

Introduction to Mathematica

Thomas Hahn

Max-Planck-Institut für Physik
München



Computer Algebra Systems

- **Commercial systems:** Mathematica, Maple, Matlab/MuPAD, Macsyma, MathCad, Fermat, Derive ...
- **Free systems:** FORM, GiNaC, Axiom, Cadabra, GAP, Maxima, SAGE, Singular, Reduce, MAGMA ...
- **Generic systems:** Mathematica, Maple, Matlab/MuPAD, Macsyma/Maxima, MathCad, Reduce, SAGE, Axiom, MAGMA, GiNaC ...
- **Specialized systems:** Cadabra, Singular, Magma, CoCoA, GAP ...
- **Many more...**



Expert Systems


In technical terms, Mathematica is an **Expert System**.
Knowledge is added in form of **Transformation Rules**.
An expression is transformed until no more rules apply.

Example:

```
myAbs[x_] := x /; NonNegative[x]  
myAbs[x_] := -x /; Negative[x]
```

We get:

myAbs[3]  3

myAbs[-5]  5

myAbs[2 + 3 I]  myAbs[2 + 3 I]

– no rule for complex arguments so far

myAbs[x]  myAbs[x]

– no match either



Immediate and Delayed Assignment

Transformations can either be

- added “permanently” in form of Definitions,

```
norm[vec_] := Sqrt[vec . vec]
```

```
norm[{1, 0, 2}]  Sqrt[5]
```

- applied once using Rules:

```
a + b + c /. a -> 2 c  b + 3 c
```

Transformations can be **Immediate** or **Delayed**. Consider:

```
{r, r} /. r -> Random[]  {0.823919, 0.823919}
```

```
{r, r} /. r :=> Random[]  {0.356028, 0.100983}
```

Mathematica is one of those programs, like $\text{T}_{\text{E}}\text{X}$, where you wish you'd gotten a US keyboard for all those braces and brackets.



Almost everything is a List

All Mathematica objects are either **Atomic**, e.g.

`Head[133]`  `Integer`

`Head[a]`  `Symbol`

or (generalized) **Lists** with a **Head** and **Elements**:

`expr = a + b`

`FullForm[expr]`  `Plus[a, b]`

`Head[expr]`  `Plus`

`expr[[0]]`  `Plus` — same as `Head[expr]`

`expr[[1]]`  `a`

`expr[[2]]`  `b`



List-oriented Programming

Using Mathematica's list-oriented commands is almost always of advantage in both speed and elegance.

Consider:

```
array = Table[Random[], {10^7}];
```

```
test1 := Block[ {sum = 0},  
  Do[ sum += array[[i]], {i, Length[array]} ];  
  sum ]
```

```
test2 := Apply[Plus, array]
```

Here are the timings:

```
Timing[test1][[1]]  31.63 Second
```


```
Timing[test2][[1]]  3.04 Second
```



Map, Apply, and Pure Functions

Map applies a function to all elements of a list:

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

`f /@ {a, b, c}`  `{f[a], f[b], f[c]}` – short form

Apply exchanges the head of a list:

`Apply[Plus, {a, b, c}]`  `a + b + c`

`Plus @@ {a, b, c}`  `a + b + c` – short form

Pure Functions are a concept from formal logic. A pure function is defined ‘on the fly’:

`(# + 1)& /@ {4, 8}`  `{5, 9}`

The `#` (same as `#1`) represents the first argument, and the `&` defines everything to its left as the pure function.



List Operations

Flatten removes all sub-lists:

`Flatten[f[x, f[y], f[f[z]]]]` \rightarrow `f[x, y, z]`

Sort and **Union** sort a list. **Union** also removes duplicates:

`Sort[{3, 10, 1, 8}]` \rightarrow `{1, 3, 8, 10}`

`Union[{c, c, a, b, a}]` \rightarrow `{a, b, c}`

Prepend and **Append** add elements at the front or back:

`Prepend[r[a, b], c]` \rightarrow `r[c, a, b]`

`Append[r[a, b], c]` \rightarrow `r[a, b, c]`

Insert and **Delete** insert and delete elements:

`Insert[h[a, b, c], x, {2}]` \rightarrow `h[a, x, b, c]`

`Delete[h[a, b, c], {2}]` \rightarrow `h[a, c]`



Patterns

One of the most useful features is **Pattern Matching**:


- `_` – matches one object
- `__` – matches one or more objects
- `---` – matches zero or more objects
- `x_` – named pattern (for use on the r.h.s.)
- `x_h` – pattern with head `h`
- `x_:1` – default value
- `x_?NumberQ` – conditional pattern
- `x_ /; x > 0` – conditional pattern

Patterns take function overloading to the limit, i.e. functions behave differently depending on *details* of their arguments:

```
Attributes[Pair] = {Orderless}
Pair[p_Plus, j_] := Pair[#, j]& /@ p
Pair[n_?NumberQ i_, j_] := n Pair[i, j]
```

Attributes

Attributes characterize a function's behaviour before and while it is subjected to pattern matching. For example,

```
Attributes[f] = {Listable}
f[l_List] := g[l]
f[{1, 2}]  {f[1], f[2]} — definition is never seen
```

Important attributes: Flat, Orderless, Listable,
HoldAll, HoldFirst, HoldRest.

The Hold... attributes are needed to pass variables by reference:

```
Attributes[listadd] = {HoldFirst}
listadd[x_, other_] := x = Flatten[{x, other}]
```

This would not work if x were expanded before invoking listadd, i.e. passed by value.



Memorizing Values

For longer computations, it may be desirable to 'remember' values once computed. For example:

```
fib[1] = fib[2] = 1
```

```
fib[i_] := fib[i] = fib[i - 2] + fib[i - 1]
```

```
fib[4]  3
```

```
?fib  Global`fib
```

```
fib[1] = 1
```

```
fib[2] = 1
```

```
fib[3] = 2
```

```
fib[4] = 3
```

```
fib[i_] := fib[i] = fib[i - 2] + fib[i - 1]
```

Note that Mathematica places more specific definitions before more generic ones.



Decisions

Mathematica's **If Statement** has three entries: for True, for False, but also for Undecidable. For example:

`If[8 > 9, yes, no]`  no

`If[a > b, yes, no]`  `If[a > b, yes, no]`

`If[a > b, yes, no, dunno]`  dunno

Property-testing Functions end in Q: `EvenQ`, `PrimeQ`, `NumberQ`, `MatchQ`, `OrderedQ`, ... These functions have *no undecided state*: in case of doubt they return `False`.

Conditional Patterns are usually faster:

```
good[a_, b_] := If[TrueQ[a > b], 1, 2]
```

– `TrueQ` removes ambiguity

```
better[a_, b_] := 1 /; a > b
```

```
better[a_, b_] = 2
```



Equality

Just as with decisions, there are several types of equality, decidable and undecidable:

`a == b`  `a == b`


`a === b`  `False`

`a == a`  `True`

`a === a`  `True`

The full name of '===’ is `SameQ` and works as the `Q` indicates: in case of doubt, it gives `False`. It tests for **Structural Equality**.

Of course, equations to be solved are stated with '==':

`Solve[x^2 == 1, x]`  `{{x -> -1}, {x -> 1}}`

Needless to add, '=' is a definition and quite different:

`x = 3` — assign 3 to x



Selecting Elements

Select selects elements fulfilling a criterium:

```
Select[{1, 2, 3, 4, 5}, # > 3 &] → {4, 5}
```

Cases selects elements matching a pattern:

```
Cases[{1, a, f[x]}, _Symbol] → {a}
```

Using **Levels** is generally a very fast way to extract parts:

```
list = {f[x], 4, {g[y], h}}
```

```
Depth[list] → 4 — list is 4 levels deep (0, 1, 2, 3)
```

```
Level[list, {1}] → {f[x], 4, {g[y], h}}
```

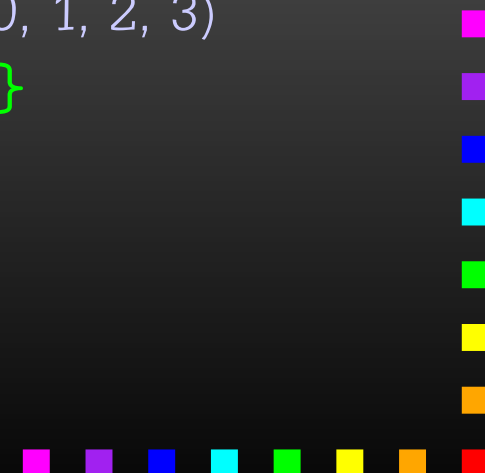
```
Level[list, {2}] → {x, g[y], h}
```

```
Level[list, {3}] → {y}
```

```
Level[list, {-1}] → {x, 4, y, h}
```

```
Cases[expr, _Symbol, {-1}]/Union
```

— find all variables in expr



Mathematical Functions

Mathematica is equipped with a large set of mathematical functions, both for symbolic and numeric operations.

Some examples:

`Integrate[x^2, {x,3,5}]`

– integral

`D[f[x], x]`

– derivative

`Sum[i, {i,50}]`

– sum

`Series[Sin[x], {x,1,5}]`

– series expansion

`Simplify[(x^2 - x y)/x]`

– simplify

`Together[1/x + 1/y]`

– put on common denominator

`Inverse[mat]`

– matrix inverse

`Eigenvalues[mat]`

– eigenvalues

`PolyLog[2, 1/3]`

– polylogarithm

`LegendreP[11, x]`

– Legendre polynomial

`Gamma[.567]`

– Gamma function



Graphics

Mathematica has formidable graphics capabilities:

```
Plot[ArcTan[x], {x, 0, 2.5}]
```

```
ParametricPlot[{Sin[x], 2 Cos[x]}, {x, 0, 2 Pi}]
```

```
Plot3D[1/(x^2 + y^2), {x, -1, 1}, {y, -1, 1}]
```

```
ContourPlot[x y, {x, 0, 10}, {y, 0, 10}]
```

Output can be saved to a file with `Export`:

```
plot = Plot[Abs[Zeta[1/2 + x I]], {x, 0, 50}]
```

```
Export["zeta.eps", plot, "EPS"]
```

Hint: To get a high-quality plot with proper \LaTeX labels, don't waste your time fiddling with the `Plot` options. Use the `psfrag` \LaTeX package.



Numerics

Mathematica can express **Exact Numbers**, e.g.

`Sqrt[2]`, `Pi`, $\frac{27}{4}$

It can also do **Arbitrary-precision Arithmetic**, e.g.

`N[Erf[28/33], 25]`  0.7698368826185349656257148

But: Exact or arbitrary-precision arithmetic is fairly slow!
Mathematica uses **Machine-precision Reals** for fast arithmetic.

`N[Erf[28/33]]`  0.769836882618535

Arrays of machine-precision reals are internally stored as **Packed Arrays** (this is invisible to the user) and in this form attain speeds close to compiled languages on certain operations, e.g. eigenvalues of a large matrix.



Compiled Functions

Mathematica can 'compile' certain functions for efficiency.

This is not compilation into assembler language, but rather a **strong typing** of an expression such that intermediate data types do not have to be determined dynamically.

```
fun[x_] := Exp[-((x - 3)^2/5)]  
cfun = Compile[{x}, Exp[-((x - 3)^2/5)]]  
time[f_] := Timing[Table[f[1.2], {10^5}]] [[1]]  
time[fun]    ➡ 2.4 Second  
time[cfun]   ➡ 0.43 Second
```

Compile is implicit in many numerical functions, e.g. in Plot.

In a similar manner, Dispatch hashes long lists of rules beforehand, to make the actual substitution faster.



Blocks and Modules


Block implements Dynamical Scoping


A local variable is known everywhere, but only for as long as the block executes (“temporal localization”).

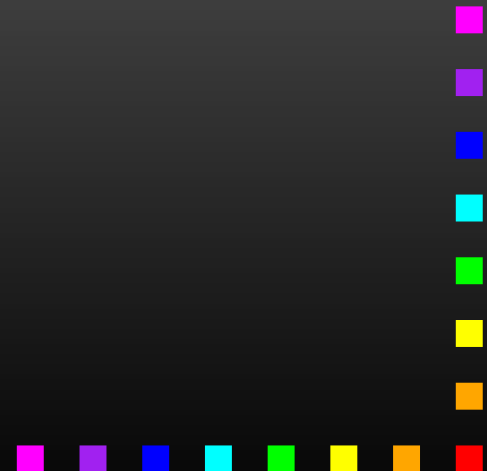
Module implements Lexical Scoping

A local variable is known only in the block it is defined in (“spatial localization”). This is how scoping works in most high-level languages.

```
a = 7
printa := Print[a]
btest := Block[{a = 5}, printa]
mtest := Module[{a = 5}, printa]
```

btest  5

mtest  7



Compare to C Scoping

Mathematica:

```
a = 7  
printa := Print[a]  
btest := Block[{a = 5}, printa]  
mtest := Module[{a = 5}, printa]
```

C:

```
int a = 7;  
void printa() { printf("%d\n", a); }  
void mtest() {  
    int a = 5;  
    printa();  
}
```




DownValues and UpValues

Definitions are usually assigned to the symbol being defined: this is called **DownValue**.

For seldomly used definitions, it is better to assign the definition to the next lower level: this is an **UpValue**.

```
x/: Plus[x, y] = z
```

```
?x  Global`x  
x /: x + y = z
```

This is better than assigning to `Plus` directly, because `Plus` is a very common operation.

In other words, Mathematica “**looks**” one level inside each **object** when working off transformations.



Output Forms

Mathematica knows some functions to be **Output Forms**. These are used to format output, but don't "stick" to the result:


`{{1, 2}, {3, 4}}//MatrixForm`  $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$

`Head[%]`  `List` — not `MatrixForm`

Some important output forms:

`InputForm`, `FullForm`, `Shallow`, `MatrixForm`, `TableForm`, `TeXForm`, `CForm`, `FortranForm`.

`TeXForm[alpha/(4 Pi)]`  $\frac{\alpha}{4\pi}$

`CForm[alpha/(4 Pi)]`  `alpha/(4.*Pi)`

`FullForm[alpha/(4 Pi)]`

 `Times[Rational[1, 4], alpha, Power[Pi, -1]]`

MathLink

The **MathLink API** connects Mathematica with external C/C++ programs (and vice versa). **J/Link** does the same for Java.

```
:Begin:  
:Function:      copysign  
:Pattern:      CopySign[x_?NumberQ, s_?NumberQ]  
:Arguments:    {N[x], N[s]}  
:ArgumentTypes: {Real, Real}  
:ReturnType:   Real  
:End:
```

```
#include "mathlink.h"
```

```
double copysign(double x, double s) {  
    return (s < 0) ? -fabs(x) : fabs(x);  
}
```

```
int main(int argc, char **argv) {  
    return MLMain(argc, argv);  
}
```

In-depth tutorial: <http://library.wolfram.com/infocenter/TechNotes/174>



Scripting Mathematica

Efficient batch processing with Mathematica:

Put everything into a script, using **sh's Here documents**:

```
#!/bin/sh ..... Shell Magic
math << \_EOF_ ..... start Here document (note the \)
  << FeynArts'
  << FormCalc'
  top = CreateTopologies[...];
  ...
\_EOF_ ..... end Here document
```

Everything between “<< *tag*” and “*tag*” goes to Mathematica as if it were typed from the keyboard.

Note the “\” before *tag*, it makes the shell pass everything literally to Mathematica, without shell substitutions.



Scripting Mathematica

- Everything contained in **one compact shell script**, even if it involves several Mathematica sessions.
- Can combine with arbitrary shell programming, e.g. can use **command-line arguments** efficiently:

```
#!/bin/sh
math -run "arg1=$1" -run "arg2=$2" ... << \END
...
END
```

- Can easily be **run in the background**, or combined with utilities such as **make**.

Debugging hint: **-x flag** makes shell echo every statement,

```
#!/bin/sh -x
```



Mathematica Summary

- Mathematica makes it wonderfully easy, even for fairly unskilled users, to manipulate expressions.
- Most functions you will ever need are already built in. Many third-party packages are available at MathSource, <http://library.wolfram.com/infocenter/MathSource>.
- When using its capabilities (in particular list-oriented programming and pattern matching) right, Mathematica can be very efficient.
Wrong: `FullSimplify[veryLongExpression]`.
- Mathematica is a general-purpose system, i.e. convenient to use, but not ideal for everything.
For example, in numerical functions, Mathematica usually selects the algorithm automatically, which may or may not be a good thing.



Shameless Advertisement

Want to stay adrift of trends in the Computer Algebra scene?

Fachgruppe Computeralgebra

www.fachgruppe-computeralgebra.de

The Fachgruppe Computeralgebra is a committee on Computer Algebra jointly organized by DMV, GAMM, and GI.

Twice a year it issues the “Computeralgebra-Rundbrief” which contains news and information related to computer algebra, e.g. conference announcements, surveys of publications, etc.

Membership is € 9 per year.

