Rascal
the Advanced Scientific CALculator

Userguide

Sebastian Ritterbusch

7th May 2002

Contents

1 Introduction 1

2 Basic Usage 1

3 Basic Data Types 2

3.1 Integers and Doubles 2

3.2 Strings 3

3.3 Generic Matrices and Vectors 3

4 Loops 4

5 Signals 4

6 Generic Complex Arithmetics 4

7 Generic Polynomial Arithmetics 5

8 Generic Taylor Arithmetics 5

9 C-XSC Interval Arithmetics 6

10 Plotting functions 7

11 C++ Toolbox for Verified Computing 7

11.1 Evaluation of Polynomials 7

11.2 Zeros of Complex Polynomials 7

11.3 Linear Systems of Equations 8

11.4 Linear Optimization 8

12 Questions? 8

13 Licence 9

1 Introduction

Rascal is based on an extremely modular approach; depending on the user's needs, various powerful modules can be compiled into the system. Thus Rascal can be a light-weight tool as well as a very powerful computational system. This user documentation is created during compilation and describes the included features.

2 Basic Usage

Invoking the program leads to a prompt where you can enter expressions. The expressions or definitions are being evaluated as soon you press return. A semicolon at the end suppresses any output. As an example:

```
>5+2
7
>5+2;
>
```

Variables are also supported, where the names are case-sensitive alphabetic letters and may be followed by numbers. An assignment is done using the equal sign:

```
>MyVariable2=7;
>MyVariable2*MyVariable2
49
>
```

Undefined variables are always assumed to be integer zeros.

The user can also define functions of one variable (n-ary functions can be implemented using vectors or matrices as arguments). The names follow the same rules as the names of variables:

```
>MyNiceFunction17(x)=x*x+5*x;
>MyNiceFunction17(2)
14
>EvalAt12(F)=F(12);
>EvalAt12(sqr)
 144
```

The semicolon at the end of the definition is mandatory and the calling-method is "call by variable", similar to parametric "#define"s in C++ with the difference that changes in the operand-variables are not forwarded to the original variable. Of course functions can be nested and invoked with any data-type. If the data-type does not support a certain operation, an error occurs. The last example shows how functions can be used as variables, here the function value of "square" at 12 was computed.

Conditional expressions can be realized using the C-style "?:"-operator:

```
>5==3?2.3:17
17
>abs(x)=x>0?x:-x;
>abs(-18)
18
>abs(12)
12
>
```

If the condition before the questionmark is true, the first expression is returned, else the second. In contrast to C the result-types of the two alternatives here can be different, also both alternatives are computed before the decision takes place, thus recursions are not possible.

You can exit Rascal by entering "quit" followed by a return.

If Rascal was compiled using the "libreadline", then you may use the cursor-keys to flip back to commands and results from before, as well as benefit of command completion using the "tab"-key.

Furthermore the different modules predefine functions, which are documented in the following sections.

3 Basic Data Types

Rascal has a generic subsystem, which supports integers, doubles, strings and matrices.

3.1 Integers and Doubles

Simple numbers are being interpreted as integers, which in this version of Rascal are represented as integers on the underlying computer architecture. No rounding errors will occur, but undetected overflows may occur.

Using double precision floating-point numbers overcomes this problem, but rounding errors may occur. A number is being interpreted as a floating-point number when there is a decimal-point within or at the end of the number and/or an exponent. As an example `1.234e+12` represents $1.234 \cdot 10^{+12}$. Be warned that there is no exact representation for 0.1 and many many other numbers in binary floating-point representations.

All operands like `+,-,*,/,^,%` are defined for integers and doubles, together with standard functions `sin, cos, tan, cot, asin, acos, atan, acot, sinh, cosh, tanh, coth, asinh, acosh, atanh, acoth, log, exp, sqrt, sqr`. A postfix `!` computes the factorial of the operand (currently just in integer).

The values can be compared using the operators `==, !=, <, <=, >, >=`, logical expression can be connected using `&`, `|` and the logical negation `~`.

3.2 Strings

Mostly strings are used internally, as an example all datatypes have a function called `output` which defines how values of that type can be printed on the screen.

Besides strings can be used as "evaluation variable": As all the standard operators and functions are also defined for strings, one can determine the expression Rascal would evaluate if the argument only had been a real value.

Strings are enclosed in quotes:

```
>f(x)=x*x+2*x;
>f(f("y"))
 "y*y+2*(y)*y*y+2*(y)+2*(y*y+2*(y))"
>strlen(f(f("y")))
 49
```

But the operators `==, !=` are still used to compare two strings. The `strlen` function returns the length of the string.

3.3 Generic Matrices and Vectors

In Rascal vectors are just matrices with either just one row or one column. You can enter a matrix by using brackets, where values within a line are separated by spaces, lines are separated by semicolons. matrices of same size can be added and subtracted using the usual operators:

```
>A=[1 2;3 4;5 6]
 [1 2;3 4;5 6]
>B=[-3 -4;2 7;2 9]
 [-3 -4;2 7;2 9]
>A+B
 [-2 -2;5 11;7 15]
>A-B
 [4 6;1 -3;3 -3]
>rows(A)
 3
>cols(A)
 2
>size(A)
 3
```

Scalars can be multiplied to the matrices and two matrices can be multiplied if the number of columns of the first matches the number of rows of the second. Dividing by a matrix means multiplying with the inverse, of course this is only defined for quadratic matrices. The functions `rows, cols, size` return the number of rows, columns and maximum of the two values of a matrix or vector.

Types using the matrix inversion must be able to be compared to integers. If there is no inverse, an empty matrix is being returned. You may compute the determinant using the `det` function.

```
>[1 2]*[3;4]
 [11]
>[1;2]*[3 4]
 [3 4;6 8]
>A=[1 3;4 13];
>det(A)
 1
>1/A
 [13 -3;-4 1]
>[2 0;0 3]/A
 [26 -6;-12 3]
>
```

The cells of matrices can be of any type; here an example for an integer, double, matrix, string matrix:

```
>A=[1 2.34;[1 2;3 4] "hu"];
>A(1,2)
 2.340000
```

Now A looks like the following:

$$A = \begin{pmatrix} 1 & 2.34 \\ \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} & "hu" \end{pmatrix}$$

And the value at cell 1,2 is 2.34.

The summation of matrices and scalars is defined as the addition between the matrix and the identity of same size times the scalar. Vector valued functions can be defined easily, also with multiple arguments.

```
>f(x)=[x*x;x+2];
>f(8)
 [64;10]
>f([1 2;3 4])
 [[7 10;15 22];[3 2;3 6]]
>f[1 2;3 4]
 [[7 10;15 22];[3 2;3 6]]
>g(x,y,z)=[x+y*z x*z];
>g(12,-5,2)
 [2 24]
>
```

4 Loops

The support for loops is experimental- it is very likely that the syntax will change in future releases.

Rascal supports C(++)-style loops to a certain extend. The syntax is

```
for( <initialization> ; <condition> ;
<expression 1> ) <expression 2>
```

First the initialization expression is being evaluated- this should be used to initialize the variable used for the loop. If the given condition is true then expression 2 and 1 are evaluated (in that order) until the condition is false after the evaluation of 1. Have a look at this example:

```
>sum=0;
>for(x=1;x<10;x++) sum=sum+x;
>sum
 4950
```

The above example computes the sum of the numbers between 0 and 10. If you would like to see the intermediate results, leave out the semicolon at the end of the for statement:

```
>sum=0;for(x=1;x<10;x++)
sum=sum+x
 1
 3
 6
 10
 15
 21
 28
 36
 45
```

In contrast to C(++) there is no "comma-operator" in rascal yet, thus you cannot use it in for-statements. Sorry.

5 Signals

You can interrupt running computations in Rascal by sending a SIGINT signal- just press CTRL-C. Rascal will then try to stop all computations. If you press CTRL-C again before Rascal was able to get back to the interactive shell, Rascal will terminate.

If you press CTRL-C while you are in the interactive shell, Rascal will beep you and terminate if you repeat your signal.

Thus if you "for"-statement doesn't seem to terminate or you were trying to compute the factorial of "100000", just press CTRL-C and you won't lose you work done before.

6 Generic Complex Arithmetics

This module introduces complex arithmetics to Rascal. Complex numbers can be created by using the "complex"-function and it is often handy to define the purely imaginary constant i. As soon i has been defined, Rascal uses i for output. Politically correct you may also use j as the imaginary unit.

```
>complex(1,2)+3
 complex(4,2)
>i=complex(0,1);
>(5+3*i)*(2+i)
 (7+4*i)
>
```

The real and imaginary part of a complex number can be accessed like the cells of vectors: The first cell is the real, the second the imaginary value. Additionally like matrices complex values can consist of all different types of data. The "transpose" of a complex value is the complex conjugate.

```
>f(x)=x*x(1)+x(2);
>f(complex("x",4))
 complex("x*x+4","4*(x)")
>(2+3*complex(0,1))'
 complex(2,-3)
```

The following standard functions are defined for complex: `exp, log, pow, sqrt, sin, cos, tan, cot, asin, acos, atan, acot, sinh, cosh, tanh, coth, asinh, acoth, atanh, acoth`. The function `arg` computes the argument, the function `abs` the absolute value. The standard functions `sqrt, pow` for integers and doubles are being overridden by the corresponding complex versions.

The `log` represents the main value of the natural logarithm, the `pow(a,b)` or `a^b` is evaluated as $a^b = e^{b \ln a}$, the `sqrt(a)` as $a^{\frac{1}{2}}$ and the rest is determined based on these functions and the pendants on the real axis.

One could also define a complex out of matrices, but this is not advisable as this easily gets confusing and Rascal prefers complex values within matrices. You should also be aware that naively plugging interval-datatypes into a generic complex datatype will not yield verified results: As an example the `arg` function has to be treated differently, and most other standard functions depend on this.

7 Generic Polynomial Arithmetics

This module introduces polynomials in one variable to Rascal. You may create polynomials by the "poly"-function, but it is very useful to assign the variable X the monome 1*X^1. Then you can easily enter the polynomials.

```
>X=poly("X");
>p=2*X+X*X+4
 X^2+2*X+4
>p*p
 X^4+4*X^3+12*X^2+16*X+16
```

You can evaluate the polynomials like functions, as well as compute the derivate polynomial.

```
>p(2)
 12
>p'
 2*X+2
>p'(2)
 6
```

With this arithmetics you may easily compute powers of polynomial expressions or compute shifted polynomials:

```
>q=(3-X)^4
 X^4-12*X^3+54*X^2-108*X+81
>q(X+2)
 X^4-4*X^3+6*X^2-4*X+1
```

For polynomials the evaluation, derivation, addition, substraction, equality, inequality and powering by an integer are defined.

8 Generic Taylor Arithmetics

This concept offers the opportunity to accurately compute derivatives of functions. The relationship between the resulting vector and the derivative at the evaluation point is the following:

$$f(\mathrm{taylor}[x\ 1\ 0\ 0\ldots]) = [f(x)\ \frac{f'(x)}{1!}\ \frac{f''(x)}{2!}\ \ldots]$$

As an example $f(x) = \frac{x+2}{x-1}$ with $f'(x) = -\frac{3}{(x-1)^2}$:

```
>f(x)=(x+2)/(x-1);
>f(taylor[2 1])
 taylor[4 -3]
>X(u)=taylor[u 1];
>f(X(0))
 taylor[-2 -3]
>
```

Together with the string arithmetic this module can also be used to determine formulas for the derivatives:

```
>f(x)=(x+2)/(x-1);
>X(u)=taylor[u 1];
>f(X("z"))
 taylor["(z+2)/(z-1)"
"(1-(z+2)/(z-1))/(z-1)"]
>f'
 "(1-(x+2)/(x-1))/(x-1)"
```

As you can see there is a short-cut to this functionality, but you can only compute the first derivatives, for higher order derivatives you need to use `taylor` by yourself.

Besides the basic operations `+,-,*,/` the following standard functions are defined: `sqrt, sqr, exp, log, pow, sin, cos, tan, cot, asin, acos, atan, acot, sinh, cosh, tanh, coth, asinh, acosh, atanh, acoth.` Of course the datatype you put into the taylor arithmetic should have defined these functions as well. The cells can be accessed the same way like the cells of vectors.

It is also possible to define a function which computes the derivative of an arbitrary function at an arbitrary point:

```
>D(X)=X(1)(taylor[X(2) 1])(2);
>D[sqr;5]
 10
```

As the derivative of "square" is two times the argument, this result is correct.

9 C-XSC Interval Arithmetics

This module includes the C-XSC Interval Arithmetics into Rascal. You can find more information about C-XSC at `http://www.xsc.de` .

Intervals can be constructed out of integers and doubles using the constructor `Interval` or the bracket-notation:

```
>a=Interval(1,2)
 [ 1.000000, 2.000000]
>b=[3,4]
 [ 3.000000, 4.000000]
>c=Interval(.1)
 [ 0.099999, 0.100001]
```

Note that `[12]` would denote a 1×1 matrix, not a point interval. It seems to be confusing that `c` is not a point interval, but the actual value 0.1 cannot be represented using the binary floating point representation, thus it is correct to represent this value as an actual interval. This module overrides the parsing of floating point numbers to achieve this. For compatibility reasons you can still also use the string constructor:

```
>alternativeC=Interval(".1,.1")
 [ 0.099999, 0.100001]
```

You can compute the intersection of two intervals using the &-operator, the convex hull using the |-operator. The relational operators <,<=,>,>= can be used to compute the relations $\subset, \subseteq, \supset, \supseteq$.

You can access the upper and lower boundary of an interval using the functions `sup, inf`, the middle point using `mid` and the diameter with the help of the function `diam`.

```
>sup(a)
 2.000000
>mid(a)
 1.500000
```

The following standard functions are defined for intervals: `abs, exp, log, pow, sqrt, sqr, sin, cos, tan, cot, asin, acos, atan, acot, sinh, cosh, tanh, coth, asinh, acoth, atanh, acoth.`

The output format can be configured using the `Precision(width,digits)` command:

```
>Precision(15);pi=atan([1,1])*4
 [3.141592653589787,3.141592653589801]
>diam(pi)
 1.287858708565182E-014
```

As you can see doubles are formatted accordingly.

10 Plotting functions

Using gnuplot rascal can plot functions:

```
>f(x)=(x^3)/30-2*x+2;
>plot[f;-10;10;0.1];
```

This will plot the function f from -10 to 10, computing values with stepsize 0.1.

Parametric plots are done as follows:

```
>spiral(t)=[t*sin(t*pi*t)
t*cos(t*pi*t)];
>pplot[spiral,0,2,0.1];
```

Last but not least rascal can do surface plots:

```
>g(x,y)=x*x-y*y;
>splot[g;-2;2;-3;3;.2;.1];
```

The output of plots may be saved in postscript files using the plotOutput function:

```
>plotOutput("surface.ps");
>splot[g;-2;2;-3;3;.2;.1];
```

This is very basic plotting support and it is likely that the syntax will change in later versions.

11 C++ Toolbox for Verified Computing

This modules includes several algorithms of the Toolbox into rascal. Please refer to the book

R. Hammer, M. Hocks, U. Kulisch, D. Ratz: C++ Toolbox for Verified Computing I - Basic Numerical Problems. Springer-Verlag, Heidelberg, New York, 1995.

or the webpage `http://www.xsc.de/` for more information.

In Rascal you may enter `help toolbox` to check what toolbox modules are included.

11.1 Evaluation of Polynomials

The function `RPolyeval(`p`,`t`)` evaluates the real polynomial p at t at maximum accuracy, even in the neighborhood of a root where cancellation dooms an ordinary floating-point evaluation.

The function returns a vector with the entries `[`y`;`iy`;`it`;`err`]` where y is the result of a Horner scheme evaluation, iy represents a maximum accuracy enclosure, it the number of iterations needed and err an eventual error message. The function `rpeval(`p`,`t`)` just returns the resulting interval.

```
>X=poly("X");
>p=(X-2)^4
 X^4-8*X^3+24*X^2-32*X+16
>p(2.0001)
 -3.5527136788E-015
>RPolyEval(p,2.0001)
 [-3.5527136788E-
015;[1.0000000000E-
016,1.0000000001E-016];2;""]
>rpeval(p,2.0001)
 [1.0000000000E-
016,1.0000000001E-016]
```

11.2 Zeros of Complex Polynomials

The function `CPolyZero(`p`,`t`)` computes verified enclosures of the roots of a complex polynomial p starting from t by enclosing the zeros in narrow bounds. The coefficients of the deflated polynomial are also enclosed.

The function returns a vector with entries [y;p;err] where y is a complex enclosure of the root close to t, p the complex enclosure of the reduced polynomial and err represents an eventual error messages. The functions cpzero(p,t) and cpreduce(p,t) just return the enclosure of the root or of the reduced polynomial.

```
>X=poly("X");p=X^4-
(5+3*i)*X^3+(6+7*i)*X*X-(54-
22*i)*X+120+90*i;
>CPolyZero(p,1+i)
 [([ 1.4142135623,
1.4142135624]+[ 2.8284271247,
2.8284271248]*i);
 X^3+([-3.5857864377,
-3.5857864376]+[-0.1715728753,-
0.1715728752]*i)*X^2
 +([ 1.4142135623,
1.4142135624]+[ -3.3847763109,-
3.3847763108]*i)*X
 +([-42.4264068712,-
42.4264068711]+[
21.2132034355,21.2132034356]*i);""]
>cpzero(p,4)
 ([4.9999999999,5.0000000001]+[-
2.2569491536E-
036,2.2569491536E-036]*i)
```

11.3 Linear Systems of Equations

The function LinSolve(A,b) computes a verified solution to the square linear system $A * x = b$ for a real square matrix A and real vector b based on a Newton-like method for an equivalent fixed-point problem.

The function returns a vector with entries [x;$cond$;err] where x represents the verified enclosure or exact solution to the system, $cond$ a condition number estimate and err an eventual error message. The function linsys(A,b) just returns the resulting vector enclosure.

```
>A=1.*BoothroydDecker(6);
>b=[1;2;3;4;5;6];
>Precision(-10);
>(1/A)*b
 [0;1;-2;3.0000000001;-
4.0000000002;5.0000000005]
>LinSolve(A,b)
 [[0;1;-2;3;-
4;5];112890625.00085557;""]
>A*linsys(A,b)-b
 [0;0;0;0;0;0]
```

11.4 Linear Optimization

The functions RevSimplex[A;b;c] and LinOpt[A;b;c;$BStart$] determine the optimal solution of the linear optimization problem: Maximize $z = c^T * x$ where $A * x = b$ and $x \geq 0$. LinOpt validates the solution and determines enclosures of the true optimal solution set.

RevSimplex returns a vector with entries [x;v;z;err] where x denotes the optimal solution vector, v the optimal basic index set, which can be used as $bstart$ for LinOpt, z the optimal value and err might contain an error message. LinOpt returns a vector [z;V;X;no;err] where z is the enclose of the optimal value, V is a matrix which no lines represent the different index vectors, X a matrix which lines represent the different solution vectors and err again is the eventual error message.

```
>Precision(-10);A=[1 0 1 0 0;0
1 0 1 0;100 18 0 0 1];
>b=[50;200;5000];c=[50;9;0;0;0];
>S=RevSimplex[A;b;c]
 [[14;200;36;0;0];[1;3;2];2500;""]
>LinOpt[A;b;c;S(2)]
 [2500;[1 2 3;1 2 4;1 3 4];[14
200 36 0 0;50 0 0 200 0;50 0 0
200 0];3;""]
```

12 Questions?

Send questions, ideas, hints and congratulations to *rascal@ritterbusch.de.*

13 Licence