstract version of data types. Instead one can work with a more suitable Σ algebra. The integers `{0,1,2,...}` are in one-to-one correspondence with the terms `{0,s[0],s[s[0]], ...}`

Example 2 : Data type *rational numbers*

The data type for rational numbers can be defined with the *constructor* `MakeRational[z,n]` and the selectors `Zaehler[r]` and `Nenner[r]`. The representation of a rational number is given through a data element of the form `rational[z,n]` which represent the rational number $\frac{z}{n}$.

Rules

```
makeRational[z_, n_]       := rational[z, n]
Zaehler[rational[z_, n_]]  := z
Nenner[rational[z_, n_]]   := n
```

Data type rational

The data type of rational numbers is *rational*. Usually the datatype is represented al a normal expression with the sort name als head.

```
Head[makeRational[z, n]]
rational
```

Normal form

The current representation of rational number is however *not* unambiguous, because `rational[-1,2]`, `rational[1,-2]` and `rational[-2,4]` describe the *same* rational number. The representation becomes unambiguous if one requires that numerators are always positive and in addition, that numerator and denominator have not a common divisor. The last two rules (with built-in function GCD) for `rational[z,n]` guarantee that data elements `rational[z,n]` will be transformed to normal form.

```
rational[z_, n_] := rational[-z, -n] /; n < 0
rational[z_, n_] := rational[z/GCD[z, n], n/GCD[z, n]] /; GCD[z, n] > 1
```

Connections

Rules for summation and multiplication of 2 rational numbers which operate exclusively with *constructors* and *selectors* (being the only functions which have access on the internal representation of data elements) are

```
rational /: a_rational + b_rational :=
  makeRational[
  Zaehler[a] Nenner[b] + Zaehler[b] Nenner[a], Nenner[a] Nenner[b]]
rational /: a_rational b_rational :=
  makeRational[Zaehler[a] Zaehler[b], Nenner[a] Nenner[b]]
```

The existing connections "+" and "×" are overloaded. The easiest way to do this is using *upvalues* of the form type /: `f[n_type, ... ]  := ...` . Upvalues provide a convenient mechanism to define how operations act on object of special type.

Tests

```
makeRational[4, -6]
rational[-2, 3]
```

```
makeRational[2, 3] + makeRational[5, 4]
rational[23, 12]
```

```
makeRational[2, 3] makeRational[3, 2]
rational[1, 1]
```

```
makeRational[6, 5] (makeRational[2, 3] + makeRational[3, -4])
rational[-1, 10]
```

Example 3 : Data type *modular numbers*

In a similar way the arithmetics for *modular numbers* can be formulated. This example is intended to model $\mathcal{Z}_p$, the ring of integers modulo some positive integer p. It can be built on top of the natural numbers, extending S by a sort of **mod**, and adding the following operations.

$$
\begin{aligned}
\Sigma_{\text{int}} &= \{p\} \\
\Sigma_{\text{mod,int}} &= \{\mathbf{rep}\} \\
\Sigma_{\text{int,modl}} &= \{\mathbf{makemod}\} \\
\Sigma_{\text{mod, mod, mod}} &= \{\mathbf{add},\ \mathbf{mult}\} \\
\Sigma_{\text{int, int, int}} &= \{\mathbf{rem}\}
\end{aligned}
\qquad (5)
$$

There is no problem with using the same operation name (here **add** and **mult**) in different sets $\Sigma_{w,s}$. They are distinguished by the type of arguments and return values. **rep** is meant to give a representative of a modular number, an integer that can be used to implement arithmetic in **mod**, as will be seen in the following equations.

$$
\begin{aligned}
\mathbf{makemod}(\mathbf{rep}(m)) &= m \\
\mathbf{rem}(n\text{-}\mathbf{rep}(\mathbf{makemod}(\mathrm{n})),\mathbf{p}) &= \mathbf{0}
\end{aligned}
\qquad (6)
$$

$$\mathbf{add}(m_1,m_2) = \mathbf{makemod}(\ \mathbf{add}(\ \mathbf{rep}(m_1)\,,\ \mathbf{rep}(m_2)\ ))$$
$$\mathbf{mult}(m_1,\ m_2) = \mathbf{makemod}(\ \mathbf{mult}(\ \mathbf{rep}(m_1)\,,\ \mathbf{rep}(m_2)\ ))$$

The second equation says that the representative of a modular number made from integer n is concruent to n mod p . ("-" being integer subtraction). This is usually expressed as

$$\mathbf{rep}(\mathbf{makemod}(\mathrm{n})) \equiv \mathrm{n}\ (\mathrm{mod}\ \mathbf{p}) \tag{7}$$

Addition and multiplication are then defined in terms of their integer counterparts through the use of **rep**. A possible implementation, using expressions of the form `mod[n]` with *n* integer, holds elements of $\mathcal{B}_{\mathrm{mod}}$

Constructors

`MakeModular` is called a *constructor*, because it creates values of a certain type (here mod).

```
SetModulus[p_Integer?Positive] := (theModulus = p)
MakeModular[n_Integer] := mod[ Mod[n, theModulus]]
```

Selectors

`Representative` is a *selector*, since it returns parts of an expression (**mod** elements have only one part, the representative of the modular number).

```
Representative[ mod[n_] ] := n
```

```
mod /: a_mod + b_mod := MakeModular[Representative[a] + Representative[b] ]
mod /: a_mod * b_mod := MakeModular[Representative[a] * Representative[b] ]
mod /:
  a_mod ^ q_Integer := MakeModular[ PowerMod[Representative[a], q, theModulus] ]
mod /: a_mod + b_Integer := a + MakeModular[b]
mod /: a_mod * b_Integer := a * MakeModular[b]
```

Output Formatting

```
Format[m_mod] := SequenceForm[ Representative[m], " mod ", theModulus]
```

The use of the built-in function `Mod[n,p]` in **MakeModular** ensures that a unique representation is chosen (an integer in the range 0 ...p-1). Putting this computation into the constructor removes all worries about the choice of representative from the rest of the code and trivially satisfies the first and second equation.

Note, that the built-in addition and subtraction functions have been *overloaded* which corresponds directly to the use of the same names in the specification.

The *modulus* is set to *17* for the computation and the *constant* 1 mod 17 is generated.

```
SetModulus[17];
one = MakeModular[1]        (* constant 1 mod 17 *)
1 mod 17
```

Any integer can be turned into a modular number by multiplying it with ***one***

```
m1 = one 100
15 mod 17
```

Large powers can be efficiently computed using the built-in function `PowerMod`

```
m2 = m1 ^ 9999999999999999999999
8 mod 17
```

The rule works for *subtraction*

```
m3 = 1 - m2
10 mod 17
```

Division is turned into *multiplication* and a negative power. For a modular number *a*, the quantity a^{-1} denotes the *modular inverse.*

```
m4 = 1 / m3
12 mod 17
```

The *product* of a number and its inverse should be 1.

```
m4 m3
1 mod 17
```

The following loop computes the *order* of the modular number 2 which is the smallest exponent n such that $x^n = 1$ *(mod p)*.

```
x = MakeModular[2];
n = 1 ; While[x^n =!= one, n++]; n
8
```

In quite similar way it is possible to model the data structure of LISP by means of *Mathematica* (See for example "The Mathematica Programmer" (1994) by Roman Maeder)

Object Oriented Programming

Object-oriented programming (OOP) is a programming style that is becoming more and more popular. It promises the *re-use of program code* and easier *maintenance of larger projects* than is possible with traditional procedural programming languages. Its use of *methods* and *message passing* instead of procedure calls shifts the programmer's view towards closer integration of data and operations.

OOP is an alternative method as regards to modular programming : one does not comprise functions doing similar things but combines functions operating on similar objects. At first the *objects* will be defined and then the corresponding *methods* for manipulating the objects. In order to clarify this distinction consider for example the *print function*: in a modular program it contains instructions for printing all possible objects whereas in an object-oriented style for every object a print method will be defined.

The important concepts of objected-oriented programming languages are *objects and classes* (associated with the objects) and *inheritance.*

The *first* important aspect of object-oriented languages is that functions are considered part of *data.* A data object "knows" which operations can be performed on it. The functions defined for a certain type of object are part of that object. Thus an *object* is a collection of *data elements* and *operations* that act on these date elements. The operations are called *methods.* A uniform mechanism, called *message passing,* is provided for invoking the correct piece of code when a function is called or a message is passed to an object in order to execute a certain method. Methods are usually not defined for each object separately but are collected in a *class*. Objects then belong to a class from which they take their methods.

The *second* important aspect is *inheritance*. Often a number of related data types have some common characteristics. Some operations on them can be written in a way that does not depend on which of the data types they are applied to. Common characteristics of related data types can then be isolated and encapsulated in a new data type. The related data types are made sub-types of the new type. They *inherit* the characteristics of the common type and need only implement those aspects in which they differ from their super-type. Thus much of the code needs to be written only once and hence save development time and ensure consistency, since a change needs to be made only once, instead of being applied to several almost identical pieces of code.

An interactive OOP language can easily be implemented into *Mathematica.* Although *Mathematica* does not presuppose data types the arguments of functions can nevertheless be restricted to certain types (through *patterns* of the form `x_head` or `x_?test` or *conditions* of the form `expr /; conditions`). Using appropriate heads it is thus possible to simulate certain data types.

The mechanism to define user-specific objects is `Head[expr]`. Moreover it is possible to associate definitions (or transformation rules) with specific objects. Instead of associating definitions of the form `f[g[x_]]:=...` with the symbol `f` (the *downvalue* of `f`) one may also associate the definition with the symbol `g` (the *upvalue* of `g`) which is not the head of the expression on the left hand side. The syntax is `f[g[x_]^:= ...` or equivalently `g/:f[g[x_]:=...` This construct allows *operator overloading* which means that an operator (e.g. in *example 2* `Plus` and in *example 3* `Equal`) has a different meaning according to the object it acts on. (This feature is not possible in programming languages such as Pascal or Modula-2.)

Obviously, the term *object-oriented* means that the transformation rule associated with an object is not categorized under the rule's name but rather under the object's name. Therefore it makes sense to order the transformation rules in an object-oriented way. This concept allows to find the corresponding rules rather quickly and load resp. unload a whole contingent of rules if necessary. This is important under the view point of economic memory management. *Mathematica* supports a distinct object-oriented arrangement of rules in that it signes the rules with the corresponding names of the objects.

```
objectName /: pattern := instruction /; condition
```

Obviously, same functions can be defined for different objects in *multiple context*. Within a package it is possible to define a private context which contains auxiliary functions and variables not accessible from outside the package. In *TMJ Vol 3 #1 p 23-31* Roman Maeder introduced two packages `Classes.m` and `Collections.m` which implement the concept of *classes* and *inheritance*.

***Example 1* : Exponential function**

```
exp /: exp[x_] exp[y_] := exp[x + y]
exp /: Sqrt[exp[x_]] := exp[x/2]
```

```
exp[a] exp[b]
exp[a + b]
```

```
Sqrt[exp[3 y^2]]
exp[3 y^2 / 2]
```

***Example 2* : Modular Arithmetic**

As an example the *data type for modular numbers* is defined. At first the output formatting is fixed through the following definition

```
modInteger /: Format[modInteger[i_, n_]] := SequenceForm[i, " mod ", n]
```

Similar to this definition of the output format of the function `modInteger` other format types (e.g. TeXForm) can be modified too such that the *type* is listed as a second argument of `Format[expr,type]:=form`, e.g.

```
modInteger /: Format[modInteger[i_, n_], TeXForm] :=
                  SequenceForm[ "{", i, " \bmod ", n, "}"]
```

Then one defines the *sum* and the *product* of objects of type modInteger (by means of overloading the opertators `Plus` and `x`.)

```
modInteger /:
 modInteger[i_, n_] + modInteger[j_, n_] := modInteger[Mod[i + j, n], n]
modInteger /:
 modInteger[i_, n_] modInteger[j_, n_]   := modInteger[Mod[i j, n], n]
modInteger /: modInteger[i_?Negative, n_]          := modInteger[Mod[i, n], n]
modInteger /: -modInteger[i_, n_]                  := modInteger[-i, n]
```

With these definitions the modular arithmetic is complete.

Test

```
modInteger[4, 7] // TeXForm
{4 \bmod 7}
```

```
modInteger[2, 5] + modInteger[3, 5] modInteger[2, 5]
3 mod 5
```

```
modInteger[-2, 5]
3 mod 5
```

```
-modInteger[4, 5] modInteger[2, 5] - modInteger[3, 5]
4 mod 5
```

Example 3 : **Comparison and Sorting of Lists of Names**

In order to compare and sort name lists a new data type *name* is introduced with two arguments for first and last name. For this data type the output format is defined first.

```
name /: Format[name[f_String, l_String]] := SequenceForm[f, " ", l]
```

```
name["Fritz", "Meier"]
Fritz Meier
```

Then a *comparison function* will be defined. This function tests whether first or last name of two persons are equal. Associating this definition with data type *name* leads to overloading of the function **Equal** (resp. ==) already existing.

```
Equal[name[f1_String, l1_String], name[f2_String, l2_String]] ^:=
      f1 == f2 && l1 == l2
```

```
name["Fritz", "Meier"] == name["Fritz", "Meier"]
True
```

```
name["Fritz", "Huber"] == name["Fritz", "Meier"]
False
```

Unfortunately, association is only possible for symbols on the highest level of the argument list. Thus, in order to introduce a *sort function* which accepts a list of names this sort function must be named newly. For its definition the built-in function **Sort** may be used with second parameter being a user-defined comparsion function which takes into account the last names to be sorted first.

```
nameSort[x : {name[_String, _String]..}] :=
 Sort[x, OrderedQ[{#1[[2]] <> #1[[1]], #2[[2]] <> #2[[1]]}] &]
```

Test

For a given list of names

```
list = { name["Hans", "Meier"],
       name["Hans", "Huber"],
       name["Fritz", "Meier"],
       name["Fritz", "Huber"] };
```

```
nameSort[list]
{Fritz Huber, Hans Huber, Fritz Meier, Hans Meier}
```

Modularization

Modern programming languages provide features for organizing large programming projects. Most important are *modularization* (or encapsulation) and *information hiding*. In *Mathematica* such programs are called *packages*. A package consists of two parts :an *interface definition* and an *implementation part*. The interface describes the aspects that a user needs to know. The implementation provides that functionality but it is hidden from users of the package.

Here is the skeleton of how a *Mathematica* package looks like :

```
BeginPackage[" PackageName`"]
 proc::usage = "proc[v1, v2] is a procedure that ... "
   ...
   Begin["`Private`"]
     f[intArgs_] := value
         ...
     proc[args_, intArgs_] := ...
   End[ ]
 Protect[ proc ]
EndPackage[ ]
```

The part between initial `BeginPackage[]` and `Begin["`Private`"]` is the interface. It declares all functions exported from this package (here it is `proc[]`) and documents them.

By means of `BeginPackage[ ]` the *context name* `"PackageName`"` is given to the package which by convention must be stored in a file with the same name `PackageName.m`. If `PackageName`` comprises a sub-context (e.g. `Graphics`Colors``) then the file `Colors.m` must be located in the subdirectory `Algebra`.

Expressions of the form `proc::usage = "comment"` following after `BeginPackage` can be understood as an *export list* of the package. Here, all objects which are provided to the user will be endowed with the documentation. This documentation is given in `"comment"` . With the command `?proc` the help text is displayed as output.

After these lines of documentation there follows a *private context*. The part bracketed by `Begin` and `End` is the proper implementation. With `Begin["`Private`"]` a private sub-context is opened and the system variable `$Context` set to `` `Private` ``.The context comprises the definitions of the *exported functions* (here `proc[ ]`) and *local quantities* (auxiliary functions such as `f[ ]`and *local variables* such as `intArgs`) which are not available for the user of the package. The private context is closed again with `End[ ]`.

The outer bracket `BeginPackage[ ] ... EndPackage[ ]` is slightly more complicated. With `BeginPackage[ ]` the *system variables* `$Context` and `$ContextPath` are stored. Then `$Context` is put to `PackageName`` and `$ContextPath` to `{"PackageName`","System`"}`. *New names* (i.e. names of *hidden* quantities) are thus introduced in the context `` `PackageName` `` (and not in the context `` `Global` ``). All names are considered as new ones even if they already exist in `"Global`"`context. All function names which appear in lines starting with `proc::usage` are put in the context `"PackageName`"`

`EndPackage` finally restores `$Context` and `$ContextPath` with context `"PackageName`"` prepended. Only those functions exported from the package are available to the user because their context is contained in the list `$ContextPath` . Names however which appear in the internal `Begin-End` bracket can only be quoted through explicit referencing of their context.

In order to *protect* the function definitions within a package the command `Protect` must be placed between `End` and `EndPackage`. With `Protect[proc]` one prevents any modification of the exported function `proc` through users of the package. If, however, a protected symbol `s` within a package should be modified the symbol `s` must be freed with `Unprotect[s]` first. Then, a new definition of this symbol may follow which is protected again with `Protect[s]` .

Packages are extension of the built-in functions provided by *Mathematica*. Packages are either loaded with the

command `Get[PackageName.m]` (or `<<PackageName.m`) or more recommendable with `Needs["PackageName`"]` thus avoiding multiple loading of packages. Packages can load additional packages if necessary. The corresponding context is then stored in the global variable $Packages.

▪ Framework for a User-defined Package

The subsequent skeleton of a package illustrates how a consistent form faciliates maintenance and readability of *Mathematica* programs. All important concepts are contained in the prototype package

```
(*  --   Skeleton.m  --          Skeleton of a package *)

(*Set up packagecontext, includinganyimports*)
BeginPackage["Skeleton`", "Package1`", "Package2`"];
Needs["Package3`"];                         (* Read in any hidden imports *)

(* Usage message for the exported functions and context itself *)
Skeleton::usage = "Skeleton.m is a package doing nothing."
Fnct1::usage = "Fnct1[n] does nothing"
Fnct2::usage = "Fnct2[n,(m:17)] does even more nothing!"

Begin["`Private`"]                                (* Begin private context *)
(*Unprotect any built-in functions for which new rules will be defined *)
protected = Unprotect[Sin, Cos]

(* Definition of auxiliary functions and local variables *)
AuxFct[f_] := do[something]
localVar = 0

(* Error messages of exported objects *)
Skeleton::badarg = "Error, called `1` with argument `2` !"

(* Definition of exported functions *)
Fnct1[n_] := n
Fnct2[n_, m_: 17] := n m /; n < 5 || Message[Skeleton::badarg, Fct2, n]

(* Rules for system functions *)
Sin /: Sin[x_]^2 := 1 - Cos[x]^2

Protect[Evaluate[protected]]
   (* Restore protection of system symbols *)

End[];                                           (* End of private context *)

Protect[Fnct1, Fnct2]                        (* Protect exported symbol *)
EndPackage[];                                   (* End of package context *)
```

Mathematica as Developing Tool : from an Interactive Evaluation to a Package

The following task serves as an example to illustrate the step-wise transition from an interactive evaluation to the development of a package.

For a given sequence of n random numbers ordered by their size the *median* is to be calculated. If n is *odd* then the value is a the location $\frac{n+1}{2}$, if n is even the median is the average of the values at location $\frac{n}{2}$ and $\frac{n+1}{2}$. The result will at first be calculated step by step by an interactive mode, afterwards realized in form a *Mathematica* package.

■ Evaluation in Dialog Mode

```
tlist = Table[Random[], {i, 1, 10}]
{0.65224, 0.0164685, 0.197755, 0.351424, 0.543858, 0.684035, 0.585712,
 0.196561, 0.0132099, 0.911238}
```

```
len = Length[tlist]
10
```

```
slist = Sort[tlist]
{0.0132099, 0.0164685, 0.196561, 0.197755, 0.351424, 0.543858,
 0.585712, 0.65224, 0.684035, 0.911238}
```

```
? Odd*
OddQ[expr] gives True if expr is an odd integer, and False otherwise.
```

```
OddQ[len]
False
```

Calculation of the *median* value

```
median0 = If[OddQ[len], slist[[(len + 1)/2]], 1/2 (slist[[len/2]] + slist[[len/2 + 1]])]
0.447641
```

■ Combine Commands in terms of a Function

The single commands are comprised in terms of a module function.

```
median1[l_List] :=
 Module[{len, slist},
       len = Length[l]; slist = Sort[l];
       If[OddQ[len], slist[[(len + 1)/2]], 1/2 (slist[[len/2]] + slist[[len/2 + 1]])]
      ]
```

```
median1[tlist]
0.627426
```

Integrate Functions in a Package

Alternatively, one may define a package

```
BeginPackage["MedianContext`"]
median2::"usage" =
 "median2[list] calculates the median of list and returns the result."

Begin["`Private`"]
median2[l_List] :=
   Module[{len, slist},
          len = Length[l]; slist = Sort[l];
          If[OddQ[len], slist[[(len + 1)/2]], 1/2 (slist[[len/2]] + slist[[len/2 + 1]])]
         ]
End[]
 Protect[median2]
EndPackage[]
MedianContext`
```

```
median2[list] calculates the median of list and returns the result.
```

```
MedianContext`Private`
```

```
MedianContext`Private`
```

```
{median2}
```

```
?? median2
median2[list] calculates the median of list and returns the result.
```

```
Attributes[median2] = {Protected}

median2[MedianContext`Private`l_List] :=
  Module[{MedianContext`Private`len, MedianContext`Private`slist},
   MedianContext`Private`len = Length[MedianContext`Private`l];
   MedianContext`Private`slist = Sort[MedianContext`Private`l];
   If[OddQ[MedianContext`Private`len],
    MedianContext`Private`slist[[(MedianContext`Private`len + 1) / 2]],
    1 / 2 * (MedianContext`Private`slist[[MedianContext`Private`len / 2]] +
       MedianContext`Private`slist[[MedianContext`Private`len / 2 + 1]])]]
```

The symbol **median2** is protected in **MedianContext**. Hence, it is not possible to assign a value to the protected symbol as shows the following assignment. Loading a package which contains the same procedure gives rise to a conflict because there occurs a shadowing of the same symbol from different contexts.

```
median2 = 4
Set::wrsym : Symbol median2 is Protected.
4
```

```
<< "Median.m"
SetDelayed::write : Tag median2 in median2[l_List] is Protected.
```

▪ Errors and Tracing

In every non-trivial program there occur errors. Therefore, it is essential to search for them and provide some tools for this task. In order to find programming errors *Mathematica* provides two main tools : **Trace** and **Dialog**.

The function **Trace** lists all sub-expressions generated during evaluation. Hence, it is possible to watch all *intermediate steps* .

```
a = 3;
(a^2 + a + 1) // Trace
{{{a, 3}, 3^2, 9}, {a, 3}, 9 + 3 + 1, 13}
```

As regards to evaluation of a *Mathematica* expression one has to be aware of the fact that the tree structure of an expression is evaluated bottom up. In the case of nested function calls the calculation proceeds from inside to outside which thus enables functional programming. The head of an expression is processed first.

```
s = Sin;  s[{π/4, π/2}] // Trace
{{s, Sin}, {{{1/4, 1/4}, π/4, π/4}, {{1/2, 1/2}, π/2, π/2}, {π/4, π/2}},
 Sin[{π/4, π/2}], {Sin[π/4], Sin[π/2]}, {Sin[π/4], 1/√2}, {Sin[π/2], 1}, {1/√2, 1}}
```

Consider the following calculation which if simply **Trace** is applied turns out to be quite voluminous. However, called with a second parameter it is possible to trace only the values of j.

```
Trace[ For[i = 1; j = 1; p = 5, p < 5000, i++,
        p = Prime[i + j];
        j = Floor[√((p + 1 / 2)^2) ]] , j]
{{{{{{j, 1}}}}}, {{{{{j, 3}}}}}, {{{{{j, 11}}}}}, {{{{{j, 43}}}}},
 {{{{{j, 211}}}}}, {{{{{j, 1321}}}}}}
```

The output can also be restricted to suitable patterns given as second parameter to **Trace**. All definitions in this example are obtain with _=_

```
Trace[ For[i = 1; j = 1; p = 5, p < 5000, i++,
        p = Prime[i + j];
        j = Floor[√((p + 1 / 2)^2) ]] , _ = _] // ColumnForm
{{i = 1}, {j = 1}, {p = 5}}
{{p = 3}, {j = 3}}
{{i = 2}}
{{p = 11}, {j = 11}}
{{i = 3}}
{{p = 43}, {j = 43}}
{{i = 4}}
{{p = 211}, {j = 211}}
{{i = 5}}
{{p = 1321}, {j = 1321}}
{{i = 6}}
{{p = 10909}, {j = 10909}}
{{i = 7}}
```

With the following options of `Trace` such as

```
TraceBackwards→False
TraceForward→False
TraceDepth→Infinity
TraceOff→None
TraceOn→pattern
TraceOriginal→False
```

it is possible to control the output in detail.

```
Trace[ For[i = 1; j = 1; p = 5, p < 5000, i++,
        p = Prime[i + j];
        j = Floor[Sqrt[(p + 1 / 2)^2 ]] ] , TraceOn → Floor ] // ColumnForm
{{{Floor[7/2], 3}}}
{{{Floor[23/2], 11}}}
{{{Floor[87/2], 43}}}
{{{Floor[423/2], 211}}}
{{{Floor[2643/2], 1321}}}
{{{Floor[21819/2], 10909}}}
```

▪ Dialog

Mathematica allows *debugging* that is direct interference in a running program in order to trace errors. In this way it is possible to inspect actual values and change them if necessary.
In an interactive session the command `Dialog[]` or `Dialog[expr]` starts another subsession or dialog which is often useful if one wants to interact with *Mathematica* while it is in the middle of doing a calculation. With `Return[]` one exits from the dialog. It is also possible to return a value from this dialog using `Return[expr]`.

`TraceDialog[expr]` automatically starts a dialog for every *Mathematica* expression which is used for the evaluation of ***expr***. In the subsequent example the tracing is restricted to assignments of the form `y = a_. + _^4`

```
y = t;
TraceDialog[ Do[y = Expand[(y + i) ^2], {i, 5}], y = a_. + _^4 ]
TraceDialog::dgbgn : Entering Dialog; use Return[] to exit.
y = 9 + 12 t + 10 t^2 + 4 t^3 + t^4
```

```
TraceDialog::dgend : Exiting Dialog.
```

While the dialog session is running it is possible to look e.g. for the value of `i`

```
i
2
```

or change in a running calculation the value of the variable t which is set to 1

```
t = 1;
1
```

Entering `Return[]` exits the dialog session.

```
Return[ ]
```

In this case a numerical value is obtained instead of a polynomial in `y`.

```
y
5408554896900
```

Finally, using the command `Dialog` a dialog session is started when `f[3]` is calculed

```
f[0] = 1;
f[n_] := (If[n == 3, Dialog[ ] ]; n f[n - 1] )
```

```
f[8]
40320
```

The command `Stack[pat]` gives the list of all expressions under evaluation which fit the pattern `pat`.

```
Stack[_]
{If[8 == 3, Dialog[]]; 8 f[8 - 1], 8 f[8 - 1], If[7 == 3, Dialog[]];
 7 f[7 - 1], 7 f[7 - 1], If[6 == 3, Dialog[]]; 6 f[6 - 1], 6 f[6 - 1],
 If[5 == 3, Dialog[]]; 5 f[5 - 1], 5 f[5 - 1], If[4 == 3, Dialog[]];
 4 f[4 - 1], 4 f[4 - 1], If[3 == 3, Dialog[]]; 3 f[3 - 1], Dialog[]}
```

or all products using the pattern Times

```
Stack[Times]
{8 f[8 - 1], 7 f[7 - 1], 6 f[6 - 1], 5 f[5 - 1], 4 f[4 - 1]}
```

The dialog is closed with `Return[]`.

```
Return[ ]
```

Programmer's Recommendation and Conclusion

Finally, there is a Golden Rule of *Mathematica* programming : avoid *iteration*. An old saying amongst computer scientists is: *To iterate is human, to recurse, divine.*

As was shown there is a whole assortment of *list manipulation* functions built into *Mathematica*, which are in general much faster than iterations. Moreover, these list manipulation operations can be threaded together to construct fast, elegant and efficient solutions to programming problems which to demonstrate was the intention of the previous subsections.

As a final remark it seem quite convincing that due to the fact that *Mathematica* is an interpreter system its programming language is most suitable for rapid prototyping, for the development and testing of algorithms etc. . The compactness of the *Mathematica* code which comprised more than 1000 commands and high level functions not available in other programming languages allow a reduction of length of traditional programs by a factor of 10 - 20. Due to increasing computational power and performance of computers speed is (in most cases) no longer an essential argument in favour of compiler languages. The high flexibility of the *Mathematica* language which supports very different programming styles as was demonstrated above, and the advanced notation which admits the usage of symbols identical to the ones occuring in the mathematical formulation of the problem enable the experienced *Mathematica* user to write sophisticated programs. It is challenging to write the programs such that the programming code seems to be more or less a straightforward transcription of the mathematical language in which the problem was formulated. It is the experience of a growing user community that *Mathematica* is the most suitable tool for this sort of tasks.

Programming Literature

Concerning the literature for programming in *Mathematica* there exist several *books* to be recommended on which this tutorial is based on.

Programming literature

Robert M. Dickau / WRI
Mathematica Training:
Programming with *Mathematica* (1996)

Richard J. Gaylord, Samuel N. Kamin, Paul R. Wellin
Introduction to Programming with *Mathematica* , 2nd Edition
TELOS/Springer-Verlag/Berlin (1996)
ISBN 0-387-94434-6 (with disk)

John W. Gray
Mastering *Mathematica* : Programming Methods and Applications, 2nd Edition
Academic Press Professional (1997)
ISBN 0-12-296105-6 (with CD-ROM)

Stephan Kaufmann
Mathematica as a tool.
An Introduction with practical examples
Birkhäuser Verlag (1994)
ISBN 0-8176-5031-8 (with disk)

Roman E. Maeder
Programming in *Mathematica* , 3rd Ed.
Addison-Wesley-Longman Inc. (1997)
ISBN 0-201-85449-X

Roman Maeder
The *Mathematica* Programmer
Academic Press Professional (1994)
ISBN 0-12-464990-4 (with disk)

Roman Maeder
The *Mathematica* Programmer II
Academic Press, Inc. (1996)
ISBN 0-12-464992-0 (with CD-ROM)

Troels Petersen, Ed.
The Elements of *Mathematica* Programming
TELOS/Springer-Verlag (to be published)
ISBN 0-387-94590-3 (with CD-ROM)

David B. Wagner
Power Programming with *Mathematica* : The Kernel

McGraw-Hill (1996)
ISBN 0-07-912237-X (with disk)

Stephen Wolfram
The *Mathematica* Book, 3rd Edition
Wolfram Media/Cambridge University Press (1996)
ISBN 0-9650532-0-2 (Hardcover)
ISBN 0-9650532-1-0 (Paperback)

University Center
Joint Institute for Nuclear Research / Dubna

Mathematica Seminars
October 27-29, 1998

Mathematica Front End

Prof. Dr. Robert Kragler
FH Ravensburg-Weingarten /University of Applied Sciences
kragler@fh-weingarten.de

Abstract :

The purpose of this tutorial is to give an introduction to some of the features of the *Mathematica* Front End, i.e. the notebook interface with help of which the user interacts with the *Mathematica* kernel in order to create interactive documents. The Front End is a purely *graphical interface* which is supported on most computer systems.

Alternatively one may also communicate with a text-based interface by typing text on the keyboard.

The lecture will deal with different technicalities such as

- the layout of *Mathematica* notebooks
- how to change text styles
- how to customize stylesheets for layout
- how to change the style environment of notebooks
- an overview of the Option Inspector

Moreover, due to the fact that notebooks are by themselves nothing but *Mathematica* expressions they can therefore be generated with of *Mathematica* programs and may be modified by the *Mathematica* kernel itself. Furthermore, examples will be given on

- discussion of the some of the new typesetting features
- how to generate buttoms and palettes,
- how to rotate icons for open/close of cell groups
- how to use hyperlinks etc.

Initializations

Interactive Documents and Front End Features

- *Mathematica* is more than merely a Computer Algebra System.
  Since version 3.0 it is a System for Technical Computing used for the creation of interactive documents

Besides *symbolic* and *numeric* calculations and *2D/3D graphics Mathematica*'s salient features are :

- graphic user interface, the so-called Front End
- worksheets, i.e. *Mathematica* Notebooks

***Mathematica* Notebooks** are *interactive* electronic documents which may contain :

Mathematica **code** (to be executed), **text** (with most DTP features supported such as different fonts, colour, mathematical formulae etc.), **graphics** and **animation,** (even in real-time using the add-on programs *MathLive* or *Dynamic Visualizer*) and **sound** (which is encoded in PostScript code as is graphics and other elements in terms of cells).

Mathematica Notebooks : *Interactive* Documents

The statement : "every *Mathematica* notebook is a *Mathematica* expression" is more than only a credo; it means these worksheets can be generated or manipulated by the kernel.

In more detail : ***Mathematica* notebooks** are ordinary *Mathematica* expressions. Therefore, they can immediately be *manipulated* by the user without any special setup etc. Hence, the results given in *Mathematica* notebooks can easily be checked or altered. *Mathematica* notebooks are organized in terms of *hierarchical cells* the layout of which can be pre-defined in so-called *StyleSheets* which are again special *Mathematica* notebooks in a specific stylesheet subdirectory. Notebooks can even be created by the *Mathematica* kernel as output. There are no limitations as regards to the appearance of *Mathematica* notebooks. Due to the hierarchical cell concept notebooks can be read like books comprising table of content, sections and subsections etc. which are accessible after opening the corresponding pages.

However, the *advantage* of the electronic document in comparison to conventional books is its *interactivity*, i.e. probing and manipulating the *Mathematica* code incorporated in the notebook. (The very first pioneering book which contained all chapters in terms of *Mathematica* notebooks on a CD-ROM disk was "Exploring Mathematics with *Mathematica*" by Th. Gray and J. Glynn published by Addison-Wesley Publ. Company already in 1991.

But the *real important* new feature in *Mathematica* version 3.0 is the *portability* of *Mathematica* notebooks. *Mathematica* notebooks contain all information in terms of ASCII code. They are *platform independent*. In earlier versions it was always a big problem to convert *Mathematica* notebooks which had been generated, say, under Windows operating system to Unix workstations (OS Solaris), NeXT (OS NeXTSTEP) or Macintosh (Mac OS 7) computers. Especially international characters such as German umlauts (or kyrillic letters) and other special symbols were a tedious problem to cope with in the past because there is even today no general accepted standard for character encoding. In *Mathematica* V 3.0 this problem is definitely solved. There are *code translation tables* for all *Mathematica* implementations which are provided for a large number of different hardware platforms.

Thus *Mathematica* notebooks may be used as a mean for *exchange of scientific information* and could become a kind of standard for interactive electronic documents in science. On a wider scale, *Mathematica* notebooks can be provided to academia via Web. There is already the built-in capability to save *Mathematica* notebooks in *HTML format* so that they can be used as WWW pages.

Recent developements from Wolfram Research are *Publicon* and *MathML* :

Publicon - a comprehensive *solution for interactive technical publishing* which allows the creation of professional-quality technical documents for on-screen, web, and printed use. A free beta version of Publicon can be downloaded from URL <http://www.publicom.com/> .

MathML (*Math*ematical *M*arkup *L*anguage) - a new standard recently ratified by W3C (i.e. the WWW Consortium) becomes the framework for Web typesetting technology. MathML is designed to allow mathematical expressions to be transmitted over the Web, preserving the structure needed to do computations with them in *Mathematica*. Information on MathML can be downloaded from the Web at <http://www.wolfram.com/news/mathml/ > .

Mathematica Front End features

Details of the *Mathematica* Front End are given in another notebook which can be invoked with the button `FrontEndE .nb`.

This feature to work with buttons is a convenient mean of referring to other *Mathematica* notebooks in different subdirectories and thus organizing a talk or lecture avoiding the danger not to find the proper subdirectory with the notebook wanted.

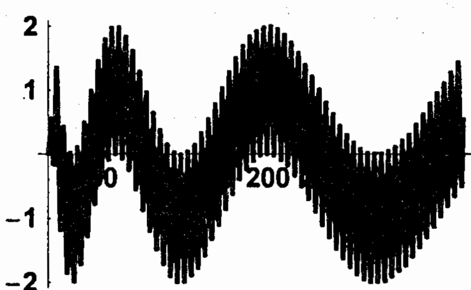
Mathematica Notebooks

Every *Mathematica* notebook is a complete interactive document which combines text, tables, graphics, calculations and other elements.

This document *is* a notebook.

A calculation

```
Plot[Sin[ξ] + Sin[√ξ], {ξ, 0, 120 π}];
```



- Notebooks are automatically organized in a hierarchy of cells.
- One can close groups of cells so that only their headings are to be seen.
- One can use Hyperlinks to jump *within* a notebook to certain location or *between* notebooks.

Each cell can be assigned a style or format from a style sheet.

Mathematica notebooks are automatically adjusted for output on *screen* or *printer.* Fonts and layout for each medium are optimized.

Everything in the*Mathematica* Help Browser is a notebook - including the complete on-line *Mathematica* book with 1500 pages.

• *Mathematica* provides hundreds of options which allow to give notebooks any look to be wanted and to generate full *publication-quality documents.*

Here is an ordinary text. It can be any font, **face,** size, **color,** etc. There are hundreds of *special characters* such as 💡 as well as formulas such as $\int \frac{1}{x^3-1}\, dx$ embedded in text.

Mathematica makes it easy to set up *tables* and *arrays*

$\alpha^2 - \beta$	$(\alpha - \beta)\ (\alpha + \beta)$	$\alpha^2 - \beta^3$
$\alpha^3 - \beta$	$\alpha^3 - \beta^2$	$(\alpha - \beta)\ (\alpha^2 + \alpha\, \beta + \beta^2)$

Mathematica lets one set up *spacing* and *justification*

of

text

as wanted.

• Like other objects in *Mathematica*, the *cells* in a notebook, and in fact the *whole notebook* itself, are all ultimately represented as *Mathematica* expressions. With the standard notebook front end one can use the command `Show Expression` (which corresponds to the keyboard sequence CTRL+SHIFT+E) to see the underlying text of the *Mathematica* expression that corresponds to any particular cell.

The *Mathematica* language itself can be used to specify all aspects of notebooks.

Here is a typical cell in a notebook.

This shows how *Mathematica* represents the cell internally.

```
Cell[TextData[StyleBox["This  is a typical  (text)  cell.",
  FontWeight->"Bold",
  FontSlant->"Italic",
  Background->RGBColor[0,  1, 1]]], "Text",
  PageBreakBelow->False,
  CellTags->"T.8"]
```

***Mathematica* notebooks can be built up using explicit commands as well as interactively.**

This tells *Mathematica* to print three cells in *Subsubsection* style with numbering.

```
Do[StylePrint[StringJoin["Heading ", ToString[i]], "Subsubsection"], {i, 3}]
```

Heading 1

Heading 2

Heading 3

Palettes and Buttons

Palettes and buttons provide a simple but fully customizable interface to *Mathematica* with point-and-click features.

***Mathematica* comes with a collection of ready-to-use standard palettes.**

Here is part of the palette(Lists and Matrices Matrix Operations) for Basic Calculations

.

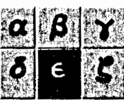
■.□
Cross[■, □]
Outer[■, □, □]
Det[■]
Inverse[■]
Transpose[■]
Eigenvalues[■]
Eigenvectors[■]
LinearSolve[■, □]
RowReduce[■]

Here is the standard palette International Characters for European characters

à	á	â	ã	ä	å	æ	ç
è	é	ê	ë	ì	í	î	ï
ð	ł	ñ	ò	ó	ô	õ	ö
ø		ú	û	ü	ý	ÿ	þ
ß	À	Á	Â	Ã	Ä	Å	Æ
Ç	È	É	Ê	Ë	Ì	Í	Î
Ï	Ð	Ł	Ñ	Ò	Ó	Ô	Õ
Ö	Ø	Ù	Ú	Û	Ü	Ý	Þ
£	¥	«	»	¿	¡		

Palettes work like *keyboard extensions.*

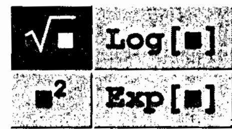
In a palette like this, pressing the key ■ inserts an ϵ into the notebook.



2 (1 + |

2 (1 + ϵ |

In a palette like this the placeholder ■ indicates where the current selection should be inserted.



Clicking the button takes the highlighted selection and wraps a square root around it.

1 + Sin[x] + Cos[x]

1 + √(Sin[x] + Cos[x])

It is easy to create own custom palettes.

One can create a blank palette using the **Create Table/Matrix/Palette** item in the Input menu. The empty symbols □ can be specified later.

Darken[■]	Lighten[■]
EdgeSelect[■]	□
□	□

One can create custom palettes to do any function or manipulate any expression by clicking a button.

Expand[■]
Factor[■]
Simplify[■]

Clicking the button immediately factors *in place* the expression selected.

1 + a^2 + 2 a b + b^2 + (p + q) ^2

1 + $(a + b)^2$ + (p + q) ^2

Use the following HyperLink to see other examples of palettes.

Here is an interactive *Mathematica* program PolyhedronExplorer.nb in order to manipulate *polyhedra*.

Mathematical Notation

Mathematica notebooks fully support standard mathematical notation - for both output and input.

***Mathematica* combines the *compactness* of mathematical notation with the *precision* of a computer language.**

Here is an integral input using only *ordinary keyboard characters*.

```
Integrate[Log[1 + x] / Sqrt[x], x]
```

$-4\sqrt{x} + 4\,\text{ArcTan}\left[\sqrt{x}\right] + 2\sqrt{x}\,\text{Log}[1 + x]$

Here is the same integral entered in *two-dimensional form* with special characters. One can enter this form using a palette or directly the keyboard.

$$\int \frac{\text{Log}[1 + \xi]}{\sqrt{\xi}}\, d\xi$$

$-4\sqrt{\xi} + 4\,\text{ArcTan}\left[\sqrt{\xi}\right] + 2\sqrt{\xi}\,\text{Log}[1 + \xi]$

This shows the keys one needs to type in order to get the input above. ⋮ stands for the ESC key.

```
⋮int⋮ Log[1 + ⋮x⋮] CTRL[/] CTRL[2] ⋮x⋮ CTRL[ ] ⋮dd⋮ ⋮x⋮
```

Mathematica always allows to edit output - use it again as input.

$-4\sqrt{\xi} + 4\,\text{ArcTan}\left[\sqrt{\xi}\right] + 2\sqrt{\xi}\,\text{Log}[1 + \xi]$

$$-4\sqrt{\xi} + \left(4\,\text{ArcTan}\left[\sqrt{\xi}\right] + 2\pi\right) + 2\sqrt{\xi}\,\text{Log}[1 + \xi]$$

Mathematica can generate output in *traditional textbook form.* Note that *Mathematica*'s **StandardForm** is precise and unambiguous whereas **TraditionalForm** requires heuristics for interpretation.

This input commands *Mathematica* to compute the integral and then display the result in *TraditionalForm.*

$$\int \frac{\text{Log}[1 + \xi]}{\sqrt{\xi}}\, d\xi \;\; \text{// TraditionalForm}$$

$4\tan^{-1}\left(\sqrt{\xi}\right) + 2\sqrt{\xi}\,\log(\xi + 1) - 4\sqrt{\xi}$

Mathematica produces top-quality output for formulas of any size or complexity.

$$\sum_{\mu=0}^{\infty} \frac{\varphi^{\mu}\,\text{Cos}\left[\frac{\pi\mu}{4}\right]}{\mu!^{2}\,(\mu^{2} + \kappa)\,(\mu^{2} - \lambda)} \;\; \text{// TraditionalForm}$$

$$(-\lambda\,{}_1F_2(-i\sqrt{\kappa};1,1-i\sqrt{\kappa};\sqrt[4]{-1}\,\varphi)-\lambda\,{}_1F_2(i\sqrt{\kappa};1,i\sqrt{\kappa}+1;\sqrt[4]{-1}\,\varphi)-$$
$$\kappa\,{}_1F_2(-\sqrt{\lambda};1,1-\sqrt{\lambda};\sqrt[4]{-1}\,\varphi)-\kappa\,{}_1F_2(\sqrt{\lambda};1,\sqrt{\lambda}+1;\sqrt[4]{-1}\,\varphi))/(4\kappa(\sqrt{\kappa}-i\sqrt{\lambda})(\sqrt{\kappa}+i\sqrt{\lambda})\lambda)+$$
$$(-\lambda\,{}_1F_2(-i\sqrt{\kappa};1,1-i\sqrt{\kappa};-(-1)^{3/4}\varphi)-\lambda\,{}_1F_2(i\sqrt{\kappa};1,i\sqrt{\kappa}+1;-(-1)^{3/4}\varphi)-$$
$$\kappa\,{}_1F_2(-\sqrt{\lambda};1,1-\sqrt{\lambda};-(-1)^{3/4}\varphi)-\kappa\,{}_1F_2(\sqrt{\lambda};1,\sqrt{\lambda}+1;-(-1)^{3/4}\varphi))/$$
$$(4\kappa(\sqrt{\kappa}-i\sqrt{\lambda})(\sqrt{\kappa}+i\sqrt{\lambda})\lambda)$$

Look at the Formula Gallery for other examples of mathematical formulas generated by *Mathematica.*

Mathematica makes it easy to work with abstract notation.

$$\texttt{Table}\left[\mathcal{G}\diamond\overline{\alpha_i\oplus\beta_i}\Longrightarrow\overset{6-i}{\underset{i}{\mathfrak{Z}}},\ \{i,\,6\}\right]$$

$$\{\mathcal{G}\diamond\overline{\alpha_i\oplus\beta_i}\Longrightarrow\overset{5}{\underset{1}{\mathfrak{Z}}},\ \mathcal{G}\diamond\overline{\alpha_i\oplus\beta_i}\Longrightarrow\overset{4}{\underset{2}{\mathfrak{Z}}},\ \mathcal{G}\diamond\overline{\alpha_i\oplus\beta_i}\Longrightarrow\overset{3}{\underset{3}{\mathfrak{Z}}},\ \mathcal{G}\diamond\overline{\alpha_i\oplus\beta_i}\Longrightarrow\overset{2}{\underset{4}{\mathfrak{Z}}},\ \mathcal{G}\diamond\overline{\alpha_i\oplus\beta_i}\Longrightarrow\overset{1}{\underset{5}{\mathfrak{Z}}},\ \mathcal{G}\diamond\overline{\alpha_i\oplus\beta_i}\Longrightarrow\overset{0}{\underset{6}{\mathfrak{Z}}}\}$$

Mathematica supports over 700 special characters with new fonts optimized for both screen and printer. All these characters can be found in the Complete Characters palette. All of them have consistent full names; some also have aliases, as well as TeX and SGML names.

Conclusion

In conclusion, the front end features of *Mathematica* version 3.0 are - compared with the previous version 2.2 - a break through. Mathematical typesetting in version 3.0 does in most cases what one wants. And even when one wants to do something out of the ordinary, options allow to override the default behavior, thus making the typesetting system extremely flexible. One can make changes and gets immediate feedback, or one can make changes to be part of some output form. If it turns out that one constantly want a few different changes or groups of changes to be immediately accessible, then one may create a palette with buttons that apply those changes with one mouse click.

Mathematica Notebooks

Notebook Technicalities

Prof. Dr. Robert Kragler
***FH Ravensburg-Weingarten** / University of Applied Sciences*
kragler@rz.fh-weingarten.de

Front End Features of *Mathematica* v3.0

■ ***Initializations***

Set Working Directory *"\\CD_Dubna98\\Lecture"*

PackageInfo

openNotebookButton & filesInDirectory

Load *Mathematica* Packages

Cell-Group with Icon ▹ resp. ▿

Set notebook options

Information on important System Variables

```
Print["Current working directory : `` ",Directory[] ]
Print["Initializations finished ",tag,zeit ]
```

```
Current working directory : `` D:\CD_Dubna98

Initializations finished on 1999.5.18 at 2:01 h
```

1 Using the *Mathematica* 3.0 Front End

1.1 Intoduction to Notebooks

The *Mathematica* system consists of two programs: the *front end* and the *kernel*. The front end sends questions to the kernel which in turn answers the questions and returns them to the front end for display.

Electronic documents created by the *Mathematica* front end are called *notebooks*. Because every *Mathematica* notebook is a *Mathematica expression* it can therefore be manipulated by the kernel.

■ 1.1.1 Notebooks

Notebooks are complete interactive document combining text, tables, graphics, sound, calculations, and other elements in terms of cells.

A notebook can contain styled text, pictures, animations, sounds, and typeset formulas. In addition, notebooks can contain live mathematical expressions, *hyperlinks* (e.g. the standard palette **International Characters** for European characters) and *buttons* that execute arbitrary *Mathematica* functions `NotebookE.nb`

One special type of notebook is a *palette*, e.g. `CompleteCharacters.nb` which typically contains a collection of special characters, 2D forms, or *Mathematica* functions to be pasted into the current notebook.

To create a new notebook, pull down the **File** menu and choose **New**.

■ 1.1.2 Cells

One fundamental difference between *Mathematica* notebooks and traditional word-processing documents is the notion of *cells*. Every piece of information in a notebook goes inside a separate cell, and the type of a cell determines the default appearance and properties of its content. E.g., mathematical questions go inside **Input** cells, whose contents are sent to the *Mathematica* kernel when evaluated the answers are placed inside new **Output** cells. Titles, section headings, and explanatory text go inside **Title, Section,** and **Text** cells etc., which are *not* sent to the kernel for interpretation.

Cells are denoted by *vertical bracket* on the rhs of the screen (here shown in red). When the key sequence SHIFT +RET is pressed after entering 50! in an input cell the result is the following

```
50!
```

```
30414093201713378043612608166064768844377641568960512000000000000
```

Cells can be collected into groups, which organize the contents of a *notebook* similar to the way a *book* is organized into chapters and sections. Above there is a (red) cell bracket, that groups the input and output cells.

Cell groups can be closed, which hides all the information in a cell group except for the *first* cell. To *close a cell group*, double-click the cell bracket that encloses the cell group.

There occurs a *small filled triangle* at the bottom of the closed cell group indicating that there are cells *inside* the closed group. Double-clicking the closed cell bracket opens the cell and reveals the hidden output cell.

A *horizontal line* crossing the screen is called the cell insertion point, which is where *Mathematica* will create a new cell (which is by default a new *input cell*). In order to create a new cell between two existing cells one must place the cell insertion point at the appropriate place. To move the cell insertion point, the (vertical) mouse pointer **I** has to be moved upward until it becomes a (horizontal) ⊢⊣ bar, which happens only when the pointer is *between* two existing cells. Pressing the left mouse button places the cell insertion point between two cells. When one starts typing, a new input cell is created at the cell insertion point.

(Pressing SHIFT +RET afterward places the matching output cell directly under the new input cell, places the cell insertion point beneath the new output cell.)

To create e.g. a *text cell* at the current insertion point, one has to change the cell type by first selecting the input cell bracket by clicking it with the mouse pointer. Next pull down the **Format** menu and chose **Text** from the **Style** submenu. By default, text cells have margins to the left of the margin for an input or output cell, and use a different font (here e.g. *Helvetica* resp. *Arial*).

One can insert *headings* by creating new cells, typing the text for each heading, and changing the cell into *section cell* by choosing **Section** from the Style submenu. S(ubsubs)ection cells usually use a *larger font size* than text cells, and have a *dingbat* such as □ , ■ , ■ or ● etc. to the left of the text. The cell dingbat can be changed by using in the Format menu the **Cell dingbat** submenu which offers a great number of different symbols etc. e.g. 💡 or ☺ . Of course the actual layout depends on the underlying *Stylesheet*

In the case of the *Classroom* Stylesheet the layout is as such:

Exercise Main

Exercise

Exercise Text

By default, *section cells* are set up so that they put *all* cells from the section heading down to the next section cell into a cell group. To *turn off automatic cell grouping*, choose **Cell** menu and select from the **Cell Grouping** submenu the option **Manual Grouping** ; the default is **Automatic Grouping**. To *group cells manually*, select the cells to be grouped, then choose from the **Cell Grouping** submenu the option **Group Cells** or the keystroke sequence CTRL+SHIFT+**G** .

Of course, one can insert other types of cells (depending on the actual Stylesheet) by creating a new cell and then change the cell type. For certain cell types **Title** etc. till **Input** there are keyboard short cuts such as ALT + **n** (with n = 1 ... 9).

By using **Show Toolbar** from the **Format** menu the notebook header will then have an extra toolbar which contains in particular a *popup window* with all *Cell types* available for this notebook and another *popup window* with the *Magnification.*

Cells resp. cell groups can be *deleted* by selecting them and choosing **Clear** from the **Edit** menu or DEL on the keyboard. Cell can be *merged* by selecting two or more adjacent cells, pulling down the **Cell** menu and choosing **Merge Cells** (or CTRL+SHIFT+**M**). A cell can be *divided* into two cells by placing the text insertion point at the place the cell should be divided, and choosing **Divide Cell** (or CTRL+SHIFT+**D**)

■ 1.1.3 Text Styles

Most of the front end's word-processing features such as Font, ***Faces***, Size, **Text color**, **Background color**, text alignment (i.e. *left, center* and *right*) and so on are under the **Format** menu. The format of the content of a cell can be changed by selecting the text to be changed, then making modifications with the commands under the Format menu. In order to change the format of one or several cells one has to select the cell bracket(s) and make the modifications the same way. The menu options **Show Ruler** and **Show Toolbar** under the **Format** menu make it easier to change a notebook's *margin, text alignment* and *cell styles.*

Besides the characters on the keyboard there are hundreds of special characters available such as ∅ . A convenient way is to choose from in the **File** menu from the **Palettes** submenu the following palettes CompleteCharacters or InternationalCharacters . In addition *inline formulas* such as $\int \frac{1}{x^5-1}\, dx$ are possible.

■ 1.1.4 Style Sheets

One can *change the style of cells* of a certain type in a notebook by editing the notebook's *style sheet*, which is by itself another *Mathematica* notebook document that contains cells with styles representative for all the cell styles that will be used when a new cell is created.

Usually, the style sheets such as Default.nb or Classroom.nb etc. that come with the system are all contained in the subdirectory

C:\Programs\Wolfram Research*Mathematica*\3.0\SystemFiles\FrontEnd\StyleSheets

```
dir = $TopDirectory <> "\\SystemFiles\\FrontEnd\\StyleSheets" ;
openNotebookButton["*.nb", 1, dir]
```

ArticleClassic.nb	ArticleModern.nb	Classic.nb
Classroom.nb	Default.nb	Demo.nb
DemoText.nb	HelpBrowser.nb	HTML.nb
MathsnFun.nb	NaturalColor.nb	NotepadMono.nb
Notepad.nb	PasteColor.nb	PrimaryColor.nb
Report.nb	Textbook.nb	TutorialBook.nb

It is, however, recommended to place additional user-defined style sheets into a separate (empty) directory

C:\Programs\Wolfram Research*Mathematica*\3.0\Configuration\FrontEnd\StyleSheets\

```
dir = $TopDirectory <> "\\Configuration\\FrontEnd\\StyleSheets" ;
openNotebookButton["*.nb", 1, dir]
```

ApplManual .nb	AWPageTrim .nb
AWStyles .nb	CD .nb
CDStyles .nb	TestStyle1 .nb
Training1 .nb	Training .nb

where they will show up together with the style sheets provided by the system

To *edit a style sheet*, pull down the **Format** menu and choose **Edit Style Sheet.**

To apply *changes only to current notebook*, click the **Import Private Copy** button. To edit a style sheet that defines styles for future new notebooks, click the **Edit Shared Style Sheet** button.

Warning : However, before changing a style sheet it is advisible to *keep a copy of the old style sheet* before editing. It may happen that the modified style sheet will turn out to be corrupted after saving it so that all *Mathematica* notebooks using this particular style sheet are affected likewise.

A typical style sheet is `Training .nb`

To *change the style of all cells of certain type*, one has to open the appropriate cell group in the style sheet notebook and select the *prototype cell* wanted.

If one wants to change e.g. the style of the prototype **Input** cell to *italic*, one may pull down the **Format** menu and choose **Italic** from the **Face** menu. As soon as the style sheet window is closed, all existing and new input cells in the notebook will have slanted text. Most versions of *Mathematica* come with several style sheets. Hence, one may choose a predefined style sheet by pulling down the **Format** menu and choosing the desired style sheet from the **Style Sheet** submenu.

1.1.5 Style Environments

The appearance of a notebook can be further changed by switching to a different *style environment*. Typical environments are

- **Working**
- **Presentation**
- **Condensed**
- **Printout**

Documents viewed in the *Presentation* environment typically use larger font sizes than documents in the *Working* environment. As regards to the *Condensed* environment there a font smaller than in the Working environment is used in order to display expressions which extend over the normal width of the screen. There is also the *Printout* environment, which uses smaller fonts than the Working environment, and uses styles that look better on the printed page. In addition all colored cell backgrounds are converted to gray scales.

In the **Format** menu there are two submenues **Screen Style Environment** and **Printing Style Environment** ; when chosen the style environment which are defined in the style sheet being used pop up and one can switch between different environments.

There is part of a notebook in the Working environment, in which text typically wraps around at the actual window width, and uses fonts large enough to be viewed comfortably on the screen

Use *Working* environment `TestTraining.nb`

In the *Printout* environment `TestTraining.nb` the same notebook uses smaller fonts and *line breaks* occur at the *page width* rather than on the actual *window width.* If one chooses right at the bottom of the **Format** menu the option **Show Page Breaks** then the style environment is switched to Printout so that the line breaks are visible. By default *cell brackets* are not printed. This can be controlled in the **Printing Settings** of the **File** menu, particularly in the **Printing Options** where there is an option **Print cell** bracket [^ √]

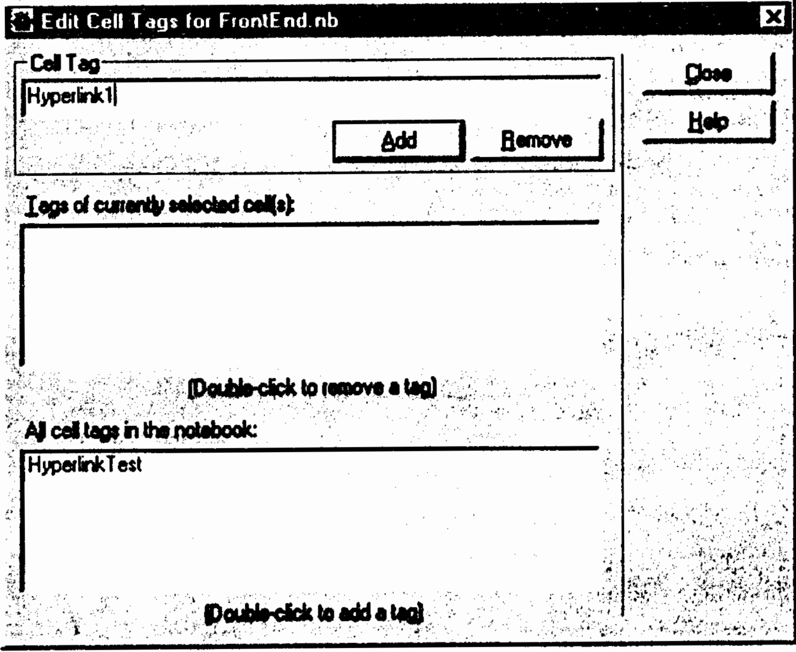
■ 1.1.6 Hyperlinks

Hyperlink1

To enter a *hyperlink* in a notebook, first select the destination of the link (i.e. the cell that the link should jump to), then add a *tag* to the the destination cell. A *tag* is a label associated with a cell that allows one to refer to a particular cell.

Tags are not displayed on the screen unless the option **Show Cell Tags** from the **Find** menu is chosen.

In order to add a tag to the cell one chooses **Add/Remove Cell Tags** in the **Find** menu, then type the desired tag into the uppermost text field. To add the tag to the cell, click the **Add** button.



Next, select the text one wants to act as the hyperlink, then pull down the **Input** menu and choose **CreateHyperlink.** In the dialog box that appears, one has to click the desired *target cell tag* from the list at the bottom of the dialog box.

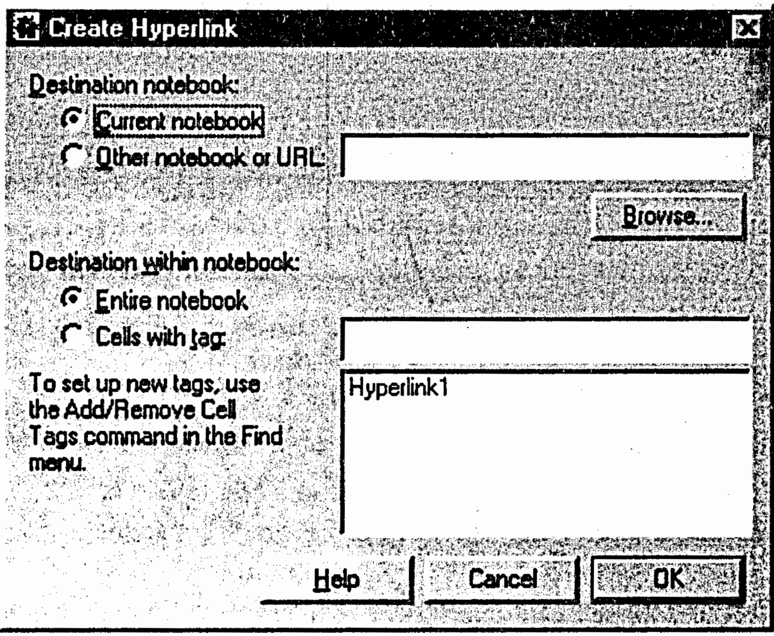


Figure 2

The selected text (here hyperlink) appears (by default) as underlined blue text. When a hyperlink is clicked the front end scrolls the selected notebook to the destination of the link (in the present case the tag **Hyperlink1**)

■ 1.1.7 Numbered Equations and Figures

To create numbered equations or figures one has to select the cell containing the equation or figure, then choose **NumberedEquation** (or **NumberedFigure**) from the **Style** submenu in the **Format** menu.

Note, that not every style sheet contains the numbered-equation or numbered-figure style definition. Some of the style sheets that do define automatic-numbering cells are Classroom, Classic, Report and Article styles.

Here is a cell containing some formulae inside numbered-equation cells.

$$\int dx = x \tag{1}$$

$$\int \lambda \, dx = \lambda x \tag{2}$$

$$\int x^n \, dx = \frac{x^{n+1}}{n+1} \quad (n \neq 1) \tag{3}$$

$$\int (f(x) + g(x)) \, dx = \int f(x) \, dx + \int g(x) \, dx \tag{4}$$

If an equation is inserted between existing cells, *Mathematica* automatically updates the equation numbers.

```
Plot[Sin[x]/x, {x, -4 π, 4 π}, PlotRange → All, Frame → True];
```

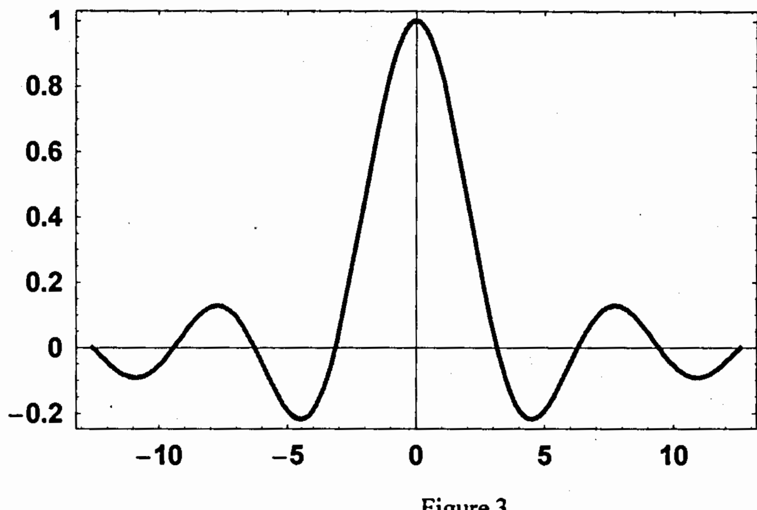


Figure 3

1.1.8 Converting Notebooks

Mathematica can save notebooks in several different formats, including **HTML**, **TeX** , **Package_Format** and **Pre-3.0_Notebook**. The different formats available are lissted under **Save_As_Special** submenu in the **File** menu.

Mathematica also opens files other than notebooks. Some files can be opened directly by choosing **Open** from the File menu, and others require use of **Open_Special**. The front end is set up automatically convert notebooks from versions of *Mathematica* earlier than 3.0.

Text and graphics in notebooks can be directly copied to the system clipboard and pasted into other programs. Some notebook elements, such as graphs and typeset formulae, can be converted into several formats e.g.

- **Text**
- **Cell Expression**
- **Notebook Expression**
- **Complete Notebook**
- **Package Format**
- **EPS**
- **Adobe Illustrator**
- **Bitmap (BMP)**
- **Enhanced Metafile (EMF)**
- **Windows Metafile (WMF)**
- **Rich Text (RFT)**
- **Wave (WAV)**

using commands under the **Save_Selection_As** and **Copy_As** submenus of the **Edit** menu.

See also `NotebookConvert, HTMLSave, TeXSave, Display, DisplayString`

```
?NotebookConvert
```

```
System`NotebookConvert

Attributes [NotebookConvert ] = {Protected , ReadProtected }

Options [NotebookConvert ] = {InputToStandardForm -> False ,
   OutputToStandardForm -> False , PreserveStyleSheet -> False ,
   GenerateBitmapCaches -> False , InputToInputForm -> False ,
   OutputToOutputForm -> False , Interactive -> False }
```

? HTMLSave

```
HTMLSave ["file .html ", notebook , options ] converts the Notebook
  object notebook into an html document .
```

? TeXSave

```
TeXSave ["file .tex ", notebook , options ] converts the Notebook object
  notebook into a TeX document .
```

? Display

Display [channel , graphics] writes graphics or sound to the specified output channel in *Mathematica* PostScript format . Display [channel , graphics , "format "] writes graphics or sound in the specified format . Display [channel , expr , "format "] writes boxes , cells or notebook expressions in the specified format .

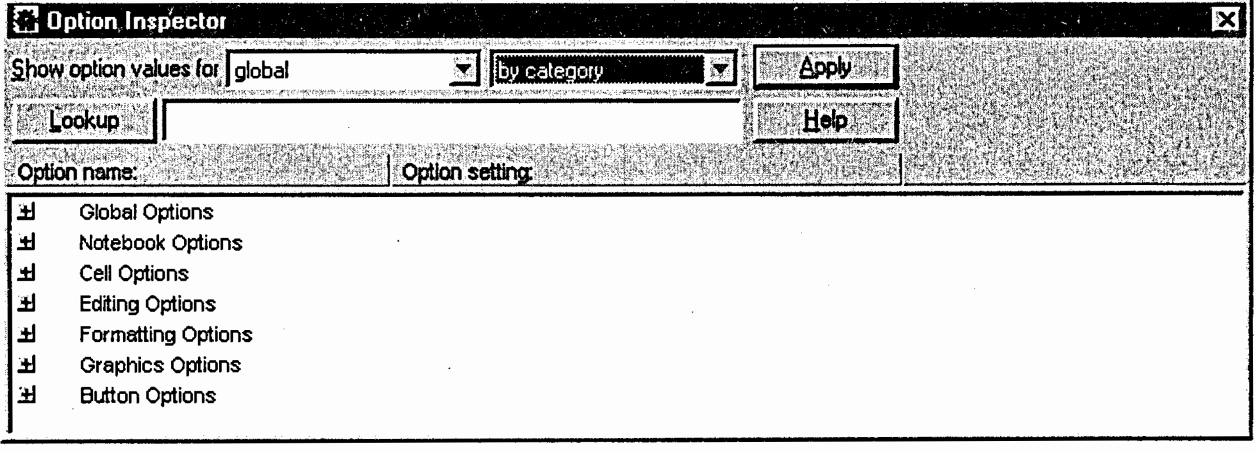
? DisplayString

DisplayString [graphics] generates a string giving graphics or sound in *Mathematica* PostScript format . DisplayString [graphics , "format "] generates a string giving graphics or sound in the specified format . DisplayString [expr , "format "] generates a string giving boxes , cells or notebook expressions in the specified format .

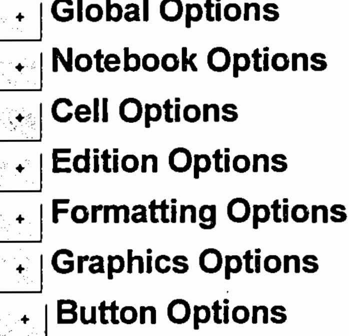
1.2 Cells and Notebook Options

1.2.1 Setting Options

Almost any feature of a notebook, cell, or expression can be altered using the **Option Inspector,** available under the **Format** menu. The **Option Inspector** window typically looks like the following figure



To reveal or hide the choices under on of the headings



To reveal or hide the choices under one of the headings (Global Options, Notebook Options, etc.) click the button to the left. When using the Option Inspector one can set the scope of the changes to make; that is, one can set options

- **for the current selection only ,**
- **for every cell with the same style as the current seclection,**
- **for the entire current notebook, or**
- **change the global default settings for options in all current and future notebooks.**

To *change the scope of the option settings*, select the ddesired scope from the pop-up menu near the top of the Option Inspector window.

To *remove any option settings* in a cell or notebook, pull down the **Format** menu and choose **Remove_Options.**

The **Lookup** text field allows one to find options pertaining to particular topics, without having to navigate the different headings. To find options pertaining to cells, for example, type "cell" into the text filed, and then click the Lookup button to reveal the next option name that contains the word "cell". A partial listing of the commands in the Option Inspector is contained in Appendix A of this tutorial.

■ 1.2.2 Global Options

Some of the global options that can be changed are

- **the list of palettes to open when *Mathematica* is started,**
- **the directories to search for configuration files,**
- **the default style sheet to use for new notebooks.**

To change global options, choose **Global** from the pop-up menu in the Option Inspector window.

For example, to turn off the *Mathematica* startup sound, open the **Global Options** category, open the **System Configuration** subcategory, and set the option **StartupSound** to **False.** There is typically a checkbox to the right of an option that takes only the values **True** or **False,** and clicking the box toggles the option settings.

One can also change the global editing and formatting behavior of the *Mathematica* front end. For example, the front end quickly flashes an open bracket (**[**, **{** or **(**) when the matching closing bracket (**]**, **}** or **)**) is typed. In the **Edition Options** category on can set the option **DelimiterFlashTime** to the length of time, in seconds, that the bracket should flash. (The default setting is 0.3 sec.) To turn delimiter flashing off, type 0 for the option value.

Also in the **Editing Options** category is an option called **DragAndDrop**. When set to **True** (the default value is **False**), one can highlight some text of part of a formula, and "drag" the selection to another location in the same cell.

■ 1.2.3 Notebook Options

To change the options for an entire notebook, choose **Notebook** from the pop-up menu in the Option Inspector, making certain the notebook or palette whose options to be changed is the currently selected notebook.

Some of the options one can change at the notebook level are the notebook's *background color*, the *magnification*, the *toolbars* and other elements present in a notebook, and the *window title*. Furthermore, one can change the notebook's printing properties, such as how many copies to pring and the range of pages to print. These and many other option settings are found in the **Notebook Options** category.

To show the timing for each evaluation in a notebook, open the **Notebook Options** category and the **Evaluation Options** subcategory, and then choose **Show_Timing** form the list of choices.

In the **Editing Options** category, one can change text options, such as whether and how much paragraphs should be intended, how text should be aligned, and how much space should fall between lines and paragraphs.

In the **Graphics Options** category, one can set options that control how graphics are rendered in the notebook. For example, in the **Rendering Options** subcategory one can tell *Mathematica* to draw dummy graphics in place of PostScript graphics (in order to save memory), and whether to draw lines and filled polygons in graphs.

■ 1.2.4 Cell Options

Some options one can change at the cell level are a *cell's background color*, how to *display cell brackets*, whether to *draw a frame around a cell*, and the *margins of a cell*. To change the option settings for one or more cells, select the cell(s) to change, choose **Selection** from the pop-up menu that sets the scope of an option, open the **Cell Options** category , and then make the desired changes. (To apply the changes to every cell in the notebook, choose Notebook from the pop-up menu in the Option Inspector.)

An important set of options for a cell are its *properties*. By default, most cells have the property that they are *editable*, that is, text inside a cell can be changed. To remove this property, select a cell bracket, pull down the **Cell** menu, and open the **Cell Properties** submenu. If the cell is *editable*, there will be a check or other indication that the editable property is currently set for the cell. Choosing **Cell Editable** from the **Cell Properties** submenu will toggle between the property being set or unset for the selected cell(s). If the editable property is unset, the text in the cell cannot be changed.

A property that input cells typically have is that they are *evaluatable*, meaning they can be sent to the kernel for interpretation. (Cells containing titles, section headings, and other text usually do not have this property set.) To make an input cell unevaluatable (which is desirable if the cell contains a typeset expression not intended for interpretation by the kernel), choose **Cell Evaluatable** from the **Cell Properties** submenu to unset the evaluatable property of the cell.

Other properties that cell can have set include the property that an attempt to edit the contents of a cell results in a copy of the cell being created (se **Cell Edit Duplicate** in the **Cell Properties** submenu - this property is set by default for output cells), and the property that the contents of a cell are automatically sent to the kernel, when the notobook containing the cell opened (**Initialization Cell**).

A cell's properties can be changed by opening the **General Properties** subcategory inside the **Cell Options** category in the Option Inspector. To change the properties for every cell of a particular type in a notebook, choose **Selection's Style** from the pop-up menu, or directly edit the notebook's style sheet.

2 Special Characters and Forms

2.1 Introduction

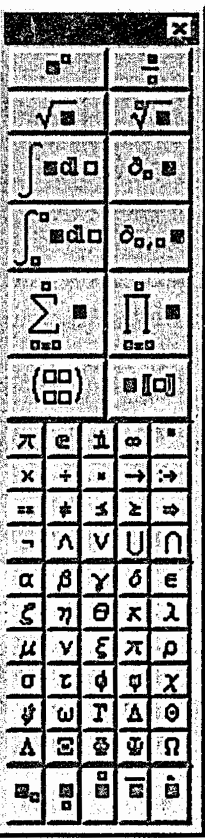
Mathematica works with several hundred special characters, including Greek, script, Gothik, and double-struck letters, accented Latin letters, relational signs, and mathematical operators. In addition, *Mathematica* interprets and generates many two-dimensional input and output forms for mathematical and other functions, and allows to create our own formatting rules for input and output.

One can create arbitarily sophisticated typeset formulas in many ways, using graphical features of the front end, keyboard shortcuts, and *Mathematica*'s programming language.

2.2 Using Palettes

One way to enter special characters and two-dimensional forms is to use a *palette*. Palettes are special notebooks that contain an array of buttons, and each button can be used as an extra key on the keyboard. For example, to enter the character α into a notebook, click a button containing an α, and the α character appears at the current insertion point.

On most systems, a palette that looks like the following figure will be present on the screen. (If not, pull down the **File** menu and choose **BasicInput** from the **Palettes** submenu.



To enter the expression **1+δ** into an input cell, type **1+** into a new input cell using the keyboard, then click the *δ* button. The *δ* appears at the current text insertion point, after the plus sign.

The same palette can be used to enter some two-dimensional expressions. To enter $\sqrt{(1+\delta)}$ into a notebook, click the √■ button, then type **1+δ** into the space that appear under the radical sign. (Pressing the arrow key or CTRL+SPACE gets the cursor out of the radical sign.) Another way is to type and select **1+δ**, then click the √■ button on the palette.

Here are the steps to enter the expression **Expand[(α + β)^10]** into an input cell.

First type **Expand[** int an input cell. With the text insertion point still located after the bracket, click the exponent button ■□. In the lower placeholder type (*α+β*) using a palette to enter the *α* and *β* characters.

```
Expand[(α + β)□
```

Next click in the exponent placeholder with the mouse. (Pressing the TAB key also moves the text insertion point into the next empty placeholder.) When the insertion point is in place, type **10** for the exponent. Pressing the right arrow key → or CTRL+SPACE moves the text insertion point out of the exponent position and back to the baseline of the expression. With the insertion point at the proper location, then type the closing bracket], and SHIFT+RET to evaluate the expression.

$$\text{Expand}[(\alpha+\beta)^{10}]$$

$$\alpha^{10} + 10\,\alpha^9\,\beta + 45\,\alpha^8\,\beta^2 + 120\,\alpha^7\,\beta^3 + 210\,\alpha^6\,\beta^4 + 252\,\alpha^5\,\beta^5 + 210\,\alpha^4\,\beta^6 + 120\,\alpha^3\,\beta^7 + 45\,\alpha^2\,\beta^8 + 10\,\alpha\,\beta^9 + \beta^{10}$$

Another way to do the same thing is to type everything except the exponent into a cell, and select the expression to be the base of the exponent.

```
Expand[(α+β)]
```

With (*α*+*β*) selected, click the exponent button, and *Mathematica* puts an *empty placeholder* (□) at the superscript position, into which the exponent `10` is typed. Pressing SHIFT+RET evaluates the expression.

In general, clicking a *button that contains a formula or special character* places the character at the current text insertion point in the selected notebook (which usually has a highlighted title bar), replacing any selected text.

Clicking a button that contains a selection placeholder ■ "wraps" the contents of the button around the current selection, if any, replacing the ■ with the selected expression. If nothing is currently selected, clicking a button pastes the blank form into the notebook at the current insertion poit. To fill in any empty placeholder (□), click the placeholder with the mouse pointer (the TAB key also moves the text insertion point from one placeholder to the next), then type the desired expression, pressing SHIFT+RET when finished to evaluate the expression.

Choosing the CompleteCharacters palette from the **Palettes** submenu of the **File** menu brings up a palette containing every special character in the *Mathematica* character set (click a triangle ▷ to expand a particular section of the palette), and choosing BasicTypesetting from the **Palettes** submenu offers a larger selection of buttons for entering two-dimensional typeset forms.

2.3 Character Full Names

Every special character in *Mathematica* has a full name of the form `\[FullName]`, and typically the *Mathematica* front end converts the form `\[FullName]` into the named character once the closing square bracket is typed. (In order to prevent the front end from processing the form `\[FullName]` use before the closing bracket] an *invisible comma* :,:)

For example, to enta a character π using its full name, type `\[Pi]`; and to enter Π type `\[CapitalPi]`. As with all *Mathematica* names, the full names of characters are case sensitive. To enter the expression `Expand[(α+β) (θ+Δ)]`, type the following characters and then press SHIFT+ RET. By default, *Mathematica* converts `\[Alpha]` into α, `\[Beta]` into β, and so forth, once each closing square bracket is typed.

```
Expand[(\[Alpha] + \[Beta]) (\[Theta] + \[CapitalDelta])]
```

```
α Δ + β Δ + α θ + β θ
```

When using the **CompleteCharacters** palette, *Mathematica* displays the full name of a character when the mouse pointer is over a button containing the character. In chapter 6 "*Structure of Typeset Expressions*", the internal representation of two-dimensional typeset forms is examined in more detail.

See also **ShowSpecialCharacters, ToCharacterCode, FromCharacterCode, CharacterRange**

```
? ShowSpecialCharacters
```

```
ShowSpecialCharacters  is an option for Cell which specifies whether
  to replace [Name], :nnnn, etc. by explicit special characters.
```

```
? ToCharacterCode
```

ToCharacterCode["string"] gives a list of the integer codes corresponding to the characters in a string. ToCharacterCode["string", "encoding"] gives integer codes according to the specified encoding.

```
? FromCharacterCode
```

FromCharacterCode[n] gives a string consisting of the character with integer code n. FromCharacterCode[{n1, n2, ... }] gives a string consisting of the sequence of characters with codes ni. FromCharacterCode[... , "encoding"] generates a string using the specified character encoding.

```
? CharacterRange
```

CharacterRange["c1", "c2"] yields a list of the characters in the range from "c1" to "c2".

2.4 Menus

Under the **Edit** menu there is a submenu called **Expression Input**, in which the commands for entering *subscripts, superscripts, underscripts,* and *overscripts*, as well as other two-dimensional forms are listed.

To enter a *subscripted expression* such as x_0 type the base **x**, choose **Subscript** from the **Expression Input** submenu, fill in the base, and then press the right arrow key or CTRL+SPACE to move the insertion point back to the baseline of the expression.

x_0

Similarly, to enter expression $\hat{x}$, type an **x**, choose **Above** from the **Expression Input** submenu, and type the caret character ^ . (Note, that on a non-standard English keyboard, *Mathematica* will almost always accept CTRL**+6** but may *not* accept CTRL+^)

$\hat{x}$

To enter a *matrix* or other array-like form, pull down the **Input** menu and choose the option **CreateTable/Matrix/Palette** ,click the **Matrix** button, and then fill in the blanks in the dialog box that appears.

$\text{Det}\left[\begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix}\right]$

```
-1
```

To *add a row or column* to an already existing matrix, select the pull down the **Edit** menu and choose from the **Expression Input** submenu **AddRow** (or **AddColumn**) .

2.5 Keyboard Aliases and Shortcuts

Many *special characters* have a keyboard alias of the form ⋮*alias*⋮ where the character ⋮ (or [ESC] **esc** [ESC]) represents the Escape key [ESC]. *Mathematica* typically converts the form ⋮*alias*⋮ into the corresponding character once the second ⋮ is typed. To enter a π, type [ESC] **p** [ESC]; and to enter a **Π** type [ESC] **P** [ESC] because keyboard aliases are case sensitive. Most one-letter aliases stand for Greek letters; hence [ESC] **m** [ESC] gives μ, [ESC] **G** [ESC] is **Γ**, and so forth.

To enter the command `Solve[4ρ+5μ==0,ρ]`, type

```
Solve[ 4 ⋮r⋮ + 5 ⋮m⋮ == 0, ⋮r⋮]
```

$$\{\{\rho \to -\frac{5\mu}{4}\}\}$$

Other families of letters are

Greek letters	[ESC] G [ESC]	Γ
double-struck letters	[ESC] dsA [ESC]	𝔸
Gothic letters	[ESC] goA [ESC]	𝔄
script letters	[ESC] scA [ESC]	𝒜
accented Latin	[ESC] A punctuation [ESC]	Ä

To enter a character for a particular font family, just prepend to the character the abbreviation for the font family.

The aliases for *accented Latin characters* are typically the letter with a *punctuation mark* appended that resembles the accent to be used. Some aliases for accented letters are ⋮a~⋮ for ã, ⋮c,⋮ for ç, ⋮e`⋮ for è, ⋮o/⋮ for ø, and ⋮Ao⋮ for Å etc.

Greek, double-struck, Gothic, script and extended Latin letters can be used in variables and function names too

```
𝓅ℴ𝓁𝓎ℊℴ𝓃[n_Integer] := Table[{Cos[θ], Sin[θ]}, {θ, 0, 1.99 π, 2 π/n}]
𝓅ℴ𝓁𝓎ℊℴ𝓃[3]
```

$$\{\{1, 0\}, \{-\frac{1}{2}, \frac{\sqrt{3}}{2}\}, \{-\frac{1}{2}, -\frac{\sqrt{3}}{2}\}\}$$

When using the **CompleteCharacters** palette, placing the mouse pointer over a special character displays the keyboard alias for that character, if one is defined.

There are also *keyboard shortcuts* for most *two-dimensional forms*, most involving use of the [CTRL] key. In many cases, the keyboard shortcuts for two-dimensional forms mimic the keyboard form for one-dimensional commands. For example, enter an exponent using a caret ^, which is entered on most keyboards by typing [CTRL][6]. To enter the two-dimensional form of 2^{100}, after typing the base 2, press either [CTRL][6] (or [CTRL][^]), which moves the text insertion point into the superscript position, and then fill in the placeholder. Thus, the following keystrokes will create input and output :[CTRL][2] 2[CTRL][6] 100 [SHIFT]+[RET]

$$\sqrt{2^{100}}$$

```
1125899906842624
```

Similarly, in one dimension a *fraction* is typed as **x/y** ; to enter the two-dimensional form type the keystrokes **x**[CTRL][/]**y**

$\{x/y, \frac{x}{y}\}$

$\{\frac{x}{y}, \frac{x}{y}\}$

Overscripts and *underscripts* are created by typing [CTRL][7] and [CTRL][=]. To type $x \overset{f}{\rightarrow} x^2$, type $x \rightarrow x^2$

(where → is entered as \ **[RightArrow]** , then select the arrow, and then press [CTRL][7]. Then type the contents of the overscript (f) into the placeholder above the arrow. Note, that the expression is not a valid *Mathematica* input, hence, one should not press [SHIFT][RET] after creating the form.

In general, pressing a keyboard shortcut for a *subscript*, *superscript, fraction,* etc. applies the form to the first *complete subexpression* directly before the insertion point. If one types **a+b+c** into an input cell and then press [CTRL][6] for a superscript (exponent), the superscript is applied only to **c**.

$a + b + c^x$

On the other hand, if on types **a+ (b+c)** followed by [CTRL][6], the exponent is applied to the entire expression **(b+c)**.

$a + (b + c)^x$

If part of an expression is highlighted, the subscript or superscript is applied to the *entire selection*. Here, type **a+b+c+d** into an input cell, then select the subexpression **b+c**

a+■**+d**

Pressing the keyboard shortcut for a fraction, i.e. [CTRL][/], places the selected expression in the numerator position of the fraction

$a + \frac{b + c}{\blacksquare} + d$

Some characters in typeset expressions have *both* a *subscript* and *superscript*, or both an *underscript* or *overscript*. In such an expression, the keyboard shortcut [CTRL][5] (or [CTRL][%]) moves to the opposite position in the expression. To enter the expression x_0^2, type the base of the expression and then one of the scipts, in this case the subscript, by typing **x**[CTRL][-]**0** which gives x_0. With the insertion point still in the subscript position, pressing [CTRL][5] creates a placeholder in the superscript position and moves the insertion point into it, where one can type the superscript **2**.

x[CTRL][-]**0** [CTRL][5]**2**   results in x_0^2

Similarly, to enter a summation first type the character \ **[Sum]** (which can be entered as **:sum:**, and which is different from the Greek letter \ **[CapitalSigma]**).

Σ

The variable of summation and the lower limit are usually placed as an underscript to the summation sign, and one moves to the *underscript* position by typing [CTRL][=], so that

$\sum_{i=1}$

To move from the underscript to the overscript position, type [CTRL][5], and then type the upper limit of the summation

$$\sum_{i=1}^{n}$$

Then press the right arrow key or [CTRL][SPACE] to move out of the overscript position, type the function to be summed and press [SHIFT][RET] to evaluate the expression.

$$\sum_{i=1}^{n} i^8$$

$$\frac{1}{90} n (1+n) (1+2n) (-3+9n-n^2-15n^3+5n^4+15n^5+5n^6)$$

The **Other Information** category of the **Help** Browser contains further examples of entering and editing two-dimensional expressions. NotationExamples .nb

See also the documentation in the subdirectory **"...\OtherInformation"**

```
dir = $TopDirectory <> "\\Documentation\\English\\OtherInformation";
filesInDirectory[ dir ] // ColumnForm
```

```
BrowserIndex .nb
CellMenu .nb
ContactInfo .nb
EditingCommands .nb
EditMenu .nb
FileMenu .nb
FindMenu .nb
FormatMenu .nb
HelpMenu .nb
InputMenu .nb
KernelMenu .nb
MacKeyboardCommands .nb
NeXTKeyboardCommands .nb
NotationExamples .nb
StyleSheets .nb
WindowMenu .nb
WindowsKeyboardCommands .nb
XKeyboardCommands .nb
```

Special input forms based on control characters are summarized in the table below

[CTRL][@] or [CTRL][2] or \! \(\@x \)	go into a *square root*	$\sqrt{\blacksquare}$	because of key (@ / 2)
[CTRL][^] or [CTRL][6] or \! \(x\^y \)	go to *superscript* position	$\blacksquare^{\square}$	because of key (^ / 6)
[CTRL][_] or [CTRL][-] or \! \(x_y \)	go to *subscript* position	$\blacksquare_{\square}$	because of key (_ / -)
[CTRL][+] or [CTRL][=] or \! \(x\+y \)	go to *underscript* position	$\underset{\square}{\blacksquare}$	because of key (+ / =)
[CTRL][&] or [CTRL][7] or \! \(x\&y \)	go to *overscript* position	$\overset{\square}{\blacksquare}$	because of key (& / 7)
[CTRL][%] or [CTRL][5] or \! \(x\^y\%z \)	go from *subscript* to *superscript* and vice versa or to *exponent* position *in a root*	$\blacksquare_{\square}^{\square}$	because of key (% / 5)

On a standard English-language keyboard, the *shifted versions* of keystrokes in the first column are accepted on standard English-language keyboards. However, on a *non-English* keyboard *Mathematica* will almost always accept the *non-shifted versions* in the second column.

A third alternative are the \! *sequences* followed by \(... \) in the third column using ordinary printable characters from the keyboard. Note, that if one *copies* a \! sequence into *Mathematica,* it will automatically jump into two-dimensional form. But if one enters the sequence *directly from the keyboard,* the \(... \) sequences are shown in literal form. Choosing the **Make_2D** item (or the keystrokes SHIFT+ CTRL+**Y**) from the **Edit** menu convert these sequences into two-dimensional forms wanted.

2.6 Extensible Characters

Some characters that *Mathematica* uses are *extensible,* meaning they may stretch to surround an expression that is taller or wider than a single character. Different types of brackets and arrows are typically extensible characters.

For example, the *parentheses* surrounding a matrix are extensible. Here is a 2 x 2 matrix.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

If an *additional row* is added to the matrix, the parentheses extend to the new height of the matrix.

$$\begin{pmatrix} a & b \\ c & d \\ e & f \end{pmatrix}$$

The following input line is one-dimensional, so the *square brackets* are as tall as a single character

```
Gamma[11 / 2]
```

$$\frac{945\sqrt{\pi}}{32}$$

Entering the argument $\frac{11}{2}$ in two-dimensional form, however, causes the *brackets* to grow vertically.

$$\text{Gamma}\left[\frac{11}{2}\right]$$

The characters **\[UnderBracket]** and **\[OverBracket]** are examples of characters that grow horizontally. Here is a simple expression.

1+2+3

To put a horizontal bracket above the entire expression, select the whole sum, and then click a $\overset{\square}{\blacksquare}$ button in a palette or press CTRL[7]. Then type **\[OverBracket]** or its alias **:o[:** in the placeholder above the sum. *Mathematica* extends the bracket to the length of the sum

$$\overline{1 + 2 + 3}$$

If terms to the sum are added, the brackets extends to the proper length.

$$\overline{1 + 2 + a + b + 3}$$

There are some special *expandable characters* such as *parentheses, arrows, bracketing bars* which grow without bound to span whatever expressions they contain, whereas *brackets, brashes* and *slashes* grow by default to limited

size only. These default characteristics of expandable characters, however, can be controlled by **StyleBox** options, e.g.

```
RowBox[{"(", "{", GridBox[{{X}, {Y}, {Z}}]}] // DisplayForm
```

```
( X
({ Y
(  Z
```

However, with option **SpanMaxSize→Infinity** specifies that all characters inside the **StyleBox** should be allowed to grow as large as they need.

```
StyleBox[RowBox[{"{", GridBox[{{X}, {Y}, {Z}}]}], SpanMaxSize → Infinity] //
 DisplayForm
```

```
⎧X
⎨Y
⎩Z
```

By default expandable characters grow *symmetrically*. For a **GridBox**, one can use the option **GridBaseline** to specify where the baseline should be taken to lie. The possible settings are **Center, Top, Bottom** and **Axis** (default)

```
{ GridBox[{{■, ■}, {■, ■}}, GridBaseline → Top],
  GridBox[{{□, □}, {□, □}}, GridBaseline → Center],
  GridBox[{{□, □}, {□, □}}, GridBaseline → Axis],
  GridBox[{{■, ■}, {■, ■}}, GridBaseline → Bottom]} // DisplayForm
```

```
                           ■  ■
{        □  □    □  □
 ■  ■ ,  □  □ ,  □  □ ,    ■  ■}
 ■  ■
```

which explains the following arrangement of the **GridBox**

```
(rb = RowBox[{ "(", GridBox[{{X}, {Y}}, GridBaseline → Bottom], ")" }]) //
 DisplayForm
```

```
⎛X⎞
⎜Y⎟
⎝ ⎠
```

Setting **SpanSymmetric→False** allows expandable characters to grow asymmetrically.

```
{ StyleBox[rb, SpanSymmetric → False], Z } // DisplayForm
```

```
{(X), Z}
  (Y)
```

The notebook front end typically provides a **Spanning_Characters** submenu in **Edit\Expression_Input** which allwos one to change the spanning characteristics of all characters within the current selection.

The *limits of a sum* are usually displayed as underscripts and overscripts.

```
s = Sum[f[i], {i, 0, n}]
```

$$\sum_{i=0}^{n} f[i]$$

When the sum is shown smaller, it is conventional for the limits to be displayed in subscript and superscript position

```
1 / s
```

$$\frac{1}{\sum_{i=0}^{n} f[i]}$$

The option `LimitsPositioning` (with default value `Automatic`) to `UnderoverscriptBox` and related boxes controls whether to change positioning of limits in the conventional way.

```
(s = UnderoverscriptBox["∑", "i=0", "n", LimitsPositioning → False]) //
  DisplayForm
```

$$\underset{i=0}{\overset{n}{\sum}}$$

In order to avoid that the underscript and overscript positions are moved to subscript and superscript positions in the case of a fraction, use the option `LimitsPositioning→False` instead of `True` resp. `Automatic`.

```
FractionBox[ "1", s] // DisplayForm
```

$$\frac{1}{\underset{i=0}{\overset{n}{\sum}}}$$

Here is a sophisticated example of *extensible characters* which shows how horizontal brackets span the objects they enclose.

$$\text{Table}\left[\overbrace{\underbrace{\text{Factor}[x^i - 1]}}, \{i, 5\}\right]$$

$$\{\overline{\underline{-1 + x}}, \overline{\underline{(-1 + x)\ (1 + x)}}, \overline{\underline{(-1 + x)\ (1 + x + x^2)}}, \overline{\underline{(-1 + x)\ (1 + x)\ (1 + x^2)}}, \overline{\underline{(-1 + x)\ (1 + x + x^2 + x^3 + x^4)}}\}$$

See also `OverBar, OverVector, UnderBar,`

`SpanMaxSize, SpanMinSize, SpanSymmetric`

```
? SpanMaxSize
```

```
? SpanMinSize
```

```
? SpanSymmetric
```

2.7 Typeset Expressions within Text (*InlineInput*)

After creating a typeset expression, one can copy it and paste it into the text contained in a text, title, section, or any other cell using the commands in the Edit menu. Here is a two-dimensional expression inside an input cell

$$\frac{1}{\sqrt{2\pi}}$$

And after creating the expression, select it and choose Copy or Cut from the Edit menu, and then paste it at the desired location inside a text cell.

Here is a formula in text : $\frac{1}{\sqrt{2\pi}}$

Alternatively, *inside text* one can create *inline cells* that contain typeset structures. To create an inline typeset cell inside a text cell, begin by typing text as usual. When one wishes to start entering a typeset expression, press [CTRL][9] or [CTRL][(] , then enter a two-dimensional form using palettes and keyboard shortcuts. (Usually ther is a frame or other indication of the boundaries of the inline cell.) When one has finished entering a typeset expression in text, press [CTRL][0] or [CTRL][)] , and continue typing ordinary text.

Begin of Inline Mode : [CTRL][(... *inline text or formula* ... [CTRL][)] end of Inline Mode

2.8 Examples

Note, that many of the following examples are not valid as *Mathematica* input, so do not press [SHIFT][RET] to evaluate them.

$$(\alpha+\beta)^n \neq \alpha^n+\beta^n \quad (5)$$

$$\hbar \simeq 6.6260754 \ \frac{\text{Joule}}{\text{sec}} \quad (6)$$

$$F_n = F_{n-1} + F_{n-2}, \ n \quad 2 \quad (7)$$

$$(\forall y)(\forall z)(y \in x \wedge z \in y \Longrightarrow z \in x) \quad (8)$$

$$2^k n \sin\left(\frac{\theta}{2^k}\right) < 2^k < 2^k n \tan\left(\frac{\theta}{2^k}\right) \quad (9)$$

$$d = \sqrt{(x_2-x_1)^2+(y_2-y_1)^2} \quad (10)$$

$$\sin\frac{\theta}{2} = \sqrt{\frac{1-\cos\theta}{2}} \quad (11)$$

$$\tan(\theta\pm\phi) = \frac{\tan\theta\pm\tan\phi}{1\mp\tan\theta\tan\phi} \quad (12)$$

$$\left(\hat{\Pi}\right)^{(0)}_{\alpha\beta} = n\, g(n,\epsilon)\, u_\alpha\, u_\beta + p\,\delta_{\alpha\beta} \quad (13)$$

$$1/\pi = \left(\sqrt{8}\right)/9801 \sum_{i=1}^{\infty} ((4n!)(1103+26390n))/((n!)\hat{}4\,396\hat{}(4n)) \quad (14)$$

In order to expand the bracket in the way wanted one is referred to the discussion in the previous section 2.7.

$$U(x) = \begin{cases} 0, & x<0 \\ \frac{1}{2}, & x=0 \\ 1, & x>0 \end{cases} \quad (15)$$

$$A\ \text{adj}(A) = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{i1} & a_{i2} & \cdots & a_{in} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{j1} & \cdots & C_{n1} \\ C_{12} & C_{22} & \cdots & C_{j2} & \cdots & C_{n2} \\ \vdots & \vdots & & \vdots & & \vdots \\ C_{1n} & C_{2n} & \cdots & C_{jn} & \cdots & C_{nn} \end{pmatrix} \quad (16)$$

In order to create the *extensible overbar* use `OverscriptBox["expr","_"]//DisplayForm`

$$\langle f,g\rangle = \int_a^b [f_1(x) + i\, f_2(x)]\, \overline{[g_1(x)++ i\, g_2(x)]}\; dx \tag{17}$$

$$\iint (f \nabla g - g \nabla f)\cdot n\; d\sigma = \iiint (f \nabla^2 g - g \nabla^2 f)\; d\tau \tag{18}$$

In order to avoid the underscript and overscript positions for the summation or product signs use the option `LimitsPositioning→False` (instead of `True` resp. `Automatic`) for the `UnderoverscriptBox` in order to control the positioning of limits

$$\| A \|^2 = \sum_{i=1}^{n} \sum_{j=1}^{m} |a_{ij}|^2 \tag{19}$$

$$\prod_{i=0}^{n} p_i^{k_i} \tag{20}$$

$$\left((\sqrt{2})^{\sqrt{2}} \right)^{\sqrt{2}} = 2 \tag{21}$$

```
StyleBox[ SuperscriptBox[
        RowBox[{ StyleBox[ "(", FontSize → 20],
              SuperscriptBox[ "(√2 )", "√2 "],
              StyleBox[ ")", FontSize → 20]
            }] , "√2 "],
              SpanMaxSize → Infinity,
              ScriptBaselineShifts → {Automatic, 1} ] // DisplayForm
```

$$\Pi(F,G) \approx 1 - \Phi \left[\frac{c - \frac{1}{2} - m\, n\, p_1}{\sqrt{\mathrm{Var}(W_{XY})}} \right] \tag{22}$$

$$\mathrm{Cov}\left(\sum_{i=1}^{r} K_i ,\ \sum_{j=1}^{s} L_j \right) = \sum_{i=1}^{r} \sum_{j=1}^{s} \mathrm{Cov}(K_i, L_j) \tag{23}$$

$$N(a_1' a_2' \ldots a_r') = N - \sum_{i} N(a_i) + \sum_{i \neq j} N(a_i\, a_j) - \sum_{i,j,k} N(a_i\, a_j\, a_k) + \ldots (-1)^r\, N(a_1\, a_2 \ldots a_r) \tag{24}$$

3 Doing Calculations

3.1 Different Input and Output Forms

Mathematica allows a few different forms for input and output of mathematical expressions which are suitable for different purposes.

Versions of *Mathematica* prior to 3.0 allowed input only in a one-dimensional form called **`InputForm`**. Expressions in **`InputForm`** use only ordinary keyboard characters, and requires the use of spelled-out function names and square brackets to enclose function arguments in order to make questions unambiguous.

```
Integrate[1 / (x^4 + 1), x] // OutputForm
```

```
          -Sqrt[2] + 2 x                Sqrt[2] + 2 x
ArcTan[--------------]      ArcTan[-------------]
             Sqrt[2]                    Sqrt[2]
----------------------- + ---------------------- -
       2 Sqrt[2]                  2 Sqrt[2]

                              2                              2
  Log[-1 + Sqrt[2] x - x ]     Log[1 + Sqrt[2] x + x ]
  ------------------------ + -----------------------
        4 Sqrt[2]                    4 Sqrt[2]
```

The corresponding *output form* is **`OutputForm`**, a two-dimensional form which approximates traditional mathematical notation by printing numerators, denominators, and exponents on separate lines. Expressions in **`OutputForm`** cannot be edited directly; to create an editable copy of an expression in **`OutputForm`** place the cell insertion point directly und the **`OutputForm`** expression, then choose **Copy Output from Above** from the **Input** menu. The result of the integral above displayed using **`OutputForm`** is given below.

```
          -Sqrt[2] + 2 x                Sqrt[2] + 2 x
ArcTan[--------------]      ArcTan[-------------]
             Sqrt[2]                    Sqrt[2]
----------------------- + ---------------------- -
       2 Sqrt[2]                  2 Sqrt[2]

                              2                              2
  Log[-1 + Sqrt[2] x - x ]     Log[1 + Sqrt[2] x + x ]
  ------------------------ + -----------------------
        4 Sqrt[2]                    4 Sqrt[2]
```

The default form of input and output in *Mathematica* 3.0 is **`StandardForm`** which uses special characters and two-dimensional forms. **`StandardForm`** understands input (and generaated output) using radical signs, superscripted exponents, and two-dimensional fractions; as was as traditional input forms for operators such as integrals, sums, and products, and some special form for constants, such as π and ∞ for **`Pi`** and **`Infinity`**.

$$\int \frac{1}{1+\Phi^4}\, d\Phi \quad \texttt{// StandardForm}$$

Mathematica by default generates output in **`StandardForm`**. Note that **`StandardForm`** uses *Mathematica* names for mathematical functions (**`ArcTan`** and **`Log`** for arctangent and logarithm) and square brackets for enclosing function arguments, making output in **`StandardForm`** *unambiguous* and therefore suitable for use as further input.

$$\frac{\text{ArcTan}\left[\frac{-\sqrt{2}+2\,\Phi}{\sqrt{2}}\right]}{2\sqrt{2}} + \frac{\text{ArcTan}\left[\frac{\sqrt{2}+2\,\Phi}{\sqrt{2}}\right]}{2\sqrt{2}} - \frac{\text{Log}\left[-1+\sqrt{2}\;\Phi-\Phi^2\right]}{4\sqrt{2}} + \frac{\text{Log}\left[1+\sqrt{2}\;\Phi+\Phi^2\right]}{4\sqrt{2}}$$

To edit output in **StandardForm**, type directly inside the expression; the front end normally creates a copy of the output in a new input cell, and one edits the copy.

$$\frac{\text{ArcTan}\left[\frac{-\sqrt{2}+2\,\Phi+111111}{\sqrt{2}}\right]}{2\sqrt{2}}+\frac{\text{ArcTan}\left[\frac{\sqrt{2}+2\,\Phi}{\sqrt{2}}\right]}{2\sqrt{2}}-\frac{\text{Log}\left[-1+\sqrt{2}\,\Phi-\Phi^2\right]}{4\sqrt{2}}+\frac{\text{Log}\left[1+\sqrt{2}\,\Phi+\Phi^2\right]}{4\sqrt{2}}$$

Another input and output form availabe in *Mathematica* 3.0 is **TraditionalForm**, which mimics traditional mathematical and technical notation with italic single-letter variabl names, parentheses instead of square brackets surrounding the arguments to a function, and traditional forms for special functions (***tan***$^{-1}$ and ***log*** for **ArcTan** and **Log**)

$$\int \frac{1}{1+\Phi^4}\,d\Phi \quad \text{// TraditionalForm}$$

$$\frac{\tan^{-1}\left(\frac{2\Phi-\sqrt{2}}{\sqrt{2}}\right)}{2\sqrt{2}}+\frac{\tan^{-1}\left(\frac{2\Phi+\sqrt{2}}{\sqrt{2}}\right)}{2\sqrt{2}}-\frac{\log\left(-\Phi^2+\sqrt{2}\,\Phi-1\right)}{4\sqrt{2}}+\frac{\log\left(\Phi^2+\sqrt{2}\,\Phi+1\right)}{4\sqrt{2}}$$

A difficulty with using traditional notation, and therefore with **TraditionalForm**, for input is that it is *ambiguous*. For example, in traditional notation it is unclear if ***q(1-p)*** means a variable ***q*** times ***1-p*** or a function ***q*** evaluated at ***1-p*** (except by the context in which it is used); when using **TraditionalForm** for input, a *space* between the ***q*** and the parentheses denotes multiplication, and *no space* signifies that ***q*** is a function. In the following input in **TraditionalForm**, there is no space after the ***q*** in the first item, and a space after the ***q*** in the second item. (The output is in **StandardForm**.)

$$\{q(1-p),\ q\,(1-p)\}$$

```
{q[1 - p], (1 - p) q}
```

Mathematica understands very common abbreviations for functions (such as $\Gamma(z)$ for **Gamma[z]** and $\sin^{-1}(x)$ for **ArcSin[x]**), and allows parentheses to surround the arguments to a function.

(Note that the parentheses or brackets are required, so typing **sin x** to mean **sin (x)** is not permitted.)

$$\Gamma\left(\frac{1}{2}\right)-\sin^{-1}\left(\frac{\sqrt{3}}{2}\right)$$

$$\sqrt{\pi}-\frac{\pi}{3}$$

In general, *Mathematica* understands input in **TraditionalForm** if it uses only the most common notation and the above convention regarding parentheses, or involves simple editiong of previously generated **TraditionalForm** output. Here is an integral in **TraditionalForm** ; the expression $J_\nu(z)$ stands for the Bessel function of first kind, called **BesselJ**. The answer, also in **TraditionalForm**, contains typeset forms for the gamma function and a hypergeometric function.

$$\int z J_\nu(z)\,dz \quad \text{// TraditionalForm}$$

$$2^{-\nu-1} z^{\nu+2}\,\Gamma\left(\frac{\nu}{2}+1\right)\frac{{}_1F_2\left(\frac{\nu}{2}+1;\nu+1,\frac{\nu}{2}+2;-\frac{z^2}{4}\right)}{\Gamma(\nu+1)\,\Gamma\left(\frac{\nu}{2}+2\right)}$$

Three common *special characters* to be aware of in `StandardForm` and `TraditionalForm` are ***e***, ***i*** and ***d***. The characters *e* (entered as :ee: or `\[ExponentialE]`) and *i* (entered as :ii: or `\[ImaginaryI]`) stands for `E`, the base of the natural logarithm, and `I`, the imaginary unit; the keyboard letters *e* and *i* are uönderstood to be ordinary variable names. Similarly, the character *d* (entered as :dd: or `\[DifferentialD]`) is used in integrals, while the keyboard letter *d* is also an ordinary variable name.

The following integral uses the special characters ***e***, ***i*** and ***d***

$$\int e^{iz}\, dz$$

```
-I E^(I z)
```

The default input and output format types are set by pulling down the **Cell** menu and choosing the desired type from the **Default Input** (or **Output) FormatType** submenu.

3.2 Converting between Forms

To convert from one display form to another, select the cell containing the expression to be converted, then pull down the **Cell** menu and choose the desired form from the **Convert To** submenu. Note, that converting from one form to another requires the *Mathematica* kernel to be running.

For example, if one types `Limit[Sin[x^2] / x^2, x -> 0]` into an input cell, select the cell, choose **Convert To TraditionalForm**, the following is the result: *Mathematica* knows that `Limit` traditionally has a special two-dimensional form

$$\lim_{x \to 0} \frac{\sin(x^2)}{x^2}$$

One can superficially change from one form to another by pullilng down the Cell menu and choosing a form from the **Display As** submenu. Displaying `Limit[Sin[x^2]/ x^2, x->0]` as `TraditionalForm` displays the formula with a (proportional) font and italic variable names, but does *not* translate `Limit` into its traditional form.

Limit[Sin[*x*^2]/ *x*^2, *x* –> 0]

Display As \ TraditionalForm (as opposed to **Convert To \TraditionalForm**) is useful when formatting expressions that are not intended for use as input. Displaying an expression in a certain form does not require the *Mathematica* kerel to be running. Here is an expression in `StandardForm`

$$\Pi\ (F,\ G)\ \approx\ 1 - \Phi\left[\frac{c - \frac{1}{2}\, m\, m\, p_1}{\sqrt{Var\ (W_{XY})}}\right]$$

Here is the same expression after choosing **Display As \ TraditionalForm**

$$\Pi(F, G) \approx 1 - \Phi\left[\frac{c - \frac{1}{2} m\, m\, p_1}{\sqrt{\mathrm{Var}(W_{XY})}}\right]$$

Mathematica returns an error message if one tries to *convert* the previous expression to `TraditionalForm` (instead of *displaying* it as `TraditionalForm`) because the expression is not valid input.

$$\Pi\ (F,\ G) \approx 1 - \Phi\left[\frac{c - \frac{1}{2}\, m\, m\, p_1}{\sqrt{Var\ (W_{XY})}}\right]$$

$$\Pi\ (\underline{F,\ G}) \approx 1 - \Phi\left[\frac{c - \frac{1}{2}\, m\, m\, p_1}{\sqrt{Var\ (W_{XY})}}\right]$$

To *override the default format type* for output of a formula, wrap the formula with the name of the desired form.

```
(Sin[n]^2 + Binomial[n, m] - Gamma[m]) // TraditionalForm
```

$$\sin^2(n) + \binom{n}{m} - \Gamma(m)$$

To generate traditional notation for a formula *without evaluating* the formula, use **HoldForm**

```
(F[b] - F[a] = Limit[ Sum[f[x*i] Δx, {i, 1, n}], Δx → 0]) // HoldForm //
 TraditionalForm
```

$$F(b) - F(a) = \lim_{\Delta x\ 0} \sum_{i=1}^{n} f((x_i)^*)\,\Delta x$$

3.3 Special Characters and Forms in *Input*

Some *special characters* have built-in interpretations as *Mathematica* constants, operators, or functions, and those that do not there can be assigned one. For example, the symbol π is interpreted as the constant **Pi**, ad the symbols ***e*** and ***i*** represent the constants **E** and **I**. (Lower-case *pi* is the only Greek letter with a built-in interpretation.)

```
e^(i π)
```

```
-1
```

The character ϕ does not have a built-in interpretation, so it may be used as a variable name. Here, a value, **GoldenRatio**, is assigned to ϕ

```
ϕ = 1/2 (1 + √5);
```

The variable ϕ can now be used in calculations.

```
(ϕ^2 == ϕ + 1) // Simplify
```

```
True
```

Many functions can be represented by a special character, and in most cases the character has the full name **\[*FunctionName*]**. For example, the *logical function* **And** can be represented by the character ∧ (entered as **\[And]** or **:and:**)

```
(π > e) ∧ (4 > 3)
```

```
True
```

The *set function* **Intersection** can be entered using the character ⋂ by typing **\|Intersection|**

```
{a, b, c} ⋂ {b, c, d}
```

```
{b, c}
```

Other such functions are **Or** (∨), **Union** (⋃), **Sum** (∑), **Product** (∏), and **Integral** (∫)

It is important to keep in mind that many pairs of special characters look alike but have different meanings.

For example, \[Sum] (∑) looks similar to \[CapitalSigma] (Σ), but *Mathematica* interprets the former as the mathematical function **Sum** and the latter as the letter **Σ**.

Other pairs to be aware of are \[Product] (∏) and \[CapitalPi] (Π), \[Union] (⋃) and the keyboard capital letter U, and ordinary *multiplication* \[Times] (×) and the *vector cross product* \[Cross] (×).

In addition, many *Mathematica* operators have special forms that can be entered using keyboard aliases. One way to enter a *replacement rule* is to use the form **x→π**.

```
Cos[x] /. x → π
```

```
-1
```

Mathematica also understands the arrow character →, entered by typing :->: or \[Rule]. (Note that the character \[Rule] is different from the character \[RightArrow].

The *relational operators* ==, !=, >= and <= have corresponding keyboard aliases. But this is *not* the case for >~ , ~~, ~= or === etc..

```
{3 == π, 3 ≠ π, 3 ≥ π, 3 ≤ π , 3 ≳ π, 3 ≈ π, 3 ≃ π, 3 ≡ π}
```

```
{False , True , False , True , 3 ≳ π, 3 ≈ π, 3 ≃ π, 3 ≡ π}
```

Mathematica preserves *style information* for input on the screen, but ignores the styles when sending input to the kernel. Each **a** in the following input is identical for the purposes of calculation.

```
a + a + a + a
```

```
4 a
```

However, the same letter represented in *different styles* such as Script, Gothic, Double-Struck or Extended Latin is treated as different. Hence ordinary keyboard letters do not mean the same as *accented, script, Gothic*, or *double-struck* letters.

```
a + 𝒶 + 𝔞 + 𝕒
```

```
a + 𝒶 + 𝔞 + 𝕒
```

Because *Mathematica* normally *ignores style information* in input, one can *highlight* interesting parts of a formula without changing its meaning. In calculus teaching, for example, it may be desirable to highlight related parts of an integral when teaching integration by substitution.

```
∫2 x Cos[x^2] ⅆx
```

```
Sin[x^2]
```

3.4 Styled Text in *Output*

StyleForm generates output using styld text. **StyleForm** take as arguments the expression to change, along with a sequence of options describing the changes. Here is a symbol printed with blue text

```
StyleForm[Ω^6 + 1, FontColor → RGBColor[0, 0, 1]]
```

```
1 + Ω^6
```

The expression is valid *Mathematica* input, so one can apply mathematical functions to it. The kernel ignores the style changes, the result is therefore in the default output style

```
Factor[%]
```

```
(1 + Ω^2) (1 - Ω^2 + Ω^4)
```

Options that alter the textual characteristics of an expression typically start with the keyword **Font**. Some of the options that control common changes are **FontSize, FontSlant, FontWeight**, and **FontFamily**.

Here is the lower-case Greek alphabet using progressively larger font sizes, which is controlled by the option **FontSize**.

```
SequenceForm @@ Table[ StyleForm[FromCharacterCode[944 + n],
                          FontColor → Hue[n/25.],
                          FontSize → n + 10,
                                FontWeight → "Bold" ], {n, 25}]
```

αβγδεζηθικλμνξοπρςστυφχψω

where

```
? StyleForm
```

```
StyleForm[expr, options] prints using the specified style options.
  StyleForm[expr, "style"] prints using the specified cell style
  in the current notebook.
```

```
? SequenceForm
```

```
SequenceForm[expr1, expr2, ... ] prints as the textual concatenation
  of the printed forms of the expri.
```

```
SequenceForm @@ {"a", "b", "c"}
```

```
abc
```

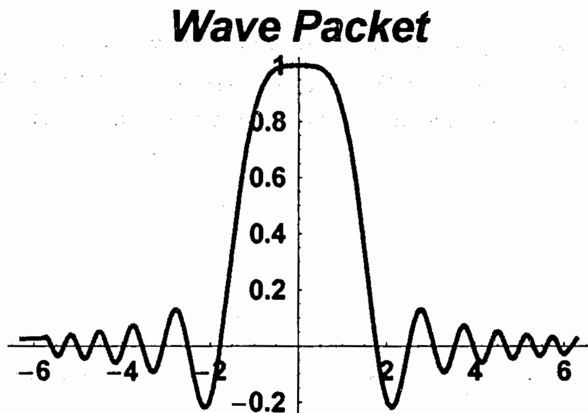
Typical values for **FontSlant** are "**Plain**", "**Italic**" and "**Oblique**". Here is an expression in *italic* type.

```
StyleForm["Slanted text", FontSlant → "Italic"]
```

Slanted text

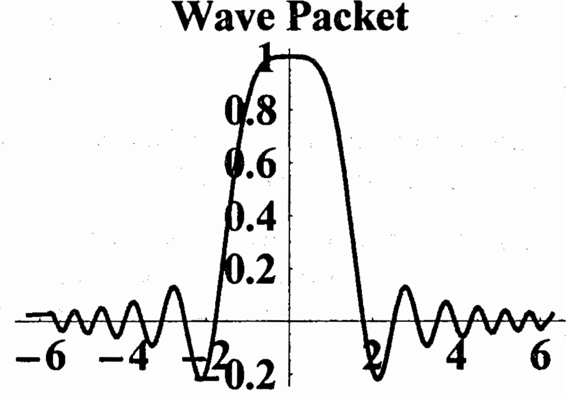
The style form can also be controlled in a graphic.

```
Plot[Sin[x^2]/x^2, {x, -2 π, 2 π},
    PlotLabel → StyleForm["Wave Packet", FontSize → 16,
                    FontWeight → "Bold", FontSlant → "Italic"]];
```



The graphics option **TextStyle** controls the default text styles used in a graph. The value assigned to **TextStyle** is a list of font options to apply to all the text in the graph.

```
Plot[Sin[x^2]/x^2, {x, -2 π, 2 π},
    PlotLabel → "Wave Packet",
    TextStyle → {FontSize → 16, FontFamily -> "Times", FontWeight → "Bold"}];
```



See also **StyleBox, StylePrint, CellPrint, $TextStyle, FormatType, Format, FontVariations**

4 Notebook Programming

4.1 Notebooks Are Expressions

In *Mathematica* 3.0 notebooks and cells are represented as *Mathematica* expressions, which means that they can be created and manipulated using the *Mathematica* programming language. To see the *expression form* of part of a notebook, one selects one or more cells in the notebook, and chooses **Show Expression** (resp. SHIFT CTRL E) from the **Format** menu.

4.2 Cells

A cell in a notebook is represented by the expression **Cell[*contents*, "*cellstyle*"]**. Common values for cellstyle are **Title, Section, Text, Input,** and **Output.**

For example, here is the expression representing a text cell containing the word "Hello".

```
Cell["Hello", "Text"]
```

```
Cell [Hello , Text ]
```

The function `StylePrint` creates a new cell of a given style, and prints a given expression into it. Here a text cell is created

```
StylePrint["A new cell of type Definition", "Definition"]
```

A new cell of type Definition

StylePrint and all other notebook-manipulation commands work inside programming functions. Therefore, a **Do** loop can be used to create a sequence of several small-text numbered cells.

```
Do[ StylePrint[ "Cell number " <> ToString[n], "SmallText",
                                        FontSize -> 10 + 2 n,
                                        TextAlignment -> Center],
   {n, 5}]
```

Cell number 1

Cell number 2

Cell number 3

Cell number 4

Cell number 5

`CellPrint` prints a given cell expression into the current notebook.

```
CellPrint[ Cell["This is a new cell", "SmallText",
                                      CellFrame -> True ]]
```

This is a new cell

Cell objects accept *options*, just as other *Mathematica* functions do.

```
CellPrint[ Cell["Appendix A", "Text",
                             Background -> GrayLevel[0.85],
                             CellFrame -> True,
                             FontWeight -> "Bold", FontSize -> 18,
                             TextAlignment -> Center] ]
```

Appendix A

To learn the types of options a cell can take, select a cell an change some of its options using the commands under the **Format** menu or the **Option Inspector,** and then show the expression form of the cell using the **Show Expression** menu command. Here is a cell with several options changed

■ **Appendix B.1 : Tables of formulas**

Here is the expression form of the cell.

```
Cell["Appendix  B.1  :  Tables  of  formulas",  "Text",
  CellFrame->3,
  CellDingbat->"\[FilledSquare]",
  CellMargins->{{18.375,  Inherited},  {Inherited,  Inherited}},
  FontWeight->"Bold",
  Background->GrayLevel[1]]
```

The expression form of a cell can be edited directly, and any changes made are reflected when the cell is converted back into its display form. If a cell contains styled text, the separate strings of text are part of a list inside a `TextData` object, as in the following example

```
CellPrint[ Cell[ TextData[{ "Here is some ",
                          StyleForm["styled  ", FontColor -> Hue[0],
                                         FontSlant -> "Italic"],
                          "text." }],
               "Text", CellFrame -> 3, TextAlignment -> "Center" ]]
```

Here is some *styled* text.

Because a cell or cell group is a *Mathematica* expression, hence *Mathematica* programs can be used to generate cells automatically. The function `CellPrint` in a `Do`-loop generates a sequence of cells as output.

```
Do[CellPrint[Cell["This is a cell", "Text",
        CellFrame -> 3, CellFrameColor -> Hue[i/20.],
        FontSize -> i, FontColor -> Hue[(i - 10)/20.]]], {i, 10, 20, 2}]
```

This is a cell

This is a cell

This is a cell

This is a cell

This is a cell

This is a cell

In this example, StyleBox objects are used to produce colored text. The size, color, and background of each character is individually set according to the increment *i*.

```
n = 71.;
CellPrint[Cell[TextData[
    Table[StyleBox[ToString[i], FontColor → Hue[i/n],
                Background → Hue[Mod[.5 + i/n, 1]],
                FontSize → i/2.], {i, 20, n}]], "Output"]]
```

2021222324252627282930313233343536373839404142434445464748495051525354555657585960616263646566676869707071

Groups of cells are represented by a list of cells inside a **CellGroupData** object. If a notebook has automatic grouping turned on, *Mathematica* automatically wraps **CellGroupData** objects around groups of cells; if not, *Mathematica* groups cells according to the groups manually set.

4.3 Notebooks

■ 4.3.1 Notebooks as Expressions

A *Mathematica* notebook is represented by a list of cells inside a **Notebook** object. Here is a notebook consisting of three cells.

```
nb = Notebook[{ Cell["Notebooks", "Title"],
                Cell["Creating notebooks", "Section"],
                Cell["To create a new notebook, ...", "Text"] }]
```

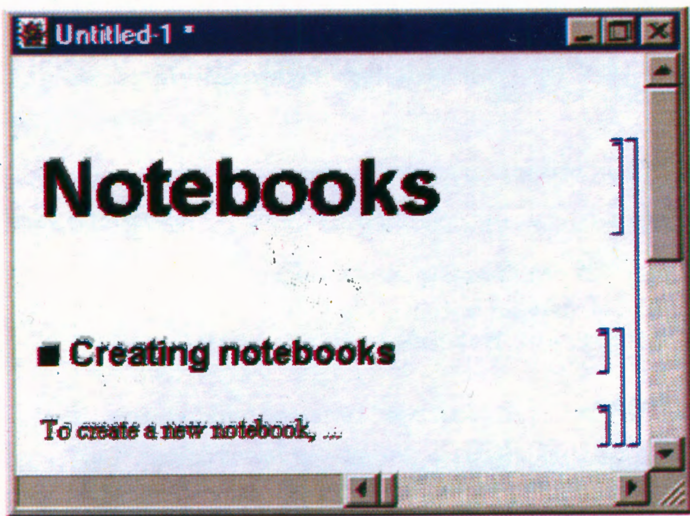
```
Notebook[{Cell[Notebooks, Title], Cell[Creating notebooks, Section],
  Cell[To create a new notebook, ..., Text]}]
```

However, *Mathematica* does not automatically convert a **Notebook** expression into a displayed notebook. To create a new notebook from a **Notebook** expression, use **NotebookPut**.

```
newNb = NotebookPut[ nb ]
```

```
- NotebookObject -
```

The notebook appears in a *separate* window



SetOptions allows one to change the *option settings* for a notebook expression, in this case the notebook background is colored to yellow.

```
SetOptions[ newNb, Background → RGBColor[1, 1, 0] ];
```

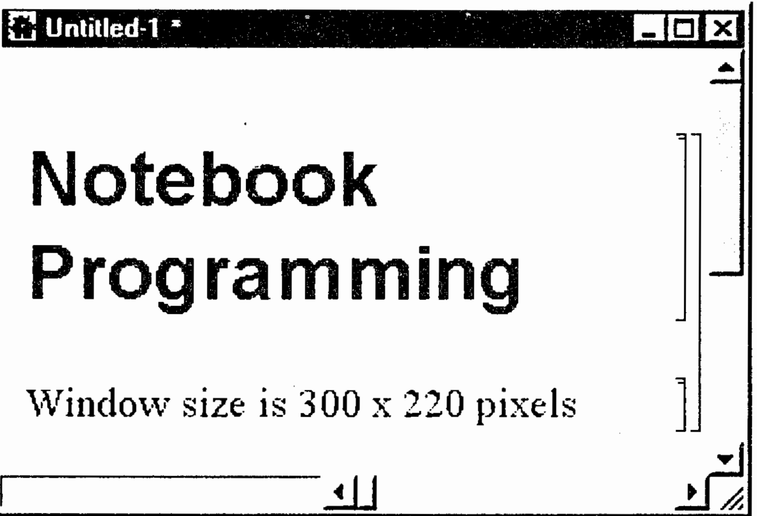
NotebookPut can be used the same way as **CellPrint** above. The command **Notebook** creates a new window which therefore represents a complete notebook document. For **Notebook** expressions one may use the same options

as for cells, and *additional* options that are defined only at the `Notebook` level. For example, `WindowSize` specifies the extension of the window.

The subsequent input

```
NotebookPut[ Notebook[{ Cell["Notebook Programming", "Title"],
                        Cell["Window size is 300 x 220 pixels", "Text"] },
                        FontSize -> 20,
                        FontColor -> RGBColor[0, 0, 1],
                        WindowSize -> {350, 220}] ];
```

creates a window of size 350×220 pixels containing blue text.



The command `NotebookGet` *reads an open notebook* as an expression to be manipulated by the *Mathematica* kernel.

The command `NotebookCreate` *creates* an empty notebook.

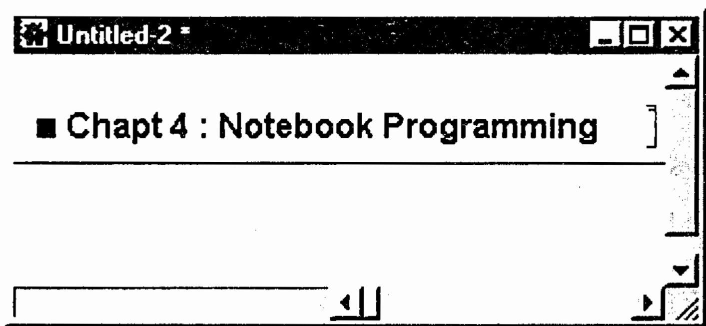
```
newerNb = NotebookCreate[]
```

```
- NotebookObject -
```

To *place cells* into the empty notebook just created, use `NotebookWrite`

```
NotebookWrite[newerNb, Cell["Chapt 4 : Notebook Programming", "Section"]]
```

The section cell appears at the top of the new notebook.



4.4 Controlling Notebooks with the Kernel

There are several ways to refer to a notebook using kernel functions. `EvaluationNotebook[]` refers to the notebook in which the command **`EvaluationNotebook[]`** is being evaluated. To *change the background color* of the current evaluation notebook, type the following

```
SetOptions[EvaluationNotebook[], Background → GrayLevel[.8] ];
```

`SelectedNotebook` refers to the currently selected notebook (usually indicated by having a highlighted title bar), and `InputNotebook` refers to the notebook in which keyboard input will appear. In many cases the two functions refer to the same notebook; one exception is when a *palette* is the selected notebook. Ordinary, palettes do not receive typed input, so a palette may be the selected notebook while another non-palette notebook is the input notebook.

Many features of the front end, such as opening, closing, saving, and printing notebooks, can be duplicated using kernel functions. Here, another new notebook is created.

```
anotherNb = NotebookPut[Notebook[{Cell["Another Notebook", "Subtitle"]}] ]
```

```
- NotebookObject -
```

`NotebookSave` *saves* the new notebook under the given *file name.*

```
NotebookSave[anotherNb, "New.nb"]
```

With the command `NotebookClose` the new notebook is closed.

```
NotebookClose[anotherNb]
```

`NotebookOpen["name"]` *opens* the notebook with file name "**name**" from the hard disk (assuming the file is in the current directory, or one specifies the full pathname of the file) into a new window, and `NotebookPrint` *prints* a notebook to the currently selected printer.

Furthermore, *Mathematica* maintains information about the *current selection in a notebook* in the object `NotebookSelection`.

One can *read the contents of the current selection* using `NotebookRead`. One may change the options of the current selection using **`SetOptions`**.

Moreover, on can locate contents of a notebook using the commands `NotebookLocate` and `NotebookFind`. The command **`NotebookLocate["tag"]`** sets the current selection to be the first cell in the selected notebook that has the cell tag *`tag`*.

For example, evaluating **`NotebookLocate["Hyperlink1"]`** scrolls the current notebook to the first cell that has the cell tag "*Hyperlink1*" (which actually is in Chapt. 1.1.6 of this notebook)

```
NotebookLocate["Hyperlink1"]
```

A *tag* may be assigned to a cell by selecting a particular cell, and choosing from the **Find** menu **Add/Remove_Cell_Tags**, or by changing the value of the cell option **`CellTags`** in the expression form of a cell. Cell tags are generally not displayed on the screen unless **Show_Cell_Tags** from the **Find** menu is chosen.

Hyperlinks inside notebooks are typically implemented using **NotebookLocate** . One can also specifiy a notebook to search other than the currently selected notebook.

NotebookFind[*notebook,data*] sets the current selection to be the next occurence of ***data*** in the notebook object ***notebook***.

For example, **NotebookFind[SelectedNotebook[],"math"]** generally scrolls the currently selected notebook to the next occurence of the string **math**. Additonally, one can specify what part of a notebook to search by specifying elements such as **CellContents**, **CellStyle**, or **CellTags**.

The following command selects the next graphics cell in the selected notebook. The result is a **NotebookSelection** object.

```
NotebookFind[SelectedNotebook[], "Graphics", Next, CellStyle]
```

```
NotebookSelection  [- NotebookObject  -]
```

The selection jumps to Chapt. 3.4 in the current notebook.

Similarly, the following command *selects all output cells* in the selected notebook, and then *deletes* the cells using **NotebookDelete**.

```
With[ {selectedNb = SelectedNotebook[]},
     NotebookFind[selectedNb, "Output", All, CellStyle];
     NotebookDelete[selectedNotebook]
   ]
```

The current selection of a notebook can be moved, evaluated, copied and so forth, using notebook-manipulation commands such as SelectionMove, SelectionEvaluate and NotebookRead.

See also **GraphicsData, SetSelectedNotebook, SelectionMove, SelectionEvaluate, Notebooks, ButtonNotebook, Options, FullOptions**

```
? SetSelectedNotebook
```

```
SetSelectedNotebook [notebook ] makes the specified notebook be the
  currently selected one in the front end.
```

```
? SelectionMove
```

```
SelectionMove [obj, dir, unit] moves the current selection in an open
  notebook in the front end in the direction dir by the specified
  unit. SelectionMove [obj, dir, unit, n] repeats the move n times.
```

```
? SelectionEvaluate
```

```
SelectionEvaluate [notebook ] replaces the current selection in
  a notebook with the result obtained by evaluating the contents
  of the selection in the kernel. SelectionEvaluate [notebook, sel]
  sets the current selection after the evaluation to be as specified by sel.
```

```
? Notebooks
```

```
Notebooks [ ] gives a list of notebooks currently open in the front end.
```

?ButtonNotebook

ButtonNotebook [ ] gives the notebook , if any, that contains the button which initiated the current evaluation .

?Options

Options [symbol] gives the list of default options assigned to a symbol . Options [expr] gives the options explicitly specified in a particular expression such as a graphics object . Options [stream] or Options ["sname "] gives options associated with a particular stream . Options [object] gives options associated with an external object such as a NotebookObject . Options [obj, name] gives the setting for the option name . Options [obj, {name1 , name2 , ... }] gives a list of the settings for the options namei .

?FullOptions

FullOptions [expr] gives the full settings of options explicitly specified in an expression such as a graphics object . FullOptions [expr, name] gives the full setting for the option name . FullOptions [expr, {name1 , name2 , ... }] gives a list of the full settings for the options namei . FullOptions [object] gives the full settings for options associated with an external object such as a NotebookObject .

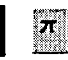
5 Creating Buttons and Palettes

5.1 Predefined Typs of Buttons

■ 5.1.1 Simple Buttons for Textual Substitution

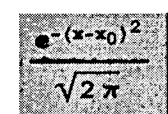
In order to create a *button* pull down the **Input menu** and choose the type of button wanted from the **Create Button** submenu, e.g. **Custom...** . *Mathematica* places the button chosen with an empty placeholder [□] in the notebook at the current cell insertion point, e.g. here

Then, one may type some contents for the button, such as the single character π



The new button is by default *inactive* hence clicking has no effect. To make the button *active* select the cell containing the button, pull down the **Cell menu** and choose in the **Cell Properties** submenu the item **Cell Active** . The tiny "A" next to the cell bracket indicates the cell is active. Then, clicking the acitive button will paste the character π at the current cell insertion point.

Hence buttons can be used to paste arbitrarily complicated expressions into a notebook. Note, that the text style displayed in a button has *no* effect on the input pasted into the notebook, thus font size or color of the expression displayed in a button can be changed.



In order to make a button appear in a *separate (movable) window* select the button (or the cell containing the button), then pull down the **File** menu and choose the menu item **Generate_Palette_from_ Selection**. The palette window can be moved by "dragging" the pallette by its title bar, and can be closed by clicking the window's close box [×] .

Note that the contents of a button replace the current text selection if any $\frac{e^{-(x-x_0)^2}}{\sqrt{2\pi}}$. Here a selected text has been replaced by the content of the button with the Gaussian function.

■ 5.1.2 Buttons with Wrap around

To create a button whose contents are pasted around the current selection use the notion of placeholders. E.g. , to create a button that allows to wrap the function `Factor` around the current selection, type the expression **Factor[■]** in the button. The symbol ■ is called a *selection placeholder* and it can be typed either by \ **[SelectionPlaceholder]** or its alias ⋮**spl**⋮ .

(Note that the selection placeholder character is different from the character \ **[FilledSquare]** .)

Here is a button created by choosing **Custom...** from the **Create Button** submenu of the **Input** menu, then typing **Factor[\ [SelectionPlaceholder]]** (or the shortcut **Factor[:spl:]**) into the button.

```
Factor[■]
```

```
Cell[BoxData[
    ButtonBox[
      RowBox[{"Factor",   "[",  "\[SelectionPlaceholder]",    "]"}],
      ButtonStyle->None]],   "Input",
  Active->True]
```

Then the button has to be made *active*, either by making the cell containing it active or generating a (floating) palette from the button. Next one types the expression to be factored into a new input cell and select it. Clicking the button pastes the contents of the button into the input cell, replacing the selection placeholder with the expression to befactored. Typing SHIFT RET *evaluates* the expression

```
Factor[x^4 - 10 x^3 + 35 x^2 - 50 x + 24]
```

```
(-4 + x) (-3 + x) (-2 + x) (-1 + x)
```

Creating a button by changing **ButtonStyle→None** to **ButtonStyle→"Evaluate"** (which has to be done by hand in the unformatted cell because the **CreateButton** menu does *not* offer these options) pastes its contents around the current selection and then evaluates it in place, replacing the current selection with the result.

```
Factor[■ ]      (* ButtonStyle->"Evaluate" *)
```

```
Cell[BoxData[
    RowBox[{
      ButtonBox[
        RowBox[{"Factor",   "[",  "\[SelectionPlaceholder]",    " ",  "]"}],
        ButtonStyle->"Evaluate"],    "               ",
      RowBox[{"(*",   " ",
        RowBox[{"ButtonStyle",    "->",  "\"\<Evaluate\>\""}],    " ",  "*)"}]}]]
  "Input",
  Active->True]
```

Here is a fourth order polynomial $x^4 - 10x^3 + 35x^2 - 50x + 24$ in order to test the button.

```
x^4 - 10 x^3 + 35 x^2 - 50 x + 24          (* Use the Button with Factor[■] *)
```

The buttons in the standard AlgebraicManipulation palette use the **Evaluate** style.

(Note, that when using an **Evaluate** button be aware that % refers to the result of the subexpression evaluated, and not necessarily to the entire contents of the cell.)

Two other predefined styles of buttons are **ButtonStyle→**"EvaluateCell" and **ButtonStyle→**"CopyEvaluate". The former pastes the contents of the button around the current selection, and evaluates the *entire cell* containing the expression. The latter pastes the button contents around a *copy* of the selected input, then evaluates it in place.

To enter a button that pastes a function with *more than one argument* use the *selection placeholder* ■ to represent the current selection, then use *placeholders* to represent slots for additional arguments of the function. A placeholder □ is entered by typing either □ or its alias :pl: . (Note that the placeholder is different from the empty square character □ .)

For example, the function **Solve** takes *two* arguments, (i) the equation to be solved, and (ii) the variable(s) for which the solution has to be solved. To enter a button that pasted **Solve** around the current selection and leaves space for the second argument, type **Solve[■ , □]** into a button.

```
Solve[■, □]
```

After making the botton "Solve" *active* it can be used to solve the following fourth order equation
$x^4 - 10x^3 + 35x^2 - 50x + 24 == 0$

```
x^4 - 10 x^3 + 35 x^2 - 50 x + 24 == 0
```

```
Solve[x^4 - 10 x^3 + 35 x^2 - 50 x + 24 == 0 ,  ]
```

The selected equation appears in place of the selection placeholder ■, and the text insertion point moves to the next placeholder □ , where the variable x is typed in.

```
Solve[x^4 - 10 x^3 + 35 x^2 - 50 x + 24 == 0 , x]
```

```
{{x → 1}, {x → 2}, {x → 3}, {x → 4}}
```

Then press [SHIFT][RET] for evaluation.

Differentiation button

Another suitable example is *differentiation*; the function D takes two arguments, a function to be differentiated and a variable of differentiation. To enter a button that pastes D around the current selection an leaves space for the second argument, type the expression D[■,□] into a button.

```
D[■, □]
```

After making the button *active*, one may use it by typing an expression (e.g. f[x] g[x]) to be differentiated into a new input cell and selecting it.

When the button containing D[■,□] is pressed, the following is the result

```
D[f[x] g[x], □]
```

The selected expression `f[x] g[x]` appears in place of the selection placeholder, and the text insertion point moves to the placeholder, where one types the variable of differentiation, then press SHIFT+RET to evaluate the input.

```
D[f[x] g[x], x]
```

```
g[x] f'[x] + f[x] g'[x]
```

Series button

A button can contain only *one selection placeholder*, but as *many* regular *placeholders* as desired. Here is a button containing several placeholders

i^2

When the button is clicked, the current selection replace the selection placeholder ■ in the button's contents, and one moves from one placeholder to the next by pressing the TAB key or using the mouse pointer.

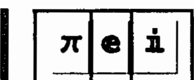
5.2 Creating Palettes

A *palette* is simply a notebook that contains an array of buttons, and has special settings that dictate its properties and appearance on the screen.

To create a new palette, choose **Create_Table/Matrix/Palette** from the **Input** menu. In the dialog box that appears, enter the number of rows and columns the palette should contain, and check the boxes indicating whether to draw lines separating the rows and columns, and whether to draw a frame around the palette.

The following palette has one row and three columns of buttons. By default, each button contains an *empty* placeholder.

To define symbols to be pasted when the button is pressed, type a symbol into each placeholder. Here the placeholders are filled with ***π***, ***e*** and ***i*** . Then activate the cell containing the palette.



To put the palette into a *separate window*, select the entire palette with the mouse, and then pull down the File menu and choose **Generate_Palette_from_Selection**. A copy of the palette will appear in a *separate window*. When one of the (activated) buttons in the palette is clicked, the contents of the button will appear in the active notebook at the current text insertion point.

To *convert a palette into an editable notebook* by selecting the palette window (by clicking on its title bar or selecting its name from the **Window** menu) and choose **Generate_Notebook_from_Palette** from the **File** menu.

This was done for the PageBreakButton palette which was converted to an ordinary notebook in order to study the underlying code in detail by unformatting the button cell `PageBreakButtonNotebook.nb`

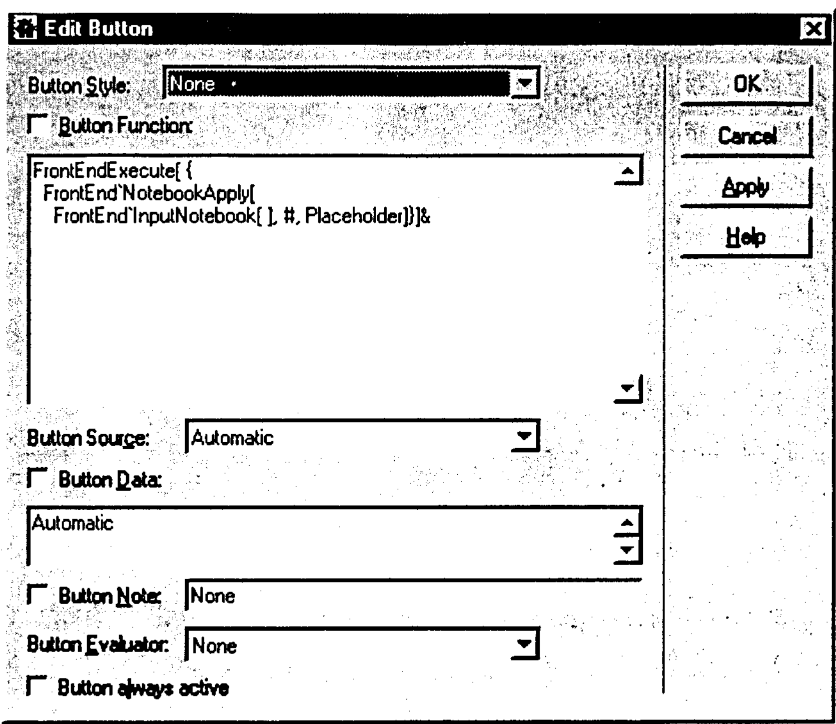
```
openNotebookButton["PageBreakButtonNotebook.nb", 1, subDir]
```

5.3 Custom Button Actions

A button can perform any action that can be programmed with *Mathematica*'s programming language.

The internal expression form of a button is `ButtonBox[contents,options]`. The options to **ButtonBox** control (i) the action taken when the button is clicked, (ii) the appearance of the button, and (iii) any additional data associated with the button.

The option that changes the function which a button performs, is `ButtonFunction`, which accepts a function to evaluate when the button is pressed. To create a button with a custom function, pull down the **Input** menu and choose **Custom...** from the **Create Button** submenu, .



and then type the desired **ButtonFunction** in the button function field in the dialog box.

Whe one a create a *button* that uses *kernel functions*, one must change the value of the **ButtonStyle** to **None** and change the value of **ButtonEvaluator** to the name of an available kernel (such as **"Local"**) or **Automatic**. The setting for **ButtonFunction** can be changed by directly editing the expression form of the button, or selecting the button and then choosing **Edit_Button** from the **Input** menu.

Button "*Load package DiracDelta*"

Here is the expression form of a button that *loads the package* **Calculus`DiracDelta`**.

```
Cell[BoxData[
           ButtonBox["Load  package  DiracDelta",
           ButtonFunction:>Needs["Calculus`DiracDelta`"],
           ButtonEvaluator->"Local",
           ButtonStyle->None
           ]], "Input"]
```

Load package DiracDelta

Button "*red*"

Here is the expression form of an active button that *changes the currently selected text* in the input notebook *to red.*

```
Cell[BoxData[
      ButtonBox["red",
          ButtonFunction:>  SetOptions[NotebookSelection[InputNotebook[]],
                                            FontColor->  Hue[0]],
          ButtonEvaluator->Automatic,
          ButtonStyle->None,
          Active->True
          ]], "Input"]
```

red

When the button is clicked, any currently selected text becomes red.

The function used in **ButtonFunction** needs to specify where any *output* for the function should be placed. (The button functions defined above did *not return* a value.)

Button "*Create random number*"

For example, the button function setting **ButtonFunction:>Random[]** will not display any output unless one creates an output cell using a command such as **NotebookWrite**.

Here is the expression form of a button that creates a new output cell containing a random number in the currently selected notebook, using StylePrint.

```
Cell[BoxData[
       ButtonBox[
              RowBox[{"Create",   " ",  "random",   " ","number"}],
              ButtonFunction:>StylePrint[   Random[],  "Output"],
              ButtonEvaluator->"Local",
              ButtonStyle->None,
              Active->True]],   "Input"]
```

Create random number

```
0.718919
```

Note, that button definitions must be created using a subsetz of standard *Mathematica* syntax conventions. As a general rule, button definitions can be made using the **FullForm** representation of expressions, as well as the operators -> and :> , lists in { } form, and pure functions using # and & .

For example, if one wants to use a compound expression as a value to **ButtonFunction**, one cannot use the input form $\mathbf{expr_1}$; $\mathbf{expr_2}$; instead one must use the internally used form **CompoundExpression[**$\mathbf{expr_1}$, $\mathbf{expr_2}$**]**, and to assign a value to a variable use the full form **Set** instead of its abbreviation " = ".

Button "*Insert heading*"

Here is the expression form of an active button that inserts a title cell at the begining of the input notebook.

```
Cell[BoxData[
      ButtonBox[
         RowBox[{"Insert",   " ", "heading"}],
         ButtonFunction:>With[{Set[iNb,    InputNotebook[]  ]},
                                CompoundExpression[
                                    SelectionMove[iNb,   Before,  Notebook],
                                    NotebookWrite[iNb,   Cell[TextData[{
                                           "Notebook  by  [",
                                           StyleBox["R.Kragler",
                                           FontSlant->"Italic"],    "]" }],
                                           "Example",
                                           Background->Hue[0.15],
                                           TextAlignment->Center,
                                           CellFrame->2
                                                  ]]
                                ]
                  ],ButtonEvaluator->Automatic,
                    ButtonStyle->None   ]], "Input",
         Active->True  ]
```

Insert heading

A button's options can be changed using items under the **Button Options** category of the Option Inspector. More sophisticated button actions can be programmed by changing the values of the options **ButtonData** and **ButtonSource**. Examples of more sophisticated buttons and palettes are available in the **Help Browser,** under **Getting Started/Demos** category.

See also **ButtonData, ButtonSource, FrontEndTokenExecute, FrontEndToken, FrontEndExecute, ButtonFrame, ButtonNote**

?ButtonData

```
ButtonData  is  an  option  for  ButtonBox  which  specifies  the  second
  argument  to  give  to  the  ButtonFunction  for  the  button  when  the
  button  is  active  and  is  clicked  on.
```

?ButtonSource

```
ButtonSource  is  an  option  for  ButtonBox  which  specifies  the  first
  argument  to  give  to  the  ButtonFunction  for  the  button  when  the
  button  is  active  and  is  clicked  on.
```

?FrontEndExecute

```
FrontEndExecute [expr] sends  expr  to  be  executed  by  the  Mathematica  front  end.
```

?ButtonFrame

```
ButtonFrame  is  an  option  for  ButtonBox  which  specifies  the  type  of
  frame  to  display  around  a  button.
```

?ButtonNote

```
ButtonNote  is  an  option  for  ButtonBox  which  specifies  what  should  be
  displayed  in  the  status  line  of  the  current  notebook  window  when
  the  button  is  active  and  the  cursor  is  placed  on  top  of  it.
```

CONTENTS

Survey of *Mathematica* Features: Numerics, Symbolic Computation and Graphics

Abstract 3
Some general remarks on CAS 4
- Motivation 4
- Reasons using CAS 4
- Possible Application of CAS 5

Why just *Mathematica*? 6
- Features of *Mathematica* 6
- *Mathematica* Notebooks for Exchange of Scientific Information 6
- *Mathematica's* Capabilities in Technical Computing 7
- Symbolic Calculator 9
- Instrument for Extensive Numerical Calculations 10
- System for Image Data Processing 12
- Visualization 14
- Real Time Animation *(Dynamic Vizualizer or Conix 3DExplorer)* 16
- High-Level Programming Language 16
- Expert System 17

Examples of *Mathematica* Notebooks 20
Outlook on the Development of CAS 22
- Four Color Theorem 22
- Fraktals and Iterated Function-Systems (IFS) 22
- Cellular Automata 23

Mathematica Performance 25
Mandelbrot and Julia Sets 29
- Visualization of Complex Functions 29
- Iterative Maps 30
- Julia Sets 30
- Mandelbrot Set 32

MathLink Applications 34
- *Mathematica* as a Software Component 34

Mathematica Programming

Abstract 39
General Features of *Mathematica* 39
- *Mathematica* Notebooks: Interactive Documents 40
- *Mathematica* Programming Language 41

Programming Demo 43
- Unifying Concept of the *Mathematica* Programming Language 43
- *Mathematica* Programming Styles 47

Writing Programs in *Mathematica* 51
Programming Efficiency 56
Create a Sequence of 0's and 1's 56
List all Elements larger than all Preceding Elements 57
Encode Sequences in Lists (Run-length Encoding) 58
Programming Styles 60
General Remarks on the *Mathematica* Programming Language 60
Different *Mathematica* Notations 61
Overview of *Mathematica* Programming Styles 66
Factorial in Procedural Programming Style 66
Factorial in Recursive Programming Style 66
Factorial in Functional Programming Style 66
Factorial in Rule-based Programming Style 67
Programming Styles in Detail 68
Procedural Programming 68
Recursive Programming 70
Functional Programming 72
Concept of pure (or anonymous) functions 74
Rule-based Programming 75
Programming with Binding Propagation 80
Logic Programming 82
Abstract Data Types 84
Object Oriented Programming 92
Modularization 96
Mathematica as Developing Tool:
from an Interactive Evaluation to a Package 98
Programming Literature 105

Mathematica Front End

Abstract 107
Interactive Documents and Front End Features 108
Mathematica Notebooks 109
Palettes and Buttons 111
Mathematical Notation 113
Front End Features of *Mathematica* v3.0 115
Using the *Mathematica* 3.0 Front End 116
Introduction to Notebooks 116
Notebooks 116
Cells 116
Text Styles 118
Style Sheets 118
Style Environments 119
Hyperlinks 120
Numbered Equations and Figures 121

Converting Notebooks 122
Cells and Notebook Options 123
Setting Options 123
Global Options 125
Notebook Options 125
Cell Options.. 126
Special Characters and Forms 127
Introduction 127
Using Palettes 127
Character Full Names 129
Menus 130
Keyboard Aliases and Shortcuts 131
Extensible Characters 134
Typeset Expressions within Text (*InlineInput)*.. 136
Examples.. 137
Doing calculations 139
Different Input and Output Forms 139
Converting between Forms 141
Special Characters and Forms in *Input* 142
Styled Text in *Output* 144
Notebook Programming 146
Notebooks Are Expressions 146
Cells 146
Notebooks 149
Notebooks as Expressions 149
Controlling Notebooks with the Kernel 151
Creating Buttons and Palettes 154
Predefined Types of Buttons 154
Simple Buttons for Textual Substitution 154
Buttons with Wrap around 154
Creating Palettes 157
Custom Button Actions 158