Package 'hyper.fit'

August 1, 2016

Type Package

Title Generic N-Dimensional Hyperplane Fitting with Heteroscedastic Covariant Errors and Intrinsic Scatter

Version 1.0.3

Date 2016-08-01

Author Aaron Robotham and Danail Obreschkow

Maintainer Aaron Robotham <aaron.robotham@uwa.edu.au>

Description Includes two main high level codes for hyperplane fitting (hyper.fit) and visualising (hyper.plot2d / hyper.plot3d). In simple terms this allows the user to produce robust 1D linear fits for 2D x vs y type data, and robust 2D plane fits to 3D x vs y vs z type data. This hyperplane fitting works generically for any N-1 hyperplane model being fit to a N dimension dataset. All fits include intrinsic scatter in the generative model orthogonal to the hyperplane.

License GPL-3

Depends R (>= 3.00), magicaxis, MASS, rgl, LaplacesDemon

NeedsCompilation no

Repository CRAN

Date/Publication 2016-08-01 17:29:45

R topics documented:

`hyper.basic` *Functions to calculate various basic properties important for line fitting*

Description

rotdata2d: Function to generate a 2xN matrix with rotated columns x and y.

rotdata3d: Function to generate a 3xN matrix with rotated columns x,y and z.

makerotmat2d: Function to generate a 2x2 rotation matrix.

makerotmat3d: Function to generate a 3x3 rotation matrix.

rotcovmat: Function to generate a rotated covariance matrix, either 2x2 or 3x3.

ranrotcovmat2d: Function to generate a randomly rotated 2x2 covariance matrix.

ranrotcovmat3d: Function to generate a randomly rotated 3x3 covariance matrix.

makecovarray2d: Function to generate a 2x2xN covariance array.

makecovarray3d: Function to generate a 3x3xN covariance array.

makecovmat2d: Function to generate a 2x2 covariance matrix.

makecovmat3d: Function to generate a 3x3 covariance matrix.

projX: Projection of position (or possibly velocity) elements along a vector (any number of dimensions, but matrices must conform).

projcovmat: Projection of a covariance matrix along a vector (any number of dimensions, but matrices must conform).

projcovarray: Projection of a covariance array (dim x dim x N) along a vector (any number of dimensions, but matrices must conform).

arrayvecmult: Matrix multiply array elements by a vector. To behave sensibly the second dimension of the AxBxC (i.e. B) array should be the same length as the multiplication vector.

Usage

```
rotdata2d(x, y, theta)

rotdata3d(x, y, z, theta, dim = 'z')

makerotmat2d(theta)

makerotmat3d(theta, dim = 'z')

rotcovmat(covmat, theta, dim = 'x')

ranrotcovmat2d(covmat)

ranrotcovmat3d(covmat)
```

```
makecovarray2d(sx, sy, corxy)

makecovarray3d(sx, sy, sz, corxy, corxz, coryz)

makecovmat2d(sx, sy, corxy)

makecovmat3d(sx, sy, sz, corxy, corxz, coryz)

projX(X, projvec)

projcovmat(covmat, projvec)

projcovarray(covarray, projvec)

arrayvecmult(array, vec)
```

Arguments

`x`	Vector of x data. Should be the same length as y.
`y`	Vector of y data. Should be the same length as x.
`z`	Vector of z data. Should be the same length as x/y.
`X`	A position matrix with the N (number of data points) rows by d (number of dimensions) columns.
`sx`	Vector of x errors. Should be the same length as sy.
`sy`	Vector of y errors. Should be the same length as sx.
`sz`	Vector of z errors. Should be the same length as sx/sy.
`corxy`	Vector of correlation values between sx and sy values. Should be the same length as sx.
`corxz`	Vector of correlation values between sx and sz values. Should be the same length as sx/sy/sz/corxy/coryz.
`coryz`	Vector of correlation values between sy and sz values. Should be the same length as sx/sy/sz/corxy/corxz.
`covmat`	A dxd (d=dimensions, i.e. 2x2 for 2d or 3x3 for 3d). The makecovmat2d and makecovmat3d are convenience functions that make populating 2x2 and 3x3 matrices easier for a novice user.
`covarray`	A dxdxN array containing the full covariance (d=dimensions, N=number of dxd matrices in the array stack). The makecovarray2d and makecovarray3d are convenience functions that make populating 2x2xN and 3x3xN arrays easier for a novice user.
`theta`	Angle in degrees for rotation. x -> y = +ve rotation, x -> z = +ve rotation, y -> z = +ve rotation.
`dim`	In 3D this specifies the axis around which the rotation takes place. If dim='x' rotation is in the yz plane and y -> z = +ve rotation. If dim='y' rotation is in the xz plane and x -> z = +ve rotation. If dim='z' rotation is in the xy plane and x -> y = +ve rotation.

`projvec`	The vector defining the desired projection. This does not need to be of length 1, but the user should be aware that unless it is of length 1 then the solution will not be normalised correctly for the unit vector case (if this is desired the input should be projvec=exprojvec/(sqrt(sum(exprojvec^2))), where exprojvec is the desired direction of projection).
`array`	A AxBxC array.
`vec`	A vector of length B (same length as the second array dimension of array argument).

Value

rotdata2d: A 2xN matrix with rotated columns x and y.

rotdata3d: A 3xN matrix with rotated columns x,y and z.

makerotmat2d: A 2x2 rotation matrix.

makerotmat3d: A 3x3 rotation matrix.

rotcovmat: A rotated covariance matrix, either 2x2 or 3x3.

ranrotcovmat2d: A randomly rotated 2x2 covariance matrix.

ranrotcovmat3d: A randomly rotated 3x3 covariance matrix.

makecovarray2d: A 2x2 covariance matrix.

makecovarray3d: A 3x3 covariance matrix.

makecovmat2d: A 2x2xN covariance array.

makecovmat3d: A 3x3xN covariance array.

projX: Projection of X along vector (length N).

projcovmat: Projection of covariance matrix along vector (length 1).

projcovarray: Projection of covariance array along vector (length N).

arrayvecmult: Matrix multiplication of array stack slices by vector (size dxN).

Author(s)

Aaron Robotham and Danail Obreschkow

References

Robotham, A.S.G., & Obreschkow, D., PASA, in press

See Also

`hyper.basic`, `hyper.convert`, `hyper.data`, `hyper.fit`, `hyper.plot`, `hyper.sigcor`, `hyper.summary`

Examples

```
extheta=30    #Experiment with changing this
exdim='z'     #Experiment with chaging to 'x' or 'y'
exvecx=1:10   #Experiment with changin this
exvecy=1:10   #Experiment with changin this
exvecz=1:10   #Experiment with changin this

print(cbind(exvecx, exvecy))
print(rotdata2d(exvecx, exvecy, extheta))
print(rotdata3d(exvecx, exvecy, exvecz, extheta, exdim))
print(makerotmat2d(extheta))
print(makerotmat3d(extheta, dim=exdim))

exsx=1        #Experiment with changing this
exsy=2        #Experiment with changing this
exsz=3        #Experiment with changing this
excorxy=0.8   #Experiment with changing this between -1 and 1
excorxz=-0.3  #Experiment with changing this between -1 and 1
excoryz=0.5   #Experiment with changing this between -1 and 1

print(makecovmat2d(exsx, exsy, excorxy))
print(makecovmat3d(exsx, exsy, exsz, excorxy, excorxz, excoryz))
print(makecovarray2d(exsx*1:4, exsy*1:4, excorxy))
print(makecovarray3d(exsx*1:4, exsy*1:4, exsz*1:4, excorxy, excorxz, excoryz))

excovmat2d=makecovmat2d(exsx, exsy, excorxy)
excovmat3d=makecovmat3d(exsx, exsy, exsz, excorxy, excorxz, excoryz)
excovarray2d=makecovarray2d(exsx*1:4, exsy*1:4, excorxy)
excovarray3d=makecovarray3d(exsx*1:4, exsy*1:4, exsz*1:4, excorxy, excorxz, excoryz)

print(rotcovmat(excovmat2d, extheta))
print(rotcovmat(excovmat3d, extheta, exdim))
print(ranrotcovmat2d(excovmat2d))
print(ranrotcovmat3d(excovmat3d))

exprojvec2d=c(1, 2)
exprojvec2d=exprojvec2d/sqrt(sum(exprojvec2d^2))
exprojvec3d=c(1, 2, 3)
exprojvec3d=exprojvec3d/sqrt(sum(exprojvec3d^2))

print(projX(cbind(exvecx, exvecy), exprojvec2d))
print(projX(cbind(exvecx, exvecy, exvecz), exprojvec3d))
print(projcovmat(excovmat2d, exprojvec2d))
print(projcovmat(excovmat3d, exprojvec3d))
print(projcovarray(excovarray2d, exprojvec2d))
print(projcovarray(excovarray3d, exprojvec3d))

#Notice that the first outputs of the 2d/3d projcovarray example correspond to the outputs
#of the 2d/3d projcovmat examples.

#First for comparison:
```

```
print(t(matrix(1:9,3) %*% 1:3))
print(t(matrix(10:18,3) %*% 1:3))
print(t(matrix(19:27,3) %*% 1:3))

#And now an array example of the above operations:

print(arrayvecmult(array(1:27,c(3,3,3)),1:3))

#And an example where all array dimensions are different:

print(matrix(1:6,2) %*% 1:3)
print(matrix(7:12,2) %*% 1:3)
print(matrix(13:18,2) %*% 1:3)
print(matrix(19:24,2) %*% 1:3)
print(arrayvecmult(array(1:24,c(2,3,4)),1:3))

#Note that the following is not allowed:

## Not run:
print(arrayvecmult(array(1:24,c(3,2,4)),1:3))

## End(Not run)
```

`hyper.convert` *Parameterisation conversion functions.*

Description

The hyper.convert function allows the user to generically convert their current plane definition system to an alternative. The obvious use case might be when the user has an equation defined as a projection formula along a preferred axis (e.g. z=ax+by+c) and they want to find the orthogonal offset of the hyperplane to the origin, and the intrinsic scatter orthogonal to the hyperplane.

To ease use, and minimise mistakes, hyper.convert creates an object of type hyper.plane.param which can then be converted using the class specific convert function. This has the advantage of knowing how the current projection is defined, and minmises the user inputs to simply the new projection desired.

Usage

```
hyper.convert(parm, coord, beta = 0, scat = 0, in.coord.type = "alpha", out.coord.type,
in.scat.type = "vert.axis", out.scat.type, in.vert.axis, out.vert.axis)

#To ease usability the package also included a hyper.plane.param class specific convert
#function:

## S3 method for class 'hyper.plane.param'
convert(x,coord.type='alpha',scat.type='vert.axis',vert.axis,...)
```

Arguments

`x` Argument for the class dependent convert.hyper.plane.param function. An object of class hyper.plane.param, i.e. the output of the hyper.convert function.

`coord.type` Argument for the class dependent convert.hyper.plane.param function. This specifies whether the output coord vector is defined in terms of the normal vector to the hyperplane (normvec) gradients defined to produce values along the vert.axis dimension (alpha) or by the values of the angles that form the gradients (theta). If missing it takes the value of in.coord.type.

`scat.type` Argument for the class dependent convert.hyper.plane.param function. This specifies whether the output beta/scat should be defined as orthogonal to the plane (orth) or along the vert.axis of interest (vert.axis). If missing it takes the value of in.scat.type.

`vert.axis` Argument for the class dependent convert.hyper.plane.param function. This specifies the output/requested vertical projection axis. If missing it takes the value of in.vert.axis.

`parm` Vector of all parameters. This should be a concatenation of c(coord,beta,scat). Either 'parm' or 'coord' must be specified.

`coord` The current coordinate parameters using the in.coord.type coordinate system. This should be a vector of length dimensions-1 (i.e. 1 for 2D xy data and 2 for 3D xyz data). The coord argument must be explictly specified by the user.

`beta` The current offset of the hyperplane defined using the in.scat.type projection system. The default is 0.

`scat` The current intrinsic scatter of the hyperplane defined using the in.scat.type projection system. The default is 0.

`in.coord.type` This specifies whether the input coord vector is defined in terms of the normal vector to the hyperplane (normvec) gradients defined to produce values along the vert.axis dimension (alpha, default) or by the values of the angles that form the gradients (theta).

`out.coord.type` This specifies whether the output coord vector is defined in terms of the normal vector to the hyperplane (normvec) gradients defined to produce values along the vert.axis dimension (alpha) or by the values of the angles that form the gradients (theta). If missing it takes the value of in.coord.type.

`in.scat.type` This specifies whether the input scat is defined as orthogonal to the plane (orth) or along the vert.axis of interest (vert.axis).

`out.scat.type` This specifies whether the output scat should be defined as orthogonal to the plane (orth) or along the vert.axis of interest (vert.axis). If missing it takes the value of in.scat.type.

`in.vert.axis` This specifies the input vertical projection axis. If missing it uses the maximum dimension value (i.e. y-axis for a 2d dataset).

`out.vert.axis` This specifies the output/requested vertical projection axis. If missing it takes the value of in.vert.axis.

`...` Additional arguments to pass to convert.hyper.plane.param, namely coord.type, scat.type, vert.axis.

Value

hyper.convert returns a multi-component list of class hyper.plane.param containing:

`parm`	parm is a concatination of the parameters that fully decribe the hyperplane with intrinsic scatter. It can be used as a direct input to other hyper.fit functions that can accept a parm type input.
`coord`	The output coordinate parameters using the out.coord.type coordinate system.
`beta`	The output offset of the hyperplane defined using the out.scat.type projection system. Potentially standardised by the abs.beta.orth argument.
`scat`	The output intrinsic scatter of the hyperplane defined using the out.scat.type projection system.
`unitvec`	The unit vector orthogonal to the hyperplane. This is always the vector pointing from the origin *to* the hyperplane.
`beta.orth`	The absolute distance of the hyperplane to the origin. By definition this will always be positive.
`scat.orth`	The intrinsic scatter orthogonal to the hyperplane. By definition this will always be positive.
`coord.type`	The requested out.coord.type.
`scat.type`	The requested out.scat.type.
`vert.axis`	The requested out.vert.axis (if out.vert.axis is not specified, this will be the maximum dimension value, see in.vert.axis in arguments section above).

Author(s)

Aaron Robotham and Danail Obreschkow

References

Robotham, A.S.G., & Obreschkow, D., PASA, in press

See Also

`hyper.basic`, `hyper.convert`, `hyper.data`, `hyper.fit`, `hyper.plot`, `hyper.sigcor`, `hyper.summary`

Examples

```
#Here we are assuming our plane forula is z=2x+3y+1 plus Gaussian intrinsic scatter along
#z with sd=4:

excoord.alpha=c(2,3)
exbeta.vert.axis=1
exscat.vert.axis=4

print(hyper.convert(coord=excoord.alpha, beta=exbeta.vert.axis, scat=exscat.vert.axis,
out.coord.type='theta', out.scat.type='orth'))
print(hyper.convert(coord=excoord.alpha, beta=exbeta.vert.axis, scat=exscat.vert.axis,
```

```
out.vert.axis=2))

#To simplify conversions and reduce mistakes you can use the class dependent method:
temp=hyper.convert(coord=excoord.alpha, beta=exbeta.vert.axis, scat=exscat.vert.axis)
print(convert(temp, coord.type='normvec')$parm)
print(convert(temp, coord.type='theta')$parm)
print(convert(temp, coord.type='theta', vert.axis=2)$parm)
print(convert(temp, coord.type='theta', scat.type='orth')$parm)
#We can check the conversions by
print(temp$parm)
print(convert(convert(convert(convert(temp, 'normvec'), 'theta', 'vert.axis',1), 'alpha',
'orth', 2))$parm)

#The conversions back and forth won't return *exactly* the same values:
print(all(convert(convert(convert(convert(temp, 'normvec'), 'theta', 'vert.axis', 1),'alpha',
'orth', 2))$parm==temp$parm))
#But they will be very close for the most part:
print(all(round(convert(convert(convert(convert(temp, 'normvec'), 'theta', 'vert.axis', 1),
'alpha', 'orth', 2))$parm)==temp$parm))
```

`hyper.data` *Data included in hyper.fit package*

Description

hogg: Toy ASCII table taken from Hogg 2010. Contains x/y/sx/sy/corxy columns. Provided by David Hogg.

intrin: Toy ASCII table that has intrinsic scatter. Contains x/y/sx/sy/corxy columns. Provided by Michelle Cluver.

trumpet: Toy ASCII table that has trumpet-like covariance errors and no intrinsic scatter. Contains x/y/sx/sy/corxy columns. Provided by Johannes Buchner.

FP6dFGS: 6dFGS fundamental plane data taken from table 8 of Campbell et al 2014. We use columns 6/7, 12/13, 18/19 for the FP parameters and their errors, and column 26 = 111111 to create a clean selection. Provided By Christina Magoulas.

GAMAsmVsize: Galaxy mass vs size data taken from Lange et al 2014. Bottom-right panel of Figure 3 (i.e. r-band elliptical relation data). Provided by Rebecca Lange.

TFR: Tully-Fisher Relation data taken from Obreschkow and Meyer 2013. Provided by Danail Obreschkow.

MJB: Mase-Angular Momentum-Bulge/Total data taken from Obreschkow & Glazebrook 2014. Provided by Danail Obreschkow.

convtest2dOpt: Intrinsic scatter convergence test data for 2 DoF and optim fitted simulations, shown as an example in `hyper.sigcor`.

convtest2dLD: Intrinsic scatter convergence test data for 2 DoF and LaplacesDemon fitted simulations, shown as an example in `hyper.sigcor`.

convtest1dNorm: Intrinsic scatter convergence test data for 1 DoF and direct sample SD estimation, shown as an example in `hyper.sigcor`.

Usage

```
data(hogg)
data(intrin)
data(trumpet)
data(FP6dFGS)
data(GAMAsmVsize)
data(TFR)
data(MJB)
data(convtest2dOpt)
data(convtest2dLD)
data(convtest1dNorm)
```

Author(s)

Aaron Robotham and Danail Obreschkow

References

Robotham, A.S.G., & Obreschkow, D., PASA, in press

Campbell, L., et al., 2014, MNRAS, 443, 1231 (http://mnras.oxfordjournals.org/content/443/2/1231)

Cluver, M., et al., 2014, ApJ, 782, 90 (http://arxiv.org/pdf/1401.0837v1.pdf)

Hogg, D., Bovy, J., Lang, D., 2010 (http://arxiv.org/pdf/1008.4686v1.pdf)

Lange, R., et al., MNRAS, accepted

Obreschkow & Meyer, 2013, ApJ, 777, 140

Obreschkow & Glazebrook, 2014, ApJ, 784, 26

See Also

hyper.basic, hyper.convert, hyper.data, hyper.fit, hyper.plot, hyper.sigcor, hyper.summary

Examples

```
hogg=read.table(system.file('data/hogg.tab', package='hyper.fit'),header=TRUE)
#or
data(hogg)
print(hogg[1:10,])

intrin=read.table(system.file('data/intrin.tab', package='hyper.fit'), header=TRUE)
#or
data(intrin)
print(intrin[1:10,])

trumpet=read.table(system.file('data/trumpet.tab', package='hyper.fit'), header=TRUE)
#or
data(trumpet)
print(trumpet[1:10,])

FP6dFGS=read.table(system.file('data/FP6dFGS.tab', package='hyper.fit'), header=TRUE)
```

```
#or
data(FP6dFGS)
print(FP6dFGS[1:10,])

GAMAsmVsize=read.table(system.file('data/GAMAsmVsize.tab', package='hyper.fit'), header=TRUE)
#or
data(GAMAsmVsize)
print(GAMAsmVsize[1:10,])

TFR=read.table(system.file('data/TFR.tab', package='hyper.fit'), header=TRUE)
#or
data(TFR)
print(TFR[1:10,])

MBJ=read.table(system.file('data/MJB.tab', package='hyper.fit'), header=TRUE)
#or
data(MJB)
print(MJB[1:10,])
```

hyper.fit — *Top level function that attempts to fit a hyperplane to provided data.*

Description

Top level fitting function that uses downhill searches (optim/LaplaceApproximation) or MCMC (LaplacesDemon) to search out the best fitting parameters for a hyperplane (minimum a 1D line for 2D data), including the intrinsic scatter as part of the fit.

Usage

```
hyper.fit(X, covarray, vars, parm, parm.coord, parm.beta, parm.scat, parm.errorscale=1,
vert.axis, weights, k.vec, itermax = 1e4, coord.type = 'alpha', scat.type = 'vert.axis',
algo.func = 'optim', algo.method = 'default', Specs = list(Grid=seq(-0.1,0.1, len=5),
dparm=NULL, CPUs=1, Packages=NULL, Dyn.libs=NULL), doerrorscale = FALSE, ...)
```

Arguments

`X`	A position matrix with N (number of data points) rows by d (number of dimensions) columns.
`covarray`	A dxdxN array containing the full covariance (d=dimensions, N=number of dxd matrices in the array stack). The makecovarray2d and makecovarray3d are convenience functions that make populating 2x2xN and 3x3xN arrays easier for a novice user.
`vars`	A variance matrix with the N (numver of data points) rows by d (number of dimensions) columns. In effect this is the diagonal elements of covarray where all other terms are zero. If covarray is also provided that input argument is used instead.

- `parm` — Vector of all initial parameters. This should be a concatenation of c(parm.coord, parm.beta, parm.scat). If doerrorscale=TRUE then there should be an extrat element to give c(parm.coord, parm.beta, parm.scat, parm.errorscale). See the parm.coord, parm.beta, parm.scat and parm.errorscale arguments below for more information on what these different quantities describe.
- `parm.coord` — Vector of initial coord parameters. These are either angles that produce the vectors that predict the vert.axis dimension (coord.type="theta"), the gradients of these (coord.type="alpha") or they are the vector elements normal to the hyperplane (coord.type="normvec"). Depending on the argument used, either beta along the vertical axis is generated (coord.type="alpha"/theta") or no beta is required (coord.type="normvec").
- `parm.beta` — Initial value of beta. This is either specified as the absolute distance from the origin to the hyperplane or as the intersection of the hyperplane on the vert.axis dimension being predicted.
- `parm.scat` — Initial value of the intrinsic scatter. This is either specified as the scatter orthogonal to the hyperplane or as the scatter along the vert.axis dimension being predicted.
- `parm.errorscale` — If doerrorscale=TRUE, this argument is the initial errorscale (default is 1). See the doerrorscale argument below for more details.
- `vert.axis` — Which axis should the hyperplane equation be formulated for. This must be a number which specifies the column of position matrix X to be defined by the hyperplane. If missing, then the projection dimension is assumed to be the last column of the X matrix.
- `weights` — Vector of multiplicative weights for each row of the X data matrix. i.e. if this is 2 then it is equivalent to having two identical data points with weights equal to 1. Should be either of length 1 (in which case elements are repeated as required) or the same length as the number of rows in the data matrix X.
- `k.vec` — A vector defining the direction of an exponential sampling distribution in the data. The length is the scaling "a" of the generative exponent (i.e., exp(a*x)), and it points in the direction of *increasing* density (see example below). If provided, k.vec must be the same length as the dimensions of the data. k.vec has the most noticeable effect on the beta offset parameter. It is correcting for the phenomenom Astronomers often call Eddington bias. For discussion on the use of k.vec see Appendix B of Robotham & Obreschkow.
- `itermax` — The maximum iterations to use for either the LaplaceApproximation function or LaplacesDemon function.
- `coord.type` — This specifies whether the fit should be done in terms of the normal vector to the hyperplane (coord.type="normvec") gradients defined to produce values along the vert.axis dimension (coord.type="alpha") or by the values of the angles that form the gradients (coord.type="theta"). "alpha" is the default since it is the most common means of parameterising hyperplane fits in the astronomy literature.
- `scat.type` — This specifies whether the intrinsic scatter should be defined orthogonal to the plane (orth) or along the vert.axis of interest (vert.axis).
- `algo.func` — If '`algo.func="optim"`' (default) hyper.fit will optimise using the R base `optim` function. If '`algo.func="LA"`' will optimise using the `LaplaceApproximation`

	function. If 'algo.func="LD"' will optimise using the LaplacesDemon function. For both 'algo.func="LA"' and 'algo.func="LD"' the LaplacesDemon package will be used (see http://www.bayesian-inference.com/software).
algo.method	Specifies the 'method' argument of optim function when using 'algo.func="optim"' (if not specified hyper.fit will use "Nelder-Mead" for optim). Specifies 'Method' argument of LaplaceApproximation function when using 'algo.func="LA"' (if not specified hyper.fit will use "NM" for LaplaceApproximation). Specifies 'Algorithm' argument of LaplacesDemon function when using 'algo.func="LD"' (if not specified hyper.fit will use "CHARM" for LaplacesDemon). When using 'algo.func="LD"' the user can also specify further options via the Specs argument below.
Specs	Inputs to pass to the LaplacesDemon function. Default Specs=list(alpha.star = 0.44) option is for the default CHARM algorithm (see algo.method above).
doerrorscale	If FALSE then the provided covariance are treated as it. If TRUE then the likelihood function is also allowed to rescale all errors by a uniform multiplicative value.
...	Further arguments to be passed to optim, LaplaceApproximation or LaplacesDemon depending on the option used for 'algo.func'.

Details

Setting doerrorscale to TRUE allows for stable solutions when errors are overestimated, e.g. when the intrinsic scatter is equal to zero but the data is more clustered around the optimal likelihood plane than expected from the data covariance array. See Examples below for a 2D scenario where this is helpful.

algo.func="LD" also returns the probability that the generative model has exactly zero intrinsic scatter (zeroscatprob in the output). The other available functions (algo.func="optim" and algo.func="LA") can find exact solutions equal to zero since they are strictly mode finding (i.e. maximum likelihood) routines. algo.func="LD" is MCMC based, so naturally returns a mean/expectation which *must* have a finite positive value for the intrinsic scatter (it's not allowed to travel below zero). The zeroscatprob output works better with a Metropolis type scheme, e.g. algo.method='CHARM', Specs=list(alpha.star = 0.44) works well for many cases.

Value

The function returns a multi-component list containing:

parm	Vector of the main paramter fit outputs specified as set by the coord.type and scat.type options.
parm.vert.axis	Vector of the main parameter fit outputs specified strictly along the last column dimension of X (for both the intrinsic scatter and the beta offset). The order of the columns in X therefore affects the contents of this vector. This output assume a coord.type="alpha" and scat.type="vert.axis".
fit	The direct output of the specified algo.func. So either the natural return from optim (class type "optim"), LaplaceApproximation (class type "laplace") or LaplacesDemon (class type "demonoid").

`sigcor`	If algo.func="optim" or algo.func="LA" then this list element contains the unbiased population estimator for the intrinsic scatter corrected via the sampbias2popunbias output of `hyper.sigcor`, i.e. it uses the full correction from hyper.sigcor (element 3 of the correction vector generated). If algo.func="LD" then this list element contains the unbiased population estimator for the intrinsic scatter corrected via the bias2unbias output of `hyper.sigcor`, i.e. it uses the standard deviation bias correction from hyper.sigcor (element 1 of the correction vector generated). See `hyper.sigcor` for details on the different outputs and the convergence tests the demonstrate why these different correction methods are desirable.
`parm.covar`	The covariance matrix for parm. Only for algo.func="optim" and algo.func="LA".
`zeroscatprob`	The fraction of samples for the intrinsic scatter which are at *exactly* zero, which provides a guideline probability for the intrinsic scatter being truly zero, rather than the expectation which will always be a finite amount above zero. Only for algo.func="LD". See Details above.
`hyper.plane`	Object class of type hyper.plane.param, as output by hyper.covert. This standardises the final fit in the users requested coord.type and scat.type, and allows easy conversion to other systems by using the class dependent convert function on it.
`N`	The number of rows of matrix X.
`dims`	The number of columns of matrix X.
`X`	The input matrix X.
`covarray`	The covarray used for fitting.
`weights`	The weights used for fitting.
`call`	The actual call used to run hyper.fit.
`args`	The arguments used to run hyper.fit.
`LL`	A list containing elements sum (the total log-likelihodd), val (the individual log likelihoods) and sig (the effective sigma offsets). See hyper.like for more information on these outputs.

Author(s)

Aaron Robotham and Danail Obreschkow

References

Robotham, A.S.G., & Obreschkow, D., PASA, in press

See Also

`hyper.basic`, `hyper.convert`, `hyper.data`, `hyper.fit`, `hyper.plot`, `hyper.sigcor`, `hyper.summary`

Examples

```
#### A very simple 2D example ####

#Make the simple data:

simpledata=cbind(x=1:10,y=c(12.2, 14.2, 15.9, 18.0, 20.1, 22.1, 23.9, 26.0, 27.9, 30.1))
simpfit=hyper.fit(simpledata)
summary(simpfit)
plot(simpfit)

#Increase the scatter:

simpledata2=cbind(x=1:10,y=c(11.6, 13.7, 15.5, 18.2, 21.2, 21.5, 23.6, 25.6, 27.9, 30.1))
simpfit2=hyper.fit(simpledata2)
summary(simpfit2)
plot(simpfit2)

#Assuming the error in each y data point is the same sy=0.5, we no longer need any
#component of intrinsic scatter to explain the data:

simpledata2err=cbind(sx=0, sy=rep(0.5, length(simpledata2[, 1])))
simpfit2werr=hyper.fit(simpledata2, vars=simpledata2err)
summary(simpfit2werr)
plot(simpfit2werr)

#We can fit for 6 different combinations of coordinate system:

print(hyper.fit(simpledata, coord.type='theta', scat.type='orth')$parm)
print(hyper.fit(simpledata, coord.type='alpha', scat.type='orth')$parm)
print(hyper.fit(simpledata, coord.type='normvec', scat.type='orth')$parm)
print(hyper.fit(simpledata, coord.type='theta', scat.type='vert.axis')$parm)
print(hyper.fit(simpledata, coord.type='alpha', scat.type='vert.axis')$parm)
print(hyper.fit(simpledata, coord.type='normvec', scat.type='vert.axis')$parm)

#These all describe the same hyperplane (or line in this case). We can convert between
#systems by using the hyper.convert utility function:

fit4normvert=hyper.fit(simpledata, coord.type='normvec', scat.type='vert.axis')$parm
hyper.convert(fit4normvert, in.coord.type='normvec', out.coord.type='theta',
in.scat.type='vert.axis', out.scat.type='orth')$parm

#### Simple Example in hyper.fit paper ####

#Fit with no error:

xval = c(-1.22, -0.78, 0.44, 1.01, 1.22)
yval = c(-0.15, 0.49, 1.17, 0.72, 1.22)

fitnoerror=hyper.fit(cbind(xval, yval))
plot(fitnoerror)

#Fit with independent x and y error:
```

```
xerr = c(0.12, 0.14, 0.20, 0.07, 0.06)
yerr = c(0.13, 0.03, 0.07, 0.11, 0.08)
fitwitherror=hyper.fit(cbind(xval, yval), vars=cbind(xerr, yerr)^2)
plot(fitwitherror)

#Fit with correlated x and y error:

xycor = c(0.90, -0.40, -0.25, 0.00, -0.20)
fitwitherrorandcor=hyper.fit(cbind(xval, yval), covarray=makecovarray2d(xerr, yerr, xycor))
plot(fitwitherrorandcor)

#### A 2D example with fitting a line ####

#Setup the initial data:

## Not run:

set.seed(650)
sampN=200
initscat=3
randatax=runif(sampN, -100, 100)
randatay=rnorm(sampN, sd=initscat)
sx=runif(sampN, 0, 10); sy=runif(sampN, 0, 10)

mockvararray=makecovarray2d(sx, sy, corxy=0)

errxy={}
for(i in 1:sampN){
  rancovmat=ranrotcovmat2d(mockvararray[,,i])
  errxy=rbind(errxy, mvrnorm(1, mu=c(0, 0), Sigma=rancovmat))
  mockvararray[,,i]=rancovmat
  }
randatax=randatax+errxy[,1]
randatay=randatay+errxy[,2]

#Rotate the data to an arbitrary angle theta:

ang=30
mock=rotdata2d(randatax, randatay, theta=ang)
xerrang={}; yerrang={}; corxyang={}
for(i in 1:sampN){
  covmatrot=rotcovmat(mockvararray[,,i], theta=ang)
  xerrang=c(xerrang, sqrt(covmatrot[1,1])); yerrang=c(yerrang, sqrt(covmatrot[2,2]))
  corxyang=c(corxyang, covmatrot[1,2]/(xerrang[i]*yerrang[i]))
}
corxyang[xerrang==0 & yerrang==0]=0
mock=data.frame(x=mock[,1], y=mock[,2], sx=xerrang, sy=yerrang, corxy=corxyang)

#Do the fit:

X=cbind(mock$x, mock$y)
covarray=makecovarray2d(mock$sx, mock$sy, mock$corxy)
```

```
fitline=hyper.fit(X=X, covarray=covarray, coord.type='theta')
summary(fitline)
plot(fitline, trans=0.2, asp=1)

#We can test to see if the errors are compatable with the intrinsic scatter:

fitlineerrscale=hyper.fit(X=X, covarray=covarray, coord.type='theta', doerrorscale=TRUE)
summary(fitlineerrscale)
plot(fitline, parm.errorscale=fitlineerrscale$parm['errorscale'], trans=0.2, asp=1)

#Within errors the errorscale parameter is 1, i.e. the errors are realistic, which we know
#they should be a priori since we made them ourselves.


## End(Not run)

#### A 2D example with exponential sampling & fitting a line ####

#Setup the initial data:

## Not run:

set.seed(650)

#The effect of an exponential density function along y is to offset the Gaussian mean by
#0.5 times the factor 'a' in exp(a*x), i.e.:

normfac=dnorm(0,sd=1.1)/(dnorm(10*1.1^2,sd=1.1)*exp(10*10*1.1^2))
magplot(seq(5,15,by=0.01), normfac*dnorm(seq(5,15, by=0.01), sd=1.1)*exp(10*seq(5,15,
by=0.01)), type='l')
abline(v=10*1.1^2,lty=2)

#The above will not be correctly normalised to form a true PDF, but the shift in the mean
#is clear, and it doesn't alter the standard deviation at all:

points(seq(5,15,by=0.1), dnorm(seq(5,15, by=0.1), mean=10*1.1^2, sd=1.1),col='red')

#Applying the same principal to our random data we apply the offset due to our exponential
#generative slope in y:

set.seed(650)

sampN=200
vert.scat=10
sampexp=0.1
ang=30
randatax=runif(200,-100,100)
randatay=randatax*tan(ang*pi/180)+rnorm(sampN, mean=sampexp*vert.scat^2, sd=vert.scat)
sx=runif(sampN, 0, 10); sy=runif(sampN, 0, 10)

mockvararray=makecovarray2d(sx, sy, corxy=0)

errxy={}
```

```
for(i in 1:sampN){
  rancovmat=ranrotcovmat2d(mockvararray[,,i])
  errxy=rbind(errxy, mvrnorm(1, mu=c(0, sampexp*sy[i]^2), Sigma=rancovmat))
  mockvararray[,,i]=rancovmat
  }
randatax=randatax+errxy[,1]
randatay=randatay+errxy[,2]
sx=sqrt(mockvararray[1,1,]); sy=sqrt(mockvararray[2,2,]); corxy=mockvararray[1,2,]/(sx*sy)
mock=data.frame(x=randatax, y=randatay, sx=sx, sy=sy, corxy=corxy)

#Do the fit. Notice that the second element of k.vec has the positive sign, i.e. we are moving
#data that has been shifted positively by the positive exponential slope in y back to where it
#would exist without the slope (i.e. if it had an equal chance of being scattered in both
#directions, rather than being preferentially offset in the direction away from denser data).
#This dense -> less-dense shift i s known as Eddington bias in astronomy, and is common in all
#power-law distributions that have intrinsic scatter (e.g. Schechter LF and dark matter HMF).

X=cbind(mock$x, mock$y)
covarray=makecovarray2d(mock$sx, mock$sy, mock$corxy)
fitlineexp=hyper.fit(X=X, covarray=covarray, coord.type='theta', k.vec=c(0,sampexp),
scat.type='vert.axis')
summary(fitlineexp)
plot(fitlineexp, k.vec=c(0,sampexp))

#If we ignore the k.vec when calculating the plotting sigma values you can see it has
#a significant effect:

plot(fitlineexp, trans=0.2, asp=1)

#Compare this to not including the known exponential slope:

fitlinenoexp=hyper.fit(X=X, covarray=covarray, coord.type='theta', k.vec=c(0,0),
scat.type='vert.axis')
summary(fitlinenoexp)
plot(fitlinenoexp, trans=0.2, asp=1)


## End(Not run)

#The theta and intrinsic scatter are similar, but the offset is shifted significantly
#away from zero.

#### A 3D example with fitting a plane ####

#Setup the initial data:

## Not run:

set.seed(650)
sampN=200
initscat=3
randatax=runif(sampN, -100, 100)
randatay=runif(sampN, -100, 100)
```

```
randataz=rnorm(sampN, sd=initscat)
sx=runif(sampN, 0, 5); sy=runif(sampN, 0, 5); sz=runif(sampN, 0, 5)

mockvararray=makecovarray3d(sx, sy, sz, corxy=0, corxz=0, coryz=0)

errxyz={}
for(i in 1:sampN){
  rancovmat=ranrotcovmat3d(mockvararray[,,i])
  errxyz=rbind(errxyz,mvrnorm(1, mu=c(0, 0, 0), Sigma=rancovmat))
  mockvararray[,,i]=rancovmat
  }
randatax=randatax+errxyz[,1]
randatay=randatay+errxyz[,2]
randataz=randataz+errxyz[,3]
sx=sqrt(mockvararray[1,1,]); sy=sqrt(mockvararray[2,2,]); sz=sqrt(mockvararray[3,3,])
corxy=mockvararray[1,2,]/(sx*sy); corxz=mockvararray[1,3,]/(sx*sz)
coryz=mockvararray[2,3,]/(sy*sz)

#Rotate the data to an arbitrary angle theta/phi:

desiredxtozang=10
desiredytozang=40
ang=c(desiredxtozang*cos(desiredytozang*pi/180), desiredytozang)
newxyz=rotdata3d(randatax, randatay, randataz, theta=ang[1], dim='y')
newxyz=rotdata3d(newxyz[,1], newxyz[,2], newxyz[,3], theta=ang[2], dim='x')
mockplane=data.frame(x=newxyz[,1], y=newxyz[,2], z=newxyz[,3])

xerrang={};yerrang={};zerrang={}
corxyang={};corxzang={};coryzang={}
for(i in 1:sampN){
  newcovmatrot=rotcovmat(makecovmat3d(sx=sx[i], sy=sy[i], sz=sz[i], corxy=corxy[i],
  corxz=corxz[i], coryz=coryz[i]), theta=ang[1], dim='y')
  newcovmatrot=rotcovmat(newcovmatrot, theta=ang[2], dim='x')
  xerrang=c(xerrang, sqrt(newcovmatrot[1,1]))
  yerrang=c(yerrang, sqrt(newcovmatrot[2,2]))
  zerrang=c(zerrang, sqrt(newcovmatrot[3,3]))
  corxyang=c(corxyang, newcovmatrot[1,2]/(xerrang[i]*yerrang[i]))
  corxzang=c(corxzang, newcovmatrot[1,3]/(xerrang[i]*zerrang[i]))
  coryzang=c(coryzang, newcovmatrot[2,3]/(yerrang[i]*zerrang[i]))
}
corxyang[xerrang==0 & yerrang==0]=0
corxzang[xerrang==0 & zerrang==0]=0
coryzang[yerrang==0 & zerrang==0]=0
mockplane=data.frame(x=mockplane$x, y=mockplane$y, z=mockplane$z, sx=xerrang, sy=yerrang,
sz=zerrang, corxy=corxyang, corxz=corxzang, coryz=coryzang)

X=cbind(mockplane$x, mockplane$y, mockplane$z)
covarray=makecovarray3d(mockplane$sx, mockplane$sy, mockplane$sz, mockplane$corxy,
mockplane$corxz, mockplane$coryz)
fitplane=hyper.fit(X=X, covarray=covarray, coord.type='theta', scat.type='orth')
summary(fitplane)
plot(fitplane)
```

```
## End(Not run)

#### Example using the data from Hogg 2010 ####

#Example using the data from Hogg 2010: http://arxiv.org/pdf/1008.4686v1.pdf

#Load data:

## Not run:

data(hogg)

#Fit:

fithogg=hyper.fit(X=cbind(hogg$x, hogg$y), covarray=makecovarray2d(hogg$x_err, hogg$y_err,
hogg$corxy), coord.type='theta', scat.type='orth')
summary(fithogg)
plot(fithogg, trans=0.2)

#We now do exercise 17 of Hogg 2010 using trimmed data:

hoggtrim=hogg[-3,]
fithoggtrim=hyper.fit(X=cbind(hoggtrim$x, hoggtrim$y), covarray=makecovarray2d(hoggtrim$x_err,
hoggtrim$y_err, hoggtrim$corxy), coord.type='theta', scat.type='orth', algo.func='LA')
summary(fithoggtrim)
plot(fithoggtrim, trans=0.2)

#We can get more info from looking at the Summary1 output of the LaplaceApproximation:

print(fithoggtrim$fit$Summary1)

#MCMC (exercise 18):

fithoggtrimMCMC=hyper.fit(X=cbind(hoggtrim$x, hoggtrim$y), covarray=
makecovarray2d(hoggtrim$x_err, hoggtrim$y_err, hoggtrim$corxy), coord.type='theta',
scat.type='orth', algo.func='LD', algo.method='CHARM', Specs=list(alpha.star = 0.44))
summary(fithoggtrimMCMC)

#We can get additional info from looking at the Summary1 output of the LaplacesDemon:

print(fithoggtrimMCMC$fit$Summary2)

magplot(density(fithoggtrimMCMC$fit$Posterior2[,3]), xlab='Intrinsic Scatter',
ylab='Probability Density')
abline(v=quantile(fithoggtrimMCMC$fit$Posterior2[,3], c(0.95,0.99)), lty=2)


## End(Not run)

#### Example using 'real' data with intrinsic scatter ####

## Not run:
```

```
data(intrin)

fitintrin=hyper.fit(X=cbind(intrin$x, intrin$y), vars=cbind(intrin$x_err,
intrin$y_err)^2, coord.type='theta', scat.type='orth', algo.func='LA')
summary(fitintrin)
plot(fitintrin, trans=0.1, pch='.', asp=1)

fitintrincor=hyper.fit(X=cbind(intrin$x, intrin$y), covarray=makecovarray2d(intrin$x_err,
intrin$y_err, intrin$corxy), coord.type='theta', scat.type='orth', algo.func='LA')
summary(fitintrincor)
plot(fitintrincor, trans=0.1, pch='.', asp=1)


## End(Not run)

#### Example using flaring trumpet data ####

## Not run:

data(trumpet)
fittrumpet=hyper.fit(X=cbind(trumpet$x, trumpet$y), covarray=makecovarray2d(trumpet$x_err,
trumpet$y_err, trumpet$corxy), coord.type='normvec', algo.func='LA')
summary(fittrumpet)
plot(fittrumpet, trans=0.1, pch='.', asp=1)

#The best fit solution has a scat.orth very close to 0, so it is worth considering if the
#data should truly have 0 intrinsic scatter.


## End(Not run)

## Not run:

#To find the likelihood of zero intrinsic scatter we will need to run LaplacesDemon. The
#following will take a couple of minutes to run:

set.seed(650)
fittrumpetMCMC=hyper.fit(X=cbind(trumpet$x, trumpet$y), covarray=makecovarray2d(trumpet$x_err,
trumpet$y_err, trumpet$corxy), coord.type='normvec', algo.func='LD', itermax=1e5)

#Assuming the user has specified the same initial seed we should find that the data
#has exactly zero intrinsic scatter with ~47% likelihood:

print(fittrumpetMCMC$zeroscatprob)

#We can also make an assessment of whether the data has even less scatter than expected
#given the expected errors:

set.seed(650)
fittrumpetMCMCerrscale=hyper.fit(X=cbind(trumpet$x, trumpet$y), covarray=makecovarray2d(
trumpet$x_err, trumpet$y_err, trumpet$corxy), itermax=1e5, coord.type='normvec', algo.func='LD',
algo.method='CHARM', Specs=list(alpha.star = 0.44), doerrorscale=TRUE)
```

```
#Assuming the user has specified the same initial seed we should find that the data
#has exactly zero intrinsic scatter with ~69% likelihood:

print(fittrumpetMCMCerrscale$zeroscatprob)


## End(Not run)

#### Example using 6dFGS Fundamental Plane data ####

## Not run:

data(FP6dFGS)

#First we try the fit without using any weights:

fitFP6dFGS=hyper.fit(FP6dFGS[,c('logIe_J', 'logsigma', 'logRe_J')],
vars=FP6dFGS[,c('logIe_J_err', 'logsigma_err', 'logRe_J_err')]^2, coord.type='alpha',
scat.type='vert.axis')
summary(fitFP6dFGS)
plot(fitFP6dFGS, doellipse=FALSE, alpha=0.5)


## End(Not run)

#Next we add the censoring weights provided by C. Magoulas:

## Not run:

fitFP6dFGSw=hyper.fit(FP6dFGS[,c('logIe_J', 'logsigma', 'logRe_J')],
vars=FP6dFGS[,c('logIe_J_err', 'logsigma_err', 'logRe_J_err')]^2, weights=FP6dFGS[,'weights'],
coord.type='alpha', scat.type='vert.axis')
summary(fitFP6dFGSw)
plot(fitFP6dFGSw, doellipse=FALSE, alpha=0.5)

#It is interesting to note the scatter orthogonal to the plane for the fundmental plane:

print(hyper.convert(coord=fitFP6dFGSw$parm[1:2], beta=fitFP6dFGSw$parm[3],
scat=fitFP6dFGSw$parm[4], in.scat.type='vert.axis', out.scat.type='orth',
in.coord.type='alpha'))


## End(Not run)

#### Example using GAMA mass-size relation data ####

## Not run:

data(GAMAsmVsize)
fitGAMAsmVsize=hyper.fit(GAMAsmVsize[,c('logmstar', 'rekpc')],
vars=GAMAsmVsize[,c('logmstar_err', 'rekpc_err')]^2, weights=GAMAsmVsize[,'weights'],
coord.type='alpha', scat.type='vert.axis')
```

```
summary(fitGAMAsmVsize)
#We turn the ellipse plotting off to speed things up:
plot(fitGAMAsmVsize, doellipse=FALSE, unlog='x')

#This is obviously a poor fit since the y data has a non-linear dependence on x. Let's try
#using the logged y-axis and converted errors:

fitGAMAsmVsizelogre=hyper.fit(GAMAsmVsize[,c('logmstar', 'logrekpc')],
vars=GAMAsmVsize[,c('logmstar_err', 'logrekpc_err')]^2, weights=GAMAsmVsize[,'weights'],
coord.type='alpha', scat.type='vert.axis')
summary(fitGAMAsmVsizelogre)
#We turn the ellipse plotting off to speed things up:
plot(fitGAMAsmVsizelogre, doellipse=FALSE, unlog='xy')

#We can compare to a fit with no errors used:

fitGAMAsmVsizelogrenoerr=hyper.fit(GAMAsmVsize[,c('logmstar', 'logrekpc')],
weights=GAMAsmVsize[,'weights'], coord.type='alpha', scat.type='vert.axis')
summary(fitGAMAsmVsizelogrenoerr)
#We turn the ellipse plotting off to speed things up:
plot(fitGAMAsmVsizelogrenoerr, doellipse=FALSE, unlog='xy')


## End(Not run)

### Example using Tully-Fisher relation data ###

## Not run:

data(TFR)
TFRfit=hyper.fit(X=TFR[,c('logv','M_K')],vars=TFR[,c('logv_err','M_K_err')]^2)
plot(TFRfit, xlim=c(1.7,2.5), ylim=c(-19,-26))


## End(Not run)

### Mase-Angular Momentum-Bulge/Total ###

## Not run:

data(MJB)
MJBfit=hyper.fit(X=MJB[,c('logM','logj','B.T')], covarray=makecovarray3d(MJB$logM_err,
MJB$logj_err, MJB$B.T_err, MJB$corMJ, 0, 0))
plot(MJBfit)


## End(Not run)
```

---

hyper.fit package — *Generic N-Dimensional Hyperplane Fitting with Heteroscedastic Covariant Errors and Intrinsic Scatter*

Description

Includes two main high level codes for hyperplane fitting (hyper.fit) and visualising (hyper.plot2d / hyper.plot3d). In simple terms this allows the user to produce robust 1D linear fits for 2D x vs y type data, and robust 2D plane fits to 3D x vs y vs z type data. This hyperplane fitting works generically for any N-1 hyperplane model being fit to a N dimension dataset. All fits include intrinsic scatter in the generative model orthogonal to the hyperplane.

Details

Package:	hyper.fit
Type:	Package
Version:	1.0.3
Date:	2016-08-01
License:	GPL-3

Most users will interact with this package through the high level hyper.fit and hyper.plot functions. The utility functions to construct covariance matrices and arrays will also be useful (see makecovmat2d and makecovarray2d).

Author(s)

Aaron Robotham and Danail Obreschkow

Maintainer: Aaron Robotham <aaron.robotham@uwa.edu.au>

References

Robotham, A.S.G., & Obreschkow, D., PASA, in press

See Also

hyper.basic, hyper.convert, hyper.data, hyper.fit, hyper.plot, hyper.sigcor, hyper.summary

---

hyper.like *The likelihood of a given set of data and an specified hyperplane*

Description

This is the mid-level likelihood solving function. Most users will not use this directly, but it is called by hyper.fit and hyper.plot2d/hyper.plot3d. Users can interact with the function directly, but it only takes arguments using the normal vector (coord.orth) orthogonal offset of the origin to the hyperplane (beta.orth) and orthogonal scatter to the hyperplane (scat.orth).

Usage

```
hyper.like(parm, X, covarray, weights = 1, errorscale = 1, k.vec = FALSE, output = 'sum')
```

Arguments

`parm` A vector specifying the current paramters to compute the likelihood for. This should be a concatentation of 'normvec' coordinates and 'scat.orth' intrinsic scatter. e.g. for 3d data this would look like c(normvec1, normvec2, normvec3, scat.orth).

`X` A position matrix with the N (number of data points) rows by d (number of dimensions) columns.

`covarray` A dxdxN array containing the full covariance (d=dimensions, N=number of dxd matrices in the array stack). The 'makecovarray2d' and 'makecovarray3d' are convenience functions that make populating 2x2xN and 3x3xN arrays easier for a novice user.

`weights` Vector of multiplicative weights for each row of the X data matrix. i.e. if this is 2 then it is equivalent to having two itentical data points with weights equal to 1. Should be either of length 1 (in which case elements are repeated as required) or the same length as the number of rows in the data matrix X.

`errorscale` Value to multiplicatively re-scale the errors by (i.e. the covariance array become scaled by errorscale^2). This might be useful when trying to decide if the provided errors are too large.

`k.vec` A vector defining the direction of an exponential sampling distribution in the data. The length is the magnitude of the exponent, and it points in the direction of *increasing* density (see example below). Must be the same length as coord.orth, or FALSE.

`output` If 'sum' then the output is the sum of the log likelihood. If 'val' then the output is the individual log likelihoods of the data provided. If 'sig' then the output represents the sigma tension of the data point with the current model, and can be thought of as -0.5chi^2 for each data point.

Details

hyper.convert is a convenience function that manipulates different parameterisations into the type required for the parm argument of hyper.like. See example below.

Value

If output='sum' then the output is the sum of the log likelihood. If output='val' then the output is the individual log likelihoods of the data provided. If output='sig' then the output represents the sigma tension of the data point with the current model, and can be thought of as -0.5chi^2 for each data point.

Author(s)

Aaron Robotham and Danail Obreschkow

References

Robotham, A.S.G., & Obreschkow, D., PASA, in press

See Also

hyper.basic, hyper.convert, hyper.data, hyper.fit, hyper.plot, hyper.sigcor, hyper.summary

Examples

```
#Setup the initial data:

set.seed(650)
sampN=200
initscat=3
randatax=runif(sampN, -100,100)
randatay=rnorm(sampN, sd=initscat)
sx=runif(sampN, 0,10); sy=runif(sampN, 0,10)

mockvararray=makecovarray2d(sx, sy, corxy=0)

errxy={}
for(i in 1:sampN){
  rancovmat=ranrotcovmat2d(mockvararray[,,i])
  errxy=rbind(errxy, mvrnorm(1,mu=c(0,0), Sigma=rancovmat))
  mockvararray[,,i]=rancovmat
  }
randatax=randatax+errxy[,1]
randatay=randatay+errxy[,2]

#Rotate the data to an arbitrary angle theta:

ang=30
mock=rotdata2d(randatax, randatay, theta=ang)
xerrang={}; yerrang={}; corxyang={}
for(i in 1:sampN){
  covmatrot=rotcovmat(mockvararray[,,i], theta=ang)
  xerrang=c(xerrang, sqrt(covmatrot[1,1])); yerrang=c(yerrang, sqrt(covmatrot[2,2]))
  corxyang=c(corxyang, covmatrot[1,2]/(xerrang[i]*yerrang[i]))
}
corxyang[xerrang==0 & yerrang==0]=0
mock=data.frame(x=mock[,1], y=mock[,2], sx=xerrang, sy=yerrang, corxy=corxyang)

#Do the fit:

X=cbind(mock$x, mock$y)
covarray=makecovarray2d(mock$sx, mock$sy, mock$corxy)

#Create our orthogonal vector. This does not need to be normalised to 1:

coord.orth=hyper.convert(coord=ang, in.coord.type = "theta", out.coord.type = "normvec")

#Feed this into the hyper.like function:

print(hyper.like(parm=coord.orth$parm, X, covarray, errorscale=1, output = "sum"))

#Comapre to a worse option:
```

```
print(hyper.like(parm=c(0.5, -1, 0, 4), X, covarray, errorscale=1, output = "sum"))

#As we can see, the paramters used to generate the data produce a higher likelihood.
```

---

hyper.plot *A 2d and 3d likelihood diagnostic plot for optimal line fitting*

Description

These functions produce helpful 2d and 3d diagnostic plots for post hyper.fit analysis and for manual experimentation with parameter options. Error llipses and ellipsodis are added to the plots, with colouring scaled by 'sigma-tension' of the data points (where red is high tension). It also overplots the current line (2d) or plane (3d). If the data is either 2d/3d then a simple interface to the relevant plot.hyper function is provided by the hyper.fit class dependent function plot.hyper.fit, where the user only has to execute plot(fitoutput).

Usage

```
## S3 method for class 'hyper.fit'
plot(x, ...)

hyper.plot2d(X, covarray, vars, fitobj, parm.coord, parm.beta, parm.scat,
parm.errorscale = 1, vert.axis, weights, k.vec, coord.type = 'alpha', scat.type = 'orth',
doellipse = TRUE, sigscale=c(0,4), trans=1, dobar=FALSE, position='topright', ...)

hyper.plot3d(X, covarray, vars, fitobj, parm.coord, parm.beta, parm.scat,
parm.errorscale = 1, vert.axis, weights, k.vec, coord.type = 'alpha', scat.type = 'orth',
doellipse = TRUE, sigscale=c(0,4),trans=1, ...)
```

Arguments

x	Argument for the class dependent plot.hyper.fit function. An object of class hyper.fit. This is the only structure that needs to be provided when executing plot(fitobj) class dependent plotting, which will use the plot.hyper.fit function.
X	A position matrix with the N (number of data points) rows by d (number of dimensions) columns.
covarray	A dxdxN array containing the full covariance (d=dimensions, N=number of dxd matrices in the array stack). The 'makecovarray2d' and 'makecovarray3d' are convenience functions that make populating 2x2xN and 3x3xN arrays easier for a novice user.
vars	A variance matrix with the N (numver of data points) rows by Dim (number of dimensions) columns. In effect this is the diagonal elements of the 'covarray' array where all other terms are zero. If 'covarray' is also provided that is used instead.

fitobj
: For simplicity the user can provide the direct output of hyper.fit to this argument, which sets the hyperplane parameter values to those found during the fitting process. If this is not provided then parm.coord / parm.beta / parm.scat must all be specified.

parm.coord
: Vector of initial coord paramters. These are either angles that produce the vectors that predict the vert.axis dimension (coord.type='theta'), the gradients of these (coord.type='alpha') or they are the unit vector normal to the hyperplane (coord.type='unitvec').

parm.beta
: Initial value of beta. This is either specified as the absolute distance from the origin to the hyperplane or as the intersection of the hyperplane on the vert.axis dimension being predicted.

parm.scat
: Initial value of the intrinsic scatter. This is either specified as the scatter orthogonal to the hyperplane or as the scatter along the vert.axis dimension being predicted.

parm.errorscale
: Value to multiplicatively rescale the errors by (i.e. the covarince array becomes scaled by errorscale^2). This might be useful when trying to decide if the provided errors are too large. Default is 1, and therefore it does not need to be specified explicitly.

vert.axis
: Which axis should the plane equation be formulated for. This must be a number which specifies the column of position matrix 'X' to be defined by the hyperplane. If missing, then the projection dimension is assumed to be the last column of the 'X' matrix.

weights
: Vector of multiplicative weights for each row of the X data matrix. i.e. if this is 2 then it is equivalent to having two itentical data points with weights equal to 1. Should be either of length 1 (in which case elements are repeated as required) or the same length as the number of rows in the data matrix X.

k.vec
: A vector defining the direction of an exponential sampling distribution in the data. The length is the scaling 'a' of the generative exponent (i.e., exp(a*x)), and it points in the direction of *increasing* density (see example below). If provided, k.vec must be the same length as the dimensions of the data. k.vec has the most noticeable effect on the beta offset parameter. It is correcting for the phenomenom Astronomers often call Eddington bias.

coord.type
: This specifies whether the parm.coord parameter is defined in terms of the unit vector of the line (alpha) or for the values of the angles that form the unit vector (theta).

scat.type
: This specifies whether the parm.beta and the parm.scat are defined as orthogonal to the plane (orth) or along the vert.axis of interest (vert.axis).

doellipse
: Should 2d/3d error ellipses be drawn on the plot? This should only be TRUE if data errors are actually provided, else it should be FALSE.

sigscale
: Vector of length 2 specifying the lower and upper limits for the linear blue->green->red mapping used to colour the data. The default will map to 0->2->4 sigma offset tension.

trans
: Transparency of the ellipses (hyper.plot2d) or ellipsoids (hyper.plot3d).

`dobar` Logical specifying whether a magbar colour scale is added to the 2D plot. Only available for hyper.plot2d.

`position` If dobar=TRUE, then position specifies where the magbar colour scale is placed within the plot. Specify one of 'bottom', 'bottomleft', 'left', 'topleft', 'top', 'topright', 'right', 'bottomright' and 'centre'.

`...` Arguments to pass to magplot (hyper.plot2d) or plot3d (hyper.plot3d). When executing the hyper.fit class function plot.hyper.fit dots/ellipses are first passed to hyper.plot2d / hyper.plot3d and then onto magplot / plot3d for any unmatched arguments.

Value

The plotting functions also return the sigma tension of the data points given the inputs. i.e. the output of hyper.like with output='sig'.

Author(s)

Aaron Robotham and Danail Obreschkow

References

Robotham, A.S.G., & Obreschkow, D., PASA, in press

See Also

hyper.basic, hyper.convert, hyper.data, hyper.fit, hyper.plot, hyper.sigcor, hyper.summary

Examples

```
#### A very simple 2D example ####

#Make the simple data:

simpledata=cbind(x=1:10,y=c(12.2, 14.2, 15.9, 18.0, 20.1, 22.1, 23.9, 26.0, 27.9, 30.1))
simpfit=hyper.fit(simpledata)
summary(simpfit)
plot(simpfit)

#Increase the scatter:

simpledata2=cbind(x=1:10,y=c(11.6, 13.7, 15.5, 18.2, 21.2, 21.5, 23.6, 25.6, 27.9, 30.1))
simpfit2=hyper.fit(simpledata2)
summary(simpfit2)
plot(simpfit2)

#Assuming the error in each y data point is the same sy=0.5, we no longer need any
#component of intrinsic scatter to explain the data:

simpledata2err=cbind(sx=0, sy=rep(0.5, length(simpledata2[, 1])))
simpfit2werr=hyper.fit(simpledata2, vars=simpledata2err)
summary(simpfit2werr)
```

```
plot(simpfit2werr)

#### Simple Example in hyper.fit paper ####

#Fit with no error:

xval = c(-1.22, -0.78, 0.44, 1.01, 1.22)
yval = c(-0.15, 0.49, 1.17, 0.72, 1.22)

fitnoerror=hyper.fit(cbind(xval, yval))
plot(fitnoerror)

#Fit with independent x and y error:

xerr = c(0.12, 0.14, 0.20, 0.07, 0.06)
yerr = c(0.13, 0.03, 0.07, 0.11, 0.08)
fitwitherror=hyper.fit(cbind(xval, yval), vars=cbind(xerr, yerr)^2)
plot(fitwitherror)

#Fit with correlated x and y error:

xycor = c(0.90, -0.40, -0.25, 0.00, -0.20)
fitwitherrorandcor=hyper.fit(cbind(xval, yval), covarray=makecovarray2d(xerr, yerr, xycor))
plot(fitwitherrorandcor)

#### A 2D example with fitting a line ####

#Setup the initial data:

set.seed(650)
sampN=200
initscat=3
randatax=runif(sampN, -100, 100)
randatay=rnorm(sampN, sd=initscat)
sx=runif(sampN, 0, 10); sy=runif(sampN, 0, 10)

mockvararray=makecovarray2d(sx, sy, corxy=0)

errxy={}
for(i in 1:sampN){
  rancovmat=ranrotcovmat2d(mockvararray[,,i])
  errxy=rbind(errxy, mvrnorm(1, mu=c(0, 0), Sigma=rancovmat))
  mockvararray[,,i]=rancovmat
  }
randatax=randatax+errxy[,1]
randatay=randatay+errxy[,2]

#Rotate the data to an arbitrary angle theta:

ang=30
mock=rotdata2d(randatax, randatay, theta=ang)
xerrang={}; yerrang={}; corxyang={}
for(i in 1:sampN){
```

```
  covmatrot=rotcovmat(mockvararray[,,i], theta=ang)
  xerrang=c(xerrang, sqrt(covmatrot[1,1])); yerrang=c(yerrang, sqrt(covmatrot[2,2]))
  corxyang=c(corxyang, covmatrot[1,2]/(xerrang[i]*yerrang[i]))
}
corxyang[xerrang==0 & yerrang==0]=0
mock=data.frame(x=mock[,1], y=mock[,2], sx=xerrang, sy=yerrang, corxy=corxyang)

#Do the fit:

X=cbind(mock$x, mock$y)
covarray=makecovarray2d(mock$sx, mock$sy, mock$corxy)
fitline=hyper.fit(X=X, covarray=covarray, coord.type='theta')
hyper.plot2d(X=X, covarray=covarray, fitobj=fitline, trans=0.2, asp=1)
#Or even easier:
plot(fitline, trans=0.2, asp=1)

#### A 3D example with fitting a plane ####

## Not run:

#Setup the initial data:

set.seed(650)
sampN=200
initscat=3
randatax=runif(sampN, -100, 100)
randatay=runif(sampN, -100, 100)
randataz=rnorm(sampN, sd=initscat)
sx=runif(sampN, 0, 5); sy=runif(sampN,0,5); sz=runif(sampN, 0, 5)

mockvararray=makecovarray3d(sx, sy, sz, corxy=0, corxz=0, coryz=0)

errxyz={}
for(i in 1:sampN){
  rancovmat=ranrotcovmat3d(mockvararray[,,i])
  errxyz=rbind(errxyz, mvrnorm(1, mu=c(0, 0, 0), Sigma=rancovmat))
  mockvararray[,,i]=rancovmat
  }
randatax=randatax+errxyz[,1]
randatay=randatay+errxyz[,2]
randataz=randataz+errxyz[,3]
sx=sqrt(mockvararray[1,1,]); sy=sqrt(mockvararray[2,2,]); sz=sqrt(mockvararray[3,3,])
corxy=mockvararray[1,2,]/(sx*sy); corxz=mockvararray[1,3,]/(sx*sz)
coryz=mockvararray[2,3,]/(sy*sz)

#Rotate the data to an arbitrary angle theta/phi:
desiredxtozang=10
desiredytozang=40
ang=c(desiredxtozang*cos(desiredytozang*pi/180), desiredytozang)
newxyz=rotdata3d(randatax, randatay, randataz, theta=ang[1], dim='y')
newxyz=rotdata3d(newxyz[,1], newxyz[,2], newxyz[,3], theta=ang[2], dim='x')
mockplane=data.frame(x=newxyz[,1], y=newxyz[,2], z=newxyz[,3])
```

```
xerrang={}; yerrang={}; zerrang={}
corxyang={}; corxzang={}; coryzang={}
for(i in 1:sampN){
  newcovmatrot=rotcovmat(makecovmat3d(sx=sx[i], sy=sy[i], sz=sz[i], corxy=corxy[i],
  corxz=corxz[i], coryz=coryz[i]), theta=ang[1], dim='y')
  newcovmatrot=rotcovmat(newcovmatrot, theta=ang[2], dim='x')
  xerrang=c(xerrang, sqrt(newcovmatrot[1,1]))
  yerrang=c(yerrang, sqrt(newcovmatrot[2,2]))
  zerrang=c(zerrang, sqrt(newcovmatrot[3,3]))
  corxyang=c(corxyang, newcovmatrot[1,2]/(xerrang[i]*yerrang[i]))
  corxzang=c(corxzang, newcovmatrot[1,3]/(xerrang[i]*zerrang[i]))
  coryzang=c(coryzang, newcovmatrot[2,3]/(yerrang[i]*zerrang[i]))
}
corxyang[xerrang==0 & yerrang==0]=0
corxzang[xerrang==0 & zerrang==0]=0
coryzang[yerrang==0 & zerrang==0]=0
mockplane=data.frame(x=mockplane$x, y=mockplane$y, z=mockplane$z, sx=xerrang, sy=yerrang,
sz=zerrang, corxy=corxyang, corxz=corxzang, coryz=coryzang)

X=cbind(mockplane$x, mockplane$y, mockplane$z)
covarray=makecovarray3d(mockplane$sx, mockplane$sy, mockplane$sz, mockplane$corxy,
mockplane$corxz, mockplane$coryz)
fitplane=hyper.fit(X=X, covarray=covarray, coord.type='theta', scat.type='orth')
hyper.plot3d(X=X, covarray=covarray, fitobj=fitplane)
#Or even easier:
plot(fitplane)

## End(Not run)

#### Example using the data from Hogg 2010 ####

#Example using the data from Hogg 2010: http://arxiv.org/pdf/1008.4686v1.pdf

#Full data

## Not run:

data(hogg)
fithogg=hyper.fit(X=cbind(hogg$x, hogg$y), covarray=makecovarray2d(hogg$x_err, hogg$y_err,
hogg$corxy), coord.type='theta', scat.type='orth')
hyper.plot2d(X=cbind(hogg$x, hogg$y), covarray=makecovarray2d(hogg$x_err, hogg$y_err,
hogg$corxy), fitobj=fithogg, trans=0.2, xlim=c(0, 300), ylim=c(0, 700))
#Or even easier:
plot(fithogg, trans=0.2)

#We now do exercise 17 of Hogg 2010 using trimmed data, where we remove the high tension
#data point 3 (which we can see as the reddest point in the above plot:

data(hogg)
hoggtrim=hogg[-3,]
fithoggtrim=hyper.fit(X=cbind(hoggtrim$x, hoggtrim$y), covarray=makecovarray2d(hoggtrim$x_err,
hoggtrim$y_err, hoggtrim$corxy), coord.type='theta', scat.type='orth', algo.func='LA')
hyper.plot2d(X=cbind(hoggtrim$x, hoggtrim$y), covarray=makecovarray2d(hoggtrim$x_err,
```

```
hoggtrim$y_err, hoggtrim$corxy), fitobj=fithoggtrim, trans=0.2, xlim=c(0, 300), ylim=c(0, 700))
#Or even easier:
plot(fithoggtrim, trans=0.2)

#We can compare this against the previous fit with:
hyper.plot2d(cbind(hoggtrim$x, hoggtrim$y), covarray=makecovarray2d(hoggtrim$x_err,
hoggtrim$y_err, hoggtrim$corxy), fitobj=fithogg, trans=0.2, xlim=c(0, 300), ylim=c(0, 700))


## End(Not run)

#### Example using 'real' data with intrinsic scatter ####

## Not run:

data(intrin)
fitintrin=hyper.fit(X=cbind(intrin$x, intrin$y), vars=cbind(intrin$x_err,
intrin$y_err)^2, coord.type='theta', scat.type='orth', algo.func='LA')
hyper.plot2d(cbind(intrin$x, intrin$y), covarray=makecovarray2d(intrin$x_err,
intrin$y_err, intrin$corxy), fitobj=fitintrin, trans=0.1, pch='.', asp=1)
#Or even easier:
plot(fitintrin, trans=0.1, pch='.', asp=1)


## End(Not run)

#### Example using flaring trumpet data ####

## Not run:

data(trumpet)
fittrumpet=hyper.fit(X=cbind(trumpet$x, trumpet$y), covarray=makecovarray2d(trumpet$x_err,
trumpet$y_err, trumpet$corxy), coord.type='theta', algo.func='LA')
hyper.plot2d(cbind(trumpet$x, trumpet$y), covarray=makecovarray2d(trumpet$x_err,
trumpet$y_err, trumpet$corxy), fitobj=fittrumpet, trans=0.1, pch='.', asp=1)
#Or even easier:
plot(fittrumpet, trans=0.1, pch='.', asp=1)
#If you look at the ?hyper.fit example we find that zero intrinsic scatter is actually
#preferred, but we don't see this in the above plot.


## End(Not run)

#### Example using 6dFGS Fundamental Plane data ####

## Not run:

data(FP6dFGS)
fitFP6dFGSw=hyper.fit(FP6dFGS[,c('logIe_J', 'logsigma', 'logRe_J')],
vars=FP6dFGS[,c('logIe_J_err', 'logsigma_err', 'logRe_J_err')]^2, weights=FP6dFGS[,'weights'],
coord.type='alpha', scat.type='vert.axis')
#We turn the ellipse plotting off to speed things up:
plot(fitFP6dFGSw, doellipse=FALSE, alpha=0.5)
```

```
## End(Not run)

#### Example using GAMA mass-size relation data ####

## Not run:

data(GAMAsmVsize)
fitGAMAsmVsize=hyper.fit(GAMAsmVsize[,c('logmstar', 'rekpc')],
vars=GAMAsmVsize[,c('logmstar_err', 'rekpc_err')]^2, weights=GAMAsmVsize[,'weights'],
coord.type='alpha', scat.type='vert.axis')
#We turn the ellipse plotting off to speed things up:
plot(fitGAMAsmVsize, doellipse=FALSE, unlog='x')

#This is obviously a poor fit since the y data has a non-linear dependence on x. Let's try
#using the logged y-axis and converted errors:

fitGAMAsmVsizelogre=hyper.fit(GAMAsmVsize[,c('logmstar', 'logrekpc')],
vars=GAMAsmVsize[,c('logmstar_err', 'logrekpc_err')]^2, weights=GAMAsmVsize[,'weights'],
coord.type='alpha', scat.type='vert.axis')
#We turn the ellipse plotting off to speed things up:
plot(fitGAMAsmVsizelogre, doellipse=FALSE, unlog='xy')

#We can compare to a fit with no errors used:

fitGAMAsmVsizelogrenoerr=hyper.fit(GAMAsmVsize[,c('logmstar', 'logrekpc')],
weights=GAMAsmVsize[,'weights'], coord.type='alpha', scat.type='vert.axis')
#We turn the ellipse plotting off to speed things up:
plot(fitGAMAsmVsizelogrenoerr, doellipse=FALSE, unlog='xy')


## End(Not run)

### Example using Tully-Fisher relation data ###

## Not run:

data(TFR)
TFRfit=hyper.fit(X=TFR[,c('logv','M_K')],vars=TFR[,c('logv_err','M_K_err')]^2)
plot(TFRfit, xlim=c(1.7,2.5), ylim=c(-19,-26))


## End(Not run)
```

---

hyper.sigcor *Function to convert from biased sample sigma to unbiased population sigma*

Description

Calculates the required corrections for transforming the biased estimate of sigma to an unbiased estimate, and for transforming the sample expectation to the population expectation.

Usage

```
hyper.sigcor(N,parmDF)
```

Arguments

N	The number of data points in the sample where sigma is being estimated.
parmDF	The number of degrees of freedom for the parameters. In the case of fitting a 1d Gaussian to data via maximum likelihood (equivalent the measuring the standard deviation of the data) parmDF=1, when fitting a 1d line with intrinsic scatter parmDF=2 and when fitting a plane to 3d data with intrinsic scatter parmDF=3.

Value

A vector of length 3. The first element is the correction from the biased to unbiased estimate for sigma. The second element is the correction from the sample to population estimate for sigma. The third is the combination of the previous two (i.e. the total correction the user will typically want to apply to their data).

Author(s)

Aaron Robotham and Danail Obreschkow

References

Robotham, A.S.G., & Obreschkow, D., PASA, in press

See Also

hyper.basic, hyper.convert, hyper.data, hyper.fit, hyper.plot, hyper.sigcor, hyper.summary

Examples

```
#The below will take *a long* time to run- of the order a few days for the LD tests.
## Not run:
Ngen=1e3
sdsamp=3
testvec=c(5,10,20,50,100)

set.seed(650)

fittestmean={}
for(Nsamp in testvec){
  print(paste('Nsamp=', Nsamp))
  fittest=matrix(0, nrow=Ngen, ncol=3)
  for(i in 1:Ngen){
```

```
    if(i
    mockx=runif(Nsamp, -100, 100)
    mocky=mockx*tan(45*pi/180)+rnorm(Nsamp, sd=sdsamp)
   fittest[i,]=hyper.fit(X=cbind(mockx,mocky), coord.type='theta', scat.type='vert.axis')$parm
  }
  convtest2dOpt=rbind(convtest2dOpt, c(N=Nsamp,Raw=mean(fittest[,3]),
  mean(fittest[,3])*hyper.sigcor(Nsamp, 2)))
}

fittestmeanLD={}
for(Nsamp in testvec){
  print(paste('Nsamp=', Nsamp))
  fittest=matrix(0, nrow=Ngen, ncol=3)
  for(i in 1:Ngen){
    if(i
    mockx=runif(Nsamp, -100, 100)
    mocky=mockx*tan(45*pi/180)+rnorm(Nsamp, sd=sdsamp)
   fittest[i,]=hyper.fit(X=cbind(mockx,mocky), coord.type='theta', scat.type='vert.axis',
    algo.func='LD', algo.method='GG', Specs=list(Grid=seq(-0.1,0.1, len=5), dparm=NULL,
    CPUs=1, Packages=NULL, Dyn.libs=NULL))$parm
    print(fittest[i,])
  }
  convtest2dLD=rbind(convtest2dLD, c(N=Nsamp, Raw=mean(fittest[,3]),
  mean(fittest[,3])*hyper.sigcor(Nsamp, 2)))
}

normtestmean={}
for(Nsamp in testvec){
  print(paste('Nsamp=', Nsamp))
  normtest={}
  for(i in 1:Ngen){
    if(i
    normtemp=rnorm(Nsamp, sd=sdsamp)
    normtest=c(normtest, sqrt(sum((normtemp-mean(normtemp))^2)/Nsamp))
  }
  convtest1dNorm=rbind(convtest1dNorm, c(N=Nsamp, Raw=mean(normtest),
  mean(normtest)*hyper.sigcor(Nsamp, 1)))
}

## End(Not run)
#The runs above have been pre-generated and can be loaded via

data(convtest2dOpt)
data(convtest2dLD)
data(convtest1dNorm)

magplot(convtest2dOpt[,c('N','Raw')],xlim=c(5,100),ylim=c(0,4),type='b',log='x')
lines(convtest2dOpt[,c('N','sampbias2popunbias')],type='b',lty=2,pch=4)
lines(convtest2dLD[,c('N','Raw')],type='b',col='blue')
lines(convtest2dLD[,c('N','bias2unbias')],type='b',lty=2,pch=4,col='blue')
lines(convtest1dNorm[,c('N','Raw')],type='b',col='red')
lines(convtest1dNorm[,c('N','sampbias2popunbias')],type='b',lty=2,pch=4,col='red')
legend('topleft', legend=c('2 DoF and optim fit','2 DoF and LD fit', '1 DoF and direct SD'),
```

```
col=c('black','blue','red'),pch=1)
legend('topright', legend=c('Raw intrinsic scatter', 'Corrected intrinsic scatter'),
lty=c(1,2))
```

---

hyper.summary *Summary function for hyper.fit object*

Description

Prints out basic summary information for hyper.fit objects output by the hyper.fit function.

Usage

```
## S3 method for class 'hyper.fit'
summary(object, ...)
```

Arguments

object	An object of class hyper.fit. This is the only structure that needs to be provided when executing summary(fitobj) class dependent plotting, which will use the summary.hyper.fit function.
...	Arguments passed to summary function.

Details

Outputs basic summary of the hyper.fit output.

Value

Prints various summary outputs.

Author(s)

Aaron Robotham and Danail Obreschkow

References

Robotham, A.S.G., & Obreschkow, D., PASA, in press

See Also

hyper.basic, hyper.convert, hyper.data, hyper.fit, hyper.plot, hyper.sigcor, hyper.summary

Examples

```
#### Example using 6dFGS Fundamental Plane data ####

FP6dFGS=read.table(system.file('data/FP6dFGS.tab', package='hyper.fit'), header=TRUE)
fitFP6dFGSw=hyper.fit(FP6dFGS[,c('logIe_J', 'logsigma', 'logRe_J')],
vars=FP6dFGS[,c('logIe_J_err', 'logsigma_err', 'logRe_J_err')]^2, weights=FP6dFGS[,'weights'],
coord.type='alpha', scat.type='vert.axis')
summary(fitFP6dFGSw)
```

Index