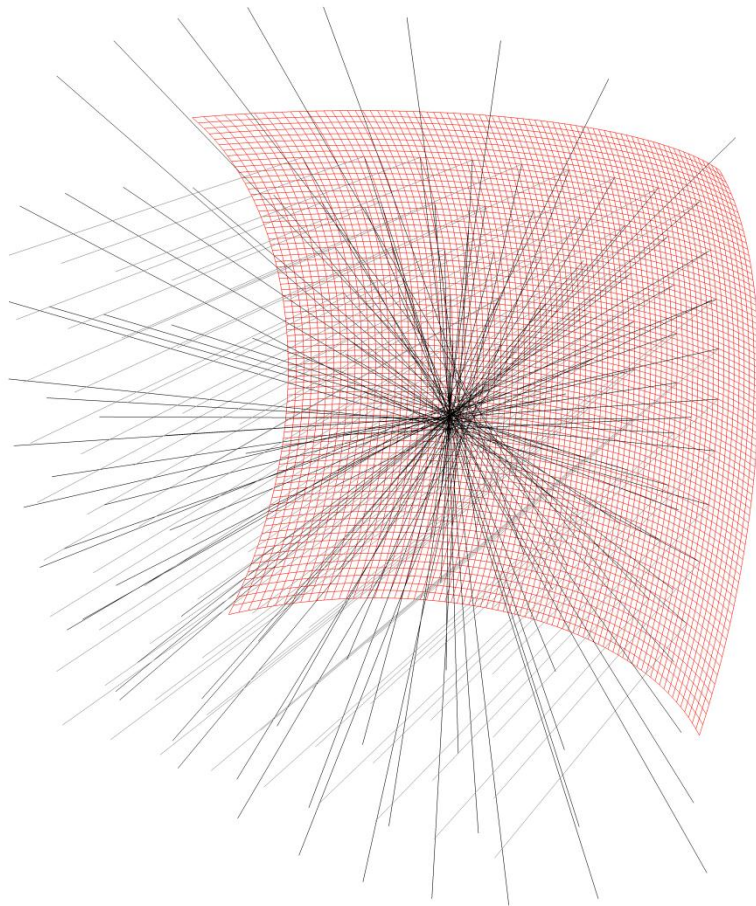


Andreas Rejbrand

# AlgoSim 2.0

## User's Guide





# Table of Contents

<b>PREFACE</b> .....	<b>6</b>
<b>THE MAIN WINDOW</b> .....	<b>7</b>
<b>PERFORMING CALCULATIONS</b> .....	<b>9</b>
KEYBOARD INPUT .....	10
VARIABLES AND FUNCTIONS .....	12
THE EXPONENTIATION OPERATOR .....	12
BASE- <i>N</i> CALCULATIONS .....	12
ABORTING A SLOW PROCEDURE.....	12
AUTOMATIC "ANS" ARGUMENT (AAA) .....	19
THE SEMICOLON OPERATOR.....	20
<b>VISUALISATION</b> .....	<b>21</b>
2D GRAPHS.....	21
2D PARAMETRISED CURVES .....	22
COLOURED SETS .....	23
3D SURFACES .....	25
3D CURVES .....	29
IMPLICIT SETS.....	31
NON-CARTESIAN COORDINATE SYSTEMS .....	32
COMPLEX VISUALISATION .....	34
COLOURED PLANES.....	36
THE BEAUTY OF THE TWO-STEP APPROACH.....	38
FINAL WORDS ON VISUALISATION .....	41
<b>PHYSICAL SIMULATIONS</b> .....	<b>42</b>
FORCE FIELDS.....	42
FLOWS.....	43
<b>AUDITORY VISUALISATION</b> .....	<b>45</b>
MIDI FUNCTIONS .....	45
<b>SOME MORE FUNCTIONS IN FOCUS</b> .....	<b>46</b>
REAL AND COMPLEX NUMBERS.....	46
VECTORS AND MATRICES.....	46
TEXTS (STRINGS) .....	47
PIXMAPS .....	47
SOUNDS AND MIDI FUNCTIONS.....	48
MORE.....	48
<b>THE OPERATOR TABLE</b> .....	<b>49</b>
<b>PROGRAMMING</b> .....	<b>50</b>
THE IF CONDITIONAL .....	50
THE REPEAT LOOP.....	51
THE DOWHILE LOOP .....	53
THE FOR LOOP .....	53
THE ITERATE LOOP .....	54

ENTERING PROGRAMS .....	54
A FEW EXAMPLES .....	55
<i>Möbius.prg</i> .....	55
<i>doors.prg</i> .....	55
<i>waveSim.prg</i> .....	56
<i>rutherfordScattering2.prg</i> .....	58
<i>mirrorSim.prg</i> .....	59
PROGRAMMING REFERENCE CHART .....	62
<b>DATABASE OF MATHEMATICAL AND PHYSICAL CONSTANTS.....</b>	<b>63</b>
<b>DICTIONARIES .....</b>	<b>64</b>
<b>SAVING/LOADING DATA .....</b>	<b>65</b>
<b>APPENDIX I: FUNCTION REFERENCE .....</b>	<b>67</b>
<b>APPENDIX II: PRE-DEFINED USER-CUSTOMISABLE FUNCTIONS.....</b>	<b>100</b>
<i>startup.prg</i> .....	100
<b>APPENDIX III: EXAMPLE PROGRAMS .....</b>	<b>101</b>
<b>APPENDIX IV: DEFAULT OPERATOR TABLE .....</b>	<b>104</b>
<b>APPENDIX V: DEFAULT TABLE OF CONSTANTS.....</b>	<b>106</b>
<b>APPENDIX VI: ONLINE HELP.....</b>	<b>107</b>
<b>APPENDIX VII: A FEW TIPS &amp; TRICKS.....</b>	<b>108</b>

*Happiness is like a wind, at times blowing through the consciousness,  
filling the individual with delight and inspiration to live.  
One ought to capture this wind, to locate its sources, and try to multiply them.  
Then, in times of darkness, these sources may be opened, flooding the individual with exhilaration.*

## **Preface**

This document is intended to give the reader the knowledge required to use all major features of AlgoSim 2.0, an advanced numerical mathematical software developed by Andreas Rejbrand. More information about the software is available at [www.algosim.se](http://www.algosim.se). You should not be afraid of contacting the developer/author at [andreas@rejbrand.se](mailto:andreas@rejbrand.se) in case you have any questions or comments.

## The Main Window

The Main Window is divided into four parts. To the left there is a column of buttons. These control the global behaviour of the application. From the top to the bottom, these are

- **Complex Mode**

If active (orange) AlgoSim will assume the user is working with the entire number system  $\mathbb{C}$  rather than  $\mathbb{R}$ . For the vast majority of cases, it does not matter whether Complex Mode is on or off – all real calculations will work in Complex Mode, and most complex calculations will work in Real Mode. But there are cases where this setting really matters. For instance, if you try to compute  $\arcsin(2)$  in Real Mode, you will get an error, because there is no real number  $x$  satisfying the equation  $\sin x = 2$ . In Complex Mode, however,  $\arcsin(2)$  will return  $1.57079632679 - 1.31695789692i$ .

- **Approximate Eq.**

If active (orange), very small numbers will be approximated by zero. In most cases this is desirable. For instance, in Approximate Eq. Mode,  $\sin(\pi) = 0$ , whereas  $\sin(\pi) = -5.42101086243 \cdot 10^{-20}$  in normal mode, due to rounding errors. However, if you are working with very small numbers, such as the elementary charge, the electron mass, or Planck's constant, you must not use this mode, for if you do, all these small numbers will be treated as equal to zero!

- **Num. Digits =  $N$**

Click this button to set the number of digits that are displayed in outputs. The maximum number of digits is 18.

- **Basis Vect. Notation**

If active (orange) the notation

$$\underline{\mathbf{e}} \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix}$$

will be used instead of  $(v_1, \dots, v_n)$  when it comes to vectors.

- **True Sets**

If active (orange), AlgoSim will make sure that a set does not contain the same element more than one time. When you add an element to a set, if this option is on, AlgoSim will iterate through all existing elements in the set to see if the new element already belongs to the set. If it does, the "new" element will not be added to the set. If the option is off, no such control is performed, so a set may very well contain the same element more than once. In most cases, when you work with sets as mathematical sets, you – of course – want to use this option. However, there are cases where this option needs to off. First, it does take quite some time to perform the check in large sets, so when working with huge sets (of points, for instance), you might want to disable this option. Also, if you have a set that you will use to draw a small number of points connected by straight lines, you might actually *want* the same element (point) to occur more than once.

- **Modular Arithm. Off/ $N$**

Click this button to start counting modulo  $N$ . To disable the option, click the button and enter "off" rather than a positive integer.

- **Full Screen**

If active (orange), the Main Window will occupy the entire computer screen, rather than just a window. In AlgoSim, entering full-screen is also possible by maximizing the Main Window.

- **Modular Messages**

If active (orange), error messages will not be printed out in the console (as normal output), but will rather be displayed in modal message boxes.

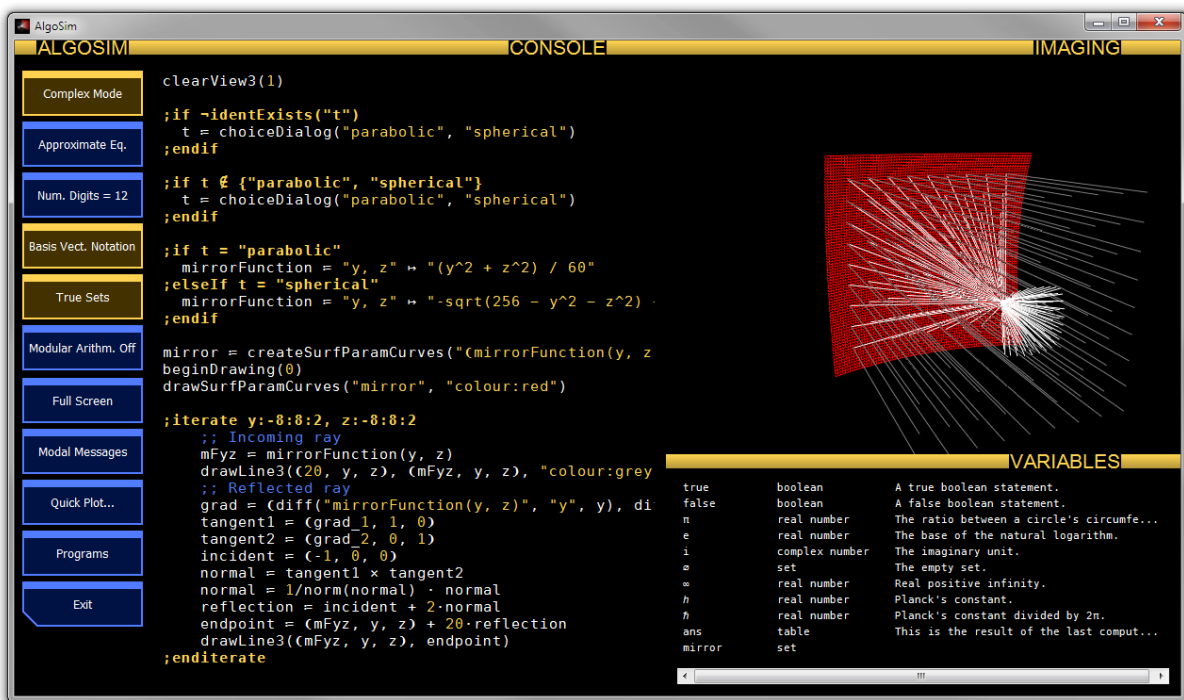
- **Quick Plot**

Displays the Quick Plot dialog box. The standard way of plotting function graphs is described in great detail this document. The Quick Plot function is a non-standard way of plotting 2D function graphs. It does not allow any fine control over the plotting process, but is very simple and convenient to use. Feel free to use this function for quick plotting of simple graphs.

- **Programs**

Displays the Programs menu, from which you can execute, create, and browse AlgoSim programs.

The largest panel in the Main Window is the *Console*. Here you enter commands and here the results are displayed. To the right there are two panels: *Imaging* and *Variables*. In Imaging graphical output is displayed, i.e. 2D spaces, 3D spaces, and bitmaps. In Variables all variables are displayed, together with their data types and (optional) descriptions. If you double-click somewhere in the list view of variables, the advanced variable manager window will appear. From this it is easy to search and export variables.



- To show/hide the left-most column of buttons, press F5.
- To show both the Console and the Imaging/Variables column (as default), press F2.
- To show only the Console, and not the Imaging/Variables column, press F3.
- To show only the Imaging panel, and not the console or the variables panel, press F4.
- To toggle full-screen mode on/off, press F11.



## Performing Calculations

Below the fundamental data types in AlgoSim are listed.

- **Real/Complex Numbers**

Real numbers are entered as usual, using the period "." as the decimal separator. To write a negative real number, place the **unary** minus sign "-" prefix operator before the number, as in -21. Notice that the unary minus operator "-" is shorter than the binary minus operator "-".

Complex numbers are constructed from real numbers by means of complex addition and multiplication, and the predefined constant  $i$ , the imaginary unit. For instance, one may write  $5 + 2i$ . Using the complex exponential function, complex numbers may also be entered in polar form:  $6 \cdot e^{(4 \cdot i)}$  or  $6 \cdot \exp(4 \cdot i)$ .

- **Real/Complex Vectors**

Vectors are entered using the pre-defined vector creator circumfix operator  $()$ , which can be inserted by Ctrl+E (as in **vector**). For instance,  $(2, 5, 2)$  or  $(5 \cdot i, 0, 5)$ . Vectors can be added, and multiplied by a scalar. The scalar product may either be written  $(u|v)$  or  $u \cdot v$  where  $u$  and  $v$  are real or complex vectors. The cross product is written  $u \times v$ .

- **Real/Complex Matrices**

Matrices are entered as vectors of vectors using the same operator  $()$ , inserted using Ctrl+E. Each vector in the vector is a *row* in the matrix. For instance, the orthogonal projection on the plane  $z = 0$  in  $\mathbb{R}^3$  may be represented by the matrix  $((1, 0, 0), (0, 1, 0), (0, 0, 0))$ . Matrices can be added, multiplied by a scalar, and matrices may be multiplied to each other. They may also be raised to any integer power; a negative power indicates the computation of the inverse matrix, which commutes with and non-zero power of the matrix. In particular, to compute the inverse of the matrix  $A$ , simply write  $A^{(-1)}$ . A matrix may be multiplied with a vector from the right – this is very important because this may be considered a linear transformation. (Of course, the vector is then considered to be a column matrix.) For a real matrix  $A^*$  (A asterisk) means the transpose, and for a complex matrix  $A^*$  means the Hermitian transpose, the adjoint matrix, or the conjugate transpose. If you only wish to transpose a complex matrix, use the command `transpose(A)`.

- **Strings**

A string is a piece of text – an array of Unicode characters. A string is represented by double quotes, such as "Hello World!". Strings may be added to each other, and the product between a natural number  $n$  and a string is equal to the string plus itself  $n$  times.

- **Sets**

A set is a collection of other objects, written  $\{ a, b, c, \dots \}$  where  $a, b, c, \dots$  are the members of the set. A set may contain any AlgoSim object except other sets and sounds. The binary operators  $\cup$  (union),  $\cap$  (intersection),  $\setminus$  (set difference), and  $\times$  (Cartesian product) are all fundamental set operators. However,  $\times$  only works for sets of numbers, because the elements in the Cartesian product must be valid AlgoSim objects, and a *vector* is indeed a valid AlgoSim object.

- **Pixmaps**

A pixmap is a raster (bitmap) image, like an illustration, or a photograph of Albus Dumbledore. Pixmaps may be saved as BMP, PNG (recommended), and PM files, and BMP, PNG, PM, and XBM image files may be imported.

- **Sounds**

A sound is a sampled waveform, like the ones found on audio compact discs. Sounds may be saved as WAV (PCM) files, and such files may also be imported. (Ordered) sets (of real numbers), such as the image of a function like  $t \mapsto \sin \omega t$ , may be converted to sounds, and played using the computer's speakers.

- **Tables**

A table is a two-dimensional array of strings, possibly with per-cell formatting.

- **Logical Values (true or false)**

Valid operators include  $\wedge$ ,  $\vee$ ,  $\underline{\vee}$ , and  $\neg$ .

- **Structures (advanced)**

Structures are objects that contain named members, each of which has a value: either a number, a string, a boolean, or another structure. For instance, the **date** function returns a structure with the members **year**, **month**, **day**, etc. To obtain the value of a particular member, use the colon operator, as in "ans:year". Custom structures can be created by means of the **createStruct** function.

### Keyboard Input

By now you have probably asked yourself how to enter characters that are not on your keyboard. Of course you could use some standard method, such as charmap, but this would be extremely tedious. Rather, in AlgoSim special characters are entered by typing "`\chrname`" in the console. When such a code has been input, followed by a non-alphanumeric character (such as a space, a bracket, a comma, a colon, etc.) it is replaced by the actual character. For instance, to enter  $\cup$ , simply type "`\union`" at the console. Below is a table of all pre-defined codes.

<code>\c</code>	©	<code>\Gamma</code>	Γ	<code>\mu</code>	μ	<code>\Upsilon</code>	Υ
<code>\r</code>	®	<code>\delta</code>	δ	<code>\Mu</code>	Μ	<code>\phi</code>	φ
<code>\tm</code>	™	<code>\Delta</code>	Δ	<code>\nu</code>	ν	<code>\Phi</code>	Φ
<code>\interrobang</code>	‡	<code>\epsilon</code>	ε	<code>\Nu</code>	Ν	<code>\chi</code>	χ
<code>\deg</code>	°	<code>\Epsilon</code>	Ε	<code>\xi</code>	ξ	<code>\Chi</code>	Χ
<code>\alef</code>	ℵ	<code>\zeta</code>	ζ	<code>\Xi</code>	Ξ	<code>\psi</code>	ψ
<code>\numero</code>	№	<code>\Zeta</code>	Ζ	<code>\omicron</code>	ο	<code>\Psi</code>	Ψ
<code>\benzene</code>	⬡	<code>\eta</code>	η	<code>\Omicron</code>	Ο	<code>\omega</code>	ω
<code>\keyboard</code>	⌨	<code>\Eta</code>	Η	<code>\pi</code>	π	<code>\Omega</code>	Ω
<code>\floralheart</code>	☻	<code>\theta</code>	θ	<code>\Pi</code>	Π	<code>\OmegaPi</code>	ω
<code>\h</code>	ℏ	<code>\Theta</code>	Θ	<code>\rho</code>	ρ	<code>\cdot</code>	·
<code>\hbar</code>	ℏ	<code>\iota</code>	ι	<code>\Rho</code>	Ρ	<code>\times</code>	×
<code>\alpha</code>	α	<code>\Iota</code>	Ι	<code>\sigma</code>	σ	<code>\mult</code>	×
<code>\Alpha</code>	Α	<code>\kappa</code>	κ	<code>\Sigma</code>	Σ	<code>\cross</code>	×
<code>\beta</code>	β	<code>\Kappa</code>	Κ	<code>\tau</code>	τ	<code>\minus</code>	−
<code>\Beta</code>	Β	<code>\lambda</code>	λ	<code>\Tau</code>	Τ	<code>\forall</code>	∀
<code>\gamma</code>	γ	<code>\Lambda</code>	Λ	<code>\upsilon</code>	υ	<code>\complement</code>	⊂

<code>\partial</code>	$\partial$	<code>\intersection</code>	$\cap$	<code>&gt;=</code>	$\geq$	<code>\Uarr</code>	$\uparrow$
<code>\exists</code>	$\exists$	<code>\intersect</code>	$\cap$	<code>\muchlessthan</code>	$\ll$	<code>\Rightarrow</code>	$\Rightarrow$
<code>\nexists</code>	$\nexists$	<code>\isect</code>	$\cap$	<code>&lt;&lt;</code>	$\ll$	<code>\Rarr</code>	$\Rightarrow$
<code>\emptyset</code>	$\emptyset$	<code>\union</code>	$\cup$	<code>\muchgreaterthan</code>		<code>\Downarrow</code>	$\Downarrow$
<code>\nabla</code>	$\nabla$	<code>\subset</code>	$\subset$		$\gg$	<code>\Darr</code>	$\Downarrow$
<code>\in</code>	$\in$	<code>\subseteq</code>	$\subseteq$	<code>&gt;&gt;</code>	$\gg$	<code>\Leftrightarrow</code>	$\Leftrightarrow$
<code>\notin</code>	$\notin$	<code>\subsetneq</code>	$\subsetneq$	<code>\cdots</code>	$\dots$	<code>\LRarr</code>	$\Leftrightarrow$
<code>\contains</code>	$\ni$	<code>\superset</code>	$\supset$	<code>\dots</code>	$\dots$	<code>\Updownarrow</code>	$\Updownarrow$
<code>\ncontains</code>	$\not\ni$	<code>\supseteq</code>	$\supseteq$	<code>\approx</code>	$\approx$	<code>\UDarr</code>	$\Updownarrow$
<code>\qed</code>	$\blacksquare$	<code>\supsetneq</code>	$\supsetneq$	<code>\napprox</code>	$\not\approx$	<code>\leftoverrightarrow</code>	
<code>\endofproof</code>	$\blacksquare$	<code>\integral</code>	$\int$	<code>\approx eq</code>	$\approx$		$\Leftrightarrow$
<code>\product</code>	$\prod$	<code>\int</code>	$\int$	<code>\approx lt</code>	$\lesssim$	<code>\lorarr</code>	$\Leftrightarrow$
<code>\sum</code>	$\sum$	<code>\iint</code>	$\iint$	<code>\approx gt</code>	$\gtrsim$	<code>\leftoverrightarrow</code>	$\Leftarrow$
<code>\pm</code>	$\pm$	<code>\iiint</code>	$\iiint$	<code>\permille</code>	$\text{‰}$	<code>n</code>	$\approx$
<code>\mp</code>	$\pm$	<code>\cint</code>	$\oint$	<code>\R</code>	$\mathbb{R}$	<code>\lorhar</code>	$\Leftrightarrow$
<code>\minusplus</code>	$\mp$	<code>\ciint</code>	$\oint$	<code>\Q</code>	$\mathbb{Q}$	<code>\mapsto</code>	$\mapsto$
<code>\mp</code>	$\mp$	<code>\ciiint</code>	$\oint$	<code>\Z</code>	$\mathbb{Z}$	<code>\to</code>	$\mapsto$
<code>\setminus</code>	$\setminus$	<code>\therefore</code>	$\therefore$	<code>\N</code>	$\mathbb{N}$	<code>-&gt;</code>	$\mapsto$
<code>\sqrteroot</code>	$\sqrt{\quad}$	<code>\because</code>	$\because$	<code>\C</code>	$\mathbb{C}$	<code>\lfloor</code>	$\lfloor$
<code>\sqrt</code>	$\sqrt{\quad}$	<code>\assign</code>	$\equiv$	<code>\H</code>	$\mathbb{H}$	<code>\lceil</code>	$\lceil$
<code>\infty</code>	$\infty$	<code>:=</code>	$\equiv$	<code>\leftarrow</code>	$\leftarrow$	<code>\rfloor</code>	$\rfloor$
<code>\inf</code>	$\infty$	<code>\definition</code>	$\equiv$	<code>\larr</code>	$\leftarrow$	<code>\rf</code>	$\rfloor$
<code>\proportionalto</code>	$\propto$	<code>\def</code>	$\equiv$	<code>\uparrow</code>	$\uparrow$	<code>\lceil</code>	$\lceil$
<code>\proportional</code>	$\propto$	<code>\eqdef</code>	$\equiv$	<code>\uarr</code>	$\uparrow$	<code>\rc</code>	$\rfloor$
<code>\prop</code>	$\propto$	<code>\req</code>	$\approx$	<code>\rightarrow</code>	$\rightarrow$	<code>\rceil</code>	$\rfloor$
<code>\rightangle</code>	$\angle$	<code>\notequalto</code>	$\neq$	<code>\rarr</code>	$\rightarrow$	<code>\dot</code>	$\cdot$
<code>\angle</code>	$\angle$	<code>\notequal</code>	$\neq$	<code>\downarrow</code>	$\downarrow$	<code>\ortho</code>	$\perp$
<code>\parallelto</code>	$\parallel$	<code>\ne</code>	$\neq$	<code>\darr</code>	$\downarrow$	<code>\vect</code>	$\langle$
<code>\parallel</code>	$\parallel$	<code>&lt;&gt;</code>	$\neq$	<code>\leftleftrightarrow</code>	$\leftrightarrow$	<code>\lv</code>	$\langle$
<code>\nparallelto</code>	$\not\parallel$	<code>!=</code>	$\neq$	<code>\rarr</code>	$\leftrightarrow$	<code>\rvect</code>	$\rangle$
<code>\nparallel</code>	$\not\parallel$	<code>\leq</code>	$\leq$	<code>\updownarrow</code>	$\updownarrow$	<code>\rv</code>	$\rangle$
<code>\and</code>	$\wedge$	<code>\le</code>	$\leq$	<code>\udarr</code>	$\updownarrow$	<code>\vect</code>	$\langle$
<code>\or</code>	$\vee$	<code>&lt;=</code>	$\leq$	<code>\Leftarrow</code>	$\leftarrow$	<code>\v</code>	$\langle$
<code>\xor</code>	$\vee$	<code>\geq</code>	$\geq$	<code>\Larr</code>	$\leftarrow$	<code>\v</code>	$\langle$
<code>\not</code>	$\neg$	<code>\ge</code>	$\geq$	<code>\Uparrow</code>	$\uparrow$	<code>\uparr</code>	$\uparrow$

\oplus      ⊕      |      =:      =

The symbols  $\cdot$  and  $\times$  are so common that they can be entered with the "\*" key on the keyboard. If you press this key once,  $\cdot$  is inserted. Twice and  $\times$  is inserted, and the third time  $*$  is inserted. In addition, the first time you press the "-" key, the **binary** minus sign  $-$  is inserted, and the second time the **unary** minus sign  $-$  is inserted.

Furthermore, vector brackets  $\langle \rangle$  are inserted by Ctrl+E, set brackets  $\{ \}$  by Ctrl+S and interval brackets  $[ ]$  with Ctrl+I. Ceiling brackets (round up)  $\lceil \rceil$  are inserted by Ctrl+U, and floor brackets (round down)  $\lfloor \rfloor$  by Ctrl+D. The exponentiation operator  $\uparrow$  is inserted by Ctrl+R.

### Variables and Functions

Two very important concepts are the *variable* and the *function*. To declare a variable named MyVar and assign the value MyVal to it, simply write  $\text{MyVar} := \text{MyVal}$ , such as  $\phi := 2 \cdot \pi$ . The character "==" is inserted by typing ":" followed by "=". To declare a function, use the binary function creator operator  $\mapsto$ , which takes a comma-separated string of independent variables (function arguments) and an expression (also a string) in these variables, and assign the function to an identifier. A function may accept any type of AlgoSim variable, and may return a value of any type. The types need not be specified in advanced, and the very same function may work with inputs of different types, as well as it may output data of different types depending on the input. For instance,

```
double = "x" ↦ "2·x"
double(100)
double((3, 1, 4))
double("test")
```

will return 200, (6, 2, 8), and "testtest", respectively. The following declares a function in two variables:

```
f = "x, y" ↦ "3·x + 5·y"
```

### The Exponentiation Operator

To write very large and very small numbers, the exponentiation operator  $\uparrow$  (inserted by Ctrl+R) is very handy. The exponentiation operator is an infix operator, and  $a \uparrow b$  is exactly equivalent to  $a \cdot 10^b$ . For instance, the rest mass of an electron is  $9.10938215 \uparrow -31$  and the mass of the sun is  $1.9891 \uparrow 30$ .

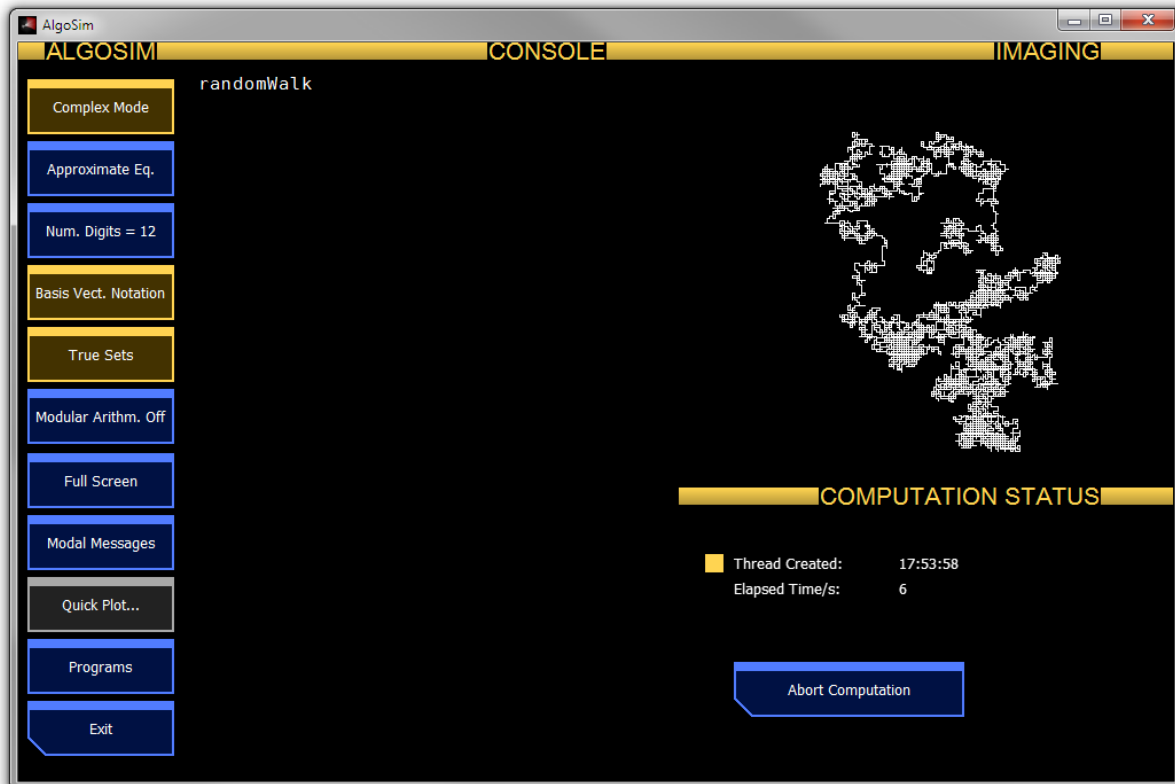
### Base-N Calculations

You can use the # infix operator to enter non-negative integers in any number base, such as binary (base 2), octal (base 8), and hexadecimal (base 16), rather than in decimal (base 10), as usual. The base- $N$  digits used are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, ..., X, Y, Z. Hence you can only work with bases less than or equal to 36. For example,  $567 \# 10$ ,  $\text{FF} \# 16$ ,  $1000 \# 2$ ,  $43\text{HA}2 \# 20$ , and  $43\text{HA}2 \# 16$  will return 567, 255, 8, 671002, and the error message "The base- $N$  digit H is not used in base 16", respectively.

To write a given decimal number in base- $N$  notation, use the function **toBaseN**, which takes a non-negative integer and a base as arguments. For instance,  $\text{toBaseN}(255, 16)$  will return "FF".

### Aborting a Slow Procedure

Some calculations take very long time to complete, of course. You can abort the current computation by pressing the "Abort Computation" button. This will stop the current thread, so you can input a new command.



The following screenshots illustrate how elementary calculations are performed in AlgoSim. Refer to the Reference section of this document for the details regarding each used function.

```

AlgoSim
CONSOLE

4 + 5! - sin(4·π/7)
123.025072088

i^i
0.207879576351

arccos(2) + ln(sin(i))
0.161439361571 + 0.25383842987·i

factor(852497)
852497 = 71·12007

isPrime(199231)
False

gcd(72, 96)
24

lcm(72, 96)
288

count([1, 100], "n", "isPrime(n)")
25

filter([1, 100], "n", "isPrime(n)")
{ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47 }

[3.6] + [-2.6]
1

count([1, 100], "n", "isPrime(n) ^ isPrime(n + 2)")
8

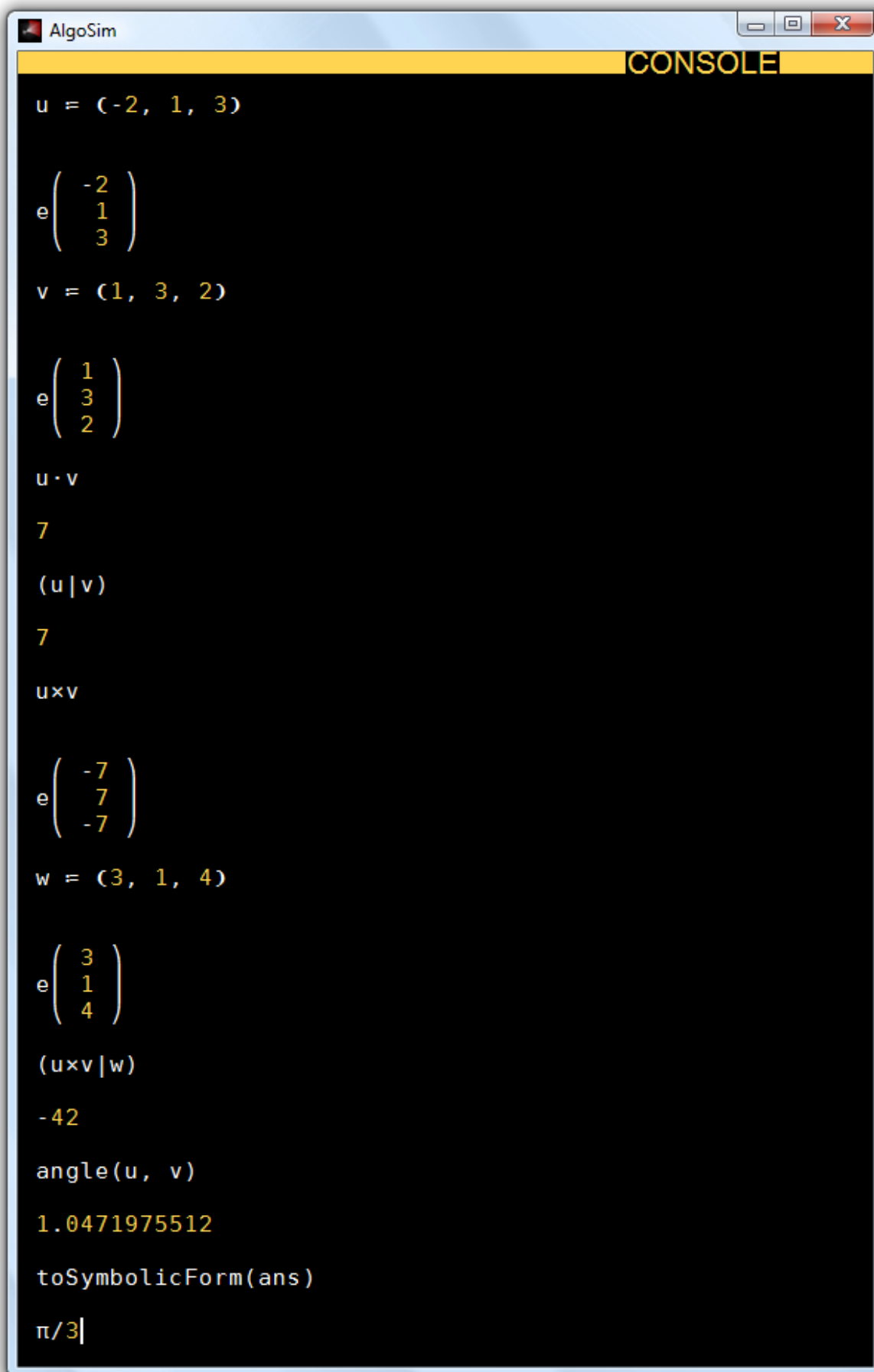
"Yes, " + 5·"yes, " + "yes!"
Yes, yes, yes, yes, yes, yes, yes!
    
```

The screenshot shows a window titled "AlgoSim" with a "CONSOLE" header. The console displays the following text:

```

integrate("sin(x)", "x", 0, pi)
2
sum("1/n!", "n", 0, 1000)
2.71828182846
solve("sinc(x) = 0", "x", 6)
6.28318530718
toSymbolicForm(ans)
2pi
A = {1, 2, 3, 4}
{ 1, 2, 3, 4 }
B = {3, 4, 5, 6}
{ 3, 4, 5, 6 }
A u B
{ 1, 2, 3, 4, 5, 6 }
A n B
{ 3, 4 }
A x B
{ (1, 3), (1, 4), (1, 5), (1, 6), (2, 3), (2, 4), (2, 5),
(3, 5) }
(3, 5) in AxB
True
(5, 3) in AxB
False
[0, 1, 0.1]^2
{ (0, 0), (0, 0.1), (0, 0.2), (0, 0.3), (0, 0.4), (0, 0.5),
(0, 0.6), (0, 0.7), (0, 0.8), (0, 0.9), (0, 1) }

```



The screenshot shows a window titled "AlgoSim" with a "CONSOLE" header. The console displays the following text:

```
u = (-2, 1, 3)
e  $\begin{pmatrix} -2 \\ 1 \\ 3 \end{pmatrix}$ 
v = (1, 3, 2)
e  $\begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix}$ 
u · v
7
(u | v)
7
u × v
e  $\begin{pmatrix} -7 \\ 7 \\ -7 \end{pmatrix}$ 
w = (3, 1, 4)
e  $\begin{pmatrix} 3 \\ 1 \\ 4 \end{pmatrix}$ 
(u × v | w)
-42
angle(u, v)
1.0471975512
toSymbolicForm(ans)
 $\pi/3$ 
```



The screenshot shows the AlgoSim software interface with a console window. The console displays the following operations and results:

```

AlgoSim
CONSOLE
A = ((1, 2, 0), (0, 1, 0), (-1, 0, 1))
( 1  2  0 )
( 0  1  0 )
(-1  0  1 )
det(A)
1
B = (3, 1, 2)
e ( 3 )
  ( 1 )
  ( 2 )
A^(-1) * B
e ( 1 )
  ( 1 )
  ( 3 )
sysSolve(A, B)
e ( 1 )
  ( 1 )
  ( 3 )
augment(A, B)
( 1  2  0  3 )
( 0  1  0  1 )
(-1  0  1  2 )
sysSolve(ans)
e ( 1 )
  ( 1 )
  ( 3 )
;; As shown above, there are several ways of solving
;; linear equation systems.

```

The screenshot shows a window titled "AlgoSim" with a "CONSOLE" header. The console displays the following text and mathematical expressions:

```

A

$$\begin{pmatrix} 1 & 2 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix}$$

toEchelonForm(A)

$$\begin{pmatrix} 1 & 2 & 0 \\ 0 & 1 & 0.5 \\ 0 & 0 & 1 \end{pmatrix}$$

v = (1, 0, 2)

e  $\begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}$ 
A · v

e  $\begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$ 
A · v ⊥ v
False
tr(A)
3
norm(v)
2.2360679775
taxiNorm(v)
3
maxNorm(v)
2

```

**Automatic “Ans” Argument (AAA)**

Some AlgoSim functions require no argument, such as **date**, **time**, and **exit**. But in AlgoSim, all functions must be called with a list of arguments, to differentiate functions from variables. Hence – in principle – one must add a dummy argument when calling such functions, as in

```
date(0)
```

However, this is a bit tedious. To overcome this problem, AlgoSim will automatically add the argument “ans” to all “simple” (see “Technically” below) function calls lacking argument. Thus you may write simply

```
date
```

The fact that ans is added, and not (for instance) 0, also makes it possible to apply a function to the latest output without adding any arguments at all, as in the following examples.

```
arccos(1/2)
1.0471975512
toSymbolicForm
π/3
arcsin(0.23)
0.232077682863
sqrt
0.481744416535
date
year: 2010
month: 6
day: 23
weekOfYear: 25
dayOfYear: 174
dayOfWeek: 3
```

Technically, if the command entered in the console

- is a valid identifier, and
- is not equal to the identifier (name) of a previously declared variable,

then

- the string “(ans)” will be appended to the command, before it is executed by the kernel.

### **The Semicolon Operator**

In AlgoSim, the semicolon is a binary infix operator that returns the last operand. For instance, "5; 6; 2; 3" will return 3. This makes the operator ideal for evaluating several expressions on a single line. For instance,

```
a = 1; b = 2; c = 3
```

will assign three variables on a single line of code.

## Visualisation

The perhaps most important feature of AlgoSim is its advanced capabilities when it comes to visualisation of data. In this section we will discuss all major approaches of two- and three-dimensional visualisation. The general idea is first to create a set, and then draw it. This separation in two steps makes the visualisation capabilities much more powerful, as a set may be drawn using the same methods independently on how the set was created.

### 2D graphs

A 2D graph is a set  $\{(x, y) \in \mathbb{R}^2: y = f(x), x \in D_f\}$  associated with a function  $f$  in a domain  $D_f$ . The graph is created by **createGraph** and is drawn by, for instance, **drawSet**. For example,

```
graph = createGraph("sin(x)", "x", [-10, 10, 0.001])
```

creates the graph of the function  $x \mapsto \sin x$  where  $x \in [-10, 10]$ . Because a set (in the computer's memory) cannot contain an infinite number of points, we must specify the resolution of the points in  $[-10, 10]$ . In this case we choose 0.001 so that the domain becomes

$$\{-10, -9.999, -9.998, -9.997, -9.996, -9.995, -9.994, \dots\}.$$

In general, when it comes to simple curves, 0.001 is a good resolution.

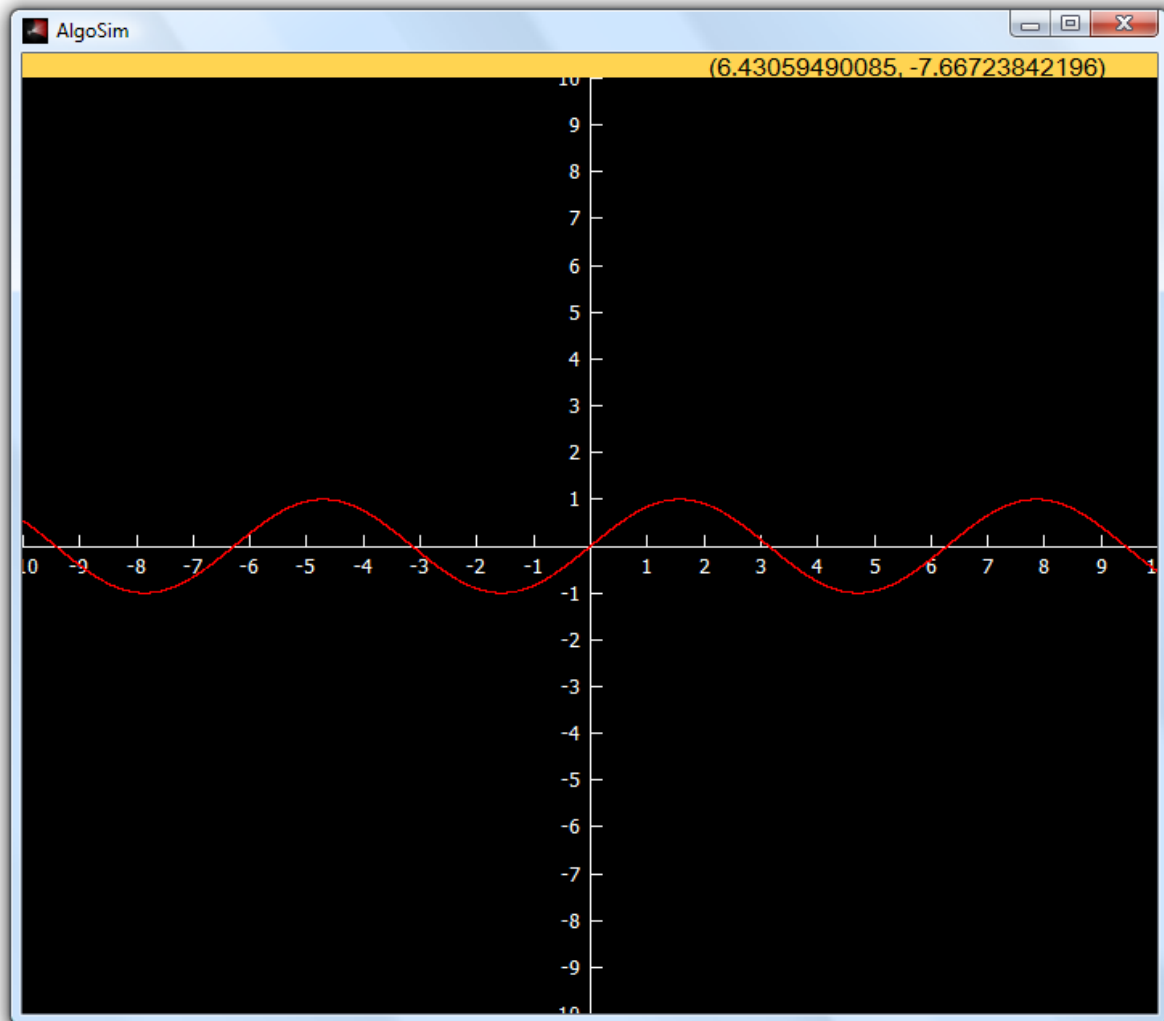
Now **graph** is the graph of the sine curve in this interval. This set can be drawn by

```
drawSet("graph")
```

To make things a bit more interesting, we can also draw the axes, by using **drawAxes(0)**. We can also write

```
drawSet("graph", "colour:red")
```

(instead of the line above) to let the graph be red. The result is shown in the screenshot below.



Notice that the coordinates of the cursor are shown in the orange panel. You can use the mouse (drag) to move the plane and you can use the scroll wheel to zoom in or out, indefinitely. If you zoom in enough, however, you will notice that the resolution 0.001 will become insufficient to make a solid curve. This can be compensated by using a higher resolution, or by using **drawLines** instead of **drawSet**. **drawLines** works exactly like **drawSet**, except that straight lines are drawn connecting the discrete points in the set. In fact, it is good practice *always* to use **drawLines** instead of **drawSet**, because you then can lower the resolution (and hence making the computation must less intensive), often without any noticeable effect.

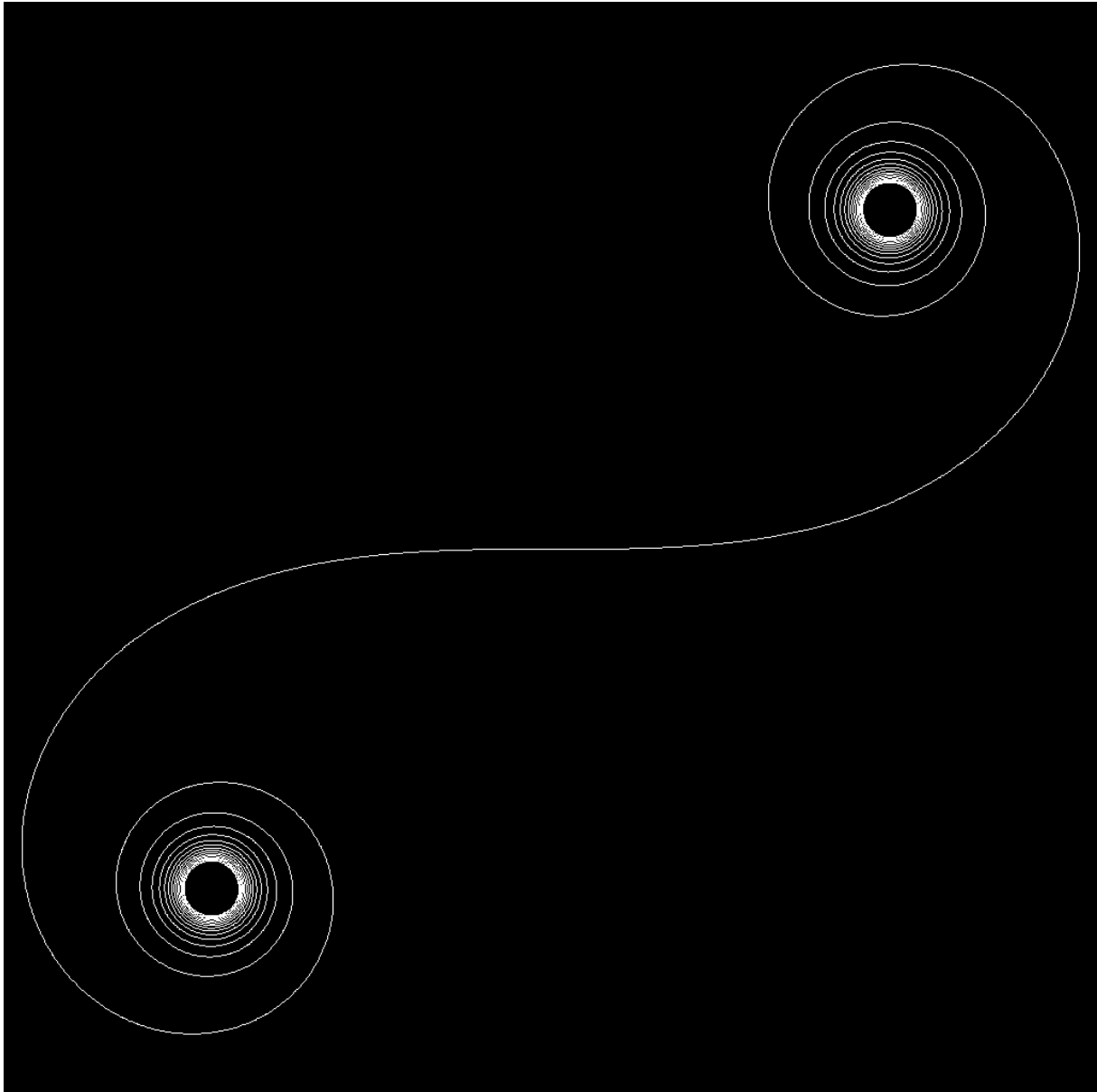
### 2D parametrised curves

Of course not all 2D curves can be written in the form  $y = f(x)$ . Rather, a general planar curve is the *image* of a parametrisation function  $t \mapsto F(t) = (x(t), y(t))$  in a domain  $D \ni t$ . In other words, the curve is  $F(D)$ . In AlgoSim, the image of a set under a function is created by the extremely fundamental function **createImage**. The output of this function is – of course – a set and may be rendered using either **drawSet** or **drawLines**. As an example, let us draw the unit circle.

```
clearView(0)
circle = createImage("(sin(θ), cos(θ))", "θ", [0, 2·π, 0.001])
drawLines("circle")
```

A slightly more interesting example is the Euler spiral.

```
clearView(0)
spiral = createImage("(FresnelC(t), FresnelS(t))", "t", [-10,
                10, 0.02])
drawLines("spiral")
```

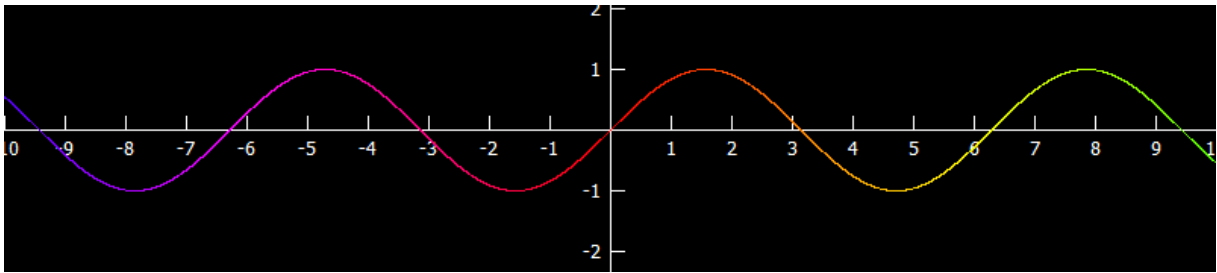


### Coloured Sets

Now is the time to wonder if it is possible to create sets where each pixel has its own colour. This is very straight-forward in AlgoSim: to create a planar, coloured, curve, simply create a set  $\{(x, y, c)\}$  instead of a set  $\{(x, y)\}$  where  $c$  is a colour code. To create a colour code, use the **rgb** or **hsv** functions, which take the three RGB or HSV coordinates of the colour as arguments, respectively. Coloured sets are drawn with the functions **drawColouredSet** and **drawColouredLines**, as one might expect.

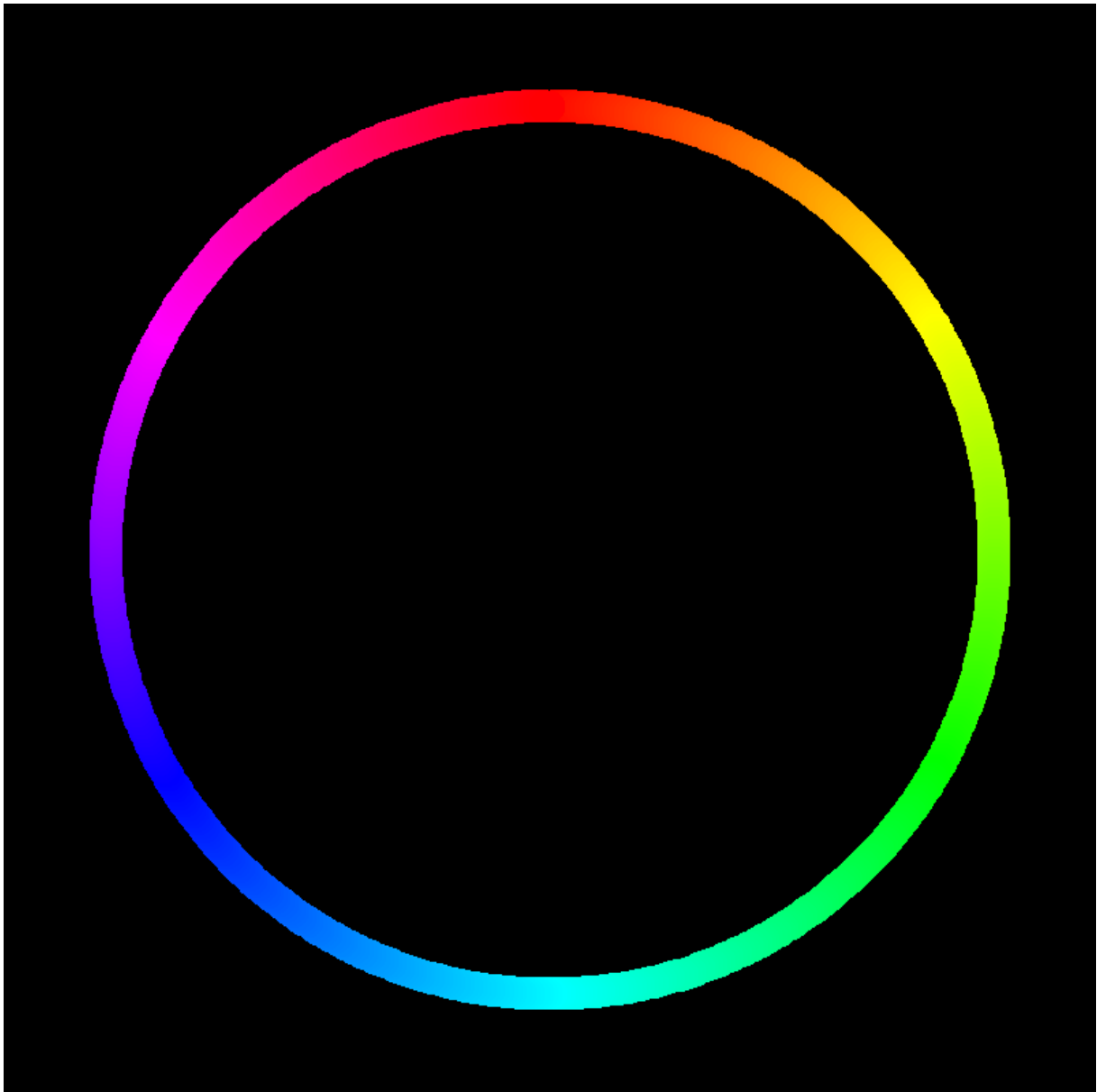
For example, let us create a coloured sine curve, where the hue of the pixel colour is a function of the  $x$ -coordinate:

```
sine = createImage("(x, sin(x), hsv(10·x, 1, 1))", "x", [-10,
                10, 0.001])
drawColouredLines("sine")
```



Or why not draw a thick, coloured unit circle?

```
circle = createImage("(sin(θ), cos(θ), hsv(360·θ/(2·π)), 1, 1)",  
                    "θ", [0, 2·π, 0.01])  
drawColouredLines("circle", "width:24")
```



That was fun.

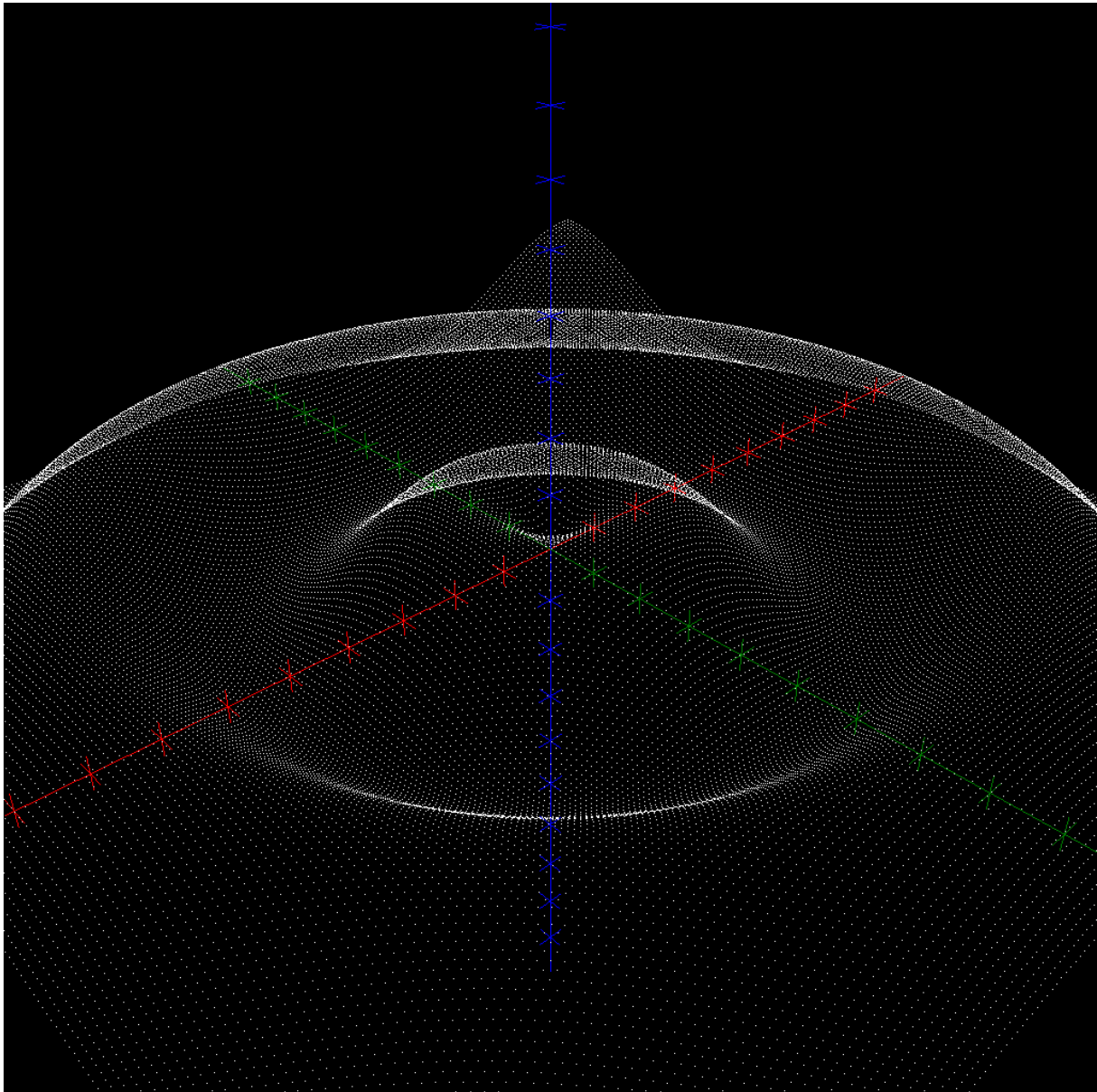


**3D surfaces**

A 3D graph is a set  $\{(x, y, z) \in \mathbb{R}^3: z = f(x, y), (x, y) \in D_f\}$  associated with a function  $f$  in a domain  $D_f$ . The graph is created by **createGraph3** and is drawn by, for instance, **drawSet3**. For example,

```
sine = createGraph3("sin(sqrt(x^2+y^2))", "x, y", [-10, 10,
0.1]^2)
drawAxes3(0)
drawSet3("sine")
```

Of course drawSet3 could also be instructed to use a specific colour, point size, etc, as in the two-dimensional case with drawSet. Also, a more general surface can be obtained by createImage instead of createGraph3.



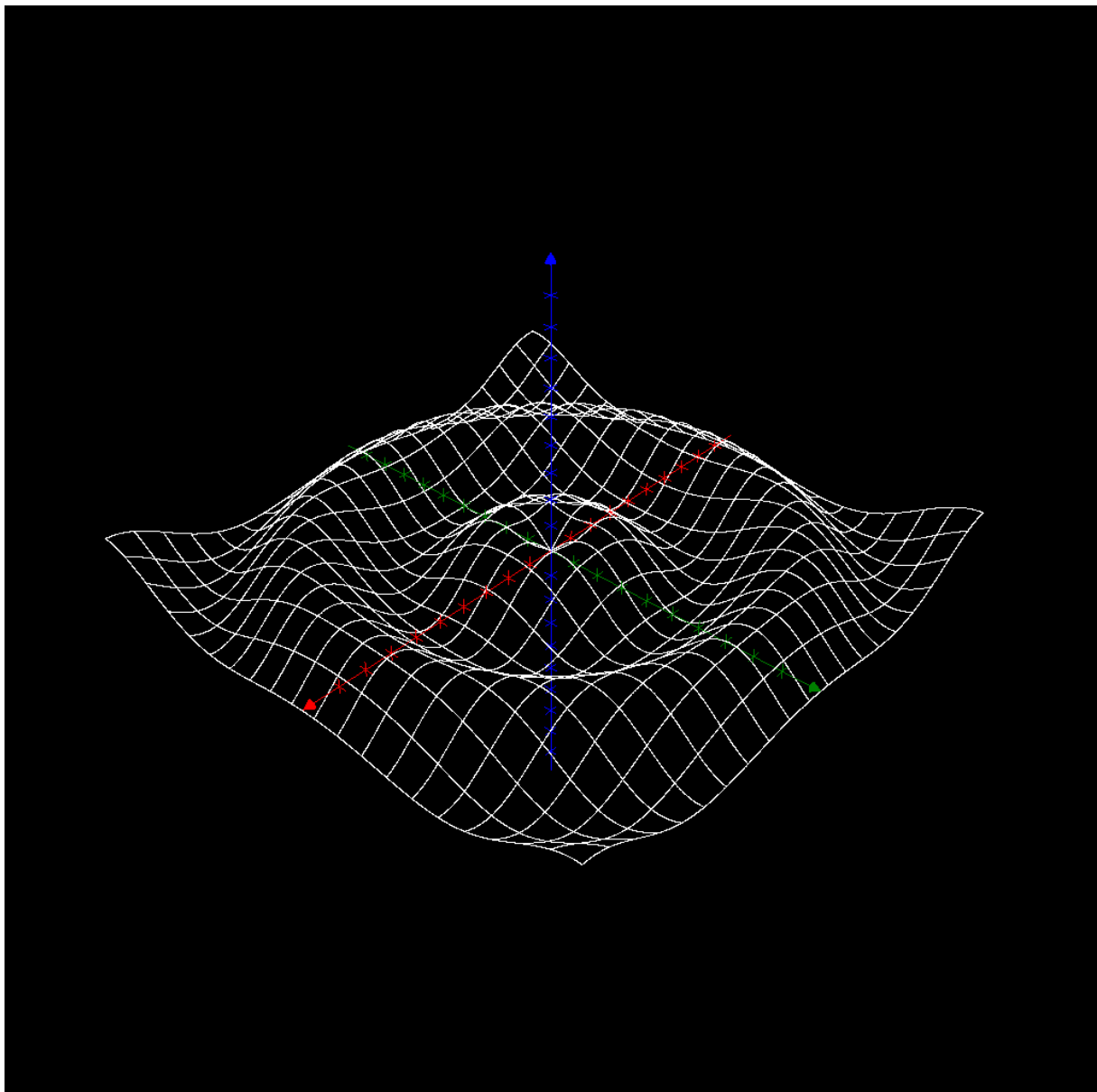
As you can see, even though the surface contains 40 000 points (try

```
contents(sine)
```

to see this), it looks highly ... "hollow". But even worse, if we increase the number of points so that it will become opaque at this magnification, the computation will not only take too long time to complete, but the entire surface will become white! We will only see the silhouette of the surface. A much better approach is to draw only the parameter curves of the surface. To this end, we change the domain from the filled square  $[-10, 10]^2$  in the parameter plane to a *net*. This is quite clever, isn't it?

```
clearView3(0)
net = createNet(-10, 10, 0.01, 1, -10, 10, 0.01, 1)
sine = createGraph3("sin(sqrt(x^2+y^2))", "x, y", net)
drawAxes3(0)
drawSet3("sine")
```

The result looks much better, don't you think?



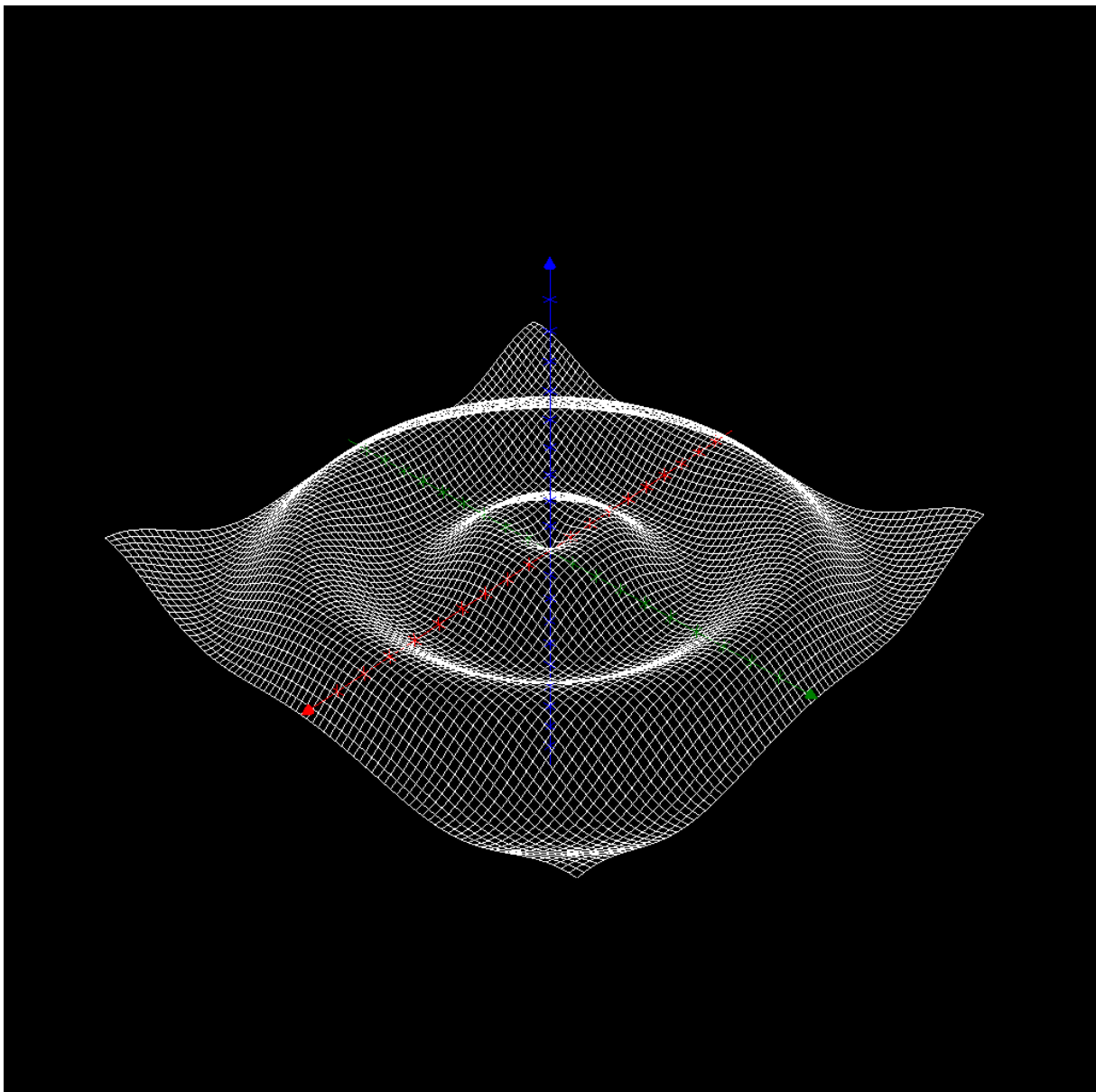
But, as you might have noticed, it took quite some time to compute the surface. And no wonder:

```
contents(sine)
```

returns "84042 real vector(s)". The procedure would be much faster if we only computed the value of the function at some points, and then connected the parameter curves with straight lines (cf. drawSet vs. drawLines).

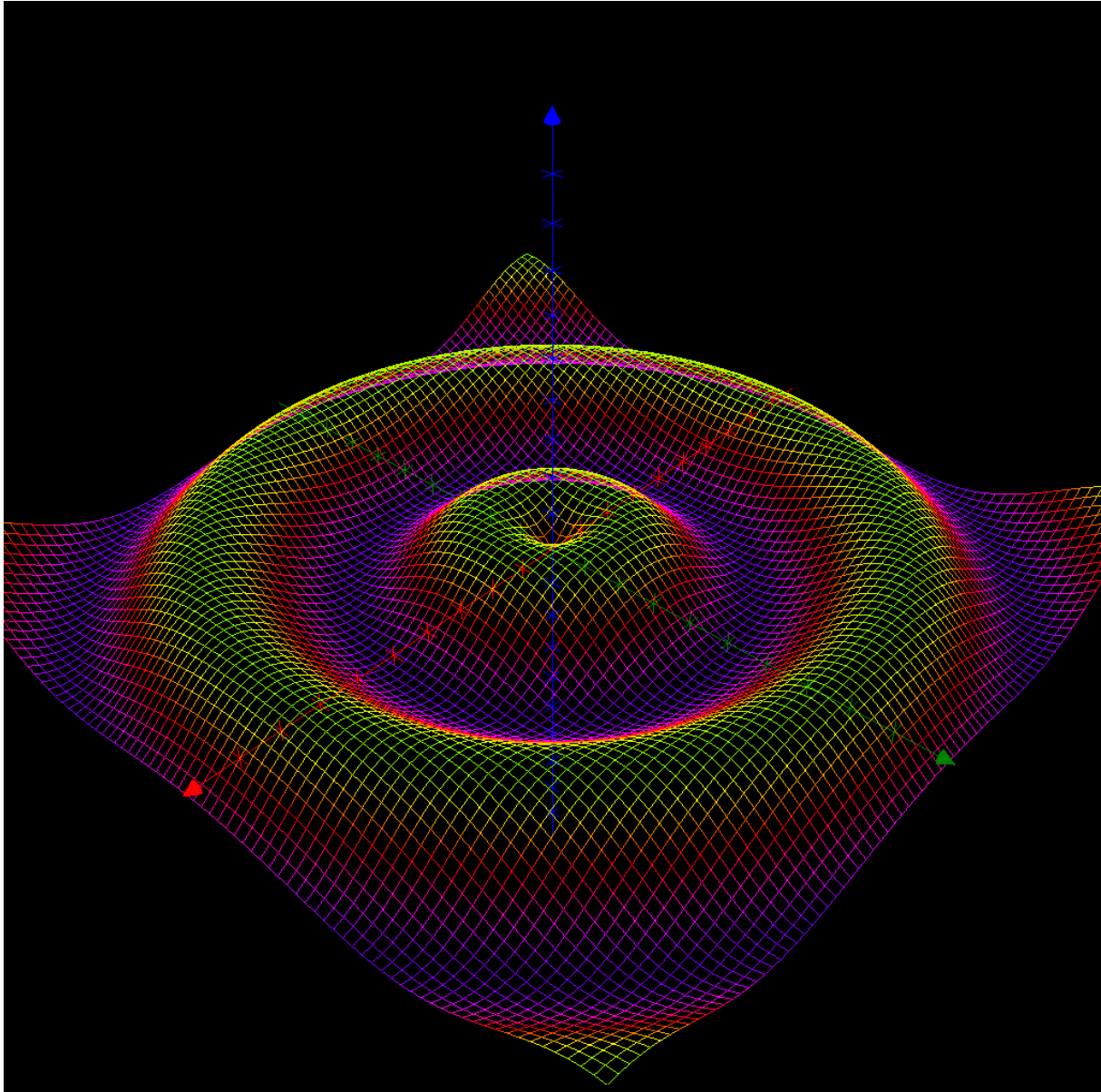
In AlgoSim, the way of drawing the parameter curves of a surface is to use the special functions **createSurfParamCurves** and **drawSurfParamCurves**. As an example:

```
clearView3(0)
sine = createSurfParamCurves("(x, y, sin(sqrt(x^2+y^2)))", "x,
                             y", -10, 10, -10, 10)
drawAxes3(0)
drawSurfParamCurves("sine")
```



This looks really good, and it was fast. Of course, there are also functions **drawColouredSet3** and **drawColouredSurfParamCurves**. To use this, simply let create a set of four-dimensional vectors, the fourth component being the colour code of each pixel.

```
clearView3(0)
sine = createSurfParamCurves("(x, y, sin(sqrt(x^2+y^2)),
                             hsv(180·sin(sqrt(x^2+y^2)), 1, 1))", "x, y", -10, 10,
                             -10, 10)
drawAxes3(0)
drawSurfParamCurves("sine")
```



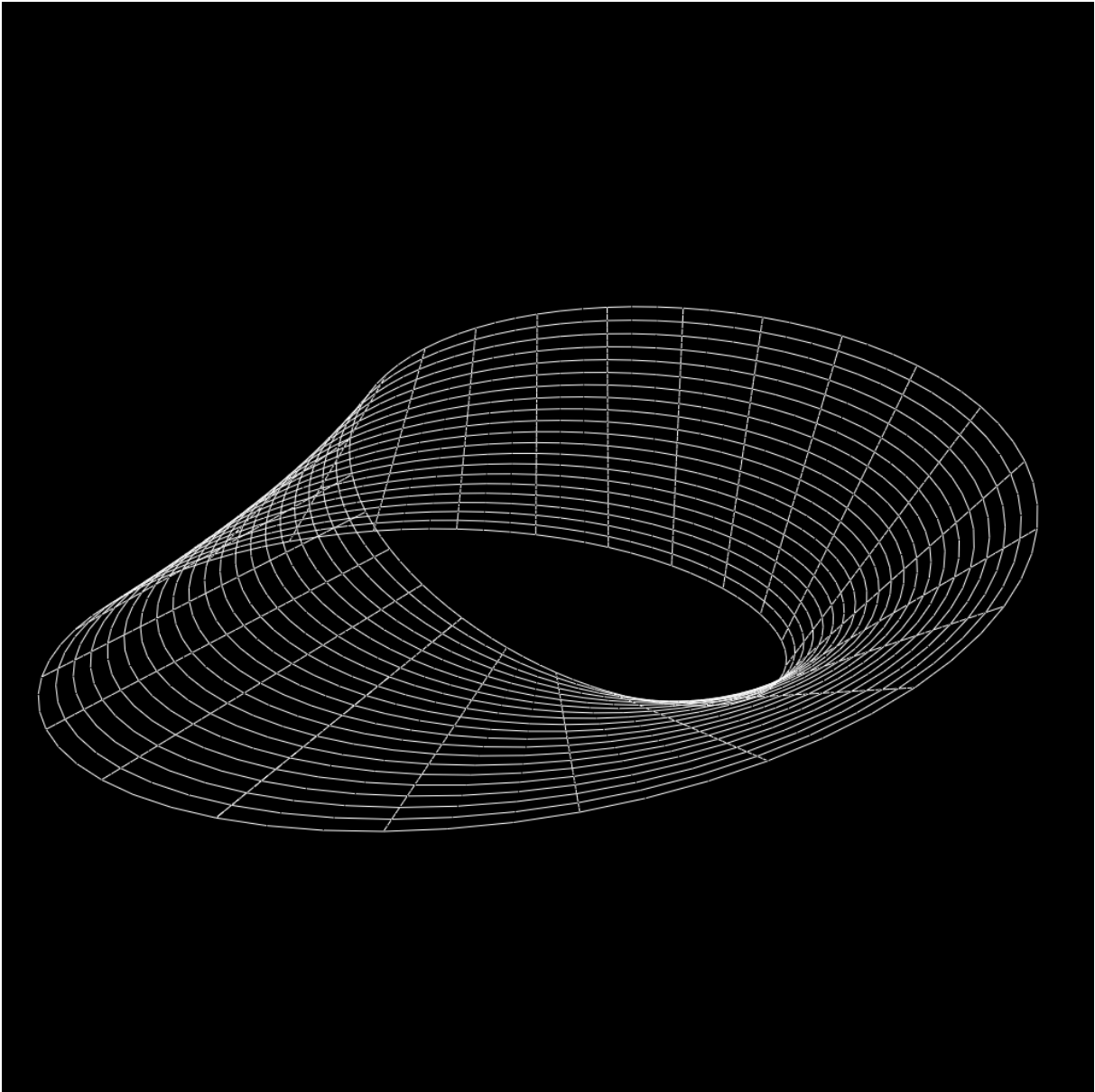
As an example of a 3D surface that is not a graph, we give the Möbius strip.

```
clearView3(1)
Möbius = createSurfParamCurves("5·((1 +
                                0.5·v·cos(0.5·u))·cos(u), (1 +
```

```

0.5·v·cos(0.5·u))·sin(u), 0.5·v·sin(0.5·u))", "u, v",
0, 2·π, π/36, π/12, -1, 1.01, 0.05, 0.1)
drawSurfParamCurves("Möbius")

```



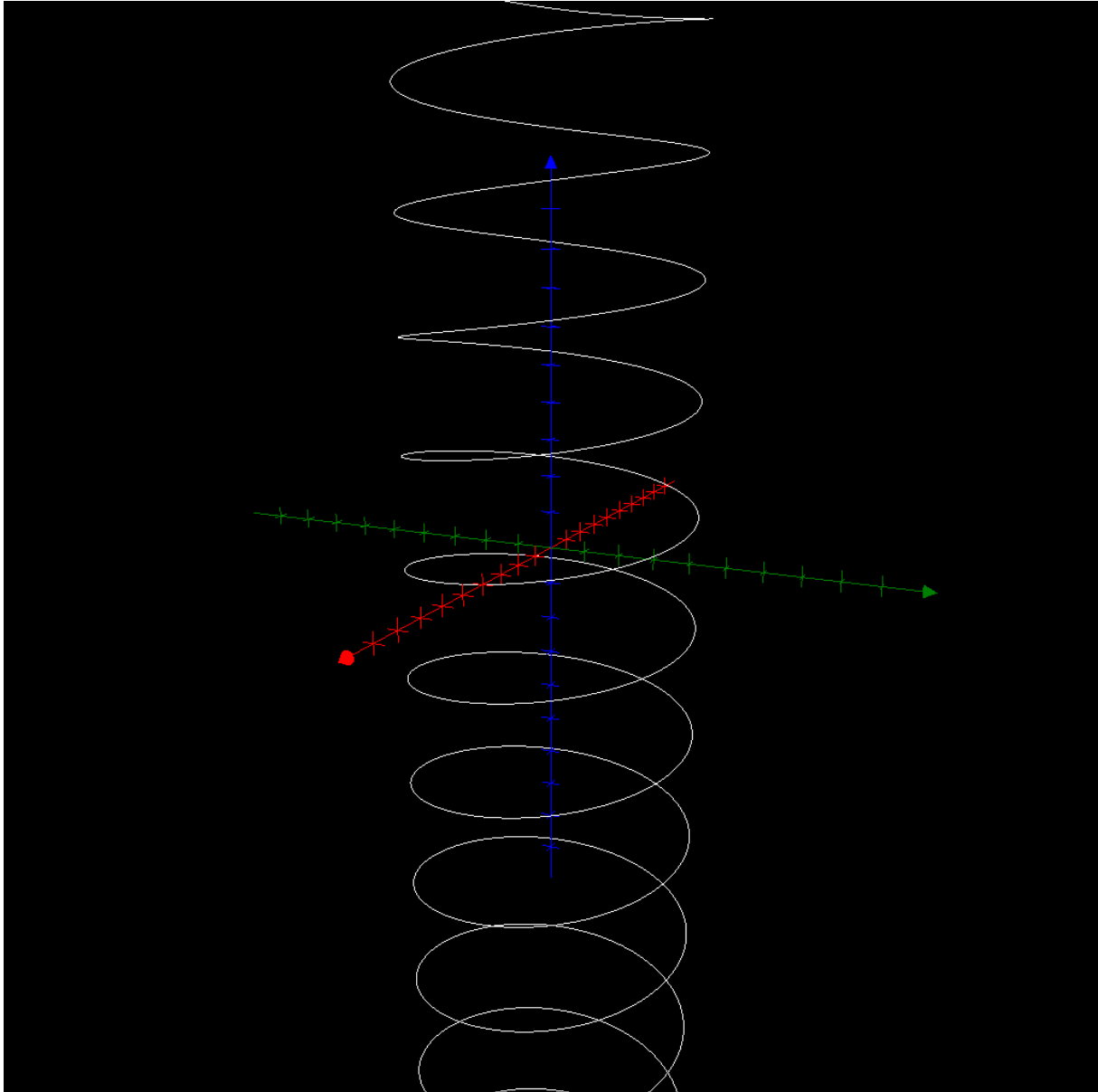
### 3D curves

3D curves are not graphs, of course. Hence a three-dimensional curve must be given by parametrisation, i.e. the curve is the image of a one-dimensional domain under a three-dimensional, vector-valued, function. As an example, consider the circular helix.

```

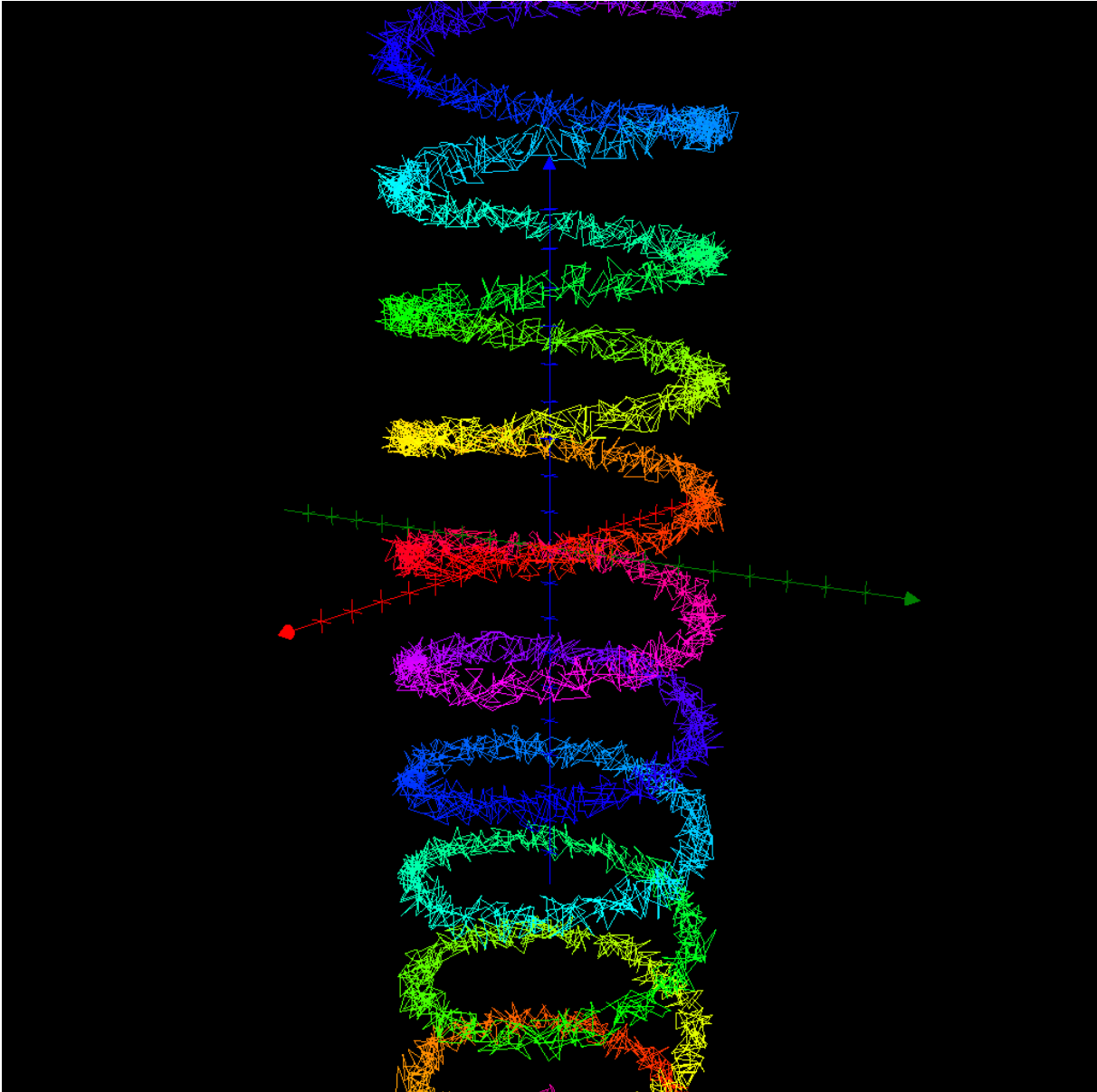
clearView3(0)
helix = createImage("(4·cos(φ), 4·sin(φ), φ/2)", "φ", [-50, 50,
0.01])
drawAxes3(0)
drawLines3("helix")

```



To make this a bit more interesting, we add some fuzz and colour.

```
clearView3(0)
helix = createImage("(4*cos(φ) + randomReal(1), 4*sin(φ) +
    randomReal(1), φ/2 + randomReal(1), hsv(10*φ, 1,
    1))", "φ", [-50, 50, 0.01])
drawAxes3(0)
drawColouredLines3("helix")
```



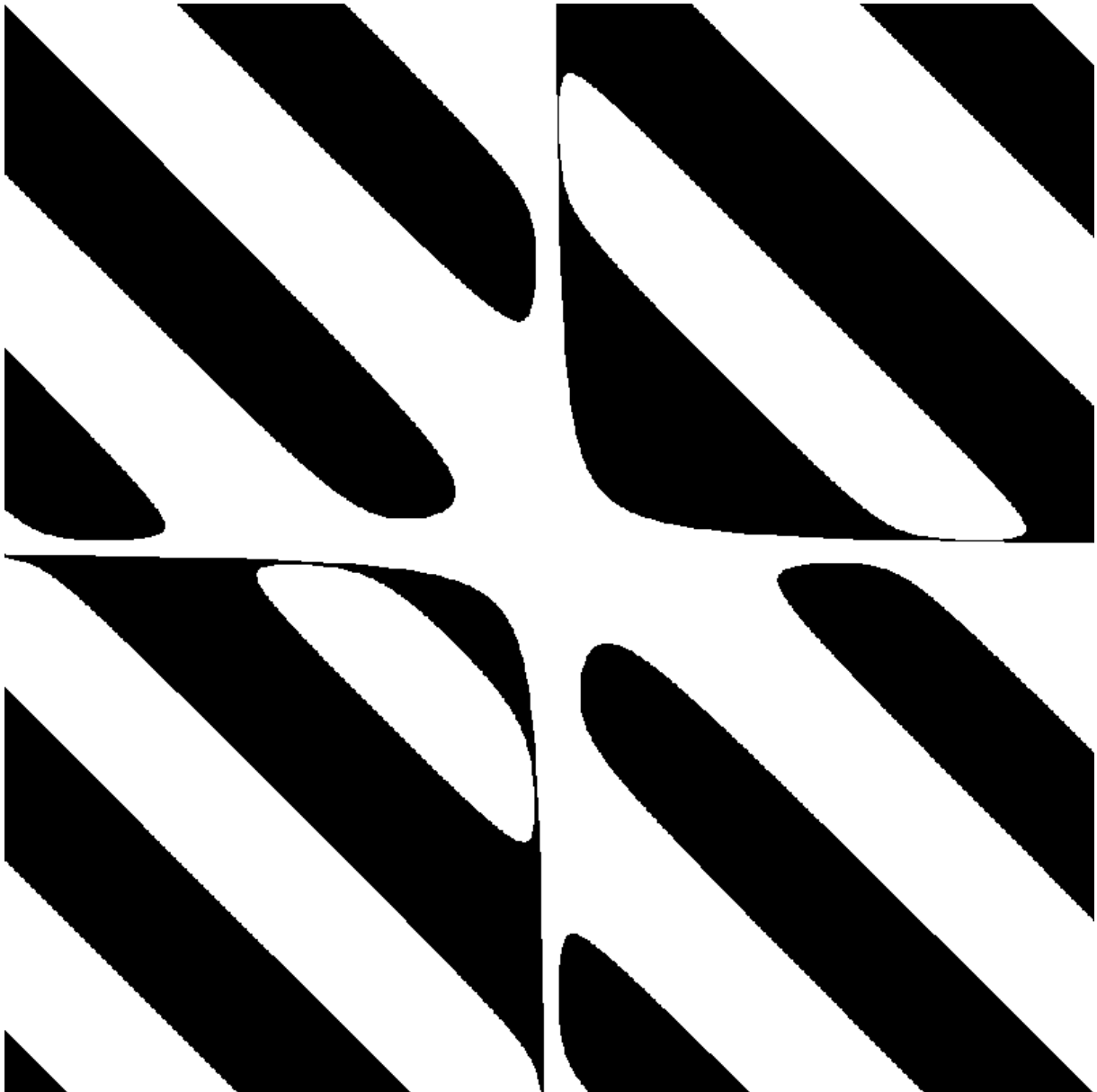
### Implicit Sets

Some sets cannot even be parametrised. But even so, AlgoSim is able to create them. Implicit plotting means that you create a set of all points that satisfy a condition, usually an equation or inequality, in the spatial coordinates. Say you want to plot

$$S = \{(x, y) \in \mathbb{R}^2 : \operatorname{arccoth} xy < \sin(x + y)\}.$$

This set is not possible to parameterise using any functions of self-respect. But AlgoSim can use a brute-force iteration over a given rectangle in  $\mathbb{R}^2$  and using a given resolution, to find points in  $S$ . The function we need is **createSet**. Let us try this.

```
clearView(0)
S = createSet("arccoth(x·y) < sin(x+y)")
drawSet("S")
```



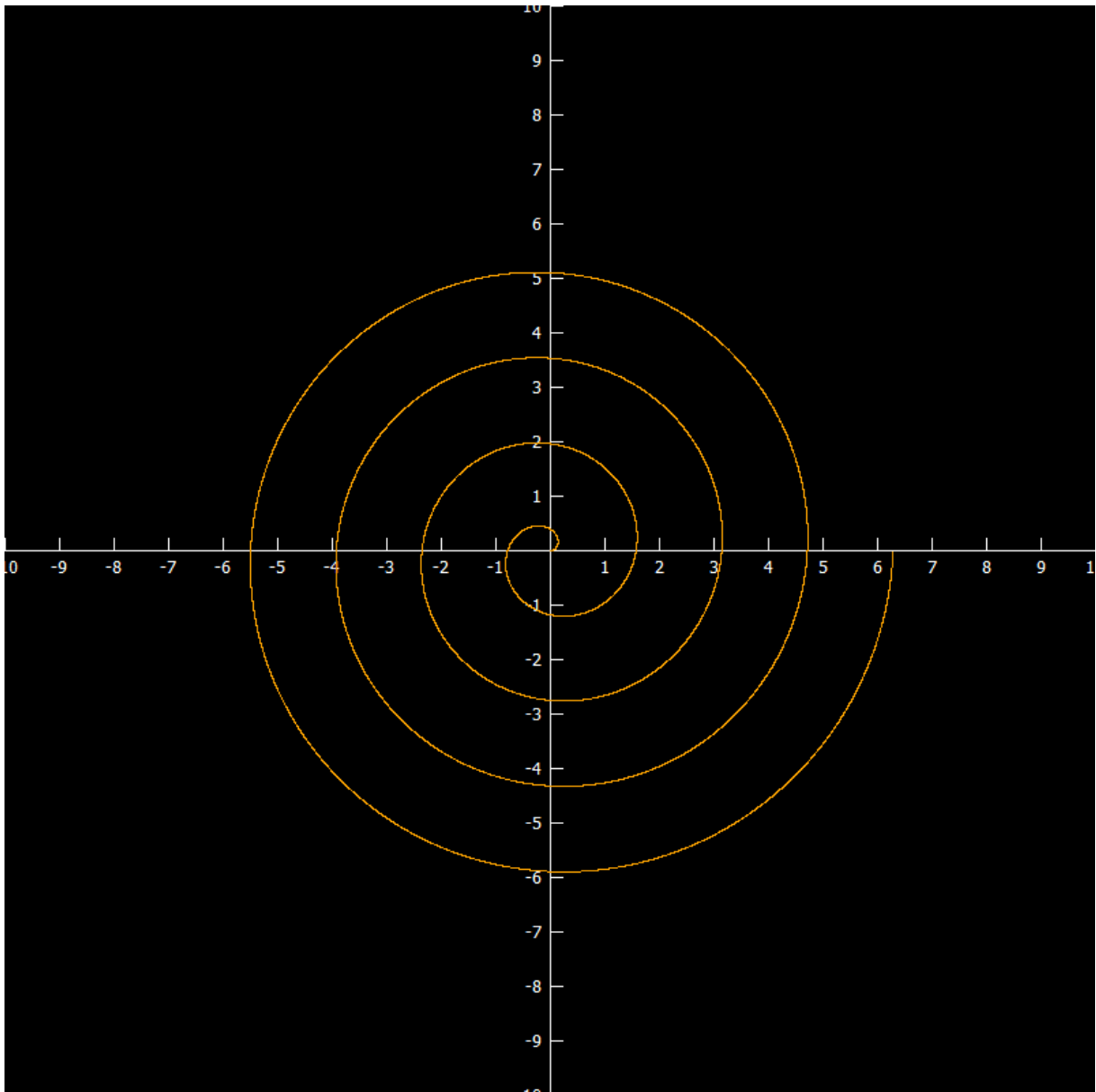
### Non-Cartesian Coordinate Systems

AlgoSim can also visualise data using non-Cartesian coordinate systems. The procedure is rather simple. First create a set of points in the coordinate system of your choice, for instance a set of points  $(r, \phi)$  in polar coordinates. Then transform this set to the corresponding set of Cartesian coordinates, and then plot it.

As an illustration, we can draw an Archimedean spiral  $r = \phi/4$  in plane polar coordinates.

```
clearView(0)
spiral = createImage("(phi/4, phi)", "phi", [0, 8*pi, 0.001])
spiral = polarCoords(spiral)
drawAxes(0)
drawLines("spiral", "colour:orange")
```



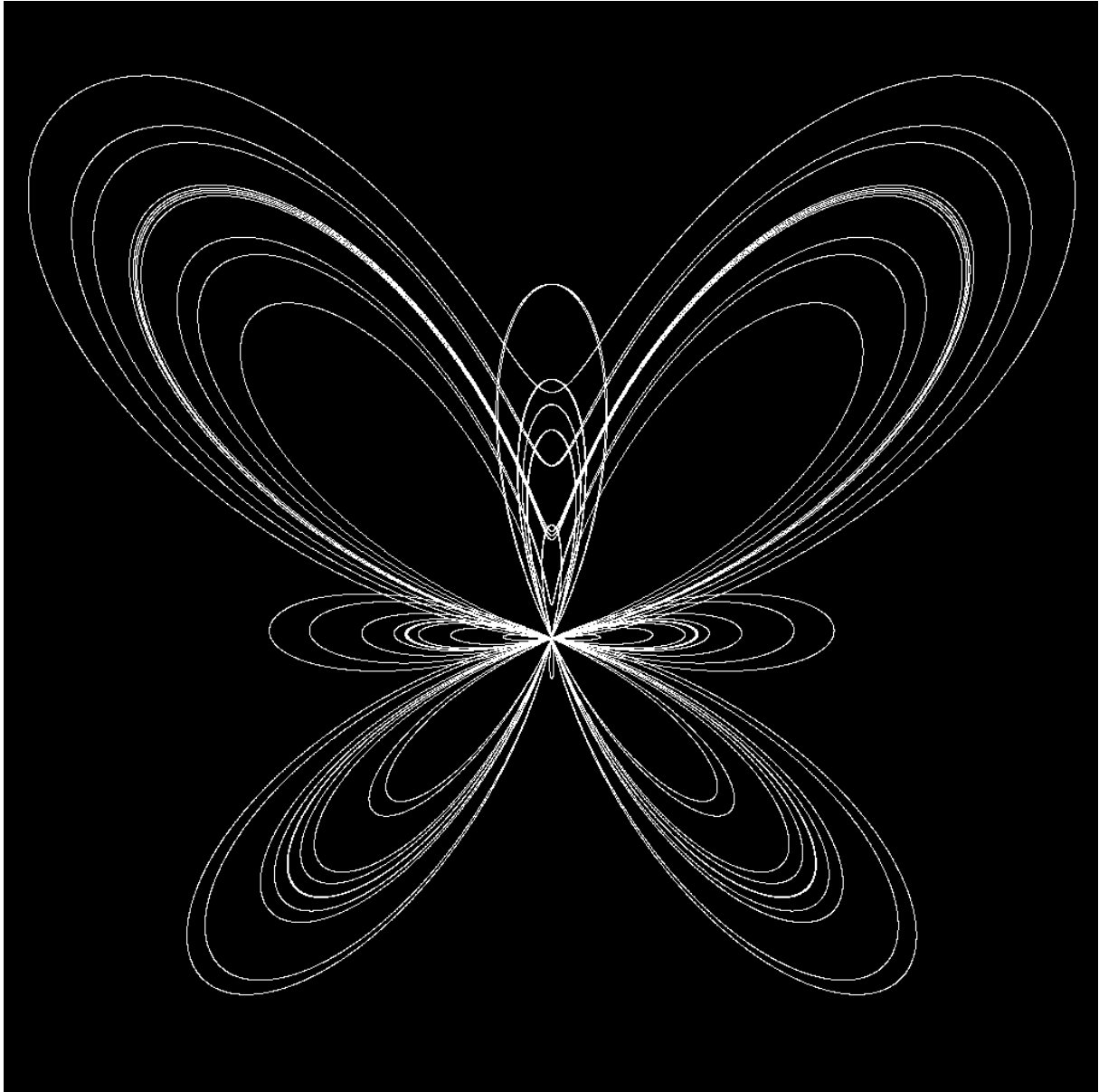


A slightly more appealing curve is the Butterfly curve

$$r = e^{\sin \theta} - 2 \cos 4\theta + \sin^5 \left( \frac{1}{24} [2\theta - \pi] \right).$$

Let us render it.

```
clearView(0)
set = polarCoords(createImage("(exp(sin(theta)) - 2*cos(4*theta) +
    sin((2*theta - pi)/24)^5, theta)", "theta", [0, 100, 0.01]))
drawLines("set")
```



Thus, **polarCoords** takes care of planar polar coordinates. But we also have **cylindricalCoords** and **sphericalCoords** that take care of three-dimensional cylindrical and spherical coordinates, respectively. Hence, an easy way to draw a cylinder of radius 4 (say) is to create a part of the  $r = 4$  plane in  $(r, \phi, z)$  space and then transform it to a cylinder in Cartesian  $(x, y, z)$  space, as illustrated below.

```
cylinder = createImage("(4, r_1, r_2)", "r", [0, 2·π, 0.1]×[-5,
5, 0.1])
cylinder = cylindricalCoords(cylinder)
drawSet3("cylinder")
```

### Complex Visualisation

Visualisation in  $\mathbb{C}$  is done by creating a set in  $\mathbb{C}$  and then use **complexCoords** to transform the set to  $\mathbb{R}^2$ . As an example of this procedure, we will use a conformal mapping, more precisely a Möbius mapping.

First we create an interesting set in  $\mathbb{C}$ . Let us create a few circles and a line.

```

circle1 = createImage("4 + 2·i + exp(i·φ)", "φ", [0, 2·π,
0.001])
circle2 = createImage("-3 - 5·i + 2·exp(i·φ)", "φ", [0, 2·π,
0.001])
circle3 = createImage("4·exp(i·φ)", "φ", [0, 2·π, 0.001])
line = createImage("1-2·i + t·(1+i)", "t", [-10, 10, 0.001])
set = circle1 u circle2 u circle3 u line

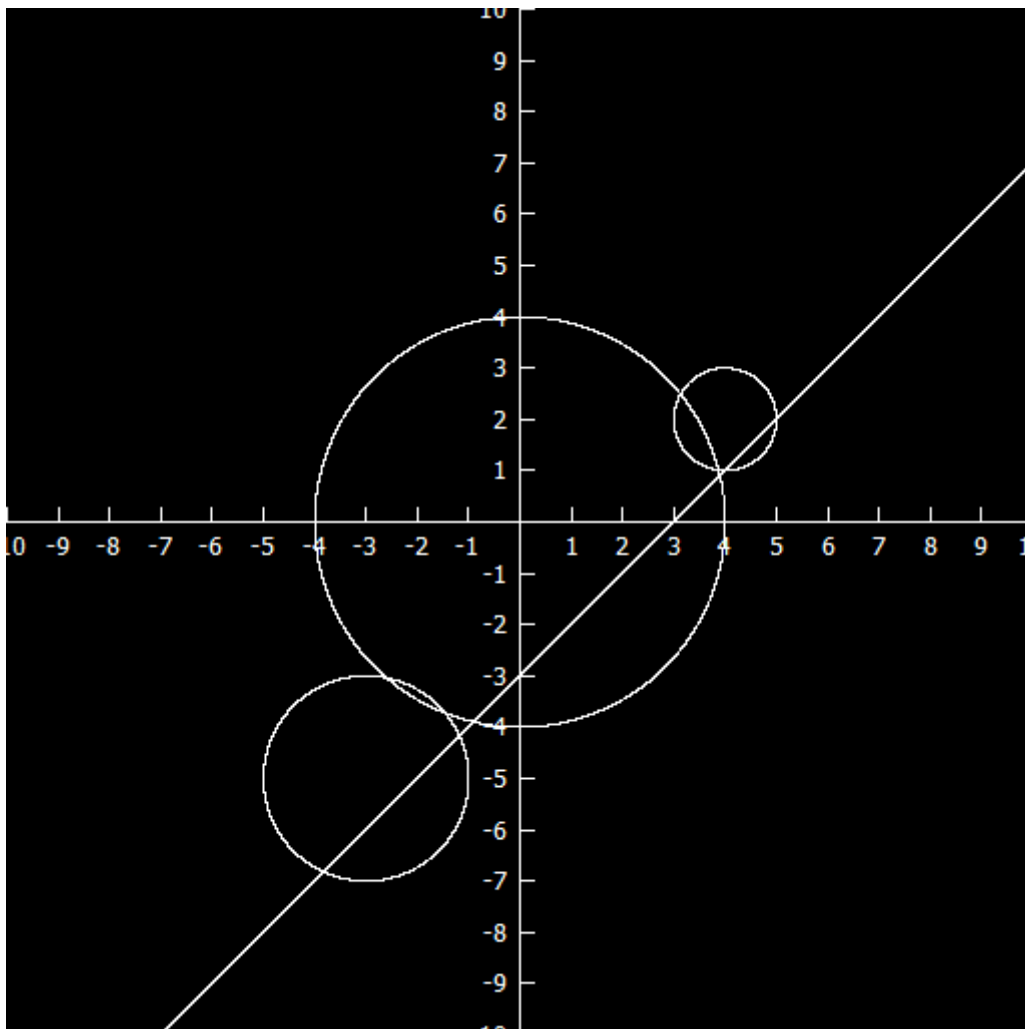
```

Now we want to see how the set looks. To this end we write

```

clearView(0)
setR2 = complexCoords(set)
drawAxes(0)
drawSet("setR2")

```



We consider the Möbius transformation

$$z \mapsto \frac{2z + i}{iz - 3 + i}$$

and implement it as

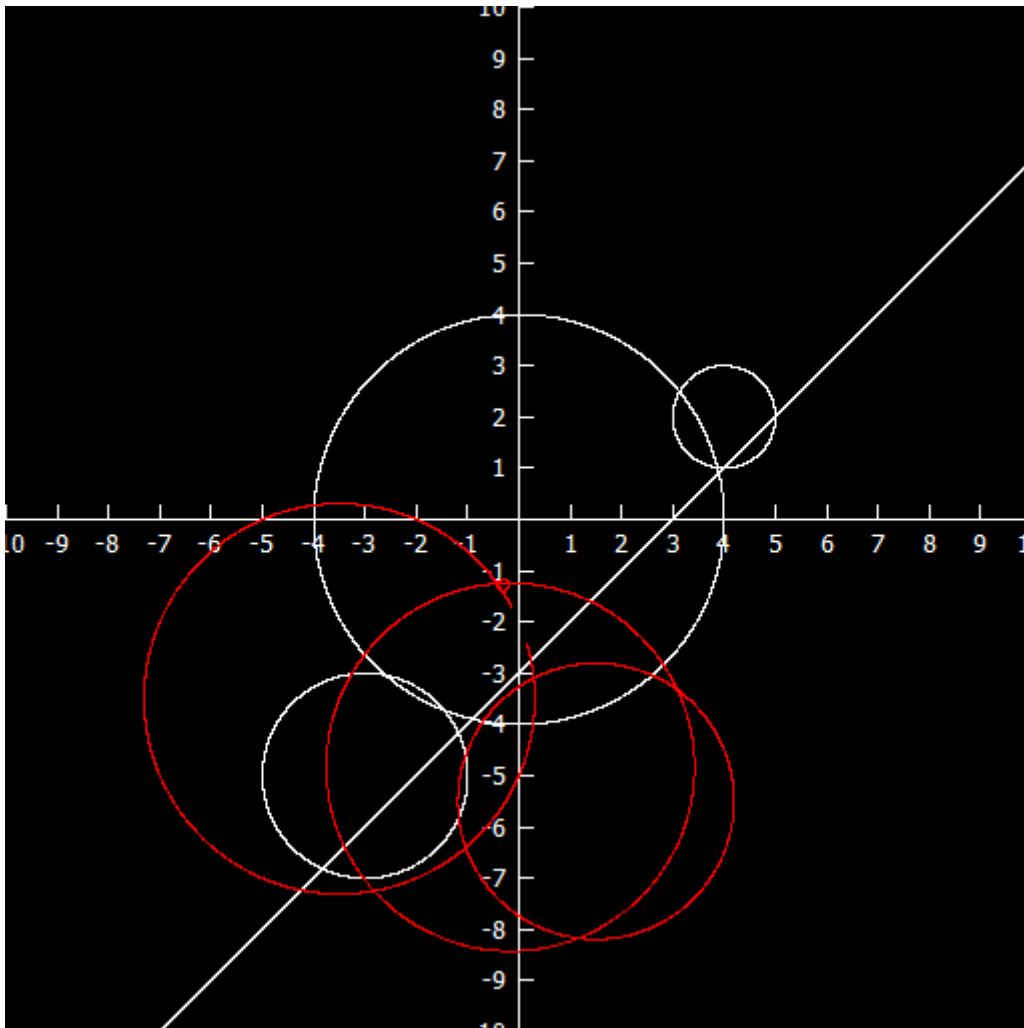
```
Möbius = "z" ↦ "(2·z + i)/(i·z - 3 + i)"
```

Now we transform set.

```
image = createImage("Möbius(z)", "z", set)
```

Finally we look at the result.

```
imageR2 = complexCoords(image)
drawSet("imageR2", "colour:red")
```



As we would expect, the four  $\mathbb{C}$ -circles are mapped to four other  $\mathbb{C}$ -circles (although one of the circles is very small, near the point  $-i$ ). In fact you can easily determine which circle the straight line was mapped to. (How?)

### Coloured Planes

We end this chapter by describing an alternative to 3D graphs  $z = f(x, y)$ . Instead of such a graph,  $f$  may be visualised by colouring each pixel  $(x, y)$  according to the value  $f(x, y)$  at that pixel. To accomplish this we only need to find a mapping from  $f(x, y)$  to a colour code, but this we have done before. The relevant AlgoSim functions are **createColouredPlane** and **drawColouredPlane**. To illustrate these, we will consider the problem of superposition of two idealised, circular water waves. Mathematically the waves are

```

 $\psi = "r" \mapsto "sin(4 \cdot norm((2, 2) - r))/6"$ 
 $\Phi = "r" \mapsto "sin(4 \cdot norm((0, 0) - r))/6"$ 

```

and the superposition is

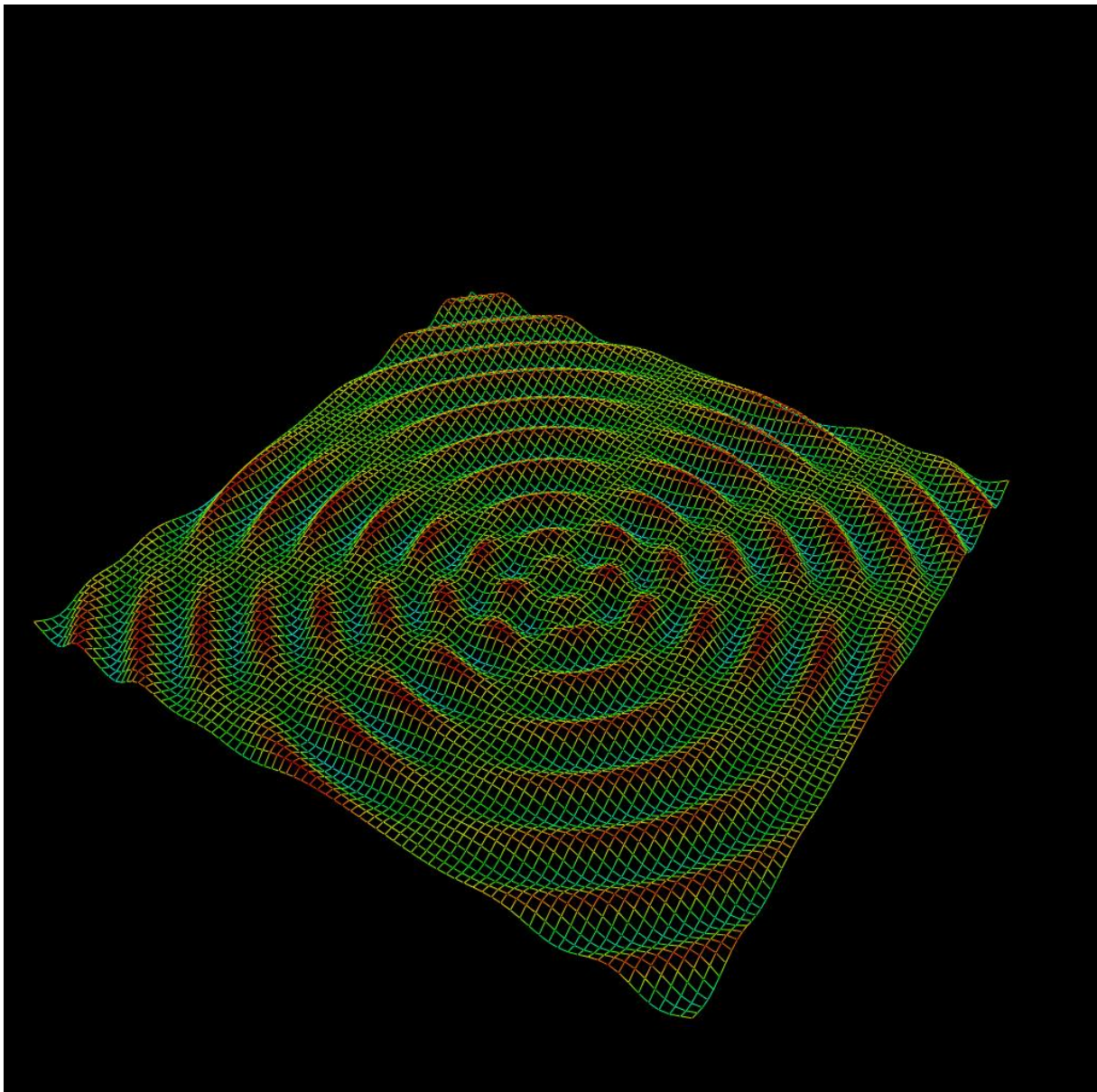
```
S = "r"  $\mapsto$  " $\psi(r) + \Phi(r)$ "
```

We could visualise this using a graph; to compare the two methods, we will do this as well. Hence we type

```

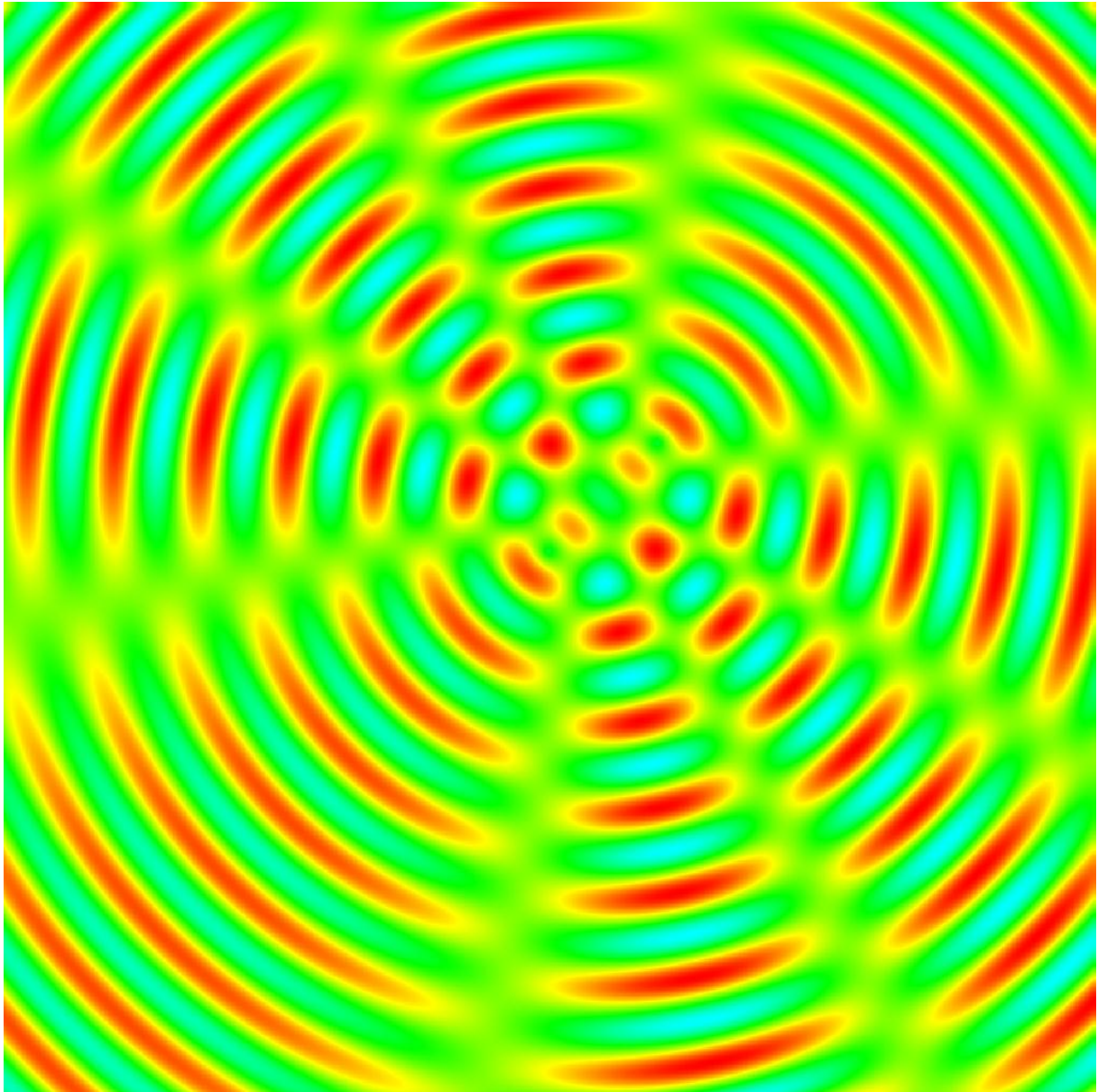
set = createSurfParamCurves("(x, y, S(x, y)), hsv(90 -
    270 * S(x, y), 1, 1)", "x, y", -10, 10, -10, 10)
drawColouredSurfParamCurves("set")

```



We now try to produce a coloured plane instead.

```
set2 = createColouredPlane("hsv(90 - 270·S((x, y)), 1, 1)", -10,  
    10, 0.1, -10, 10, 0.1)  
drawColouredPlane("set2")
```



### The Beauty of the Two-Step Approach

By now the reader is acquainted with the way visualisation is performed in AlgoSim: one first creates a set, and then one draws it. One of the major benefits of this two-step approach is that we can create a set, and then transform it using whatever algorithm we want, and then draw the result, in precisely the same way as if we had not transformed the set at all. We will now give a couple of examples of this.

First: let us draw a sine curve in space.

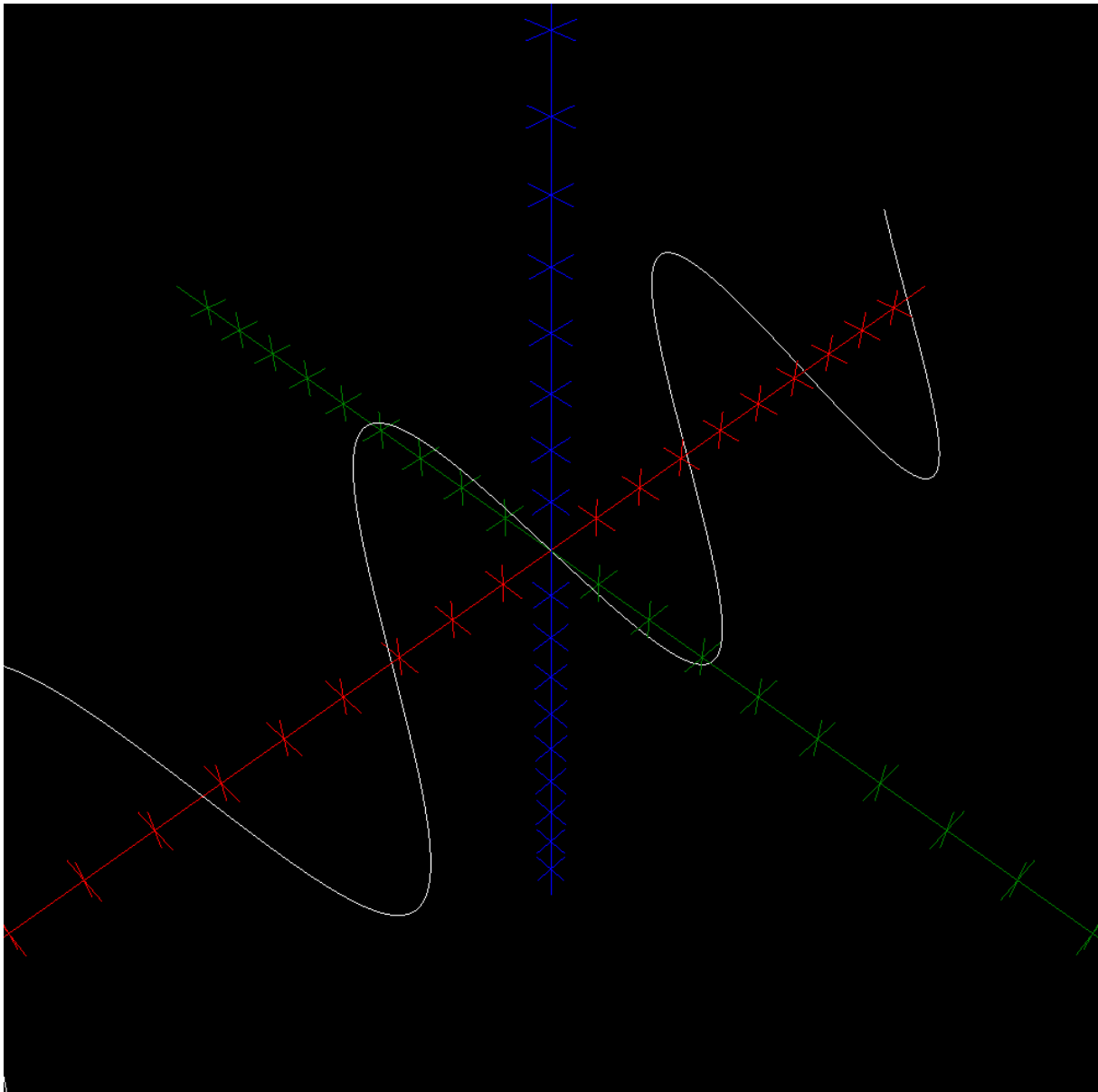
```
sine = createImage("(x, 0, 3·sin(x))", "x", [-10, 10, 0.001])
```

This is a sine curve in the plane  $y = 0$ . Assume we want to rotate the curve, so it is contained in the plane  $y + z = 0$  instead. This is done by applying the linear transformation

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1/\sqrt{2} & -1/\sqrt{2} \\ 0 & 1/\sqrt{2} & 1/\sqrt{2} \end{pmatrix}.$$

And so we do:

```
A = ((1, 0,0), (0, 1/sqrt(2), -1/sqrt(2)), (0, 1/sqrt(2),
      1/sqrt(2)))
sine2 = createImage("A.v", "v", sine)
defView3(0)
drawLines3("sine2")
```



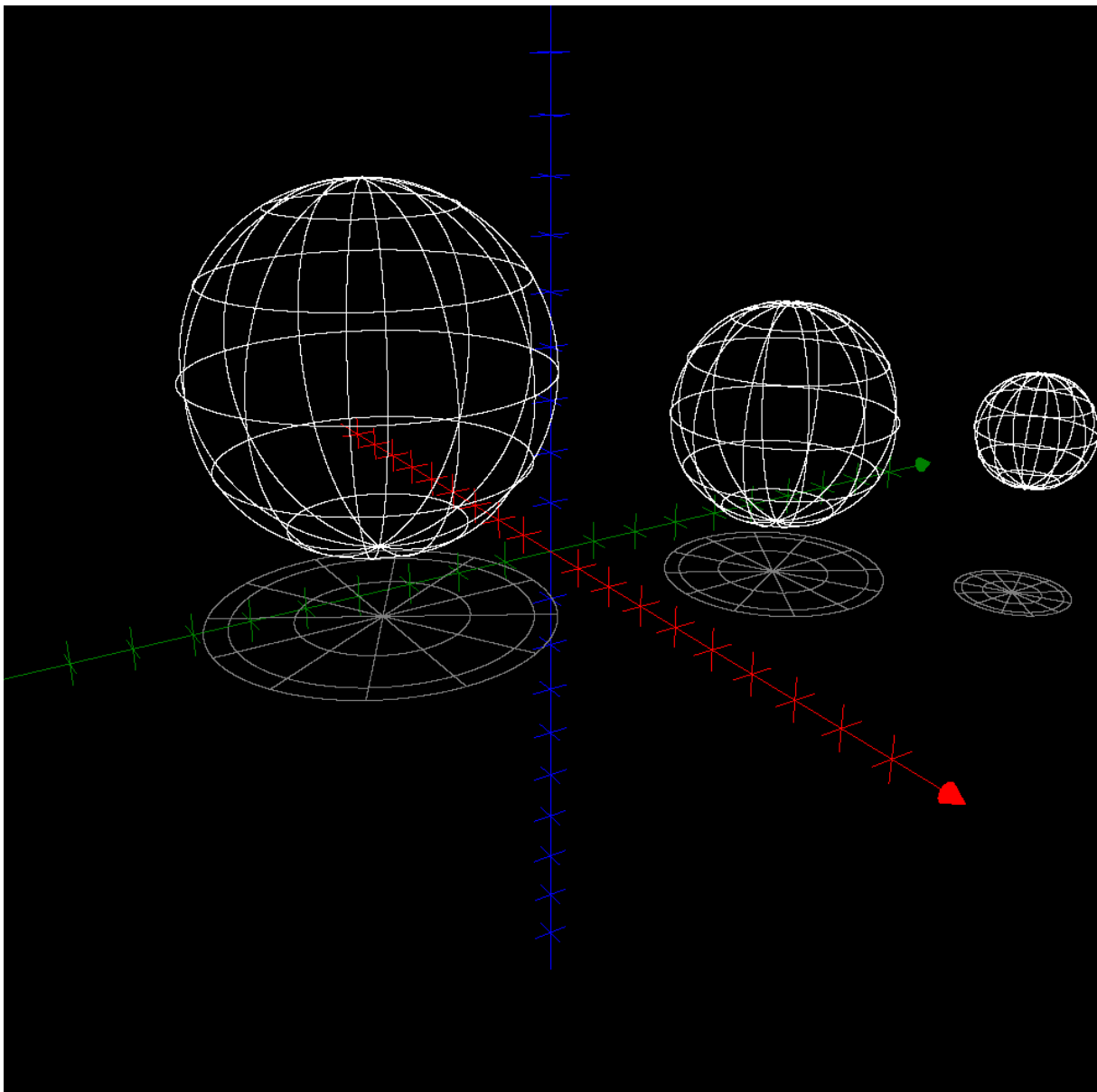
As our second example, we consider the orthogonal projection

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

We will draw a few spheres and then project them:

```

paramNet = createNet(0, π, 0.001, π/6, 0, 2·π, 0.001, π/6)
sphere = "r, θ, φ" ↦ "(r·sin(θ)·cos(φ), r·sin(θ)·sin(φ),
    r·cos(θ))"
S1 = createImage("sphere(2, r_1, r_2) + (3, 3, 3)", "r",
    paramNet)
S2 = createImage("sphere(3, r_1, r_2) + (1, -4, 4.3)", "r",
    paramNet)
S3 = createImage("sphere(1, r_1, r_2) + (6, 6, 3)", "r",
    paramNet)
spheres = S1 u S2 u S3
shadows = createImage("A·v", "v", spheres)
defView3(0)
drawSet3("spheres")
drawSet3("shadows", "colour:grey")
    
```





### Final Words on Visualisation

As you have seen in the examples, **clearView(0)** is used to remove all drawings from the 2D visualisation window, and **clearView3(0)** does the same on the 3D visualisation window. Slightly more sophisticated is **defView(0)** that first clears the 2D visualisation window, and then resets its initial range (i.e.  $[-10, 10]^2$ ) and draws the axes. **defView3(0)** does the same thing on the 3D visualisation window. Yet another useful function is **undo(0)** which removes the most recently added object in the 2D window, and – of course – there is an **undo3(0)** function as well. You might also be interested in the **removeDrawing**, **removeDrawing3**, **redraw**, and **redraw3** functions. Please see the reference section in this document for their full documentation.

By now you might be wondering how you can *export* an image rendered in AlgoSim. Actually, you should be wondering! After all, what fun is there to produce magnificent artwork, if you cannot share it? (Well, it is fun, but it is even more fun to share.) If you want to save the current 2D visualisation image, simply write

```
saveViewAsBitmap(fn)
```

where “fn” is the file name of the output. You can save images in the 24-bit Windows Bitmap (BMP), Portable Network Graphics (PNG), and AlgoSim Pixmap (ASD) formats. In almost all cases, PNG is the best format, for PNG uses a highly efficient, but lossless, compression, and is supported on all computer platforms. The image is saved in the format indicated by the file suffix (\*.bmp, \*.png, or \*.asd, respectively). For instance,

```
saveViewAsBitmap("C:\Users\Andreas Rejbrand\Pictures\image.png")
```

If you want to use a Windows “Save As” dialog box instead of entering the file name in the console, use

```
saveViewAsBitmap(fileSaveDialog(1))
```

The resulting image will have the width and height of the 2D visualisation window. Because BMP/PNG/ASD is a raster graphics format (not vector graphics), you will not be able to scale the image. To resolve this, at least to some degree, you can specify the width and height of the output bitmap image:

```
saveViewAsBitmap(fn, width, height)
```

as in

```
saveViewAsBitmap("C:\Users\Andreas Rejbrand\Pictures\image.png",
    1680, 1680)
```

To save the current 3D image, simply use **saveViewAsBitmap3** instead of **saveViewAsBitmap**; these functions work exactly the same.

You might also want to get an AlgoSim pixmap object with the image of the 2D/3D visualisation window. To get such an object, use **getViewAsBitmap(0)** or **getViewAsBitmap(w, h)** in the 2D case, and **getViewAsBitmap3(0)** or **getViewAsBitmap3(w, h)** in the 3D case. An algosim pixmap may be saved as a BMP file by means of **savePixmapToFile(fn)**.

This concludes the chapter on graphical visualisation.

## Physical Simulations

In AlgoSim, it is possible to specify a vector field, and simulate particle motion in it. There are two ways to do this:

- **A force-field.** The acceleration of the particle is a function of its spatial position.
- **A flow.** The velocity of the particle is a function of its spatial position.

### Force Fields

We begin to investigate force-fields. First of all, it might be nice to be able to visualise the field itself. This is done by plotting vectors at a discrete grid of points. Formally, a vector field in  $n$  dimensions is a set of  $2n$ -dimensional vectors, the first  $n$  components of each is the vector's position, the last  $n$  components being the vector itself. For instance, if the value of the vector field is  $(1, 2)$  at the point  $(5, 5) \in \mathbb{R}^2$ , then the vector  $(1, 2, 5, 5)$  is one of the members of the vector field set. Vector fields are created by **createVectorField** and drawn by **drawVectorField**.

We use a simple constant vector field as a first example; you might think of it as gravity or the electric field between two planar conductors held at different voltages.

```
vfield = createVectorField("(0, -1)", "x, y", [-10, 10]^2)
drawVectorField("vfield", "colour:#333333")
```

Let us now simulate a ball in this field. The relevant function is **computeParticleTrajectory**. It takes the initial position and velocity of the ball as arguments, as well as the initial and final times, and the temporal resolution of the numerical integration. As an example,

```
traj = computeParticleTrajectory("(0, -1)", "r", (-8, 8), (1,
                                0), 0, 100, 0.001)
drawSet("traj")
```

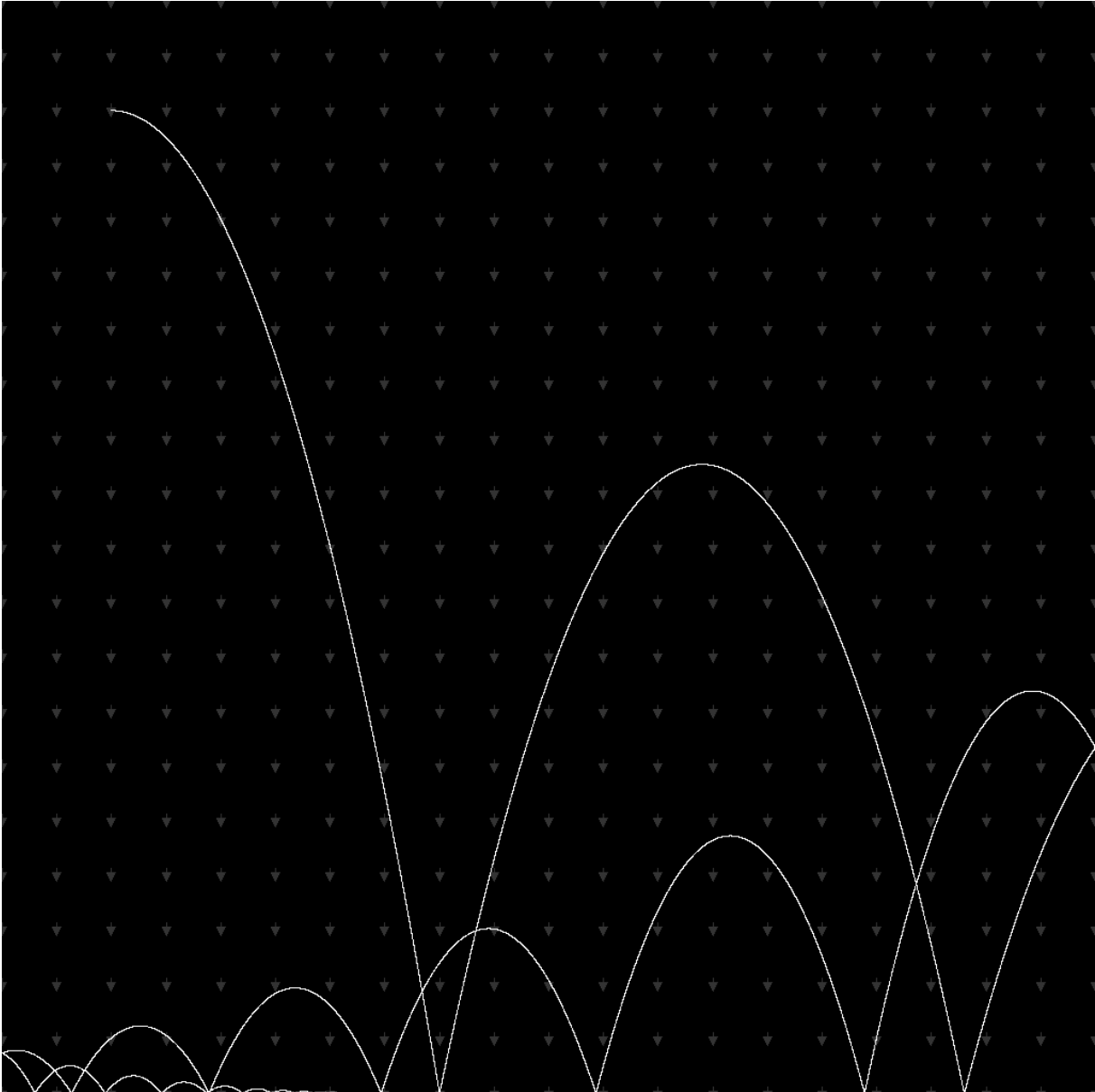
would draw the trajectory. However, to make things a bit more interesting, we can make the ball bounce when it hits the edges of the visualisation window, i.e. the box  $[-10, 10]^2$ . We choose to let the ball retain 80 % of its *speed*<sup>1</sup> at each bounce.

```
undo(0)
traj = computeParticleTrajectory("(0, -1)", "r", (-8, 8), (1,
                                0), 0, 100, 0.001, (-10, 10, -10, 10), 0.8)
drawSet("traj")
```

The output of this is shown below.

---

<sup>1</sup> Hence the ball retains 64 % of its kinetic energy.



We can also animate the motion of the ball.

```
animateTrajectory("traj", 70, false)
```

The second argument is the speed of the animations (steps per frame), while the third argument will make the ball leave a trace if set to true. Unfortunately, however, due to technical limitations in the contemporary art of printing, I am not able to display the animation on this page.

### Flows

Flows work the same way as force-fields. The only exception is the integration: in a flow, the *velocity* of the particle is a function of its position. The new function **computeFlowTrajectory** replaces `computeParticleTrajectory`. Of course, `computeFlowTrajectory` will not need an initial velocity. As an example, we consider a model of an oscillating chemical reaction. The  $x$ - and  $y$ -axes are the concentrations of the two major compounds.

```
a = 2
b = 3.001
```

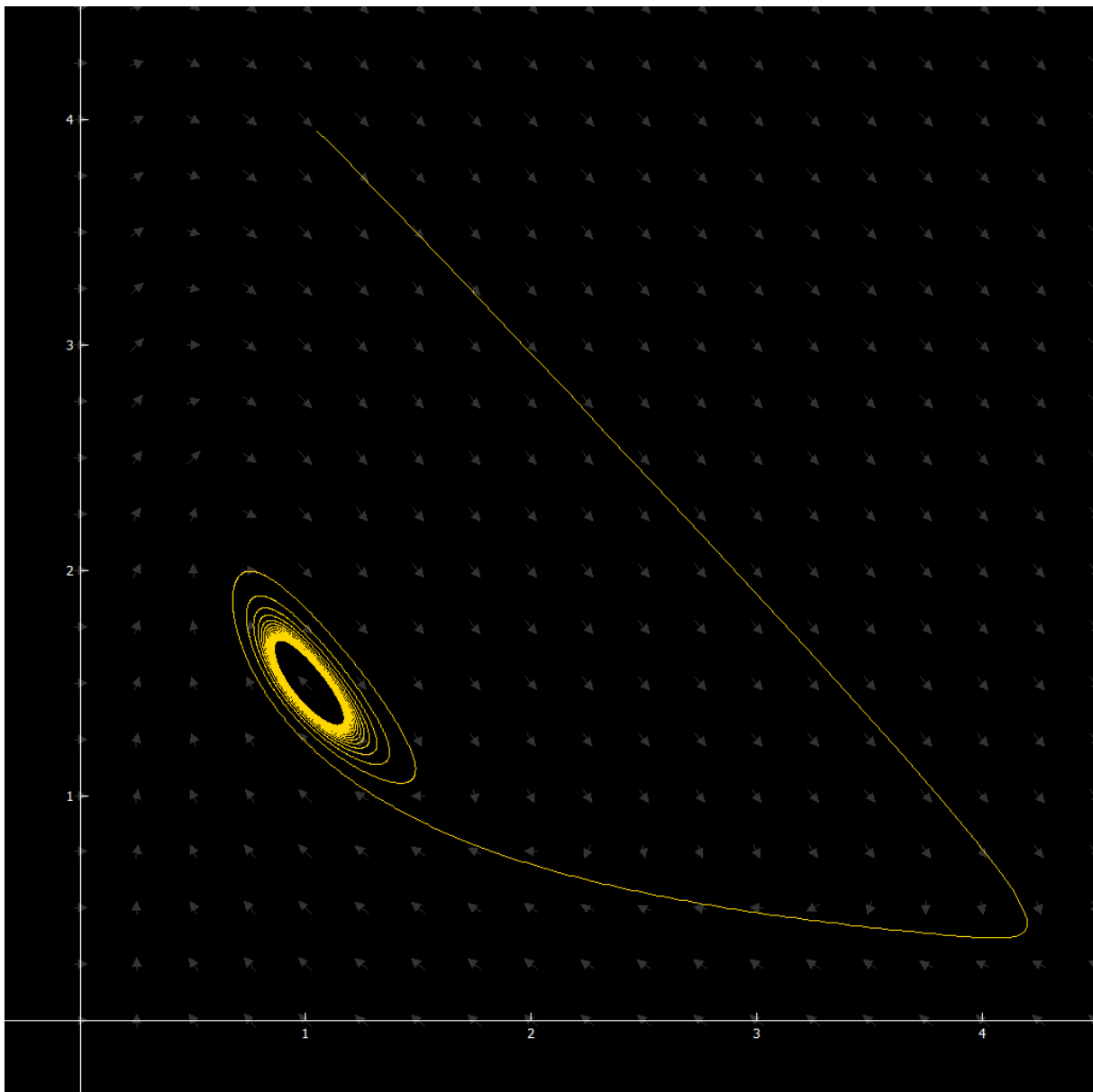
```

clearView(1)
setView(-1/3, 4.5, -1/3, 4.5)

vectorField = createVectorField("(1 + a·x^2·y - b·x-x, -a·x^2·y
    + b·x)", "x, y", [0, 10, 0.25]^2)
drawVectorField("vectorField", "colour:#333333")
drawAxes(1)

flow = computeFlowTrajectory("(1 + a·r_1^2·r_2 - b·r_1 - r_1, -
    a·r_1^2·r_2 + b·r_1)", "r", (1, 4), 0, 100, 0.01)
drawLines("flow", "colour:gold")

```



Of course this trajectory can be animated as well. (Try to play with  $a$  and  $b$ .)

## Auditory Visualisation

In AlgoSim, you can also visualise data by means of waveform audio. AlgoSim can import and export WAV PCM files (\*.wav), and send waveform data to the computer's speakers. A sound is implemented as an own data type, but to create and edit waveform audio, column matrices are used. Hence you have to convert between column matrices and sounds. **sndMatrixToSound** takes one column matrix and a sample rate, and return a sound object. **sndGetSamples**, on the other hand, takes a sound and returns the column matrix. As a first example, we generate a 2 s 400 Hz sine tone with a sampling frequency of 4 000 Hz.

```
v = 400
ω = 2·π·v
A = 2^31
snd = createImage("A·sin(ω·t)", "t", [0, 2, 1/4000])
snd = sndMatrixToSound(setToMat(snd), 4000)
```

A more interesting example:

```
v = 400
ω = 2·π·v
A = 2^31
snd = createImage("A·sin(ω·(sin(t)·t))", "t", [0, 4·π, 1/10000])
snd = sndMatrixToSound(setToMat(snd), 10000)
```

A faster way of creating a pure sine tone is to use the **createSineTone** function. The tone above may for instance be created by

```
createSineTone(400, 2)
```

This way it is very easy to study, for instance, beat. Try

```
s1 = createSineTone(400, 2)
s2 = createSineTone(401, 2)
s = sndSuperpose(s1, s2)
```

where the function of **sndSuperpose** ought to be obvious.

### MIDI Functions

You can produce MIDI sounds, i.e. the sounds of 128 pre-defined musical instruments. This is very fun. To play a note, use **note**. The first argument is an integer in [0, 127] and defines the note (~the frequency) of the note, while the second (optional) argument, also an integer in the same interval, determines the velocity (the volume, the intensity of the sound produced by the speakers). For some instruments, e.g. piano, this produce a tone with a small duration in time. Some instruments, however, will continue to produce the tone until you send the command **noteOff** using the same arguments. To set the instrument, use **changeInstrument**, the first argument of which – yes, that's right – is an integer in [0, 127]. The default instrument, with identification 0, is "Grand Acoustic Piano". **notes** plays a set of notes.

Try these functions! For instance, try

```
note(70)
```

## Some More Functions in Focus

So far we have only come across a few of the almost 500 functions built into AlgoSim. Here we let a few more functions glance in the spotlight. For full details on each function, see the reference section.

### Real and Complex Numbers

The functions **isPrime**, **nextPrime**, **prevPrime**, **Fibonacci**, **coprime**, **ceil**, **floor**, **round**, **trunc**, **frac**, **to-tient**, **mod**, **divisors** etc. do exactly what one would expect. Notice that the **ceil** and **floor** of a number  $n$  also may be written  $\lceil n \rceil$  and  $\lfloor n \rfloor$ , respectively. Hence, technically speaking,  $\lceil \cdot \rceil$  and  $\lfloor \cdot \rfloor$  are defined as circumfix operators. There is also an infix operator for coprime:  $a \perp b$  returns true iff  $a$  and  $b$  are relatively prime.  $n!$  is  $n$ -factorial, so that  $!$  is a postfix operator.  $\%$  and  $\%_0$  are postfix operators with the obvious functions, i.e.  $x \mapsto 0.01x$  and  $x \mapsto 0.001x$ , respectively.  $\text{mod}(a, b)$  adds or subtracts an integral number of  $b$ 's from  $a$ , so that the result lies within  $[0, b[$ .  $a | b$  returns true if  $a$  divides  $b$ , and false otherwise. **divisors** returns the vector of divisors of the argument.

The Iverson bracket notation  $[expr]$  is also very handy.  $[expr]$  returns 1 if  $expr$  is true, and 0 otherwise. Observe that both the Kronecker delta function and the rectangular function are special cases of the Iverson bracket, corresponding to the expressions " $x=y$ " and " $x > -1/2 \wedge x < 1/2$ ", respectively.

When it comes to elementary functions, AlgoSim got them all. **sin**, **cos**, **tan**, **cot**, **sec**, **csc**, **arcsin**, **arccos**, **arctan**, **arccot**, **arcsec**, **arccsc**, **sinh**, **cosh**, **tanh**, **coth**, **sech**, **csch**, **arcsinh**, **arccosh**, **arctanh**, **arcoth**, **arcsech**, **arccsch**, **exp**, **ln**, and **sqrt** are all defined for both real and complex arguments.

For complex numbers, **arg** and **abs** return the argument and the modulus. All complex functions use the principal branch of the argument, i.e.  $\arg z \in ] -\pi, \pi] \quad \forall z \in \mathbb{C}$ .

If  $f$  is a function and  $S$  is a set, it is not possible to compute the image of  $S$  under  $f$  by writing  $f(S)$ . [Indeed, there are functions that really take a set as an argument, so it would be inconsistent to use this syntax.] But as we have seen, we can use `createImage("f(x)", "x", S)`.

Special functions include integrals such as **erf** (Error Function), **erfc** (Complementary Error Function), **Ci** (Cosine Integral), **Si** (Sine Integral), **FresnelC** (Fresnel Cosine Integral), **FresnelS** (Fresnel Sine Integral), and polynomials such as **hermiteProb**, **hermitePhys**, and **Bernstein**. We also have **harmonicNumber**, **gammaFunction**, and **bessel**.

**diffGraph** and **intGraph** take a graph  $\{(x, y) \in \mathbb{R}^2: y = f(x)\}$  as argument, and returns the graph of the derivative or integral, respectively.

### Vectors and Matrices

When it comes to vectors, the functions **norm**, **taxiNorm**, **maxNorm**, **pNorm**, **absVect**, **max**, **min**, **sum**, **mean**, **product**, **angle**, and **sort** are available.

The functions **identityMatrix** and **zeroMatrix** return the identity matrix of size  $n$  and the zero matrix of size  $m \times n$ , respectively. **fillMatrix** returns a  $m \times n$  matrix with all entries set to a constant. **computeMatrix** returns a  $m \times n$  matrix where the element  $(i, j)$  is  $f(i, j)$  for some function  $f$ . This is a rather powerful function. For example,

```
computeMatrix(12, 12, "m·n", "m, n")
```

returns the 12 by 12 multiplication table

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 2 & 4 & 6 & 8 & 10 & 12 & 14 & 16 & 18 & 20 & 22 & 24 \\ 3 & 6 & 9 & 12 & 15 & 18 & 21 & 24 & 27 & 30 & 33 & 36 \\ 4 & 8 & 12 & 16 & 20 & 24 & 28 & 32 & 36 & 40 & 44 & 48 \\ 5 & 10 & 15 & 20 & 25 & 30 & 35 & 40 & 45 & 50 & 55 & 60 \\ 6 & 12 & 18 & 24 & 30 & 36 & 42 & 48 & 54 & 60 & 66 & 72 \\ 7 & 14 & 21 & 28 & 35 & 42 & 49 & 56 & 63 & 70 & 77 & 84 \\ 8 & 16 & 24 & 32 & 40 & 48 & 56 & 64 & 72 & 80 & 88 & 96 \\ 9 & 18 & 27 & 36 & 45 & 54 & 63 & 72 & 81 & 90 & 99 & 108 \\ 10 & 20 & 30 & 40 & 50 & 60 & 70 & 80 & 90 & 100 & 110 & 120 \\ 11 & 22 & 33 & 44 & 55 & 66 & 77 & 88 & 99 & 110 & 121 & 132 \\ 12 & 24 & 36 & 48 & 60 & 72 & 84 & 96 & 108 & 120 & 132 & 144 \end{pmatrix}$$

We can also obtain a list of prime numbers:

```
computeMatrix(1, 12, "prime(n)", "m, n")
( 2 3 5 7 11 13 17 19 23 29 31 37 )
```

We can even create a numbered table of primes:

```
computeMatrix(2, 12, "ifThen(m=1, n, prime(n))", "m, n")
( 1 2 3 4 5 6 7 8 9 10 11 12 )
( 2 3 5 7 11 13 17 19 23 29 31 37 )
```

We have already seen **toEchelonForm** and **sysSolve**. Other convenient functions include **rank**, **rowScale**, **rowMove**, **rowAddMul**, **matRows** (the number of), **getRow**, **matCols** (the number of), and **getCol**.

If  $A$  is a vector, then  $A_i$  returns the  $i$ th component of  $A$ . If  $A$  is a matrix, then  $A_(i, j)$  returns the  $i, j$  element of the matrix. Technically,  $_$  is an infix operator that takes two arguments, either a vector and a real number, or a matrix and a vector, and returns a number.

### Texts (strings)

String functions include **length**, **substring**, **strSplit**, **strPos**, **strLeft**, **strRight**, **strBeginsWith**, **strEndsWith**, **strContains**, **strReplaceAll**, **txtPos**, **txtBeginsWith**, **txtEndsWith**, **txtContains**, and **txtReplaceAll**. In general, the **str\*** functions are case-sensitive, whereas the **txt\*** functions are not.

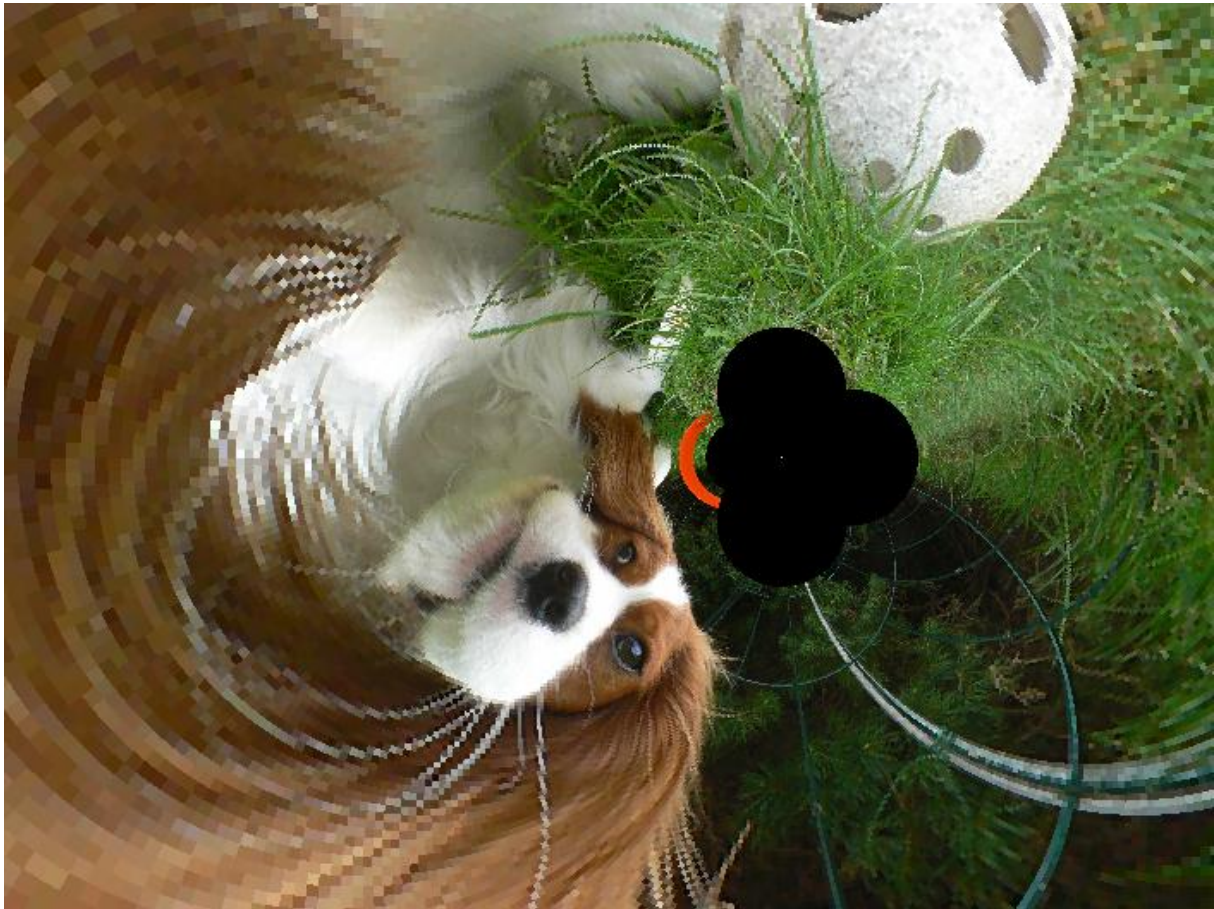
When it comes to ciphers and encryption, we have **ROT13**, **CaesarCipher**, **VigenèreEncrypt**, and **VigenèreDecrypt**. ROT13 is a simple involution, and the inverse of  $\text{str} \mapsto \text{CaesarCipher}(\text{str}, n)$  is  $\text{str} \mapsto \text{CaesarCipher}(\text{str}, -n)$  where  $n \in [0, 25] \cap \mathbb{Z}$ . But a text encrypted using the Vigenère algorithm, which uses a password, or key, to encrypt and decrypt the text, is a bit harder to crack without knowledge of the key. For fun, I challenge you to crack the following message (I hope no one succeeds):

ggwormlmfmnuperdwetxwcuucjtqyv

### Pixmaps

Pixmap (bitmap) functions include **pmInvert**, **pmToBitmap**, **pmFlipV**, **pmFlipH**, **pmRot90P**, **pmRot90N**, **pmRotateEx**, **pmRotate**, **pmShear**, **pmScale**, **pmToGreyscale**, **pmFixHue**, **pmToMonochromatic**, **pmPixelate**, **pmTransform**, **pmMöbius**, **pmGetRect**, **pmHeight**, **pmWidth**, **pmGetRAMSize**, **pmShiftHue**, **pmResize**, **pmAddSizeToEdges**, **pmRemoveSizeFromEdges**, **pmBlend**, **pmContrast**, **pmInvertValue**, **pmInvertLightness**, **pmSwapBW**, **pmReplaceColour**, **pmRGBAdjustment**, **pmHSVAdjustment** and many others, as well as **loadPixmapFromFile** and **savePixmapToFile**.

Below is pmMöbius exemplified. A picture of a dog, a floorball ball and a red, vertical, bar in the grass is transformed using the standard Möbius transformation  $z \mapsto (z + 1)/(z - 1)$ . In the middle you see four circles, the inside of which are missing. These circles are the images of the four edges of the original image, the outside of which – of course – is undefined.



### Sounds and MIDI Functions

Have a look at **sndSuperpose**, **sndMakeMultichannel**, **sndSplitChannels**, **sndGetSampleRate**, **sndGetNumChannels**, **sndAppend**, **reduceSound** and **sndGetNumSamples**. There is also **changeMidiVolume** and **sendMidiMsg** (for low-level interaction with the computer's sound card).

### More

In the reference section, you will find all functions in AlgoSim. Have a look at them!



## The Operator Table

By now you know there are many operators you can use in AlgoSim. Examples include

- Unary operators
  - Prefix operators:  $\neg$ ,  $-$ , ...
  - Postfix operators:  $!$ ,  $\%$ ,  $*$ , ...
- Binary operators
  - Infix operators:  $+$ ,  $-$ ,  $:$ ,  $\times$ ,  $^$ ,  $\perp$ ,  $|$ , ...
- $n$ -ary operators
  - Circumfix operators:  $[, , \dots]$ ,  $\{, , \dots\}$ ,  $(, , \dots)$ ,  $\lfloor, , \dots\rfloor$ , ...

One of the truly original features of AlgoSim is that no operators are hard-coded, i.e. the end-user is able to define new operators (prefix, postfix, infix, and circumfix) and remove, or redefine, existing operators.

Given a Windows user, there are two operator tables, one in the AlgoSim subdirectory of the Program Files folder, common to all users, and one in the local user's AppData folder. Typical paths may be

- C:\Program Files (x86)\AlgoSim\ops.asd
- C:\Users\Andreas Rejbrand\AppData\Local\Rejbrand\AlgoSim\2.0\ops.asd

If the local file exists, it will be used, and the common will be ignored. If not, the common file will be used. Thus, to restore the operator table to its default appearance, simply copy the common ops.asd to the local directory. This can in fact be done automatically from inside AlgoSim itself, by means of the **restoreOperatorTable** program. This will copy the common file to the local directory (overwriting any existing file there), and then call the kernel function **reloadOperatorTable**, which will cause AlgoSim to reload the operator table.

So, you can alter the (local) operator table to any degree you like. Simply double-click ops.asd to open it in the AlgoSim Data viewer (and editor), make your changes, and save the result. Then call **reloadOperatorTable** to reload the table, if you edited the file during an AlgoSim session.

Each row in the table corresponds to one operator, and the columns are

type | c1 | c2 | fname | lr | rr | rtl.

type is either "prefix", "postfix", "infix", or "circumfix". For prefix, postfix, and infix operators, c1 is the operator symbol, a Unicode character, but not an alphanumerical one. For a circumfix operator, c1 and c2 are the initial and final operator symbols, which must be different. fname is the name of the function that the operator calls. It must accept the right number of arguments (the number of the operands), of course. If lr is -1 (rather than 0), the left operand (if any) will be *raw*, i.e. converted to a string before it is sent to the function named fname. rr means "right operand is raw". For instance, the assignment operator  $:=$  is raw to the left, so you can write  $a := 5$  rather than " $a$ "  $:= 5$  (why?). "rtl" means that the operators with the same level of precedence will be read from the right to the left, as suitable for  $:=$  and  $^$ . However, the current implementation ignores this option, so that all operators will be read from the left to the right, regardless of this setting.

Defining your own operators might be extremely useful in many situations.

## Programming

You can write simple programs in AlgoSim. If you create a program called MyProgram, you can call it almost like an ordinary function. If your program requires no arguments, you simply write MyProgram(0). But what if it does require arguments? Well, no semantics for this is implemented. However, this is not a major problem. Indeed, the value in place of the argument to MyProgram is never used, so you can assign variables here. For instance you might write MyProgram(a := 2, b := 5). This will assign the value 2 to a, and 5 to b, before executing MyProgram. The only drawback is that there – obviously – is no such thing as “local variables”. An AlgoSim program may return nothing (which will be translated to 0), or a value of any data type, as is the case of ordinary functions. Hence you can write a program (or a function) that inputs a table and returns a pixmap. Or a function that inputs a sound, and returns a string. Or whatever.

Basically, an AlgoSim program is simply a number of console input lines. But there are also flow control structures, such as **if** conditionals and **repeat** loops. Such commands are always preceded by a semicolon.

### The If Conditional

The structure of a simple **if** construct is

```

; if <expr logical value>
    command1
    command2
    :
    commandN
; endIf

```

This construct works as in most programming and scripting languages. When the program interpreter encounters the **if** line, it will evaluate the expression on this line. If it evaluates to true, the contents of the **if** block will be executed. If it evaluates to false, the program interpreter will skip the entire **if** block and continue execution of the program on the next line after **endIf**.

The **if** construct supports an **else** block, and the syntax is

```

; if <expr logical value>
    command1
    command2
    :
    commandN
; else
    command1b
    command2b
    :
    commandNb
; endIf

```

The most general conditional is

```

; if <expr logical value>
    :
; elsif <expr logical value>
    :
; elsif <expr logical value>

```

```

        :
;else
        :
;endIf

```

with an arbitrary number of **elseif** statements. Exactly one of the blocks (denoted by vertical ellipsis) will be executed, namely the one after the first <expr logical value> that evaluates to true; if non of them does, the code under **else** will execute. If there is no **else** block, nothing will execute.

All indentation is optional, yet recommended. The advice is to use four (4) spaces as indent. The flow commands ;cmd (such as **if**, **elseif**, **else**, and **endif**) are all case-insensitive; thus, all of the following statements are identical:

```

;elseif
;elseif
;ELSEIF
;elseif

```

The first variant is recommended, though.

### The Repeat Loop

The most basic loop is

```

;repeat
    <commands>
;indefinitely

```

When execution hits the **indefinitely** line, it will jump back to **repeat**. Hence <commands> will be repeated indefinitely. To exit such a loop, use the **break** command.

```

;repeat
    <commands>
    ;break
    <commands>
;indefinitely

```

The **break** command will continue execution of the program on the line immediately below **indefinitely**. The **continue** command will jump directly to the next iteration, skipping all that remains in the current iteration. That is, when the program interpreter encounters a **continue** statement, it will jump to the **indefinitely** line, and then go back to **repeat**, as expected.

For example: to print all numbers 1, 2, ..., 10, write

```

n = 1
;repeat
    print(n)
    n = n + 1
    ;if n > 10
        ;break
    ;endif
;indefinitely

```

A variant of the **repeat** loop is

```

;repeat
    <commands>
;until <expr logical value>
    
```

This will perform <commands>, and then check <expr logical value>. If this evaluates to false, the **repeat** block will execute once more. If it evaluates to true, the loop will break, i.e. execution will continue on the next line after the **until** statement. Hence the above is equivalent to

```

;repeat
    <commands>
    ;if <expr logical value>
        ;break
    ;endif
;indefinitely
    
```

The above example can thus be written more elegantly

```

n = 1
;repeat
    print(n)
    n = n + 1
;until n > 10
    
```

The final variant is

```

;repeat
    <commands>
;while <expr logical value>
    
```

which will repeat <commands> *as long as* <expr logical value> is true. Hence the above is equivalent to

```

;repeat
    <commands>
;until ¬ <expr logical value>
    
```

and also

```

;repeat
    <commands>
    ;if ¬ <expr logical value>
        ;break
    ;endif
;indefinitely
    
```

Hence our example can be written

```

n = 1
;repeat
    print(n)
    n = n + 1
;while n ≤ 10

```

### The DoWhile Loop

The **repeat ... until** and **repeat ... while** loops check the <expr logical value> at *the end* of each iteration.

The **DoWhile** loop does the opposite:

```

;doWhile <expr logical value>
    <commands>
;stop

```

will begin with checking <expr logical value>. If this evaluates to true, execution continues on the next line. When execution hits **stop**, the program will jump back to **doWhile** and check <expr logical value> again. If <expr logical value> should evaluate to false, execution will continue on the first line after **stop**. Hence the above is equivalent to

```

;repeat
    <commands>
    ;if ¬ <expr logical value>
        ;break
    ;endif
;indefinitely

```

Compare this with the **repeat ... while** loop above. Of course, **break** and **continue** can be used in **doWhile** loops as well as in any kind of the **repeat** loop.

### The For Loop

The AlgoSim **for** works as in most languages, but is slightly more powerful. The syntax is show below.

```

;for <init>; <each but first>
    <commands>
;stop

```

This is equivalent to

```

<init>
<commands>
;repeat
    <commands>
    ;if ¬ <each but first>
        ;break
    ;endif
;indefinitely

```

Now, recall The Semicolon Operator on page 20. Using this, we can write our example in the very concise form

```

;for x = 1; x = x + 1; x ≤ 10
    print(x)
;stop

```

which works as **for** loops usually work. Please notice that the first semicolon (after  $x := 0$ ) is part of the **for** flow command syntax, whereas the second semicolon (after  $x := x + 1$ ) is a semicolon operator. The command  $x := x + 1; x \leq 10$  will thus execute at each iteration (but the first), and will return true if and only if  $x \leq 10$ . Of course, **break** and **continue** statements are perfectly valid in **for** loops.

### The Iterate Loop

The **iterate** loop is a very powerful extension of the **for** loop to several variables. Per definition,

```

;iterate x:a:b:c, y:α:β:γ, ...
    <commands>
;endIterate

```

is exactly equivalent to

```

;for x = a; x = x + c; x ≤ b
    ;for y = α; y = y + γ; y ≤ β
        ...
        <commands>
        ...
    ;stop
;stop

```

If the step sizes ( $c, \gamma, \dots$ ) are not specified (i.e. `;iterate x:a:b,y:α:β, ...`), then they are assumed to be equal to unity.

For instance, the sample program `gitter.prg`, that draws a simple cubic lattice (of atoms or ions, for instance), is implemented as

```

clearView3(1)
beginDrawing(1)
setLight(true)

;iterate x:-3:3, y:-3:3, z:-3:3
    drawSphere(2*(x, y, z), 1/5, "slices:16; loops:16")
;endIterate

endDrawing(1)
redraw3(1)

```

### Entering Programs

An AlgoSim program is a UTF-8-encoded plain text file with the suffix ".prg". You can use the editor of your choice to write programs, but preferably one with support for AlgoSim syntax highlighting, such as Rejbrand Text Editor. To make programs available to AlgoSim, you must save them in a directory that AlgoSim looks in. On a given computer (and in a given user account), there are generally two such directories: one common to all users of the computer, and one specific to the current user. On a typical Windows 7 system, the two directories are

C:\Program Files (x86)\AlgoSim\programs

and

C:\Users\<<User Name>\AppData\Local\Rejbrand\AlgoSim\2.0\programs,

respectively. To find out the exact directories on your computer, enter the command **getProgramLocations(0)**. The programs in these two directories are loaded and interpreted automatically when AlgoSim starts. If you alter any program while AlgoSim is running, you must tell AlgoSim to reinterpret the program before you can use the new version of it. This is done by the **reloadPrograms(0)** command.

### A Few Examples

The simplest programs are those that do not contain any flow control constructs. For instance,

```
Möbius.prg


---


clearView3(1)
Möbius = createSurfParamCurves("5*((1 +
    0.5*v*cos(0.5*u))*cos(u), (1 +
    0.5*v*cos(0.5*u))*sin(u), 0.5*v*sin(0.5*u))", "u, v",
    0, 2*pi, pi/36, pi/12, -1, 1.01, 0.05, 0.1)
drawSurfParamCurves("Möbius")


---


```

might be a handy program for drawing a Möbius strip.

A slightly more complicated program can solve the Monty Hall problem for us.

```
doors.prg


---


;; Three Doors (probability paradox)

N = 10000

nCarsStay = 0
nCarsSwap = 0

;; Simulate "stay" scenario
;for j = 1; j = j + 1; j ≤ N
    rightDoor = randomInt(3)
    guess = randomInt(3)
    ;if rightDoor = guess
        nCarsStay = nCarsStay + 1
    ;endif
;stop

;; Simulate "change door" scenario
;for j = 1; j = j + 1; j ≤ N
    rightDoor = randomInt(3)
    guess = randomInt(3)

    ;; Pick one wrong, unchosen door
    wrongDoor = 0
    ;doWhile wrongDoor ∈ {guess, rightDoor}
        wrongDoor = mod(wrongDoor + 1, 3)
    ;stop
```

```

;; Change door
newDoor = 0
;doWhile newDoor ∈ {guess, wrongDoor}
    newDoor = mod(newDoor + 1, 3)
;stop

;if rightDoor = newDoor
    nCarsSwap = nCarsSwap + 1
;endif

;stop

result = "Stay: " + toString(nCarsStay / N) + "   Change: " +
        toString(nCarsSwap / N)

delete("rightDoor")
delete("guess")
delete("wrongDoor")
delete("newDoor")
delete("j")
delete("N")
delete("nCarsStay")
delete("nCarsSwap")

;return result

```

---

The output of this program might look like

```
Stay: 0.3291   Change: 0.6703
```

A more interactive example is the wave superposition simulator.

```

waveSim.prg
;; Wave simulator

setView(-10, 10, -10, 10)
int = [-10, 10, 0.1]

;; Arguments: λ1, λ2, v1, v2, A1, A2, δ

λ1 = 3
λ2 = 4
v1 = 0.6
v2 = 0.4
A1 = 2
A2 = 2.3
δ = 0
inputParams("λ1", "λ2", "v1", "v2", "A1", "A2", "δ")

k1 = 2·π/λ1

```



```

k2 = 2·π/λ2
ω1 = 2·π·ν1
ω2 = 2·π·ν2

clearView(1)

drawLines("wave1")
drawLines("wave2")
drawLines("waveΣ", "colour:red")

waveFunction1 = "x, t" ↦ "A1 · sin(k1·x - ω1·t)"
waveFunction2 = "x, t" ↦ "A2 · sin(k2·x - ω2·t + δ)"

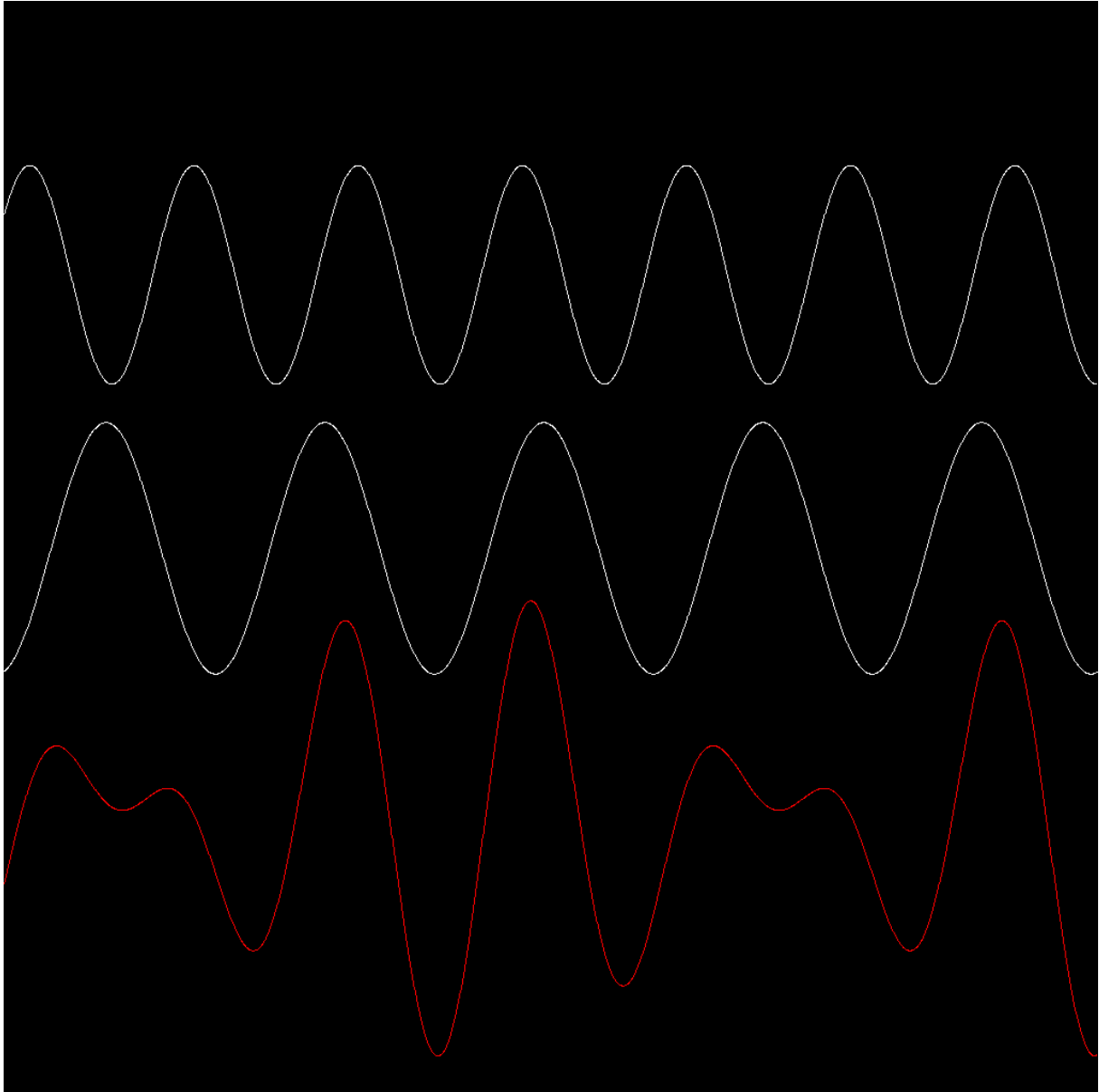
t = 0
tc = getTickCount(1)

;repeat
    wave1 = createGraph("waveFunction1(x, t) + 5", "x", int)
    wave2 = createGraph("waveFunction2(x, t)", "x", int)
    waveΣ = createGraph("waveFunction1(x, t) + waveFunction2(x,
        t) - 5", "x", int)
    redraw(1)
    t = t + (getTickCount(1) - tc) / 1000
    tc = getTickCount(1)
;indefinitely

delete("λ1")
delete("λ2")
delete("ν1")
delete("ν2")
delete("A1")
delete("A2")
delete("k1")
delete("k2")
delete("ω1")
delete("ω2")
delete("δ")
delete("doStopWaves")
delete("t")
delete("tc")
delete("wave1")
delete("wave2")
delete("waveΣ")
delete("int")

```

---



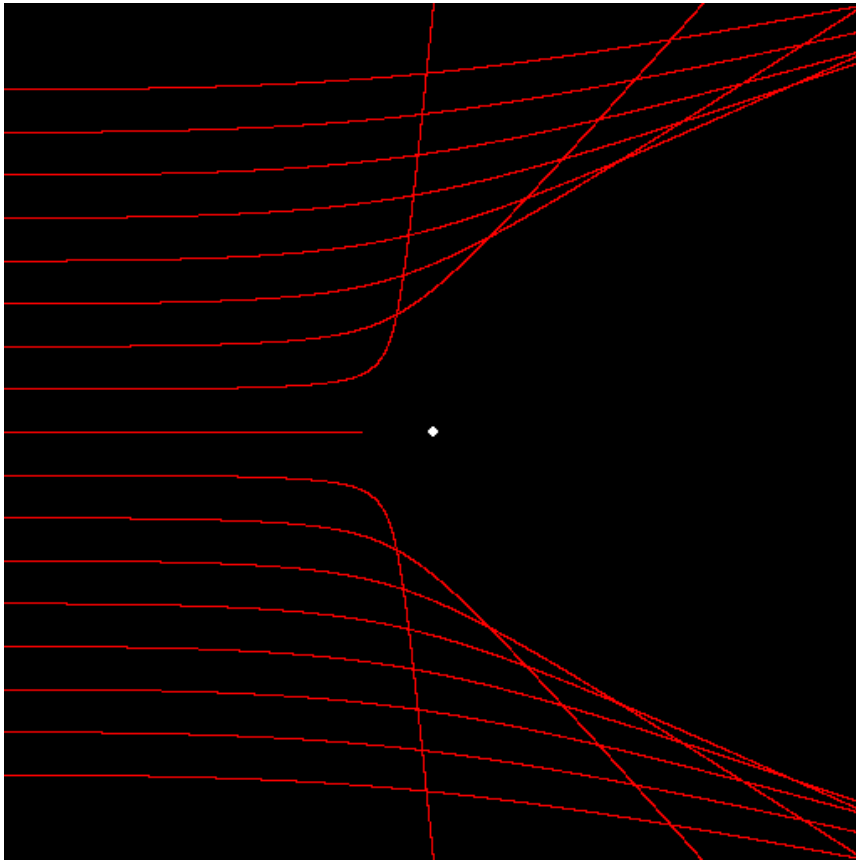
Using this program, the user can choose the parameters (wavelength, frequency, amplitude, and initial phase) of two sine waves, and then the program displays the two waves propagating in real-time, together with their superposition. Using this application, one can study wave phenomena such as constructive/destructive interference, beat, and standing waves.

Our next example makes use of the **computeParticleTrajectory** function to visualise Rutherford scattering of  $\alpha$  particles on gold atoms, for example.

```
rutherfordScattering2.prg
clearView(0)
drawCircle((0, 0), 0.1, "colour:white; border-colour:white")
 $\alpha$ traj = {}
drawSet(" $\alpha$ traj", "colour:red")
;iterate impactParameter:-8:8
     $\alpha$ traj =  $\alpha$ traj u computeParticleTrajectory("1/norm(r)^3 * r",
        "r", (-10, -impactParameter), (1, 0), 0, 100, 0.005)
redraw(0)
```

```
;enditerate
```

---



As our final example, we choose the very nice mirror simulator.

```
mirrorSim.prg
;; Mirror simulator

;if -identExists("t")
  t = choiceDialog("parabolic", "circular", "convex
    parabolic", "sine", "line")
;endif

;if t ∉ {"parabolic", "circular", "convex parabolic", "sine",
  "line"}
  t = choiceDialog("parabolic", "circular", "convex
    parabolic", "sine", "line")
;endif

xmin = -20

setView(xmin, 1, -10, 10)
clearView(1)

;if t = "parabolic"
  mirrorFunction = "y" ↦ "-(y^2) / 40"
;elseif t = "circular"
```

```

    mirrorFunction = "y" ↦ "sqrt(144 - y^2) - 12"
;elseif t = "convex parabolic"
    mirrorFunction = "y" ↦ "y^2 / 40"
;elseif t = "sine"
    mirrorFunction = "y" ↦ "sin(y/2)"
;elseif t = "line"
    mirrorFunction = "y" ↦ "0.8·y"
;endif

mirror = createImage("(mirrorFunction(y), y)", "y", [-10, 10,
    0.01])
beginDrawing(0)
drawLines("mirror", "colour:red")

;iterate y:-8:8
    ;; Incoming ray
    mFy = mirrorFunction(y)
    drawLine((xmin, y), (mFy, y))
    ;; Reflected ray
    dxdy = diff("mirrorFunction(y)", "y", y)
    ;;tangent = (dxdy, 1)
    ;;normal = (-1, dxdy)
    θ = 2·angle((-1, 0), (-1, dxdy))·sgn(dxdy)
    endpoint = (xmin, (-xmin + mFy)·tan(θ) + y)
    drawLine((mFy, y), endpoint)
;enditerate

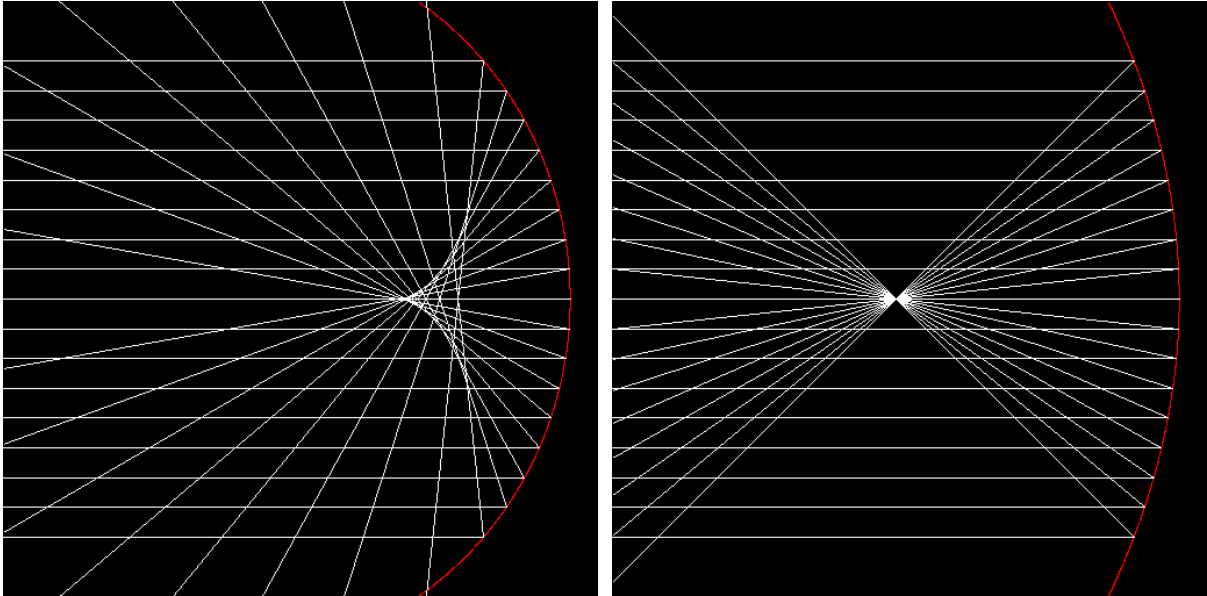
delete("mFy")
delete("θ")
delete("y")
delete("xmin")
delete("dxdy")
delete("endpoint")
delete("t")

endDrawing(0)
redraw(0)

```

---

The user can call it either with `mirrorSim(t := "circular")` or `mirrorSim(t := "parabolic")`, to simulate light reflection in a circular or a parabolic mirror. The results are interesting.



In the case of a circular mirror, when you send parallel light rays to the mirror, the reflected rays do *not* intersect in a common point, as is the case of a parabolic mirror.

The program **mirrorSim3**, which I do not list here, tells us that the “same” thing applies to three-dimensional mirrors as well, i.e. a parabolic mirror has a well-defined focus, whereas a spherical mirror does not. Try `mirrorSim3(t := "spherical")` and `mirrorSim3(t := "parabolic")`.

All these programs are included in a normal installation of AlgoSim. Hence, you can start any of them at the console. For instance, to start the wave simulator, simply execute

```
waveSim(0)
```

**Programming Reference Chart**

**Conditionals**

```

; if <cond>
. . .
; elsif <cond>
. . .
; elsif <cond>
. . .
; else
. . .
; endif
    
```

**For Loop**

```

; for <init>; <every-but-first>
. . .
; stop
    
```

*Typical Usage:*

```

; for x = 1; x = x + 1; x ≤ 10
. . .
; stop
    
```

**Repeat Loops**

*Simple Condition. Eval At End*

```

; repeat
. . .
; indefinitely
    
```

---

```

; repeat
. . .
; until <cond>
    
```

---

```

; repeat
. . .
; while <cond>
    
```

**Iterate Loop**

```

; iterate x:a:b, y:α:β, ...
. . .
; endIterate
    
```

*or*

```

; iterate x:a:b:c, y:α:β:γ, ...
. . .
; endIterate
    
```

**DoWhile Loop**

*Simple Condition. Eval At Beginning*

```

; doWhile <cond>
. . .
; stop
    
```

**In Any Loop**

```

; break
; continue
    
```

**Anywhere**

```

; exit
; return <value>
    
```

```

;; This line is a comment
    
```

## Database of Mathematical and Physical Constants

While working with physics and engineering problems, one often needs to input physical constants, such as the mass of an electron, the elementary charge, or Wien's displacement constant. Using AlgoSim, you no longer need to find these numbers in an external database. Instead, the function call **constant**("name of constant") returns the value of the desired constant.

For instance,

```
m = constant("proton mass")
1.672621637·10^-27

q = constant("elementary charge")
1.602176487·10^-19

v = 3.7↑6
3.7·10^6

B = 0.45
0.45

r = m·v^2/(q·v·B)
0.0858374024628
```

You do not need to remember the exact name of the constant. For instance, all these identifiers return the same constant:

- "Avogadro constant"
- "Avogadro number"
- "Avogadro's constant"
- "Avogadro's number"
- "The Avogadro constant"
- "The Avogadro number".

You can view and edit the database of constants yourself. There is one constants.asd file in the common directory, and one in the local directory. On a typical Windows 7 system, the complete paths are

- C:\Program Files (x86)\AlgoSim\constants.asd
- C:\Users\Andreas Rejbrand\AppData\Local\Rejbrand\AlgoSim\2.0\constants.asd

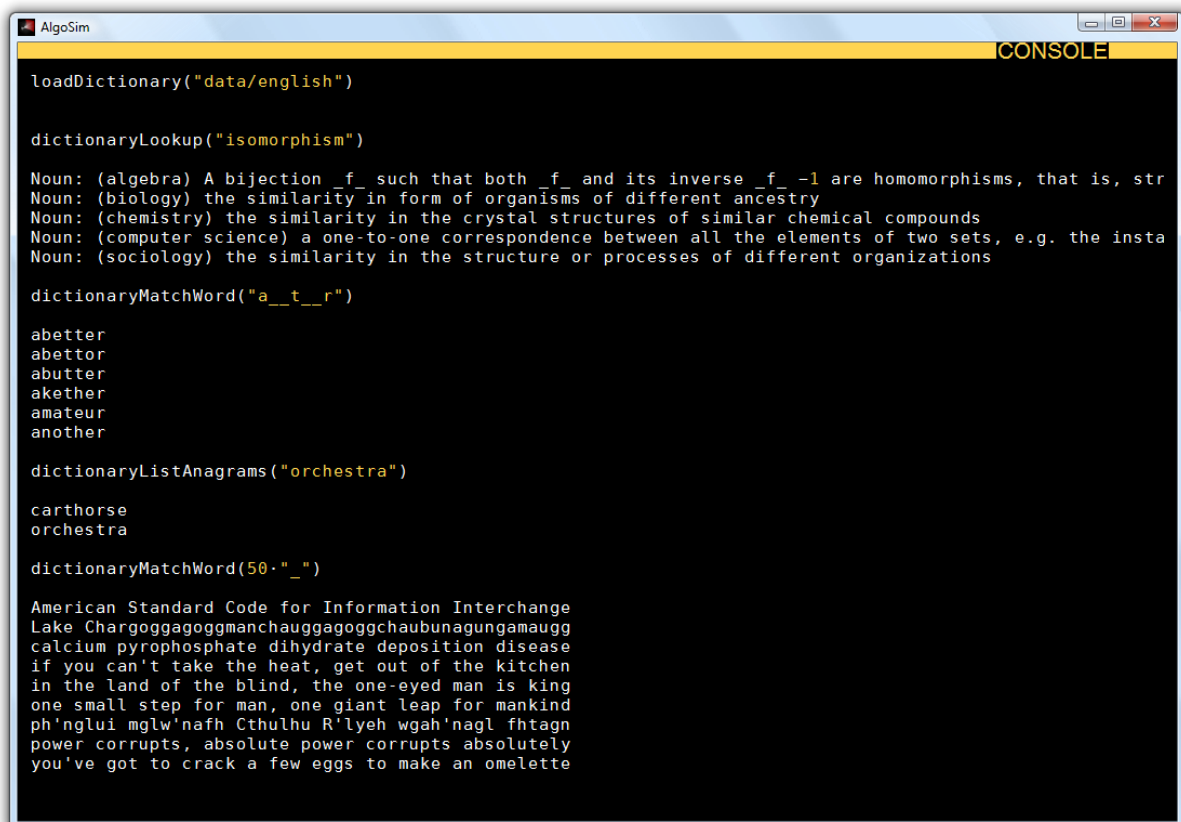
Simply double-click the local constants.asd file to add, remove, or change constants. The changes will only affect the current user, so each Windows user can have her own table. If the local constants.asd file is missing, the common table will be used. To restore the default operator table, remove the local one or replace it with the common one.

## Dictionaries

AlgoSim is not only about computations, but general reference as well. The most important (so far) implementation in this area is the dictionary interface. A dictionary is a text file where each row is an entry containing the word, the class (noun, verb, etc.) and the definition, separated by a horizontal tabulation character (U+0009). In the current version, AlgoSim is shipped with a comprehensive English dictionary. To load it to computer memory (which might take a few seconds), use the **loadDictionary** command and specify the file name of the dictionary, i.e. write

```
loadDictionary("data/english")
```

To look up a word in the dictionary, use the function **dictionaryLookup**, and specify the word as the argument (as a string, of course). You can also find words matching a pattern by using **dictionaryMatchWord**. There are functions related to palindromes and anagrams: **dictionaryListAnagrams** list all anagrams of a word, **dictionaryListWordsWithAnagrams** returns the list of all words that have at least one anagram (it might take a few hours to compile the list), and **dictionaryListPalindromes** returns the list of all non-trivial palindromes (such as “detartrated”). Furthermore, you can use **dictionaryGetWordSet** to obtain the set of all English words, literally.



```
AlgoSim
CONSOLE

loadDictionary("data/english")

dictionaryLookup("isomorphism")

Noun: (algebra) A bijection  $f$  such that both  $f$  and its inverse  $f^{-1}$  are homomorphisms, that is, str
Noun: (biology) the similarity in form of organisms of different ancestry
Noun: (chemistry) the similarity in the crystal structures of similar chemical compounds
Noun: (computer science) a one-to-one correspondence between all the elements of two sets, e.g. the insta
Noun: (sociology) the similarity in the structure or processes of different organizations

dictionaryMatchWord("a__t__r")

abettor
abettor
abutter
akether
amateur
another

dictionaryListAnagrams("orchestra")

carthorse
orchestra

dictionaryMatchWord(50-"_")

American Standard Code for Information Interchange
Lake Chargoggagoggmanchauggagoggchaubunagungamaugg
calcium pyrophosphate dihydrate deposition disease
if you can't take the heat, get out of the kitchen
in the land of the blind, the one-eyed man is king
one small step for man, one giant leap for mankind
ph'nglui mglw'nafh Cthulhu R'lyeh wgah'nagl fhtagn
power corrupts, absolute power corrupts absolutely
you've got to crack a few eggs to make an omelette
```

The English dictionary comes from the English-language Wiktionary at <http://en.wiktionary.org>, a Wiki-Media project. All data is licensed under the Creative Commons Attribution/Share-Alike License 3.0 found at <http://creativecommons.org/licenses/by-sa/3.0/legalcode>. A human-readable version of the license is available at <http://creativecommons.org/licenses/by-sa/3.0/>.



## Saving/Loading Data

You can save data using the **saveTextToFile**, **saveVectorToFile**, **saveMatrixToFile**, **savePixmapToFile**, **saveSoundToFile**, **saveStructToFile**, **savePointSetToFile**, **saveTableToFile**, and **saveTableDataToFile** functions. They all require two arguments: the object to save, and a file name. Remember that you can use the function **fileSaveDialog(0)** instead of writing the file name manually. Below is a description of the files created:

- **Text**

A plain-text UTF-8 file. Please use file extension \*.txt or \*.asd.

- **Vector**

A plain-text UTF-8 file; one component per row. Please use file extension \*.txt or \*.asd.

- **Matrix**

A plain-text UTF-8 file; one row per row, columns separated by tab characters (U+0009). Please use file extension \*.txt or \*.asd.

- **Pixmap**

Depending on the file extension, an AlgoSim Pixmap (\*.asd), Windows Bitmap (\*.bmp), or Portable Network Graphics (\*.png) raster image file will be created.

- **Sound**

A PCM WAV (\*.wav) file will be created. Please use extension \*.wav.

- **Structures**

An AlgoSim structure file will be created. This is a binary, non-plain text file. Please use extension \*.asd.

- **Table**

An AlgoSim table file will be created. Both table data and table style will be saved. This is a binary, non-plain text file. Please use extension \*.asd.

- **Table Data**

A plain-text UTF-8 file is created; one row per row, columns separated by tab characters (U+0009). Please use file extension \*.txt or \*.asd.

- **Point Set**

An AlgoSim point set file is created. If the set to be saved contains other elements than real points (vectors), only the real points (vectors) will be saved, and all other elements will be ignored. Please use extension \*.asd.

To each `save*ToFile` function, there is a corresponding `load*FromFile` function. This takes one single argument, the file name, and returns the object. However, the simplest way of opening a \*.txt, \*.asd, \*.bmp, \*.png, or \*.wav file in AlgoSim is to drag it to the AlgoSim main window from Windows Explorer. Try this!

Moreover, the simplest way of *saving* data files from AlgoSim, is to use the advanced variable manager. This can be opened by double-clicking somewhere inside the Identifiers list view in the main window.

This ends the first part of this User's Guide. The following part describes all built-in functions in detail.

## Appendix I: Function Reference

### $\forall$

$\forall(S, \text{var}, \text{expr})$  returns true if every element in the set  $S$  satisfies the boolean-valued expression  $\text{expr}$  in the variable  $\text{var}$ .

Example:  $S := [1, 100]$   
 $\forall(S, "n", "isPerfect(n)") = \text{false}$

### $\exists$

$\exists(S, \text{var}, \text{expr})$  returns true if there exists (at least one) element in the set  $S$  satisfying the boolean-valued expression  $\text{expr}$  in the variable  $\text{var}$ .

Example:  $S := [1, 100]$   
 $\exists(S, "n", "isPerfect(n)") = \text{true}$

### $\prod$

$\prod(\text{expr}, \text{var}, a, b)$  returns the product  $\text{expr}(a) \cdot \text{expr}(a + 1) \cdot \dots \cdot \text{expr}(b)$  where  $\text{expr}$  is a string representing a real-valued expression in  $\text{var}$ .  $\text{var}$  is a string representing a valid identifier, used as the independent variable in  $\text{expr}$ .  $a$  and  $b$  are integers, corresponding to the first and last factor of the product, respectively; thus, there are exactly  $b - a + 1$  factors in the product.

### $\sum$

$\sum(\text{expr}, \text{var}, a, b)$  returns the sum  $\text{expr}(a) + \text{expr}(a + 1) + \dots + \text{expr}(b)$  where  $\text{expr}$  is a string representing a real-valued expression in  $\text{var}$ .  $\text{var}$  is a string representing a valid identifier, used as the independent variable in  $\text{expr}$ .  $a$  and  $b$  are integers, corresponding to the first and last term of the summation, respectively; thus, there are exactly  $b - a + 1$  terms in the sum.

Examples:  $\sum("1/n!", "n", 0, 100) = 2.71828182846$

### $\int$

$\int(\text{expr}, \text{var}, a, b)$  computes the definite integral of  $\text{expr}$  (the integrand, a string representing a real-valued expression) with the variable  $\text{var}$ , a string representing a valid identifier that may occur inside  $\text{expr}$ , between the values  $a$  and  $b$  of the independent variable  $\text{var}$ .

Examples:  $\int("sin(x)", "x", 0, \pi) = 2$   
 $\int("exp(sin(x))", "x", 0, 5) = 7.18911925363$

### **abs**

$\text{abs}(x)$  returns the absolute value of the real or complex number  $x$ , i.e. the distance between the point  $x$  on the real number line or complex plane and the origin.

### **absVect**

$\text{absVect}(v)$  returns the vector  $v$  with all components replaced by their 2-norm.

### **addDays**

$\text{addDays}(d, n)$  returns the date and time structure corresponding to  $n$  days after the date structure  $d$ .

Example:

$\text{addDays}(\text{encodeDate}(2010, 06, 19), -1)$

year: 2010  
 month: 6  
 day: 18  
 weekOfYear: 24  
 dayOfYear: 169  
 dayOfWeek: 5  
 hour: 0

minute: 0  
 second: 0  
 millisecond: 0

### **addHours**

$\text{addHours}(d, n)$  returns the date and time structure corresponding to  $n$  hours after the date and time structure  $d$ .

### **addMemberToStruct**

$\text{addMemberToStruct}(\text{str}, \text{ident}, \text{val})$  adds to the structure  $\text{str}$  (a string representing the identifier of a structure variable) a new member  $\text{ident}$  (a string) with value  $\text{val}$  (a number, a string, a boolean, or another structure).

### **addMilliseconds**

$\text{addMilliseconds}(d, n)$  returns the date and time structure corresponding to  $n$  milliseconds after the date and time structure  $d$ .

### **addMinutes**

$\text{addMinutes}(d, n)$  returns the date and time structure corresponding to  $n$  minutes after the date and time structure  $d$ .

### **addSeconds**

$\text{addSeconds}(d, n)$  returns the date and time structure corresponding to  $n$  seconds after the date and time structure  $d$ .

### **addVectorComponent**

$\text{addVectorComponent}(s, x)$  adds the real or complex number  $x$  to the real or complex vector named  $s$ , a string representing the valid identifier of an existing real or complex vector variable, as a new component. Thus the dimension of the vector named  $s$  will increase by one.

### **addWeeks**

$\text{addWeeks}(d, n)$  returns the date and time structure corresponding to  $n$  weeks after the date and time structure  $d$ .

Example:

$\text{addWeeks}(\text{encodeDate}(2010, 06, 19), 10)$

year: 2010  
 month: 8  
 day: 28  
 weekOfYear: 34  
 dayOfYear: 240  
 dayOfWeek: 6  
 hour: 0  
 minute: 0  
 second: 0  
 millisecond: 0

### **angle**

$\text{angle}(v1, v2)$  returns the angle between the vectors  $v1$  and  $v2$  [as given by the 2-norm].

### **animateTrajectory**

$\text{animateTrajectory}(S, I, b)$  animates the particle or flow trajectory set  $S$ , created using  $\text{computeParticleTrajectory}$  or  $\text{computeFlowTrajectory}$ .  $S$  is required to be a subset of  $\mathbb{R}^2$ .  $I$  is the number of steps in time per frame. If  $b = \text{true}$ , the particle's trace is shown on the screen.

Example: Bouncing ball in gravity in a 2D box

$\text{trajectory} := \text{computeParticleTrajectory}(" (0, -1)", "r", (-10, 10), (1, 0), 0, 100, 0.01, (-10, 10, -10, 10), 0.8)$   
 $\text{animateTrajectory}("trajectory", 30, \text{true})$

### **animateTrajectory3**

$\text{animateTrajectory3}(S, I, b)$  animates the particle or flow trajectory set  $S$ , created using  $\text{computeParticleTrajectory}$  or  $\text{computeFlowTrajectory}$ .  $S$  is required to be a subset of  $\mathbb{R}^3$ .  $I$

is the number of steps in time per frame. If `b=true`, the particle's trace is shown on the screen.

Example: Bouncing ball in gravity in a 3D box  
`trajectory := computeParticleTrajectory("(0, 0, -1)", "r",  
(-10, 10, 10), (1, 2, 0), 0, 100, 0.01, (-10, 10, -10, 10, -10,  
10), 0.8)  
animateTrajectory3("trajectory", 30, true)`

**arccos**

`arccos(x)` is the inverse of `cos(x)` restricted to the interval  $[0, \pi]$ . For the real `arccos` function, `x` must be in the interval  $[-1, 1]$ .

**arccosh**

`arccosh(x)` is the inverse of `cosh(x)`.

**arccot**

`arccot(x)` is the inverse of `cot(x)` restricted to the interval  $]\pi, 0]$ .

**arccoth**

`arccoth(x)` is the inverse of `coth(x)`.

**arccsc**

`arccsc(x)` is the inverse of `csc(x)` restricted to the interval  $[-\pi/2, 0[$ .

**arccsch**

`arccsch(x)` is the inverse of `csch(x)`.

**arcsec**

`arcsec(x)` is the inverse of `sec(x)` restricted to the interval  $[0, \pi/2[$ .

**arcsech**

`arcsech(x)` is the inverse of `sech(x)`.

**arcsin**

`arcsin(x)` is the inverse of `sin(x)` restricted to the interval  $[-\pi/2, \pi/2]$ . For the real `arcsin` function, `x` must be in the interval  $[-1, 1]$ .

**arcsinh**

`arcsinh(x)` is the inverse of `sinh(x)`.

**arctan**

`arctan(x)` is the inverse of `tan(x)` restricted to the interval  $]-\pi/2, \pi/2[$ .

**arctan2**

`arctan2(x, y)` returns the angle between the positive `x`-axis and the vector from the origin to the point  $(x, y) \in \mathbb{R}^2$ , within the interval  $]-\pi, \pi]$ .

**arctanh**

`arctanh(x)` is the inverse of `tanh(x)`.

**arg**

`arg(z)` returns the complex argument of the complex number `z`, i.e. the angle between the positive real (`x`) axis and the vector from the origin to `z` in the complex plane, in the interval  $]-\pi, \pi]$ .

Thus if  $z = x + yi$  where `x` and `y` are real numbers, `arg(z) = arctan2(x, y)`.

**assert**

`assert(b)` will yield an exception (error) if `b`, a boolean, is false. If `b` is true, `assert` will do nothing.

**augment**

`augment(M1, M2)` returns the real or complex matrix composed of the real or complex matrices `M1` and `M2` side by side, if only they have the same number of rows.

`augment(M, v)` returns the real or complex matrix composed of the real or complex matrix `M` with the real or complex vector `v` added to the right of `M`, if only `M` has as many rows as the dimension (number of entries) of `v`.

**baseNInput**

`baseNInput(s, N)` returns the integer represented by the string `s` in base-`N`, where `s` is a string of the `N - 1` base-`N` digits `0, 1, 2, ..., 8, 9, A, B, ..., Y, Z`.

For example, `baseNInput("567", 10)`, `baseNInput("FF", 16)`, `baseNInput("1000", 2)`, `baseNInput("43HA2", 20)`, and `baseNInput("43HA2", 16)` will return `567`, `255`, `8`, `671002`, and the error message "The base-`N` digit `H` is not used in base `16`", respectively.

By default, the infix operator `#` is mapped to the `baseNInput` function. Its first operand is a raw string, and the second operand is an integer. Hence

`567#10`, `FF#16`, `1000#2`, `43HA2#20`, and `43HA2#16` will return the same results as those obtained above.

**beep**

`beep(0)` produces a brief Windows default beep sound.

`beep(0, 1)` produces a Windows default "Asterisk" sound.  
`beep(0, 2)` produces a Windows default "Exclamation" sound.  
`beep(0, 3)` produces a Windows default "Critical Stop" sound.  
`beep(0, 4)` produces a Windows default "Question" sound.

`beep(1, freq, dur)` produces a sine tone with frequency `freq` [Hz] and duration `dur` [ms] using Windows API.

**beginDrawing**

`beginDrawing(0)` increases the common halt index of the 2D and 3D visualization windows by one. Initially, this index is equal to zero, and if it is positive and an object (set, pixmap, geometrical entity, etc.) is added to the window, the window is not redrawn on-screen.

**Bernstein**

`Bernstein(i, n, x)` returns the `i`th Bernstein basis polynomial of degree `n` at `x`.

$Bernstein(i, n, x) = \text{comb}(n, i) \cdot x^i \cdot (1-x)^{(n-i)}$ , if  $i = 0, 1, \dots, n$  and `n` are integers.

**bessel**

`bessel(n, x)` is the `n`th order ( $n \in \mathbb{N}$ ) Bessel function of the first kind evaluated at  $x \in \mathbb{R}$ .

**Bézier**

`Bézier(S)` returns the Bézier curve (as a point set) of the control points in `S`, a set containing `n`-dimensional vectors.

Example: Cubic Bézier curve (i.e. four control points)  
`be := Bézier({(1, 3), (2, 6), (3, -7), (5, 3)})  
drawLines("be")`

**CaesarCipher**

`CaesarCipher(str, n)` transforms the string `str` using the Caesar cipher, i.e. shifts all English letters `n` steps to the right along the alphabet (modulo 26).

`CaesarCipher(str)` assumes `n = 3`, and the inverse of `str`  $\mapsto$  `CaesarCipher(str, n)` is `str`  $\mapsto$  `CaesarCipher(str, -n)`.

**ceil**

$\lceil x \rceil = \text{ceil}(x)$  returns the smallest integer greater than or equal to  $x$ , i.e. rounds  $x$  to the nearest integer in the direction of  $+\infty$ .

**centerOfMass**

$\text{centerOfMass}(S)$  returns the center of mass as a real vector in the set  $S$  of real vectors.

Example:  $\text{centerOfMass}(\{(1, 3, 2), (2, 1, 3), (5, 2, 1), (-1, 2, 3), (2, 1, 3)\}) = (1.75, 2, 2.25)$

**changeInstrument**

$\text{changeInstrument}(n)$  changes the current MIDI musical instrument to  $n \in [0, 127]$  (an integer).

**changeMidiVolume**

$\text{changeMidiVolume}(x)$  sets the sound intensity (volume) of MIDI sounds to  $x \in [0, 1]$  on all channels.

$\text{changeMidiVolume}(l, r)$  sets the sound intensity of MIDI sounds to  $l \in [0, 1]$  and  $r \in [0, 1]$  on the left and right stereo audio channel, respectively.

**choiceDialog**

$\text{choiceDialog}(S_1, S_2, \dots, S_n)$  displays a dialog in which the user can choose between the strings  $S_1, S_2, \dots, S_n$ . The choice of the user, one of the  $S_k$ 's, is returned.

**chooseDirectoryDialog**

$\text{chooseDirectoryDialog}(0)$  displays a directory browser dialog and returns the selected directory.

**chr**

$\text{chr}(n)$  returns the Unicode character with integer codepoint  $n$ . In addition, if  $n \in [0, 127]$ , this is the same as the  $n$ th ASCII character.

Example:  $\text{chr}(8882) = "\<"$

**chrBlock**

$\text{chrBlock}(s)$  returns the Unicode block name of the Unicode character (i.e., single-character string)  $s$ .

Example:  $\text{chrBlock}("\<") = \text{"Mathematical Operators"}$

**chrDescribe**

$\text{chrDescribe}(s)$  returns the description of the Unicode character (i.e., one-character string)  $s$ .

Example:  $\text{chrDescribe}("\<") = \text{"NORMAL SUBGROUP OF"}$

**Ci**

$\text{Ci}(x)$  is the cosine integral.

$\text{Ci}(x) = \gamma + \ln(x) + \int (\cos(t) - 1)/t \, dt$  from 0 to  $x$  where

$\gamma = 0.57721\ 56649\ 01532\dots$  is the Euler-Mascheroni constant.

**clearView**

$\text{clearView}(0)$  clears the 2D visualization window, i.e. removes all drawings.

**clearView3**

$\text{clearView3}(0)$  clears the current 3D visualization window, i.e. removes all drawings in it.

**cls**

$\text{cls}(0)$  clear the text console.

**comb**

$\text{comb}(n, k) = n!/(k! \cdot (n-k)!)$  returns the number of subsets with  $k$  elements that can be created using elements from a set of  $n$  elements.

$\text{comb}(n, k)$  is also known as a binomial coefficient. It is required that  $k \in [0, n]$ .

**combineStructs**

$\text{combineStructs}(\text{str1}, \text{str2})$  returns the structure, the set of its members being the union of the corresponding sets of the structures  $\text{str1}$  and  $\text{str2}$ .

Example:

$\text{combineStructs}(\text{date}(0), \text{time}(0))$

```
year: 2010
month: 6
day: 23
weekOfYear: 25
dayOfYear: 174
dayOfWeek: 3
hour: 15
minute: 20
second: 34
millisecond: 774
```

because

$\text{date}(0)$

```
year: 2010
month: 6
day: 23
weekOfYear: 25
dayOfYear: 174
dayOfWeek: 3
```

$\text{time}(0)$

```
hour: 15
minute: 20
second: 48
millisecond: 615.
```

**complexCoords**

$\text{complexCoords}(S)$  returns the  $\mathbb{R}^2$  subset naturally corresponding to the  $\mathbb{C}$  subset  $S$ . Useful for plotting complex sets using  $\text{drawSet}$ ,  $\text{drawLines}$ , etc.

**complexIdentityMatrix**

$\text{complexIdentityMatrix}(n)$  returns the  $n$ -dimensional complex identity matrix, i.e. the  $n \times n$  square matrix with all entries equal to  $\delta(m, n)$  where  $\delta$  is the Kronecker delta function.

**complexZeroMatrix**

$\text{complexZeroMatrix}(m, n)$  returns the complex  $m \times n$  matrix with all zero entries.

**computeFlowTrajectory**

$\text{computeFlowTrajectory}(\text{expr}, \text{var}, r, t_0, t_1, \delta t)$  returns the curve in  $n$ -dimensional space traced out by a particle that has the velocity  $\text{expr}(\text{var})$  at point  $\text{var}$  in space.  $\text{expr}$  is thus a real  $n$ -dimensional vector-valued function in  $\text{var}$ , a valid identifier for a  $n$ -dimensional position vector.  $r \in \mathbb{R}^n$  is the initial position of the particle.  $t_0$  is the initial time of the simulation, and  $t_1$  is the final time.  $\delta t$  is the temporal resolution.

Example:  $a := 2$   
 $b := 3.001$

```
vectorField := createVectorField("(1 + a·x^2·y - b·x - x, -
a·x^2·y + b·x)", "x, y", [0, 10, 0.25]^2)
drawVectorField("vectorField", "colour:#333333")
```

```
flow := computeFlowTrajectory("(1 + a·r_1^2·r_2 -
b·r_1 - r_1, -a·r_1^2·r_2 + b·r_1)", "r", (1, 4), 0, 100, 0.01)
drawLines("flow", "colour:gold")
```

### computeMatrix

computeMatrix(m, n, expr, vars) computes a m×n matrix from a function expr of the matrix element's indices, vars, a comma-separated list of valid identifiers.

Example: Computes a 10 by 10 multiplication table  
 computeMatrix(10, 10, "m·n", "m, n")

Computes a list of the 100 first prime numbers  
 computeMatrix(100, 1, "prime(m)", "m, n")

Computes a list of the 100 first prime numbers, and their 5 first powers  
 computeMatrix(100, 5, "prime(m)^n", "m, n")

### computeParticleTrajectory

computeParticleTrajectory(expr, var, r, v, t0, t1, δt) computes the curve in n-dimensional space that a particle will trace out if the force field is expr, a string representing a real-valued expression in var, a string representing a valid identifier.  $r \in \mathbb{R}^n$  is the particle's initial position, and  $v \in \mathbb{R}^n$  is its initial velocity. t0 and t1 is the initial and final time, respectively. δt is the temporal resolution, typically 0.001 for a high-res simulation of a few seconds. To plot the trajectory, simply use drawSet/drawLines or drawSet3/drawLines3, if n = 2 or n = 3, respectively.

computeParticleTrajectory(expr, var, r, v, t0, t1, δt, BB), where  $BB \in \mathbb{R}^{2n}$ , will impose a box which the particle will not be able to move outside of; instead, if it hits the inside of the box, it will bounce back. The two first components of BB are the lowest and highest bounds of the first dimension, the third and fourth components are the lowest and highest bounds of the second dimension, and so on. Thus (-10, 10, -10, 10) is a symmetric 20×20 2D box, and (-10, 10, -10, 10, -10, 10) is a symmetric 20×20×20 3D box, and so on.

computeParticleTrajectory(expr, var, r, v, t0, t1, δt, BB, f) uses the box BB and the factor f of speed preservation at each bounce. If f = 0, all kinetic energy is lost at every (in practice, the first) bounce, and if f = 1 it will continue to bounce indefinitely.

Example: Bouncing ball in gravity in a 2D box  
 trajectory := computeParticleTrajectory("(0, -1)", "r", (-10, 10), (1, 0), 0, 100, 0.01, (-10, 10, -10, 10), 0.8)
 drawSet("trajectory", "colour:red")

### constant

constant(s) returns the value of the physical/mathematical constant named s, a string.

For example, constant("electron mass") = 9.10938215·10<sup>-31</sup>.

The database of constants is stored in constants.asd in the AlgoSim installation directory. You can edit this file (by double-clicking it) to add, change, and remove constants.

### containsDuplicate

containsDuplicate(v) returns true if the real or complex vector v contains the same number at at least two different components.

### contents

contents(S) returns a list of the contents of the set S as the number of elements of each possible data type.

Example: contents({1, i, (1, 2), "test"})  
 1 real number(s)  
 1 complex number(s)  
 1 real vector(s)  
 1 string(s)

contents(s) returns a list of the contents of the set named s, a string representing the valid identifier of a set variable.

Remark: Calling contents with a reference to the set, i.e. using a string as argument, is generally much faster than passing the entire set (by value, copying the set data) to the function.

### coprime

$m \perp n = \text{coprime}(m, n)$  returns True if m and n are relatively prime, i.e. if  $\text{gcd}(m, n) = 1$ , and False otherwise.

Example:  $19 \perp 8 = \text{True}$ .

### copyFile

copyFile(f1, f2) copies the file f1 (a string containing a valid file name) to f2 (a string containing a valid file name).

### copyMatrixToClipboard

copyMatrixToClipboard(M) copies the real or complex matrix M to Window's clipboard.

### copyPixmapToClipboard

copyPixmapToClipboard(pm) copies the pixmap pm to Window's clipboard, as a bitmap.

### copyStructToClipboard

copyStructToClipboard(str) copies the structure str to clipboard, in plain-text format.

### copyTableToClipboard

copyTableToClipboard(T) copies the table T to Window's clipboard.

### copyTextToClipboard

copyTextToClipboard(s) copies the string s to clipboard.

### copyVectorToClipboard

copyVectorToClipboard(v) copies the real or complex vector v to Window's clipboard.

### cos

cos(x) returns the cosine of x. x is a real or complex number.

Construct the unit circle

$$x^2 + y^2 = 1$$

in  $\mathbb{R}^2$ . Draw the line from the origin to the point P at this circle, such that the angle to this line, counted from the positive x-axis (anticlockwise is the positive direction) is equal to x. Then cos(x) is the x-coordinate of P.

For a general complex number z, Euler's identity

$$\cos(z) = (1/2) \cdot (\exp(iz) + \exp(-iz))$$

defines cos(z). exp is the complex exponential function, defined such that

$$\exp(z) = e^{\text{Re } z} \cdot (\cos(\text{Im } z) + i \sin(\text{Im } z))$$

where i is the imaginary unit ( $i^2 = -1$ ) and Re z and Im z are the real and imaginary parts of z, respectively.

**cosh**

cosh(x) is the hyperbolic cosine, i.e.  $\cosh(x) = (1/2) \cdot (e^{ix} + e^{-ix})$ .

**cot**

$\cot(x) = \cos(x) / \sin(x)$ . x is a real or complex number.

**coth**

coth(x) is the hyperbolic cotangent, i.e.  $\coth(x) = \cosh(x) / \sinh(x)$ .

**cototient**

cototient(n) = n - totient(n) is the number of positive integers less than or equal to n that are \*not\* coprime to n.

**count**

count(S, var, expr) returns the number of elements in the set S which satisfy the boolean expression expr, a string containing a boolean expression in one variable, var, which will iterate over all elements in the set S.

Examples: `count({1, 4, 6, 8}, "x", "x>5") = 2`  
`count({(1, 0), (1, 1), (0, 1)}, "x", "norm(x) = 1") = 2`  
`count([1, 100], "n", "isPrime(n)") = 25`

count(v, var, expr) returns the number of components in the real or complex vector v that satisfy the boolean expression expr in the variable var.

Examples: `count((1, 3, 3, 4, 5), "x", "x=3") = 2`

count(M, var, expr) returns the number of elements in the real or complex matrix M that satisfy the boolean expression expr in the variable var.

**createColouredPlane**

createColouredPlane(expr, xmin, xmax, xres, ymin, ymax, yres) creates a set describing a coloured plane using the function expr of x and y. expr returns a colour code, e.g. using the rgb or hsv function.  $x \in [xmin, xmax]$ ,  $y \in [ymin, ymax]$ , and the resolution is xres and yres in the horizontal and vertical direction, respectively. The resulting set (coloured plane) is drawn using the drawColouredPlane function.

Example:

```
Superposition of two water waves.
ψ := "r" ↦ "sin(4·norm((2, 2) - r))/6"
Φ := "r" ↦ "sin(4·norm((0, 0) - r))/6"
S := "r" ↦ "ψ(r) + Φ(r)"
waves := createColouredPlane("hsv(90 - 270·S((x, y)),
1, 1)", -10, 10, 0.1, -10, 10, 0.1)
drawColouredPlane("waves")
```

**createComplexMatrix**

createComplexMatrix(s, m, n) creates a new complex m×n matrix with the name s (which must be a string, and a valid identifier), and opens the matrix editor so its entries can be entered.

**createDlaFractal**

createDlaFractal(w, h, n) creates a DLA (diffusion-limited aggregation) fractal pixmap of width w and height h, using n iterations.

Example: `createDlaFractal(500, 500, 100 000)`

**createGraph**

createGraph(expr, var, int) returns the graph  $\{(x, y) : y = \text{expr}(var), var \in \text{int}\}$ , where expr (a string, an expression) is a function of var (a string, a valid identifier), and the pre-image is the interval (or, generally, set) int.

Examples: `set := createGraph("sin(x)", "x", [-10, 10, 0.001])`

`drawSet("set")`

**createGraph3**

createGraph3(expr, vars, set) creates the three-dimensional graph  $\{(x, y, z) \in \mathbb{R}^3 : (x, y, z) = (x, y, \text{expr}(x, y)), (x, y) \in \text{set}\}$  of expr, a string representing a real-valued expression in two variables, listed in vars, a comma-separated string of valid identifiers, where the independent variables, as a vector (x, y), take on each value in set, a set of planar vectors.

Examples: `surf := createGraph3("sin(sqrt(x^2 + y^2))", "x, y", [-10, 10, 0.1]^2)`  
`drawSet("surf")`

Important Remark: In most cases, createSurfParamCurves and drawSurfParamCurves are much more efficient and visually pleasing than createGraph3/createImage and drawSet3.

createGraph3 and drawSet3 draws a surface as a uniform point set, that is, to make a dense surface you need as many points as required by the screen resolution, which takes very long time to compute. In addition, because this is a mere point set and not a true 3D surface, realistic lightning is not applied, and so it might be difficult to view the surface.

createNet, createImage, and drawSet3 partly resolve this problem. Using createNet, you can create a grid in the parameter plane of the surface, and then apply createImage to these parameter lines instead. When drawn using drawSet3, only the parameter curves (at some distance and resolution) are drawn, thus greatly simplifying the interpretation of the resulting image. In addition, only a small fraction of the points required by the naïve approach (createGraph3 and drawSet3) are required. (Indeed, we only draw the surface's parameter curves, not the entire surface.) However, the parameter curves are drawn by simply plotting points of them, and so this method is as insufficient as drawSet as compared to drawLines when drawing a 2D curve (or drawSet3 as compared to drawLines3).

To draw a surface parameter curves using polyline approximation (usually not even a visible loss in quality, but extremely fast), use createSurfParamCurves and drawSurfParamCurves instead.

**createImage**

createImage(expr, var, set) creates the image of the set set under the function expr of the variable var.

Examples: `spiral := createImage("(FresnelC(t), FresnelS(t))", "t", [-10, 10, 0.01])`  
`drawLines("spiral")`

`spiral := createImage("(3·cos(t), 3·sin(t), t/2, hsv(10-t, 1, 1))", "t", [-30, 30, 0.01])`  
`drawColouredLines3("spiral")`

`createImage("2·x", "x", {"test", 10, (1, 2)}) = {"testtest", 20, (2, 4)}`

**createImageOfVectors**

createImageOfVectors works exactly as createImage, but will only accept real vector-valued functions. In return, it is much faster than the more versatile createImage.

**createMatrix**

createMatrix(s, m, n) creates a new m×n matrix with the name s (which must be a string, and a valid identifier), and opens the matrix editor so its entries can be entered.

**createNet**

createNet(x0, x1, δx, Δx, y0, y1, δy, Δy) returns a planar set with a 2D grid, in the region  $x \in [x0, x1]$ ,  $y \in [y0, y1]$ . The

horizontal lines have the resolution  $\delta x$ , and the vertical lines have the resolution  $\delta y$ . The spacing between vertical lines is  $\Delta x$ , and the spacing between horizontal lines is  $\Delta y$ .

Example: A square grid.

```
net := createNet(-10, 10, 0.01, 1, -10, 10, 0.01, 1)
drawSet("net")
```

An illustrative way to draw a grid cylinder with radius 4.

```
net := createNet(0, 2·π, 0.01, π/12, -10, 10, 0.1, 1)
paramNet := createImage("C4, r_1, r_2)", "r", net)
cylinder := cylindricalCoords(paramNet)
drawSet3("cylinder")
```

## createPixmap

createPixmap(w, h) creates and returns a new pixmap with width w and height h.

## createSet

createSet(expr) returns the set of all points (x, y) in  $[-10, 10]^2$  that satisfies expr, a string containing an expression in x and y. Typically, this expression is a boolean statement utilizing a relation operator, such as =, <, or >. The default resolution 0.05 is used.

createSet(expr, res) uses the resolution res.

createSet(expr, xmin, xmax, ymin, ymax) tests only points within  $[xmin, xmax] \times [ymin, ymax]$  with the default resolution 0.05.

createSet(expr, xmin, xmax, ymin, ymax, res) tests only points within  $[xmin, xmax] \times [ymin, ymax]$  with the resolution res.

Example: Draw the (open) unit disk:

```
set := createSet("x^2 + y^2 < 1", -1, 1, -1, 1)
drawSet("set")
```

Remark: The open unit disk may also be parametrized via  $\gamma(r, \varphi) = (r \cdot \cos(\varphi), r \cdot \sin(\varphi))$  where  $r \in [0, 1]$  and  $\varphi \in [0, 2 \cdot \pi[$ . The parametric approach is much faster. The implicit createSet function is more useful for sets that cannot be (easily) parametrised.

## createSineTone

createSineTone(f, d) returns a sound of a pure sine tone of frequency f [Hz] with duration d [s].

Example: createSineTone(400, 1) creates a 400 Hz sine tone with a duration of one second.

## createStruct

createStruct(I1, V1, I2, V2, ..., In, Vn) creates a structure with identifiers I1, I2, ..., In with values V1, V2, ..., Vn. Every I<sub>k</sub> must be a string (and a valid identifier), and every V<sub>k</sub> must be a number, a string, a boolean, or another structure.

Examples:

```
createStruct("firstName", "Andreas", "lastName", "Rejbrand",
"yearOfBirth", 1987, "IQ", ∞)
```

```
firstName: Andreas
lastName: Rejbrand
yearOfBirth: 1987
IQ: ∞
```

```
createStruct("date", date(0), "time", time(0))
```

```
date:year: 2010
date:month: 6
date:day: 23
date:weekOfYear: 25
```

```
date:dayOfYear: 174
date:dayOfWeek: 3
time:hour: 15
time:minute: 3
time:second: 43
time:millisecond: 192
```

```
ans:time:millisecond = 192
```

## createSurfParamCurves

createSurfParamCurves(expr, vars, x0, x1, y0, y1) creates a special set with the parameter curves of the surface described by expr, a real three-dimensional vector-valued function in the two variables listed in the comma-separated string vars. The first variable will run through  $[x0, x1]$ , and the second through  $[y0, y1]$ . The resulting set is drawn by the drawSurfParamCurves function.

Examples: garden := createSurfParamCurves("(x, y, sin(x·randomReal(1)) - sin(y))", "x, y", -10, 10, -10, 10)
drawSurfParamCurves("garden", "colour:gold")

```
grass := createSurfParamCurves("(x, y, randomReal(1))", "x, y", -10, 10, -10, 10)
drawSurfParamCurves("grass", "colour:forestgreen")
```

```
surf := createSurfParamCurves("(x, y, sin(sqrt(x^2 + y^2))), hsv(x^2 + y^2, 1, 1))", "x, y", -10, 10, -10, 10)
drawColouredSurfParamCurves("surf")
```

// Superposition of water waves (without attenuation...)

```
ψ := "r" ↦ "sin(4·norm((2, 2) - r))/6"
Φ := "r" ↦ "sin(4·norm((0, 0) - r))/6"
S := "r" ↦ "ψ(r) + Φ(r)"
set := createSurfParamCurves("(x, y, S((x, y))), hsv(90 - 270·S((x, y)), 1, 1)", "x, y", -10, 10, -10, 10)
drawAxes3(1)
drawColouredSurfParamCurves("set")
```

See also: drawSurfParamCurves. Compare to: createSet, createGraph3, drawSet3, createNet

## createTable

createTable(s, m, n) creates a new  $m \times n$  string table with the name s (which must be a string, and a valid identifier), and opens the table editor so its entries can be entered.

## createVectorField

createVectorField(expr, vars, set) creates a vector field from the equation expr. expr is a string representing a real vector-valued expression in two variables, given by vars, a comma-separated string of the valid identifiers. expr(x, y) is supposed to give the vector at the point (x, y) in the plane. The resulting set is a vector field, a set of vectors (x, y, vx, vy) associating a vector (vx(x, y), vy(x, y)) to each point (x, y) in the plane. set is the set of points (x, y) for which the expression expr is evaluated.

Examples: Vertical constant vector field (e.g. gravity):
gravity := createVectorField("(0, -1)", "x, y", [-10, 10]^2)
drawVectorField("gravity", "colour:#333333")

Horizontal linear vector field (e.g. a spring force)
force := createVectorField("( -x, 0)", "x, y", [-10, 10]^2)
drawVectorField("force", "colour:#333333")

An oscillating chemical reaction

```
a := 2
b := 3.001
```

```
vectorField := createVectorField("(1 + a·x^2·y - b·x - x, -a·x^2·y + b·x)", "x, y", [0, 10, 0.25]^2)
```



```
drawVectorField("vectorField", "colour:#333333")
```

```
flow := computeFlowTrajectory("(1 + a·r_1^2·r_2 -
b·r_1 - r_1, -a·r_1^2·r_2 + b·r_1)", "r", (1, 4), 0, 100, 0.01)
drawLines("flow", "colour:gold")
```

**csc**

csc(x) = 1 / sin(x). x is a real or complex number.

**csch**

csch(x) is the hyperbolic cosecant, i.e. csch(x) = 1 / sinh(x).

**cylindricalCoords**

cylindricalCoords(S) applies the transformation

```
x = r·sin(φ)
y = r·cos(φ)
z = z
```

to all three-dimensional cylindrical real vectors (r, φ, z) in the set S, and returns the new set of Cartesian coordinates (x, y, z).

This is useful for plotting cylindrical graphs. Simply create a set S of cylindrical coordinates (r, φ, z) and then transform it using

```
S := cylindricalCoords(S)
```

after which it can be plotted using drawSet3, drawLines3, etc.

Example: An illustrative way to draw a grid cylinder with radius 4.

```
net := createNet(0, 2·π, 0.01, π/12, -10, 10, 0.1, 1)
paramNet := createImage("(4, r_1, r_2)", "r", net)
cylinder := cylindricalCoords(paramNet)
drawSet3("cylinder")
```

**date**

date(0) returns a structure containing the current date. The members are year, month, day, weekOfYear, dayOfYear, and dayOfWeek.

**day**

day(0) returns the name of the current weekday as a string.

**daysBetween**

daysBetween(d1, d2) returns the number of days between the date structures d1 and d2.

Example:

```
d1 := encodeDate(2010, 06, 19)
```

```
year: 2010
month: 6
day: 19
weekOfYear: 24
dayOfYear: 170
dayOfWeek: 6
```

```
d2 := date(0)
```

```
year: 2010
month: 6
day: 23
weekOfYear: 25
dayOfYear: 174
dayOfWeek: 3
```

```
daysBetween(d1, d2)
```

4

**defineOperator**

defineOperator(kind, symb, func) defines a new operator with symbol symb, a single character (i.e. a one-character string) corresponding to the function func, a string with the name of a (defined) function.

kind is either

- \* "postfix",
- \* "prefix", or
- \* "infix".

In the first two cases, func must accept exactly one argument, and in the case of an infix operator, it must accept two arguments.

defineOperator(kind, symb1, symb2, func) defines a new operator with symbol symb1 ... symb2, two single characters (i.e. two one-character strings) corresponding to the function func, a string with the name of a (defined) function.

kind must be "circumfix", and func must accept exactly one argument.

The newly added operator will have a priority lower than all previously defined operators.

Examples: defineOperator("postfix", "?", "isPrime")

```
53? = true
```

```
defineOperator("circumfix", "$", "@", "totient")
```

```
$80@ = 32
```

**delete**

delete(s) removes the identifier (variable) named s. s is the name of the identifier, and this a string.

Example: delete("r") removes the identifier named "r".

**deleteFile**

deleteFile(f) deletes the file f (a string containing a valid file name).

**deleteFunction**

deleteFunction(fname) deletes the user function with identifier "fname".

Example:

```
f := "x, y, z" ↦ "2·x^2 + y^2 - 3·z^2"
```

```
f(6, 2, 3)
```

```
49
```

```
deleteFunction("f")
```

```
f(6, 2, 3)
```

Unknown identifier: Unknown identifier "f".

**describe**

describe(x) returns the string with the description associated with the identifier named x, which is the name of the identifier, i.e. a string.

Example: describe("π") = "The ratio between a circle's circumference and diameter."

**det**

det(M) returns the determinant of the real or complex matrix M.

**diag**

diag(a1, a2, ..., an) returns the n×n square matrix with the entry Mij equal to ai δ(i, j) where δ is the Kronecker delta function.

diag((a1, a2, ..., an)) returns the n×n square matrix with the entry Mij equal to ai δ(i, j) where δ is the Kronecker delta function.

**dictionaryGetWordList**

dictionaryGetWordList(0) returns the entire (currently loaded, see loadDictionary) dictionary as a list of words.

**dictionaryGetWordSet**

dictionaryGetWordSet(0) returns the entire (currently loaded, see loadDictionary) dictionary as a set of words (as strings).

Example: random(dictionaryGetWordSet(0))  
undulated

**dictionaryListAnagrams**

dictionaryListAnagrams(s) returns the list of all anagrams to the word (or phrase) s, using the currently loaded dictionary (see loadDictionary).

Example: dictionaryListAnagrams("algorithm")  
algorithm  
logarithm

**dictionaryListPalindromes**

dictionaryListPalindromes(0) returns the list of all non-trivial palindromes in the currently loaded (see loadDictionary) dictionary. A non-trivial palindrome is a word s of at least three characters, such that s = reverse(s).

**dictionaryListWordsWithAnagrams**

dictionaryListWordsWithAnagrams(0) returns the list of all words in the currently loaded dictionary (see loadDictionary) that have non-trivial anagrams. A non-trivial anagram to a word s is a word, not equal to s, that has the same number of all letters as s, i.e. if it is a permutation of s. It might take a few hours to compile the list.

**dictionaryLookup**

dictionaryLookup(s) searches the currently loaded dictionary (see loadDictionary) for the entry s, a string, and returns the definition(s) of the word.

Example: dictionaryLookup("isomorphism")  
Noun: (algebra) A bijection  $f$  such that both  $f$  and its inverse  $f^{-1}$  are homomorphisms, that is, structure-preserving mappings.

Noun: (biology) the similarity in form of organisms of different ancestry

Noun: (chemistry) the similarity in the crystal structures of similar chemical compounds

Noun: (computer science) a one-to-one correspondence between all the elements of two sets, e.g. the instances of two classes, or the records in two datasets

Noun: (sociology) the similarity in the structure or processes of different organizations

**dictionaryMatchWord**

dictionaryMatchWord(s) searches the currently loaded dictionary (see loadDictionary) for entries matching the filter s, a string containing characters and the "\_" placeholder. All entries matching this filter are returned, in a list (i.e., one-dimensional (vertical) string table). A word matches s iff it has the same number of characters as s, and the ith character in s is equal to the "\_" placeholder or to the ith character in the word, for all i.

Examples: dictionaryMatchWord("g\_ta\_")

geotag  
gluta-  
guitar

dictionaryMatch-

Word("\_\_\_\_\_")  
Federal Democratic Republic of Ethiopia  
acute necrotising ulcerative gingivitis  
acute necrotizing ulcerative gingivitis  
born with a silver spoon in one's mouth  
cut one's coat according to one's cloth  
defense-independent pitching statistics  
discretion is the better part of valour  
hepaticocholangiocholecystenterostomies  
if my aunt had balls, she'd be my uncle  
it's not what you know but who you know  
out of the frying pan and into the fire  
pairwise linkage disequilibrium diagram  
program evaluation and review technique  
project evaluation and review technique  
selective serotonin reuptake inhibitors  
serum glutamic oxaloacetic transaminase  
there's more than one way to skin a cat  
transmissible spongiform encephalopathy  
well ain't that the catfish in the trap

**diff**

diff(expr, var, x) returns the derivative of expr (a string representing an expression in var evaluating to a real number), as a function of var (a string with a valid identifier) at x, a real number in the domain of definition of expr.

diff(expr, var, x, h) uses the the explicit distance h in the independent variable var when computing the derivative. In some cases, a rather large (e.g. 0.01) value of h might be required, if expr is only computed with a limited resolution.

Examples: diff("sin(x)", "x", 0) = 1  
diff("FresnelC(t)", "t", π/2, 0.01) = -0.782...

**diffGraph**

diffGraph(S) returns the graph of the derivative f' of the function f with the graph S = { (x, f(x)) }.

Example: sine := createGraph("sin(x)", "x", [-10, 10, 0.001])  
cosine := diffGraph(sine)  
drawSet("cosine")

**dim**

dim(v) returns the dimension of the real or complex vector v.

Example: dim((1, 0, 0)) = 3

**directSum**

directSum(S1, S2) = S1 ⊕ S2 returns the direct sum of the sets S1 and S2, both containing real vectors of the same dimension.

Example:

s1 := {(1, 2), (3, 5)}

{ (1, 2), (3, 5) }

s2 := {(5, 7), (2, 4)}

{ (5, 7), (2, 4) }

s1 ⊕ s2

{ (6, 9), (3, 6), (8, 12), (5, 9) }

**dirExists**

dirExists(s) returns True if the directory s (a string), exists and False otherwise.

**divisors**

divisors(n) returns the vector of all positive divisors of the integer n.

**drawArrow**

drawArrow(v) draws the arrow of the vector pointing from the origin to  $v \in \mathbb{R}^2$  in the current 2D visualization window.

drawArrow(a, b) draws the arrow between the points  $a \in \mathbb{R}^2$  and  $b \in \mathbb{R}^2$ .

drawArrow(a, b, s) draws the arrow between the points  $a \in \mathbb{R}^2$  and  $b \in \mathbb{R}^2$  using the style (CSS) s.

Example: drawArrow((0, 0), (3, 5), "line-colour:red; triangle-colour:red")

**drawArrow3**

drawArrow3(v) draws the arrow of the vector pointing from the origin to  $v \in \mathbb{R}^3$  in the current 3D visualization window.

drawArrow3(a, b) draws the arrow between the points  $a \in \mathbb{R}^3$  and  $b \in \mathbb{R}^3$ .

drawArrow3(a, b, s) draws the arrow between the points  $a \in \mathbb{R}^3$  and  $b \in \mathbb{R}^3$  using the style (CSS) s.

Example: drawArrow3((0, 0, 0), (3, 5, 5), "line-colour:red; triangle-colour:red")

**drawAxes**

drawAxes(0) draws two-dimensional coordinate axes in the 2D visualization window.

**drawAxes3**

drawAxes3(0) draws three orthogonal axes in the current 3D visualization window.

**drawBox3**

drawBox3(v, Δ) draws a box in the current 3D visualization window, with an edge at  $v \in \mathbb{R}^3$  and dimensions  $\Delta = (w, h, d) \in \mathbb{R}^3$ .

drawBox3(v, Δ, s) draws a box in the current 3D visualization window, with an edge at  $v \in \mathbb{R}^3$  and dimensions  $\Delta = (w, h, d) \in \mathbb{R}^3$  using the style (CSS) s.

**drawCircle**

drawCircle(v, r) draws a circle with center  $v \in \mathbb{R}^2$  and radius r in the current 2D visualization window.

drawCircle(v, r, s) draws a circle with center  $v \in \mathbb{R}^2$  and radius r in the current 2D visualization window using the style (CSS) s.

All numbers refer to the visualization window's coordinate system.

Example: drawCircle((2, 2), 1, "colour:red; border-colour:white")

**drawColouredLines**

drawColouredLines(S) plots the point set S in the current 2D visualization window and connects the points. Each vector in S needs to be a three-dimensional vector (SIC!), where the third component is the colour code of the pixel.

Compare: drawColouredSet, and drawLines.

**drawColouredLines3**

drawColouredLines3(S) plots the point set S in the current 3D visualization window and connects the points. Each vector in S needs to be a four-dimensional vector (SIC!), where the fourth component is the colour code of the pixel.

Compare: drawColouredSet3, and drawLines3.

Examples: Draw a coloured circular helix

```
helix := createImage("(3*cos(t), 3*sin(t), t/2, hsv(10*t, 1, 1))", "t", [-30, 30, 0.01])
drawColouredLines3("helix")
```

**drawColouredPlane**

drawColouredPlane(S) draws the coloured plane S in the current 2D visualization window. The set S is created by the createColouredPlane function.

Example:

```
Superposition of two water waves.
ψ := "r" ↦ "sin(4*norm((2, 2) - r))/6"
Φ := "r" ↦ "sin(4*norm((0, 0) - r))/6"
S := "r" ↦ "ψ(r) + Φ(r)"
waves := createColouredPlane("hsv(90 - 270*S((x, y)), 1, 1)", -10, 10, 0.1, -10, 10, 0.1)
drawColouredPlane("waves")
```

**drawColouredSet**

drawColouredSet(S) plots the point set S in the current 2D visualization window. Each vector in S needs to be a three-dimensional vector (SIC!), where the third component is the colour code of the pixel.

Compare: drawColouredLines, and drawSet.

**drawColouredSet3**

drawColouredSet3(S) plots the point set S in the current 3D visualization window. Each vector in S needs to be a four-dimensional vector (SIC!), where the fourth component is the colour code of the pixel.

Compare: drawColouredLines3, and drawSet3.

**drawColouredSurfParamCurves**

drawColouredSurfParamCurves(S) plots the surface parameter curves S in the current 3D visualization window. S is created by the createSurfParamCurves function.

Examples: surf := createSurfParamCurves("(x, y, sin(sqrt(x^2 + y^2))), hsv(x^2 + y^2, 1, 1)", "x, y", -10, 10, -10, 10)

```
drawColouredSurfParamCurves("surf")

// Superposition of water waves (without attenuation...)
ψ := "r" ↦ "sin(4*norm((2, 2) - r))/6"
Φ := "r" ↦ "sin(4*norm((0, 0) - r))/6"
S := "r" ↦ "ψ(r) + Φ(r)"
set := createSurfParamCurves("(x, y, S((x, y))), hsv(90 - 270*S((x, y)), 1, 1)", "x, y", -10, 10, -10, 10)
drawAxes3(1)
drawColouredSurfParamCurves("set")
```

**drawCone**

drawCone(r, h) draws a cone of radius  $r > 0$  and height  $h > 0$  with the z-axis as its symmetry axis and occupying the region  $z \in [0, h]$  in the current 3D visualization window.

drawCone(r1, r2, h) draws a truncated cone of bottom radius  $r1 > 0$ , top radius  $r2 > 0$ , and height  $h > 0$  with the z-axis as its symmetry axis and occupying the region  $z \in [0, h]$  in the current 3D visualization window.

`drawCone(v, r1, r2, h)` draws a truncated cone of bottom radius  $r1 > 0$ , top radius  $r2 > 0$ , and height  $h > 0$ . The mid-point of the bottom plane is  $v \in \mathbb{R}^3$ , so that it occupies the vertical region  $z \in [v_3, v_3 + h]$ .

`drawCone(v, r1, r2, h, s)` draws a truncated cone of bottom radius  $r1 > 0$ , top radius  $r2 > 0$ , and height  $h > 0$ . The mid-point of the bottom plane is  $v \in \mathbb{R}^3$ , so that it occupies the vertical region  $z \in [v_3, v_3 + h]$ .  $s$  is the style (CSS) used to render the object.

**drawCylinder**

`drawCylinder(r, h)` draws a cylinder of radius  $r > 0$  and height  $h > 0$  in the current 3D visualization window. The cylinder will have the  $z$ -axis as its symmetry axis and will occupy the region  $z \in [0, h]$  around it.

`drawCylinder(v, r, h)` draws a cylinder of radius  $r > 0$  and height  $h > 0$  in the current 3D visualization window. The cylinder will be directed in the  $z$  axis and will have the mid-point of its bottom plane at  $v \in \mathbb{R}^3$ , so that it will occupy the vertical region  $z \in [v_3, v_3 + h]$ .

`drawCylinder(v, r, h, s)` draws a cylinder of radius  $r > 0$  and height  $h > 0$  in the current 3D visualization window. The cylinder will be directed in the  $z$  axis and will have the mid-point of its bottom plane at  $v \in \mathbb{R}^3$ , so that it will occupy the vertical region  $z \in [v_3, v_3 + h]$ . The cylinder is drawn using the style (CSS)  $s$ .

Example: `drawCylinder((0, 0, 0), 3.00, 5, "colour:gray")`  
`drawCylinder((0, 0, 0), 3.01, 5, "colour:red")`

**drawGrid3**

`drawGrid3(0)` draws a square grid in the  $z = 0$  plane, that is, the plane spanned by the  $x$  and  $y$  unit vectors.

`drawGrid3(0, s)` draws a square grid in the  $z = 0$  plane, that is, the plane spanned by the  $x$  and  $y$  unit vectors, using the style (CSS)  $s$ .

Example: `drawGrid3(0, "colour:green")`

**drawGrids**

`drawGrids(s)` draws grids in the planes specified by the comma-separated list (string)  $s$ .

The following planes are supported.

Identifier	Plane
x	$x = -10$
X	$x = +10$
y	$y = -10$
Y	$y = +10$
z	$z = -10$
Z	$z = +10$

`drawGrids(s, fmt)` draws grids in the planes specified by the comma-separated list (string)  $s$ , using the style (CSS)  $fmt$ .

**drawLine**

`drawLine(v1, v2)` draws the straight line segment between  $v1 \in \mathbb{R}^2$  and  $v2 \in \mathbb{R}^2$ .

`drawLine(v1, v2, s)` draws the straight line segment between  $v1 \in \mathbb{R}^2$  and  $v2 \in \mathbb{R}^2$  using the style (CSS)  $s$ .

**drawLine3**

`drawLine3(v1, v2)` draws the line segment from  $v1 \in \mathbb{R}^3$  to  $v2 \in \mathbb{R}^3$  in the current 3D visualization window.

**drawLines**

`drawLines(S)` plots the set  $S \subset \mathbb{R}^2$  in the 2D visualization window, and connects the points using lines.

`drawLines(S, s)` plots the set  $S \subset \mathbb{R}^2$  in the 2D visualization window, and connects the points using lines, and using the style (CSS)  $s$ .

Example: `spiral := createImage("(FresnelC(t), FresnelS(t))", "t", [-10, 10, 0.01])`  
`drawLines("spiral")`

**drawLines3**

`drawLines3(S)` plots the set  $S \subset \mathbb{R}^3$  in the 3D visualization window, and connects the points using lines.

`drawLines(S, s)` plots the set  $S \subset \mathbb{R}^3$  in the 3D visualization window, and connects the points using lines, and using the style (CSS)  $s$ .

Examples: A circular helix with radius 3 and pitch  $\pi$   
`helix := createImage("(3*cos(t), 3*sin(t), t/2)", "t", [-30, 30, 0.01])`  
`drawLines3("helix")`

**drawPixmap**

`drawPixmap(s, x, y)` draws the pixmap named  $s$  (a string representing a valid identifier of a pixmap variable) at the point  $(x, y)$  in the current 2D visualization window.  $(x, y)$  is given in the visualization window's logical coordinates.

`drawPixmap(s, x, y, fmt)` draws the pixmap using the style (CSS)  $fmt$ .

Possible parameters for  $fmt$ :

box-origin: top-left, top-center, top-right, middle-left, middle-center, middle-right, bottom-left, bottom-center, and bottom-right specifies what point on the pixmap is to be located above  $(x, y)$ .

Example: `drawPixmap("ball", 0, 0, "box-origin: middle-center")` if "ball" is a previously defined pixmap.

**drawPolygon**

`drawPolygon(S)` draws the polygon with vertices at the points of  $S \subset \mathbb{R}^2$  in the current 2D visualization window.

`drawPolygon(S, s)` draws the polygon with vertices at the points of  $S \subset \mathbb{R}^2$  using the style (CSS)  $s$ .

Example: `vertices := {(1, 1), (2, 2), (3, 1)}`  
`drawPolygon("vertices")`

Compare to: `drawLines`, e.g. `drawLines("vertices")`, or `drawSet`, e.g. `drawSet("vertices")`.

**drawRect**

`drawRect(v, (w, h))` draws a rectangle at  $v \in \mathbb{R}^2$  with width  $w$  and height  $h$  in the current 2D visualization window.

`drawRect(v, (w, h), s)` draws a rectangle at  $v \in \mathbb{R}^2$  with width  $w$  and height  $h$  in the current 2D visualization window using the style (CSS)  $s$ .

All numbers refer to the visualization window's coordinate system.

**drawSet**

`drawSet(S)` plots the set  $S \subset \mathbb{R}^2$  in the 2D visualization window.

`drawSet(S, s)` plots the set  $S \subset \mathbb{R}^2$  in the 2D visualization window using the style (CSS)  $s$ .

Examples: `set := createGraph("sin(x)", "x", [-10, 10, 0.001])`  
`derivative := createGraph("cos(x)", "x", [-10, 10, 0.001])`  
`drawSet("set")`

```
drawSet("derivative", "colour:gold")
```

### drawSet3

drawSet3(S) plots the set  $S \subset \mathbb{R}^3$  in the 3D visualization window.

drawSet3(S, s) plots the set  $S \subset \mathbb{R}^3$  in the 3D visualization window using the style (CSS) s.

Examples: A straight line.

```
r0 := (2, 3, 1)
v := (1, 1, -2)
line := createImage("r0 + t*v", "t", [-10, 10, 0.001])
drawSet3("line", "colour:red")
```

Remark: For most curves, drawLines3 is much more efficient, because a seemingly continuous curve is generated even if there is a visible distance between the points in the set. In fact, in this example of a line, only two points are required to draw the line segment, compared to the 20 000 points used by drawSet3. However, if the curve makes sudden jumps, drawLines3 will draw unwanted lines, and drawSet3 might be required. This is the case, for instance, when plotting hyperbolas using the sec/tan parametrisation.

### drawSphere

drawSphere(r) draws a sphere of radius  $r > 0$  and midpoint  $(0, 0, 0)$  in the current 3D visualisation window.

drawSphere(v, r) draws a sphere of radius  $r > 0$  and midpoint  $v \in \mathbb{R}^3$  in the current 3D visualisation window.

drawSphere(v, r, s) draws a sphere of radius  $r > 0$  and midpoint  $v \in \mathbb{R}^3$  in the current 3D visualisation window using the style (CSS) s, a string.

Example: drawSphere((0, 0, 0), 5, "colour:red")

### drawSurfParamCurves

drawSurfParamCurves(S) draws the surface parameter curves S, a set created by the createSurfParamCurves function, in the current 3D visualization window.

Examples: garden := createSurfParamCurves("(x, y, sin(x-randomReal(1)) \* sin(y))", "x, y", -10, 10, -10, 10)  
drawSurfParamCurves("garden", "colour:gold")

```
grass := createSurfParamCurves("(x, y, randomReal(1))", "x, y", -10, 10, -10, 10)
drawSurfParamCurves("grass", "colour:forestgreen")
```

See also: createSurfParamCurves

### drawText

drawText(s, x, y) draws the text s (a string) in the 2D visualization window, at (screen) coordinates (x, y), by default the top-left corner of the text rectangle.

drawText(s, x, y, fmt) draws the text s (a string) in the 2D visualization window, at (screen) coordinates (x, y), by default the top-left corner of the text rectangle, using the style (CSS) fmt.

Example: drawText("Graphs", 10, 10, "colour:red; text-size:20")

### drawVectorField

drawVectorField(s) draws the vector field s, a string representing the valid identifier of a vector field set, i.e. a set of vectors  $(x, y, vx, vy)$  associating a vector  $(vx(x, y), vy(x, y))$  to each point  $(x, y)$  of the plane. Such sets are easily generated from any vector-valued expression using the function createVectorField.

drawVectorField(s, fmt) uses the format (CSS) fmt.

Examples: Vertical constant vector field (e.g. gravity):  
gravity := createVectorField("(0, -1)", "x, y", [-10, 10]^2)  
drawVectorField("gravity", "colour:#333333")

Horizontal linear vector field (e.g. a spring force)  
force := createVectorField("(-x, 0)", "x, y", [-10, 10]^2)  
drawVectorField("force", "colour:#333333")

An oscillating chemical reaction  
a := 2  
b := 3.001

```
vectorField := createVectorField("(1 + a*x^2*y - b*x-x, -a*x^2*y + b*x)", "x, y", [0, 10, 0.25]^2)
drawVectorField("vectorField", "colour:#333333")
```

```
flow := computeFlowTrajectory("(1 + a-r_1^2-r_2 - b-r_1 - r_1, -a-r_1^2-r_2 + b-r_1)", "r", (1, 4), 0, 100, 0.01)
drawLines("flow", "colour:gold")
```

### editMatrix

editMatrix(s) opens the matrix editor and edits the real or complex matrix called s. Thus, s is the name of the matrix, i.e. a string.

### editOperatorTable

editOperatorTable(0) opens the table editor with the current operator table loaded, and any changes made to the table will be used when parsing new expressions.

### editTable

editTable(s) opens the table editor and edits the string table called s. Thus, s is the name of the table, i.e. a string.

### eigenvalues

eigenvalues(A) returns the vector of eigenvalues of the real square matrix A.

In the current implementation, eigenvalues works only for real non-singular matrices A.

### encodeDate

encodeDate(year, month, day) returns the date structure corresponding to the specified year, month, and day.

Example:

```
encodeDate(2010, 06, 19)
```

```
year: 2010
month: 6
day: 19
weekOfYear: 24
dayOfYear: 170
dayOfWeek: 6
```

### encodeDateTime

encodeDateTime(year, month, day, hour, minute, second) returns the date and time structure corresponding to the given year, month, day, hour, minute, and second.

Example:

```
encodeDateTime(2010, 06, 19, 16, 00, 00)
```

```
year: 2010
month: 6
day: 19
weekOfYear: 24
dayOfYear: 170
dayOfWeek: 6
```

hour: 16  
 minute: 0  
 second: 0  
 millisecond: 0

## encodeTime

encodeTime(hour, minute, second) returns the time structure corresponding to the specified time.

## endDrawing

endDrawing(0) decreases the common halt index of the 2D and 3D visualization windows by one. Initially, this index is equal to zero, and if it is positive and an object (set, pixmap, geometrical entity, etc.) is added to the window, the window is not redrawn on-screen.

## erf

erf(x) is the error function at x.

$\text{erf}(x) = \int \exp(-t^2) dt$  where t goes from 0 to x.

## erfc

erfc(x) is the complementary error function evaluated at x, i.e.  $\text{erfc}(x) = 1 - \text{erf}(x)$ .

## error

error(s) displays s as an error message. s is a string.

## exit

exit(0) exits AlgoSim.

## exp

$e^x = \exp(x)$  is the exponential function.

For a complex number z,

$e^z = \exp(z) = e^{\text{Re } z} \cdot (\cos(\text{Im } z) + i \cdot \sin(\text{Im } z))$

where Re z and Im z are the real and imaginary parts of z, respectively.

## exportToMetafile

exportToMetafile(fn) creates a Windows Enhanced Metafile (EMF) file with filename fn, a fully qualified file name with extension ".emf", of the current 2D scene.

Example: exportToMetafile(fileSaveDialog(0))

## exportToSVG

exportToSVG(fn) creates a Scalable Vector Graphics (SVG) 1.1 file with filename fn, a fully qualified file name with extension ".svg", of the current 2D scene.

Example: exportToSVG(fileSaveDialog(0))

Please notice that any coloured planes and pixmaps will not be saved in the SVG file, due to their bitmap nature.

## extractFileDrive

extractFileDrive(s), where s is a string representing a file name, returns the drive of the file name.

For example, extractFileDrive("C:\WINDOWS\notepad.exe") returns "C:".

## extractFileExt

extractFileExt(s), where s is a string representing a file name, returns the file extension of the file name, including the period.

For example, extractFileExt("C:\WINDOWS\notepad.exe") returns ".exe".

## extractFileName

extractFileName(s), where s is a string representing a file name, returns the pure file name of s without the path.

For example, extractFileName("C:\WINDOWS\notepad.exe") returns "notepad.exe".

## extractFilePath

extractFilePath(s), where s is a string representing a file name, returns the path of the file name.

For example, extractFilePath("C:\WINDOWS\notepad.exe") returns "C:\WINDOWS\".

## factor

factor(x) returns the prime factorization (decomposition) of the integer  $x \geq 2$ .

## factors

factors(x) returns the unique vector of prime factors (sorted by increasing magnitude) of the integer  $x \geq 2$ .

## fibonacci

fibonacci(n) returns the nth Fibonacci number. fibonacci(0) = 0, fibonacci(1) = 1, fibonacci(2) = 1, fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2).

## fileExists

fileExists(s) returns True if the file s (a string) exists, and False otherwise.

## fileOpenDialog

fileOpenDialog(0) displays a file open dialog and returns the chosen file name.

## fileSaveDialog

fileSaveDialog(0) displays a file save dialog and returns the chosen file name.

## fillMatrix

fillMatrix(m, n, x) returns the  $m \times n$  matrix with all entries equal to x.

## filter

filter(S, var, expr) returns the subset of the set S whose elements satisfy the boolean expression expr, a string containing a boolean expression in one variable, var.

Examples: filter([1, 100], "n", "isPrime(n)") = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97 }

filter(v, var, expr) returns the vector v where all elements not satisfying expr are removed.

## find

find(s) lists all identifiers (variables and functions) containing the string s in their name or description.

## flattenStruct

flattenStruct(str) flattens the structure str, i.e. incorporates the members of the substructures into the root of str.

Example:

createStruct("date", date(0), "time", time(0))

date:year: 2010  
 date:month: 6  
 date:day: 23  
 date:weekOfYear: 25  
 date:dayOfYear: 174  
 date:dayOfWeek: 3  
 time:hour: 15

time:minute: 21  
 time:second: 45  
 time:millisecond: 560

flattenStruct(ans)

year: 2010  
 month: 6  
 day: 23  
 weekOfYear: 25  
 dayOfYear: 174  
 dayOfWeek: 3  
 hour: 15  
 minute: 21  
 second: 55  
 millisecond: 131

#### **floor**

$\lfloor x \rfloor = \text{floor}(x)$  returns the largest integer smaller than or equal to  $x$ , i.e. rounds  $x$  to the nearest integer in the direction of  $-\infty$ .

#### **fmt**

$\text{fmt}(S)$  returns the string  $S$  with variable placeholders replaced by the corresponding variable values. A placeholder has the form "&ident" (without the quotation marks) where "ident" is the valid identifier of an existing variable. "&&" will produce a single ampersand character.

Example:  $\text{fmt}(\text{"The radio is } \&\pi.\text{"})$  will return "The radio is 3.14159265359."

#### **frac**

$\text{frac}(x) = x - \text{trunc}(x)$  returns the fractional part of  $x$ .

#### **FresnelC**

$\text{FresnelC}(x)$  is the Fresnel cosine integral.

$\text{FresnelC}(x) = \int_0^x \cos(t^2) dt$  from 0 to  $x$ .

Example:  $\text{spiral} := \text{createImage}(\text{"(FresnelC}(t), \text{FresnelS}(t))", "t", [-10, 10, 0.01]})$   
 $\text{drawLines}(\text{"spiral"})$

#### **FresnelS**

$\text{FresnelS}(x)$  is the Fresnel sine integral.

$\text{FresnelS}(x) = \int_0^x \sin(t^2) dt$  from 0 to  $x$ .

Example:  $\text{spiral} := \text{createImage}(\text{"(FresnelC}(t), \text{FresnelS}(t))", "t", [-10, 10, 0.01]})$   
 $\text{drawLines}(\text{"spiral"})$

#### **gammaFunction**

$\text{gammaFunction}(x)$  is the gamma function evaluated at  $x$ .  $x$  must lie within  $]-1, 0[ \cup ]0, \infty[$ .

#### **gcd**

$\text{gcd}(m, n)$  returns the greatest common divisor of the integers  $m$  and  $n$ , that is the greatest number that divides both  $m$  and  $n$ .

Example:  $\text{gcd}(8, 20) = 4$ .

If  $\text{gcd}(m, n) = 1$ ,  $m$  and  $n$  are said to be coprime, or relatively prime.

#### **getClipboardText**

$\text{getClipboardText}(0)$  returns the current textual content of the Windows clipboard, if it exists.

#### **getCol**

$\text{getCol}(M, n)$  returns the  $n$ th column of the real or complex matrix  $M$  as a vector.

#### **getCPUBrandString**

$\text{getCPUBrandString}(0)$  returns the CPU's brand string, e.g. "Intel(R) Pentium(R) D CPU 2.80GHz".

#### **getCPUFeatureSet**

$\text{getCPUFeatureSet}(0)$  returns the set of (the strings identifying) the CPU's features, e.g. { "FPU", "MMX", "SSE", "SSE2", "SSE3", "HT", "x64" }.

#### **getCPUFeatureString**

$\text{getCPUFeatureString}(0)$  returns a comma-separated string of the CPU's features, e.g. "FPU, MMX, SSE, SSE2, SSE3, HT, x64".

#### **getCPUInfo**

$\text{getCPUInfo}(0)$  returns a structure containing information about the CPU running AlgoSim. The fields are vendor, model, brandString, and features. Model is by itself a structure containing the fields family, model, and stepping, as integers. All other fields are strings.

#### **getCPUModel**

$\text{getCPUModel}(0)$  returns the three-dimensional vector containing the CPU's family, model, and stepping, respectively.

#### **getCPUVendor**

$\text{getCPUVendor}(0)$  returns the CPU vendor, e.g. "GenuineIntel".

#### **getDateCorrespondingToDayOfYear**

$\text{getDateCorrespondingToDayOfYear}(Y, n)$  returns the date and time structure corresponding to the zeroth millisecond of the  $n$ th day of the year  $Y$ .

Example:

$\text{getDateCorrespondingToDayOfYear}(2010, 100)$

year: 2010  
 month: 4  
 day: 10  
 weekOfYear: 14  
 dayOfYear: 100  
 dayOfWeek: 6  
 hour: 0  
 minute: 0  
 second: 0  
 millisecond: 0

#### **getDateCorrespondingToWeekOfYear**

$\text{getDateCorrespondingToWeekOfYear}(Y, n)$  returns the date and time structure corresponding to the zeroth millisecond of the  $n$ th week of the year  $Y$ .

Example:

$\text{getDateCorrespondingToWeekOfYear}(1987, 50)$

year: 1987  
 month: 12  
 day: 7  
 weekOfYear: 50  
 dayOfYear: 341  
 dayOfWeek: 1  
 hour: 0  
 minute: 0  
 second: 0  
 millisecond: 0

#### **getDigit**

$\text{getDigit}(x, n)$  returns the  $n$ th digit of the integer (part of)  $x$ .

#### **getDir**

$\text{getDir}(0)$  returns the current working directory.

### **getGenerator**

Internal function: `getGenerator(S)` returns the generator string of the set `S`.

### **getHsl**

`getHsl(c)` returns a vector with the H, S, and L coordinates of the colour code `c`.

### **getHsv**

`getHsv(c)` returns a vector with the H, S, and V coordinates of the colour code `c`.

### **getNextSpecificDayOfWeek**

`getNextSpecificDayOfWeek(D, d)` returns the time and date structure corresponding to the firsts day `d` (1 = monday, ..., 7 = sunday) after the date and time given by the date and time structure `D`.

`getNextSpecificDayOfWeek(D, d, n)` returns the time and date structure corresponding to the firsts day `d` (1 = monday, ..., 7 = sunday) after the date and time given by the date and time structure `D` plus `n` entire weeks.

Example:

```
getNextSpecificDayOfWeek(encodeDate(2010, 06, 23), 7)
```

```
year: 2010
month: 6
day: 27
weekOfYear: 25
dayOfYear: 178
dayOfWeek: 7
hour: 0
minute: 0
second: 0
millisecond: 0
```

### **getOperatorTable**

`getOperatorTable(0)` returns the currently loaded operator table.

### **getParameter**

`getParameter(s)` return the system parameter named `s`, a string. The returned value depends on `s`, but may be a string, a real number, or a boolean.

`s` may be any of the following strings:

`xmin` - the logical left limit of the 2D visualisation window (real number).  
`xmax` - the logical right limit of the 2D visualisation window (real number).  
`ymin` - the logical lower limit of the 2D visualisation window (real number).  
`ymax` - the logical upper limit of the 2D visualisation window (real number).  
`fullscreen` - the AlgoSim main window is in full-screen mode (boolean).  
`complex mode` - complex mode is on (boolean).  
`true sets` - "true sets" mode is on (boolean).  
`anti-aliasing` - anti-aliasing is active in the 3D visualisation window (boolean).  
`3d lighting` - realistic 3d lighting is active in the 3D visualisation window (boolean).  
`screen width` - the width of the desktop in pixels (real number).  
`screen height` - the height of the desktop in pixels (real number).  
`path` - the path of `AlgoSim.exe` (string).  
`RTE installed` - Rejbrand Text Editor is installed (boolean).  
`RTE path` - the path of Rejbrand Text Editor (string).  
`winver` - the version of Windows (string).

`Windows major version` - the major version number of Windows (real number).  
`Windows minor version` - the minor version number of Windows (real number).  
`Windows build number` - the build number of Windows (real number).  
`computer name` - the computer's name (string).  
`user name` - the active Windows user's name (string).  
`CPU vendor` - the CPU vendor (string).  
`CPU brand string` - the CPU brand string (string).  
`CPU feature string` - the CPU feature string (string).  
`ver` - AlgoSim version (string).  
`RTE ver` - the version of Rejbrand Text Editor (string).  
`AlgoSim common directory` - the path of the AlgoSim Program Files directory (string).  
`AlgoSim user directory` - the local user's AlgoSim directory (string).

In addition, `getParameter("system metrics", n)` is a wrapper for the `GetSystemMetrics` Windows API function.

`getParameter("handle", str)` returns window handles (HWNDs) depending on the string `str`. Possible values of `str` are

"application" - the handle of the main AlgoSim window.  
"window" - the handle of the main AlgoSim window.  
"console" - the handle of the console.  
"v2d" - the handle of the 2D visualisation control.  
"v3d" - the handle of the 3D visualisation control.  
"pixmaps" - the handle of the pixmap viewer.  
"identifiers" - the handle of the Identifiers list box.  
"broadcast" - `HWND_BROADCAST`  
"desktop" - `HWND_DESKTOP`

### **getProgramLocations**

`getProgramLocations(0)` returns a list of the directories in which AlgoSim programs (\*.prg files) are located. Typically there is one (high-security) directory common to all users of the computer, and one specific to the current user.

### **getRgb**

`getRgb(c)` returns a vector with the R, G, and B coordinates of the colour code `c`.

### **getRow**

`getRow(M, n)` returns the `n`th row of the real or complex matrix `M` as a vector.

### **getStructMemberFromIndex**

`getStructMemberFromIndex(str, n)` returns the value of the `n`th member of the structure `str`.

Example:

```
createStruct("firstName", "Andreas", "lastName", "Rejbrand",
"yearOfBirth", 1987, "IQ", ∞)
```

```
firstName: Andreas
lastName: Rejbrand
yearOfBirth: 1987
IQ: ∞
```

```
getStructMemberFromIndex(ans, 2) = "Rejbrand"
```

### **getStructNameFromIndex**

`getStructNameFromIndex(str, n)` returns the name (that is, the identifier, not the associated value) of the `n`th (1, 2, ...) member of the structure `str`, as a string. To obtain the value, use `getStructMemberFromIndex(str, n)` instead.

### **getStructNumMembers**

`getStructNumMembers(str)` returns the number of members of the structure `str`.



Example:

```
createStruct("firstName", "Andreas", "lastName", "Rejbrand",
"yearOfBirth", 1987, "IQ", ∞)
```

```
firstName: Andreas
lastName: Rejbrand
yearOfBirth: 1987
IQ: ∞
```

```
getStructNumMembers(ans) = 4
```

#### **getTickCount**

getTickCount(0) returns the (approximate) number of milliseconds since system power-on (modulo  $2^{32}$ ). Good for performance testing of time-synchronization inside loops.

#### **getViewAsBitmap**

getViewAsBitmap(0) returns the current 2D visualization window as a pixmap.

getViewAsBitmap(w, h) returns the current 2D visualization window as a  $w \times h$  pixmap.

#### **getViewAsBitmap3**

getViewAsBitmap3(0) returns the current 3D visualization window as a pixmap.

getViewAsBitmap3(w, h) returns the current 3D visualization window as a  $w \times h$  pixmap.

#### **GramSchmidt**

Let  $A$  be a  $m \times n$  real matrix, and consider the  $n$  columns as vectors in  $\mathbb{R}^m$ . Then GramSchmidt( $A$ ) returns the  $m \times n$  matrix whose columns are the vectors that are produced by the Gram-Schmidt orthonormalisation algorithm. In particular, if  $A$  is a square  $n \times n$  matrix of full rank, then GramSchmidt( $A$ ) will be orthogonal, i.e. the columns of  $A$  will constitute an orthonormal basis of  $\mathbb{R}^n$ .

#### **harmonicNumber**

harmonicNumber( $n$ ) returns the  $n$ th harmonic number.  $n$  is a positive integer.

#### **heaviside**

heaviside( $x$ ) = 1 if  $x > 0$ , and heaviside( $x$ ) = 0 otherwise.

heaviside is the Heaviside unit step function.

#### **hermitePhys**

hermitePhys( $n, x$ ) returns the physicist's Hermite polynomial of degree  $n$  evaluated at the point  $x$ , a real number.

#### **hermiteProb**

hermiteProb( $n, x$ ) returns the probabilist's Hermite polynomial of degree  $n$  evaluated at the point  $x$ , a real number.

#### **hoursBetween**

hoursBetween( $t1, t2$ ) returns the number of hours between the date and time structures  $t1$  and  $t2$ .

#### **hsl**

hsl( $h, s, l$ ) returns the colour code for the colour with the HSL coordinates  $h, s$ , and  $l$ .

#### **hsv**

hsv( $h, s, v$ ) returns the colour code for the colour with the HSV coordinates  $h, s$ , and  $v$ .

#### **hsvToRgb**

hsvToRgb( $h, s, v$ ) returns the three-dimensional RGB coordinates of the colour given by the HSV coordinates  $h, s$ , and  $v$ , as a real vector.

#### **identExists**

identExists( $str$ ) returns true if there exists a variable named  $str$  (a string), and false otherwise.

#### **identifyProblems**

identifyProblems(0) will perform an automatic search for potential problems in the current AlgoSim session. Among other things, identifyProblems will verify that

- \* no fundamental constant ( $\pi, e, i, \text{true}, \text{false}, \dots$ ) has been removed or redefined.
- \* the operator table has not been altered so much that simple expressions like  $4 + (3 \cdot 5!)/2$  cannot be evaluated.
- \* there is not a variable  $foo$  that has the same name as a function or program  $foo()$ .
- \* the operator table `ops.asd` and the constants table `constants.asd` are not missing.
- \* the system font `DejaVu Sans Mono` is installed.

If a problem can be resolved automatically (with no risk of loss of data), identifyProblems will automatically try to fix the problem. Otherwise, the function will return instructions how to manually resolve the issue.

#### **identity**

identity( $a1, a2, \dots, an$ ) returns an (but all arguments are evaluated, from left to right).

Thus

$$a1; a2 = \text{identity}(a1, a2) = a2$$

The semicolon operator is often used in for loops.

#### **identityMatrix**

identityMatrix( $n$ ) returns the  $n$ -dimensional identity matrix, i.e. the  $n \times n$  square matrix with all entries equal to  $\delta(m, n)$  where  $\delta$  is the Kronecker delta function.

#### **ifThen**

ifThen( $b, X1, X2$ ) returns  $X1$  if  $b = \text{true}$ , and  $X2$  otherwise.

Example: `ifThen(isOdd(n), "n is odd", "n is even")`

#### **Im**

Im( $z$ ) returns the imaginary part of a complex number  $z$ . If  $x$  is a real number,  $\text{Im}(x) = 0$ .

#### **include**

include( $s, x$ ) includes the element  $x$  into the set named  $s$ , a string.  $x$  might be any element that can be included in a set, i.e. a real or complex number, vector, matrix, string, or pixmap.

#### **indexOf**

indexOf( $v, x$ ) returns the index of the first occurrence of the number  $x$  in the vector  $v$ . If  $x$  does not occur in  $v$ , `indexOf(v, x) = -1`.

indexOf( $M, x$ ) returns the index of the first occurrence of the number  $x$  in the matrix  $M$ . If  $x$  does not occur in  $M$ , `indexOf(M, x) = (-1, -1)`.

indexOf( $tbl, str$ ) returns the index of the first occurrence of the string  $str$  in the table  $tbl$ . If  $str$  does not occur in  $tbl$ , `indexOf(tbl, str) = (-1, -1)`.

indexOf( $pm, clr$ ) returns the index of the first occurrence of the pixel colour  $clr$  in the pixmap  $pm$ . If  $clr$  does not occur in  $pm$ , `indexOf(pm, clr) = (-1, -1)`.

#### **info**

info( $s$ ) displays  $s$  as an informational message.  $s$  is a string.

**inputDialog**

inputDialog(0) displays a multi-line text input window, and returns the input string (using CRLF as line breaks).

**inputParams**

inputParams(s1, ..., sN) displays a parameter input dialog box. s1, ..., sN are valid identifiers of existing variables, the values of which are to be adjusted.

Example (from waveSim.prg):

```
λ1 := 3
λ2 := 4
v1 := 0.6
v2 := 0.4
A1 := 2
A2 := 2.3
inputParams("λ1", "λ2", "v1", "v2", "A1", "A2")
```

**intGraph**

intGraph(S) returns the graph of the primitive function F of f with graph  $S = \{ (x, f(x)) \}$  such that  $F(x_{min}) = 0$ , where  $x_{min}$  is the lowest (or, rather, first) x-value in S.

```
Example: gauss := createGraph("exp(-x^2)", "x", [-10, 10, 0.01])
erf := intGraph(gauss)
drawSet("erf")
```

**inv**

inv(x) returns the multiplicative inverse of x, a real number, a complex number, or a real or complex matrix, i.e.  $inv(x) = x^{-1}$  when applicable.

Example:  $inv(5) = 0.2$

**invertCase**

invertCase(str) returns the string str with all characters case-inverted.

Example:

```
invertCase("this is a brief text. a very brief text, actually.") =
"THIS IS A BRIEF TEXT. A VERY BRIEF TEXT, ACTUALLY."
```

**isEven**

isEven(x) returns True if x is even, and False otherwise. x is any integer.

```
Examples: isEven(5) = False
isEven(10) = True
```

isEven(expr, var, domain) returns True if the expression expr in the single variable var restricted to the domain domain is an even function, and False otherwise.

```
Examples: isEven("sin(x)", "x", [0, 10, 0.1]) = False
isEven("x·sin(x)", "x", [0, 10, 0.1]) = True
```

**isLeapYear**

isLeapYear(year) returns true if year is a leap year, and false otherwise.

Per definition,

```
isLeapYear(year) = (year mod 4 = 0) ∧ ((year mod 100 ≠ 0) ∨ (year mod 400 = 0))
```

Example:

```
isLeapYear(2010) = false
```

**isMagicSquare**

isMagicSquare(M) returns true iff the real matrix M is a magic square, i.e. a square matrix such that the sum of all elements is equal in every row, column, and diagonal. The magic square need not be normal.

See also: isNormalMagicSquare.

**isNormalMagicSquare**

isNormalMagicSquare(M), where M is a real matrix, returns true if and only if M is a normal magic square, i.e. a  $n \times n$  magic square containing every integer between 1 and  $n^2$  (exactly once).

Thus

```
isNormalMagicSquare(M) = isMagicSquare(M) ∧ isNormalSquare(M)
```

**isNormalPanmagicSquare**

isNormalPanmagicSquare(M), where M is a real matrix, returns true if and only if M is a normal panmagic (or, diabolic) square, i.e. a  $N \times N$  panmagic square in which every integer from 1 to  $N^2$  occurs (exactly once).

Thus,

```
isNormalPanmagicSquare(M) = isPanmagicSquare(M) ∧ isNormalSquare(M).
```

**isNormalSemimagicSquare**

isNormalSemimagicSquare(M), where M is a real matrix, returns true if and only if M is a normal semi-magic square, i.e. a  $M \times N$  semi-magic square in which every integer between 1 and  $N^2$  occurs (exactly once).

Thus,

```
isNormalSemimagicSquare(M) = isSemimagicSquare(M) ∧ isNormalSquare(M).
```

**isNormalSquare**

isNormalSquare(M), where M is a real matrix, returns true if and only if M has format  $n \times n$  and contains every integer between 1 and  $n^2$  (exactly once).

**isOdd**

isOdd(x) returns True if x is odd, and False otherwise. x is any integer.

```
Examples: isOdd(5) = True
isOdd(10) = False
```

isOdd(expr, var, domain) returns True if the expression expr in the single variable var restricted to the domain domain is an odd function, and False otherwise.

```
Examples: isOdd("sin(x)", "x", [0, 10, 0.1]) = True
isOdd("x·sin(x)", "x", [0, 10, 0.1]) = False
```

**isPanmagicSquare**

isPanmagicSquare(M), where M is a real matrix, returns true if and only if M is a panmagic (or, diabolic) square. A panmagic square is a  $N \times N$  magic square such that also every broken diagonal sums up to the magic constant of the square. The panmagic square need not be a normal magic square.

See also: isNormalPanmagicSquare

**isPrime**

isPrime(n) returns True if n is a prime number, and False otherwise.

**isSemimagicSquare**

isSemimagicSquare(M), where M is a real matrix, returns true if and only if M is a semi-magic square, i.e. a  $N \times N$  matrix such that the sum of all elements in every row and column equals the same number, the so-called "magic number" of the semi-magic square. The diagonals need not sum up to the magic number (if they do, the square is also a magic square).

See also isNormalSemimagicSquare, isMagicSquare, and isNormalMagicSquare.

**killSound**

killSound(0) stops all currently playing waveform sounds associated with the AlgoSim process.

**kronecker**

kronecker(x, y) is the Kronecker  $\delta$  function, i.e.  $\text{kronecker}(x, y) = 1$  if  $x = y$ , and  $\text{kronecker}(x, y) = 0$  if  $x \neq y$ .

**lcm**

lcm(m, n) returns the least common multiple of the integers m and n, that is the smallest number divisible by both m and n.

Example:  $\text{lcm}(12, 64) = 192$ .

**length**

length(s) returns the length (that is, the number of characters) in the string s.

**leviCivita**

leviCivita(a1, a2, ..., aN) returns 1 if (a1, a2, ..., aN) is an even permutation of the N first positive integers, -1 if (a1, a2, ..., aN) is an odd permutation of the N first positive integers, and 0 otherwise. Every ak must be an integer between 1 and N.

**listDir**

listDir(0) lists the contents of the current working directory.

**listFonts**

listFonts(0) returns a list (a one-column table) of all installed fonts.

Examples:  $\text{tableRows}(\text{listFonts}(0))$  returns (more or less) the number of installed fonts.

**listFunctions**

listFunctions(1) lists all declared functions, built-in or defined at runtime (user-defined).

**listNamedColours**

listNamedColours(0) returns a list of all named colours in AlgoSim. These colour identifiers may be used in CSS styles.

**listVars**

listVars(1) lists all variables.

**ln**

$\ln(x)$  is the natural logarithm, i.e. the inverse of the exponential function  $\exp(x)$ . x is a real or complex number.

For a complex number z,

$$\ln(z) = \ln(|z|) + i \cdot \arg(z)$$

where the argument  $\arg(z) \in ]-\pi, \pi]$  and i is the imaginary unit, i.e.  $i^2 = -1$ .

**loadDictionary**

loadDictionary(s) loads the dictionary file with file name s.

Example:  $\text{loadDictionary}(\text{"data/english"})$

**loadMatrixFromFile**

loadMatrixFromFile(s) loads the file with file name s, which must be a matrix saved in AlgoSim, and returns the real or complex matrix.

**loadPixmapFromFile**

loadPixmapFromFile(s) loads a AlgoSim (\*.asd) or Windows (\*.bmp) pixmap from a file. s is a fully-qualified name of the file. The pixmap is returned.

**loadPointSetFromFile**

loadPointSetFromFile(fn) returns the point set saved in the .asd file with name fn, a string containing a fully-qualified file name.

**loadSoundFromFile**

loadSoundFromFile(s) returns the sound in the file with name s. s must be a fully-qualified file name of an uncompressed WAV file.

**loadStructFromFile**

loadStructFromFile(fn) returns the structure saved in the file fn, a valid file name.

**loadTableDataFromTextFile**

loadTableDataFromTextFile(s) loads a AlgoSim table from a file. s is the fully-qualified name of the file. The file is required to contain only the textual part of the table, and thus no information regarding the cell's formatting. The table is returned.

**loadTableFromFile**

loadTableFromFile(s) loads a AlgoSim table from a file. s is the fully-qualified name of the file. The table is returned.

**loadTextFromFile**

loadTextFromFile(S) returns the string containing the text in the file named S.

**loadVectorFromFile**

loadVectorFromFile(s) loads the file with file name s, which must be a vector saved in AlgoSim, and returns the real or complex vector.

**log**

$\log(x)$  is the 10-logarithm of x.

**magicSquare**

magicSquare(N) returns a normal magic square of order N, i.e. a  $N \times N$  matrix of the integers 1, 2, ...,  $N^2$  such that the sum of the elements of each row, column, and diagonal equals the same number, the so called magic number of the matrix. A matrix has exactly two diagonals, in the usual sense.

The magic constant is always  $M = n(n^2 + 1) / 2$ . There is no magic square of order 2.

AlgoSim uses different algorithms when computing magic squares of

- 1) odd order ( $N = 2k + 1$ ),
- 2) double even order ( $N = 4k$ ), and
- 3) singly even order ( $N = 4k + 2$ ).

In general, the magic square of order N is not unique; AlgoSim will only produce \*one\* magic square of that order.

**matCols**

matCols(M) returns the number of columns in the real or complex matrix M.

**matRows**

matRows(M) returns the number of rows in the real or complex matrix M.

**matToPointSet**

matToPointSet(mat), where mat is a  $n \times 2$  (or  $n \times 3$ ) real matrix, returns the set of  $n$  points in  $R^2$  (or  $R^3$ ), in which the coordinates of point  $i$  are the elements of the  $i$ th row of mat.

Example: Given a matrix

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 5 \\ 2 & 5 \\ 2 & 7 \\ 4 & 6 \end{pmatrix},$$

matToPointSet(A) = {(1, 2), (3, 5), (2, 5), (2, 7), (4, 6)}.

**matToSet**

matToSet(M) returns the set of all entries in the  $m \times n$  matrix M.

**matToTable**

matToTable(M) returns the table with entries from the matrix M, at corresponding places.

**max**

max(a1, a2, ..., an) returns the greatest number of a1, a2, ..., and an.

max(v) returns the maximum (greatest) component in the real vector v.

**maxNorm**

maxNorm(v) returns the  $\infty$ -norm of the real or complex vector v, i.e. the component of v with greatest modulus (length).

**mean**

mean(a1, a2, ..., an) returns the arithmetical mean of the real or complex numbers a1, a2, ..., and an.

sum(v) returns the arithmetical mean of all components in the real or complex vector v.

**messageBox**

messageBox(str) displays a message box with the string str.

**millisecondsBetween**

millisecondsBetween(t1, t2) returns the number of milliseconds between the date and time structures t1 and t2.

**min**

min(a1, a2, ..., an) returns the smallest number of a1, a2, ..., and an.

min(v) returns the minimum (smallest) component in the real vector v.

**minutesBetween**

minutesBetween(t1, t2) returns the number of minutes between the date and time structures t1 and t2.

**mod**

mod(n, m) returns the remainder when the integer n is divided by the integer m.

Example: mod(12, 5) = 2

For general arguments x and y, mod(x, y) adds an integer multiple of y to x so that the result lies within [0, y].

Example: mod(14.71,  $\pi/2$ ) = 0.572833058846

**modSq**

modSq(x) returns the modulus squared of x, if x is a real number, a complex number, a real vector or a complex vector.

**monthsBetween**

monthsBetween(d1, d2) returns the number of months between the date structures d1 and d2.

Example:

d1 := encodeDate(2010, 06, 19)

year: 2010  
month: 6  
day: 19  
weekOfYear: 24  
dayOfYear: 170  
dayOfWeek: 6

d2 := date(0)

year: 2010  
month: 6  
day: 23  
weekOfYear: 25  
dayOfYear: 174  
dayOfWeek: 3

monthsBetween(d1, d2)

0.131416837782

**moveFile**

moveFile(f1, f2) moves the file f1 (a string containing a valid file name) to f2 (a string containing a valid file name).

**multiMagicSquareOrder**

multiMagicSquareOrder(M) returns the M-order of the square real matrix M. A multi-magic square of M-order k is a  $N \times N$  matrix such that  $M^{\wedge l}$  is a magic square  $\forall l \in \{1, 2, \dots, k\}$ , where  $M^{\wedge l}$  is defined by  $(M^{\wedge l})_{ij} = (M_{ij})^l$ , i.e. every element in  $M^{\wedge l}$  is obtain by raising the corresponding element of M to the power of l.

Every magic square has M-order at least 1 as a multi-magic square. A non-magic square has M-order 0, but it might still be a semi-magic square.

**nextPrime**

nextPrime(n) returns the smallest prime number greater than or equal to n.

**norm**

norm(v) returns the 2-norm of the vector  $v = (a1, a2, \dots, an)$ , i.e. the real number

$$\text{norm}(v) = \sqrt{v \cdot v} = (|a1|^2 + |a2|^2 + \dots + |an|^2)^{(1/2)}.$$

**note**

note(n) plays the MIDI tone  $n \in [0, 127]$  (an integer) using the current musical instrument and full intensity (127).

note(n, v) plays the MIDI tone  $n \in [0, 127]$  using the current musical instrument and the intensity  $v \in [0, 127]$ .

Example: note(60)

**noteOff**

noteOff(n) disables the tone  $n \in [0, 127]$  (an integer).

noteOff(n, v) disables the tone  $n \in [0, 127]$  (an integer) with the intensity  $v \in [0, 127]$ .

Example: `noteOff(60, 127)` if the current instrument requires the notes to be disabled manually.

**notes**

`notes(S)` plays the set of MIDI notes S. Each element in S is an integer in [0, 127], i.e. a valid MIDI note. The notes are played simultaneously and using the current instrument and full intensity (127).

Example: `notes({50, 60, 70})`

**notesOff**

`notesOff(S)` disables the MIDI notes in S, if necessary. See `notes` for more information.

Example: `notesOff({50, 60, 70})` if the current instrument requires the notes to be disabled manually.

**now**

`now(0)` returns a structure with the current date and time. The structure contains the members `year`, `month`, `day`, `weekOfYear`, `dayOfYear`, `dayOfWeek`, `hour`, `minute`, `second`, and `millisecond`.

Example:

`now(0)`

```
year: 2010
month: 6
day: 23
weekOfYear: 25
dayOfYear: 174
dayOfWeek: 3
hour: 14
minute: 7
second: 30
millisecond: 534
```

**numberVector**

If v is a N-dimensional vector and a an integer, then `numberVector(v, a)` returns a N×2 matrix where the first element of the nth row (indexed from 0) is n + a and the second element is `v(n + 1)`.

`numberVector(v) = numberVector(v, 0)`.

Example: `primes := sieveOfEratosthenes(10)`  
`numberVector(primes)`

**OPADD**

`a + b = OPADD(a, b)` returns the sum of a and b. a and b can be real or complex numbers, vectors, or matrices, or strings.

**OPAND**

`^` is the logical and operator.

Truth table for `p ^ q`:

```
^ 1 0
1 1 0
0 0 0
```

**OPAPPROX**

Internal function used to check approximate equality. Do not use explicitly.

**OPASSIGN**

`x := y` assigned the value returned by the expression y to the AlgoSim variable x.

The left operand is raw, and thus will be treated as a string.

**OPASTERISK**

`z*` returns the complex conjugate of the complex number z.

`M*` returns the transpose of the real matrix M.

`M*` returns the conjugate transpose (/Hermitian/ conjugate/transpose, or adjoint) of the complex matrix M, i.e. the transpose of M with all entries replaced by their complex conjugate.

**OPBAR**

`m | n` returns True if m divides n, i.e. if there exists an integer x such that `n = xm`, and False otherwise.

`(u|v)` returns the scalar (dot) product between the n-dimensional real or complex vectors u and v. `(u|v) = u·v`.

**OPCROSSMUL**

`v1×v2` returns the cross product of the two-dimensional real or complex vectors v1 and v2, i.e. a vector perpendicular to both v1 and v2, with norm (length) equal to `|v1||v2|sin θ` where θ is the angle between v1 and v2, and such that v1, v2, and v1×v2 is a right-handed system.

Example: If a plane  $\Pi \in R^3$  is generated\* by a point r and two vectors v1 and v2, the plane's normal vector is (up to a scaling factor) equal to `v1×v2`.

\*  $\Pi = \{(x, y, z) \in R^3: (x, y, z) = r + s v1 + t v2; s, t \in R\}$

**OPDEGREES**

`x° = OPDEGREES(x)` returns `x·(π/180)`, i.e. the angle expressed in radians (when x is expressed in degrees).

Example: `sin(90°) = 1`

**OPDIV**

`a / b = OPDIV(a, b)` returns the quotient when a is divided by b. a and b are real or complex numbers. If b is a number, a might also be a vector or matrix; this is equivalent to multiplying (i.e., scaling) the vector or matrix by a factor of `1 / b`.

**OPEQUALS**

`a = b`, or `OPEQUALS(a, b)`, returns True if a and b are identical objects, and False otherwise.

a and b might be real or complex numbers, vectors, matrices, pixmaps, sounds, tables, strings, booleans, or sets.

**OPEXP**

`a↑b = OPEXP(a, b) = a·10^(b)`

For example, the rest mass of an electron is `9.109382151↑31` and the mass of the sun is `1.9891↑30`.

**OPFACT**

`n! = OPFACT(n)` returns the factorial of the natural number n, i.e. the number `n·(n-1)·...·1`.

**OPGREATEROREQUAL**

`a ≥ b`, or `OPGREATEROREQUAL(a, b)`, returns True if `a > b` or if `a = b`, and False otherwise.

**OPGREATERTHAN**

`a > b`, or `OPGREATERTHAN(a, b)`, returns False if a is a smaller real number than b, and True otherwise.

**OPIN**

`x ∈ S` returns True if the element x is a member of the set S, and False otherwise.

**OPINDEX**

`vi` returns the ith component of the real or complex vector i.

$M_{(i, j)}$  returns entry  $(i, j)$  in the real or complex matrix  $M$ .

$s_i$  returns the  $i$ th character in the string  $s$ .

**OPINTERSECT**

$A \cap B$  returns the intersection between the sets  $A$  and  $B$ .

**OPINTERVAL**

$[a, b]$  returns the set of all integers between  $a$  and  $b$ , inclusively.

$[a, b, \delta]$  returns the set of all real numbers  $a, a + \delta, a + 2\delta, \dots$  less than or equal to  $b$ .

**OPLESSOREQUAL**

$a \leq b$ , or  $OPLESSOREQUAL(a, b)$ , returns True if  $a < b$  or if  $a = b$ , and False otherwise.

**OPLESSTHAN**

$a < b$ , or  $OPLESSTHAN(a, b)$ , returns True if  $a$  is a smaller real number than  $b$ , and False otherwise.

**OPMAPSTO**

$args \mapsto expr$  returns the function with the parameter list  $args$  (a comma-separated list of identifiers as a string) and expression  $expr$  (a string representing a valid expression, possible including the identifiers of  $args$ ).

A function is represented by a number. Assign this number to a valid identifier, to give the function a symbol.

Examples:  $sq := "x" \mapsto "x^2"$   
 $geoMean := "a, b" \mapsto "sqrt(a \cdot b)"$

**OPMEMBER**

$OPMEMBER(struct, "mem") = struct:mem$  returns the value of the member "mem" in the structure "struct".

Example:

`date(0)`

year: 2010  
 month: 6  
 day: 23  
 weekOfYear: 25  
 dayOfYear: 174  
 dayOfWeek: 3

`date(0):day`

23

`date(0):dayOfWeek`

3

**OPMINUS**

$-$  is unary minus. Thus  $-x$  is equivalent to  $0 - x$ , where  $-$  is binary minus (the subtraction operator) and  $x$  is a real or complex number, vector or matrix.

**OPMUL**

$a \cdot b = OPMUL(a, b)$  returns the product of  $a$  and  $b$ .  $a$  and  $b$  can be real or complex numbers, vectors, or matrices. In the case of vectors, the scalar (dot) product is returned, i.e. a real or complex number. Both vectors must have the same dimension (length). Two matrices, with formats  $r \times m$  and  $m \times c$ , respectively, may be multiplied, according to the rules of matrix multiplication, and the result is a  $r \times c$  matrix. If  $a$  is a natural number and  $b$  a string, the product is the string added to itself  $b$  times.

**OPNAND**

OPNAND is the logical nand (not and) operator.

Truth table for OPNAND( $p, q$ ):

NAND	1	0
1	0	1
0	1	1

**OPNGISSA**

$b := a$  assigns the value  $b$  to  $a$ .

Example:

$5 := a$

is equivalent to

$a := 5$ .

**OPNOR**

OPNOR is the logical nor (not or, neither) operator.

Truth table for OPNOR( $p, q$ ):

NOR	1	0
1	0	0
0	0	1

**OPNOT**

$\neg$  is logical negation.

$\neg q$  returns True if  $q$  is False, and False if  $q$  is True.

**OPNOTEQUAL**

$a \neq b$ , or  $OPNOTEQUAL(a, b)$ , returns False if  $a$  and  $b$  are identical objects, and True otherwise.

$a$  and  $b$  might be real or complex numbers, vectors, matrices, pixmaps, sounds, tables, strings, booleans, or sets.

**OPNOTIN**

$a \notin S = \neg(a \in S)$  returns true iff the value  $a$  is not a member of the set  $S$ .

**OPNULL**

The null operation. Should never be used.

**OPOR**

$\vee$  is the logical or operator.

Truth table for  $p \vee q$ :

$\vee$	1	0
1	1	1
0	1	0

**OPORTHOGONAL**

$m \perp n$  returns True if the integers  $m$  and  $n$  are relatively prime (that is, if  $\gcd(m, n) = 1$ ), and False otherwise.

$v1 \perp v2$  returns True if the  $n$ -dimensional vectors  $v1$  and  $v2$  are orthogonal w.r.t. the standard inner product of  $\mathbb{R}^n$  (or  $\mathbb{E}^n$ ), that is if their scalar (dot) product  $(v1|v2) = v1 \cdot v2$  vanishes.

**OPPERCENT**

$x \% = OPPERCENT(x) = x / 100$ . For instance,  $50 \% = 0.50$ .

**OPPERMILLE**

$x \%_0 = OPPERMILLE(x) = x / 100$ . For instance,  $100 \%_0 = 0.100$ .

**OPPOWER**

$a^b = \text{OPPOWER}(a, b)$  returns a raised to the power of b. If a and b are complex numbers,

$$a^b = \exp(b \cdot \ln(a)) \quad (1)$$

as long as  $a \neq 0$ . (If we are working with real numbers only, it is necessary that  $a > 0$  in most cases.)

If b is a positive number,  $a^b = a \cdot a \cdot \dots \cdot a$ , where the factor a occurs exactly b times.  $a^0 = 1$  (the empty product) unless  $a = 0$ , for  $0^0$  is undefined.  $0^b = 0$  for all non-zero numbers b.  $a^{-b} = 1/a^b$ , and  $a^{(1/n)}$ , for a integer n, is the nth root of a, i.e. one of the roots to the equation  $x^n = a$ . Furthermore, in general,  $(x^a)^b = x^{(ab)}$ .

For real numbers, the following applies to  $a^{(1/n)}$ : If  $a > 0$  and n is even, there are two roots, and the positive root is returned (example:  $16^{(1/2)} = 4$ ). If  $a > 0$  and n is odd, there is only one root.

For complex numbers, the primary definition (1) is used to compute most values. The principal branch of ln is utilized, where the argument of a complex number lies within  $]-\pi, \pi]$ .

**OPSET**

$\{ X1, X2, \dots, Xn \}$  returns the set of the elements X1, X2, ..., and Xn, which might be real or complex numbers, vectors, matrices, strings, or pixmaps, but not booleans, sounds, tables or other sets.

**OPSUB**

$a - b = \text{OPSUB}(a, b)$  returns the difference between a and b. a and b can be real or complex numbers, vectors, or matrices.

**OPUNION**

$A \cup B$  returns the union of the sets A and B.

**OPVECT**

Create a vector or matrix.

$(a1, a2, \dots, an) = \text{OPVECT}(a1, a2, \dots, an)$  returns the n-dimensional vector with real or complex coordinates  $a_i, i=1, 2, \dots, n$ .

$(v1, v2, \dots, vm) = \text{OPVECT}(v1, v2, \dots, vm)$  returns the  $n \times m$  real or complex matrix where the ith column is the vector  $v_i, i=1, 2, \dots, m$ , and  $v_i$  is a n-dimensional real or complex vector.

For instance, the two-dimensional unit matrix is  $((1, 0), (0, 1))$ .

**OPXOR**

$\vee$  is the logical xor (exclusive or) operator.

Truth table for  $p \vee q$ :

$\vee$	1	0
1	0	1
0	1	0

**ord**

$\text{ord}(s)$ , where s is a single character, i.e. a string of length one, returns the Unicode codepoint (or, for the 128 first characters, equivalently, the ASCII code) of s.

Example:  $\text{ord}("<") = 8882$

**parseOperators**

$\text{parseOperators}(s)$ , where s is a string containing a valid AlgoSim expression using operators, returns the expression s, as a string, where all operators have been translated to function calls using the default operator table.

Example:  $\text{parseOperators}("<4 \cdot (x^2+4)!<")$  will return  $\text{OPMUL}(4, \text{OPFACT}(\text{OPADD}(\text{OPPOWER}(x,2),4)))$

**playSound**

$\text{playSound}(s)$  plays the sound named s synchronically. s is a string representing the valid identifier of an AlgoSim sound object.  $\text{playSound}$  does not return until the playback of the sound is complete.

**pmAddSizeToEdges**

$\text{pmAddSizeToEdges}(pm, r, b, l, t)$  returns the pixmap pm with r, b, l, and t black pixels added to the right, bottom, left, and top, respectively.

**pmBlend**

$\text{pmBlend}(pm1, pm2, x, y, t, opq)$  returns the pixmap pm1 with the pixmap pm2 added on top of it, with the pixel (x, y) under its top-left pixel, using the pixel blend method t (an integer), and the opacity  $opq \in [0, 1]$ .

The following pixel blend methods can be used:

1. Normal
2. Average
3. Lighten
4. Darken
5. Add
6. Subtract
7. Distance
8. Negation
9. Exclusion
10. Multiply
11. Screen
12. SoftLight
13. HardLight
14. Overlay
15. Dodge
16. InvDodge
17. Burn
18. InvBurn
19. Reflect
20. InvReflect
21. Freeze
22. InvFreeze
23. Stamp
24. InvStamp
25. Cosine
26. Xor
27. And
28. Or
29. Red
30. Yellow
31. Green
32. Cyan
33. Blue
34. Magenta
35. Dissolve
36. PartialDissolve

**pmContrast**

$\text{pmContrast}$  is not implemented in this version.

**pmCreateGradientPixmap**

$\text{pmCreateGradientPixmap}(W, H, C1, C2, \text{type})$  creates a  $W \times H$  pixmap containing a linear gradient from the colour C1 to the colour C2. C1 and C2 are colour codes, and W, and H are positive integers. If type is "horizontal", a horizontal gradient is created. If type is "vertical", a vertical gradient is created.

Example:  $\text{pmCreateGradientPixmap}(100, 25, \text{rgb}(1, 0, 0), \text{rgb}(0, 0, 0), \text{"horizontal"})$

**pmDistortHue**

pmDistortHue(pm) changes in a random fashion the hue component of each pixel by at most 10 degrees and returns the result.

pmDistortHue(pm, shift) changes in a random fashion the hue component of each pixel by at most shift degrees.

**pmDistortRGB**

pmDistortRGB(pm) applies a random distortion to the R, G, and B components of each pixel separately, and returns the result. Each R, G, and B component is shifted by -1, 0, or 1.

pmDistortRGB(pm, r) applies a random distortion to the R, G, and B components of each pixel separately, and returns the result. Each R, G, and B component is shifted by -r, -r + 1, ..., 0, ..., r - 1, or r. r is an integer between 1 and 255.

**pmDistortSpace**

pmDistortSpace(pm) applies a small random distortion to the pixels of the pixmap pm, and returns the result. Specifically, each pixel will be moved -1, 0, or 1 pixels to the right and -1, 0, or 1 pixels downwards (The distortion applies to each pixel position once, from (0, 0), to (width - 1, height - 1) in row-major order.)

pmDistortSpace(pm, r) moves each pixel a maximum of r pixels to the left/right and a maximum of r pixels upwards/downwards. Hence, the default value of r is 1.

**pmEdgeDetection**

pmEdgeDetection(pm), where pm is a pixmap, returns the pixmap containing the grey-scaled edges of pm.

pmEdgeDetection(pm, "horizontal"), where pm is a pixmap, returns the pixmap containing the grey-scaled edges of pm. Only horizontal edges are detected.

pmEdgeDetection(pm, "vertical"), where pm is a pixmap, returns the pixmap containing the grey-scaled edges of pm. Only vertical edges are detected.

**pmEmboss**

pmEmboss(pm), where pm is a pixmap, returns the pixmap with an embossed effect.

pmEmboss(pm, "horizontal"), where pm is a pixmap, returns the pixmap with a horizontal embossed effect.

pmEmboss(pm, "vertical"), where pm is a pixmap, returns the pixmap with a vertical embossed effect.

**pmFixHue**

pmFixHue(pm, hue) returns the pixmap pm with the hue of all pixels set to the fixed value hue. The saturation and value of each pixel is not affected.

**pmFlipH**

pmFlipH(pm) returns the pixmap pm reflected in the line  $x = w / 2$ , where w is the width of the pixmap.

**pmFlipV**

pmFlipV(pm) returns the pixmap pm reflected in the line  $y = h / 2$ , where h is the height of the pixmap.

**pmFloodFill**

pmFloodFill(pm, x, y, c) performs a flood-fill operation on the pixmap pm and returns the result. x and y are non-negative integers specifying the point of bucket splash-out, and c is the colour code of the the paint.

**pmGetRAMSize**

pmGetRAMSize(pm) returns the amount of RAM, in bytes, occupied by the pixmap pm.

**pmGetRect**

pmGetRect(pm, x0, x1, y0, y1) returns the rectangular region  $x \in [x0, x1], y \in [y0, y1]$  of the pixmap pm.

**pmHeight**

pmHeight(pm) returns the height of the pixmap pm.

**pmHSVAdjustment**

pmHSVAdjustment(pm, Hmode, Hval, Smode, Sval, Vmode, Vval) returns the pixmap pm with the hue, saturation, and value components of each pixel transformed separately according to the component's Xmode and Xval (X = H, S, or V) parameters.

Each Xmode can be either "fixed", "add", or "mul".

\* If Xmode is "fixed", each pixel's X component will be set to Xval.

\* If Xmode is "add", each pixel's X component will be increased by Xval, that is, its new value will be the sum of its old value and Xval.

\* If Xmode is "mul", each pixel's X component will be multiplied by Xval, that is, its new value will be the product of its old value and Xval.

Notice that the component identity operation is achieved either by setting Xmode = "add" and Xval = 0 or by setting Xmode = "mul" and Xval = 1.

Notice! The resulting saturation and value components are finally truncated to [0, 1].

Examples: Let pm be a pixmap variable. Then

pmHSVAdjustment(pm, "fixed", 0, "add", 0, "add", 0)

will return the pixmap pm where the hue of each pixel has been set to 0 (red).

pmHSVAdjustment(pm, "add", 180, "add", 0, "add", 0)

will return the pixmap pm where the hue of each pixel has been shifted by 180 degrees.

pmHSVAdjustment(pm, "add", 0, "mul", 0.5, "add", 0)

will return the pixmap pm where the saturation of each pixel has been halved.

**pmHSVCombine**

Essentially, the pmHSVCombine function creates a pixmap whose hue, saturation, and value components are taken from three different pixmaps.

pmHSVCombine(H, S, V) creates a pixmap P such that

\* the hue component of pixel (i, j) in P is equal to the hue component of (i, j) in H,

\* the saturation component of pixel (i, j) in P is equal to the saturation component of (i, j) in S, and

\* the value component of pixel (i, j) in P is equal to the value component of (i, j) in V.

The pixmaps H, S, and V need to have the same dimensions, and P will also have these dimensions.

**pmHSVSplit**

Essentially, the pmHSVSplit function extracts the hue, saturation, and value components of a pixmap.

pmHSVSplit(pm, h, s, v), where pm is a pixmap, creates three new pixmaps, named h, s, and v (which have to be strings



representing valid identifiers), where h, s, and v are copies of pm, but

- \* h has all saturation and value components equal to unity,
- \* s has all hue and value components equal to zero and unity, respectively, and
- \* v has all hue and saturation components equal to zero and unity, respectively.

#### pmInvert

pmInvert(pm) returns the pixmap pm with all colours inverted. Thus the pixel (r, g, b) is replaced by (1-r, 1-g, 1-b) in RGB coordinates. This maps, for instance,

```
white (1, 1, 1) -- black (0, 0, 0),
red (1, 0, 0) -- aqua (0, 1, 1),
green (0, 1, 0) -- fuchsia (1, 0, 1),
blue (0, 0, 1) -- yellow (1, 1, 0).
```

#### pmInvertLightness

pmInvertLightness(pm) returns the pixmap pm after the HSL lightness (L) coordinate of each pixel has been inverted, that is replaced by one minus the original value.

#### pmInvertValue

pmInvertValue(pm) returns the pixmap pm after the HSV value (V) coordinate of each pixel has been inverted, that is replaced by one minus the original value.

#### pmMöbius

pmMöbius(pm) applies a Möbius transform to the pixmap pm, and returns the result.

#### pmNoise

pmNoise(pm) returns the pixmap pm with white random pixel noise. Specifically, each pixel in the result has a 50 % chance of being replaced with rgb(1, 1, 1).

pmNoise(pm, P) returns the pixmap pm with white random pixel noise. Specifically, each pixel in the result has a P ∈ [0, 1] chance of being replaced with rgb(1, 1, 1).

pmNoise(pm, P, c) returns the pixmap pm with coloured pixel noise. Specifically, each pixel in the result has a P ∈ [0, 1] chance of being replaced with c, a colour code.

Example:

```
pmNoise(pm, 0.3, rgb(1, 0, 0))
```

pmNoise(pm, P, 30000000#16) returns the pixmap pm with coloured pixel noise. Specifically, each pixel in the result has a P ∈ [0, 1] chance of being replaced with a random colour.

Example:

```
pmNoise(pm, 0.3, 30000000#16)
```

Hint: You might want to declare a global constant

```
randomColour := 30000000#16
```

in startup.prg.

#### pmPixelate

pmPixelate(pm, w, h) returns the pixmap pm composed of blocks of the same colour, with width w and height h.

#### pmRemoveSizeFromEdges

pmRemoveSizeFromEdges(pm, r, b, l, t) returns the pixmap pm with r, b, l, and t pixels removed from the right, bottom, left, and top, respectively.

#### pmReplaceColour

pmReplaceColour(pm, x, y) returns the pixmap pm with all pixels with the colour x (a colour code) replaced by the colour y (another colour code).

#### pmResize

pmResize(pm, w, h) returns the pixmap pm with the width and height changed to w and h, respectively, simply by removing pixels from the right and bottom.

#### pmRGBAdjustment

pmRGBAdjustment(pm, Rmode, Rval, Gmode, Gval, Bmode, Bval) returns the pixmap pm with the red, green, and blue intensities of each pixel transformed separately according to the component's Xmode and Xval (X = R, G, or B) parameters.

Each Xmode can be either "fixed", "add", or "mul".

\* If Xmode is "fixed", each pixel's X component will be set to Xval.

\* If Xmode is "add", each pixel's X component will be increased by Xval, that is, its new value will be the sum of its old value and Xval.

\* If Xmode is "mul", each pixel's X component will be multiplied by Xval, that is, its new value will be the product of its old value and Xval.

Notice that the component identity operation is achieved either by setting Xmode = "add" and Xval = 0 or by setting Xmode = "mul" and Xval = 1.

Notice! The resulting red, green, and blue intensities are finally truncated to [0, 1].

Examples: Let pm be a pixmap variable. Then

```
pmRGBAdjustment(pm, "fixed", 0, "add", 0, "add", 0.2)
```

will return the pixmap pm where the red component of each pixel has been set to zero and the blue component of each pixel has been increased by 0.2.

```
pmRGBAdjustment(pm, "mul", 5, "add", 0, "add", 0)
```

will multiply the red component of each pixel by 5.

#### pmRGBCombine

Essentially, the pmRGBCombine function creates a pixmap whose red, green, and blue components are taken from three different pixmaps.

pmRGBCombine(R, G, B) creates a pixmap P such that

\* the red component of pixel (i, j) in P is equal to the red component of (i, j) in R,

\* the green component of pixel (i, j) in P is equal to the green component of (i, j) in G, and

\* the blue component of pixel (i, j) in P is equal to the blue component of (i, j) in B.

The pixmaps R, G, and B need to have the same dimensions, and P will also have these dimensions.

#### pmRGBSplit

Essentially, the pmRGBSplit function extracts the red, green, and blue components of a pixmap.

pmRGBSplit(pm, r, g, b), where pm is a pixmap, creates three new pixmaps, named r, g, and b (which have to be strings representing valid identifiers), where r, g, and b are copies of pm, but

\* r has all green and blue components equal to zero,

\* g has all red and blue components equal to zero, and

\* b has all red and green components equal to zero.

#### **pmRot90N**

pmRot90N(pm) returns the pixmap pm rotated 90° clockwise (that is, negative direction).

#### **pmRot90P**

pmRot90P(pm) returns the pixmap pm rotated 90° counter-clockwise (that is, positive direction).

#### **pmRotate**

pmRotate(pm, θ) returns the pixmap pm rotated θ radians anti-clockwise, expanding the canvas as necessary.

Example: pmRotate(image, 45°)

#### **pmRotateEx**

pmRotateEx(pm, θ) returns the pixmap pm rotated θ radians anti-clockwise, without expanding the canvas even if necessary. Thus, the resulting image is likely cropped.

Example: pmRotateEx(image, 5°)

#### **pmScale**

pmScale(pm, x, y) returns the pixmap pm scaled with factors x and y in the horizontal and vertical directions, respectively. Thus, pmScale(pm, 1, 1) is the identity transformation. If x = y, the transformation preserves the shapes of object. pmScale(pm, 2, 2) magnifies all lengths by a factor 2, and the total area by a factor  $2^2 = 4$ . pmScale(pm, 2, 1) doubles the width of the pixmap, preserving the height, thus distorting shapes.

#### **pmShear**

pmShear(pm, x, y) applies a shearing transform to the pixmap pm, and returns the result. x and y are the magnitudes in the horizontal and vertical direction, respectively. Thus, pmShear(pm, 0, 0) is the identity transformation, and pmShear(pm, 0.2, 0) applies a small horizontal shear.

It is required that  $xy \neq 1$ .

#### **pmShiftHue**

pmShiftHue(pm, h) returns the pixmap pm with the hue of all pixels increased (modulo 360) by h. The hue is a number between 0 and 360.

#### **pmSwapBW**

pmSwapBW(pm) returns the pixmap pm with all black pixels replaced by white pixels, and vice versa.

#### **pmSwapRGBComponents**

pmSwapRGBComponents(pm, which) returns the pixmap pm with two of its RGB channels swapped. which is a two-character string specifying the channels by name: "R", "G", and "B".

Example:

```
pmSwapRGBComponents(pm, "rg")
```

#### **pmTiles**

pmTiles(pm, N), where pm is a pixmap and N a positive integer, returns a pixmap, no smaller than pm, consisting of a rectangular grid of  $N \times N$  rectangular tiles of pm, in a random order, such that each point in pm is represented in the result in exactly one tile.

pmTiles(pm, Nx, Ny) creates a tiled picture of Nx rows and Ny columns.

pmTiles(pm, Nx, Ny, dx) creates a tiled picture of Nx rows and Ny columns. dx is the spacing between tiles, both vertically and horizontally.

pmTiles(pm, Nx, Ny, dx, c) creates a tiled picture of Nx rows and Ny columns. dx is the spacing between tiles, both vertically and horizontally. c is the colour of the spacing.

By default, dx = 2 and c = rgb(1, 1, 1) (white). Notice that, when  $N = Nx = Ny = 1$ , this function can be used to create a solid-colour frame around pm.

#### **pmToBitmap**

pmToBitmap(pm, x) returns the pixmap pm with only two colours, black and white, where the black pixels come from "dark" pixels in pm, and white pixels come from "bright" pixels in pm. The threshold for a pixel to be considered "bright" is given by  $x \in [0, 1]$ . Thus, if  $x = 0$ , all pixels will be considered bright, and if  $x = 1$ , no pixels will be considered bright.

The "brightness" property is defined as the HSL lightness value.

#### **pmToGreyscale**

pmToGreyscale(pm) returns a grey-scale representation of the pixmap pm. A pixel with coordinates (h, s, v) is replaced by (h, 0, v) in HSV coordinates.

#### **pmToMonochromatic**

pmToMonochromatic(pm, hue) returns the pixmap pm with the hue of all pixels set to the fixed value hue, and the saturation set to 1. The value of each pixel is not affected.

#### **pmTransform**

pmTransform(pm, M) returns the image of the pixmap after being transformed by the linear transformation (i.e., a  $2 \times 2$  square matrix) M.

pmTransform(pm, a11, a12, a21, a22) returns the image of the pixmap after being transformed by the linear transformation (i.e., a  $2 \times 2$  square matrix)  $((a11, a12), (a21, a22))$ .

#### **pmWidth**

pmWidth(pm) returns the width of the pixmap pm.

#### **pNorm**

pNorm(v) returns the p-norm of the vector  $v = (a1, a2, \dots, an)$ , i.e. the real number

$$\text{norm}(v) = (|a1|^p + |a2|^p + \dots + |an|^p)^{1/p}.$$

#### **polarCoords**

polarCoords(S) applies the transformation

$$\begin{aligned} x &= r \cdot \sin(\varphi) \\ y &= r \cdot \cos(\varphi) \end{aligned}$$

to all two-dimensional polar real vectors (r, φ) in the set S, and returns the new set of Cartesian coordinates (x, y).

This is useful for plotting polar graphs. Simply create a set S of polar coordinates (r, φ) and then transform it using

```
S := polarCoords(S)
```

after which it can be plotted using drawSet, drawLines, etc.

```
Example: A cardioid
cardioid := createImage("4*(cos(φ) + 1), φ)", "φ", [0,
2*π, 0.001])
cardioid := polarCoords(cardioid)
drawSet("cardioid")
```

#### **polyFit**

polyFit(n, X, Y) returns the coefficients (a0, a1, a2, ..., an) of the polynomial  $a0 + a1x + a2x^2 + \dots + anx^n$  of degree n

that, in a least-squares sense, is the best polynomial curve of degree  $n$  that approximates the point set  $\{(X_1, Y_1), (X_2, Y_2), \dots, (X_k, Y_k)\}$  where  $k$  is the dimension of both real vectors  $X$  and  $Y$ .

#### postMessage

`postMessage(h, m, w, l)` will post the message  $m$  to the window with handle  $h$ , and passing along the `wparam`  $w$  and `lparam`  $l$ .  $h$ ,  $m$ ,  $w$ , and  $l$  are unsigned integers.

`postMessage` returns `True` or `False`, depending on what happens with the message.

Examples: `postMessage(getParameter("handle", "application"), 112#16, F140#16, 0)` will start the screen-saver  
`postMessage(getParameter("handle", "window"), 112#16, F030#16, 0)` will maximize the AlgoSim window.

#### prevPrime

`prevPrime(n)` returns the greatest prime number smaller than or equal to  $n$ , if such a number exists (i.e. if  $n \geq 2$ ).

#### prime

`prime(n)` returns the  $n$ th prime number, the first ( $n=1$ ) being 2.

#### print

`print(s)` prints the text (string)  $s$  in the console.

#### processMessages

Obsolete.

#### product

`product(a1, a2, ..., an)` returns the product of the real or complex numbers  $a_1, a_2, \dots$ , and  $a_n$ .

`sum(v)` returns the product of all components in the real or complex vector  $v$ .

#### QR

`QR(A)` computes the QR decomposition for the real square matrix  $A$ , i.e. computes an orthogonal matrix  $Q$  and an upper-triangular matrix  $R$ , such that  $QR=A$ . The matrices  $Q$  and  $R$  are stored as the identifiers "Q" and "R", respectively.

In the current implementation, QR works only for real square non-singular matrices  $A$ .

#### questionBox

`questionBox(str)` displays a question box with the message `str` (a string) and returns `True` if the user clicks the "Yes" button, and `False` if the user clicks the "No" button.

#### random

`random(S)` picks a random element from the set  $S$ . Each member of the set has equal probability of being chosen, regardless of the data types in  $S$ .

#### randomInt

`randomInt(n)` returns a random integer in  $[0, n[$ . Observe that  $n$  is never returned: all in all,  $n$  distinct numbers may be returned.

`randomInt(a, b)` returns a random integer in  $[a, b]$ . Thus  $a - b + 1$  distinct values may be returned.

#### randomize

`randomize(0)` selects a new, random, seed for random number generation.

#### randomReal

`randomReal(0)` returns a random real number in  $]0, 1[$ .

#### rank

`rank(M)` returns the rank of the real or complex matrix  $M$ , i.e. the dimension of the column (or, equivalently, row) space of  $M$ .

#### Re

`Re(z)` returns the real part of the complex number  $z$ . For a real number  $x$ , `Re(x) = x`.

#### rect

`rect(x)` is the normalized rectangular function. `rect(x) = 1` if  $|x| < 1/2$ , and `rect(x) = 0` otherwise.

#### redraw

`redraw(0)` redraws the current 2D visualization window.

Example: `spiral := createImage("FresnelC(t), FresnelS(t)", "t", [-10, 10, 0.01])`  
`drawLines("spiral")`  
`spiral := createImage("FresnelS(t), FresnelC(t)", "t", [-10, 10, 0.01])`  
`redraw(0)`

#### redraw3

`redraw3(0)` redraws the current 3D visualization window.

#### reduceMatrix

`reduceMatrix(mat, n)`, where `mat` is a real or complex matrix and  $n$  is a positive integer, returns the matrix obtained by taking every  $n$ th row of `mat`, starting with the first row.

#### reduceSound

`reduceSound(snd, n)`, where `snd` is a sound and  $n$  is a positive integer, returns the sound whose samples are every  $n$ th sample of `snd`, and whose sample rate is set to  $(1/n)$ th of the sample rate of `snd`. Hence, the returned sound has the same duration (in seconds) as `snd`, and is equal to the sound one would have obtained by recording the physical sound at a sampling rate equal to  $(1/n)$ th of the sample rate used to record `snd`.

`reduceSound(snd, n)` amounts to the same thing as

```
samples := sndGetSamples(snd)
samples := reduceMatrix(samples, n)
snd := sndMatrixToSound(samples, sndGetSampleRate(snd) / n)
```

but is faster.

#### reduceVector

`reduceVector(v, n)`, where  $v$  is a real or complex vector and  $n$  is a positive integer, returns the vector obtained by taking every  $n$ th component of  $v$ , starting with the first component.

#### regKeyExists

`regKeyExists(RootKey, KeyName)` returns `True` if the registry key named `KeyName` (a string) exists under `RootKey`. `RootKey` is "HKCU", "HKLM", "HKCR", "HKCC", "HKPD", "HKDD", or "HKU".

#### regReadValue

`regReadValue(RootKey, KeyName, ValName)` returns the data associated with the registry value `ValName` (a string) in the key named `KeyName` (a string) under `RootKey`. `RootKey` is "HKCU", "HKLM", "HKCR", "HKCC", "HKPD", "HKDD", or "HKU".

#### regValExists

`regValExists(RootKey, KeyName, ValName)` returns `True` if the registry value `ValName` (a string) exists in the key named `KeyName` (a string) under `RootKey`. `RootKey` is "HKCU", "HKLM", "HKCR", "HKCC", "HKPD", "HKDD", or "HKU".

## reloadOperatorTable

reloadOperatorTable(0) reloads the operator table.

## reloadPrograms

reloadPrograms(0) reloads and reinterprets all AlgoSim programs (\*.prg files) in the common and user-specific AS program folders.

Each time an AlgoSim program is changed, the program needs to be reinterpreted before the new version can be used in AlgoSim.

## removeDrawing

removeDrawing(s) removes the set named s from the current 2D visualization window.

## removeDrawing3

removeDrawing3(s) removes the set named s from the current 3D visualization window.

## removeDuplicates

removeDuplicates(v) returns the real or complex vector v with all duplicates removed. Thus the result is a vector where each number occurs at no more than one component.

## removeDuplicatesFromSet

removeDuplicatesFromSet(S) removes all duplicates from the set S, so it becomes a "true set".

Example:

Given a set

$$S = \{(1, 2), (1, 2), (5, 8), (2, 9)\},$$

removeDuplicatesFromSet(S) will return

$$\{(1, 2), (5, 8), (2, 9)\}.$$

## renameFile

renameFile(f1, f2) renames the file f1 (a string containing a valid file name) to f2 (a string containing a valid file name).

## reverse

reverse(s) returns the string s spelled backwards.

## rgb

rgb(r, g, b) returns the colour code for the colour with the RGB coordinates r, g, and b.

## rgbToHsv

rgbToHsv(r, g, b) returns the three-dimensional HSV coordinates of the colour given by the RGB coordinates r, g, and b, as a real vector.

## ROT13

ROT13(str) transforms the string str using ROT13, i.e. shifts all letters 13 positions to the right (modulo 26) in the English alphabet, preserving the case of each letter and all non-alphabetic characters such as spaces and punctuation marks.

Obviously,  $\text{str} \mapsto \text{ROT13}(\text{str})$  is an involution, i.e.  $\text{str} \mapsto \text{ROT13}(\text{ROT13}(\text{str}))$  is the identity operation on strings.

## round

round(x) returns the integer closest to x. If x is equally close to both  $\text{ceil}(x)$  and  $\text{floor}(x)$ ,  $\text{ceil}(x)$  is returned.

## rowAddMul

rowAddMul(M, m, n, k) returns the real or complex matrix M after the k times the nth row has been added to the mth row. This is an elementary row operation, unless  $k = 0$ .

Note: to change the matrix M, it is necessary to write

$$M := \text{rowAddMul}(M, m, n, k).$$

## rowMove

rowMove(M, n, m) returns the real or complex matrix M after rows n and m have been swapped. This is an elementary row operation.

Note: to change the matrix M, it is necessary to write

$$M := \text{rowMove}(M, n, m).$$

## rowScale

rowScale(M, n, k) returns the real or complex matrix M with the nth row multiplied by the real or complex number k.

Unless  $k = 0$ , this is an elementary row operation, not affecting the solution of a linear equation system.

Note: to change the matrix M, it is necessary to write

$$M := \text{rowScale}(M, n, k).$$

## saveMatrixToFile

saveMatrixToFile(M, s) saves the real or complex matrix M to the file with file name s.

## savePixmapToFile

savePixmapToFile(pm, s) save the pixmap pm to a file. s is the fully-qualified name of the file, including extension ".asd", ".png", ".bmp", or ".xbm".

## savePointSetToFile

savePointSetToFile(set, fn) saves the point set set to the file with name fn, a string containing a fully-qualified file name, including extension ".asd".

Important! If the set set contains other elements than real points (vectors), only the real points (vectors) will be saved, and no other elements.

By default, the coordinates of the points are saved with extended precision (10 bytes per coordinate). You can also choose double precision (8 bytes per coordinate) or single precision (4 byte per coordinate), by specifying "extended", "double", or "single" as a third parameter (a string).

## saveSoundToFile

saveSoundToFile(snd, s) saves the sound snd to the file with name s. s must be a fully-qualified file name with extension ".wav".

## saveStructToFile

saveStructToFile(str, fn) saves the structure str to file f, a string containing a valid file name.

## saveTableDataToTextFile

saveTableDataToTextFile(tbl, s) saves the table tbl to the file s. s is a fully-qualified name of the file. Only the textual data is saved, and thus all formatting is discarded.

## saveTableToFile

saveTableToFile(tbl, s) saves the table tbl to the file s. s is a fully-qualified name of the file.

## saveTextToFile

saveTextToFile(s, S) saves the string s to the file named S.

## saveVectorToFile

saveVectorToFile(v, s) saves the real or complex vector v to the file with file name s.

**saveViewAsBitmap**

saveViewAsBitmap(s) saves the current 2D visualization window to the bitmap with file name s, a fully-qualified name.

saveViewAsBitmap(s, w, h) saves the current 2D visualization window to the bitmap with file name s, a fully-qualified name. w and h is the width and height of the bitmap, respectively. The bitmap is rerendered using this canvas size.

**saveViewAsBitmap3**

saveViewAsBitmap3(s) saves the current 3D visualization window to the bitmap with file name s, a fully-qualified name.

saveViewAsBitmap3(s, w, h) saves the current 3D visualization window to the bitmap with file name s, a fully-qualified name. w and h is the width and height of the bitmap, respectively. The bitmap is rerendered using this canvas size.

**searchUpdates**

searchUpdates(0) searches for updates. It displays the current and newest version of AlgoSim, and returns True if there is a newer version, and False if you already are running the newest version.

**sec**

sec(x) = 1 / cos(x). x is a real or complex number.

**sech**

sech(x) is the hyperbolic secant, i.e. sech(x) = 1 / cosh(x).

**secondsBetween**

secondsBetween(t1, t2) returns the number of seconds between the date and time structures t1 and t2.

**selectColour**

selectColour(0) displays a Rejbrand Colour Selector dialog, in which the user can select a colour. The colour-code is returned.

selectColour(c) displays a Rejbrand Colour Selector dialog, in which the user can select a colour. The colour-code is returned. The default colour has the code c.

selectColour(c, str) displays a Rejbrand Colour Selector dialog, in which the user can select a colour. The colour-code is returned. The default colour has the code c, and the title of the dialog is str, a string.

**sendBugReport**

sendBugReport(str) sends the string str to the developer of AlgoSim (Andreas Rejbrand). This function is intended to be used for bug reports, suggestions, and other feedback.

Example: sendBugReport("sin appears not to work if ...")

Hint! If you want to write a long message, it might be much more convenient to do so in a text editor window. To do so, simply use the command

sendBugReport(inputDialog(0)).

**sendMessage**

sendMessage(h, m, w, l) will send the message m to the window with handle h, and passing along the wparam w and lparam l. h, m, w, and l are unsigned integers.

sendMessage returns the value returned by the function receiving the message.

Examples: sendMessage(getParameter("handle", "application"), 112#16, F140#16, 0) will start the screen-saver

sendMessage(getParameter("handle", "window"), 112#16, F030#16, 0) will maximize the AlgoSim window.

**sendMidiMsg**

sendMidiMsg(n, x, y) sends the MIDI message (n, x, y) to the computer's sound card.

**setAntiAliasing**

setAntiAliasing(b) activates (b=true) or disables (b=false) the anti-aliasing of lines in the current 3D visualization window.

**setApproxMode**

setApproxMode(b) sets the approximation mode to b, true or false.

In approximation mode, very small numbers (such as 10<sup>-30</sup>) are approximated to zero. In most computations this is desired, for in approximation mode sin( $\pi$ ) = 0, not -5.42101086243·10<sup>-20</sup>.

However, when actually working with very small numbers, such as Planck's constant, approximation mode cannot, of course, be used.

**setAxisStyle**

setAxisStyle(a, s) sets the style of the axes listed in a, a comma-separated string of axes in {"x", "y"}, to s, a CSS style string, in the current 2D visualization window.

Example: setAxisStyle("x, y", "colour:red; number-distance: 2; text-colour:yellow; tick-distance: 2; tick-colour:red")

Possible style parameters: colour, number-distance, numbers-visible, text-colour, ticks-visible, tick-distance, tick-colour.

**setAxisStyle3**

setAxisStyle3(a, s) sets the style of the axes listed in a, a comma-separated string of axes in {"x", "y", "z"}, to s, a CSS style string, in the current 3D visualization window.

Example: setAxisStyle3("x, y, z", "colour:red; number-distance: 2; text-colour:yellow; tick-distance: 2; tick-colour:red")

Possible style parameters: colour, number-distance, numbers-visible, text-colour, ticks-visible, tick-distance, tick-colour.

**setCameraPos**

setCameraPos(v) sets the camera position to v ∈ R<sup>3</sup> in the current 3D visualization window.

setCameraPos(v, d) sets the camera position to v ∈ R<sup>3</sup> in the current 3D visualization window. The camera is looking in the d ∈ R<sup>3</sup> direction.

setCameraPos(v, d, u) sets the camera position to v ∈ R<sup>3</sup> in the current 3D visualization window. The camera is looking in the d ∈ R<sup>3</sup> direction, and u ∈ R<sup>3</sup> is the vertical (up) direction.

**setComplexMode**

setComplexMode(b) sets the complex mode to b, true or false.

In complex mode, complex numbers are used by all functions. For instance, sqrt(-4) = 2i and arcsin(3) = 1.57079632679 - 1.76274717404·i. However, if not in complex mode (i.e., in real mode), these expressions are undefined.

**setDir**

setDir(s) sets the current working directory to s. s may be an absolute or relative path.

**setFullscreen**

setFullscreen(b) sets the fullscreen mode to b, true or false.

**setGenerator**

Not implemented in this version.

**setLight**

setLight(b) turns realistic lightning of 3D objects on (true) or off (false).

Generally, geometric objects such as boxes, cones, spheres and cylinders that are drawn using the special functions drawBox3, drawCone, drawSphere, and drawCylinder are best viewed with lightning, whereas curves and point sets are best drawn without lightning.

Examples: setLight(true) or setLight(false)

**setLightPos**

setLightPos(v) moves the light source (spotlight) to  $v \in \mathbb{R}^3$ .

**setMatElement**

setMatComponent(s, i, j, x) sets the entry i, j of the matrix named s, a string representing the valid identifier of a matrix variable, to x, a real or complex number.

**setNumDecimals**

setNumDecimals(n) sets the number of digits following the decimal point in all numerical output to n.

**setPostProcessing**

setPostProcessing(S) sets the set of post-processing operations to S in the current 2D visualization-window. S is a set of string identifiers of pre-defined post-processing operations, including

- invert (inverts each pixel's R, G, and B value),
- invertValue (inverts each pixel's V value),
- invertLightness (inverts each pixel's L value),
- greyscale (sets each pixel's S value to zero),
- swapBW (swaps white and black pixels),
- flipV (flips vertically, i.e. reflects in the line  $y = h/2$ , where h is the height of the pixmap), and
- flipH (flips horizontally, i.e. reflects in the line  $x = w/2$ , where w is the width of the pixmap).

**setRandomSeed**

setRandomSeed(N) sets the seed for random number generation to N. After each time the seed has been set to a fixed value N, the same sequence of random numbers will be returned by the various random number functions.

**setSetMode**

setSetMode(b) sets the true set mode to b, true or false.

In true set mode, if an element x is added to the set X, it is first checked if  $x \in X$ . If so, the element is not added. If not in true set mode, x is added without checking the existence of x in X. Thus, X may contain two copies of x afterwards. This will not affect any computations, but two copies of x will be shown when the contents of X is printed out.

When working with small sets, it is recommended to use true sets, but when working with large sets (containing some million elements), the performance penalty for checking (i.e. iteration over a million elements) each time an element is added (perhaps a million times) is not acceptable. Thus, when working with large sets, please do not use true sets.

**setTableCellData**

setTableCellData(s, i, j, x) sets the entry i, j of the table named s, a string representing the valid identifier of a table variable, to x, a string.

**setTableCellStyle**

setTableCellStyle(s, i, j, x) sets the style (CSS) of the entry i, j of the table named s, a string representing the valid identifier of a table variable, to x, a string.

**setToMat**

setToMat(S) returns the  $n \times 1$ -dimensional matrix with elements from the set S.

**setToVect**

setToVect(S) returns the n-dimensional vector with elements in S.

**setVectComponent**

setVectComponent(s, n, x) sets the nth component of the vector named s, a string representing the valid identifier of a vector variable, to x, a real or complex number.

**setVectMode**

setVectMode(b) sets the basis vector notation to b, true or false.

In basis vector notation, a vector  $(x, y, z)$  is written

$$\mathbf{e} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

rather than

$(x, y, z)$ .

**setView**

setView(xmin, xmax, ymin, ymax) sets the 2D viewing window to span xmin to xmax in the horizontal direction, and ymin to ymax in the vertical direction.

**setView3**

setView3(xmin, xmax, ymin, ymax, zmin, zmax) sets the 3D viewing box to the region  $x \in [xmin, xmax]$ ,  $y \in [ymin, ymax]$ , and  $z \in [zmin, zmax]$ . Note that only some functions respect this setting.

**sgn**

sgn(x) returns the sign of the real number x, i.e.  $\text{sgn}(x) = 1$  if  $x > 0$  and  $\text{sgn}(x) = -1$  if  $x < 0$ . In AlgoSim,  $\text{sgn}(0) = 0$ , so that it is not entirely true that  $\text{sgn}(x)$  is the derivative of  $\text{abs}(x)$ .

**showPixmap**

showPixmap(pm) draws the pixmap pm on the screen.

**showProgramCode**

showProgramCode(s) prints the code of the AlgoSim program named s.

**Si**

Si(x) is the sine integral.

$$\text{Si}(x) = \int \sin(t)/t \, dt = \int \text{sinc}(t) \, dt \text{ from } 0 \text{ to } x.$$

**sieveOfEratosthenes**

sieveOfEratosthenes(N) returns a  $(N + 1)$ -dimensional vector whose ith component is 1 if i is a prime number, and 0 otherwise, where  $i = 0, 1, 2, \dots, N$ . Hence, the two first components are both 0, and the third, corresponding to index  $i = 2$ , is the first 1.

sieveOfEratosthenes is very fast, and computing sieveOfEratosthenes(1000000) should take less than a half second on a modern computer.

Example: sieveOfEratosthenes(50) will return (0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0)

Hint! Pass the returned vector to numberVector to get a numbered version of it!

**sin**  
sin(x) returns the sine of x. x is a real or complex number.

Construct the unit circle

$$x^2 + y^2 = 1$$

in R<sup>2</sup>. Draw the line from the origin to the point P at this circle, such that the angle to this line, counted from the positive x-axis (anticlockwise is the positive direction) is equal to x. Then sin(x) is the y-coordinate of P.

For a general complex number z, Euler's identity

$$\sin(z) = (1/2i) \cdot (\exp(iz) - \exp(-iz))$$

defines sin(z). exp is the complex exponential function, defined such that

$$\exp(z) = e^{(\operatorname{Re} z)} \cdot (\cos(\operatorname{Im} z) + i \sin(\operatorname{Im} z))$$

where i is the imaginary unit (i<sup>2</sup> = -1) and Re z and Im z are the real and imaginary parts of z, respectively.

**sinc**  
sinc(x) = sin(x)/x if x ≠ 0, sinc(0) = 1. sinc is the sinc function.

**sinh**  
sinh(x) is the hyperbolic sine, i.e. sinh(x) = (1/2i) · (e<sup>ix</sup> - e<sup>-ix</sup>).

**size**  
size(S) returns the number of members in the set S.

Example: size([1, 100]) = 100

**sleep**  
sleep(x) pauses the execution of the script for x seconds.

**sndAppend**  
sndAppend(snd1, snd2) returns the sound snd1 with snd2 added to the end of it. Thus, the length of the result equals the sum of the length of snd1 and the length of snd2. snd1 and snd2 must have the same sample rate.

**sndGetNumChannels**  
sndGetNumChannels(snd) returns the number of channels in the sound snd.

Example: sndGetNumChannels(createSineTone(400, 1)) = 1

**sndGetNumSamples**  
sndGetNumSamples(snd) returns the number of samples in the sound snd.

Example: sndGetNumSamples(createSineTone(400, 1)) = 48000.

**sndGetSampleRate**  
sndGetSampleRate(snd) returns the sample rate of the sound snd, in samples per second.

Example: sndGetSampleRate(createSineTone(400, 1)) = 48000

**sndGetSamples**  
sndGetSamples(snd) returns a n×c real matrix with the samples of the c-channel sound snd. A given column thus represents an individual channel, and if the elements are f(t), t ∈ [1, n], then f(t) is the speaker membrane's displacement at time t. The scaling is chosen so that f(t) ∈ [-2<sup>31</sup>, 2<sup>31</sup>]. Consequently, when superposing sounds, it is important to make sure that f(t) of the resulting sound never exceeds 2<sup>31</sup> in absolute value.

**sndMakeMultichannel**  
sndMakeMultichannel(snd1, ..., sndN) returns the N-channel sound obtained when using snd1 as the first channel, ..., and sndN as the Nth channel. snd1, ..., and sndN must all have the same sample rate.

**sndMatrixToSound**  
sndMatrixToSound(M, n) returns the sound with the matrix representation M (one column per channel, one row per sample, and each element f(t) is the displacement of the speaker's membrane normalized to lie within [-2<sup>31</sup>, 2<sup>31</sup>]) and n samples per second.

Example: A pure 400 Hz sine tone

```
A := 2^31
ω := 2·π·400
set := createImage("A·sin(ω·t)", "t", [0, 2, 0.0001])
snd := sndMatrixToSound(setToMat(set), 10000)
```

A frequency-modulated 400 Hz sine tone  
set := createImage("A·sin(ω·(sin(t)·t))", "t", [0, 4·π, 0.0001])  
snd := sndMatrixToSound(setToMat(set), 10000)

**sndSplitChannels**  
sndSplitChannels(snd, s1, ..., sN). snd is a sound, and s1, ..., sN are strings with the names of valid identifiers. The sounds s1, ..., sN will be created with the channels 1, ..., N of snd.

Example: sndSplitChannels(song, "l", "r") creates the sounds l and r with the left and right channel of the stereo sound song, respectively.

**sndSuperpose**  
sndSuperpose(snd1, snd2) returns the sound obtained when superposing snd1 and snd2. snd1 and snd2 must have the same sample frequency.

**solve**  
solve(eq, var, x0) returns a root of the equation eq in the var variable, near x0, using the Newton-Raphson method.

Examples: solve("sin(x)/exp(x) + sin(x<sup>2</sup>) = 0", "x", -3) = -3.16520811808

**sort**  
sort(v) returns the vector v sorted numerically.

sort(l) returns the string list (i.e., table with one column) sorted.

**speak**  
speak(X) speaks the value X using the computer's default voice (for instance, Microsoft Anna). X is a string, a number, a vector, a matrix, a boolean, or a table.

Examples: speak(2 + i) will speak "two plus one I".  
speak((1, 2, 3)) will speak "vector: 1, 2, 3".

**sphericalCoords**  
sphericalCoords(S) applies the transformation

$$\begin{aligned}x &= r \cdot \sin(\theta) \cdot \cos(\varphi) \\y &= r \cdot \sin(\theta) \cdot \sin(\varphi) \\z &= r \cdot \cos(\theta)\end{aligned}$$

to all three-dimensional spherical real vectors  $(r, \theta, \varphi)$  in the set  $S$ , and returns the new set of Cartesian coordinates  $(x, y, z)$ .

This is useful for plotting spherical graphs. Simply create a set  $S$  of spherical coordinates  $(r, \theta, \varphi)$  and then transform it using

$S := \text{sphericalCoords}(S)$

after which it can be plotted using `drawSet3`, `drawLines3`, etc.

Example: An illustrative way to draw a grid sphere with radius 4.

```
net := createNet(0, pi, 0.01, pi/12, 0, 2*pi, 0.01, pi/12)
paramNet := createImage("C4, r_1, r_2"), "r", net)
sphere := sphericalCoords(paramNet)
drawSet3("sphere")
```

#### **sq**

`sq(x)` returns  $x^2$  if  $x$  is a real number, a complex number, a real square matrix, or a complex square matrix.

#### **sqrt**

For real numbers, `sqrt(x)`, where  $x \geq 0$ , is the positive root to the equation  $a^2 = x$  with respect to  $a$ .

For complex numbers  $z$ ,

$\text{sqrt}(z) = \exp\left(\frac{1}{2} \ln(z)\right)$ .

#### **start**

`start(s)` executes the operating system command  $s$ , using the Win32 API call `ShellExecute`.

Examples: `start(".")` opens the current working directory.

`start("C:\")` opens the folder `C:\`.

`start("http://rejbrand.se")` opens the URL `rejbrand.se` in the system's default web browser.

`start("mailto:andreas@rejbrand.se")` opens the system's default e-mail client and creates a new message for Andreas Rejbrand (whose e-mail address is `andreas@rejbrand.se`).

`start("winword")` opens Microsoft Word, if installed.

#### **stop**

Obsolete.

#### **strBeginsWith**

`strBeginsWith(S, s)` returns True if the string  $S$  begins with  $s$ , and false otherwise.

#### **strContains**

`strContains(S, s)` returns True if the string  $S$  contains  $s$ , and False otherwise.

#### **strEndsWith**

`strEndsWith(S, s)` returns True if the string  $S$  ends with  $s$ , and false otherwise.

#### **strLeft**

`strLeft(s, n)` returns the  $n$  first (that is, left-most) characters in  $s$ , a string.

Example: `strLeft("Hello World!", 5) = "Hello"`

#### **strPos**

`strPos(s, S)` returns the position of the first character in the string  $s$  of the first occurrence of  $s$  in the string  $S$ .

Example: `strPos("st", "testtest") = 3`

#### **strReplaceAll**

`strReplaceAll(S, x, y)` replaces all occurrences of  $x$  by  $y$  in the string  $S$ .

#### **strRight**

`strRight(s, n)` returns the  $n$  last (that is, right-most) characters in  $s$ , a string.

Example: `strRight("Hello World!", 6) = "World!"`

#### **strSplit**

`strSplit(s, sep)` returns a table with all parts of the string  $s$  that has been obtained by splitting it at each occurrence of the `sep` separation character. `sep` is never a part of any string in the resulting table.

Example: `strSplit("Harry|Ron|Hermione", "|")` returns the table with entries "Harry", "Ron", and "Hermione".

#### **structDeleteMember**

`structDeleteMember(str, mem)` deletes the member `mem` from the structure `str`, a string containing the identifier of a structure variable.

#### **structRenameMember**

`structRenameMember(str, OldName, NewName)` renames the member `OldName` to `NewName` inside the structure `str`. `str` is a string containing the identifier of a structure. This function alters the structure, so you can think of the first parameter as being the structure `str` "passed by reference".

Example:

```
a := createStruct("firstName", "Andreas", "lastName", "Rejbrand", "yearOfBirth", 1987, "IQ", infinity)
```

```
firstName: Andreas
lastName: Rejbrand
yearOfBirth: 1987
IQ: infinity
```

```
structRenameMember("a", "IQ", "wisdom")
```

```
a
```

```
firstName: Andreas
lastName: Rejbrand
yearOfBirth: 1987
wisdom: infinity
```

#### **substring**

`substring(s, i, n)` returns the string of characters  $i$  to  $i + n$  in  $s$ .

Example: `substring("Hello World!", 7, 5) = "World"`

#### **subvector**

`subvector(v, S)` returns the vector composed of the components of  $v$  with indices in the set  $S$ . Thus,  $S$  is a set of integers, between 1 and the dimension `dim(v)` of  $v$ .

Examples: `subvector((3, 5, 7, 9), {2, 4}) = (5, 9)`  
`subvector((3, 5, 7, 9), [2, 4]) = (5, 7, 9)`

#### **sum**

`sum(a1, a2, ..., an)` returns the sum of the real or complex numbers  $a_1, a_2, \dots, a_n$ .



`sum(v)` returns the sum of all components in the real or complex vector `v`.

**swap**

`swap(a, b)` swaps the identifiers `a` and `b`. `a` and `b` are strings containing the identifiers of two variables.

`swap(a, b)` is equivalent to

```
tmp := a
a := b
b := tmp
delete("tmp")
```

(where "tmp" is a previously non-existing variable).

**sysSolve**

`sysSolve(M, v)` returns the unique solution to the matrix equation  $MX = v$ , where  $M$  is a matrix and  $v$  is a vector (here considered a  $n \times 1$  column matrix), if such a solution exists, i.e. if  $\det(M) \neq 0$ .

`sysSolve(M)` returns the unique solution to the equation system represented by the total matrix  $M$ , i.e. the augment of a coefficient matrix and a right-hand side column vector.

`sysSolve(M, v) = sysSolve(A)` if  $A = \text{augment}(M, v)$ .

**tableCols**

`tableCols(tbl)` returns the number of columns in the table `tbl`.

**tableRows**

`tableRows(tbl)` returns the number of rows in the table `tbl`.

**tableToSet**

`tableToSet(tbl)` returns the set of all strings in the table `tbl`.

**tan**

`tan(x) = sin(x) / cos(x)`.  $x$  is a real or complex number.

**tanh**

`tanh(x)` is the hyperbolic cotangent, i.e.  $\tanh(x) = \sinh(x) / \cosh(x)$ .

**taxiNorm**

`taxiNorm(v)` returns the taxi (Manhattan) norm of the vector  $v = (a_1, a_2, \dots, a_n)$ , i.e. the real number

`taxiNorm(v) = |a1| + |a2| + ... + |an|`.

**terminatePrograms**

`terminatePrograms(0)` terminates all running AlgoSim programs.

**time**

`time(0)` returns the current time as a structure. The members are hour, minute, second, and millisecond.

**toBaseN**

`toBaseN(x, N)` returns a string with the base- $N$  representation of the non-negative integer  $x$ .

Example: `toBaseN(255, 16) = "FF"`

**toCamelCase**

`toCamelCase(str)` returns the string `str` where every character is upper-case if and only if it is the first character of a word.

Example:

`toCamelCase("this is a brief text. a very brief text, actually.") = "This Is A Brief Text. A Very Brief Text, Actually."`

**toEchelonForm**

`toEchelonForm(M)` applies elementary row operations to the matrix  $M$  (i.e. premultiplies it with elementary matrices) until it obtains echelon form.

**toFraction**

`toFraction` is used to obtain the numerator  $p$  and denominator  $q$  of a rational number (or approximated real number)  $x = p/q$  (or at least very close to).

`toFraction(x)` returns the real number  $x$  as a string of the form " $p/q$ " where  $p$  and  $q$  are integers, and so that the rational number  $p/q \approx x$ .

For instance, `toFraction(0.0843373493976) = "7/83"`.

**toLowerCase**

`toLowerCase(s)` returns the string `s` with all letters converted to lower case.

**toRealNumber**

`toRealNumber(s)` returns the real number represented by the `s`, if possible.

Example: `toRealNumber("1024") = 1024`

**toSentenceCase**

`toSentenceCase(str)` returns the string `str` where every character is upper-case if and only if it is the first character of a sentence.

Example:

`toSentenceCase("this is a brief text. a very brief text, actually.") = "This is a brief text. A very brief text, actually."`

**toString**

`toString(x)` returns the real or complex number  $x$  as a string.

Example: `toString(1024) = "1024"`

**toSymbolicForm**

`toSymbolicForm(x)` returns the real number  $x$  in symbolic (exact) form, if possible. The function returns a string with an expression using division, multiplication, square roots, and constants (such as  $e$  and  $\pi$ ), evaluating to  $x$ , if such an expression can be found.

Example: `toSymbolicForm(0.866025403784439) = " $\sqrt{3}/2$ "`

**totient**

`totient(n)` is Euler's totient, or  $\phi$  function, i.e. the number of positive integers less than or equal to  $n$  that are coprime to  $n$ .

**toUpperCase**

`toUpperCase(s)` returns the string `s` with all letters converted to upper case.

**tr**

`tr(M)` returns the trace of the real or complex matrix  $M$ .

**transpose**

`transpose(M)` returns the transpose of the real or complex matrix  $M$ .

**trim**

`trim(s)` returns the string `s` with all leading and ending whitespace characters (e.g. spaces) removed. Thus, `trim(s) = trimLeft(trimRight(s))` for all strings `s`.

**trimLeft**

`trimLeft(s)` returns the string `s` with all leading whitespace characters (e.g. spaces) removed.

## **trimRight**

trimRight(s) returns the string s with all ending whitespace characters (e.g. spaces) removed.

## **trunc**

trunc(x) returns x rounded towards 0, i.e. replaces all decimals after the decimal point with zeroes.

## **txtBeginsWith**

txtBeginsWith(S, s) returns True if the text (string) S begins with s, and false otherwise. No distinction is made between capital and small letters.

## **txtContains**

txtContains(S, s) returns True if the text (string) S contains s, and False otherwise. No distinction is made between capital and small letters.

## **txtEndsWith**

txtEndsWith(S, s) returns True if the text (string) S ends with s, and false otherwise. No distinction is made between capital and small letters.

## **txtPos**

txtPos(s, S) returns the position of the first character in the text (string) s of the first occurrence of s in the text (string) S, making no difference between capital and small letters.

Example: txtPos("st", "teSTtest") = 3, but strPos("st", "teSTtest") = 7.

## **txtReplaceAll**

txtReplaceAll(S, x, y) replaces all occurrences of x by y in the text (string) S. When locating x in S, no distinction is made between capital and small letters.

## **type**

type(x) returns a string with the [name of the] data-type of the object x, i.e. "real number", "complex number", "real vector", "complex vector", "real matrix", "complex matrix", "string", "boolean", "pixmap", "sound", "table", or "set".

## **undo**

undo(0) removes the most recently added item to the current 2D visualization window.

## **undo3**

undo3(0) removes the most recently added item to the current 3D visualization window.

## **unloadDictionary**

unloadDictionary(0) unloads the currently loaded (see loadDictionary) dictionary, thus freeing system memory (RAM).

## **URLEncode**

URLEncode(str) returns the string str properly URL encoded.

Example: URLEncode("1+1=2") = "1%2B1%3D2"

## **wait**

wait(t) suspends execution of the program for t seconds, but AlgoSim will remain responsive during this time, in contrast to sleep(t).

## **warning**

warning(s) displays s as a warning message. s is a string.

## **vectToMat**

vectToMat(v) returns a n×1 matrix with entries from the n-dimensional vector v.

## **vectToSet**

vectToSet(v) returns the set of all components in the n-dimensional vector v.

## **week**

week(0) returns the current week's number of the year, as a integer.

## **weeksBetween**

weeksBetween(d1, d2) returns the number of weeks between the date structures d1 and d2.

Example:

d1 := encodeDate(2010, 06, 19)

year: 2010  
month: 6  
day: 19  
weekOfYear: 24  
dayOfYear: 170  
dayOfWeek: 6

d2 := date(0)

year: 2010  
month: 6  
day: 23  
weekOfYear: 25  
dayOfYear: 174  
dayOfWeek: 3

weeksBetween(d1, d2)

0.571428571429

## **ver**

ver(0) returns the version of AlgoSim as a structure. The structure contains the members major, minor, release, build, and asString.

## **VignèreDecrypt**

VignèreDecrypt(str, key) decrypts the string str using the Vignère cipher and the password key.

## **VignèreEncrypt**

VignèreEncrypt(str, key) encrypts the string str using the Vignère cipher and the password key.

## **wisdom**

wisdom(0) returns a random wisdom (as a string).

## **yearsBetween**

yearsBetween(d1, d2) returns the number of years between the date structures d1 and d2.

Example:

d1 := encodeDate(2010, 06, 19)

year: 2010  
month: 6  
day: 19  
weekOfYear: 24  
dayOfYear: 170  
dayOfWeek: 6

d2 := date(0)

year: 2010  
month: 6  
day: 23  
weekOfYear: 25  
dayOfYear: 174

dayOfWeek: 3

yearsBetween(d1, d2)

0.0109514031485

**zeroMatrix**

zeroMatrix(m, n) returns the  $m \times n$  matrix with all zero entries.

**zeroVector**

zeroVector(n) returns the n-dimensional zero vector.

## Appendix II: Pre-Defined User-Customisable Functions

The following functions are implemented the same way the end-user can implement functions, i.e. by using the

$$\text{FuncName} = \text{"vars"} \mapsto \text{"expr"}$$

syntax. They are automatically loaded when AlgoSim is loaded, for they are defined in startup.prg (in the *common* program directory) that executes every time AlgoSim starts.

startup.prg

---

```

inv = "x" ↦ "x^(-1)"
isPerfect = "n" ↦ "sum(divisors(n)) = 2·n"
isAlmostPerfect = "n" ↦ "sum(divisors(n)) = 2·n - 1"
isSuperPerfect = "n" ↦ "sum(divisors(sum(divisors(n)))) = 2·n"
areAmicable = "m, n" ↦ "(sum(divisors(m)) - m = n) ∧
    (sum(divisors(n)) - n = m) ∧ (m≠n)"
isDeficient = "n" ↦ "sum(divisors(n)) < 2·n"
isAbundant = "n" ↦ "sum(divisors(n)) > 2·n"
abundance = "n" ↦ "sum(divisors(n)) - 2·n"
isSublime = "n" ↦ "isPerfect(dim(divisors(n))) ∧
    isPerfect(sum(divisors(n)))"
isSquareFree = "n" ↦ "¬containsDuplicate(factors(n))"
isNormal = "A" ↦ "A·A* = A*·A"
isSymmetric = "A" ↦ "transpose(A) = A"
isSkewSymmetric = "A" ↦ "transpose(A) = -A"
isHermitian = "A" ↦ "A* = A"
isOrthogonal = "A" ↦ "transpose(A) = A^(-1)"
isUnitary = "A" ↦ "A* = A^(-1)"
defView = "x" ↦ "drawAxes(clearView(setView(-10, 10, -10, 10),
    setAxisStyle('x, y', '')))"
defView3 = "x" ↦ "drawAxes3(clearView3(setView3(-10, 10, -10,
    10, -10, 10)))"
today = "x" ↦ "date(0)"
tomorrow = "x" ↦ "addDays(date(0), 1)"
yesterday = "x" ↦ "addDays(date(0), -1)"
fork = "x" ↦ "start(getParameter('path'))"
wolframAlpha = "query" ↦
    "start('http://www.wolframalpha.com/input/?i=' +
    URLEncode(query))"
OPNOTIN = "A, B" ↦ "¬(A ∈ B)"
numberOfPrimes = "n" ↦ "count([1, n], 'x', 'isPrime(x)')"

```

---

## Appendix III: Example Programs

In AlgoSim, a few example programs are included. (They reside in the *common* AS program directory.) Below is a brief description of the most interesting of these. Although the standard way of starting a program is to call it from the console (such as `billiard(0)` or `mirrorSim(t := "parabolic")`), the simplest way is to click the Programs button in the left-most column of buttons, and choose program in the Run a Program submenu.

- **billiard.prg**

Simulates a 2D billiard dynamical system, i.e. a system in which a particle bounces elastically in a rectangular box with constant potential (that is, no forces other than at the walls). The *trace* of the particle is displayed.

- **billiardAnim.prg**

Same as `billiard.prg`, but now the particle is animated in the box, and no trace is shown.

- **butterfly.prg**

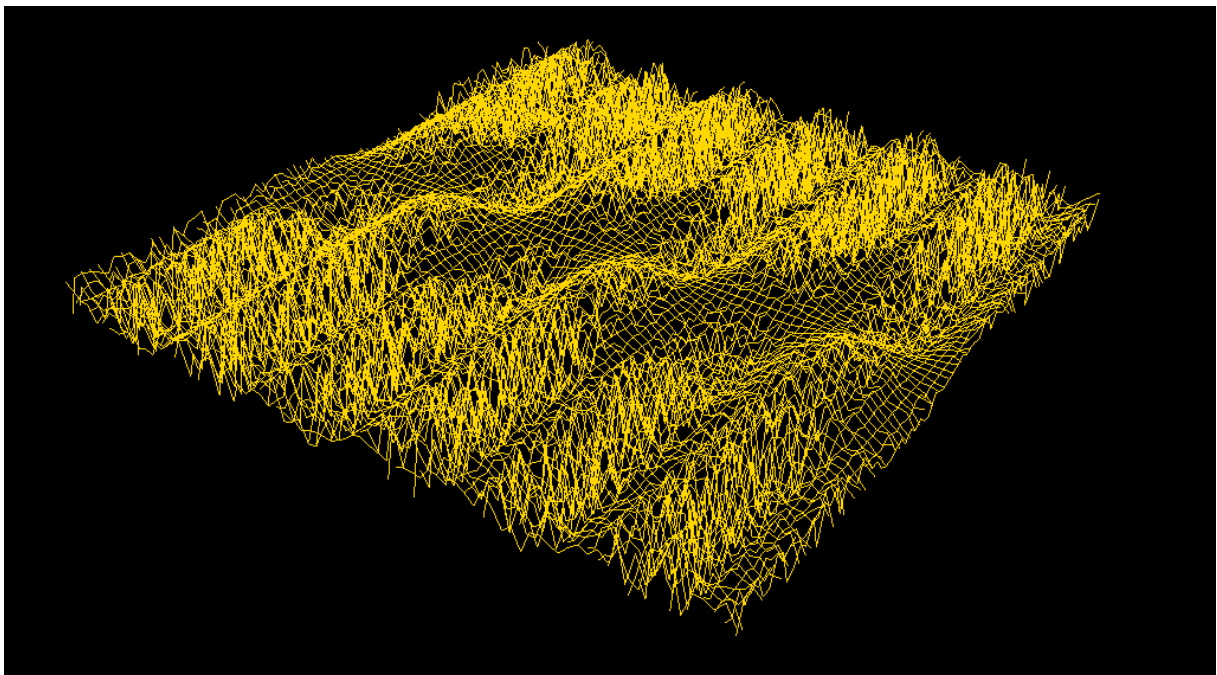
Displays the polar butterfly curve.

- **bz.prg**

Simulates a “BZ-like” flow of an oscillating chemical reaction.

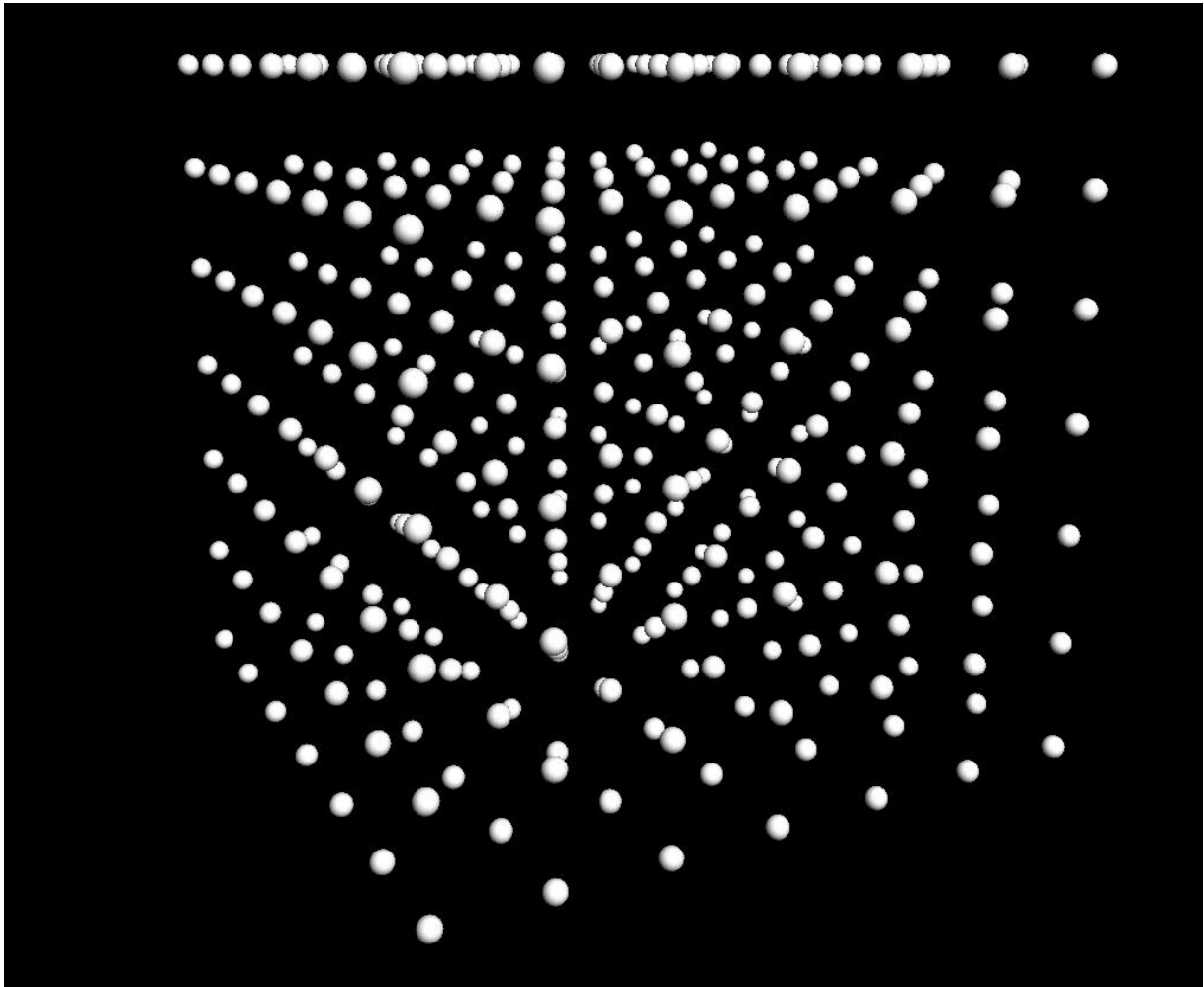
- **garaden.prg**

Renders a 3D garden.



- **gitter.prg**

Renders a simple cubic (s.c.) 3D lattice of atoms.



- **helicoid.prg, cone.prg, helix.prg, hyperboloid.prg, torus.prg, tori.prg**

Draws this spatial curve or surface.

- **mirrorSim.prg**

Simulates reflection in a 2D mirror. Call it with argument t equal to "circular", "parabolic", "sine", "line", or "convex parabolic", as in mirrorSim(t := "parabolic").

- **mirrorSim3.prg**

Simulates reflection in a 3D mirror. Call it with argument t equal to "spherical" or "parabolic", as in mirrorSim3(t := "parabolic").

- **Möbius.prg**

Draws a Möbius strip.

- **orthogonalProjection.prg**

Draws a few spheres and their orthogonal projection in the xy-plane.

- **randomWalk.prg**

Simulates 2D four-direction, discrete-step, random walk.

- **rutherfordScattering.prg**

Simulates Rutherford scattering of an  $\alpha$ -particle towards a gold nucleus with a given impact parameter. The program will, upon execution, ask about the impact parameter.

- **rutherfordScattering2.prg**

Simulates Rutherford scattering of an  $\alpha$ -particle towards a gold nucleus at a number of different impact parameters.

- **sampling.prg**

Plays Händel's *Messiah* sampled at 44.1 kHz [CD quality], 11.0 kHz, and 5.5 kHz. Because humans can hear up to 20 kHz, human music must be sampled at no less than 40 kHz if not aliasing is to appear.

```
sampling
Now we are playing the music at 44.1 kHz.
Now we are playing the music at 11.0 kHz.
Now we are playing the music at 5.5 kHz.
|
```

- **superposition.prg**

Simulates superposition of two circular water waves, and displays the result as a 3D graph.

- **superpositionPlane.prg**

Simulates superposition of two circular water waves, and displays the result as a coloured plane.

- **waveSim.prg**

Simulates superposition of two sine waves with different (user-specified) parameters (frequency, wavelength, amplitude, initial phase).

## Appendix IV: Default Operator Table

postfix	°	OPDEGREES	0	0	0
postfix	%	OPPERCENT	0	0	0
postfix	‰	OPPERMILLE	0	0	0
postfix	!	OPFACT	0	0	0
infix	#	baseNInput	1	0	0
infix		OPINDEX	0	0	0
infix	^	OPPOWER	0	0	1
prefix	-	OPMINUS	0	0	0
infix	↑	OPEXP	0	0	0
prefix	√	sqrt	0	0	0
prefix	¬	OPNOT	0	0	0
postfix	*	OPASTERISK	0	0	0
infix	/	OPDIV	0	0	0
infix	◦	OPCOMPOSITE	0	0	0
infix	×	OPCROSSMUL	0	0	0
infix	·	OPMUL	0	0	0
infix	-	OPSUB	0	0	0
infix	+	OPADD	0	0	0
infix		OPBAR	0	0	0
circumfix	[ ]	ceil	0	0	0
circumfix	[ ]	floor	0	0	0
circumfix	( )	OPVECT	0	0	0
circumfix	{ }	OPSET	0	0	0
circumfix	[ ]	OPINTERVAL	0	0	0
prefix	~	OPCOMPLEMENT	0	0	0
infix	∪	OPUNION	0	0	0
infix	∩	OPINTERSECT	0	0	0
infix	\	OPSETMINUS	0	0	0
infix	=	OPEQUALS	0	0	0
infix	≈	OPAPPROX	0	0	0
infix	≠	OPNOTEQUAL	0	0	0
infix	<	OPLESSTHAN	0	0	0
infix	>	OPGREATERTHAN	0	0	0
infix	≤	OPLESSOREQUAL	0	0	0
infix	≥	OPGREATEROREQUAL	0	0	0
infix	∧	OPAND	0	0	0
infix	∨	OPOR	0	0	0
infix	¬	OPNAND	0	0	0
infix	√	OPNOR	0	0	0
infix	∨	OPXOR	0	0	0
infix	∥	OPPARALLEL	0	0	0
infix	⊥	OPNOTPARALLEL	0	0	0
infix	⊥	OPORTHOGONAL	0	0	0
infix	∈	OPIN	0	0	0
infix	∃	OPNI	0	0	0
infix	∉	OPNOTIN	0	0	0
infix	∄	OPNOTNI	0	0	0
infix	⊂	OPSUBSET	0	0	0
infix	⊃	OPSUPERSET	0	0	0
infix	⊆	OPPROPSUBSET	0	0	0
infix	⊇	OPPROPSUPERSET	0	0	0
infix	→	OPIMPLIES	0	0	0
infix	←	OPIMPLIESLEFT	0	0	0
infix	↔	OPEQUIVALENT	0	0	0
infix	↦	OPMAPSTO	0	0	0
infix	≐	OPASSIGN	1	0	1
infix	≐	OPNGISSA	0	1	0



---

<code>infix</code>	<code>;</code>	<code>identity</code>	<code>0</code>	<code>0</code>	<code>0</code>
--------------------	----------------	-----------------------	----------------	----------------	----------------

---

## Appendix V: Default Table of Constants

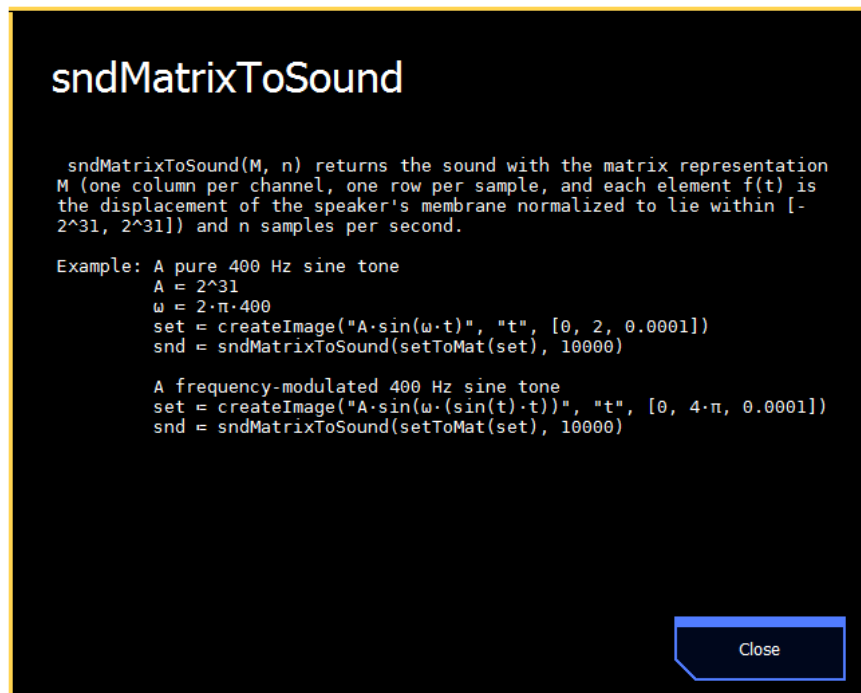
speed of light	299792458
elementary charge	$1.602176487 \cdot 10^{-19}$
electron mass	$9.10938215 \cdot 10^{-31}$
proton mass	$1.672621637 \cdot 10^{-27}$
neutron mass	$1.67492729 \cdot 10^{-27}$
atomic mass unit	$1.660538782 \cdot 10^{-27}$
electric constant	$8.854187817 \cdot 10^{-12}$
magnetic constant	$4 \cdot \pi \cdot 10^{-7}$
Coulomb's constant	$8.987551787 \cdot 10^9$
Avogadro constant	$6.02214179 \cdot 10^{23}$
Faraday constant	96485.3383
gas constant	8.314472
Boltzmann constant	$1.3806504 \cdot 10^{-23}$
Wien's displacement constant	$2.8977685 \cdot 10^{-3}$
Stefan-Boltzmann constant	$5.6704 \cdot 10^{-8}$
Planck's constant	$6.62606896 \cdot 10^{-34}$
Planck's constant over $2\pi$	$1.054571628 \cdot 10^{-34}$
gravitational constant	$6.6726 \cdot 10^{-11}$
acceleration of gravity	9.81
bohr radius	$5.29177208 \cdot 10^{-11}$
ground-state energy of hydrogen atom	13.605692
Rydberg constant	$1.0973731569 \cdot 10^7$
bohr magneton	$9.274009 \cdot 10^{-24}$
nuclear magneton	$5.0507832 \cdot 10^{-27}$
fine-structure constant	0.007297352533
solar radius	$6.96 \cdot 10^8$
solar mass	$1.9891 \cdot 10^{30}$
solar surface temperature	5778
earth's radius	$6.371 \cdot 10^6$
earth's mass	$5.9736 \cdot 10^{24}$
astronomical unit	$1.4959787 \cdot 10^{11}$
light-year	$9.46055 \cdot 10^{15}$
beard-second	$10^{-8}$
Ångström	$10^{-10}$
parsec	$3.0857 \cdot 10^{16}$
age of universe	$1.375 \cdot 10^{10}$
Hubble constant	74.2
standard atmosphere	101325
Feigenbaum constant	4.66920160910299067185320382
Napier's constant	$\exp(1)$
golden ratio	$(1 + \sqrt{5})/2$
Euler-Mascheroni constant	0.57721566490153286060651209
pi	$2 \cdot \arccos(0)$
imaginary unit	$\sqrt{-1}$
googol	$10^{100}$

## Appendix VI: Online Help

Within AlgoSim, you can search for identifiers (functions and variables) by pressing the Tab key in the console. By entering characters you can filter the list of identifiers.



In addition, when the caret is inside an identifier in the console, or when an identifier has been selected in the *Identifiers* dialog (see above), you can press F1 to show the reference associated with the identifier.



## Appendix VII: A Few Tips & Tricks

- **Numbers and Strings**

Of course you can add two numbers ( $5 + 3$ ) and two strings ("test" + " again") together, but you can also add a string to a number; then AlgoSim will automatically convert the number to a string, and add these strings. For example, "test" + 5 yields the string "test5".

- **Functions That Require no Arguments**

Functions that require no arguments will ignore all arguments sent to them. This can be highly useful. As an example, say that you want to examine the three polar graphs  $r = \sin(\varphi)$ ,  $r = \cos(\varphi)$ , and  $r = \tan(\varphi)$ . Then you can write

```
drawSet("set")
redraw(set = polarCoords(createImage("(sin(φ), φ)", "φ", [0, π/2,
0.01])))
redraw(set = polarCoords(createImage("(cos(φ), φ)", "φ", [0, π/2,
0.01])))
redraw(set = polarCoords(createImage("(tan(φ), φ)", "φ", [0, π/2,
0.01])))
```

- **Mod Operator**

You might want to redefine % to map to **mod**, so that it will work as in C/C++.

- **Common Constants**

If you use some constants very often, you might want to add them to startup.prg. For instance,

```
m = constant("electron mass")
qe = constant("elementary charge")
```

- **Bitwise Logic**

When used with unsigned 32-bit integer operands, the boolean operators  $\wedge$ ,  $\vee$ ,  $\underline{\vee}$ , etc. will act as bitwise and, or, xor, etc. For instance,  $1001010101010010\#2 \vee 0100110010101011\#2$  will return 56827, and  $\text{toBaseN}(56827, 2) = 1101110111111011$ .

- **Integer Booleans**

In many programming languages, booleans are nothing but integers. Most often, 0 represents **false**, and a non-zero integer, particularly 1 (sometimes -1 due to its most common binary representation) represents **true**. You can use integer booleans in AlgoSim as well by using trivial mappings between **{true, false}** and  $\mathbb{Z}$ . To convert a boolean to 0 or 1, use the Iverson bracket: **[true]** = 1 and **[false]** = 0. To convert an integer to a boolean, simply test  $\text{val} \neq 0$ .

- **When Something Appears to Be Wrong**

AlgoSim lets the end-user manipulate the system in great detail: the user can even redefine or remove common arithmetical operators and constants such as “+”, “-”, “ $\pi$ ”, and “e”! If you suspect that there is something wrong with the current AlgoSim session, you can enter the command **identifyProblems**. This function will make AlgoSim try automatically to identify potential problems in the current session, both technical issues, and problems caused by the user. Some issues may even be corrected automatically.

Copyright © 2010 Andreas Rejbrand

[www.english.rejbrand.se](http://www.english.rejbrand.se)