```
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** `tool`which 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])
```