1.4.1 is_functional - Function

Package utility
Module functionals

Check if functional

Synopsis
is_functional(v)

Arguments

v	in	value to be tested Allowed:any

Returns
Bool

Description
Checks if the operand is a correct functional

Example

```
- a := dfs.gaussian1d()         # make a functional
- is_functional(a)              # is it one?
T
- a.done()
T
- is_functional(a)
F
-
```

1.4.2 functionalserver - Tool

Package utility
Module functionals

functionalserver tool

Constructors

functionals	Construct functionalserver tool

Functions

butterworth	Create an approximation of butterworth bandpass
chebyshev	Create a 1D Chebyshev polynomial function
combi	Create a combination function
compiled	Create your own function
compound	Create a compound function
done	free resources used by tool
evenpoly	Create an even 1D polynomial function
functional	Create the named functional
gaussian1d	Create a 1D Gaussian function
gaussian2d	Create a 2D Gaussian function
id	identification of tool
names	Show the known functional names
oddpoly	Create an odd 1D polynomial function
poly	Create a 1D polynomial function
type	type of tool

functionals - Constructor

Package	**utility**
Module	**functionals**
Tool	**functionalserver**

Construct functionalserver tool

Synopsis
functionals(host, forceneweserver)

Arguments

host	in	host on which to run tool Allowed:valid host name string Default: ”
forceneweserver	in	force the use of a new server Allowed:bool Default: F

Returns
functionalserver tool or fail

Description
Create a functionalserver `tool`on the specified host (or by default the host you are running on).

---

functionalserver.names - Function

Package	**utility**
Module	**functionals**
Tool	**functionalserver**

Show the known functional names

Synopsis
names()

Arguments

Returns
String array

Description

Return the known functional names.

Example

```
- dfs.names()
gaussian1d gaussian2d gaussian3d gaussianNd hyperplane polynomial
evenpolynomial oddpolynomial sinusoid1d chebyshev butterworth combine
compound compiled
```

---

functionalserver.functional - Function

Package	**utility**
Module	**functionals**
Tool	**functionalserver**

Create the named functional

Synopsis
functional(name, order, params)

Arguments

name	in	name of functional to create Allowed:string Default: unset
order	in	order if appropriate Allowed:integer Default: depending on functional
params	in	parameters Allowed:numeric array Default: depending on functional

Returns
Functional

Description

Create the named functional. If an order is specified it is used, otherwise the default functional is created with the specified (or default) parameters. Naming is minimum-match, case insensitive. See the separate *combine*, *compound* and *compiled* descriptions for more information on these special cases.

Name	Order	Default order	Default parameters
gaussian1d	-	-	[1 0 1]
gaussian2d	-	-	[1 0 0 1 1 0]
gaussian3d	-	-	[1 0 0 0 1 1 1 0 0]
gaussianNd	y	2	[0.159155 0 0 1 1 0] (note height)
hyperplane	y	0	[0]
polynomial	y	0	[0] (see separate poly)
evenpolynomial	y	0	[0]
odd polynomial	y	0	[0]
sinusoid1d	-	-	[1 1 0]
chebyshev	y	0	[0]
butterworth	-	0	[0]
combine	-	-	[] (all 1 after adds)
compound	-	-	[] (copied from added functions)
compiled	y	-	[...] (a list of zeroes: see separate compiled)

A created functional (or /em function for short, but be aware of possible *Glish* terminology clashes) will have a set of basic methods, which are described in the *functional* tool.

Example

```
- a:=dfs.functional('poly', 2, [1,2,3]) # create a polynomial
- a.f([0,1,2])                          # get some values
[1 6 17]
- a.fdf([0,1,2])                        # get values and derivatives
[[1:3,]
    1  1 0 0
    6  1 1 1
    17 1 2 4]
- b:=dfs.functional('combi')            # a combine
- b.add(a)                              # with the polynomial
T
- b.f([0,1,2])                          # values are same
[1 6 17]
- b.add(b)                              # add again
T
- b.f([0,1,2])                          # as expected from the combination
[2 12 34]
```

---

functionalserver.gaussian1d - Function

Package	**utility**
Module	**functionals**
Tool	**functionalserver**

Create a 1D Gaussian function

Synopsis
gaussian1d(height, center, width)

Arguments

height	in	amplitude of Gaussian, or 3 values Allowed:numeric Default: 1
center	in	center of Gaussian Allowed:numeric Default: 0
width	in	HPBW width of Gaussian Allowed:numeric Default: 1

Returns
Functional

Description

Create a 1-dimensional Gaussian with the specified height, width and center. The created functional has methods *f* and *fdf* to calculate the function value at a series of x values, or the value and the derivatives with respect to the parameters.

Example

```
# get the value and derivatives of a Gaussian with
# height=2; center at x=1; a width of 1 at x=[0,1,2]
- dfs.gaussian1d([2,1,1]).fdf([0,1,2])
[[1:3,]
    0.125 0.0625 -0.693147 0.693147
    2     1      0         0
    0.125 0.0625 0.693147  0.693147]
```

---

functionalserver.gaussian2d - Function

Package	**utility**
Module	**functionals**
Tool	**functionalserver**

Create a 2D Gaussian function

Synopsis
gaussian2d(params)

Arguments

params	in	amplitude, centers, major width, ratio, angle of Gaussian Allowed:numeric Default: [1 0 0 1 1 0]

Returns
Functional

Description

Create a 2-dimensional Gaussian with the specified height, width and center. The created functional has methods *f* and *fdf* to calculate the function value at a series of x values, or the value and the derivatives with respect to the parameters.

Example

```
# get the value and derivatives of a Gaussian with
# height=2; centers at x,y=1; widths of 1 at x,y=[0,0,1,1,2,2]
- dfs.gaussian2d([2,1,1,1,1,0]).fdf([0,0,1,1,2,2])
[[1:3,]
    0.0078125 0.00390625 -0.0433217 -0.0433217 0.0866434 0.0433217 -0
    2         1          0          0          0         0         -0
    0.0078125 0.00390625 0.0433217  0.0433217  0.0866434 0.0433217 -0]
```

---

functionalserver.poly - Function

Package	**utility**
Module	**functionals**
Tool	**functionalserver**

Create a 1D polynomial function

Synopsis
poly(order, params)

Arguments

order	in	polynomial degree Allowed:integer Default: 0
params	in	factors of terms Allowed:numeric Default: all 1

Returns
Functional

Description

Create a polynomial of specified degree. The default parameters are all 1. (Note that using the generic *functional* function the parameters are all set to zero).

Example

```
- dfs.poly(2).f([1,2])
[3 7]
- dfs.poly(2).state()
[type=5, order=2, ndim=1, npar=3, params=[1 1 1] ]
- dfs.poly(2).type()
polynomial
-
```

---

functionalserver.oddpoly - Function

Package	**utility**
Module	**functionals**
Tool	**functionalserver**

Create an odd 1D polynomial function

Synopsis
oddpoly(order, params)

Arguments

order	in	polynomial degree Allowed:integer Default: 0
params	in	factors of terms Allowed:numeric Default: all 1

Returns
Functional

Description

Create an odd polynomial of specified degree. The default parameters are all 1. (Note that using the generic *functional* function the parameters are all set to zero).

Example

```
- dfs.oddpoly(2).f([1,2])
[2 10]
- dfs.oddpoly(2).npar()
2
```

---

functionalserver.evenpoly - Function

Package	**utility**
Module	**functionals**
Tool	**functionalserver**

Create an even 1D polynomial function

Synopsis
evenpoly(order, params)

Arguments

order	in	polynomial degree Allowed:integer Default: 0
params	in	factors of terms Allowed:numeric Default: all 1

Returns
Functional

Description

Create an even polynomial of specified degree. The default parameters are all 1. (Note that using the generic *functional* function the parameters are all set to zero).

Example

```
- dfs.evenpoly(2).fdf([1,2])
[[1:2,]
    2 1 1
    5 1 4]
- dfs.evenpoly(2).state()
[type=6, order=3, ndim=1, npar=2, params=[1 1] ]
```

---

functionalserver.chebyshev - Function

Package	**utility**
Module	**functionals**
Tool	**functionalserver**

Create a 1D Chebyshev polynomial function

Synopsis
chebyshev(order, params, xmin, xmax, ooimode, def)

Arguments

order	in	order of the polynomial Allowed:integer Default: 0
params	in	the Chebyshev coefficients Allowed:numeric Default: all 0
xmin	in	the minimum value of the interval of interest Allowed:numeric Default: -1
xmax	in	the maximum value of the interval of interest Allowed:numeric Default: -1
ooimode	in	the "out-of-interval" mode. This controls what gets returned when an input value to the function is outside the interval of interest. Allowed:'constant' Default: a minimum match to the following values: 'constant' – the value of the def argument; 'zeroth' – the value of the zero-th order coefficient; 'extrapolate' – the function evaluated explicitly at the input value; 'cyclic' – the function evaluated at the input value after "folding" it into the interval of interest; 'edge' – the function evaluated at the nearest edge of the interval of interest
def	in	the default value to return for input values outside of the interval of interest when ooimode='constant' Allowed:numeric Default: 0

Returns
Functional

Description

This function creates a functional representing a Chebyshev series, a linear combination of so-called Chebyshev polynomials.

About Chebyshev Polynomials

Chebyshev polynomials are a special type of ultraspheric polynomials that are useful in such contexts as numerical analysis and circuit design. They form an orthogobnal set. A (type I) Chebyshev polynomial, T_n, is generated via the equation:

$$T_n(x) = \cos n(\arccos x) \tag{1.1}$$

Through clever use of trigometric identities, one can express T_n as a real polynomial expression of the form

$$T_n(x) = \sum_{i=0}^{n} C_i t^i \tag{1.2}$$

The low order polynomials look like this:

$$\begin{aligned} T_0 &= 1 && (1.3)\\ T_1 &= x && (1.4)\\ T_2 &= 2x^2 - 1 && (1.5)\\ T_3 &= 4x^3 - 3x && (1.6)\\ T_4 &= 8x^4 - 8x^2 + 1 && (1.7)\\ T_5 &= 16x^5 - 20x^3 + 5x && (1.8) \end{aligned}$$

Higher order polynomials satisfy the recurrance relation,

$$T_{n+1} = 2xT_n - T_{n-1}. \tag{1.9}$$

A common use of Chebyshev polynomials is in approximating functions. In particular, any function that is approximated by a power series,

$$f(x) \sim \sum P_i x^i, \tag{1.10}$$

over the interval [-1, 1] can be approximated by a linear combination of Chebyshev polynomials:

$$f(x) \sim \sum C_i T_i(x), \tag{1.11}$$

where C_i is the set of so-called Chebyshev coefficients.
Approximating a function with Chebyshev polynomials has some important advantages. For one, if the function is well approximated by a converging power series, one can obtain an equally accurate estimate using fewer terms of the corresponding Chebyshev series. More important, though, is the property over the interval [-1, 1], each polynomial has a domain of [-1, 1]; thus, the series is nicely bounded. And because of this bounded property,

approximations calculated from a Chebyshev series are less susceptible to machine rounding errors than the equivalent power series.

The "Interval of Interest"

Of course, in most real applications of these polynomials, one doesn't want to be limited to the [-1, 1] interval; thus, it is necessary to map the actual *interval of interest* into [-1, 1] before evaluating the series. The `chebyshev` functional supports this automatically when one sets the `xmin` and `xmax` arguments to this function. The mapping of values within this interval into [-1, 1] is done as follows:

$$x' = x - (\mathtt{min} + \mathtt{max})/2)/((\mathtt{max} - \mathtt{min})/2) \tag{1.12}$$

When the functional is evaluated via `f()`, the input values are first tranformed in the above way, and then the series is evaluated using the given coefficients. This is assuming the input value is within the interval of interest.
The behavior of the function when the value is outside the interval of interest is configurable via the `ooimode`. The choices are as follows:

constant the value returned is the value set with the `def` argument.

zeroth the value returned is the value of the zero-th order coefficient. Thus, if `params=[3,1,1]`, 3 is returned.

extrapolate the function is evaluated based on its coefficients just as it would be inside the interval. Thus, the function's range is not guaranteed to remain within the characteristic bounds of the Chebyshev interval. As the input value departs from the interval, the returned value diverges toward +/- infinity.

cyclic the function is evaluated as if the range is cyclic, repeating the range values from its canonical domain. The period of the cycle will be equal to $\mathtt{xmax} - \mathtt{xmin}$. When the function is evaluated outside this interval, the input value will shifted an integer number of periods until it falls within the Chebyshev interval; the value returned is the polynomial evaluated at the shifted value.

edge the value of the nearest edge of the interval of interest is returned.

The default out-of-interval mode is `constant`.

Example

```
- cheb := dfs.chebyshev(2, [2, 3, 4])
- cheb.state()
[type=9, order=2, ndim=1, npar=3, params=[2 3 4] , masks=[T T T] ,
mode=[interval=[-1 1] , default=0, intervalMode=constant]]
- cheb.f([0, 0.5, 1, 2, 5])
[-2 1.5 9 0 0]
- cheb := dfs.chebyshev(2, [2, 3, 4], xmin=0, xmax=2, ooimode='extrap')
[type=9, order=2, ndim=1, npar=3, params=[2 3 4] , masks=[T T T] ,
mode=[interval=[0 2] , default=0, intervalMode=extrapolate]]
- cheb.state()
- cheb.f([0, 0.5, 1, 2, 5])
[3 -1.5 -2 9 138]
```

---

functionalserver.butterworth - Function

Package	**utility**
Module	**functionals**
Tool	**functionalserver**

Create an approximation of butterworth bandpass

Synopsis

butterworth(minorder, maxorder, mincut, maxcut, center, peak)

Arguments

minorder	in	the order of the minimum (high-pass) cutoff of the curve Allowed:integer Default: 1
maxorder	in	the order of the maximum (low-pass) cutoff of the curve Allowed:integer Default: 1
mincut	in	the characteristic cutoff value of the minimum (high-pass) portion of the curve. Allowed:numeric Default: -1.0
maxcut	in	the characteristic cutoff value of the maximum (low-pass) portion of the curve. Allowed:numeric Default: 1.0
center	in	the location of the peak of the bandpass curve Allowed:numeric Default: 0.0
peak	in	the maximum value of the bandpass curve Allowed:numeric Default: 1.0

Returns
Functional

Description
This function creates a functional that simulates the (amplitude) transfer function for a wideband bandpass filter constructed from the combination of a low-pass and a high-pass Butterworth filter. It is not an exact representation. In particular is includes a discontinuity at the bandpass peak. Thus, care should be taken when evaluating derivatives of or fitting this function.
In analog electronic filter design, a Butterworth low-pass filter is one in which the amplitude transfer function, $|H(j\omega)|$ (where $j = \sqrt{-1}$ and ω is the angular frequency), is given by:

$$|H(j\omega)| = \frac{1}{\sqrt{1 + (\omega/\omega_c)^{2n}}} \tag{1.13}$$

where n refers to the filter "order" and ω_c is the "cutoff frequency". When $\omega = \omega_c$, the filter output is $1/\sqrt{2}$ that of the input, and the higher the order, the steeper the drop off towards zero and the closer the approximation to a idealized step function.
Filter theory provides transformations for deriving transfer functions of high-pass and band-pass filters which reflect how the electrical circuits actually work. However, to simplify this class's implementation and to make the

transfer function behavior more predictable by the naive user, this functional does not actually use the proper transformations. Instead, the Butterworth bandpass transfer function is approximated by low pass component, given above, combined with a pseudo high-pass function that is of the same form but with ω substituted with $-\omega$. Both components are shifted such that its peak transfer point is at a given "center" position.
The character of the band edges can be set independently for both ends of the passband. The cuttoff value–where the signal is attenuated to $1/\sqrt{2}$ of the peak–is set with the `mincut` and `maxcut` arguments for the low end (i.e. high-pass) and high end (i.e. low-pass) portions of the curve, respectively. This value is the value of ω_c for each side of the passband. The order of the attenuation curve–n in the above formula–can be set using `minorder` and `maxorder`.

Example

```
- butt := dfs.butterworth(minorder=4, maxorder=4, mincut=10,
                          maxcut=50, center=30, peak=1)
- butt.state()
[type=10, order=-1, ndim=1, npar=4, params=[30 10 50 1] , masks=[T T T T] ,
mode=[minOrder=4, maxOrder=4]]
- butt.f([0, 7, 10, 13, 30, 47, 50, 53, 60])
[0.193786 0.496351 0.707107 0.886488 1 0.886488 0.707107 0.496351 0.193786]
```

To plot this function...

```
- include 'pgplotter.g'
- pg := pgplotter()
- x := [-30:90]
- y := butt.f(x)
- pg.plotxy1(x, y, ylab='Bandpass')
```

---

functionalserver.combi - Function

Package	**utility**
Module	**functionals**
Tool	**functionalserver**

Create a combination function

Synopsis
combi()

Arguments

Returns
Functional

Description

Create a function consisting of a combination of one or more functions. Each partaking function will be defined as a normal function with its own parameters. The combination function will have parameters equal to the number of functions added. I.e. the added functions are considered encapsulated simple functions. Only functions with the same number of dimensions can be combined. See also *compound*.

Example

```
- a:=dfs.poly(2,[1,2,3])        # a polynomial
- a.f(1)                        # its value at x=1
6
- b:=dfs.gaussian1d(10, 0.5, 1) # a Gaussian with height, center, width
- b.f(1)
5
- c:=dfs.combi(); c.add(a); c.add(b)  # combine them
T
T
- c.f(1)                        # value is the sum (default parameters are 1)
11
- c.setpar(1, 2.5)              # set the first parameter of combi to 2.5
[2.5 1]
- c.f(1)                        # now value is 2.5*poly + 1*gauss
20
-
```

---

functionalserver.compound - Function

Package	**utility**
Module	**functionals**
Tool	**functionalserver**

Create a compound function

Synopsis
compound()

Arguments

Returns
Functional

Description

Create a function consisting of a combination of one or more functions, creating a compound function. The compound resulting functions is a function with the sum of all the parameters of the partaking functions. If a parameter of the compound function is changed, the parameter of the constituting function is changed as well.

Example

```
- a:=dfs.poly(2,[1,2,3])        # a polynomial
- a.f(1)                        # its value at x=1
6
- b:=dfs.gaussian1d(10, 0.5, 1) # a Gaussian with height, center, width
- b.f(1)
5
- c:=dfs.compound(); c.add(a); c.add(b) # compound them
T
T
- c.f(1)                        # result is sum
11
- c.state()                     # the function description
[type=11, order=-1, ndim=1, npar=6, params=[1 2 3 10 0.5 1] ,
    nfunc=2, funcs=[__*0=[type=5, order=2, ndim=1, npar=3,
                          params=[1 2 3] ],
                    __*1=[type=0, order=-1, ndim=1, npar=3,
                          params=[10 0.5 1] ]]]
- c.setpar(1, 2.5)              # set the first parameter of compound to 2.5
```

```
[2.5 2 3 10 0.5 1]
- c.f(1)                        # now value is 1.5 greater since zeroeth
                                # order of poly is now 2.5 i.s.o. 1
12.5
-
```

---

functionalserver.compiled - Function

Package	**utility**
Module	**functionals**
Tool	**functionalserver**

Create your own function

Synopsis
compiled(code, params)

Arguments

code	in	programmable expression string Allowed:string Default: ”
params	in	optionally parameters to be set Allowed:double array Default: unset

Returns
Functional or fail

Description

Create a function based on the programable string. The string should be a single expression, which can use the standard operators and functions and parentheses, having a single value as a result. The *parameters* of the function can be addressed with the p variable. This variable can be indexed in two ways. The first way is using the standard algebraic way, where the parameters are: p (or $p0$), $p1$, $p2$, ... The second way is by glish indexing, where the parameters are addressed as: $p[1]$, $p[2]$, ... The arguments are accessed in the same way, but using the variable name x. The compilation determines the number of dimensions and parameters of the produced function.

Operators are the standard operators (including comparisons, which produce a zero or one result; and conditional expression).
In addition to the standard expected functions, there is an *atan* with either one or two arguments (although *atan*2 exists as well), and *pi* and *ee* with no or one argument. The functional created behaves as all other functionals, and hence can be used in combinations.

Example

```
- a:=dfs.compiled('sin(pi(0.5) ) +pi'); # an example
- a.state()                             # and its result
[type=12, order=-1, progtext=sin(pi(0.5) ) +pi, ndim=0, npar=0, params=]
- a.f(0)                                # note that a value is
                                        # necessary to produce non-empty
                                        # result
4.14159
-  a.fdf([0])                           # no derivatives: sine no parameters
[1:1,]
    4.14159]
- b:= dfs.functional('compil','p*exp(-(x/p[2])^2') # try one to show error
SEVERE: Method define fails!
Illegal compiled expression:
No closing function paranethesis at:
'p*exp(-(x/p[2])^2'''
- b:= dfs.functional('compil','p*exp(-(x/p[2])^2)') # try again
        # Now a Gaussian with height
# and halfwidth 0
- b.parameters()
[0 0]
- b.setparameters([10 1]) # change to height 10 and
# halfwidth 1
10 1]
- b.f([-1,-0.5,0,.5,1])
[3.67879 7.78801 10 7.78801 3.67879]
- # the next one is sync(x), catering for x=0
- # using the fact that comparisons deliver values. Note
- # the extensive calculation to make sure no divison by 0
- synca:=
 dfs.compiled('( (x==0) * 1)+( (x!=0) * sin(x+(x==0)*1)/(x+(x==0)*1) )')
- synca.f([-1,0,1])
[0.841471 1 0.841471]
- sin(1)/1
0.841471
```

```
- # using conditional expressions:
- dfs.compiled('x==0 ? 1 : sin(x)/x').f([-1,0,1])
[0.841471 1 0.841471]
```

---

functionalserver.id - Function

Package	**utility**
Module	**functionals**
Tool	**functionalserver**

identification of tool

Synopsis
id()

Arguments

Returns
object id

Description
id will return the object identification. Its use is meant for the toolmanager.

Example

```
- dfs.id()
[sequence=1, pid=6053, time=1020745837, host=cetus, agentid=4]
```

---

functionalserver.type - Function

Package	**utility**
Module	**functionals**
Tool	**functionalserver**

type of tool

Synopsis
type()

Arguments

Returns
string

Description
type will return the tool name.

Example

```
- dfs.type()
functionals
```

---

functionalserver.done - Function

Package	**utility**
Module	**functionals**
Tool	**functionalserver**

free resources used by tool

Synopsis
done(kill)

Arguments

kill	in	force kill of the default tool Allowed:bool Default: F