

RX-CACSD

Library for supporting automatic control in the Open Object Rexx language

User Manual

Version 1.0.0-pre

(generated January 7, 2026)

Michal Moučka



Contents

1 INTRODUCTION	5
1.1 Overview	5
1.2 Disclaimers	6
2 INSTALLATION	6
3 GETTING STARTED	7
3.1 Library usage	7
3.2 How to read syntax diagrams	10
4 BUILTIN CLASSES	11
4.1 DER Class	11
4.1.1 new (class method)	11
4.1.2 sim	12
4.2 DZN Class	12
4.2.1 new (class method)	12
4.2.2 set_limits	12
4.2.3 get_limits	13
4.2.4 sim	13
4.3 INT Class	13
4.3.1 new (class method)	13
4.3.2 reset	13
4.3.3 set_condition	14
4.3.4 set_limits	14
4.3.5 get_condition	14
4.3.6 get_limits	15
4.3.7 is_saturated	15
4.3.8 sim	15
4.4 LAG Class	16
4.4.1 new (class method)	16
4.4.2 set_lag	17
4.4.3 get_lag	17
4.4.4 sim	17
4.5 ODE Class	17
4.5.1 new (class method)	18
4.5.2 rk4	19
4.6 OPT Class	20
4.6.1 new (class method)	20
4.6.2 fmins	21
4.7 PID Class	23
4.7.1 new (class method)	23
4.7.2 set_limits	23
4.7.3 set_parameters	24
4.7.4 set_conf	24
4.7.5 get_conf	24
4.7.6 get_limits	24
4.7.7 get_parameters	25
4.7.8 is_saturated	25
4.7.9 sim	25
4.8 POL Class	28

4.8.1 new (class method)	28
4.8.2 [] (indexing)	29
4.8.3 [] = (assignment)	29
4.8.4 sum	29
4.8.5 sub	30
4.8.6 mul	30
4.8.7 div	30
4.8.8 eval	30
4.8.9 roots	31
4.8.10 equal	31
4.8.11 der	31
4.8.12 int	32
4.9 PFS Class	32
4.9.1 to_array	32
4.10 REL Class	32
4.10.1 new (class method)	32
4.10.2 set_switch_points	33
4.10.3 set_outputlevels	33
4.10.4 get_switch_points	33
4.10.5 get_output_levels	34
4.10.6 sim	34
4.11 SAT Class	34
4.11.1 new (class method)	34
4.11.2 set_limits	35
4.11.3 get_limits	35
4.11.4 is_saturated	35
4.11.5 sim	35
4.12 TFN Class	36
4.12.1 new (class method)	36
4.12.2 set_lag	36
4.12.3 set_numden	37
4.12.4 set_zpk	37
4.12.5 get_den	37
4.12.6 get_gain	37
4.12.7 get_lag	38
4.12.8 get_num	38
4.12.9 get_poles	38
4.12.10 get_zeros	38
4.12.11 hurwitz	39
4.12.12 residue	40
4.12.13 sim	40
4.12.14 step	41
4.12.15 nyquist	42
4.12.16 bode	43
4.12.17 feedback	44
4.12.18 negate	44
4.12.19 parallel	45
4.12.20 series	45
4.12.21 simplify	45
4.13 MAT Class	45
4.13.1 new (class method)	46

4.13.2 [] (indexing - get)	47
4.13.3 []= (indexing - set)	47
4.13.4 dim	48
4.13.5 rand	48
4.13.6 reinit	48
4.13.7 to_array	49
4.13.8 sum	49
4.13.9 had	49
4.13.10 mul	50
4.13.11 trans	50
4.13.12 det	50
4.13.13 inv	51
4.13.14 rank	51
4.14 GRP	51
4.14.1 new (class method)	51
4.14.2 bode	52
4.14.3 nyquist	52
4.14.4 plot	52
4.14.5 pzmap	54
5 BUILTIN ROUTINES	55
5.1 Array routines	55
5.1.1 linspace	55
5.1.2 logspace	55
5.2 Helper routines	56
5.2.1 beep	56
5.2.2 sleep	56
5.3 Mathematical routines	56
5.3.1 sqrt	56
5.3.2 exp	57
5.3.3 log	57
5.3.4 log10	57
5.3.5 sinh	57
5.3.6 cosh	57
5.3.7 tanh	58
5.3.8 power	58
5.3.9 sin	58
5.3.10 cos	58
5.3.11 tan	58
5.3.12 cotan	59
5.3.13 pi	59
5.3.14 arcsin	59
5.3.15 arccos	59
5.3.16 arctan	60
5.3.17 maxn	60
5.3.18 minn	60
5.3.19 sum	60
5.3.20 prod	61
5.3.21 mean	61
5.3.22 med	61
5.3.23 mode	61
5.3.24 var	62

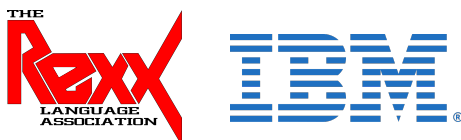
5.4 Routines for global options	62
5.4.1 set_ts	62

1 INTRODUCTION

The RX-CACSD (Computer-Aided Control System Design) library is a software tool for the design, analysis, and implementation of automatic control systems in the Open Object Rexx (ooRexx) programming environment. Its primary goal is to make control theory methods accessible to ooRexx users and to extend the languages traditional focus on scripting and automation toward technical and engineering applications.



Open Object REXX <https://www.oorexx.org/index.rsp> (ooRexx) is an object-oriented implementation of the REXX language, originally developed by IBM <https://www.ibm.com> as Object REXX for the OS/2 operating system <https://www.os2world.com>. Later, it was released as an open-source project under the name Open Object REXX (ooRexx) and is currently maintained by the Rexx Language Association (RexxLA) <https://www.rexxla.org>. As a modern enhanced version complementing the OS/2 system, there is ArcaOS from ArcaNoae <https://www.arcanoe.com>.



1.1 Overview

The RX-CACSD library provides comprehensive support for dynamic system modeling, controller design, control loop simulation, and stability analysis. It allows users to work with transfer functions, block diagrams, and standard stability criteria directly within the ooRexx environment.

The library includes implementations of commonly used linear controllers, most notably PID controllers, with flexible parameter tuning options. It also offers tools for time-domain simulation, stability evaluation, and basic controller optimization.

At the time of its introduction, no comparable library existed for ooRexx that provided integrated support for automatic control system design. While environments such as MATLAB (Control System Toolbox) and Python (python-control) offer similar capabilities, RX-CACSD brings these functionalities directly into ooRexx.

The library is portable across all platforms supported by ooRexx, including ArcaOS, OS/2 Warp 4, and FreeBSD, and integrates seamlessly with other ooRexx-based applications. At the time of publication, no other library of comparable scope existed for the ooRexx language that offered comprehensive support for the design, simulation, and tuning of automatic control systems. The CACSD library is, therefore, unique. It is the first to bring tools for modeling and controlling dynamic systems directly into the ooRexx environment, significantly extending its capabilities toward technical and engineering applications.



FreeBSD

The library has proven useful both in real-time classical systems and in modern desktop and scripting environments.

The RX-CACSD library provides a set of object classes that enable intuitive and clear expression of control

structures and mathematical operations.

Key classes include:

Transfer Function class (TFN), PID Controller class (PID), Matrix class (MAT), Polynomial class (POL), Transport delay class (LAG), Relay class (REL), Dead zone class (DZN), Saturation class (SAT), Optimization class (OPT).

1.2 Disclaimers

State any warranties and disclaimers (if applicable). This library, inclusive of examples and documentation, is provided "as is" without any warranties, express or implied. The author assumes no responsibility for any damages that may arise from the use of this library, particularly in control system applications where the incorrect operation may result in property damage, personal injury, or other consequences.

It is imperative to note that the library's use is undertaken at the user's own discretion and risk. It is presumed that the user possesses sufficient knowledge of automatic control and is capable of evaluating the appropriateness of its application in a particular context, including validating simulation outcomes.

The material has been developed for the purposes of academic study and experimental research. If one lacks expertise in the design and implementation of control systems, it is advisable to consult a specialist in the field.

2 INSTALLATION

A functional installation of Open Object Rexx version 5.0 or higher is required to use the RX-CACSD library. The interpreter can be downloaded from:
<https://www.oorexx.org/downloads.rsp>.

An exception is Open Object REXX for OS/2, which is not available from the official website but is included in the RX-CACSD library's installation package. The download link is provided further below, within this chapter.

The library does not provide native support for plotting. Instead, it utilizes Gnuplot as an external tool to generate graphical output. If graphical visualization is required, Gnuplot must be installed separately. Gnuplot is freely available for multiple platforms and can be downloaded from:
<http://www.gnuplot.info/download.html>

The CACSD library requires no complex installation procedure. It is freely available for download at:
<https://rexx.ksa.tul.cz/rx-cacsd>

This package includes the library itself, located in the lib directory, example programs in the smp directory, and the documentation you are currently reading in the doc directory.

Currently, precompiled versions of the library are available for FreeBSD, Windows, and OS/2. For other platforms, the library must be compiled from source code. Compilation can be performed using commonly available compilers such as GCC. However, the source code is not publicly available at this time because parts of the C/C++ implementation are used in the author's ongoing research activities and cannot yet be released. If you require the library for other platforms, please do not hesitate to contact me.

Once compiled or installed, the rx-cacsd library should be placed in the appropriate location where ooRexx expects to find it. Installation differs depending on the operating system:

- **ArcaOS / OS/2 (32-bit):**

To be added.

- **FreeBSD (64-bit):**

To be added.

- **Windows (64-bit):**

The library is provided as a ZIP file in this time. Extract the contents and copy all files into the directory where ooRexx is installed (e.g. C:\Program Files\ooRexx), preserving the directory structure contained in the package.

Contents of the installation package (applies to all operating systems):

- The dynamic library (DLL on Windows and OS/2, shared object on FreeBSD)
- Usage examples
- This manual in PDF format

After installation, will be able to locate and use the library automatically.

To load the library in your program, add the directive `::requires 'rxcacsd.cls'` to your program source code.

3 GETTING STARTED

The library's design aims to prioritize intuitive functionality. The utilization of individual functions does not necessitate specialized programming expertise beyond the conventional understanding of the language. The manual contains a wealth of information, including instructions for use, which are elucidated through practical examples. The format of functions and methods is illustrated using syntactic diagrams.

On the other hand, the user is assumed to have basic knowledge of automatic control, such as working with transfer functions, system stability, or feedback control. This knowledge is important for a proper understanding of the meaning of individual library functions and for their effective use in practice.

Theoretical explanations relevant to specific methods and classes are provided where necessary. For instance, the PID class includes a concise description of a PID controller, its fundamental equation, and its operating principle. For a more comprehensive understanding of the theory, readers are encouraged to consult specialized literature and scientific articles. Additionally, many concepts can be grasped from the examples included in this manual.

3.1 Library usage

The following simple example illustrates how to use the library in practice. The program demonstrates a simulation of PI control of a second-order proportional linear dynamic system.

The system is described by a linear differential equation

$$y''(t) + 2y'(t) + y(t) = 0.5u(t). \quad (1)$$

The PI controller is given by

$$u(t) = 2.5 \left(e(t) + \frac{1}{1.5} \int_0^t e(\tau) d\tau \right). \quad (2)$$

The controller output is required to be saturated between -10 and 10, with a sampling period of 0.01 seconds.

In control engineering, time-domain equations are typically not used directly; instead, they are transformed into transfer functions using the Laplace transform. This is also the approach adopted by the RXCACSD library.

The corresponding transfer function of the system can be expressed as:

$$G(s) = \frac{Y(s)}{U(s)} = \frac{0.5}{s^2 + 2s + 1}. \quad (3)$$

The transfer function of the controller is

$$R(s) = \frac{U(s)}{E(s)} = 2.5 \left(1 + \frac{1}{1.5s} \right). \quad (4)$$

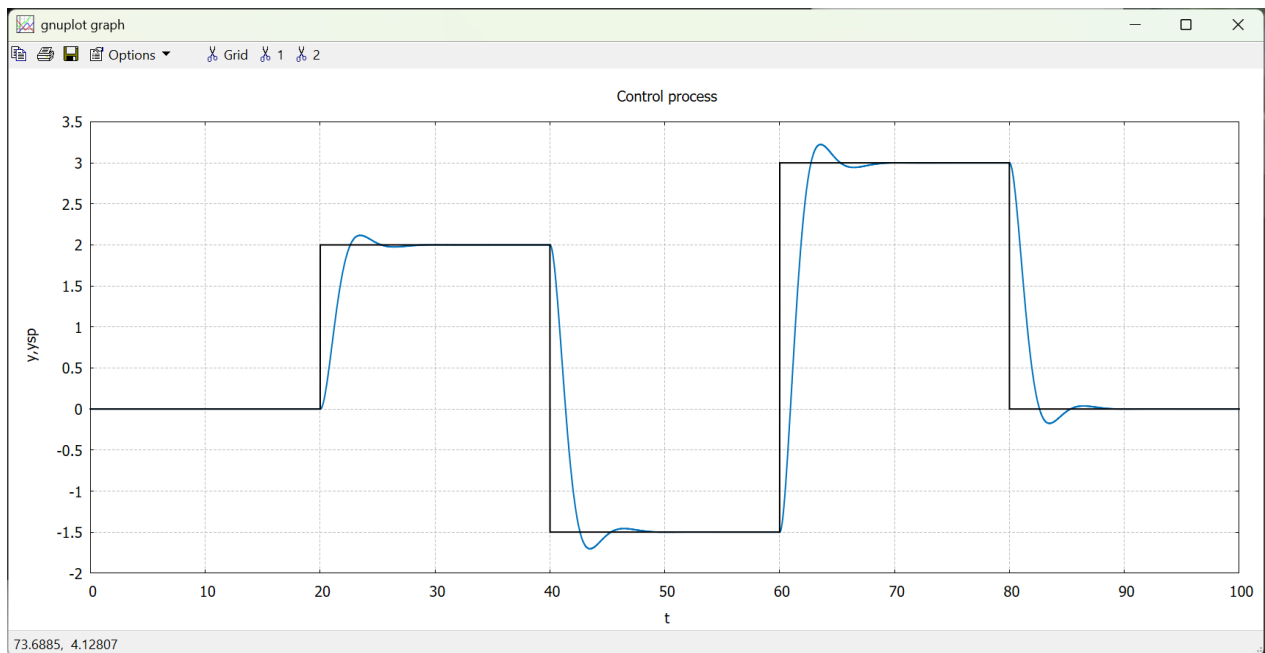
Below is a simple program listing including explanatory comments. The program demonstrates control to the setpoint values 0, 2, -1.5, 3, 0 at time 0, 20, 40, 60, 80 s.

Example: Interaction of a PI Controller with a System

```

1  cls          -- clear screen (system dependent command)
2
3  /* Initialisation */
4  ts = 0.01    -- sample period
5  t_sim = 100  -- length of the simulation
6  rc = set_ts(ts) -- set global step
7
8  G = .tf~new(.array~of(0.5),.array~of(1,2,1)) -- define transfer function of the system
9  say G -- show transfer function on the screen
10
11 R = .pid~new(2.5,1.5,0,.array~of(-10,10),'PID') -- define PI controller
12 say R -- show PI controller on the screen
13
14 ysp = .array~of(.array~of(0, 2, -1.5, 3, 0),.array~of(0,20,40,60,80))
15
16 a_t = .array~new -- arrays for plot
17 a_y = .array~new
18 a_ysp = .array~new
19
20 u = 0 -- initial value of manipulated variable
21
22 t = 0 -- time
23 itx = 1 --time index
24
25 /* control loop */
26 loop t_sim/ts
27
28   if t > ysp[2][itx+1] then -- ysp change
29     itx += 1
30
31   y = G~sim(u) -- system response to manipulated variable
32   u = R~sim(ysp[1][itx],y) -- manipulated variable from controller
33
34   a_t~append(t) -- for plot
35   a_y~append(y) -- for plot
36   a_ysp~append(ysp[1][itx]) -- for plot
37
38   say format(y,3,6) format(u,3,6) format(ysp[1][itx]-y,3,6) -- format output to the screen
39
40   t += 0.01 -- change of time
41
42 end
43
44 /* Visualisation */
45 grp = .grp~new('Control process','t','y,ysp')
46 grp~plot(.array~of(a_t,a_y),'-b',.array~of(a_t,a_ysp),'-k')
47
48 ::requires 'rxcacsd.cls' -- necessary RXCACSD library

```



3.2 How to read syntax diagrams

In this manual, the syntax is consistently described through syntactic diagrams, which is a standard approach commonly used in documentation for the REXX programming language.

a) Read the syntactic diagrams from left to right and from top to bottom, following the direction of the arrows.

Symbol ►► indicates the beginning of a command.

Symbol -... indicates that the command syntax continues on the next line.

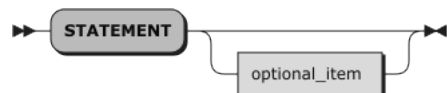
Symbol ...► indicates that the command continues from the previous line.

Symbol ►► indicates the end of a command.

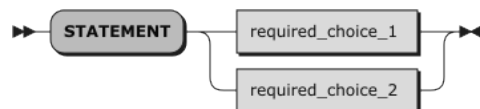
b) Mandatory elements are located on the horizontal line (the main path).



c) Optional elements are located below the main path.

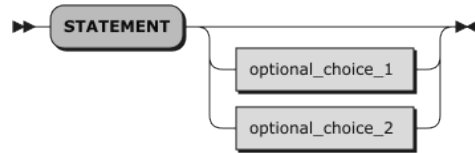


d) If you can choose from two or more elements, they are displayed vertically as a column. If exactly one of these elements must be selected, one of them is placed on the main path.

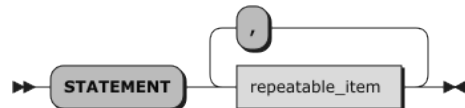


e) If it is optional to select one of the elements, the entire column is located below the main path. If one of

the elements is the default choice, it is usually listed first (at the top) in the column of elements below the main path.



g) A path looping back to the left above the main line indicates an element that can be repeated. A repetition path above a column indicates that the elements in the column can be repeated.



h) A sharp (pointed) rectangle around an element indicates a fragment, that is, a part of the syntactic diagram which is detailed further below the main diagram.



i) Keywords are displayed in UPPERCASE letters (for example, SIGNAL). They must be written exactly as shown, but you can type them in uppercase, lowercase, or a combination of both. Variables are shown in lowercase letters (for example, index) and represent user-supplied names or values.

j) If punctuation marks, parentheses, arithmetic operators, or similar symbols are displayed, you must enter them as part of the syntax.

In this documentation, variable names are composed of two parts. The first part is a prefix that indicates the data type, and the second part is a descriptive name. The prefix and the name are separated by an underscore (_). For example: `num_setpoint`, `mat_coefficients`, `tfm_process`. The prefix `num_` denotes a scalar number, `mat_` denotes an instance of the MAT class (matrix), and `tfm_` denotes an instance of the TFN class (transfer function).

4 BUILTIN CLASSES

4.1 DER Class

Signal differentiator.

4.1.1 new (class method)

A new instance of the DER class.

Syntax:



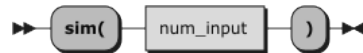
Method has no parameters.

Returns a new instance of the DER class.

4.1.2 **sim**

Performs one step of the derivative calculation.

Syntax:



The input parameter is the signal value `num_input` to be differentiated.

Returns the derivative of the input signal `num_input`.

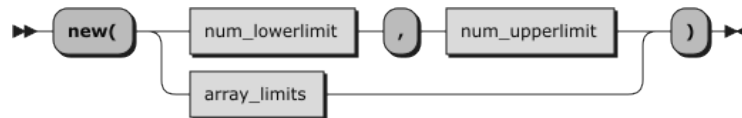
4.2 **DZN Class**

Dead zone.

4.2.1 **new (class method)**

A new instance of the DZN class.

Syntax:



The deadband range is defined by two parameters: `num_lowerlimit` and `num_upperlimit`, or alternatively by a single array using `array_limits`. The lower limit must be less than the upper limit; otherwise, an exception is raised.

Returns a new instance of the DZN class.

4.2.2 **set_limits**

Sets the limits.

Syntax:



It can be called with two separate parameters `num_lowerlimit` and `num_upperlimit`, or with a single array object `array_limits` containing exactly two elements: the lower and upper limit.

Returns no value. The method modifies the internal dead zone configuration of the object.

4.2.3 get_limits

Get limits.

Syntax:

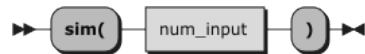


The method has no parameters.

Returns a new ARRAY class instance with the dead zone limits.

4.2.4 sim

Performs one simulation calculation step.



The parameter `num_input` specifies the current input value.

The method returns the output depending on `num_input` and the configured dead zone limits.

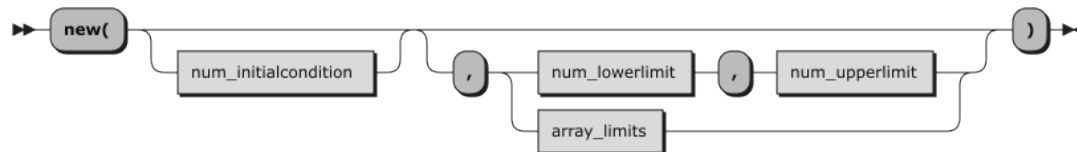
4.3 INT Class

The INT class represents a numerical integrator that accumulates the input signal over time.

4.3.1 new (class method)

A new instance of the INT class.

Syntax:



The optional parameter `num_initialcondition` sets the integrator's initial state. Output saturation can be enabled by specifying the optional parameters `num_lowerlimit` and `num_upperlimit`. Alternatively, both limits can be provided as an optional two-element ARRAY `num_arraylimits`.

Returns a new instance of the INT class with optional initial condition and saturation limits.

4.3.2 reset

Integral reset. Resets the integrator to its initial state. The initial state is defined by the initial condition and, optionally, the saturation limits.

Syntax:



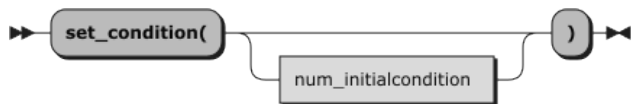
This method takes no parameters.

Returns no value.

4.3.3 set_condition

Sets the initial condition.

Syntax:



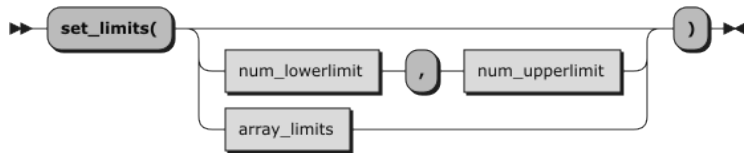
The `initial_condition` parameter is optional and, if not specified, the initial condition is set to zero by default.

Returns no value.

4.3.4 set_limits

Sets the limits of the output limiter.

Syntax:



Integrator limits are set by `num_lowerlimit` and `num_upperlimit`, or by a two-element ARRAY `array_limits`. If not specified, the output limiter is disabled.

Returns no value.

4.3.5 get_condition

Gets initial condition.

Syntax:



The method has no parameters.

Returns the initial condition of the integrator.

4.3.6 get_limits

Gets the limits of the output limiter.

Syntax:



The method has no parameters.

Returns a new instance of the ARRAY class with lower and upper limits.

4.3.7 is_saturated

Check if the output from the integrator is saturated.



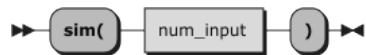
The method has no parameters.

Returns `.true` when the integrator output is at one of the saturation limits; otherwise, `.false`.

4.3.8 sim

Performs one step of the simulation calculation.

Syntax:



Method accepts one parameter `num_input`, the current input to be integrated.

Returns the current value of the integration. If saturation is enabled, the output is clipped to the specified limits.

Example: Solution of a second-order differential equation by the order reduction method.

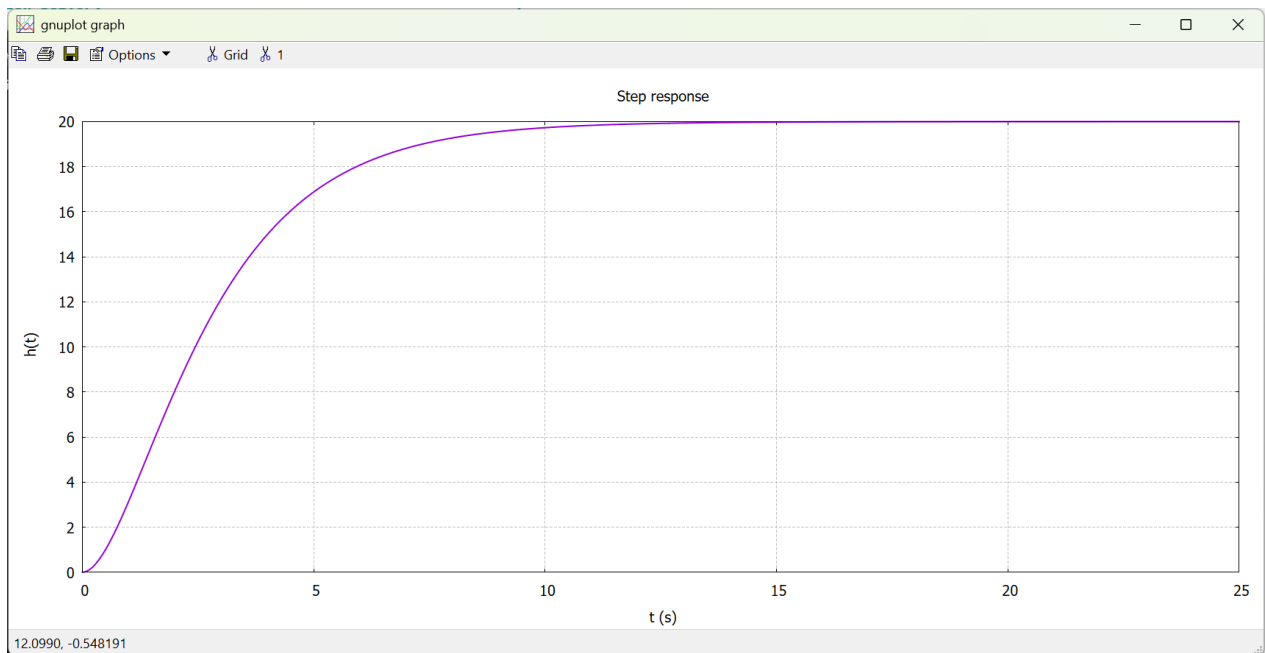
```
1 cls
2
3 int1 = .int~new
4 int2 = .int~new
5 t_arr = .array~new
6 y_arr = .array~new
7
8 y1 = 0
9 y2 = 0
10 y0 = 0
11 u = 1
12
13 ts = 0.01
14 set_ts(ts)
15
16 t = 0.0
```



```

17
18 loop while t <= 25
19
20     y2 = 10*u-1.5*y1-0.5*y0
21     y1 = int2~sim(y2)
22     y0 = int1~sim(y1)
23
24     say format(t,3,2) format(y0,8,6) format(y1,8,6) format(y2,8,6)
25
26     t_arr~append(t)
27     y_arr~append(y0)
28
29     t = t + ts
30
31 end
32
33 grp = .grp~new('Step response','t (s)', 'h(t)')
34 grp~plot(.array~of(t_arr,y_arr),'-')
35
36 ::requires 'rxcacsd.cls'

```



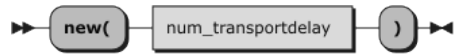
4.4 LAG Class

The class LAG represents transport delay. Delays the output by the specified value of transport delay.

4.4.1 new (class method)

A new instance of the LAG class.

Syntax:



The parameter is transport delay `num_transportdelay`.

Returns a new instance of the class LAG.

4.4.2 set_lag

Sets the transport delay.

Syntax:



The parameter is transport delay `num_transportdelay`.

Returns no value.

4.4.3 get_lag

Obtains the set transport delay.

Syntax:



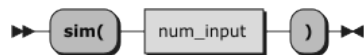
Method has no parameters.

Returns the value of the set transport delay.

4.4.4 sim

Performs one step of the simulation calculation.

Syntax:



The parameter is a `num_input` that will be delayed at the output.

Returns the delayed input signal, shifted by the transport delay time.

4.5 ODE Class

The ODE class represents a system of first-order linear differential equations.

4.5.1 new (class method)

A new instance of the ODE class.

Syntax:



The method accepts up to three parameters. The first parameter `cls_ode` is an instance of the class defining the system of first-order differential equations. This parameter is mandatory. The second parameter `array_timeinterval` specifies the time interval for the calculation. The third parameter `num_sampleperiod` is optional and defines the sample period. If the sample period is not provided, the global sample period is used.

Returns a new instance of the ODE class.

The `cls_odeeq` of system of linear differential equations has the following form:

```
1  ::class odee
2  ::method init    -- initialisation
3  ::method ode     -- mandatory ode method
4  use arg prms
5
6  i = prms[1]      -- index of equation
7  t = prms[2]      -- time
8  x1 = prms[3][1]  -- first state variable
9  x2 = prms[3][2]  -- second state variable
10 .
11 .
12 .
13 xn = prms[3][n]  -- nth state variable
14 select
15   when i = 1 then do
16     res = ... -- first equation
17   end
18   when i = 2 then do
19     res = ... -- second equation
20   end
21 .
22 .
23 .
24   when i = n then do
25     res = ... -- nth equation
26   end
27   otherwise
28     res = 0
29 end
30
31 return res
```

4.5.2 rk4

Starts solving a system of differential equations.

Syntax:



Method has no parameters.

Return a new instance of the ARRAY class with the result. The data are arranged in columns. The first column contains the time, while the remaining columns contain the values of the state variables.

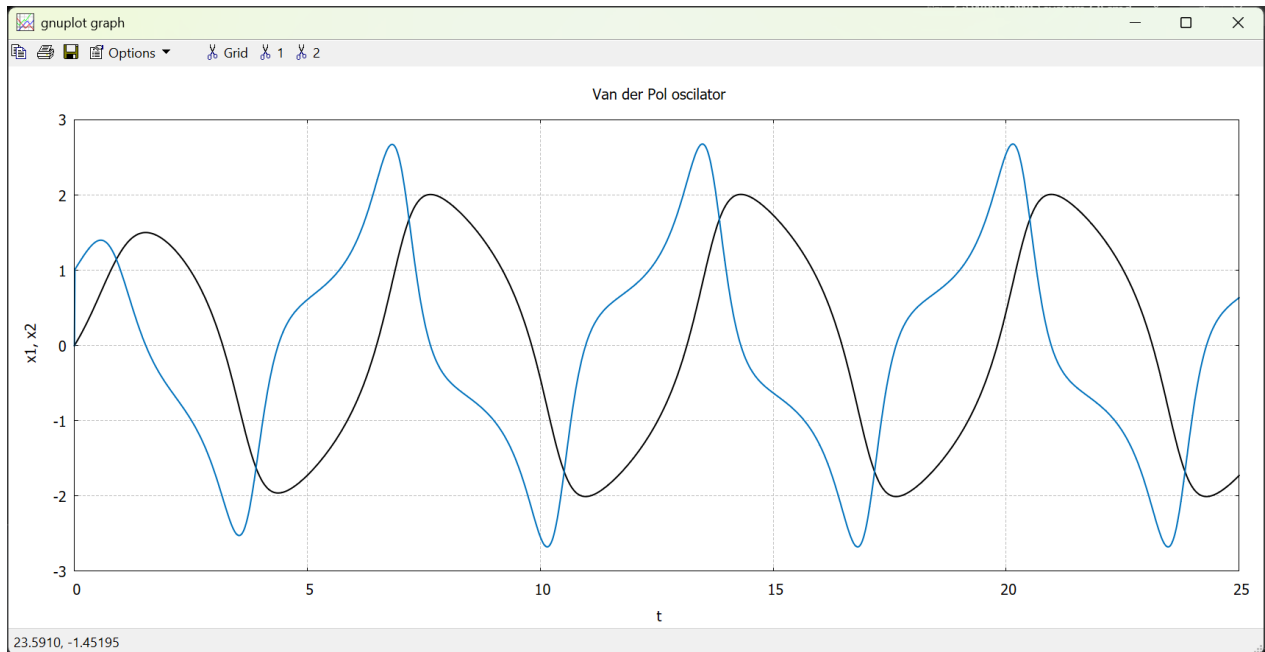
Example: Van der Pol oscillator

```
1 cls
2
3 vand = .fce~new(1.0)
4 solv = .ode~new(vand, .array~of(0,1), .array~of(0,25), 0.01)
5 ret = solv~rk4
6
7 arr_t = .array~new
8 arr_x1 = .array~new
9 arr_x2 = .array~new
10
11 loop i = 1 to ret~items
12   arr_t~append(ret[i][1])
13   arr_x1~append(ret[i][2])
14   arr_x2~append(ret[i][3])
15   say format( ret[i][1], 8, 3 ) format( ret[i][2], 8, 3 ) format( ret[i][3], 8, 3 )
16
17 end
18
19 grp = .grp~new('Van der Pol oscillator', 't', 'x1, x2')
20 grp~plot(.array~of(arr_t, arr_x1), '-k', .array~of(arr_t, arr_x2), '-b')
21
22 exit
23
24
25 ::class fce
26 ::attribute mu
27 ::method init
28   use arg mu
29
30   self~mu = mu
31   return
32
33 ::method ode
34   use arg prms
35
36   i = prms[1]
37   t = prms[2]
38   x1 = prms[3][1]
39   x2 = prms[3][2]
40
```

```

41 select
42   when i = 1 then do
43     res = x2
44   end
45   when i = 2 then do
46     res = self~mu * (1 - x1*x1) * x2 - x1
47   end
48   otherwise
49     res = 0
50 end
51
52 return res
53
54 ::requires 'rxacsd.cls'

```



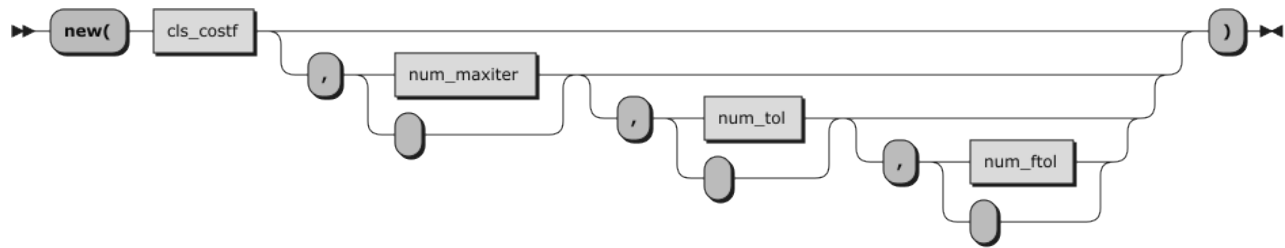
4.6 OPT Class

The OPT class represents an optimization task.

4.6.1 new (class method)

A new instance of the OPT class.

Syntax:



Method has up to four arguments. The first argument, `cls_costf`, is an instance of a class that defines the cost function for the optimization. This argument is required. The second argument, `num_maxiter`, specifies the maximum number of optimization iterations. If not provided, the default value is 1000. The third argument, `num_tol`, and the fourth, `num_ftol`, define the desired calculation accuracy. Both default to 1×10^{-8} .

Returns a new instance of the OPT class.

The cost class has the following form:

```

1  ::class cost
2  ::method init
3    -- initialisation
4  ::method cost -- mandatory cost method
5    use arg x
6    -- objective function (equation)
7    return sum_e2 -- sum of squares of deviations

```

4.6.2 fmins

Starts the optimization process.

Syntax:



The method has one parameter, `array_initialvalues`, which is an array containing the initial values. The length of this array must correspond to the number of parameters being optimized.

Returns a new instance of the ARRAY class that contains the final values obtained from the optimization.

Example: Approximation of measured step response by transfer function

```

1  cls
2
3  opt = .optim~new('data2.dat')
4  opt~go
5  exit
6
7  ::class optim
8    ::attribute k
9    ::attribute t
10   ::attribute h
11   ::method INIT
12     use arg fname

```

```

13 file = .stream~new(fname)
14 if file \= .nil then do
15   rc = file~open('read')
16   if rc = 'READY:' then do
17     self~t = .array~new(file~lines)
18     self~h = .array~new(file~lines)
19     n = 25
20     loop i = 1 to n
21       str = file~linein
22       parse var str t_tmp h_tmp k_tmp
23       self~t[i] = t_tmp
24       self~h[i] = h_tmp/k_tmp
25     end
26     file~close
27     self~k = 0
28     loop i = 1 to 10
29       self~k = self~k + self~h[self~h~items-10+i]
30     end
31     self~k = self~k/10
32     loop i = 1 to self~h~items
33       self~h[i] = self~h[i]/self~k
34     end
35   end
36 end
37 return
38 ::method go
39 say 'Optimization:'
40 opt = .opt~new(.cost~new(self~t,self~h),1000,1e-8,1e-10)
41 ret = opt~fmins(.array~of(1e-3, 1e-3))
42 say
43 say 'Result:'
44 say format(self~k,9,12)
45 say format(ret[1],9,12) format(ret[2],4,12)
46 return
47
48 ::class cost
49   ::attribute t
50   ::attribute h
51   ::method INIT
52     use arg t,h
53     self~t = t
54     self~h = h
55     return
56   ::method cost
57     use arg x
58     om = x[1]
59     xi = x[2]
60     sq = sqrt(1-xi*xi)
61     sum_e2 = 0
62     loop i = 1 to self~t~items
63       e = self~h[i]-(1-exp(-om*xi*self~t[i])/sq*sin(sq*om*self~t[i]+arctan(sq/xi)))
64       sum_e2 = sum_e2+e*e
65     end
66     return sum_e2
67
68 ::requires 'rxacsd.cls'

```

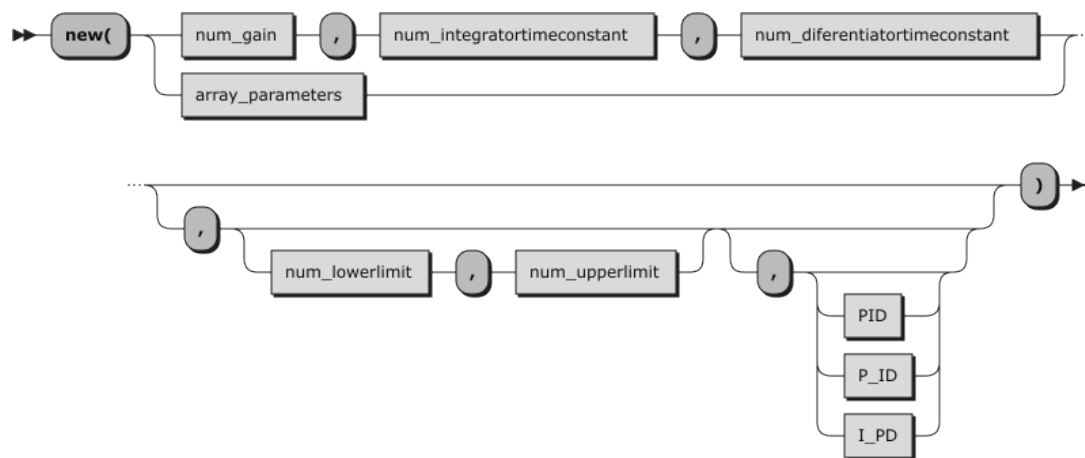
4.7 PID Class

The PID class represents a Proportional–Integral–Derivative controller (PID).

4.7.1 new (class method)

A new instance of the PID class.

Syntax:



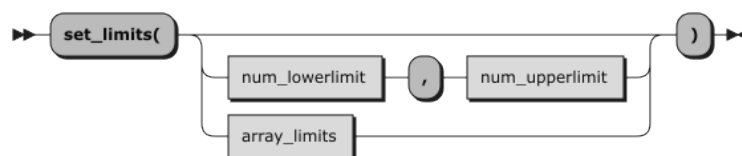
The controller parameters are: gain, integrator time constant, and differentiator time constant. The parameters can be provided separately `gain` `num_integrator_timeconstant` `num_diferentiator_timeconstant` or as an array of values `array_parameters`. It is also possible to set the `num_lowerlimit` and `num_upperlimit` of the output limiter. These can be entered as an array using the code `array_limits`. The final parameter is designated for configuring the structure of the PID controller. The options under consideration are PID, I–PD, and PI–D.

Returns a new instance of the PID class.

4.7.2 set_limits

Sets the boundaries of the output limiter.

Syntax:

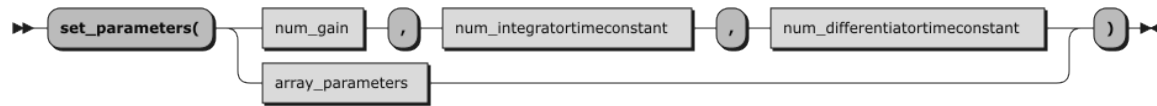


Sets the `num_lowerlimit` and `num_upperlimit`. The limits can be provided separately or as an array using `array_limits`. If no argument is given, the saturation limits default to infinity, effectively disabling the saturation function. To enable saturation, new limits must be explicitly set.

4.7.3 set_parameters

Sets parameters of PID.

Syntax:



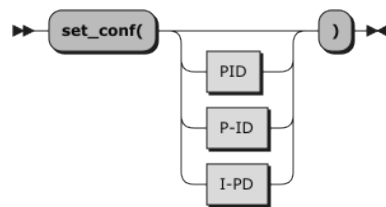
The controller parameters are: gain, integrator time constant, and differentiator time constant. The parameters can be provided separately `gain` `num_integrator_timeconstant` `num_differentiator_timeconstant` or as an array of values `array_parameters`.

Returns no values.

4.7.4 set_conf

Sets the structure of the PID controller.

Syntax:



The method incorporates an optional parameter. The selection of controller structures encompasses PID, P-ID, and I-PD options. In the absence of parameter selection, the PID structure is set.

Returns no parameters.

4.7.5 get_conf

Gets the structure of the PID controller.



Method has no parameters.

Returns a new instance of the `STRING` class that contains information about the structure of the PID controller.

4.7.6 get_limits

Gets the limits of the output limiter

Syntax:



The method has no parameters.

Returns a new instance of the ARRAY class containing the lower and upper limits.

4.7.7 get_parameters

Gets the parameters of the PID controller.

Syntax:



Methods have no parameters.

Returns a new instance of the ARRAY class containing the gain, integrator time constant, and differentiator time constant.

4.7.8 is_saturated

Check if the output from the controller is saturated.



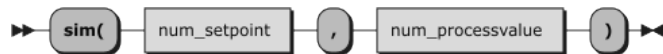
The method has no parameters.

Returns `.true` when the controller output is at one of the saturation limits; otherwise, `.false`.

4.7.9 sim

Performs one step of the simulation calculation.

Syntax:



The input is designated as the `num_setpoint` and the `num_processvalue`.

Returns the updated value of the manipulated variable based on the size of the control deviation. The control deviation is calculated as the difference between the desired value and the controlled variable.

Example: Ziegler-Nichols method of tuning PID parameters

```

1 cls
2
3 say 'PID CONTROLLER TUNING BY ULTIMATE GAIN METHOD'
4 say
5
6 Gs=.tf~new(.array~of(12),.array~of(1,6,11,6))           -- transfer function of plant

```

```

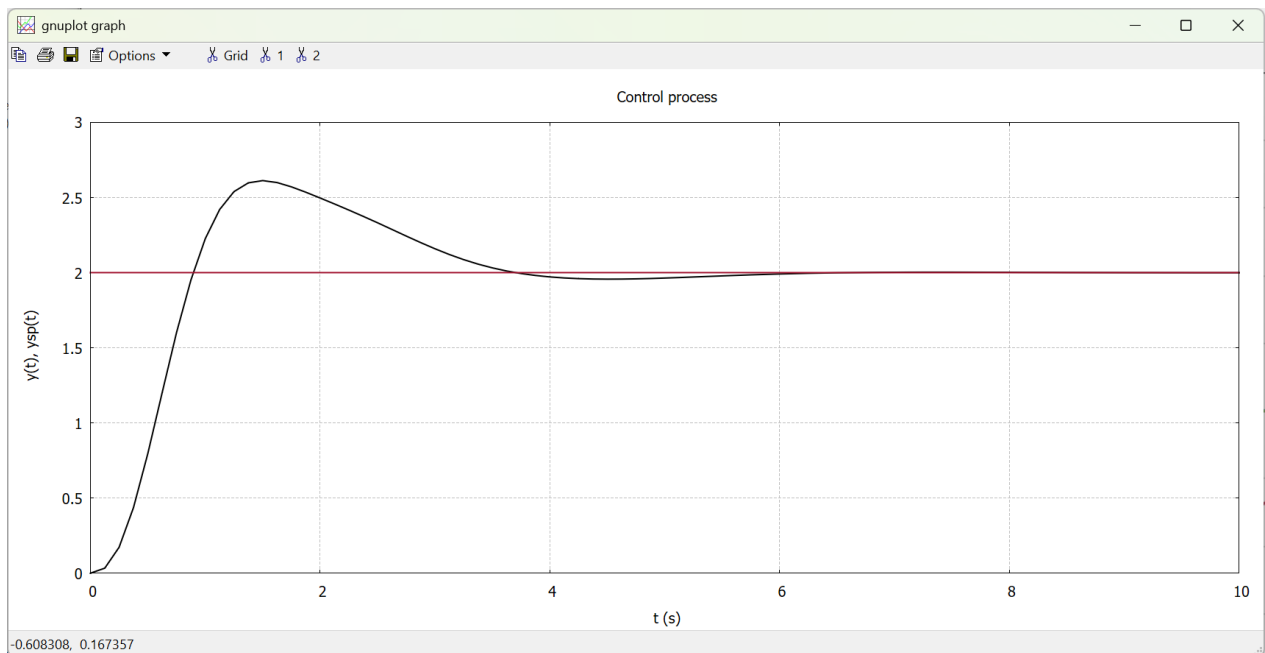
7  say 'Plant:'
8
9  say
10 say 'Gs = '
11 say Gs
12 say
13
14 -- critical gain
15 opt = .opt~new(.cost~new(Gs),1000,1e-13,1e-13)
16 ret = opt~fmins(.array~of(1e-3))
17
18 ku = ret[1]
19
20 -- critical period
21 Gru = .tfn~new(.array~of(ku),.array~of(1)) -- transfer function of P
    controller
22 Glu = feedback(series(Gru,Gs),.tfn~new(.array~of(1),.array~of(1))) -- transfer function of
    control loop with critical gain
23 den = Glu~get_den
24 an = den[1]
25 loop i = 1 to den~items
26     den[i] /= an
27 end
28 tu = (2*RxCalcPi())/RxCalcSqrt(den[den~items-1])
29
30 a_prm = .array~new
31
32 -- Ziegler-Nichols
33 Kc = 0.6*Ku
34 Ti = Tu/2
35 Td = Tu/8
36
37 say 'Critical parameters:'
38 say
39 say 'Ku =' format(Ku,3,6)
40 say 'Tu =' format(Tu,3,6) 'sec'
41 say
42 say 'PID parameters:'
43 say
44 say 'Kc =' format(Kc,3,6)
45 say 'Ti =' format(Ti,3,6) 'sec'
46 say 'Td =' format(Td,3,6) 'sec'
47 say
48
49 ts = 0.01
50 t_sim = 10
51 rc = set_ts(ts) -- set global step
52
53 R = .pid~new(Kc,Ti,Td) -- define PID controller
54
55 a_t = .array~new -- for chart
56 a_y = .array~new
57 a_ysp = .array~new
58
59 u = 0 -- initial value of manipulated variable
60 ysp = 2 -- set-point

```

```

61 t = 0
62 dis = 10
63 i = 0
64 y = 0
65 a_t~append(0)
66 a_y~append(0)
67 a_ysp~append(ysp)
68
69 loop i = 0 to t_sim/ts      -- infinite control loop
70   u = R~sim(ysp,y) -- manipulated variable from controller
71   y = Gs~sim(u)      -- system response to manipulated variable
72   say format(i*ts,8,6) format(y,8,6) format(u,8,6) format(ysp-y,8,6) -- format output to the
      screen
73   if i // 5 = 0 then do
74     a_t~append(i*ts)
75     a_y~append(y)
76     a_ysp~append(ysp)
77   end
78   nop
79 end
80
81 grp = .grp~new('Control process','t (s)', 'y(t), ysp(t)')
82 grp~plot(.array~of(a_t,a_y),'-k',.array~of(a_t,a_ysp),'-r')
83
84 exit
85
86 ::class cost
87
88 ::attribute Gs
89
90 ::method init
91   use arg Gs
92
93   self~Gs = Gs
94   return
95
96 ::method cost
97   use arg x
98
99   ku = x[1]
100
101   Gr = .tfn~new(.array~of(ku),.array~of(1)) -- transfer function of P controller
102   Gl = feedback(series(Gr,self~Gs),.tfn~new(.array~of(1),.array~of(1))) -- transfer function
      of control loop
103   mh = Gl~hurwitz -- Hurwitz matrix of transfer function of control loop
104   e = (mh[.array~of(1,1),.array~of(2,2)])~det
105   e2 = e*e
106
107   return e2
108
109 ::requires 'rxcacsd.cls'

```



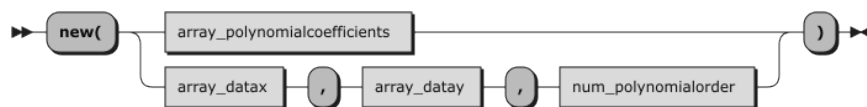
4.8 POL Class

The POL class represents a polynomial.

4.8.1 new (class method)

A new instance of the POL class.

Syntax:



Two call variants are supported.

In the first form, `new(array_polynomialcoefficients)`, the constructor takes one ARRAY with polynomial coefficients ordered from highest to lowest degree.

In the second form, `new(array_datax, array_datay, num_polynomialorder)`, the constructor takes two data arrays (`array_datax` for the independent variable and `array_datay` for the dependent values) and the polynomial order. The polynomial is then approximated to the data using the least-squares method.

Returns a new instance of the POL class.

Example: Polynomial creation

```

1 p1=.pol~new(.array~of(3,2,1)) -- new polynomial 3x^2 + 2x + 1
2 say p1 -- print to screen
3

```

```

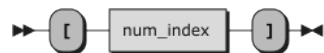
4 u=.array~of( 5.00, 6.00, 7.00, 8.00, 9.00, 10.00, 11.00)
5 y=.array~of( 3.15, 4.39, 5.49, 6.32, 7.75, 8.54, 9.71)
6 p2=.pol~new(u,y,1) -- new approximation polynomial
7 say p2 -- print to screen
8
9 up = linspace(5.00,11.00,100) -- values on u axis for evaluation
10 yp = p2~eval(up)
11
12 grp = .grp~new('Data approximation','u','y')
13 grp~plot(.array~of(u,y),'ro',.array~of(up,yp),'--k',) -- draw chart for evaluation
14
15 ::requires 'rxcacsd.cls'

```

4.8.2 [] (indexing)

Gets the value of the polynomial coefficient.

Syntax:



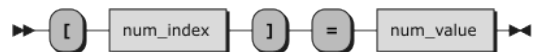
This method takes a required parameter `num_index`, which specifies the index (degree) of the coefficient being searched for.

Returns the value of the polynomial coefficient given by the index.

4.8.3 [] = (assignment)

Sets the coefficient of the polynomial by degree.

Syntax:



This method takes a required parameter `num_index`, representing the degree index (starting from 1), and a value `num_value` to assign to the coefficient of the term.

Returns no parameters.

4.8.4 sum

Adds two polynomials. The method adds the polynomial given as a parameter to the current polynomial. The original polynomial remains unchanged.

Syntax:



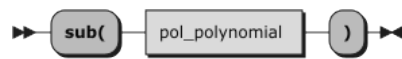
The method has a parameter `pol_polynomial`.

Returns a new instance of the POL class representing the sum of the polynomials.

4.8.5 sub

Polynomial difference. The method subtracts the polynomial given as a parameter from the current polynomial. The original polynomial remains unchanged.

Syntax:



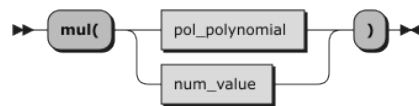
The method has a parameter `pol_polynomial`.

Returns a new instance of the POL class representing the subtraction result.

4.8.6 mul

The product of two polynomials or a polynomial and a scalar.

Syntax:



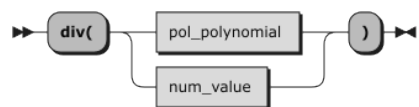
The method has one parameter `pol_polynomial` of type POL or number `num_value`.

Returns a new instance of the POL class, which contains the result of the product.

4.8.7 div

Divides two polynomials or polynomials and a number.

Syntax:



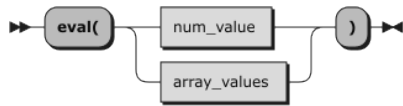
The method has one parameter. Parameter can be polynomial `pol_polynomial` or scalar value `num_value`.

Returns a new instance of the POL class with the result. This does not change the original polynomial. It divides it by the polynomial or by a number passed as a parameter.

4.8.8 eval

Evaluates a polynomial at a given value or an array of values of the variable.

Syntax:



The method has one parameter, which can be a single value `num_value` or an array `array_values` of values.

Returns a new instance of the ARRAY class. This instance contains a two-column array where the first column holds the values of the independent variable, and the second column holds the corresponding values.

4.8.9 roots

Calculates the roots of a polynomial.

Syntax:



The method has no parameters. It operates on the current instance of the POL class.

Returns a new instance of the ARRAY class containing the polynomial's real and complex roots. Each root is represented as an array, with the first element representing the real part and the second element representing the imaginary part.

4.8.10 equal

Compares two polynomials

Syntax:



The method has one parameter, `pol_polynomial`.

Returns a boolean value: `.true` if the polynomials are identical (having the same degree and coefficients), and `.false` otherwise.

4.8.11 der

Derivates a polynomial.

Syntax:



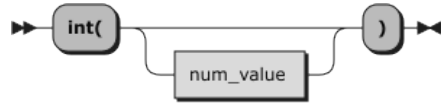
Method has no parameters.

Returns a new instance of the POL class with the result of the differentiation. The original polynomial remains unchanged.

4.8.12 int

Integrates a polynomial.

Syntax:



Method has one optional parameter `num_value`, which specifies the initial value.

Returns a new instance of the POL class with the result of the differentiation. The original polynomial remains unchanged.

4.9 PFS Class

The PFS class represents partial fractions.

An instance of this class is the result of calling the residue method of the TFN class.

4.9.1 to_array

Returns an ARRAY of ARRAYS containing residues, poles, and pole orders.

Syntax:



The method has no parameters.

Returns a new instance of the ARRAY class with the partial fraction decomposition. The returned ARRAY contains three elements: The first element is an ARRAY of residues. The second element is an ARRAY of poles. The third element is an ARRAY of pole orders.

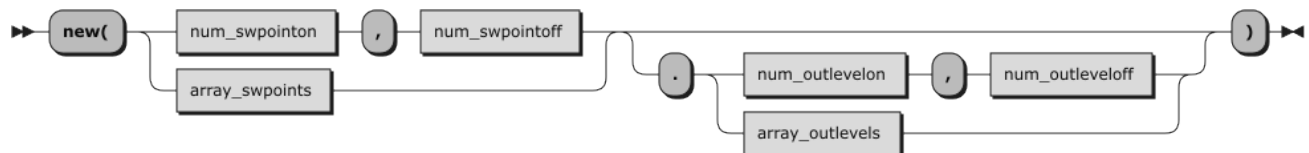
4.10 REL Class

The REL class represents a relay.

4.10.1 new (class method)

A new instance of the REL class.

Syntax:



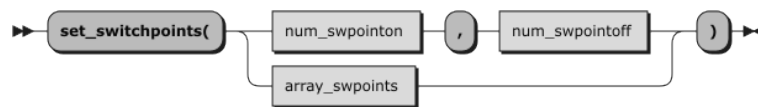
The method has two parameters, `num_swpointon` and `num_swpointoff`, which define the threshold values at which the relay switches from OFF to ON and vice versa. These thresholds can also be specified as a two-element array `array_swpoints`. Optionally, the output levels corresponding to each relay state can be configured using the `num_outlevelon` and `num_outleveloff` parameters. Alternatively, these parameters can be provided as a two-element array `array_outlevels`. If the output level parameters are not specified, the output defaults to 1 when ON and 0 when OFF.

Returns a new instance of the REL class.

4.10.2 set_switch_points

Set the switch-on and switch-off points.

Syntax:



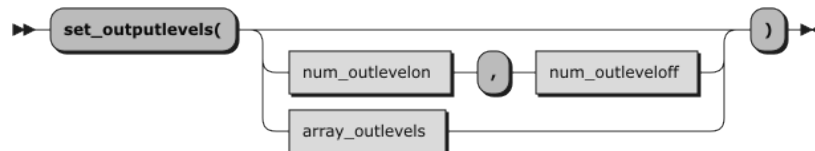
The method has two parameters, `num_swpointon` and `num_swpointoff`, which define the threshold values at which the relay switches from OFF to ON and vice versa. Thresholds can also be specified as a two-element array `array_switchpoints`.

Returns no value.

4.10.3 set_outputlevels

Sets the relay output levels.

Syntax:



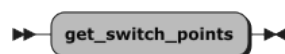
The output levels corresponding to each relay state can be configured using the `num_outlevelon` and `num_outleveloff` parameters. These parameters can also be provided as a two-element array `array_outlevels`. If the output level parameters are not specified, the output defaults to 1 when ON and 0 when OFF.

Returns no value.

4.10.4 get_switch_points

Returns switch points.

Syntax:



Method has no parameters.

Returns a new instance of the ARRAY class that contains the switch points.

4.10.5 get_output_levels

Returns levels of the output.

Syntax:



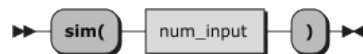
Method has no parameters.

Returns a new instance of the ARRAY class that contains the levels of the relay.

4.10.6 sim

Performs one step of the simulation calculation.

Syntax:



The input is designated as the `num_input`

Returns the relay output value based on the current input and the configured thresholds.

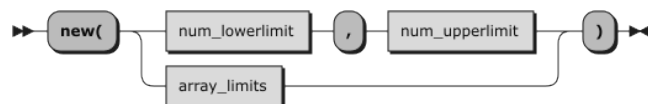
4.11 SAT Class

The SAT class represents the saturation of signals.

4.11.1 new (class method)

New instance of SAT class,

Syntax:



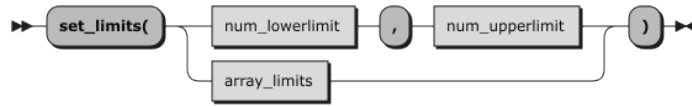
The parameters are the lower limit `num_lowerlimit` and the upper limit `num_upperlimit`. The limits can also be provided as an array `array_limits`.

Returns the new instance of SAT class.

4.11.2 set_limits

Sets limits

Syntax:



The method sets the values of **num_lowerlimit** and **num_upperlimit**. The limits can be provided either as separate parameters or as an array **array_limits**.

Returns no value. The method modifies the internal saturation configuration of the object.

4.11.3 get_limits

Gets limits.

Syntax:



The method has no parameters.

Returns a new instance of the ARRAY class containing the values of lower limit and upper limit.

4.11.4 is_saturated

Checks saturation.

Syntax:



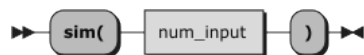
The method has no parameters.

Returns **.true** if the saturator is in the saturated state; otherwise returns **.false**.

4.11.5 sim

Performs one step of the simulation calculation.

Syntax:



The input is designated as the **num_input**.

Returns the saturated value.

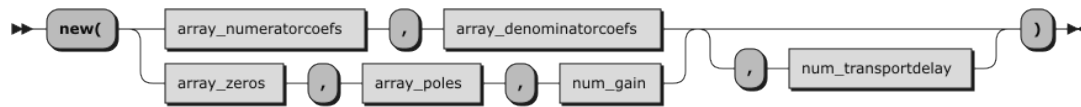
4.12 TFN Class

The TFN class represents a transfer function.

4.12.1 new (class method)

New instance of the TFN class.

Syntax:



The transfer function can be defined in two ways: by providing the coefficients of the numerator polynomial `array_numeratorcoefs` and the coefficients of the denominator polynomial `array_denominatorcoefs`, or by providing the zeros `array_zeros`, the poles `array_poles`, and the static gain (static sensitivity) `num_gain`. In both cases, a transport delay can also be specified using the parameter `num_transportdelay`.

Returns a new instance of the TFN class.

Example: Transfer function from coefficients.

```
1 num = .array~of(0.5,1.0)
2 den = .array~of(1.0,3.0,2.0)
3 td = 0.1
4 G = .tfn~new( num, den, td )
```

Zeros and poles are passed as arrays of complex numbers, where each complex number is represented as a two-element array.

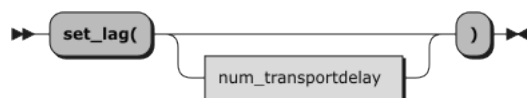
Example: Transfer function from ZPK.

```
1 zeros = .array~of(.array~of(-2.0,0),.array~of(-1.0,0))
2 poles = .array~of(.array~of(-2.0,0),.array~of(-1.0,0))
3 k = 1
4 G = .tfn~new(zeros,poles,k)
```

4.12.2 set_lag

Sets transport delay.

Syntax:



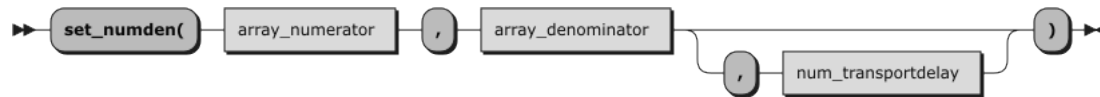
Sets the transport delay `num_transportdelay` of the plant. If no parameter is provided, the transport delay is set to zero.

Returns no value.

4.12.3 set_numden

Sets coefficients of the numerator and denominator and optionally transport delay.

Syntax:



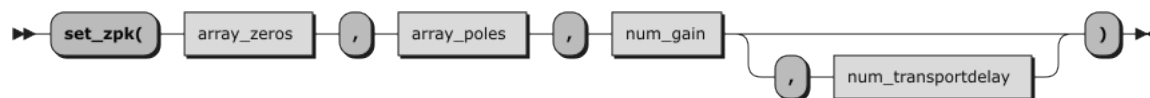
The method takes two mandatory parameters: `array_numerator` and `array_denominator`, which define the coefficients of the transfer function polynomials. The coefficients must be ordered in descending powers (from the highest to the lowest degree). An optional parameter `num_transportdelay` specifies a transport delay.

Return no value. The method reinitializes the internal data of the transfer function object.

4.12.4 set_zpk

Sets zeros, poles, static gain, and optionally transport delay.

Syntax:



The method takes three mandatory parameters: `array_zeros`, `array_poles`, and `num_gain`. The arrays define the zeros and poles of the transfer function, while the scalar gain specifies its amplification. An optional parameter `num_transportdelay` specifies a transport delay.

Returns no value. The method reinitializes the internal data of the transfer function object.

4.12.5 get_den

Gets the denominator of the transfer function.

Syntax:



The method takes no parameters.

Returns a new instance of the `ARRAY` class containing the denominator coefficients of the transfer function. The coefficients are stored in the array in descending order of powers, from the highest degree to the lowest.

4.12.6 get_gain

Gets the static gain.

Syntax:



The method takes no parameters.

Returns the static gain of the transfer function..

4.12.7 get_lag

Returns transport delay.

Syntax:



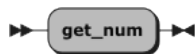
Method has no parameters.

Returns the transport delay of the system.

4.12.8 get_num

Gets the numerator of the transfer function.

Syntax:



The method takes no parameters.

Returns a new instance of the **ARRAY** class containing the numerator coefficients of the transfer function. The coefficients are stored in the array in descending order of powers, from the highest degree to the lowest.

4.12.9 get_poles

Gets the poles of the transfer function.

Syntax:



The method takes no parameters.

Returns an instance of the **ARRAY** class containing the poles.

4.12.10 get_zeros

Gets the zeros of the transfer function.

Syntax:



The method takes no parameters.

Returns an instance of the **ARRAY** class containing the zeros.

4.12.11 hurwitz

Constructs the Hurwitz matrix associated with the denominator of the transfer function.

Syntax:



The method takes no parameters.

Returns a new instance of the **MAT** class representing the Hurwitz matrix derived from the transfer function's denominator coefficients.

Example: Checking system stability by Hurwitz method

```

1 say is_stable(.tfn~new(.array~of(2),.array~of(1,5,8,4,1)))
2 say is_stable(.tfn~new(.array~of(2),.array~of(1,-2,3,-4,5)))
3 say is_stable(.tfn~new(.array~of(2),.array~of(1,2,2,2,1)))
4 say is_stable(.tfn~new(.array~of(2),.array~of(1,3,3,0,0)))
5
6 ::routine is_stable
7
8   use arg G
9
10  ret = 'UNSTABLE'
11
12  say
13  say G -- show transfer function
14
15  MH=G~hurwitz
16
17  say
18  say MH -- show Hurwitz matrix
19
20  MD = .array~new
21
22  MD~append(MH[1,1])
23  loop i = 2 to (MH~dim)[1]
24    MD~append((MH[.array~of(1,1),.array~of(i,i)]~det) -- add subdeterminants to MD
25  end
26
27  fl_s = .true -- flag (init. stable)
28  fl_z = .false -- flag (init. no zero det)
29
30  loop i = 1 to MD~items
31    if MD[i] < 0 then do -- negative det detection
32      fl_s = .false

```



```

33     leave
34   end
35   if \ (abs(MD[i])>0) then do -- zero det detection
36     fl_z = .true
37   end
38   end
39
40   if fl_s == .true then do
41     if fl_z = .false then
42       ret = 'STABLE'
43     else do
44       den = G~get_den
45       if abs( den[den~items] ) > 0 then
46         ret = 'MARGIN_OSC'
47       else
48         ret = 'MARGIN_NONOSC'
49     end
50   end
51
52   return ret

```

4.12.12 residue

Decomposes the transfer function into partial fractions.

Syntax:



The method has no parameters.

Returns a new instance of the PFS class with partial fractions.

Example: Partial fractions calculation

```

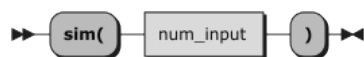
1 cls
2
3 G_TF = .tf~new(.array~of(1),.array~of(1,0.8,1))
4 frac = G_TF~residue
5 say frac
6
7 ::requires 'rxcacsd.cls'

```

4.12.13 sim

Performs one simulation iteration.

Syntax:



The method takes the parameter `num_input`

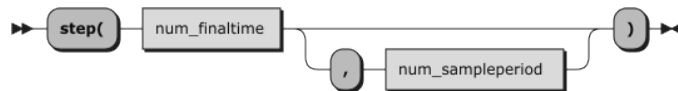
It returns the corresponding response.

Note: The simulation step size is set globally using the `set_ts` routine.

4.12.14 step

Step response. Computes discrete points of the step response curve, representing the system's output over time following a unit step input.

Syntax:

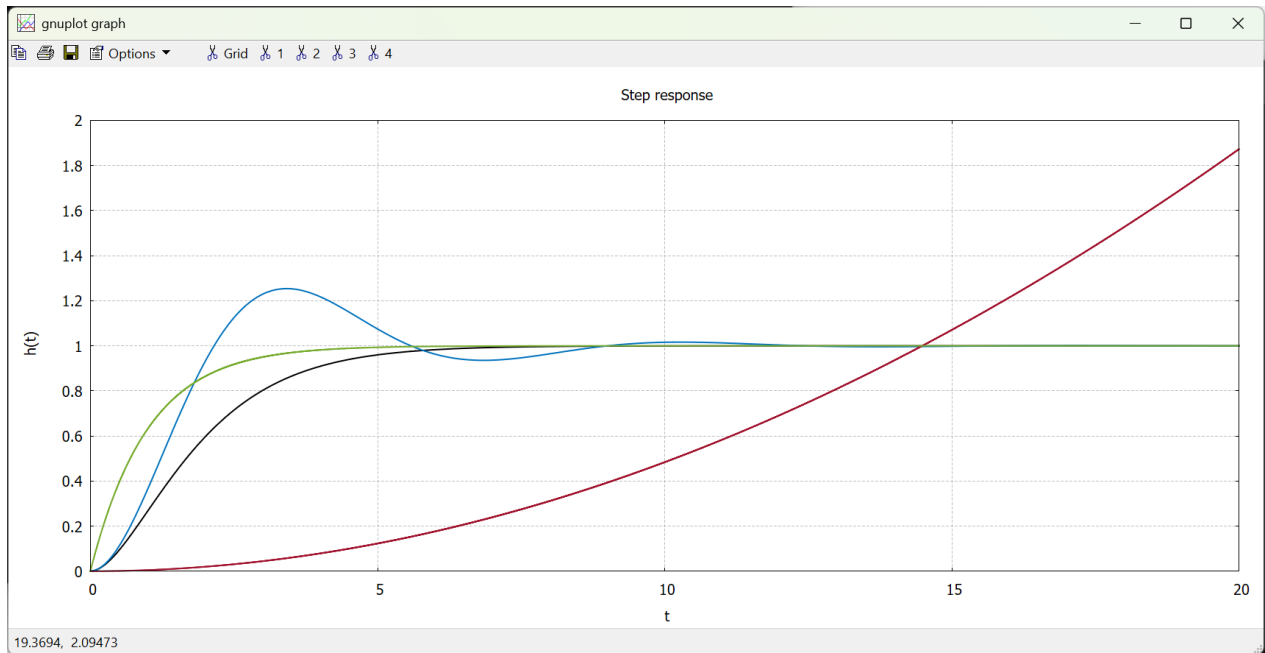


The parameter `num_finallytime` specifies the duration of the simulation. Optionally, a sampling period can be provided using `num_sampleperiod`. If no sampling period is given, a global default value is used, which can be set using the `set_ts` function.

Returns a 2D array with two columns: the first column contains time values, and the second column contains the corresponding step response. The function does not produce a plot; use the plot method from the GRP class to visualize the result.

Example: Step response plot

```
1 cls
2
3 rc = set_ts(0.01)
4
5 G1 = .tfn~new(.array~of(1),.array~of(1,2,1))
6 G2 = .tfn~new(.array~of(1),.array~of(1,0.8,1))
7 G3 = .tfn~new(.array~of(1),.array~of(100,1,0))
8 G4 = .tfn~new(.array~of(1,1),.array~of(1,2,1))
9
10 h1= G1~step(20,0.01)
11 h2= G2~step(20,0.01)
12 h3= G3~step(20,0.001)
13 h4= G4~step(20,0.001)
14
15 grp = .grp~new('Step response','t','h(t)')
16 grp~plot(h1,'-k',h2,'-b',h3,'-r',h4,'-g')
17
18 ::requires 'rxcacsd.cls'
```



4.12.15 nyquist

Nyquist Frequency Response.

Syntax:



Computes the Nyquist points of the frequency response (frequency response in the complex plane).

The parameter is an array of frequencies, **array_frequencies**, for which the frequency response is to be calculated. Frequencies are specified in rad/s. Frequencies can be generated using the **logspace** routine.

Returns a new instance of the **ARRAY** class as a two-dimensional array with three columns. The first column lists the frequencies, and the second and third columns contain the corresponding coordinates of the points on the frequency response.

Note: This method does not generate a plot. To visualize the graph, use the **nyquist** method of the **GRP** class.

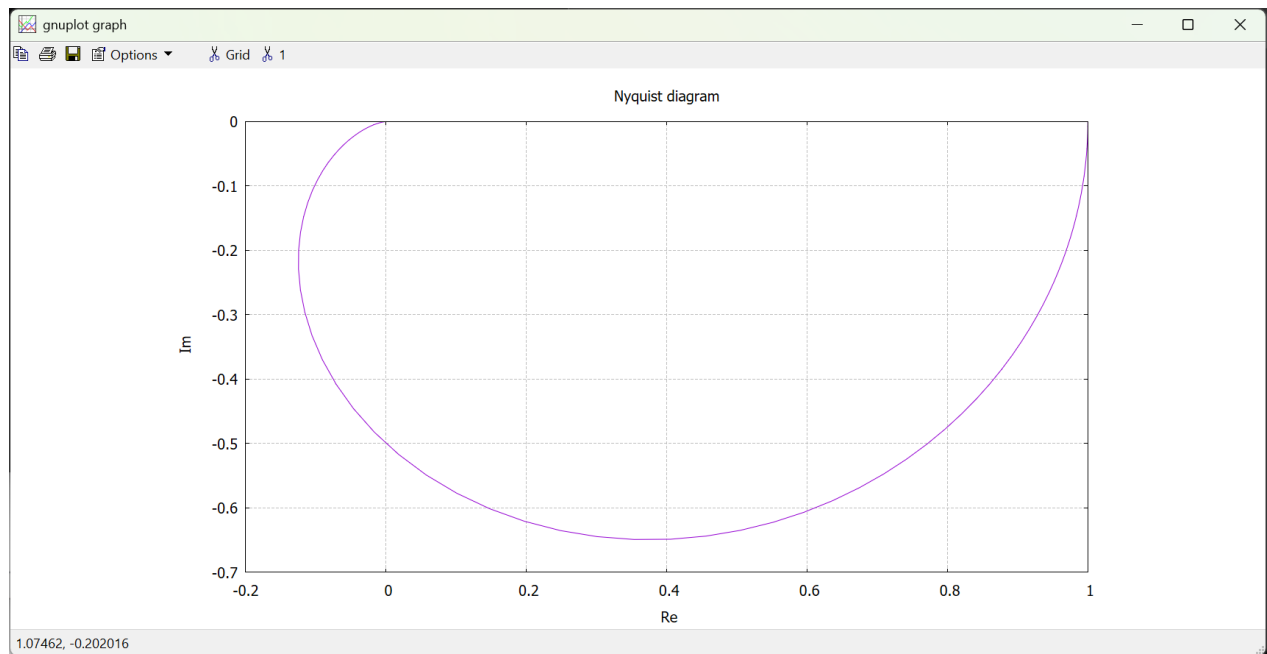
```

1  cls
2
3  G = .tf~new(.array~of(1),.array~of(1,2,1))
4  om_re_im = G~nyquist(logspace(-3,3,200))
5
6  loop i = 1 to om_re_im[1]~items
7    say format(om_re_im[1][i],8,12) format(om_re_im[2][i],8,12) format(om_re_im[3][i],8,12)
8  end
9
10 grp = .grp~new
11 grp~nyquist(om_re_im)
12

```

13

::requires 'rxcacsd.cls'



4.12.16 bode

Bode Frequency Response (Frequency Response in Logarithmic Coordinates).

Syntax:



The frequency response is computed at discrete frequencies specified by the parameter `array_frequencies`. These frequencies must be given in radians per second (rad/s). For convenience, the frequency array can be generated using the `logspace` routine, which produces logarithmically spaced values suitable for Bode plots.

Returns a new ARRAY instance with the frequency response data. The result is a two-dimensional array (array of arrays) with three columns: the first column contains the frequency values (in rad/s), the second column contains the magnitude, and the third column contains the phase shift in radians.

Note: The graph can be plotted using the `bode` method from the GRP class.

Example: Bode plot

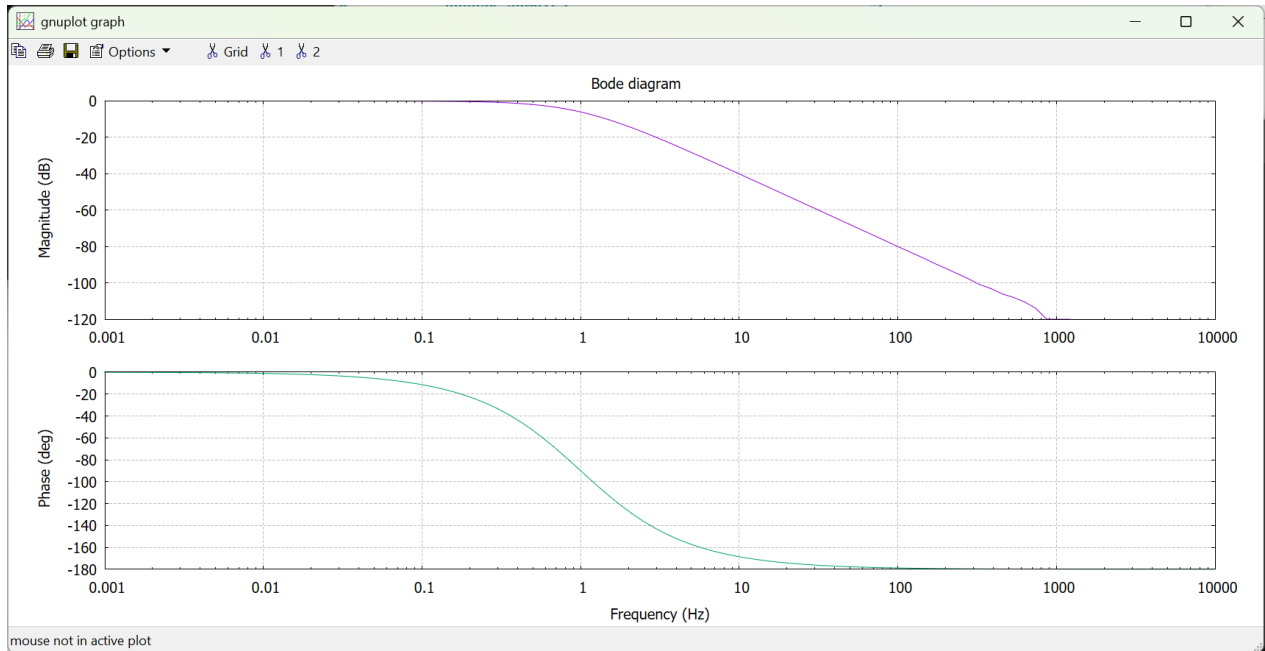
14
15
16
17
18
19
20
21
22

```
cls
G = .tfn~new(.array~of(1),.array~of(1,2,1))
om_md_ph = G~bode(logspace(-3,4,100))
loop i = 1 to om_md_ph[1]~items
  say format(om_md_ph[1][i],8,6) format(om_md_ph[2][i],8,6) format(om_md_ph[3][i],8,6)
end
```

```

23 grp = .grp~new
24 grp~bode(om_md_ph)
25
26 ::requires 'rxcacsd.cls'

```



4.12.17 feedback

Feedback connection of two transfer functions.

Syntax:



The parameter `tfn_plant` is the transfer function that forms the feedback path of the calling transfer function.

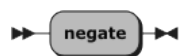
Returns a new instance of the TFN class representing the closed-loop transfer function.

Note: Method assumes unity negative feedback by default. Adjust the feedback system accordingly for positive feedback or a gain-scaled feedback path. Both systems must be proper and stable for the resulting closed-loop system to be well-defined.

4.12.18 negate

Inversion of the sign of the transfer function output.

Syntax:



The parameter `transferfcnobj` is an instance of a transfer function whose output sign is to be inverted.

Returns a new instance of the TFN class with the output sign inverted.

4.12.19 parallel

Parallel connection of two transfer functions.

Syntax:



The parameter `tfn_plant` is the transfer function to be connected in parallel with the calling transfer function. Both systems receive the same input signal, and their outputs are summed.

Returns a new instance of the TFN class representing the parallel connection.

Note: Both systems are assumed to be time-invariant and linear. The resulting system captures the combined behavior of two systems operating in parallel with their outputs added algebraically.

4.12.20 series

Series (cascaded) connection of two transfer functions.

Syntax:



The parameter `tfn_plant` is the transfer function to be connected in series with the calling transfer function. The output of the calling system becomes the input of `tfn_pant`.

Returns a new instance of the TF class representing the series connection.

4.12.21 simplify

Simplification of the transfer function.

Syntax:



The method has no parameters.

Returns a new instance of the TF class representing the simplified transfer function.

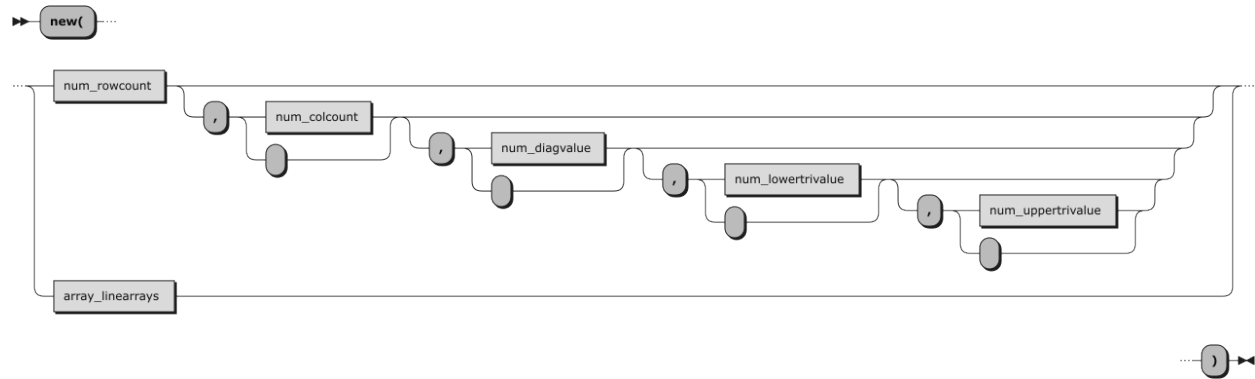
4.13 MAT Class

The MAT class represents a general real matrix.

4.13.1 new (class method)

Creates a new instance of the MAT class.

Syntax:



A matrix can be created in two ways: by specifying its dimensions and optionally the values for specific regions of the matrix, or by providing an array of arrays, where each sub-array represents a row of the matrix.

The first way to create a matrix is by calling `new(...)` with up to five numeric parameters. The first parameter, `num_rowcount`, specifies the number of rows. The second parameter, `num_colcount`, specifies the number of columns and is optional—if omitted, the matrix is assumed to be square with the same number of columns as rows. The third to fifth parameters are optional and allow for initialization of specific regions of the matrix. The third parameter, `num_diagvalue`, sets the value on the main diagonal. The fourth parameter, `num_lowertrivalue`, sets the value in the lower triangle (below the diagonal), and the fifth parameter, `num_uppertrivalue`, sets the value in the upper triangle (above the diagonal). If these values are not specified, all matrix elements are initialized to zero. If `num_uppertrivalue` is not provided but `num_lowertrivalue` is, the lower triangle value is also used for the upper triangle. This initialization method makes it easy to create zero, identity, or diagonally defined matrices.

The second way to create a matrix is by calling `new(...)` with a parameter `array_linearrays`, where each element represents a row of the matrix. Each row is a separate array of values, and the matrix dimensions are automatically determined from the structure of this array.

Returns a new instance of the MAT class with the specified dimensions and content, depending on the chosen initialization method.

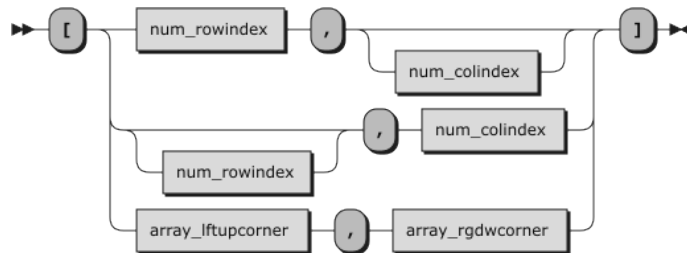
Example: Demonstration of different ways to create and initialize a matrix

```
1 M1 = .mat~new(5)           -- 5x5 zero matrix
2 M2 = .mat~new(5,2)        -- 5x2 zero matrix
3 M3 = .mat~new(5,2,,4)     -- 5x2 matrix filled with the number 4
4 M4 = .mat~new(5,,1)       -- 5x5 identity matrix
5 M5 = .mat~new(5,,1,4)     -- 5x5 matrix filled with the number 4, with 1 on the main diagonal.
6 M6 = .mat~new(5,2,1,4)    -- 5x2 matrix filled with number 4, with 1 on main diagonal
7 M7 = .mat~new(5,2,3,2,4)  -- 5x2 matrix with 2 below the diagonal, 4 above the diagonal, and 3
                             on the diagonal
8 M8 = .mat(.array~of(1,2,3),.array~of(5,6,7)) -- 2x3 matrix,
9                                     -- the first line contains numbers 1,2,3
10                                    -- the second line contains numbers
```

4.13.2 [] (indexing - get)

Gets the value or a submatrix.

Syntax:



The method supports several forms of indexing. Using [num_rowindex, num_colindex] returns a single element. The forms [num_rowindex,] or [, num_colindex] return a complete row or column. The form [array_lftupcorner, array_rgdowncorner] returns a submatrix defined by the upper-left and lower-right corners. Depending on the indexing form, the method returns either a scalar value (num_value) or another matrix (mat_matrix).

Example: Supported ways to index a matrix.

```

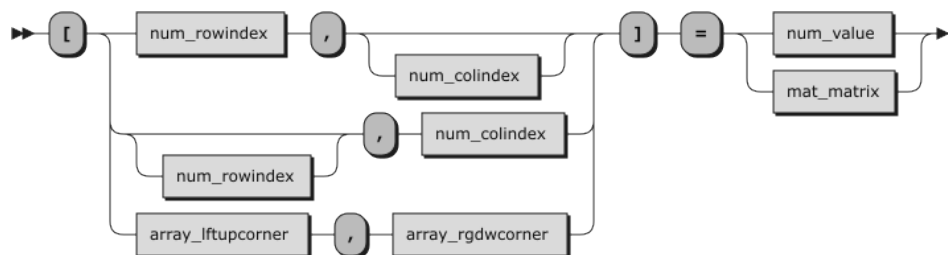
1 mx = .mat~new(4,4)
2 mx~rand
3 say mx[1,1] -- cell indexing
4 say mx[4,] -- row indexing
5 say mx[,2] -- column indexing
6 say mx[.array-of(2,1),.array-of(3,3)] -- submatrix indexing

```

4.13.3 []= (indexing - set)

Sets the value, or a submatrix.

Syntax:



The method supports several forms of indexing. Using [num_rowindex, num_colindex] = num_value assigns a single scalar value to the selected element. The forms [num_rowindex,] = mat_matrix or [, num_colindex] = mat_matrix assign values to an entire row or column. The form [array_lftupcorner, array_rgdowncorner] = mat_matrix assigns values to a submatrix defined by the upper-left and lower-right

corners. Depending on the indexing form, the assigned value can be either a scalar (**num_value**) or another matrix (**mat_matrix**).

The method returns no value. Changes are applied directly in the matrix instance.

Example: Matrix indexing

```
1 mx = .mat~new(4,4) --vytvori nulovou matici 4x4
2 mx[2,] = 3 --2. radek vplni cisle 3
3 mx[,3] = 5 -- 3. sloupec vyplni cisle 5
4 mx[2,2] = 8 -- na pozici 2,2 ulozi cislo 8
5 mx[3,1]=.mat~new(2,2,,4) --na pozici 3,1 je ulozena matic 2x2 vyplnena hodnotou 4
6 say mx
```

4.13.4 dim

Gets the dimension of the matrix.

Syntax:



Method has no parameters.

Returns a new instance of the ARRAY class containing the dimensions of the matrix.

4.13.5 rand

Reinitializes the elements of the matrix with pseudo-random numbers.



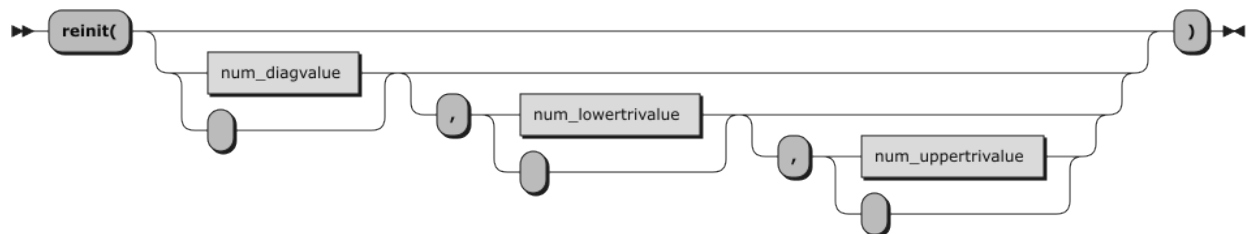
Method has no parameters.

Returns no value. Method reinitializes the internal data of the instance.

4.13.6 reinit

Reinitializes the elements of the matrix.

Syntax:



The method has three optional parameters. If none are provided, all matrix elements are set to zero. Otherwise, values are assigned to the diagonal, subdiagonal, and superdiagonal, similarly to the `new` method.

Returns no value. Method reinitializes the internal data of the instance.

4.13.7 to_array

Converts the matrix to an array.

Syntax:



Method has no parameters.

Returns a new instance of the ARRAY class representing the matrix converted to an array.

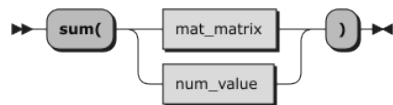
Example: Conversion of a matrix to an array.

```
1 mx=.mat~new(4,,8)
2 say mx
3 ax=mx~to_array
4 loop i=1 to ax~items
5   inx = ax[i]
6   str = ''
7   loop j=1 to inx~items
8     str = str || ' ' || inx[j]
9   end
10  say str
11 end
```

4.13.8 sum

Adds a matrix to another matrix or a scalar.

Syntax:



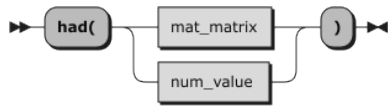
The method has a single parameter, which can be either a matrix `mat_matrix` or a scalar value `num_value`. If a matrix is provided, it is expected to have the same dimensions as the object on which the method is called.

Returns a new instance of the MAT class containing the result of the addition.

4.13.9 had

Multiplies the matrix element-wise by another matrix (Hadamard, Schur multiplication).

Syntax:



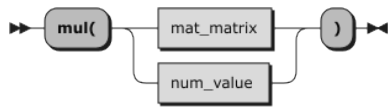
The method has a single parameter, which can be either a matrix `mat_matrix` or a scalar value `num_value`. If a matrix is provided, it is expected to have the same dimensions as the object on which the method is called.

Returns a new MAT instance containing the multiplication result.

4.13.10 mul

Multiplies a matrix by another matrix or a scalar.

Syntax:



The method has a single parameter, which can be either a matrix `mat_matrix` or a scalar value `num_value`. In the case of a matrix, it is assumed that the dimensions are compatible for multiplication (i.e., the number of columns in the first matrix equals the number of rows in `mat_matrix`).

Returns a new MAT instance containing the multiplication result.

4.13.11 trans

Gets the transposed matrix.

Syntax:



Method has no parameters.

Returns a new instance of the MAT class.

4.13.12 det

Gets the determinant of a matrix.

Syntax:



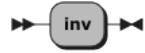
Method has no parameters.

Returns the determinant of the matrix as a scalar value.

4.13.13 inv

Gets the inverse of a matrix.

Syntax:



Method has no parameters.

Returns a new instance of the MAT class containing the inverse matrix.

Example: Solves a system of linear equations using matrix inversion.

```
1  /*
2   2*x1+x2+  x3 =  8
3  -3*x1-x2+2*x3 =-11
4  -2*x1+x2+2*x3 = -3
5  */
6  A=.mat~new(.array~of(.array~of(2,1,-1),.array~of(-3,-1,2),.array~of(-2,1,2))) -- coefficient
   matrix
7  say A
8  b=.mat~new(.array~of(.array~of(8,0,0),.array~of(-11,0,0),.array~of(-3,0,0))) -- right-hand
   side vector
9  say b
10 x = (A~inv)~mul(b) -- solution
11 say x -- display of the result
```

4.13.14 rank

Gets the rank of the matrix.

Syntax:



Method has no parameters.

Returns the rank of the matrix as a scalar value.

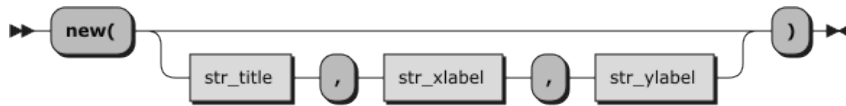
4.14 GRP

The class GRP represents a graphical plot. It provides an interface between the ooRexx application and the Gnuplot plotting engine.

4.14.1 new (class method)

New instance of the GRP class.

Syntax:



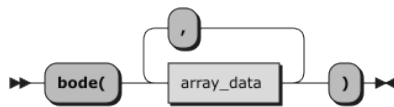
The method accepts an optional parameter for specifying the plot title `str_title` and optional `str_xlabel` and `str_ylabel` parameters for defining the axis labels. These parameters are only applicable when the plot method is invoked; they are ignored in all other contexts.

Returns a new instance of the GRP class.

4.14.2 bode

The method displays the Bode diagram.

Syntax:



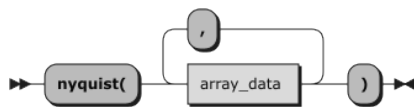
This method takes as input a three-column array `array_data`, where the first column represents frequency in radians per second, the second column corresponds to the magnitude, and the third column contains the phase in radians.

The method returns no value. It displays the Bode plot..

4.14.3 nyquist

The method displays the Nyquist diagram.

Syntax:



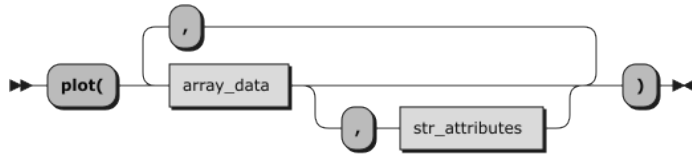
This method takes a three-column array `array_data` as input. The first column specifies the frequency in radians per second, the second column contains the real part, and the third column contains the imaginary part of the complex frequency response.

The method returns no value. It shows Nyquist diagram.

4.14.4 plot

The method displays the 2D plot.

Syntax:



The method expects input data for plotting, provided as an ARRAY object named `array_data`, containing exactly two elements, each of which is also an ARRAY. The first element represents the values of the independent variable (typically the horizontal axis, e.g., time), and the second element contains the corresponding values of the dependent variable (typically the vertical axis, e.g., system output or response). An optional argument `straa` may be provided to configure the plot's visual appearance, such as line style or color.

The method returns no value.

The string `str_attributes` defines the plot's visual style using a concise string format composed of up to three parts:

Table 1: Plot Style Attributes with Descriptions

Line Style Code	Marker Style Code	Color Code
- solid line	o circle marker	r red
- dashed line	s square marker	g green
- . dash-dot line	^ triangle marker	b blue
: dotted line	+ plus marker	k black
	* star marker	y yellow

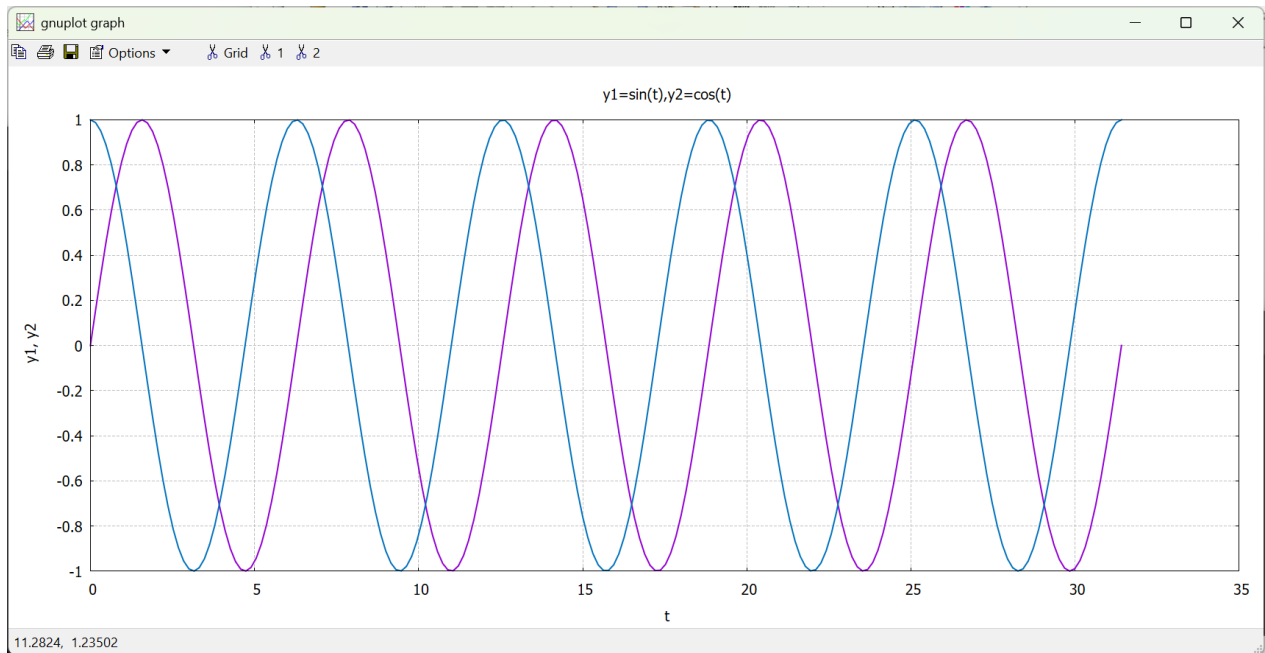
If the line style is omitted, only markers are displayed. Conversely, if the marker style is omitted, only the line is displayed. For example, the string `'-ro'` specifies a red dashed line (–) with circle markers (o), both colored red (r).

Example: Graph of the sin and cos functions

```

1  cls
2
3  t = linspace(0,10*pi(),200)
4
5  y1 = .array~new
6  y2 = .array~new
7  loop i = 1 to t~items
8      y1~append(sin(t[i]))
9      y2~append(cos([i]))
10 end
11
12 grp = .grp~new('y1=sin(t),y2=cos(t)','t','y1, y2')
13 grp~plot(.array~of(t,y1),.array~of(t,y2),'-b') -- blue solid line line for the second curve
14
15 ::requires 'rxcacsd.cls'

```



4.14.5 pzmap

Pole-zero diagram. Graphical representation of the locations of the poles and zeros of a system's transfer function in the complex plane

Syntax:



The method takes an array of zeros and an array of poles, both provided as instances of the ARRAY class.

No return value. Displays the diagram.

Example: Displaying system poles and zeros

```

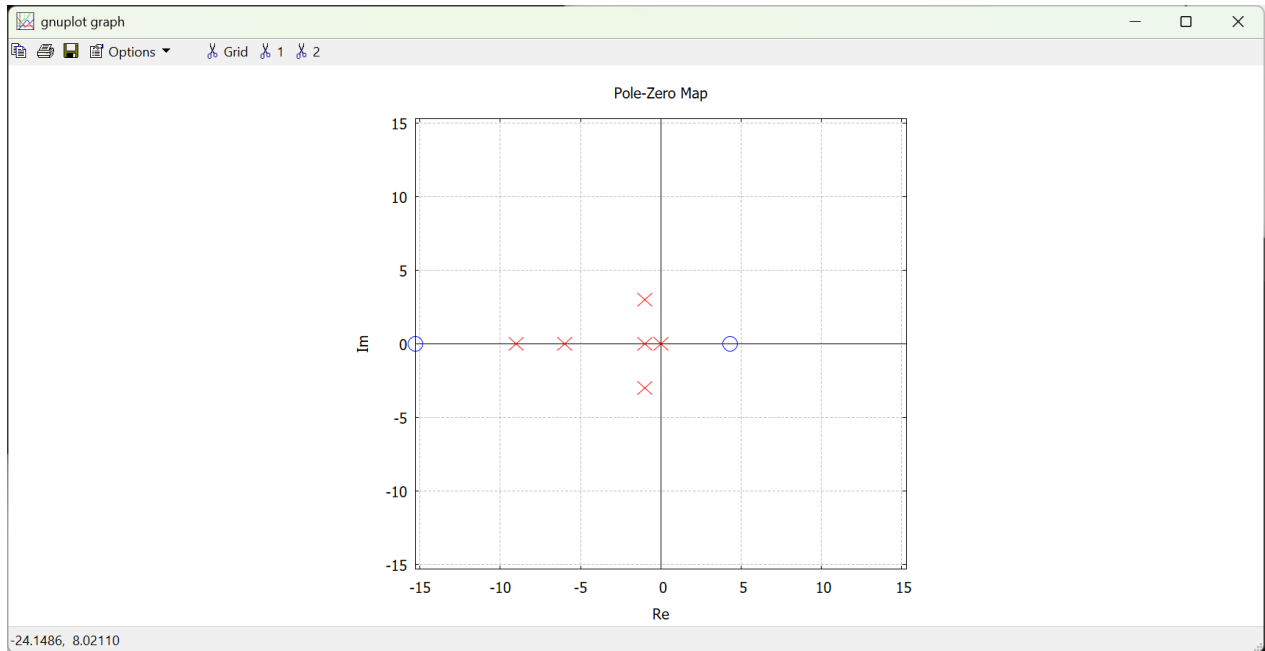
1 cls
2
3 G = .tfnew(.array-of(0.01,0.11,0.14-0.8),.array-of(0.01,0.18,1.11,3.52,7.98,5.4,0))
4 say G
5
6 zeros = G~get_zeros
7 poles = G~get_poles
8
9 loop i = 1 to zeros~items
10   say format(zeros[i][1],8,3) format(zeros[i][2],8,3)
11 end
12
13 loop i = 1 to poles~items
14   say format(poles[i][1],8,3) format(poles[i][2],8,3)
15 end
16

```

```

17 grp = .grp~new
18 grp~pzmap(zeros,poles)
19
20 ::requires 'rxacsd.cls'

```



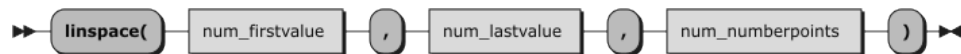
5 BUILTIN ROUTINES

5.1 Array routines

5.1.1 linspace

Generates an array of values over the specified range.

Syntax:



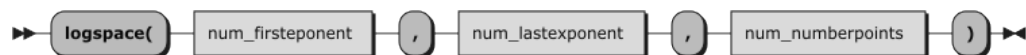
The parameters are `num_firstvalue` (start of the interval), `num_lastvalue` (end of the interval), and `num_numberpoints` (number of points to generate).

Returns a new instance of the ARRAY class containing the generated points.

5.1.2 logspace

Generates an array of values logarithmically spaced (base 10) over the specified range.

Syntax:



The parameters are `num_firstexponent` (starting exponent), `num_lastexponent` (ending exponent), and `num_numberpoints` (number of points to generate).

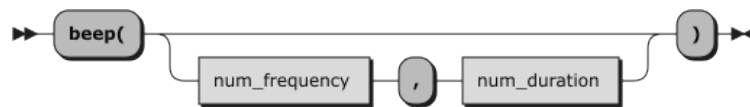
Returns a new instance of the `ARRAY` class containing the generated points.

5.2 Helper routines

5.2.1 beep

Generates an audible signal.

Syntax:



The parameters are `num_frequency`, specifying the signal frequency, and `num_duration`, specifying the signal duration.

Returns no value.

5.2.2 sleep

Pauses the execution of the program

Syntax:



The duration can be specified as a decimal number. If no duration is provided, the function uses the global sampling period set by the `set_ts` function.

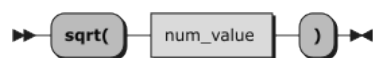
Returns no value.

5.3 Mathematical routines

5.3.1 sqrt

Square root.

Syntax:

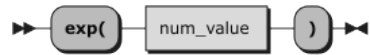


Returns the square root of the `num_value` provided as an argument.

5.3.2 exp

Exponential function.

Syntax:

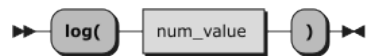


Returns the exponential of the `num_value` provided as an argument.

5.3.3 log

Natural logarithm.

Syntax:

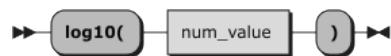


Returns the natural logarithm (base e) of the `num_value` provided as an argument.

5.3.4 log10

Logarithm to base 10.

Syntax:

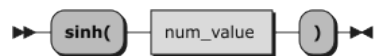


Returns the common logarithm (base 10) of the `num_value` provided as an argument.

5.3.5 sinh

Hyperbolic sine function.

Syntax:

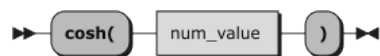


Returns the hyperbolic sine of the `num_value` provided as an argument.

5.3.6 cosh

Hyperbolic cosine function.

Syntax:

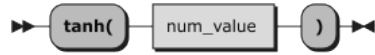


Returns the hyperbolic cosine of the `num_value` provided as an argument.

5.3.7 tanh

Hyperbolic tangent function.

Syntax:

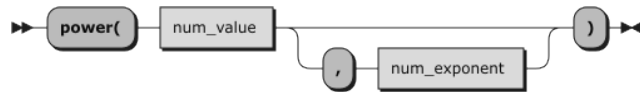


Returns the hyperbolic tangent of the `num_value` provided as an argument.

5.3.8 power

Power function.

Syntax:

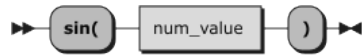


Returns the value of the first argument `num_value` raised to the power of the second argument `num_exponent`. The second argument is optional; if it is not provided, the function computes the square of the first argument.

5.3.9 sin

Sine function.

Syntax:

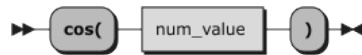


Returns the sine of the `num_value` (angle size) provided as an argument. The angle must be specified in radians.

5.3.10 cos

Cosine function.

Syntax:

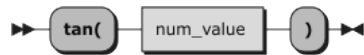


Returns the cosine of the `num_value` (angle size) provided as an argument. The angle must be specified in radians.

5.3.11 tan

Tangent function.

Syntax:

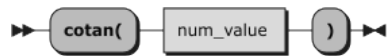


Returns the tangent of the **num_value** (angle size) provided as an argument. The angle must be specified in radians.

5.3.12 cotan

Cotangent function.

Syntax:



Returns the cotangent of the **num_value** provided as an argument. The angle must be specified in radians.

5.3.13 pi

Ludolph's number (π).

Syntax:



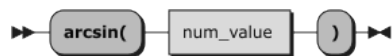
Method has no parameters.

Returns the mathematical constant π .

5.3.14 arcsin

Arcsine function.

Syntax:

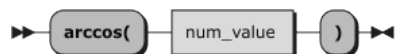


Returns the arcsine (inverse sine) of the **num_value** provided as an argument. The result is given in radians.

5.3.15 arccos

Arccosine function.

Syntax:



Returns the arcsine (inverse cosine) of the **num_value** provided as an argument. The result is given in radians.

5.3.16 arctan

Arctangent function.

Syntax:

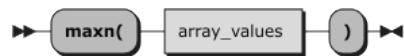


Returns the arctangent (inverse tangent) of the `num_value` provided as an argument. The result is given in radians.

5.3.17 maxn

Finds the maximum value in the given array.

Syntax:



The routine has one parameter `array_values`.

Returns the scalar value representing the maximum element of the array.

5.3.18 minn

Finds the minimum value in the given array.

Syntax:



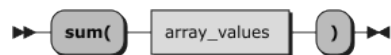
The routine has one parameter `array_values`.

Returns the scalar value representing the minimum element of the array.

5.3.19 sum

Sum of all values in the given array.

Syntax:



The routine has one parameter `arra_values`.

Returns the scalar value representing the sum of the elements of the array.

5.3.20 prod

Product of all values in the given array.

Syntax:



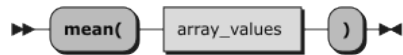
The routine has one parameter **array_values**.

Returns the scalar value representing the product of the elements of the array.

5.3.21 mean

Arithmetic mean of the values in the given array.

Syntax:



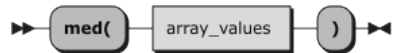
The routine has one parameter **array_values**.

Returns the scalar value representing the mean of the elements of the array.

5.3.22 med

Median of the values in the given array.

Syntax:



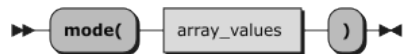
The routine has one parameter **array_values**.

Returns the scalar value representing the median of the elements of the array.

5.3.23 mode

Statistical mode of the values in the given array.

Syntax:



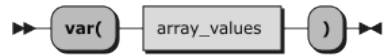
The routine has one parameter **array_values**.

Returns the scalar value representing the mode of the elements of the array.

5.3.24 var

Variance of the values in the given array.

Syntax:



The routine has one parameter `array_values`.

Returns the scalar value representing the variance of the elements of the array.

5.4 Routines for global options

5.4.1 set_ts

Sets the global computation step size. For example, this value is used by the `sim` method.

Syntax:



The input parameter is `num_sampleperiod`, specified in seconds.

No return value.