memory(verbose)

Returns
Number of warnings

Description
The physical memory is checked to verify that it is ¿ 63MB.

Arguments

verbose	Be verbose? Allowed:Boolean Default: T

1.3 fitting - Module

Fitting handling
include "fitting.g"

Description

The fitting module provides least squares fitting. It can handle linear and non-linear; real and complex (including cases where unknowns are each other's conjugate); complete and singular-value-decomposition; with or without external constraints; general or specific cases.
Two tools are currently provided. The Functionfitter `tool` is a high-level tool that enables simple fitting of user-given functional forms to data. It is built on top of the rich, low-level Fitting which enables a wide range of fitting problems to be handled.
First we show an example using the high-level Functionfitter `tool` which makes fitting easy. Of course you lose some flexibility with this `tool`.

Functionfitter tool
Access the Functionfitter `tool` by including the *functionfitter.g* Glish script. This `tool` enables you to specify the functional form directly with a string (or you can pass in a functional `tool` if you wish).

Example

```
- include 'randomnumbers.g'
- n := 20
- x := 1:n                             # Generate a straight line
- y := 2 + 3*x
- r := randomnumbers()                 # Make some noise
- g := r.normal(0,0.1,n)
- y +:= g
- yerr := y / 20.0                     # Fudge some y errors
-
- include 'functionfitter.g'           # Generate default dff tool
- dff.setdata (x, y, yerr)             # Set data
- dff.setfunction ('p0 + p1*x')        # Set function to fit; solves for parameters p0 & p1
- dff.fit(linear=T)                    # Do linear fit (no guess needed)
[1.82008 3.01639]
- dff.geterror()                       # Errors in parameters
[0.116115 0.00969311]
- dff.plot(model=F)                    # Plot data and fit
```

Fitting tool

To access the `fitting` module and its `tools`, include the *fitting.g Glish* script. It will load the fitting interface, and create the *dfit* default fitting `tool`.
A `fitting tool` can be reused for other solutions.
More fitting tools can be created by either the fitter constructor (which creates and returns a separate fitting `tool`), or by the fitter method of an existing fitting `tool`, which returns a fit identifier, which can be used to indicate a specific sub-fitter in the fitter used by including a parameter 'id=' in all calls to the fitting `tool`'s functions. The latter is especially useful in the case where many simultaneous solutions are necessary: it is more resource efficient, and also allows you to have an array of fit indices to loop over. In both cases the parameters of the tool can be given in the constructor (fitter method), or in a separate `init` method (see next example of the highest level use):

```
- include 'fitting.g';
- myfit := fitter(); # general fitting tool created
-# (needs initializing before it can be used)
- myfit.type(); # the type of the tool
fitter
- cpid := cpfit.fitter(type='complex') # and another (sub-)fitter
-                                      # with an id
```

The theory behind the fitting module's operation is described in detail in Note 224.
Fitting requires a model describing the data obtained. The model is a described as a functional with parameters to be solved for. Functionals can be

pre-programmed functionals like *poly*, *gauss1d*, or free form like *compiled.* In the latter case an expression string describes the model.
The model can depend on zero, one or more arguments, called x. The number of arguments determines the dimension of the model.
Fitting also needs a set of *data*, called y. If the model is not 0-dimensional, each value x will have an observed value. E.g. for each hour of the day x you can have a measured temperature. Or in the case of multi-dimensions e.g. each pair of hour of the day at each height above the surface (x_0, x_1) you should have a y value.

Example
Simple linear example
The example uses a set of x coordinates:

```
x := -1 + 0.1*[0:20]
```

The 'observed' values used are a simple 1-dim polynomial of order 2:

```
1 + 2(x+1) + 0.03(x+1)^2 == 3.03 +2.06x + 0.03x^2
```

We fill these values using the polynomial functional:

```
y := dfs.poly(2, [3.03, 2.06, 0.03]).f(x)
```

To take the average of these points, we can do:

```
dfit.linear(dfs.compiled('p'),[],y)
T
```

Note that an expression uses p (which is $p0$), $p1$ and x, $x0$, $x1$. Alternatively you can use $x[1]$ (for $x0$) and $x[1]$. Note also that since no argument is used in the expression, no x-values have to be given.
We can get solutions and errors from these data (see for details the separate routine descriptions):

```
dfit.solution()
3.041
# Compare with the result:
sum(y)/len(y)
3.041
dfit.sd() # standard deviation per observation
1.27824
dfit.stddev() # standard deviation per weight
1.27824
dfit.error() # errors in solved parameters
0.278934
dfit.rank() # rank of solution
1
dfit.covariance() # covariance matrix
0.047619
```

We can also try to use a 0-order polynomial. Note that a polynomial, even a zero-order one, is a 1-dim function, and we need an x defined:

```
dfit.linear(dfs.poly(0),[],y)
SEVERE: Method linear fails!
Linear fitters x and y lengths disagree
<fail>: Method linear fails!
Linear fitters x and y lengths disagree
        File:   servers.g, Line 1009
        Stack:    .(), ./fitting.g line 635
                  .()
fit.linear(dfs.poly(0),x,y)
T
dfit.solution()
3.041
```

We would like to chack the results, so we will do an average in a separate fitter:

```
id := dfit.fitter() # get a new fitter
dfit.linear(dfs.compiled('p'),[],y, id=id) # get average
T
dfit.solution() - dfit.solution(id=id) # check difference
-4.44089e-16
# to really show we recalculate and check separately:
dfit.linear(dfs.compiled('p'), [], y/2, id=id) # calculate new average
dfit.solution() - dfit.solution(id=id)
1.5205
```

A 1-order polynomial is now easy:

```
dfit.linear(dfs.poly(1),x,y)
dfit.solution()
[3.041 2.06]
dfit.chi2()
0.00201894
dfit.error()
[0.00224944 0.00371484]
dfit.sd()
0.0103082
```

Note that each 'equation' can also be given a weight or standard deviation.

2-dimensional example
A 2-dim model is done the same way. The x vector has now n pairs of values. The Glish *rbind* can help in cxreating these pairs.

```
x1 := 1:5
x2 := 0.1*1:5
x1 := rbind(x1,x2) # combine into pairs. Check:
```

```
[x1]
[1 0.1 2 0.2 3 0.3 4 0.4 5 0.5]
dfit.linear(dfs.compiled('p*x + p1*sin(x1)'),x1,
+ dfs.compiled('3*x+7*sin(x[2])').f(x1))
T
dfit.solution()
[3 7]
```

Non-linear simple example
If the model is non-linear in the parameters to be solved, the *functional* method should be used. The main difference is that a guess solution must be inserted in the model parameters. In the following that is not necessary, since the default zero values suffice if the function is linear:

```
dfit.functional(dfs.compiled('p*x + p1*sin(x1)'),x1,
     dfs.compiled('3*x+7*sin(x[2])').f(x1), id=id)
T
dfit.solution(id=id)
[3 7]
dfit.solution(id=id)-dfit.solution()
[4.14779e-13 -4.2748e-12]
# Try with an intial guess
dfit.functional(dfs.compiled('p*x + p1*sin(x1)', [3,7]),x1,
     dfs.compiled('3*x+7*sin(x[2])').f(x1), id=id2)
T
dfit.solution(id=id2)
[3 7]
dfit.solution(id=id2)-dfit.solution()
[4.14779e-13 -4.27569e-12]
```

Functional variety
Just to show the model can be anything, we redo the fit of an order 1 polynomial to the x, y data:

```
dfit.linear(dfs.poly(1), x,y);dfit.solution()
T
[3.041 2.06]
dfs.poly(1).state()
[type=5, order=1, ndim=1, npar=2, params=[1 1] , masks=[T T] ]
```

Now try the same by a sum of odd and even polynomials of default order (note the order):

```
a := dfs.compound(); a.add(dfs.functional('oddp'));
T
a.add(dfs.functional('evenp')); a.state()
T
```

```
[type=12, order=-1, ndim=1, npar=2, params=[0 0] , masks=[T T] , nfunc=2,
   funcs=[__*0=[type=7, order=1, ndim=1, npar=1, params=0, masks=T],
   __*1=[type=6, order=1, ndim=1, npar=1, params=0, masks=T]]]
dfit.linear(a,x,y,id=id2);   dfit.solution(id=id2)
T
[2.06 3.041]
```

And the combination of an odd (2x) and an even polynomial (3):

```
a := dfs.combi(); a.add(dfs.functional('oddp', params=2));
a.add(dfs.functional('evenp', params=3)); a.state()
[type=11, order=-1, ndim=1, npar=2, params=[1 1] , masks=[T T] , nfunc=2,
   funcs=[__*0=[type=7, order=1, ndim=1, npar=1, params=2, masks=T],
   __*1=[type=6, order=1, ndim=1, npar=1, params=3, masks=T]]]
dfit.linear(a,x,y);   dfit.solution()
T
[1.03 1.01367]
```

Use constraints
We have measured a number of anlgles around a triangle. Each angle is measured 10 times (nominally 50, 60, 70 deg). Solving the angles will give:

```
include 'randomnumbers.g'
T
rn := randomnumbers()
# Create 3*10 measured values
yz := [0*1:10 + 50 + rn.normal(0,1,10),
   0*1:10 + 60 + rn.normal(0,1,10),
   0*1:10 + 70 + rn.normal(0,1,10)]
# Create 3*10 equations
xz := [array([1,0,0],30), array([0,1,0],30), array([0,0,1],30)]
# The equation used and solve
f := dfs.compiled('p*x+p1*x1+p2*x2')
f.state()
[type=13, order=-1, progtext=p*x+p1*x1+p2*x2, ndim=3, npar=3,
   params=[0 0 0] , masks=[T T T] ]
dfit.linear(f,xz,yz)
T
print dfit.solution(), 'sum=', sum(dfit.solution())
[49.7079 60.2427 70.092]  sum= 180.043
dfit.error()
[0.334828 0.334828 0.334828]
# Add a constraint: sum of angles 180deg
dfit.addconstraint(x=[1,1,1],y=180)
dfit.linear(f,xz,yz)
T
print dfit.solution(), 'sum=', sum(dfit.solution())
```

```
[49.6937 60.2285 70.0778]  sum= 180
print dfit.error()
[0.273413 0.273413 0.273413]
# Add another constraint, since we know second angle 60deg
dfit.addconstraint(x=[0,1,0],yp6+p0*exp(-((x-p1)/p2)^2) + p3*exp(-((x-p4)/p5)^2)=60)
dfit.linear(f,xz,yz)
T
print dfit.solution(), 'sum=', sum(dfit.solution())
[49.8079 60 70.1921]  sum= 180
print dfit.error()
[0.239827 0 0.239827]
```

Non-linear equation and constraints

In the following we have 2 Gaussian profiles and an offset. We add some noise, and solve assuming we have a fair estimate of the position of the Gaussians. Note that if the first estimate is beyond the real half-value point, the fitting will be difficult, due to the derivatives changing sign.

```
include 'randomnumbers.g'
T
rn := randomnumbers()
# The profile to generate and the parameters to use
# (in essence 10 + 20 * exp (-((x-10)/4)^2) + 10 * exp(-((x-33)/4)^2) )
f := dfs.compiled('p6+p0*exp(-((x-p1)/p2)^2) + p3*exp(-((x-p4)/p5)^2)',
     [20, 10, 4, 10, 33, 4, 10])
xg := [0.5 * 1:100 -0.5]
yg := f.f(xg) + rn.normal(0,0.3,100)
# Make an intial guess
f.setparameters([22, 11, 5, 10, 30, 5, 9])
# Solve
dfit.clearconstraints()
dfit.functional(f,xg,yg)
T
dfit.solution()
[19.9011 10.0063 4.01328 10.0775 33.0435 3.91559 10.015]
dfit.solution() - [20, 10, 4, 10, 33, 4, 10]
[-0.0989338 0.00629358 0.0132842 0.0775453 0.0434835 -0.0844103 0.0149934]
dfit.error()
[0.211312 0.0334257 0.0527771 0.213666 0.0652003 0.102782 0.082641]
# We know that the two lines have a peak ratio of 2: Amp1-2Amp2 = 0
dfit.addconstraint([1, 0, 0, -2, 0, 0, 0])
dfit.functional(f,xg,yg)
T
dfit.solution()
[19.9461 10.0063 4.00504 9.97305 33.0435 3.93808 10.021]
dfit.solution() - [20, 10, 4, 10, 33, 4, 10]
```

```
[-0.0538968 0.00633983 0.00503951 -0.0269484 0.0435214 -0.0619237
    0.0209581]
dfit.solution()[1]/dfit.solution()[4]
2
dfit.error()
[0.195261 0.0333704 0.0505232 0.0976303 0.0661804 0.0955169 0.0822235]
# We know that the lines originated in same place: width1 == width2
# Note that the default assumed value is 0.0
dfit.addconstraint([0, 0, 1, 0, 0, -1, 0])
dfit.functional(f,xg,yg)
T
dfit.solution()
[19.9636 10.0064 3.99814 9.98182 33.0437 3.99814 10.0009]
dfit.solution() - [20, 10, 4, 10, 33, 4, 10]
[-0.0363615 0.00638397 -0.00185905 -0.0181808 0.0436526
     -0.00185905 0.000862892]
dfit.solution()[3]-dfit.solution()[6]
0
dfit.error()
err [0.194391 0.0334112 0.0498292 0.0971955 0.0668223 0.0498292 0.0778807]
# And see what happens if we assume that the widths are 4
dfit.addconstraint([0, 0, 1, 0, 0, 0, 0], 4)
dfit.functional(f,xg,yg)
T
dfit.solution()
[19.9616 10.0064 4 9.9808 33.0437 4 9.99932]
dfit.solution() - [20, 10, 4, 10, 33, 4, 10]
[-0.0383906 0.00637359 0 -0.0191953 0.0436573 0 -0.000676228]
dfit.error()
[0.186608 0.0334226 0 0.0933042 0.0668451 0 0.0660634]
```

Deficient solutions and SVD constraints

In some cases solutions of the least-squares equations is not completely possible. An example is e.g. the solution of the closures equations in synthesis calibrations, where a missing phase zero and slope and a missing absolute gain cannot be solved for. The fitting described here will always provide a solution, even in the case of a set of incomplete equations. After the solution the *deficiency* can be checked. If there is a rank deficiency, the set of 'constraints' that makes a solution possible (in a way similar to SVD, i.e. providing a missing set of orthogonal equations) is available through the *constr* function.

```
# Provide a set of equations.
include 'randomnumbers.g'
T
rn := randomnumbers()
x := [array([1,1,1], 30)]
```

```
y := 180 + 1:10 * 0 + rn.normal(0,3,10)
f := dfs.functional('hyper', 3)
dfit.linear(f,x,y)
T
dfit.deficiency()
2
dfit.solution()
[60.0262 60.0262 60.0262]
dfit.constraint()
[-1 0 1 -1 1 0]
# The SVD constraints can be used as constraints in subsequent solutions:
dfit.addconstraint(x=dfit.constraint(1))
T
dfit.addconstraint(x=dfit.constraint(2))
T
dfit.linear(f,xz,yz)
T
dfit.solution()
[60.0262 60.0262 60.0262]
dfit.rank()
3
dfit.deficiency()
0
dfit.error()
[0.202801 0.202801 0.202801]
```

Complex fitting
The fitter can handle functions of complex variables. In the following example a second order polynomial is first fitted real with a first order linear polynomial. The same is repeated complex (with real data); and then a complex value is fitted. An example of a 2-dimensional non-linear function is also given

```
# Define x and y data
x := -1 + 0.1*[0:20]
y := dfs.poly(2, [3.03, 2.06, 0.03]).f(x)

# fit a first order polynomial
dfit.linear(dfs.poly(1), x,y); print 'linear', dfit.solution()
T
linear [3.041 2.06]

# Get a complex fitter and see the same fit
id1:=dfit.fitter()
dfit.set(type=dfit.complex(), id=id1)
```

```
T
dfit.linear(dfs.poly(1), x,y, id=id1);
T
dfit.solution(id=id1)
[3.041+0i 2.06+0i]

# Make a complex yi and redo
yi := dfs.poly(2, [3.03, 2.06, 0.03]).f(x)
yi := yi - 3i*dfs.poly(2, [3.03, 2.06, 0.03]).cf(x)
dfit.linear(dfs.poly(1),x,yi,id=id1)
T
dfit.solution(id=id1)
[3.041-9.123i 2.06-6.18i]

# A non-linear 2-dimensional function, real and complex
id2:=dfit.fitter()
dfit.functional(dfs.compiled('p*x + p1*sin(x1)', [3,7]),x1,
dfs.compiled('3*x+7*sin(x[2])').f(x1), id=id2)
x1 := 1:5
x2 := 0.1*1:5
x1 := [rbind(x1,x2)] # combine into pairs.
print [x1]
[1 0.1 2 0.2 3 0.3 4 0.4 5 0.5]
dfit.functional(dfs.compiled('p*x + p1*sin(x1)', [3,7]),x1,
dfs.compiled('3*x+7*sin(x[2])').f(x1), id=id2)
T
dfit.solution(id=id2)
[3 7]
dfit.set(type=dfit.complex(), id=id2)
T
dfit.functional(dfs.compiled('p*x + p1*sin(x1)', [3,7]),x1,
dfs.compiled('3*x+7*sin(x[2])').f(x1), id=id2)
T
dfit.solution(id=id2)
[3+0i 7+0i]
```

Tools

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

S

1.3.1 functionfitter - Tool

Package utility

Module fitting

Tool to do simple fitting of numeric data
include "functionfitter.g"

Constructors

functionfitter	Construct a functionfitter tool

Functions

done	Destroy this tool
filter	Filter a data array
fit	Fit the data
getchisq	Recover the chi squared of the fit
getdata	Recover the internal data arrays
geterror	Recover the error of the fitted parameters
getfunctionstate	Recover the state of the function
getmodel	Evaluate the model
getresidual	Evaluate the residual
getsolution	Recover the solution vector
medianclip	Median clip a data array
plot	Plot data and fit
setcoordsys	Set a new Coordinate System
setdata	Set the data to fit
setdatafromtable	Set the data to fit from a table
setfunction	Set the function you wish to fit
setparameters	Set the parameters for your function
type	Return tool type

Summary
The functionfitter `tool` offers fitting of functions to (non-complex) numeric data. It can

- Accept data from Glish arrays, AIPS++ tables and ascii tables.
- Handle data and functions of arbitrary dimension
- Handles data, errors and masks
- Filter data (median and median clip)
- Accept function definitions as expression strings or functional tools
- Do linear and non-linear fits with optional fixed parameter masks
- Return parameter errors and chi squared.

- Return model and fit data arrays
- Plot 1-D data and fits

You load this tool by including the Glish script 'functionfitter.g'. This will create for you a **defaultfunctionfitter** toolwhich you can use. You can refer to to it by its short-hand name **dff**. You can of course make your own tool as well which you should do if you are using more than one at a time (because each one contains state).

Example

```
- include 'functionfitter.g'
NORMAL: defaultfunctionfitter (dff) ready for use      # Default tool created for you
- dff.type()
functionfitter                                         # Fun type function
-
- ff1 := functionfitter()                              # Make your own tool
- ff2 := functionfitter()                              # and another
```

Example
In the following example, we use the default Functionfitter tool to fit a straight line to some x and y vectors. The straight line is specified algebraicly.

```
% glish
- xx := 1:10                         # Create some data
- yy := 2 + 3*xx
-
- include 'functionfitter.g'         # Load default functionfitter tool
- dff.setdata(xx,yy,xunit='Hz')      # Set data
- dff.setfunction ('p0 + p1*x')      # Set functional form to fit (default pars [0,0])
- print dff.fit()                    # Linear fit (initial guess not needed)
[2 3]
-
- est := 1.2 * dff.getsolution()     # Set guess slightly wrongly
- dff.setparameters (est)
- dff.fit (fixed=[F,T])              # Fit holding second parameter fixe
[-1.3 3.6]                           # and see fit go a bit wrong
- print dff.getchisq()               # Print chi-squared
29.7
- dff.plot()                         # Plot data, model and fit
```

Example
In this example we add some noise to our simple data vectors, do a linear fit and recover the errors in the parameters.

```
% glish
- x := 1:10                          # Make some data
- y := 2 + 3*x
-
- include 'randomnumbers.g'          # Load randomnumbers tool
- r := randomnumbers()
- g := r.normal(0,0.1,10)            # Generate Gaussian noise
- y +:= g;                           # add to data
-
- include 'functionfitter.g'         # Load default functionfitter tool
- dff.setdata (x, y)
- dff.setfunction ('p0 + p1*x')      # Functional form is a straight line
- dff.fit()                          # Now do a linear fit and see solution
[1.96462 2.98964]
- dff.geterror()                     # Recover errors in parameters
[0.178235 0.0287251]
- dff.plot(model=F)                  # Plot data and fit
```

Example
In the following example, we use the default Functionfitter `tool` to fit a 2-D Gaussian to some data.

```
% glish
- include 'functionals.g'         # Load basic functionals module
- n := 50                         # Size of arrays
-                                 # Create a Gaussian2D functional with parameters
- p := [1.0, n/2, n/2, n/3, 0.5, 30*pi/180]    # Amp, posx, posy, major, ratio, pa
- g2d := dfs.gaussian2d(p)
-
-                                 # Generate coordinate and value arrays
- x := [];                        # X packed as tuplet coordinate vectors in a 1-D ve
- z := [];                        # i.e. like ( [x1,x2]$_1$, [x1,x2]$_2$, [x1,x2]$_3$
#
- k := 1;
- l := 1;
- for (j in 1:n) {
    for (i in 1:n) {
      x[l] := i;                  # Fill x
      x[l+1] := j;
      l +:= 2;
#
      z[k] := g2d.f([i,j])        # Fill function
      k +:= 1
    }
```

```
  }
#
- include 'functionfitter.g'         # Create fitter
- dff.setfunction(g2d)               # Set functional
- dff.setdata (x, z)                 # Set data
- sol := dff.fit(F)                  # Do non-linear fit
- print 'Expected = ', p             # True result
Expected =  [1 25 25 16.6667 0.5 0.523599]
- print 'Solution = ', sol           # Fitted result
Solution =  [1 25 25 16.6667 0.5 0.523599]
```

functionfitter - Constructor

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

Construct a functionfitter tool

Synopsis
functionfitter()

Returns
functionfitter `tool` or fail

Description

Construct a Functionfitter tool.

Example

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

---

functionfitter.done - Function

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

Destroy this tool

Synopsis
done()

Returns
T or fail

Description

If you no longer need to use the Functionfitter `tool`, calling this function will destroy it.

Example

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

---

functionfitter.filter - Function

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

Filter a data array

Synopsis
filter(method, width, progress)

Arguments

method	in	The method type Allowed:String (case insensitive) from 'median' or 'mean' Default: 'median'
width	in	Width of running median filter Allowed:Positive integer Default: 5 points
progress	in	See progress reports on median filter every progress points Allowed:Positive integer Default: No reports

Returns
T or fail

Description

This function running-filters the current data array with a variety of kernels over the specified width. The current data are replaced by the filtered data.

Example

```
include 'functionfitter.g'
ff := functionfitter()
ff.setdata (x, y)                      # Data assumed 'pre-made'
ff.filter(method='median', width=10)   # Replace with filtered data
```

---

functionfitter.fit - Function

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

Fit the data

Synopsis
fit(linear, fixed)

Arguments

linear	in	Linear or non-linear fit ? Allowed:T or F Default: T
fixed	in	Fixed parameters mask Allowed:Boolean vector Default: All parameters fitted

Returns
Vector of float or fail

Description

This function computes and returns the fit of the supplied model to the data. If you are doing a linear fit, you don't need to set any parameters. If you are doing a non-linear fit, you should supply an initial guess via the setparameters function. Only the data for which the data mask is good (T) will be considered in the fit.
You can also hold parameters fixed in the fit via the argument `fixed`. You enter a vector of Boolean values (T to hold fixed, F to fit), one for each parameter. By default, all parameters are fitted for.

Example

```
- include 'functionfitter.g'
- ff := functionfitter()
- x := 1:10
- y := 2 + 3*x
- ff.setdata (x, y)
- ff.setfunction ('p0 + p1*x')
- print ff.fit(linear=T)
[2 3]
-
- ff.setparameters([0,3.2])          # Set function parameters
- print ff.fit(fixed=[F,T])          # Hold second parameter fixed at 3.2
[0.9 3.2]                            # parameter off a bit
```

---

functionfitter.getdata - Function

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

Recover the internal data arrays

Synopsis
getdata()

Returns
Glish record

Description

This function returns a Glish record holding the internal data. The record has fields 'x', 'y', 'xerr', 'yerr' and 'mask' holding the obvious quantities. The data error fields may be unset if no data errors were given. However, the mask field will always be filled in. If you gave no mask, the mask field will hold an all T (good) vector.

Example

```
- include 'functionfitter.g'
- ff := functionfitter()
- x := 1:10
- y := 2 + 3*x
- ff.setdata (x, y)
- r := ff.getdata()
- print r
[x=[1 2 3 4 5 6 7 8 9 10] , y=[5 8 11 14 17 20 23 26 29 32] ,
 xerr=[i_am_unset=i_am_unset], yerr=[i_am_unset=i_am_unset],
 mask=[T T T T T T T T T T] ]
```

---

functionfitter.getchisq - Function

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

Recover the chi squared of the fit

Synopsis

getchisq()

Returns
Glish record

Description

This function returns the chi squared of the fit.

Example

```
- include 'functionfitter.g'
- ff := functionfitter()
- x := 1:10
- y := 2 + 3*x
- ff.setdata (x, y)
- ff.setfunction ('p1 + p2*x')
- ff.getchsq()
0
```

---

functionfitter.getfunctionstate - Function

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

Recover the state of the function

Synopsis
getfunctionstate()

Returns
Glish record

Description

This function returns a Glish record holding the internal state of the functional (describing the function you wish to fit). See function state for which this is just a wrapper.
Users should generally not need to use this function.

Example

```
- include 'functionfitter.g'
- ff := functionfitter()
- ff.setfunction ('p0 + p1 *x')
- ff.getfunctionstate()
[type=13, order=-1, progtext=p0 + p1 *x, ndim=1,
 npar=2, params=[0 0] , masks=[T T] ]
```

---

functionfitter.geterror - Function

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

Recover the error of the fitted parameters

Synopsis
geterror()

Returns
Numeric vector or fail

Description

This function returns the standard errors of the unknowns in the fit.

Example

```
- include 'functionfitter.g'
- ff := functionfitter()
- x := 1:10
```

```
- y := 2 + 3*x
- ff.setdata (x, y)
- ff.setfunction ('p0 + p1*x')
- ff.fit(linear=T)                    # Solution perfect like data
[2 3]
- ff.geterror()                       # Errors 0 as data perfect
[0 0]
-
- include 'randomnumbers.g'           # Add some noise to data
- r := randomnumbers()
- g := r.normal(0,0.1,10)
- y +:= g;
- ff.setdata (x, y)
- ff.fit(linear=T)                    # Now fit and errors non-perfect
[1.96462 2.98964]
- ff.geterror()
[0.178235 0.0287251]
```

---

functionfitter.getmodel - Function

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

Evaluate the model

Synopsis

getmodel(fit)

Arguments

fit	in	Generate the model from the fit or the functional Allowed:T or F Default: T

Returns

Numeric vector or fail

Description

This function evaluates the model with the parameters either of the solution (`fit=T`) or the function (`fit=F`) and returns a numeric vector.
If you evaluate the model with the parameters of the function (`fit=F`) they are the parameters that you set with the setparameters function. These parameters never change.

Example

```
- include 'functionfitter.g'
- ff := functionfitter()
- x := 1:10
- y := 2 + 3*x
- ff.setdata (x, y)
- ff.setfunction ('p0 + p1*x')
- ff.fit(linear=T)
[2 3]
- ff.getmodel()
[5 8 11 14 17 20 23 26 29 32]
```

---

functionfitter.getresidual - Function

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

Evaluate the residual

Synopsis

getresidual()

Returns

Numeric vector or fail

Description

This function evaluates the residual of the data and the model (evaluated with the parameters of the solution) and returns a numeric vector.

Example

```
- include 'functionfitter.g'
- ff := functionfitter()
- x := 1:10
- y := 2 + 3*x
- ff.setdata (x, y)
- ff.setfunction ('p0 + p1*x')
- ff.fit(linear=T)
[2 3]
- ff.getresidual()
[0 0 0 0 0 0 0 0 0 0]
```

---

functionfitter.getsolution - Function

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

Recover the solution vector

Synopsis
getsolution()

Returns
Numeric vector or fail

Description

This function returns the solution vector of the last fit.

Example

```
- include 'functionfitter.g'
- ff := functionfitter()
- x := 1:10
- y := 2 + 3*x
- ff.setdata (x, y)
- ff.setfunction ('p0 + p1*x')
- ff.fit(linear=T)
[2 3]
```

---

functionfitter.medianclip - Function

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

Median clip a data array

Synopsis
medianclip(width, clip, progress)

Arguments

width	in	Width of running median filter Allowed:Positive integer Default: 5 points
clip	in	The clipping amount Allowed:Positive numeric value Default: 5
progress	in	See reports on median filter computation every progress points Allowed:Positive integer Default: No reports

Returns
T or fail

Description

This function generates a median clipped mask from a data array. See function datafilter.medianclip for details of the algorithm. The internal data mask is updated with this new mask.

Example

```
- include 'functionfitter.g'
- ff := functionfitter()
- x := 1:10
- y := 3 + 2*x
- ff.setdata (x,y)
- ff.medianclip(width=10, clip=5)
```

functionfitter.plot - Function

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

Plot data and fit

Synopsis

plot(data, model, fit, resid)

Arguments

data	in	Plot the data and optional errors Allowed:T or F Default: T
model	in	Plot the model Allowed:T or F Default: T
fit	in	Plot the fit Allowed:T or F Default: T
resid	in	Plot the residual of the data and the fit Allowed:T or F Default: F

Returns

T or fail

Description

This function lets you plot the data and (optional) errors, model, fit and residuals of the data and fit.
Note that the model is evaluated from the internal functional and its parameters. If you have not explicitly set the parameters of the functional via the setparameters function, the model will evaluate to zero.
Note that at this point, the plotting interface has no impact on the x-space in which the fit is done. The fitting is always done in with the original x (abcissa) vector that you set.

Example

```
- include 'functionfitter.g'
- ff := functionfitter()
- x := 1:10
- y := 3 + 2*x
- yerr := 0.1*y;
- ff.setdata (x,y,yerr)            # Set data and y errors
- ff.setfunction ('p0 + p1*x')
- ff.setparameters ([2.8, 2.1])     # Set model (a bit wrong)
- ff.fit(fixed=[T,F])               # Fit holding p0 fixed
- ff.plot()                         # Plot data, model & fit
```

---

functionfitter.setcoordsys - Function

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

Set a new Coordinate System

Synopsis
setcoordsys(csys, axis)

Arguments

csys	in	The Coordinate System Allowed:Coordsys tool
axis	in	The axis of the Coordinate System that we are plotting Allowed:Integer Default: 1

Returns
T or fail

Description

By default, when you set data into the Functionfitter tool (with functions setdata or setdatafromtable) a tabular Coordinate System is created internally. This maps between pixel index and the x world value (with units specified in the above function calls). This Coordinate System is used by the plotting function plot.

This function allows you to set a diffferent Coordinate System. For example, if the units of the x-data are frequency, then, rather than using the internal Tabular Coordinate System, you could set a Spectral Coordinate System and then you will be able to convert to and from velocity units with the plotting interface. You would need to correctly fill in the Spectral Coordinate System with funtions from the coordsys `tool`.
Note that at this point, the plotting interface has no impact on the x-space in which the fit is done. The fitting is always done in with the original x (abcissa) vector that you set.

Example

```
- include 'functionfitter.g'
- ff := functionfitter()
- x := 1:10
- y := 3 + 2*x
- ff.setdata (x,y)
- ff.setfunction ('p0+p1*x')
- ff.fit()
- ff.plot()                         # Plots a linear axis
-
- include 'coordsys.g'
- cs := coordsys(spectral=T)        # Default spectral coordinate
- ff.setcoordsys(cs)
- ff.plot()                         # Plots a spectral axis with velocity conversions
```

---

functionfitter.setdata - Function

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

Set the data to fit

Synopsis
setdata(x, y, yerr, mask, xunit)

Arguments

x	in	The coordinate values Allowed:Numeric vector
y	in	The ordinate values Allowed:Numeric vector
yerr	in	The errors in the ordinate values Allowed:Numeric vector Default: None used
mask	in	The mask Allowed:Boolean vector Default: All good
xunit	in	The unit of the x-data Allowed:Unit string Default: 'm'

Returns
T or fail

Description

This function sets the data that you wish to fit. You must give the x and y data. You can optionally give the errors in these. You can optionally give a mask. If you don't give a mask, all the data are assumed to be good (T).
You need to be aware of how the coordinate values (x-data) are packed if the dimensionality of the function you wish to fit (and the coordinates) is greater than 1. The coordinates are stored as n-tuplets in a 1-D vector. For example, if you are fitting a 2-D function, then the x vector would consist of n 2-tuplets, where n is the length of the ordinate vector. The order would be ($[x1,x2]_1$, $[x1,x2]_2$, $[x1,x2]_3$...). This all means that if the dimensionality of the function you are fitting is dim, and the length of the y vector is n, then the length of nx must be $dim * n$. The only exception to this is if the function has 0 dimensions (e.g. a constant). In this case, you should really specify an empty x vector, but Functionfitter handles this for you internally so it doesn't matter.
If and when you plot the data, a Tabular coordsys `tool` is made internally for you. This maps pixel index to the world x values that you specified with the `x` vector. The world units are given by the `xunit` variable, which defaults, not very helpfully perhaps, to metres. You can replace this Tabular Coordinate System if you want with another via the setcoordsys function.

Example

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

```
- x := 1:10
- y := 3 + 2*x
- ff.setdata (x,y)
```

Example
In the following example, we use the default Functionfitter `tool` to fit a 2-D Gaussian to some data.

```
- include 'functionals.g'          # Load basic functionals module
- n := 50                          # Size of arrays
-                                  # Create a Gaussian2D functional with parameters
- p := [1.0, n/2, n/2, n/3, 0.5, 30*pi/180]    # Amp, posx, posy, major, ratio, pa
- g2d := dfs.gaussian2d(p)
-
-                                  # Generate coordinate and value arrays
- x := [];                         # X packed as tuplet coordinate vectors in a 1-D vector
- z := [];                         # i.e. like ( [i_1,j_1], [i_2,j_2], [i_3,j_3] )
#
- k := 1;
- l := 1;
- for (j in 1:n) {
    for (i in 1:n) {
      x[l] := i;                   # Fill x
      x[l+1] := j;
      l +:= 2;
#
      z[k] := g2d.f([i,j])         # Fill function
      k +:= 1
    }
  }
#
- include 'functionfitter.g'       # Create fitter
- dff.setfunction(g2d)             # Set functional
- dff.setdata (x, z)               # Set data
- dff.setparameters(0.8*p)         # Set initial model
- dff.fit(F)                       # Do non-linear fit
- print 'Expected = ', p           # True result
- print 'Solution = ', dff.getsolution()    # Fitted result
```

---

functionfitter.setdatafromtable - Function

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

Set the data to fit from a table

Synopsis
setdatafromtable(name, cold, cole, colm, xunit, autoheader)

Arguments

name	in	The table (AIPS++ or ascii) name Allowed:String
cold	in	The columns holding the coordinate and ordinate values Allowed:Integer vector of length 2 Default: [1,2]
cole	in	The columns holding the ordinate error data Allowed:Integer Default: None
colm	in	The column holding the mask data Allowed:Integer Default: None
xunit	in	The unit of the x-data Allowed:Unit string Default: 'm'
autoheader	in	Auto detect data type from ascii table Allowed:T or F Default: T

Returns
T or fail

Description

This function sets the data that you wish to fit by reading it directly from a table. That table can be an AIPS++ table or an ascii table (this is auto-detected). You specify which columns in the table which the appropriate data are to be found. If the table is ascii, the mask column should be integer with values 0 (bad) or 1 (good).
The coordinate column may be multi-dimensional (e.g. if you want to fit 2-D functions to the data). See function setdata for more information on coordinate vector packing and data dimensions. As an example, imagine you are fitting an N-D function. Each row of the Coordinates column should hold a vector (of length N) of coordinates.
The argument `autoheader` is only relevant if the input data are stored in an ascii file; it is ignored otherwise. It is used to control whether it is attempted

to auto-detect the type of data in the table or not. If `autoheader=T` (the default), then the first row of the text file (and it should be a proper data row) is used to work out the data types. It will only recognize I (integer), D (double or float) and A (ascii). This is usually the easiest thing to do for simple fitting of 1-D functions.
If `autoheader=F` then you **must** specify the column names (although they are not used by this Tool) in row 1, and the data types for each column in row 2. If you are fitting 1-D data, then you can use either value for `autoheader`. However, if you are fitting data of greater dimensionality than one, you **must** use `autoheader=F` and specify how the data are formatted (it cannot be auto-determined what the dimensionality of columns is). As an example, let us say you are fitting a 2-D problem. This means the coordinate vectors must be of length two. The first couple of lines of the ascii table might look like:

```
COL1     COL2                 # Column names must be given (but can be arbitrary)
R2       R                    # Must specify column shape and type
1.1 2.2  10.0                 # Line 1
2.2 4.4  20.0                 # Line 2
```

So we specify explicitly that the coordinate column is a vector of floats and the vector is of length 2. For the data column, it is just a scalar (R, not R1 which would be a vector of length 1). For more details you can look at the tablefromascii constructor which is used behind the scenes by this Tool.
If and when you plot the data, a Tabular coordsys `tool` is made internally for you. This maps pixel index to the world x values that you specified with the `x` vector. The world units are given by the `xunit` variable, which defaults, not very helpfully perhaps, to metres. You can replace this Tabular Coordinate System if you want with another via the setcoordsys function.

Example

```
- include 'functionfitter.g'
- ff := functionfitter()
- ff.setdatafromtable ('data.txt', [1,2], 3, 4);   # Ascii table;  Columns 1, 2, 3, 4
- ff.setdatafromtable ('data.tbl', [3,4], 1, 2)    # aips++ table; Columns 3, 4, 1, 2
```

---

functionfitter.setfunction - Function

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

Set the function you wish to fit

Synopsis
setfunction(fn)

Arguments

fn	in	The function to fit Allowed:String expression or functional

Returns
T or fail

Description

This function sets the function that you wish to fit to the data. You can set the function in one of two ways.
First, you can provide an expression string of an arbitrary function. See function compiled for details on the syntax for this string. Basically the parameters to be solved for are labelled $p0, p1, p2...$ or alternatively $p[1], p[2], p[3]....$ Similarly, you should use the labels $x0, x1, x2...$ or alternatively $x[1], x[2], x[3]...$ to refer to the abcissa (which must have the same dimensionality as the functional). For example, if your abcissa is 2-dimensional, you would use $x0$ (or $x[1]$ or just x in this case) and $x1$ (or $x[2]$).
Second, you can provide a pre-existing Functional `tool`. This might itself have been created from an expression string, or it may be a pre-defined functional (e.g. a Gaussian1D), or perhaps a compound functional. See the details on the Functionalserver `tool`.
Internally, a Functional `tool` is stored whether you provide it directly or indirectly.

Example

```
- include 'functionfitter.g'
- ff := functionfitter()
- ff.setfunction ('p0 + p1*x0 * p2*x0*x0')      # Second order polynomial
-
- include 'functionalserver.g'
- g := dfs.gaussian1d()                        # A 1-d Gaussian
- ff.setfunction(g)
```

---

functionfitter.setparameters - Function

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

Set the parameters for your function

Synopsis
setparameters(pars)

Arguments

pars	in	The parameters to set Allowed:Numeric vector

Returns
T or fail

Description

This function sets the parameters of the function that you wish to fit to the data. If you are going to do a linear fit, you don't need to call this function. If you are going to do a non-linear fit you need to set a guess for the parameters (else they will be zero by default).
When you set parameters, the internally stored Functional `tool` is updated with the new parameters. Actually doing a fit does not change the values of the parameters. This means that if you wanted to do something like 1) do a fit, 2) set the parameters of the function to the solution and then hold a parameter fixed, and re-fit, you would need to do that explicitly yourself (see example below).

Example

```
- include 'functionfitter.g'
- x := 1:10                          # Make some data
- y := 2 + 3*x
- ff := functionfitter()             # Create fitter
- ff.setdata(x,y)                    # Set data
- ff.setfunction ('p0 + p1*x')       # Create functional (parameters [0,0])
```

```
- 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