{ Alife Mutants Hackingsession on Systems and Organisms, Bielefeld 2004 }

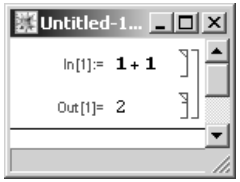
{ Introduction to Mathematica }

In this workshop you learn how to use Mathematica 5.0 and its various Packages

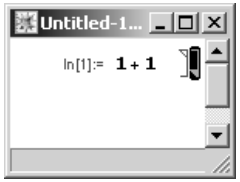
Martin Schneider, Universität Osnabrück

{ getting started }

- the **notebook window**
  - to enter input start typing...
    - example: 1 + 1
  - then press [shift] [enter] to obtain the result



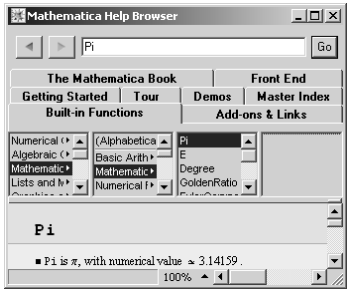
- the **cells**
  - every input and output goes into a separate cell
  - you can open and close cells
    - doubleclick the cell bracket to open and close it
  - cells can also be nested



Martin Schneider, Universität Osnabrück

{ getting help }

- to get help open the **Help Browser**
  - the Help Browser contains
    - the complete Mathematica Book
    - Build-In Functions + Examples
    - Tours & Demos
  - start it via
    - the help menu
    - pressing [shift] [f1]
    - context help: select some text, then press [f1]
  - use the Master index to search all entries



Martin Schneider, Universität Osnabrück

{ shortcuts I }

- get online help on a function
  - type **?** before the function name in the Notebook
    - example: type ?CellularAutomaton
- complete a partial function name
  - press [ctrl] k , then select from the popup window
- get a function template
  - press [shift][ctrl] k

Martin Schneider, Universität Osnabrück

shortcuts II

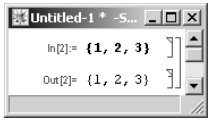
- copy **input** from the previous cell
  - press [ctrl] L
- copy **output** from the previous cell
  - press [shift][ctrl] L
- expand selection (select enclosing block)
  - doubleclick or press [ctrl] . (dot)
- to get a complete list of shortcuts
  - see: helpbrowser – ‚keyboard shortcuts'

expressions I

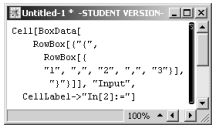
- in Mathematica everything is an expression
- **functions**
  - expressions that are immediately evaluated
    - example: Expand[(x+1)^8]
  - note:
    - Mathematica functions always start with capital letters
    - arguments in square brackets

expressions II

- Other Expressions
  - unknown expressions and lists will simply be returned unprocessed
    - example: expand[(x-1)^8]
    - example: {1,2,3}
  - Note:
    - list use curly braces

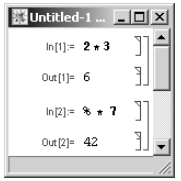


- even notebooks and cells are expressions
  - to see the cell expression select it and press [shift] [ctrl] e
  - press [shift][ctrl] e again to get the original cell back

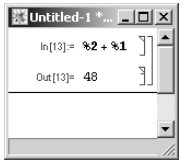


special variables

- you can refer to the previous result using %



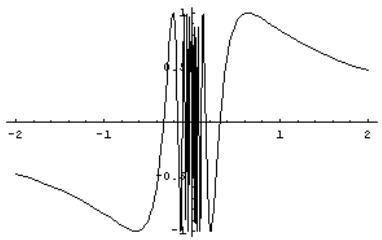
- older results can be accessed via Out[n] or simply ‚%n'



graphic plots

- 2D plot
  - plotting functions
    - one independent variable
  - 1st argument
    - function
  - 2nd argument
    - range list
      - {var,min,max}

```
In[1]:= Plot[Sin[1/x], {x, -2, 2}]
```

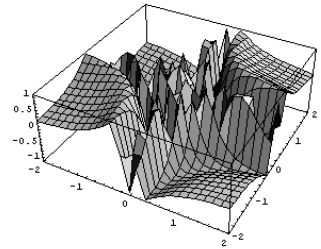


```
Out[1]= -Graphics-
```


graphic plots

- 3D plot
  - plotting functions
    - 2 independent variables
  - 1st argument
    - function
  - 2nd argument
    - range list for var1
  - 3rd argument
    - range list for var2

```
In[2]:= Plot3D[Sin[1/(x y)], {x, -2, 2}, {y, -2, 2}]
```

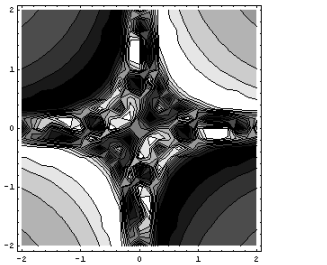


```
Out[2]= -SurfaceGraphics-
```


graphic plots

- contour plot
  - topographic map of a function
    - contourlines
    - shading value intervals

```
In[3]:= ContourPlot[Sin[1/(x y)], {x, -2, 2}, {y, -2, 2}]
```

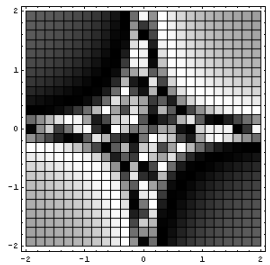


```
Out[3]= -ContourGraphics-
```


graphic plots

- density plot
  - discrete sampling of the function
    - shading corresponds to function value
    - black: lowest
    - white higehst

```
In[4]:= DensityPlot[Sin[1/(x y)], {x, -2, 2}, {y, -2, 2}]
```

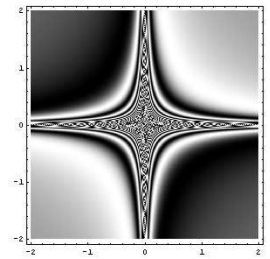


```
Out[4]= -DensityGraphics-
```

options I

- passing options
  - many functions take optional arguments (Options) written as Option → Value pairs in no particular order.
  - to get a list of all Options use Options[Function]

```
In[5]:= DensityPlot[Sin[1/(x y)], {x, -2, 2},
          {y, -2, 2}, PlotPoints → 1000, Mesh → False]
```

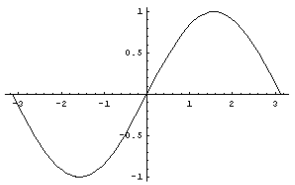


```
Out[5]= -DensityGraphics-
```


options II

- Automatic Value
  - if an option has the Value ‚Automatic' Mathematica will figure out the optimal setting automatically
  - If you want to see the current value used in a Graphics Object use AbsoluteOptions
  - to set Options permanently use SetOption
    - example:
      - SetOptions[Plot, Frame → True]

```
In[1]:= Plot[Sin[x], {x, -Pi, Pi}]
```



```
Out[1]= -Graphics-

In[2]:= Options[%1, PlotRange]
Out[2]= {PlotRange → Automatic}

In[3]:= AbsoluteOptions[%1, PlotRange]
Out[3]= {PlotRange → {{-3.29867, 3.29867},
          {-1.05, 1.05}}}
```


definitions

- immediate assignment
  - lhs = rhs
  - righthand side is evaluated before assigning it

```
In[1]:= r1 = Random[Integer, 10]
Out[1]= 6

In[2]:= r1 ^2
Out[2]= 36

In[3]:= ?r1
        Global`r1
        r1 = 6
```


functions

- delayed assignment
  - lhs := rhs
  - righthand side is evaluated when the function is used

```
In[4]:= r2 := Random[Integer, 10]

In[6]:= {r2, r2, r2, r2, r2}
Out[5]= {10, 5, 1, 8, 6}

In[6]:= ?r2
        Global`r2
        r2 := Random[Integer, 10]
```

how to combine functions

- there are basically four ways to do it
  - use previous results, employing variables and ‚%‘
    - you can built up your results step by step
    - you can reuse previous results
  - nest functions g[f[x]]
    - the expression stands on its own
    - you can include it in a package
  - prefix ‚g @ f‘
    - saves you a bracket
    - precendence may become unclear
  - postfix ‚f // g‘
    - nice for adding afterthoughts quickly

how to input functions

- there are basically four ways to do it
  - 1d notation
    - alphanumeric characters only (shell mode)
  - 2d notation using palettes
    - menu: file – palettes
  - 2d notation using special keys
    - helpbrowser: frontend – ‚entering 2d expressions‘
  - fullform notation
    - every function operator has a fullform equivalent

how to output functions I

- there are four basic forms of display
  - OutputForm
    - this form is used for output by default (2d notation)
  - InputForm
    - shows the function using 1d notation
  - TraditionalForm
    - shows the Function as you would find it in a Math Textbook
  - StandardForm
    - shows the Function in an unambigous form (using squarebrackets for functions instead of round brackets)

how to output functions II

- other useful forms:
  - FullForm
    - shows the function using 1d notation with full function names for all operators
  - Short
    - print only the beginning and the end of an expression –useful for very large lists.
  - HoldForm
    - displays the output without evaluating it first, you may combine this with FullForm

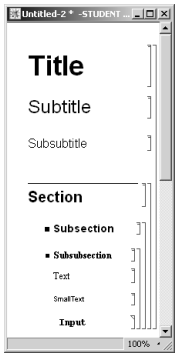
```
In[1]:= 1/2 // FullForm
Out[1]//FullForm=
        Rational[1, 2]

In[2]:= $ContextPath // Short
Out[2]//Short= {DiscreteMath`Combinatorica`,   (
         Utilities`Package`,
         <<9>>, System`}

In[3]:= 1/2 // FullForm // HoldForm
Out[3]= Times[1, Power[2, -1]]
```

{ how to save your work }

- creating a notebook (beginners)
  - use the various styles for creating a document where the actual code is interwoven with documentation
  - use the OptionInspector to define InitializationCells which will be executed when the Notebook is opened.
- creating a package (advanced)
  - put all your code in a single .m -file
  - provide function documentation
- creating a complete application (professional)
  - program packages
  - create documentation for the HelpBrowser
  - create notebooks to demonstrate your application
  - create palettes for quick and easy access
  - define OutputForms for your functions if needed

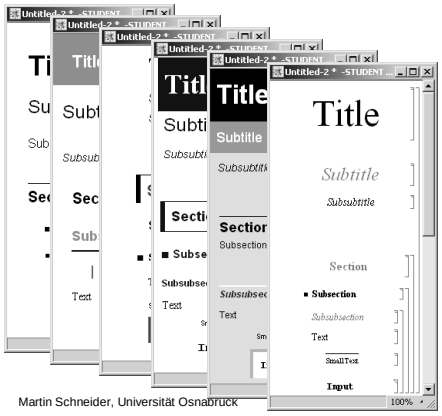
{ creating a notebook }

- a notebook can contain
  - input cells
  - output cells
  - normal text cells
- to change cell style
  - select the cell
  - select format-style from the main menu or press Alt-1 through Alt-9 to select the most common styles



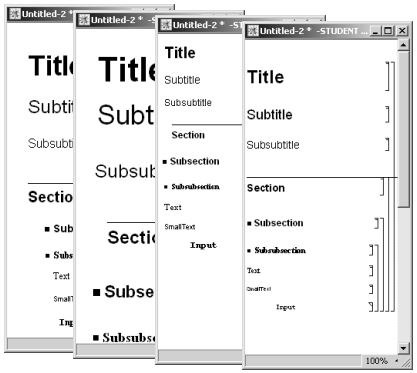
{ style templates }

- change notebook stylesheets
  - there are several predefined stylesheets
    - main menu: format – style sheets
  - you can easily modify style sheets
    - main menu: format – edit style sheet



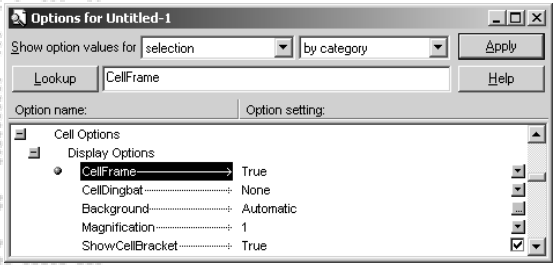
{ style templates }

- change appearance for screen and print
  - **working**
  - presentation
  - condensed
  - printout style



option inspector

- the **Option Inspector** lets you change every property of your notebooks / cells.
  - try to change the CellFrame option for a cell



using packages

- packages are arranged in a hierarchy
  - the hierarchy corresponds to the package directory structure but Mathematica use backticks rather than slashes or backslashes
- to load a package use the ‚<<' operator
  - example:
    - <<DiscreteMath`Combinatorica`
  - make sure you use backticks correctly!
- to list all Functions of the package use ‚?'
  - example
    - ?DiscreteMath`Combinatorica`*
  - note the usage of the asterisk wildcard '*' !

using packages II

- the package Utilities`Package` provides some useful procedures to analyse packages

```
In[1]:= << Utilities`Package`; << DiscreteMath`Combinatorica`

In[2]:= Annotation["DiscreteMath`Combinatorica`", "Summary"]

Out[2]= {(* :Summary:
     This package contains all the programs from the book, "Computational
     Discrete Mathematics: Combinatorics and Graph Theory in Mathematica",
     by Sriram V. Pemmaraju and Steven S. Skiena, Cambridge University Press,
     2003.
     *)}

In[3]:= FindPackages[$Path, "Combinatorica", FullPath → True]

Out[3]= {{}, {}, {}, {}, {}, {}, {}, {}, {}, {},
      {C:\Programme\Wolfram Research\Mathematica\5.0\AddOns\StandardPackages\
        DiscreteMath\Combinatorica.m}, {}, {}, {}, {}, {}, {}}
```


namespaces

- each package has its own namespace
  - $Packages returns a list of loaded packages
  - $Context returns the current namespace context
  - the namespaces on $ContextPath are searched from left to right if a symbol is not found in the current context
  - note that packages may automatically load other packages they use

```
In[1]:= << DiscreteMath`Combinatorica`

In[2]:= $Packages // Short

Out[2]//Short= {DiscreteMath`Combinatorica`,
      Statistics`Common`DistributionsCommon`,
      DiscreteMath`Tree`,
      «6», Global`, System`}

In[3]:= $Context

Out[3]= Global`

In[4]:= $ContextPath // Short

Out[4]//Short= {DiscreteMath`Combinatorica`,
      Statistics`DiscreteDistributions`,
      «7», Global`, System`}
```

creating a package

- the package skeleton
  - use standard tags in your comments to make your documentation more accessible
  - myFunction::usage will be shown when you type ,?myFunction'
  - Private Namespace for Implementation!
  - Protect your Functions at the end. Unprotect them in the beginning to allow your package to be loaded more than once

```
(* :anyTag:any Documentation*)

BeginPackage["myPackageName`"]
Unprotect[myFunction]

    BlablaFunction::usage =
     "BlablaFunction[x_] is returning a lot of bla"

    Begin["`Private`"]
        BlablaFunction[x_] := Table[bla, Random[x]]
    End[]

Protect[myFunction]
EndPackage[]
```


creating an application

- Directory Structure
  - myApplicationName
    - { notebooks.nb }
    - { packages.m }
    - Documentation
      - English
        - BrowserCategories.m
        - { Documentation Notebooks }
    - FrontEnd
      - Palettes
        - { Palette Notebooks }
      - StyleSheets
        - {StyleSheet Notebooks }
    - Kernel
      - init.m
- Look at Mathematica Add-Ons / Applications to find out how to do it
- check library.wolfram.com for developer documentation

functional programming I

- design principles
  - functions as objects
  - declarative, often recursive definitions
  - many small functions (easy to reuse and debug)
- creating functions
  - Nest
    - Nest[f,x,3] = f[f[f[x]]]
  - Fold
    - Fold[f,x,{1,2,3}] = f[f[f[x,1],2],3]
  - Table
    - Table[f[x],{x,1,3}] = {f[1],f[2],f[3]}

functional programming II

- manipulating functions
  - list functions
    - can be applied to any kind of expression
    - Position, MapAt, MapAll ...
- transforming functions
  - Map
    - f @ g[x,y] = g[f[x],f[y]]
    - applies a function to each argument of a function.
  - Apply
    - f @@ g[x,y] = f[x,y]
    - replaces the head of a function

{ functional programming III }

- analyzing functions
  - Head
    - Head[f[x,y] = f
  - Part
    - Part[f[x,y],2] = y
- pure functions
  - fn[#1,#2,##3]&
    - create a function using the ‚&' postfix
    - use # to create anonymous slots

{ procedural programming }

- design principles
  - flow of control
  - procedures are stated explicitly
  - assigning intermediate results to variables
- flow control
  - If[cond,t,f]
    - evaluate either t or f depending on the truth value of cond.
  - Which[test1,value1,test2,value2,...]
    - returns the value corresponding to the first test resulting in ‚True'
  - Switch[expr,form1,value1,form2,value2...]
    - returns the value according to the first match between expr and form
- iteration
  - Do, While, For, Goto / Label ...

{ variables and scope }

- Global Variables
  - SystemVariables
    - starting with $
  - Listing Variables
    - ?$* : system vars
    - ?@ : all user vars
- Scope
  - Module
    - lexical scoping
  - Block
    - dynamic scoping
  - With
    - lexical scoping
    - replacement, ‚local constants'
    - resolving multi-level
- Local Variables
  - Module[var-list,body]
    - take only effect in the local Module/Block
    - Mathematica creates for every local variable ‚var', a unique instance ‚var$num'

```
In[1]:= m = i^2

Out[1]= i^2

In[2]:= Block[{i = a}, i + m]

Out[2]= a + a^2

In[3]:= Module[{i = a}, i + m]

Out[3]= a + i^2
```


{ errors and debugging }

- simple output
  - use Print to output variabes to the current Notebook
  - you may also define your own output function to write to a special output window (notebook), a file etc...
- tracing
  - TracePrint shows every evaluated Expression that matches a certain pattern
  - TraceDialog
    - stop whenever the pattern occurs
    - you can then analyze variables etc.
    - type Return[ ] to keep tracing or Exit[ ] when done.
  - there are visual debugger tools at library.wolfram.com !
- throw and catch
  - you can throw and catch exceptions based on patterns
  - you can selectively mute certain exceptions messages

programming by the rule

- rule
  - pattern -> replacement (immediately)
  - pattern :> replacement (delayed)
- rules
  - List of Rules: {p1->r1,p2->r2, ... }
- ReplaceAll
  - expr /. rule(s)
  - apply the transformation rule(s) to every sub-expression
- ReplaceRepeated
  - p->x //. expr
  - apply transformation rule(s) until no more change occurs

pattern matching I

- similar to regular expressions
  - but matches can be referred to by name
- ‚_‘
  - any single expression (.)
- ‚__‘
  - sequence of one or more expressions (+)
- ‚___‘
  - sequence of zero or more expressions (*)
- name placeholders by prepending a variable name
  - name_, name__, name___
- match only expressions with the given head
  - _head, __head, ___head
- type checking
  - for atomic expressions the head is either Integer,Real or Symbol
  - for compounds the Head is also often used to identify the datatype (List, Complex,Graph...)

pattern matching II

- ‚expr..‘
  - pattern repeated one or more times
- ‚expr...‘
  - pattern repeated zero or more times
- pattern1 | pattern2 | ...
  - match any of the alternatives
- ‚pattern ? test‘
  - apply constraint on currently matched pattern
  - x_?Positive
- ‚pattern /; test
  - general constraint
  - {x_,y_} /; Positive[x]

function interfaces

- function overloading
  - different versions of the function for different data types or argument numbers
  - default values for optional arguments
    - fn[arg1, arg2 : default] := ...
    - colon operator is overloaded too
- parsing options
  - fn[arg1,arg2,..., opts___]
  - filter Options using rules
    - val = key /. { default-opts , opts }
  - use filter utilities
    - << Utilities`FilterOptions`

{ import - export }

- Import[„filename“,format] or Import[„filename“]
  - Mathematica tries to guess file format from the ending
- text
  - CSV, Lines, List, Table, Text, TSV, UnicodeText, Words
- XML
  - ExpressionML, MathML, NotebookML, SymbolicML, XML
- numerical
  - FITS, HarwellBoeing, HDF, MAT, MTX, SDTS

{ import - export }

- vector graphics
  - EPS, EPSI, EPSTIFF, MPS
- raster graphics
  - BMP, DICOM, GIF, JPEG, MGF, PBM, PGM, PNG, PNM, PPM, TIFF, XBitmap
- 3d graphics
  - DXF, STL
- sound
  - AIFF, au, snd, wav

{ graphic objects }

- Graphic Objects
  - Graphics
  - Graphics3D
  - SurfaceGraphics
  - ContourGraphics
  - DensityGraphics
  - GraphicsArray
- Graphic Primitives
  - Point, Line, Circle, Rectangle, Polygon, Text ...
- Graphics[p]
  - p can either be a Primitive or a list of Primitives
- GraphicsDirectives
  - OpenGL-like (state-machine)
    - a directive applies to all successive primitives
    - push and pop state using sublists
  - Colour, Thickness AbsoluteThickness, PointSize, AbsolutePointSize

{ creating graphics }

- GraphicsArray
  - a Table of Graphic Objects
- FullGraphics
  - turns special Elements (Axis, Marks ...) of Plots into a Graphic Primitives, so you can manipulate them
- ImplicitGraphics, InequalityPlot, InequalityPlot3D
  - create implicit Graphics using the computational Power of Mathematica !
- Animations
  - Create a List of Graphics objects
  - doubleclick to animate
- Interactive Graphics
  - <<RealTime3D`
  - interactively rotate the object in space to look at it from all angles

{ connecting and networking }

- MathLink
  - alternative frontend
  - external program
  - other kernels
- JLink
  - interface to Sun's java
- .Net/Link
  - interface to Microsofts .Net

{ sound and music }

- Sound Primitives
  - SampledSoundFunction
  - SampledSoundList
- Play and ListPlay
  - completely analogous to Plot and ListPlot
  - output Sound to arbitrary Number of Channels
- The Sound of Cellular Automata
  - listen to rule110 {01}
  - listen to rule110 {10}

{ fun with mathematica }

- dynamic nodebooks and selfreplicating code
  - rule110 loves you
- physical, bilogical and social simulations
  - various Books and Packages by Richard Gayman
- network simulations
  - Combinatorica Package
- evolutionary algorithms
  - Principica Evolvica