```
- sol := ff.fit()                      # Linear fit all parameters solved for
- print sol
[2 3]
- ff.setparameters(1.2*sol)            # Update internal functional parameters
- print ff.fit(fixed=[F,T])            # Hold second parameter fixed  at wrong value
[-1.3 3.6]                             # and drive solution off for first parameter
```

functionfitter.type - Function

Package	**utility**
Module	**fitting**
Tool	**functionfitter**

Return tool type

Synopsis
type()

Returns
String

Description
Returns tool type (functionfitter).

Example

```
- include 'functionfitter.g'
- ff := functionfitter()
- ff.type()
functionfitter
```

1.3.2 fitter - Tool

Package	**utility**
Module	**fitting**

fitting tool

Constructors

fitter	Construct fitting tool

Functions

chi2	get the chi squared
constraint	get missing rank constraint equations
covariance	get the covariance matrix
deficiency	get missing rank
done	Remove the tool's resources
error	get errors in unknowns
fitavg	Get an average
fitpoly	Solve polynomial equations
fitspoly	Solve polynomial equationswith large x-range
fitted	Test Levenberg-Marquardt loop
fitter	Create a sub-tool
functional	Solve a general fitting problem
getstate	Obtain the state of the fitter
id	identification of tool
init	Initialize a fitting tool for use or re-use
linear	Solve a linear fitting problem
rank	get solution rank
reset	Reset the tool to its initialized state
sd	get standard deviation per unit weight
set	(Re-)initialize selected fitting tool properties
solution	get solution
stddev	get mean error (standard deviation per observation)
type	Obtain the type of the tool

A fitting `tool` has properties that can be set at creation; by the `init` method, by the `set` method and implicity by the use of the `linear` and `functional` methods. Properties are:

- n: the number of unknowns. In some specialized cases this property will be set automatically. All other properties have always to be set by the user.
- *type*: the type of condition equation:
  - *real*: n real factors for n real unknowns
  - *complex*: n complex factors for n complex unknowns
  - *conjugate*: $2n$ complex factors c, for n complex unknowns. The condition equations will be assumed to be:
    $c_0.z_0 + c_1.z_0^* + \ldots$

- *asreal*: n complex factors. The condition equations: $real(c_0).real(z_0) + i(imag(c_0).imag(z_0)) + \ldots$
- *separable*: $2n$ complex factors. Condition equations: $c_0.real(z_0) + c_1.imag(z_0) + \ldots$

The types can be given, in general, as minimum-match strings.

- $colfac$: collinearity factor (used in singular value decomposition to test dependencies)
- $lmfac$: Levenberg-Marquardt factor for non-linear solutions.

Example
A few basic examples: averaging a set of real numbers, and a set of complex numbers:

```
- include 'fitting.g'                                          # 1
T
- y := 0.1*1:5                                                 # 2
- f := dfs.compiled('p')                                       # 3
- dfit.linear(f,[],y);                                         # 4
T
- dfit.solution();                                             # 5
0.3
- dfit.getstate()                                              # 6
[n=1, typ=0, colfac=1e-08, lmfac=0.001, type=real]
- dfit.rank()                                                  # 7
1
- dfit.sd()
0.158114
- dfit.stddev()
0.158114
- dfit.chi2()
0.1
- dfit.covariance()                                            # 8
0.2
- dfit.error()
0.0707107
- dfit.set(type='complex');                                    # 9
T
- dfit.getstate()
[n=1, typ=1, colfac=1e-08, lmfac=0.001, type=complex]
- dfit.linear(dfs.compiled('p*x'), [1:6]*0 + 1, [1:6] + 1.5i*(1:6)) # 10
T
- dfit.solution()                                              # 11
3.5+5.25i
```

```
# a one step option:
- dfit.linear(dfs.poly(), 1:5, 1:5); dfit.solution()          # 12
T
3+0i
- dfit.set(type='real');                                         # 13
T
- dfit.linear(dfs.poly(), 1:5, 1:5); dfit.solution()          # 14
T
3
- dfit.set(type='complex')                                       #15
T
```

1. Gives access to the fitting and creates the default `fitting tool`. It also includes the `functionals tool`.
2. initialize the default fitter for 1 unknown (real by default)
3. Create normal equations from 5 (simple) condition equations:
   $x = i$
4. Solve the normal equations
5. Show the solution
6. what is the fitter's state?
7. the rank shows the number of dependencies. In this case there is no missing rank in the set of condition equations (i.e. they can be solved). The following functions show the information available after a solution. chi2 is the χ^2; sd the standard deviation per observed point; mu the standard deviation per unit weight.
8. the covariance and errors in the solved unknowns can be obtained. The errors are the covariance scaled with the standard deviation
9. Make a complex fitter (re-use) with two simultaneous sets of equations
10. Make normal equations (Note the multiple condition equations in call) Effectively the first argument (the coefficients of the parameter to be solved) is a vector of length 5, all elements value 1; the second is a matrix, with for each condition equation two values. The values are:

    ```
    - [1:5,2*(1:5)]+1i*(1:10)
    [1+1i 2+2i 3+3i 4+4i 5+5i 2+6i 4+7i 6+8i 8+9i 10+10i]
    ```

    where alternate values belong to the same set of observables
11. solve
12. show solution (Note the indices)

13. do a fit for functional polynomial of zeroth order

This example shows the general usage of the fitting tool for linear equations:

- create and/or initialize a fitter
- use one or more make() methods to create the normal equations
- use fit() to solve the equations
- use information methods to get the solution information

Some specialized fit methods are available to combine the above in one call. The above, simple, example, it is assumed that the data will fit a zero degree polynomial (i.e. just an average). It is just as easy to fit through the same data a higher degree polynomial. Let us first make the observed values:

```
- y := [1:10]
```

The simple way to fit an average here is (note we are still in complex type mode, but easy to correct):

```
- dfit.fitavg(y)
T
- dfit.solution()
5.5+0i
- dfit.set(type='real')
T
- dfit.fitavg(y)
T
- dfit.solution()
5.5
```

Another way is to make use of a zero degree polynomial, specified as $a = y$:

```
- x:=array(1,10)
- dfit.fitpoly(0,x,y)
T
- dfit.solution()
5.5
```

What would a 2-degree polynomial through the data look like
$(a + bx + cx^2 = y)$:

```
- dfit.fitpoly(2,1:10,y)
T
- dfit.solution()
[0 1 0]
```

Non-linear fitting can be done in the same way; with slightly different calls for the full-blown case. For the generic case just replace the `linear` with `functional`, and a non-linear method will be used, using the Levenberg-Marquardt method.
The major difference is that the functional provided as model should have initial parameter estimates set.
Note that, of course, you can write your own non-linear solution by providing an initial parameter estimate in the functional to be fitted, use the linear method, and upfate the parameters until some criterium is met.

Tools

fitter	fitting tool
functionfitter	Tool to do simple fitting of numeric data

Constructors

fitter	Construct fitting tool

Functions

chi2	get the chi squared
constraint	get missing rank constraint equations
covariance	get the covariance matrix
deficiency	get missing rank
done	Remove the tool's resources
error	get errors in unknowns
fitavg	Get an average
fitpoly	Solve polynomial equations
fitspoly	Solve polynomial equationswith large x-range
fitted	Test Levenberg-Marquardt loop
fitter	Create a sub-tool
functional	Solve a general fitting problem
getstate	Obtain the state of the fitter
id	identification of tool
init	Initialize a fitting tool for use or re-use
linear	Solve a linear fitting problem
rank	get solution rank
reset	Reset the tool to its initialized state
sd	get standard deviation per unit weight
set	(Re-)initialize selected fitting tool properties
solution	get solution
stddev	get mean error (standard deviation per observation)
type	Obtain the type of the tool

fitter - Constructor

Package	**utility**
Module	**fitting**
Tool	**fitter**

Construct fitting tool

Synopsis
fitter(n, type, colfac, lmfac, host, forcenewserver)

Arguments

n	in	number of unknowns Allowed:integer Default: 0
type	in	type of solution Allowed:coded value as string. Allowed: real, complex, separable, asreal, conjugate Default: 'real'
colfac	in	collinearity factor Allowed:positive float Default: 1e-8
lmfac	in	Levenberg-Marquardt factor Allowed:positive float Default: 1e-3
host	in	name of host to run on Allowed:string Default: ''
forcenewserver	in	is a new process needed Allowed:bool Default: F

Returns
fitting `tool` or fail

Description
Create a fitting `tool`. The tool can be created without arguments (in which case it is assumed to be a real fitter), or with the arguments specifying the number of unknowns to be solved for (a number not relevant in practice); and the type of solution: real, complex, conjugate (complex with both the unknown and its conjugate in the condition equations), separable complex, asreal complex with the real and imaginary part seen as independent

unknowns. All solutions need a model (specified as a functional. All solutions are done using an SVD type method. A collinearity factor can be specified, which is in essence the sine squared of the minimum angle between two normal equation columns that are still to be considered independent. For automatic non-linear solutions, a Levenberg-Marquardt factor (see Note 224) is used, which can be specified as well.
In the case of non-linear solutions that have to be handled by the system, an initial estimate for the model parameters is necessary.
A default fitter is always created, with an id that does not have to be specified. (see fitter method for sub constructors).

---

fitter.id - Function

Package	**utility**
Module	**fitting**
Tool	**fitter**

identification of tool

Synopsis
id()

Arguments

Returns
object id

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

Example

```
- dfit.id()
[sequence=1, pid=6266, time=1020746908, host=cetus, agentid=4]
```

---

fitter.type - Function

Package	**utility**
Module	**fitting**
Tool	**fitter**

Obtain the type of the tool

Synopsis
type()

Arguments

Returns
string

Description
Return the name of the type of the tool

Example

```
- include 'fitting.g'
- fit := fitter()
- fit.type()
fitter
- dfit.type()
fitter
```

---

fitter.fitter - Function

Package	**utility**
Module	**fitting**
Tool	**fitter**

Create a sub-tool

Synopsis
fitter(n, type, colfac, lmfac)

Arguments

n	in	number of unknowns Allowed:integer Default: 0
type	in	type of solution Allowed:coded value as string. Allowed: real, complex, separable, asreal, conjugate, with case-insensitive minimum-match Default: 'real'
colfac	in	collinearity factor Allowed:positive float Default: 1e-8
lmfac	in	Levenberg-Marquardt factor Allowed:positive float Default: 1e-3

Returns
fitter id

Description
Create a sub-tool (which can be used in the same way as a fitter `tool`). This function returns an identification, which has to be used in the *id* argument of subsequent calls. The call can specify the standard constructor arguments (n, type, colfac, lmfac), or can specify them later in a set statement.

Example

```
- include 'fitting.g'
- dfit.type()
fitter
- myfit := dfit.fitter(2)  # Create a fitter for 2 real unknowns
- dfit.fitpoly(1, 1:5, 1:5, id=myfit) # solve x+y=1; x+2y=2; ... x+5y=5
T
- print dfit.solution(id=myfit), dfit.error(id=myfit) #  show solution
[0 1]  [0 0]
```

---

fitter.done - Function

Package	**utility**
Module	**fitting**
Tool	**fitter**

Remove the tool's resources

Synopsis
done(id, kill)

Arguments

id	in	the id of a sub-tool Allowed:fitter id Default: unset
kill	in	force kill of the default tool Allowed:bool Default: F

Returns
bool

Description
Remove the tool's resources. If no id specified the complete tool is removed. If the tool is the default tool (*dfit*) the done will only be executed if the kill argument is true.

Example

```
- include 'fitting.g'
- fit := fitter()
- id1 := fit.fitter()
- id2 := fit.fitter()
- fit.getstate(id1)
[n=0, typ=0, colfac=1e-06, lmfac=0.001, type=real]
- fit.getstate(id2)
[n=0, typ=0, colfac=1e-06, lmfac=0.001, type=real]
- fit.getstate()
[n=0, typ=0, colfac=1e-06, lmfac=0.001, type=real]
```

```
- fit.done(id1)
T
- fit.getstate(id2)
[n=0, typ=0, colfac=1e-06, lmfac=0.001, type=real]
- fit.getstate(id1)
SEVERE: Illegal fitter id in getstate
<fail>: Illegal fitter id in getstate
        File:   note.g, Line 58
        Stack:  throw()
                .()
- fit
[real=0 ... ]
- fit.done()
T
- fit
F
```

---

fitter.reset - Function

Package	**utility**
Module	**fitting**
Tool	**fitter**

Reset the tool to its initialized state

Synopsis
reset(id)

Arguments

id	in	the id of a sub-tool Allowed:fitter id Default: unset

Returns
bool

Description
Reset the tool's resources to its initialized state.

Example

```
- include 'fitting.g'
- fit := fitter()
- fit.init(2)
T
- fit.getstate()
[n=2, typ=0, colfac=1e-08, lmfac=0.001, type=real]
- fit.reset()
T
- fit.getstate()
[n=2, typ=0, colfac=1e-08, lmfac=0.001, type=real]
```

---

fitter.init - Function

Package	**utility**
Module	**fitting**
Tool	**fitter**

Initialize a fitting tool for use or re-use

Synopsis
init(n, type, colfac, lmfac, id)

Arguments

n	in	number of unknowns Allowed:integer Default: unset
type	in	type of solution Allowed:coded value as string. Allowed: real, complex, separable, asreal, conjugate, with case-insensitive minimum-match Default: unset
colfac	in	collinearity factor Allowed:positive float Default: unset
lmfac	in	Levenberg-Marquardt factor Allowed:positive float Default: unset
id	in	the id of a sub-tool Allowed:fitter id Default: 0

Returns
bool

Description
Initialize the fitting tool (or the sub-tool if an id is specified). Like in the constructor, the number of unknowns to be solved for; the type (as code or as case insensitive minimum-match string); and the collinearity and Levenberg-Marquardt factor can be specified (see fitter for all options). Individual values can be overwritten with the set function.

Example

```
- include 'fitting.g'
- fit := fitter()
- fit.init(5,'r')
T
- fit.getstate()
[n=5, typ=0, colfac=1e-08, lmfac=0.001, type=real]
```

---

fitter.set - Function

Package	**utility**
Module	**fitting**
Tool	**fitter**

(Re-)initialize selected fitting tool properties

Synopsis
set(n, type, colfac, lmfac, id)

Arguments

n	in	number of unknowns Allowed:integer Default: unset
type	in	type of solution Allowed:coded value as string. Allowed: real, complex, separable, asreal, conjugate, with case-insensitive minimum-match Default: unset
colfac	in	collinearity factor Allowed:positive float Default: unset
lmfac	in	Levenberg-Marquardt factor Allowed:positive float Default: unset
id	in	the id of a sub-tool Allowed:fitter id Default: 0

Returns
bool

Description
Set selected properties of the fitting tool. All *unset* properties remain the same (in the init method all properties are (re-)initialized). Like in the constructor, the number of unknowns to be solved for; the number of simultaneous solutions; the type (as code or as case insensitive minimum-match string); and the collinearity and Levenberg-Marquardt factor can be specified.

Example

```
- include 'fitting.g'
- fit := fitter()
- fit.init(5,'r')
```

```
T
- fit.getstate()
[n=5, typ=0, colfac=1e-08, lmfac=0.001, type=real]
- fit.set(type='compl')
T
- fit.getstate()
[n=5, typ=1, colfac=1e-08, lmfac=0.001, type=complex]
```

---

fitter.getstate - Function

Package	**utility**
Module	**fitting**
Tool	**fitter**

Obtain the state of the fitter

Synopsis
getstate(id)

Arguments

id	in	the id of a sub-tool Allowed:fitter id Default: 0

Returns
record with state

Description
Obtain the state of the fitter `tool` or a sub-tool.

Example

```
- include 'fitting.g'
- fit := fitter(); # a fitter with default properties
- fit.getstate() # what is it like
[n=0, typ=0, colfac=1e-08, lmfac=0.001, type=real]
- fit.init(5,'r') # initialize it
```

```
T
- id := fit.fitter() # a sub-fitter, initialized
- fit.set(type='complex',id=id) # a sub-fitter, initialized
- fit.getstate() # state of default fitter
[n=5, typ=0, colfac=1e-08, lmfac=0.001, type=real]
- fit.getstate(id=id) # state of the sub-fitter
[n=1, m=1, typ=1, colfac=1e-06, lmfac=0.001, type=complex]
```

---

fitter.linear - Function

Package	**utility**
Module	**fitting**
Tool	**fitter**

Solve a linear fitting problem

Synopsis
linear(fnct, x, y, sd, wt, id)

Arguments

fnct	in	functional to be fitted Allowed:a functional Default: unset
x	in	the x coordinate values Allowed:vector
y	in	the ordinate values Allowed:vector
sd	in	standard deviation of equations Allowed:one or more values used cyclically Default: unset
wt	in	an optional alternate for sd Allowed:one or more values used cyclically Default: 1
id	in	the id of a sub-tool Allowed:fitter id Default: 0

Returns
bool or fail

Description

linear will make a linear least squares solution for the points through the ordinates at the x values, using the specified fnct. The x can be of any dimension, depending on the number of arguments needed in the functional evaluation. The values should be given in the order: $x0[1]$, $x0[2]$, ..., $x1[1]$, ..., $xn[m]$ if there are n observations, and m arguments. x should be a vector of $m * n$ length; y (the observations) a vector of length n.

Example

```
- include 'fitting.g'                   # create default fitter
- x := [1:10];                          # some x values
- y := 3*x*x;                           # some y values
- dfit.linear(dfs.poly(2), x, y);       # solve 2nd order polynomial
T
- dfit.solution();
[0 0 3]
- include 'functionals.g'               # allow functionals
- alin := function() { # encapsulate some code
z:=dfs.compiled('p0+p1*x0+p2*x1')       # a 2-d functional
x:=[1,2,3,4,5,6]                        # 3 observations
z.setparameters([1,2,3]) # set parameters to get ys
print 'y=', y:=z.f(x)
dfit.linear(z,x,y) # fit
print 'linear:', dfit.solution()
z:=dfs.compiled('p0+p1*x0')             # try a 1-d funtional
x:=[1,3,5]
z.setparameters([1,2])
print 'y=', y:=z.f(x)
dfit.linear(z,x,y)
print 'linear:', dfit.solution()
z:=dfs.compiled('6+p0') # try a 0-d functional
x:=[]
print 'y=', y:=z.f([1:3])-2
dfit.linear(z,x,y)
print 'linear:', dfit.solution()
return T
}
- alin()
y= [9 19 29]
linear: [1 2 3]
y= [3 7 11]
linear: [1 2]
y= [4 4 4]
```

```
linear: -2
T
```

---

fitter.functional - Function

Package	**utility**
Module	**fitting**
Tool	**fitter**

Solve a general fitting problem

Synopsis

functional(fnct, x, y, sd, wt, mxiter, id)

Arguments

fnct	in	functional to be fitted Allowed:a functional Default: unset
x	in	the x coordinate values Allowed:vector
y	in	the ordinate values Allowed:vector
sd	in	standard deviation of equations Allowed:one or more values used cyclically Default: unset
wt	in	an optional alternate for sd Allowed:one or more values used cyclically Default: 1
mxiter	in	the maximum number of iterations Allowed:integer Default: 50
id	in	the id of a sub-tool Allowed:fitter id Default: 0

Returns

bool or fail

Description

`functional` will make a non-linear least squares solution for the points through the ordinates at the x values, using the specified `fnct`. Details can be found in the `linear` description.

Example

```
- include 'fitting.g'                   # create default fitter
- x := [1:10];                          # some x values
- y := 3*x*x;                           # some y values
- dfit.functional(dfs.poly(2), x, y);   # solve 2nd order polynomial
T
- dfit.solution();
[-1.10049e-15 7.44482e-16 3]
- include 'functionals.g'               # allow functionals
- anonlin := function() { # encapsulate some code
z:=dfs.compiled('p0+p1*x0+p2*x1')       # a 2-d functional
x:=[1,2,3,4,5,6]                        # 3 observations
z.setparameters([1,2,3]) # set parameters to get ys
print 'y=', y:=z.f(x)
dfit.functional(z,x,y) # fit
print 'non-linear:', dfit.solution()
z:=dfs.compiled('p0+p1*x0')             # try a 1-d funtional
x:=[1,3,5]
z.setparameters([1,2])
print 'y=', y:=z.f(x)
dfit.functional(z,x,y)
print 'non-linear:', dfit.solution()
z:=dfs.compiled('6+p0') # try a 0-d functional
x:=[]
print 'y=', y:=z.f([1:3])-2
dfit.functional(z,x,y)
print 'non-linear:', dfit.solution()
return T
}
- anonlin()
y= [9 19 29]
non-linear [1 2 3]
y= [3 7 11]
non-linear [1 2]
y= [4 4 4]
non-linear -2
T
```

---

fitter.fitavg - Function

Package	**utility**
Module	**fitting**
Tool	**fitter**

Get an average

Synopsis
fitavg(y, sd, wt, id)

Arguments

y	in	observed values of the condition equations Allowed:$N \times m$ values
sd	in	standard deviation of equations Allowed:one or more values used cyclically Default: unset
wt	in	an optional alternate for sd Allowed:one or more values used cyclically Default: 1
id	in	the id of a sub-tool Allowed:fitter id Default: 0

Returns
bool

Description
`fitavg` will calculate the average of a set of values.
The method expects that the properties of the fitter to be used have been initialized or set (like the number of simultaneous solutions m; the type; factors). The main reason is to limit the number of parameters on the one hand, and on the other hand not to depend on the actual array structure in Glish to get the variables and type. The number of unknowns is by definition 1.

Example

```
- include 'fitting.g'
- fit := fitter(); # a fitter for complex
```

```
- fit.set(type='com')
- fit.getstate() # what is it like
[n=0, typ=1, colfac=1e-08, lmfac=0.001, type=complex]
- fit.fitavg(1:6 + (2:7)*1i) # equations are x=1+2i; x=2+3i; ... x=6+7i
T
- fit.getstate() # what is it like now: note n=1
[n=1, typ=1, colfac=1e-08, lmfac=0.001, type=complex]
- print fit.solution(), fit.error() #  show solution
3.5+4.5i 0.763763+0.763763i
```

---

fitter.fitpoly - Function

Package	**utility**
Module	**fitting**
Tool	**fitter**

Solve polynomial equations

Synopsis
fitpoly(n, x, y, sd, wt, id)

Arguments

n	in	the order of the polynomial to solve for Allowed:integer
x	in	the parameter for the polynomial calculations Allowed:one or more real or complex values
y	in	observed values of the condition equations Allowed:$N \times m$ values
sd	in	standard deviation of equations Allowed:one or more values used cyclically Default: unset
wt	in	an optional alternate for sd Allowed:one or more values used cyclically Default: 1
id	in	the id of a sub-tool Allowed:fitter id Default: 0

Returns
bool

Description

`fitpoly` will create normal equations from the specified condition equations, and solve the resulting normal equations. In essence a combination of makepoly and fit.

The method expects that the properties of the fitter to be used have been initialized or set (like the number of simultaneous solutions m; the type; factors). The main reason is to limit the number of parameters on the one hand, and on the other hand not to depend on the actual array structure in Glish to get the variables and type.

Example

```
- include 'fitting.g'
- fit := fitter(); # a fitter for real unknowns
- x:=[1:10] # we have values at 10 'x' values
- y:=2+0.5*x-0.1*x^2 # which are 2 +0.5x -0.1x^2
- fit.fitpoly(3,x,y)  # fit a 3-degree polynomial
T
- print fit.solution(), fit.error() #  show solution and their errors
[2 0.5 -0.1 -6.21725e-14]  [0 0 0 0]
```

Note the big problem with polynomial fitting if you have relatively large x values: the internal precision of computers start to play a role. In general it is worth the effort to normalize x between -1 and 1. To aid in most standard cases, a function *fitspoly* has been provided.

---

fitter.fitspoly - Function

Package	**utility**
Module	**fitting**
Tool	**fitter**

Solve polynomial equationswith large x-range

Synopsis

fitspoly(n, x, y, sd, wt, id)

Arguments

n	in	the order of the polynomial to solve for Allowed:integer
x	in	the parameter for the polynomial calculations Allowed:one or more real or complex values
y	in	observed values of the condition equations Allowed:$N \times m$ values
sd	in	standard deviation of equations Allowed:one or more values used cyclically Default: unset
wt	in	an optional alternate for sd Allowed:one or more values used cyclically Default: 1
id	in	the id of a sub-tool Allowed:fitter id Default: 0

Returns
bool

Description
`fitspoly` will create normal equations from the specified condition equations, and solve the resulting normal equations. In essence a combination of makepoly and fit.
The method expects that the properties of the fitter to be used have been initialized or set (like the number of simultaneous solutions m; the type; factors). The main reason is to limit the number of parameters on the one hand, and on the other hand not to depend on the actual array structure in Glish to get the variables and type. Before fitting the x-range is normalized to values less than 1 to cater for large difference in x raised to large powers. Later a shift to make x around zero will be added as well.

Example

```
- include 'fitting.g'
- fit := fitter(); # a fitter for real unknowns
- x:=[1:10] # we have values at 10 'x' values
- y:=2+0.5*x-0.1*x^2 # which are 2 +0.5x -0.1x^2
- fit.fitspoly(3,x,y)  # fit a 3-degree polynomial
T
- print fit.solution(), fit.error() #  show solution and their errors
[2 0.5 -0.1 1.05116e-15]  [0 0 0 0]
```

---

fitter.solution - Function

Package	**utility**
Module	**fitting**
Tool	**fitter**

get solution

Synopsis
solution(id)

Arguments

id	in	the id of a sub-tool Allowed:fitter id Default: 0

Returns
array of n or $n \times m$ real or complex values

Description
Obtain the solution of a fit

Example

```
- include 'fitting.g'
- fit := fitter(n=1) # a fitter for one real unknown
- fit.linear(dfs.functional('hyper',1),1+0*1:6, 1:6)
# equations are x=1; x=2; ... x=6
T
- print fit.solution(), fit.error() #  show solution and error in average
3.5 0.763763
```

---

fitter.rank - Function

Package	**utility**
Module	**fitting**
Tool	**fitter**

get solution rank

Synopsis
rank(id)

Arguments

id	in	the id of a sub-tool Allowed:fitter id Default: 0

Returns
integer

Description
Obtain the rank (in SVD sense) of a fit. The `constraint` method will show the equations that are orthogonal to the existing ones, and which will make the solution possible.

Example

```
- include 'fitting.g'
- fit := fitter(n=1); # a fitter for one real unknown
- fit.linear(dfs.functional('hyper',1),1+0*1:6, 1:6)
# equations are x=1; x=2; ... x=6
T
- fit.rank() # actual rank
1
```

---

fitter.deficiency - Function

Package	**utility**
Module	**fitting**
Tool	**fitter**

get missing rank

Synopsis

deficiency(id)

Arguments

id	in	the id of a sub-tool Allowed:fitter id Default: 0

Returns
integer

Description
Obtain the missing rank (in SVD sense) of a fit. The `constraint` method will show the equations that are orthogonal to the existing ones, and which will make the solution possible.

Example

```
- include 'fitting.g'
- fit := fitter(n=1); # a fitter for one real unknown
- fit.linear(dfs.functional('hyper',1),1+0*1:6, 1:6)
# equations are x=1; x=2; ... x=6
T
- fit.deficiency() # missing rank
1
```

---

fitter.constraint - Function

Package	**utility**
Module	**fitting**
Tool	**fitter**

get missing rank constraint equations

Synopsis
constraint(id)

Arguments

id	in	the id of a sub-tool Allowed:fitter id Default: 0

Returns
integer

Description
Obtain the set of orthogonal equations that make the solution of the rank deficient normal equations possible.

Example

```
- include 'fitting.g'# use the default fitter
- fit.linear(dfs.functional('hyper',2),0*(1:10)+1,1:5)
# equations are: x+y=1; x+y=2; ...
- fit.rank() # there is rank deficiency
1
- fit.deficiency()
1
- fit.constraint() # constraint equation used
[-1 1]
- fit.solution() # the solution (as expected)
[1.5 1.5]
```

---

fitter.chi2 - Function

Package	**utility**
Module	**fitting**
Tool	**fitter**

get the chi squared

Synopsis
chi2(id)

Arguments

id	in	the id of a sub-tool Allowed:fitter id Default: 0

Returns
vector of m values

Description
Obtain the chi squared of a fit

Example

```
- include 'fitting.g'
- fit := fitter(n=1); # a fitter for one real unknown
- fit.linear(dfs.functional('hyper',1),1+0*1:6, 1:6)
# equations are x=1; x=2; ... x=6
T
- fit.chi2() # chi squared
17.5
```

---

fitter.sd - Function

Package	**utility**
Module	**fitting**
Tool	**fitter**

get standard deviation per unit weight

Synopsis
sd(id)

Arguments

id	in	the id of a sub-tool Allowed:fitter id Default: 0

Returns
m real values

Description
Obtain the standard deviation per unit of weight of a fit

Example

```
- include 'fitting.g'
- fit := fitter(n=1); # a fitter for one real unknown
- fit.linear(dfs.functional('hyper',1),1+0*1:6, 1:6)
# equations are x=1; x=2; ... x=6
- fit.sd()
1.87083
```

---

fitter.stddev - Function

Package	**utility**
Module	**fitting**
Tool	**fitter**

get mean error (standard deviation per observation)

Synopsis
stddev(id)

Arguments

id	in	the id of a sub-tool Allowed:fitter id Default: 0

Returns
m real values

Description
Obtain the standard deviation per condition equation of a fit

Example

```
- include 'fitting.g'
- fit := fitter(n=1); # a fitter for one real unknown
- fit.linear(dfs.functional('hyper',1),1+0*1:6, 1:6)
# equations are x=1; x=2; ... x=6
T
- fit.stddev()
1.87083
```

---

fitter.covariance - Function

Package	**utility**
Module	**fitting**
Tool	**fitter**

get the covariance matrix

Synopsis
covariance(id)

Arguments

id	in	the id of a sub-tool Allowed:fitter id Default: 0

Returns
$n \times n$ matrix

Description
Obtain the covariance matrix of a fit

Example

```
- include 'fitting.g'
- fit := fitter(n=1); # a fitter for one real unknown
- fit.linear(dfs.functional('hyper',1),1+0*1:6, 1:6)
```

```
# equations are x=1; x=2; ... x=6
T
- fit.covariance()
0.166667
```

---

fitter.error - Function

Package	**utility**
Module	**fitting**
Tool	**fitter**

get errors in unknowns

Synopsis
error(id)

Arguments

id	in	the id of a sub-tool Allowed:fitter id Default: 0

Returns
$m \times n$ matrix

Description
Obtain the errors in the unknowns of a fit

Example

```
- include 'fitting.g'
- fit := fitter(n=1); # a fitter for one real unknown
- fit.linear(dfs.functional('hyper',1),1+0*1:6, 1:6)
# equations are x=1; x=2; ... x=6
T
- fit.error()
0.763763
```

---

fitter.fitted - Function

Package	**utility**
Module	**fitting**
Tool	**fitter**

Test Levenberg-Marquardt loop

Synopsis
fitted(id)

Arguments

id	in	the id of a sub-tool Allowed:fitter id Default: 0

Returns
bool

Description
Test if enough Levenberg-Marquardt loops have been done. It returns T (true) if no improvement possible.

Example

1.4 functionals - Module

Functionals handling
include "functionals.g"

Description

Introduction
A functional is a function with parameters, defined as $f(p; x)$, where p are the parameters, and x the arguments. Methods are available to calculate the value of a function for a series of argument values for the given set of parameters, and for the automatic calculation of the derivatives with respect to the parameters.
The created functionals can be used in the fitting `tool` or in any other `tool` that needs to have generic function values or their derivatives.
A functional has a mask associated with it, to indicate if certain parameters have to be solved for. See masks for details.
To access the `functionals` module, include the *functionals.g Glish* script. This will create a default Functionalserver `tool` called **dfs**.

```
- include 'functionals.g'
- dfs.type()             # Default tool created for you
functionals
```

The Functionalserver `tool` has no state (it doesn't remember anything), so there is generally no need to create your own `tool`; just use the default one. However, if you do require one, perhaps in a script, then use the `functionals` constructor to make one.

```
- include 'functionals.g';
- myqt := functionals();
- myqt.type();
functionals
```

The basic purpose of the Functionalserver `tool` is to create Functional `tools`. You do not create Functional `tools` directly (the Functional `tool` has no constructor), you always create them via the Functionalserver `tool`.
Functionals are created in a variety of ways, in general by specifying the name of the functional, together with some necessary information like e.g. the order of a polynomial, or the code needed to compile your privately defined function. Parameters can be set at creation time or later.

```
- include 'functionals.g'
- a := dfs.gaussian1d()            # creates a 1D Gaussian, default arguments
- b := dfs.functional('gaussian1') # creates the same one
```

```
- a.state()                         # the 'state' of the functional
[type=0, order=-1, ndim=1, npar=3, params=[1 0 1] ]
- a.type()                          # its type
gaussian1d
- a.ndim()                          # its dimension (number of arguments)
1
- a.npar()                          # its number of parameters
3
- b.state()
[type=0, order=-1, ndim=1, npar=3, params=[1 0 1] ]
- a.f(1);                           # the value at x=1
0.0625
- a.fdf([0,0.5]);                   # value and derivatives
[[1:2,]
    1   1   0       0
    0.5 0.5 1.38629 0.693147]
```

In some cases an *order* can be specified as well (e.g. for polynomials):

```
- include 'functionals.g'
- a := dfs.poly(3)                # creates a 3rd order polynomial
- b := dfs.functional('polyn',3) # creates the same one, but with
                                  # original defaults
- a.state()
[type=5, order=3, ndim=1, npar=4, params=[1 1 1 1] ]
- b.state()
[type=5, order=3, ndim=1, npar=4, params=[0 0 0 0] ]
```

An extremely valuable aspect of the Functionals module is the ability to create a functional from a compiled string specifying an arbitrary function. For example, let us make our own polynomial $1 + 2 * x + 3 * x^2$ and evaluate it at a few abcissa locations

```
- include 'functionals.g'
- a := dfs.compiled ('p0 + p1*x + p2*x*x', [1,2,3])   # Define
- a.f([0,10,20])                                      # Evaluate at x=[0,10,20]
[1 321 1241]
```

The functions created can also be used to specify the function to be fitted in a least squares fit (see the fitting `tool`).

Functions

is_functional	Check if functional

Tools

functional	functional tool
functionalserver	functionalserver tool