1.2.10 checker - Tool

Package utility
Module catalog

Tool to check the AIPS++ environment
include "checker.g"

Constructors

checker	Construct an checker

Functions

all	Check all properties
display	Check display properties
gui	Check gui properties
memory	Check memory properties
perl	Check to see if perl is available
plotter	Check plotter properties
type	Return the type of this tool

Description

This tool checks the AIPS++ environment for suitability for running.
Unsuitable settings are noted and recommendations are made.
There is a defaultchecker with abbreviation dch. Normally only one should be needed.

checker - Constructor

Package utility
Module catalog
Tool checker

Construct an checker

Synopsis
checker()

Returns
checker tool

Description

This is the only `checker` constructor. Since only one checker should ever be needed, the use of this constructor should be unnecessary.

---

checker.type - Function

Package	**utility**
Module	**catalog**
Tool	**checker**

Return the type of this tool

Synopsis
type()

Returns
String

Description

This tool function returns the string "checker". If you have a tool, but don't know what kind it is, this function usually exists to tell you who the tool is.

---

checker.all - Function

Package	**utility**
Module	**catalog**
Tool	**checker**

Check all properties

Synopsis
all(verbose)

Returns
Number of warnings

Description
All the checks are performed.

Arguments

verbose	Be verbose? Allowed:Boolean Default: F

---

checker.display - Function

Package	**utility**
Module	**catalog**
Tool	**checker**

Check display properties

Synopsis
display(verbose)

Returns
Number of warnings

Description
The root visual is checked to see if it is not in TrueColor mode.

Arguments

verbose	Be verbose? Allowed:Boolean Default: T

---

checker.perl - Function

Package	**utility**
Module	**catalog**
Tool	**checker**

Check to see if perl is available

Synopsis
perl(verbose)

Returns
Number of warnings

Description
perl is used for some operations in the help system. This checks to see if perl is available.

Arguments

verbose	Be verbose? Allowed:Boolean Default: T

---

checker.plotter - Function

Package	**utility**
Module	**catalog**
Tool	**checker**

Check plotter properties

Synopsis
plotter(verbose)

Returns
Number of warnings

Description
The environment variable PGPLOT_FONT is checked to verify that it points to an existing file. Also the Glish variable pgplot is checked for existence and that it is a function.

Arguments

verbose	Be verbose? Allowed:Boolean Default: T	

---

checker.gui - Function

Package	**utility**
Module	**catalog**
Tool	**checker**

Check gui properties

Synopsis
gui(verbose)

Returns
Number of warnings

Description
have_gui() is run to see if Glish has a GUI.

Arguments

verbose	Be verbose? Allowed:Boolean Default: T	

---

checker.memory - Function

Package	**utility**
Module	**catalog**
Tool	**checker**

Check memory properties

Synopsis

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)
```