RSiena

Ruth M. Ripley*
modified by Tom A.B. Snijders†

November 14, 2021

1. Grand overview of RSiena

This overview of RSiena sometimes refers to the earlier version Siena3 to facilitate understanding by those who have a knowledge of the code of Siena3. Readers who have no such knowledge can happily ignore these references.

Two special features of the RSiena package are that it uses C++ code for time-consuming parts, and that a wrapper is available to use the package seemingly independently of R.

The script file *sienascript* starts up R in such a way that it launches a tcl/tk graphical user interface (gui) resembling the Stocnet interface for Siena3, and controls all processing thereafter. This interface is accessible on Windows via the command *siena01Gui*. (Prior to R 2.12.0 the funciton `siena.exe` was provided as an equivalent to `sienascript`: it has been removed as it was rarely used and proved difficult to maintain.)

The high-level functions called by the gui, such as *siena07* described below, are also accessible within R with the usual R-type user interface along the lines of model-fitting functions such as `lm()`. (A formula interface is still on the wish list, but already now it is relatively straightforward to use the package without the gui.)

The R functions call C++ only where speed is critical. From my profiling of *siena07*, the estimation function, I think that only the simulation function (which simply performs one simulation of the complete model for a given

*University of Oxford
†Universities of Oxford and Groningen

set of parameters and returns the statistics from the simulated networks) needs to be in C++. The bulk of the time is spent in calculating the contributions to the effects when simulating.

In Siena3, this simulation function is FRAN which, for method of moments estimation, simply calls simstats: in RSiena, it is the C++ function *model* called by one of the R functions *simstats0c* or *maxlikec*, which are the two candidates currently available for the element of the RSiena model object named *FRAN*. In this document I have used the name FRAN to refer to the simulation routine.

siena07 is intended to be written in such a way that different simulation functions could be used within the same Robbins Monro algorithm. In practice the separation is not quite complete, but it is nearly so.

It would be feasible within a C++ FRAN to call R functions for some effects or functions if desired, to facilitate adding new ones, although they would be slow to run!

We use functions from the C part of R , to provide random numbers within the C++.1 *simstats0c* and *maxlikec* have three types of calls: a initial one which calls various C++ routines to setup the data, a final one which sets the C++ data pointers to null to clean up the C++ memory, and multiple "normal" ones which call the function *model* to perform one complete simulation. (In this it does not correspond exactly to simstats in Siena3!)

With this design, we have introduced parallel processing by using multiple R processes. In *simstats0c* we run some of the simulations in each process: this was trivial to introduce into Phase 1 and Phase 3, but to use it in Phase2 we have altered the algorithm to use the average of more than one simulation at a time.

For *maxlikec* we use different processes for each wave. This is because the chains are carried from one simulation to the next, and organizing the parallel processes by simulations (update steps of the Robbins Monro algorithm) would require too much passing of information.

We use the R package `parallel` to create and control the multiple processes, and to provide multiple random number streams. The term 'cluster' below refers to the cluster of multiple processors. (I vary between using *processors* and *processes* as it is possible to run RSiena with multiple *processes* on a machine with only one processor.)

2. Data types

RSiena provides various classes of data objects, designed to interface with the functions *robmon* and simstats. A brief list:

siena Data for a single project

sienaGroup A list of *siena* objects, with global attributes, used for multi-group projects

sienaEffects Data frame of effects.

sienaGroupEffects Data frame of effects for a group object.

sienaModel Contains the fitting options.

sienaNodeSet Actor set, used to distinguish nodes in data sets with multiple or two-mode networks.

sienaDependent A single dependent variable, (i.e. network or behavior variable, all waves)

coCovar Constant covariate

coDyadCovar Constant dyadic covariate

varCovar Varying covariate

varDyadCovar Varying dyadic covariate

sienaCompositionChange List of changes, entry for each node.

sienaFit Currently contains (almost) everything from the estimation.

The structure of each is documented in the corresponding R help file: `?classname`.

Data objects of class *siena* are created by the function *sienaDataCreate*.

Effect objects of class *sienaEffects* are created by the function *getEffects*.

The creation functions can be called directly by the user or from the Gui or via *sienaDataCreateFromSession*, depending on whether the data is alrady in R objects or still on files.

The function *robmon* requires a *sienaModel* object as an argument. One element of this object is named FRAN and contains the name of the required simulation function.

The functions*simstats0c* or *maxlikec*, used as an instance of FRAN, require a *siena* (or *sienaGroup*) object and a *sienaEffects* (or *sienaGroupEffects*) object as arguments.

3. sienaDataCreate

This function has only one named argument: a list of actor (node) sets. The default is a single set of the required size named `Actors`. All other arguments are unnamed and correspond to networks, covariates or composition change files. The objects are validated and have various attributes added. For covariates the attributes are added using a *method* for their class.

Check that objects have names, using the object name if none is given in the function call.
if no objects **then**
    stop
if any duplicate names **then**
    stop
create a list of each type of object, checking that all dependent variables have the same number of observations. (Stop if not).
if no dependent variables **then**
    stop
if no node set argument **then**
    create a list of nodesets containing a single nodeset named Actors, with the number of nodes equal to the number of senders of the first dependent variable
for all covariates **do**
    Appropriate validation and processing (see below).
Process any composition change objects (see section 3.2)
Process the dependent variables (see section 3.4)
Check constraints if there are multiple networks. (section 3.8).
Calculate similarity means for alters for each covariate and dependent network (see section 3.9): dropped in version 1.1-285.

3.1 Covariates

3.1.1 Constant Covariate

Check the nodeset (section 3.3)
Create attributes: (a class method)
mean ignore missings
range Extent of range, ignore missings. Make sure is a double.
range2 Ends of range, ignore missings
moreThan2 TRUE if more than 2 distinct values, ignoring missings
vartotal sum of non-missing values
poszvar TRUE if more than 1 distinct value in the centered values or any missing
simMean See section 3.5
nonMissingCount count of non missing values
name name of object
Subtract the mean from the values

3.1.2 Changing covariate

if less than 3 waves **then**
    Stop: changing covariate inappropriate (to reduce confusion among users!)
Check the nodeset (section 3.3)
if less than (number of waves - 1) columns **then**
    Stop: not enough values
if more than (number of waves - 1) columns **then**
    remove the excess, carefully preserving the attributes apart from the dimensions.
Create attributes: (a class method)
mean ignore missings
meanp mean for each wave, ignore missings
range Extent of range, ignore missings. Make sure a double.
rangep Extent of range for each wave, ignore missings. Make sure is a double.
range2 Ends of range, ignore missings
moreThan2 TRUE if more than 2 distinct values, ignoring missings
vartotal sum of non-missing values
poszvar TRUE if more than 1 distinct value in the centered values or any missing
simMean See section 3.5
nonMissingCount count of non missing values

name name of object
Subtract the mean from the values

3.1.3 Constant dyadic covariate

Check the nodesets (section 3.3)
if attribute type is oneMode **then**
    set diagonal to *missing* so is ignored in mean and range
Create attributes: (a class method)
mean ignore missings
range Extent of range, ignore missings. Make sure a double.
range2 Ends of range, ignore missings
name name of object
if attribute type is oneMode **then**
    set diagonal to *zero*

3.1.4 Changing dyadic covariate

if less than 3 waves **then**
    Stop: changing covariate inappropriate (to reduce confusion among users!)
Check the nodesets (section 3.3)
if less than (number of waves - 1) columns **then**
    Stop: not enough values
if more than (number of waves - 1) columns **then**
    remove the excess, carefully preserving the attributes apart from the dimensions.
if attribute type is oneMode **then**
    set all diagonals to *missing* so are ignored in mean and range
Create attributes: (a class method)
mean ignore missings
range Extent of range, ignore missings. Make sure a double.
name name of object
if attribute type is oneMode **then**
    set all diagonals to *zero*

3.2 Composition change objects

Check there are no duplicates in the nodesets: only one change object per nodeset is allowed.
for all composition change objects **do**
    Check the nodeset (section 3.3)

Check that the ends of each interval in each object are not less than 1 or greater than the number of waves and that each line has an even number of digits.
Generate a data frame of events(section 3.2.1), a matrix of activeStart flags(section 3.2.2) and a matrix of actions(section 3.2.3)
Add these to the object as attributes

3.2.1 Events

Data frame with columns:
event "join" or "leave" (a factor)
period
actor
time between 0 and 1

3.2.2 ActiveStart Flags

activeStart matrix has a row per actor and a column per period
TRUE if the actor is active at the start of the period, otherwise FALSE

3.2.3 Action

Action matrix is same shape as Active flag matrix, with entries
0 Active at start
1 Inactive at start, never previously active
2 Inactive at start, previously active but never active again
3 Inactive at start, previously active and active again

3.3 Check NodeSet

if Nodeset name in the list and lengths match **then**
    Valid
else
    Invalid

3.4 Dependent variables

NB The attributes list tends to change rather quickly and some items may no longer be required. Some should be used in the C, but are not...

Validate the nodeset(s)
Create an attribute *name* with name of the object
if behavior variable **then**

Create attributes:
distance sum of absolute differences by period, ignoring missings
vals table of values by period, NA included as a value
nval vector of non-missing counts by period
noMissing vector of missing counts be period
range overall range
range2 overall min and max
moreThan2 TRUE if number of distinct values more than 2 (includes missings as a value) ??? inconsistent with covariates?
poszvar TRUE if more than one distinct value or any missing values.
modes vector of modes of rounded values per period. Might give multiple results?
missing TRUE if any missing
simMean value of similarity mean (see section 3.5)
structural FALSE Not allowed!
balmean NA
structMean NA
uponly TRUE if all changes increase, ignoring missings
downonly TRUE if all changes decrease, ignoring missings

else {bipartite or onemode}

create attributes:
distance count of changes, ignoring missing and structural values and diagonals if not bipartite.
uponly TRUE if ties are only ever created, never lost
downonly TRUE if ties are only ever lost, never created

if one-mode **then**

Create attributes:
balMean see section(3.6)
structMean see section(3.7)
symmetric TRUE if all waves are symmetric
missing TRUE if any missing values (except on diagonal)
structural TRUE if any 10 or 11
vals table of counts of values by period
nval Counts of non-missing values by period, excluding diagonal
range2 Min and max of non-structural values
noMissing Number of missing values by period
noMissingEither Number of missing values at start or finish of period (excludes final).
nonMissingEither Number of non missing values at start or

finish of period (excludes final).
simMean NA
ones Count of values equal to 1 by period
density Density of network by period
degree Average degree by period
averageOutDegree overall average degree
averageInDegree overall average degree
maxObsOutDegree Maximum observed outdegree by period
missings count of missings by period
else if bipartite **then**
Create attributes:
balMean NA
structMean NA
symmetric FALSE
missing TRUE if any missing values
structural TRUE if any 10 or 11
vals table of counts of values by period
nval Counts of non-missing values by period
range2 Min and max of non-structural values
noMissing Number of missing values by period
noMissingEither Number of missing values at start or finish of period (excludes final).
nonMissingEither Number of non missing values at start or finish of period (excludes final).
simMean NA
ones Count of values equal to 1 by period
density Density of network by period
degree Average degree by period
averageOutDegree overall average degree
averageInDegree overall average degree
missings count of missings by period

3.5 Similarity mean

for all columns of matrix **do** {waves}
for all entries in column **do** {actors}
in a copy of the column set this entry to NA
Calculate 1 - abs(this entry - copy column)/ range
Sum this over the nonmissing entries in this vector
Count the nonmissing entries in this vector
Sum nonmissing entries and counts over the columns

Calculate the similarity mean as this total sum divided by total count.
For possible later use, also return the total sum and total count.

3.6 Balance mean

In calculations, remove diagonal and replace structural values by the values represented.
Numerator = sum over all columns of
2 * count of non-zero entries * count of non-missing non-nonzero entries
Denominator = sum over all columns of
count of non-missing entries times one less than this.
Mean is numerator divided by denominator.

3.7 Structural mean

In calculations, remove diagonal and replace structural values by the values represented.
Numerator = sum over all rows of
2 * count of non-zero entries * count of non-missing non-nonzero entries
Denominator = sum over all rows of
count of non-missing entries times one less than this.
Mean is numerator divided by denominator.

3.8 Constraints between networks

Make a two column matrix containing the names of all possible pairs of dependent variables, including pairs with themselves.
Identify any dependent variables that can relate: same type and have the same node sets.
Create a list of these possibly relating dependent variables for each of access later.
for all row in the matrix of pairs where the two columns are not the same **do**

> **if** nodeset(s) match and type matches and not behavior and either both networks are symmetric or neither is **then**
>
> > In checks, replaces structurals by 0/1 first, and ignore missings
> > Check *higher*: first network always greater than or equal to second network.
> > Check *disjoint*: sum of product of two networks not greater than 0.
> > Check *atLeastOne*: sum of two networks never equal 0.

3.9 Similarity means at distance 2

for all constant covariates, varying covariates **do**
    for all dependent networks which have the node set of the constant covariate as their receivers (not behavior variables) **do**
        Calculate the corresponding similarity mean: see section 3.9.1).
for all behavior variables **do**
    for all dependent networks which have the node set of the constant covariate as their receivers (not behavior variables) **do**
        Substract the mean from the behavior variable
        Find the range of the behavior variable
        Calculate the corresponding similarity mean using the centered behavior variable values (omitting final column) and the calculated range: see section 3.9.1). (i.e. centering and range are done on complete variable)

3.9.1 Alter similarity calculation

for all observations except the last **do**
    Replace structurals by 0/1
    for all rows of network matrix **do**
        if sum of nonmissing entries is 0 **then**
            Set vi for row to 0
        else if all covariate values corresponding to non zero network entries are missing **then**
            Set vi for row to NA
        else
            Set vi for row to sum of covariate times network row divided by the sum of the network row, ignoring missings in both cases.
    Call rangeAndSimilarity using vi and the range if passed in (behavior variables) to obtain the values *simTotal* and *simCnt* for this overvation.
    Accumulate these two values
Divide sum of *simTotal* by sum of *simCnt* over observations (excluding the final one).

4. getEffects

This function generates an effects data object corresponding to a Siena Data object or a Siena Group Object.

In general, effects are driven by selecting rows in the `allEffects` data frame for some effect group and then substituting variable names into the spaces marked by `xxxxxx` and similar.

For a group object, the effects are created using the first data object plus the total number of observations. Attributes are first copied from the group level to the first data object. The only changes required are to fill in the starting values for the rate effects for the later objects and to adjust the starting values for density, reciprocity, linear effects.

The function `networkRateEffects` creates the required number of rate effects for networks. `createEffects` extracts the rows from the effects data frame for an effect group and calls the function `substituteNames` to replace the variable fields by the current variable name. It now creates the complete effect rows including endowment effect copies.

4.1 Siena Data Object

for all dependent variables **do**
    Create appropriate effects

4.2 OneMode Network Effects

Call `networkRateEffects` to get the rate effects
Use `symmetricObjective` or `nonSymmetricObjective` effect groups to create the basic objective function effects.
for all dyadic covariates with first node set matching **do**
    Use `dyadObjective` effect group to add appropriate objective function effects
for all constant covariates, behavior variables, changing covariates with the same node set **do**
    Call function `covarOneModeEff` to add the appropriate effects.
    Note *poszvar* is always TRUE for behavior variables.
if any covariates or behavior variables **then**
    Add *nintn* rows for user specified interaction effects
for all distinct dependent network variables with the same node set **do**
    if oneMode **then**
        Use `nonSymmetricSymmetricObjective` or `nonSymmetricNonSymmetricObjective` effect groups to add appropriate effects
        Use `tripleNetworkObjective` effect group to add appropriate effects for pairs of other dependent networks ('other' meaning that they have the role of explanatory variables) that either are

both one-mode, or both are bipartite with the same second node sets

else if bipartite which matches on nodeset 1 **then**

Use `nonSymmetricBipartiteObjective` effect group to add appropriate effects

for all actor covariates or behavior variables with the same node set **do**

Use `covarNetNetObjective` effect group to add appropriate effects.

if more than one network **then**

paste the network name at the front of all the objective function effects

Alter the text for endowment effects to start "Lost ties:"

Calculate the starting values for the default effects (see section 4.14)

Select the default rate effects by setting *include* to TRUE for the `basic rate` effects.

Add the starting value for the rate to the *initialValue* column of the basic rate effects

if symmetric **then**

Set *include* to TRUE for the degree (density) evaluation effect and transitive triads evaluation effect.

else

if no period is uponly or downonly **then**

Set *include* to TRUE for the degree (density) evaluation effect

Add the starting values for the degree(density) evaluation effect calculated by `getNetworkStartingVals` to the *initialValue*.

else

Remove both degree (density) effects from the data frame.

Set *include* to TRUE for the reciprocity evaluation effect.

4.3 Behavior Variable Effects

Use `behaviorRate` effect group to get the rate effects. Either remove the second one or duplicate the second and remove the first to match the number of observations.

Use `behaviorObjective` effect group to create the basic objective function effects.

for all other dependent variables which match on first node set **do**

Use `behaviorOneModeObjective` or `behaviorBipartiteObjective` to add the objective function effects with respect to this network.

Use `behaviorOneModeRate` or `behaviorBipartiteRate` to add the

rate effects with respect to this network.

for all constant covariates, other behavior variables or changing covariates **do**

Call `covBehEff` and `covBBehEff`to add the interaction effects

for all networks with same node set (first for bipartite) **do**

Use `behaviorOneModeObjective2` or `behaviorBipartiteObjective2` effect group to create a second set of objective function effects.

Add *behNintn* unspecified behavior interaction effects

Create the effects data frame by calling `createObjEffectList` and `createRateEffectList`. This creates e.g. the evaluation and endowment effect copies.

Select the default effects by setting *include* to TRUE for `basic rate` and `linear shape` (if not any period uponly or downonly) and `quadratic shape` (if the range of the variable is greater than or equal to 2) evaluation effects.

if any period uponly or downonly **then**

remove the linear effects (evaluation and endowment) from the data frame.

Add the starting values for the default effects calculated by `getBehaviorStartingVals` (see section 4.13) to the *initialValue* column of the data frame.

Alter the text for endowment effects to start with "dec. beh."

4.4 Bipartite Network Effects

Call `networkRateEffects` to get the rate effects

Use `bipartiteObjective` effect group to create the objective function effects.

for all dyadic covariates with both node sets matching **do**

Use `dyadObjective` effect group to create the appropriate effects

for all constant covariates, behavior variables, changing covariates **do**

Call function `covarBipartiteEff` to add the appropriate effects poszvar is always TRUE for behavior variables.

if any covariates or behavior variables **then**

Add *nintn* rows for user specified interaction effects

for all distinct dependent network variables with the same node set **do**

if oneMode **then**

Use `bipartiteSymmetricObjective` or `bipartiteNonSymmetricObjective` effect groups to add the appropriate the effects

else if bipartite and matches first node set) **then**
Use `bipartiteBipartiteObjective` effect group to add the appropriate effects
NB no `covarNetNetObjective` here?
if more than one network **then**
paste the network name at the front of all the objective function effects
Create the effects data frame by calling `createObjEffectList` and `createRateEffectList`. This creates e.g. the evaluation and endowment effect copies.
Alter the text for endowment effects to start "Lost ties:"
Calculate the starting values for the default effects (see section 4.15)
Select the default rate effects by setting *include* to TRUE for the `basic rate` effects.
Add the starting value for the rate to the *initialValue* column of the basic rate effects
if no period is uponly or downonly **then**
Set *include* to TRUE for the degree (density) evaluation effect
Add the starting values for the degree(density) evaluation effect calculated by `getBipartiteStartingVals` to the *initialValue*.
else
Remove both degree (density) effects from the data frame.

4.5 covarOneModeEff

Use `covarSymmetricObjective` or `covarNonSymmetricObjective` effect group to create the objective function effects
Use `covarSymmetricRate` or `covarNonSymmetricRate` to create the rate effects
if not poszvar **then**
Reduce the new objective function effects to just "altX" and "altSqX" (symmetric) or "egoX" (non symmetric)
if not morethan2 **then**
Remove the "altSqX" effect.

4.6 covarBipartiteEff

if first node set matches **then**
Use `covarBipartiteRate` effect group to create the rate effects
Use `covarBipartiteObjective` effect group to create the objective function effects
if first node set matches **then**

reduce the objective function effects to "egoX", "altDist2", and "totDist2"

else if poszvar **then**

reduce the new objective function effects to the rows "altX" and "altSqX"

if not morethan2 **then**

remove the "altSqX" effect

else

no objective function effects

4.7 covBehEff

Use `covarBehaviorObjective` effect group to create a potential set of objective function effects

if covariate and behavior variable are different **then**

Create objective function effects as the first row of potential set

for all oneMode dependent variables with the same node set **do**

Add an objective function effect from the second row of the potential set.

if any objective function effects **then**

Set *shortName* to `effFrom`

Use `covarBehaviorRate` effect group to create the rate effects

Use `covarABehaviorBipartiteObjective` effect group to create objective function effects for bipartite dependent networks combined with covariates on the first node set

4.8 covBBehEff

Use `covarBBehaviorBipartiteObjective` effect group to create objective function effects for bipartite dependent networks combined with covariates on the second node set

4.9 covarNetNetEff

if *poszvar* **then**

Use `covarNetNetObjective` effect group to create additional objective function effects if the second network is one-mode;

Use `covarABNetNetObjective` effect group to create additional objective function effects if the second network is one-mode or two-mode;

Use `covarANetNetObjective` effect group to create additional objective function effects if the second network is one-mode or

{two-mode while the covariate is defined for the first mode};
Use `covarBNetNetObjective` effect group to create additional objective function effects if the second network is one-mode or {two-mode while the covariate is defined for the second mode}.

4.10 CreateRateEffectList

Add the *name* column by duplicating the dependent variable name, and *effectFn* and `statisticFn` as empty lists.

4.11 CreateObjectEffectList

Add the *name* column by duplicating the dependent variable name, and *effectFn* and `statisticFn` as empty lists.
Add an endowment effect row if required for each objective function effect.

4.12 SienaGroupObject

First create the effects for the first data object, but inserting the correct number of basic rate effects for the whole group.
for all other data objects **do**
for all dependent variables **do**
Create the starting values for this dependent variable
Insert the rate starting values in the *initialValue* field of the correct effects
Combine the starting values to create an overall one for the objective function effects:
if behavior **then**
Add new $d_i, i = 1, \ldots, n-1$ to make one long vector of difference vectors between n observations
if rounded range of variable (max-min) is 2 (what happens with range 1) **then**
Add to $n_{min+} = \sum_i n_{i,min+}$ and the others
Tendency is

$$\log(\frac{(n_{min+}+2)*(n_{max+}+n_{max0}+4)))}{(n_{max-}+2)*(n_{min+}+n_{min0}+4)))}$$

if abs(tendency) > 2 **then**
trim to ± 2
else {range less than 2 or greater than 2}

Let $\bar{d} = \text{mean}(d_i)$, ignoring missing values
Let $\sigma_d^2 = \text{var}(d_i)$, ignoring missing values
if $\bar{d} < 0.9 * \sigma_d^2$ **then**
tendency $= 0.5 * \log((\bar{d} + \sigma_d^2)/(\bar{d} - \sigma_d^2))$
else
tendency $= \bar{d}/(\sigma_d^2 + 1)$
if abs(tendency) greater than 3) **then**
Trim to ± 3
else if onemode **then**

else {bipartite}

4.13 Behavior Starting Values

Calculate $d_i, i = 1, \ldots, n-1$ difference vectors between n observations
if rounded range of variable (max-min) is 2 (what happens with range 1) **then**
for all intervals i **do**
Let $n_{i,min+}$ = number who start at minimum and go up
Let $n_{i,min0}$ = number who start at minimum and stay there
Let $n_{i,max-}$ = number who start at maximum and go down
Let $n_{i,max0}$ = number who start at maximum and stay there
Calculate

$$v = \frac{n_{i,min+} + 1}{n_{i,min+} + n_{i,min0} + 2} + \frac{n_{max-} + 1}{n_{i,max0} + n_{i,max-} + 2}$$

if $v > 0.9$ **then**
$v = 0.5$
Starting rate is $-\log(1 - v)$
Let $n_{min+} = \sum_i n_{i,min+}$ total number who start at minimum and go up
Let $n_{min0} = \sum_i n_{i,min0}$ total number who start at minimum and stay there
Let $n_{max-} = \sum_i n_{i,max-}$ total number who start at maximum and go down
Let $n_{max0} = \sum_i n_{i,max0}$ total number who start at maximum and stay there
Tendency is

$$\log(\frac{(n_{min+} + 2) * (n_{max+} + n_{max0} + 4)))}{(n_{max-} + 2) * (n_{min+} + n_{min0} + 4)))}$$

if abs(tendency) > 2 **then**
trim to ± 2
else {range less than 2 or greater than 2}
for all intervals i **do**
starting rate is $\max(\text{var}(d_i), 0.1 * \sum_i \text{abs}(d_i)/\text{nactors}$
Let $\bar{d} = \text{mean}(d_i)$, ignoring missing values
Let $\sigma_d^2 = \text{var}(d_i)$, ignoring missing values
if $\bar{d} < 0.9 * \sigma_d^2$ **then**
tendency $= 0.5 * \log((\bar{d} + \sigma_d^2)/(\bar{d} - \sigma_d^2))$
else
tendency $= \bar{d}/(\sigma_d^2 + 1)$
if abs(tendency) greater than 3) **then**
Trim to ± 3

4.14 One mode network Starting Values

Temporarily subtract 10 from structural values
Let dif_i be the number of differences between start and end of interval i, ignoring missings
Let $n_{ijk}, j, k = 0, 1$ be counts of cells with value j at start and k at end of interval i, ignoring missings
Let n_i be the number of cells which are not missing at both start and end of interval i.
Let d_i be the sum of absolute differences by period, ignoring missings (already calculated and stored in the attribute *distance*).
Let λ_i be the starting value of basic rate parameter for interval i
if symmetric **then**

$$\lambda_i = \text{nactors} * (0.2 + d_i)/(n_i\%/\%2 + 1)$$

else

$$\lambda_i = \text{nactors} * (0.2 + 2 * d_i)/(n_i + 1)$$

Trim λ_i to be between 0.1 and 100.
if symmetric **then**
Divide n_{ijk} by 2

starting value for degree parameter:

$$\text{Define} \quad p_{i01} = \begin{cases} n_{i01}/(n_{i01}+n_{i00}) & n_{i01}+n_{i00} >= 1 \\ 0.5 & \text{otherwise} \end{cases}$$

$$p_{i10} = \begin{cases} n_{i10}/(n_{i10}+n_{i11}) & n_{i10}+n_{i11} >= 1 \\ 0.5 & \text{otherwise} \end{cases}$$

$$p_{i00} = \begin{cases} n_{i00}/(n_{i01}+n_{i00}) & n_{i01}+n_{i00} >= 1 \\ 0.5 & \text{otherwise} \end{cases}$$

$$p_{i11} = \begin{cases} n_{i11}/(n_{i10}+n_{i11}) & n_{i10}+n_{i11} >= 1 \\ 0.5 & \text{otherwise} \end{cases}$$

Trim p_{ijk} to lie between 0.02 and 0.98
Calculate p_i

$$p_i = \begin{cases} 4/(p_{i00}/n_{i01} + p_{i11}/n_{i10}) & n_{i10} * n_{i01} >= 1 \\ 1e-6 & \text{otherwise} \end{cases}$$

Starting value for degree parameter is

$$\sum_i 0.5 * \log(p_{i01}/p_{i10}) * p_i / \sum_i p_i$$

4.15 Bipartite Starting Values

Temporarily subtract 10 from structural values
Let dif_i be the number of differences between start and end of interval i, ignoring missings
Let $n_{ijk}, j, k = 0, 1$ be counts of cells with value j at start and k at end of interval i, ignoring missings
Let n_i be the number of cells which are not missing at both start and end of interval i. (Diagonal included here.)
Let d_i be the sum of absolute differences by period, ignoring missings (already calculated and stored in the attribute *distance*).
Let λ_i be the starting value of basic rate parameter for interval i

$$\lambda_i = \text{nsenders} * (0.2 + 2 * d_i)/(n_i + 1)$$

Trim λ_i to be between 0.1 and 100.

starting value for degree parameter:

$$\text{Define} \quad p_{i01} = \begin{cases} n_{i01}/(n_{i01}+n_{i00}) & n_{i01}+n_{i00} >= 1 \\ 0.5 & \text{otherwise} \end{cases}$$

$$p_{i10} = \begin{cases} n_{i10}/(n_{i10}+n_{i11}) & n_{i10}+n_{i11} >= 1 \\ 0.5 & \text{otherwise} \end{cases}$$

$$p_{i00} = \begin{cases} n_{i00}/(n_{i01}+n_{i00}) & n_{i01}+n_{i00} >= 1 \\ 0.5 & \text{otherwise} \end{cases}$$

$$p_{i11} = \begin{cases} n_{i11}/(n_{i10}+n_{i11}) & n_{i10}+n_{i11} >= 1 \\ 0.5 & \text{otherwise} \end{cases}$$

Trim p_{ijk} to lie between 0.02 and 0.98
Calculate p_i

$$p_i = \begin{cases} 4/(p_{i00}/n_{i01} + p_{i11}/n_{i10}) & n_{i10} * n_{i01} >= 1 \\ 1e-6 & \text{otherwise} \end{cases}$$

Starting value for degree parameter is

$$\sum_i 0.5 * \log(p_{i01}/p_{i10}) * p_i / \sum_i p_i$$

5. sienaGroupCreate

This function combines a list of siena data objects for common processing. It is also used in `initializeFRAN` to convert a single data object to a group one so that all later processing can have a common argument.

Some validation is performed to check that all the data objects match in terms of the dependent variables (name, type, nodesets) and covariates (names and nodesets).

If there is more than one data object, the constant covariates and constant dyadic covariates must be changed into changing ones. New covariates are created and the old ones removed from the lists. The attributes are copied over rather than recalculated (although they are mostly changed later).

Overall values are calculated for the balance mean, and network ranges. For behavior variables and covariates overall ranges, means and similarity mean values are calculated.

The following overall values are copied down to the individual objects:

dependent variables symmetric, missing, structural, poszvar, range, moreThan2 (some only if behavior)

changing covariates range, poszvar, moreThan2

changing dyadic covariates range, range2

5.1 Group attributes

The group object has various attributes, copied from or combinations of the attributes on the individual objects.

netnames the names of the dependent variables (these must be the same in each data object)

symmetric logical vector indicating whether the corresponding independent variable is symmetric (set to FALSE for behavior variables and bipartite networks.

structural logical vector indicating presence or absence of any structural values

numberNonMissingNetwork vector of count of non missing values for non behavior variables

numberMissingNetwork vector of count of missing values for non behavior variables

numberNonMissingBehavior vector of count of non missing values for behavior variables

numberMissingBehavior vector of count of missing values for behavior variables

allUpOnly logical vector indicating that the values for this independent variable never decrease over time

allDownOnly logical vector indicating that the values for this independent variable never increase over time

anyUpOnly logical vector indicating that for one or more of the intervals the values for this independent variable do not decrease

anyDownOnly logical vector indicating that for one or more of the intervals the values for this independent variable do not increase.

allHigher Table of logicals indicating whether *higher* attribute is true for each pair of networks in every data object

allDisjoint Table of logicals indicating whether *disjoint* attribute is true for each pair of networks in every data object

allAtLeastOne Table of logicals indicating whether *atLeastOne* attribute is true for each pair of networks in every data object

anyHigher Table of logicals indicating whether *higher* attribute is true for each pair of networks in any data object

anyDisjoint Table of logicals indicating whether *disjoint* attribute is true for each pair of networks in any data object

anyAtLeastOne Table of logicals indicating whether *atLeastOne* attribute is true for each pair of networks in any data object

types types of the independent variables

observations A single integer with the total number of periods to process.

periodNos A list of the period numbers (misses out the final one for each data object)

groupPeriods Vector of the number of total number of periods for each data object.

netnodesets A list containing the nodeset(s) for each dependent variable

cCovars Vector of names of constant covariates

vCovars Vector of names of changing covariates

dycCovars Vector of names of constant dyadic covariates

dyvCovars Vector of names of changing dyadic covariates

ccnodesets A vector containing the nodeset(s) for each constant covariate

cvnodesets A vector containing the nodeset(s) for each changing covariate

dycnodesets A list containing the nodeset(s) for each constant dyadic covariate

dyvnodesets A list containing the nodeset(s) for each changing dyadic covariate

compositionChange Logical vector indicating the presence of composition change data for any of the data objects

exooptions Named vector of composition change file options for the named node sets. Read from the first data object: assumed all the same

names Vector of names of the data objects

class ("sienaGroup", "siena")

balmean Vector of overall balance means, one for each dependent variable, NA for behavior variables and bipartites.

structmean Vector of overall structural means, one for each dependent variable, NA for behavior variables and bipartites.

averageOutDegree Vector of overall average outdegrees. NA for behavior variables.

averageInDegree Vector of overall average indegrees. NA for behavior variables and bipartites.

bRange Vector of overall ranges for behavior variables: entries corresponding to networks are NA

behRange Matrix with two rows, and column for each independent variable. Set to the overall min and max for behavior variables, NA for others

bSim Overall similarity mean for behavior variables, NA for networks

bPoszvar logical vector, NA for networks. For behavior variables TRUE if more than 1 distinct value in the overall values or any missing (always?)

bMorethan2 logical vector. NA for networks. For behavior variables TRUE if more than 2 distinct values in overall variable, ignoring missings

cCovarPoszvar logical vector for constant covariates indicating presence if more than one distinct value or any missing (overall). NB only exist if there is only one data object.

cCovarMoreThan2 logical vector for constant covariates indicating presence of more than 2 distinct values (missing is counted as a value)

cCovarRange vector of ranges for constant covariates

cCovarRange2 matrix of min and max for constant covariates

cCovarSim vector of overall similarity means for constant covariates

cCovarMean vector of means for constant covariates

vCovarPoszvar logical vector for changing covariates indicating presence if more than one distinct value or any missing (overall).

vCovarMoreThan2 logical vector for changing covariates indicating presence of more than 2 distinct values (missing is counted as a value)

vCovarRange vector of ranges for changing covariates

vCovarSim vector of overall similarity means for changing covariates

vCovarMean vector of means for changing covariates

dycCovarRange vector of ranges for constant dyadic covariates

dycCovarRange2 matrix of min and max for constant dyadic covariates

dycCovarMean vector of means for constant dyadic covariates

dyvCovarRange vector of ranges for changing dyadic covariates

dyvCovarRange2 matrix of min and max for constant dyadic covariates

dyvCovarMean vector of means for changing dyadic covariates

anyMissing logical vector indicating any missing values in the each dependent variable

netRanges Matrix with two rows and a column for each dependent variable. Overall min and max for networks, NA for behavior variables

6. siena07

This is a wrapper for the function *robmon* which performs the processing that used to be in polrup in Siena3. An optional tck/tk gui is provided, or progress messages are provided on the console. The choice between these is made by using `batch=FALSE` or `batch=TRUE` respectively.

Details of input and output are on the R help page. Required input is an object containing control information for the Robbins-Monro algorithm, and

any extra parameters required by the FRAN to be used. As the distinction between the two parts is not complete, flags maxlike and cconditional are on the input object, although not logically relevant to the algorithm.

There is user output written to a file (.txt), together with optional additional output to the console (suppressed unless `verbose=TRUE`), which can be redirected using the `sink()` command.

robmon attempts to duplicate the output of the Siena3 procedure polrup. It uses a special function, *Report* for all output. This function knows about four files: outf, lf, cf, bof, and can also write to the console. Currently, all files except outf are null, with any other output suppressed or written to the console. Only Report would need altering to alter this behaviour. No file connections need to be passed around as parameters.

The object returned from *siena07* is an object containing everything of interest from the run, including the estimates of the parameters and the covariance matrix. Details of the more useful parts are in the R documentation.

7. User Interrupts

These are set in callbacks from the *siena07* tcl/tk gui. When they are read, a parallel set of flags is used to store the states, so that interrupts can be processed reliably. All is done using functions, to avoid global variables, or passing variables around. There are 3 interrupts:

UserInterrupt Stop everything, but return the values so far, with (I hope) some flag to indicate we did not finish.

UserRestart Go back to the beginning of phase 1 with the current parameters

EarlyEndPhase2 Stop the estimation routine and proceed to phase 3 using the current parameters.

All six functions, if called with an argument, store the argument as the current value and if called with no argument, return the value. All values are booleans. The functions are not exported from the namespace, to avoid burdening the user with the details of their existence.

8. Robbins Monro Algorithm—*robmon*

The routine *robmon* contains the Robbins Monro (stochastic approximation) algorithm. It is the replacement for the Siena3 procedure polrup. It is not designed to be called directly by the user, so there will not be an R help page for it.

The outline of the algorithm is given in the text *Siena algorithms*, and to understand the description of the code given here it may be helpful to have read that outline.

9. *robmon*

9.1 Input (from siena07)

z Model fitting object.

x Input model object, as described in the help page for *sienaModelCreate*.

... Extra parameters for FRAN (including the data!).

It may seem surprising that the data is simply a parameter passed unchanged to FRAN. But this is the point of the separation of the simulation and estimation routines: *robmon* could be used to solve the moment equation for any data: it does not matter whether the data is a network or something else entirely, as long as a matching FRAN is used.

9.2 Output

z . More or less everything used in the processing. Details in the help page for *siena07*.

9.3 Details

9.3.1 Initialize

Copy from *x* everything that we may change during the run. Initialize number of iterations, restarted flag, force finite difference flag, etc.

9.3.2 *Initial call to FRAN*

This call is used to set up the parameters for conditional estimation, and to set up data in a call to C. The values of the statistics in the observed data (targets) are returned, along with the addresses of the data objects in C. The processed data and selected effects is written to a hidden data object within the function *FRANstore* from where it can be accessed on later calls, or passed to other processes.

9.3.3 *Initialise cluster of processes, if required*

This needs to access (but not understand!) the data object created in the call to FRAN. It sets up the processes and random number streams and passes the data object across. It then does a special call to FRAN to create the data objects in C++ for each process. Later calls to the processes only need minimal communication, done using cut-down versions of x and z.

9.3.4 *Calculate epsilon*

Used only for MoM estimation in the finite differences option. Currently 0.1, except for parameters which must be positive, where it is 0.1 times the parameter starting values.

TS: Here I would prefer min{0.1, 0.1 × starting value}.
Ruth: Easily changed, but we are using typically much bigger values than this: will it work? Are you sure you don't want max?

To be improved, to use prior information on standard error of parameter if available.

9.3.5 Main loop

Note R has no GOTO statement. I use the term *break* to indicate exit from the current loop only. Interrupts are checked after every iteration except the first few of phase 3. This documentation does not include all the details, or it would duplicate the code.

```
repeat
    repeat {this one is just to jump out of, only executed once}
        if all parameters now fixed (2 opportunities to do this in previous
        loop!) then
            set a flag to just do Phase 3.
        initialize interrupts
        announce phase 0 (set up progress bar, calculate iteration min and
        max for phase 2 subphases.)
        reset fixed flags
        if not just-phase-3 flag set then
            if need to do phase 1 then
                initialize phase 1
                run phase 1 iterations 1 to 10
                if using finite diffs then
                    check number of changes and change epsilon if
                    necessary
                if user stop or user restart or error or (using finite
                difference and need to repeat with new epsilon) then
                    break
                if using finite diffs then
                    fix PARAMETERS with 0 or 1 changes, if any exist
                run rest of phase 1 iterations
                if user stop or user restart or error then
                    break
                calculate derivative matrix
                if necessary then
                    change the length of phase 1 or force the use of finite
                    differences
                if user stop or error or user restart then
                    break
                if necessary then
                    fix some parameters.
            Initialise phase 2
            run phase 2 subphase 1
            if error or user stop or user restart then
                break
```

run phase 2 subphase 2
if error or user stop or user restart or we have restarted because of epsilon change in phase 1 and not restarted from here before! **then**
break
run rest of phase 2 subphases
if error or user stop or user restart **then**
break
run phase3
if not user restart **then**
break
until for ever
if do not need to restart because of epsilon or user restart **then**
break
until for ever

9.3.6 Final processing

- Do a final call to FRAN. In conditional estimation, the rescaling of basic rate parameters will be done here.
- reset the covariance matrix to 33, 999 as in phase3.

10. Phase 1

10.1 Input

As for *robmon*.

10.2 Output

As for *robmon*.

10.3 Details

10.3.1 Initialise

- Reset *SomeFixed* flag (have we fixed any parameters in this run).
- Announce phase 1
- Create arrays to store simulated statistics, scores and contributions to the derivative matrix from either the finite difference or maximum

likelihood routines. These arrays are currently redefined in Phase 3, so lost.

10.3.2 Timing

Timings are calculated between the start of the 2nd iteration and the start of the 6th iteration. This is just for determining the frequency of writing information to the gui. Write frequency is set to a prettified version of 20/time for 5 iterations, or 5 if elapsed time is very small. For batch mode this is multiplied by 10, which seems unnecessary in phase 1. If using multiple clusters the total number of iterations are adjusted to be a multiple of the number of processes, and the 6th is replaced by the first one greater than or equal to 6 in the iteration sequence advancing in steps of the number of processes. I think this is wrong: it should be from 2 to 6 steps... but not very important! (I have made some adjustments for those users who have more than 9 processors: we do at least 10 simulations in the first part, and ignore timing if we do them in too few steps. If there are enough steps in the second part the timing is done there.)

10.3.3 An iteration

- call FRAN. If not OK, return
- store simulated statistics, scores if present, derivatives if present, simulated networks (part!) if present.
- if required, call finite differences routine and store result
- check for user interrupts and return if requested
- Report progress via progress bar or to console

10.3.4 Check epsilons (only for finite differences option)

- If derivatives are being calculated by finite differences, check after 10 iterations that enough different values of the statistics have occurred. Ideally 5 or more. (The check may not be done immediately after 10 iterations if we have 4 processors, say, but I do only look at 10 of the results.)
- If there are less than 3, epsilon is multiplied by 3 for parameters which must be positive, and by 10 otherwise.
- If there are 3 or 4, the multipliers are 2 and $\sqrt{10}$.

- If any new values are less than 0.1 times the scale factor, or more than 100 times it, replace by the bound.
- If any are less than 5, we will restart with the new epsilons, unless we have already done so 4 times.
- If we have already restarted 4 times then we continue, after fixing the last parameter with only 0 or 1 changes, if there are any such.

10.3.5 End of phase processing

- Calculate derivative estimate.
  - For finite differences or maximum likelihood: the mean of the arrays returned at each iteration.
  - For the score based method, we need a little notation:
    Let $\mathbf{f}_i$ be the simulated deviations from the targets in iteration i, and $\mathbf{s}_i$ the score function in iteration i, N be the number of iterations. Then the estimated derivative matrix d_{ij} is given by:

$$D = \sum_i \text{outer}(\mathbf{f}_i, \mathbf{s}_i)/N - \text{outer}(\bar{\mathbf{f}}, \bar{\mathbf{s}})/N^2$$

  - For the score based method, if any of the diagonal values is non-positive, we do not continue. First we double the number of iterations in phase 1 and start again. Once this number exceeds 200, we stop increasing it and force the use of Finite differences. Then we don't come through this check!
  - For either method, if still processing, set the rows and columns related to fixed parameters to 0's with 1 on the diagonal.
  - If any diagonal values for non-fixed parameters are not positive, make them fixed, and set *newfixed* flags to record which ones have been fixed.
  - Calculate the standard deviations of the deviations.
- Invert the derivative matrix:
  - Set the rows and columns related to fixed parameters to 0's with 1 on the diagonal.
  - Replace any diagonal values less than 1e-8 by 1e-3.
  - Do the inversion

  - If it fails, add 1 to the diagonal and try again

- Quasi-Newton step
  - If inversion of matrix was successful, set fchange to 0.5 times the gain parameter times the matrix product of inverse with the mean deviations from targets, otherwise to zero.
  - Zero the change for any fixed parameters.
  - Check the jump is not too large: if the maximum absolute value of the change divided by the corresponding input gain parameter is greater than 10, divide the changes by this value and multiply them by 10. This caps the maximum ratio to the scale at 10.
  - Check that positive parameters will stay positive. If not, replace the change for that parameter by half the current value of the parameter.
  - If the requested number of subphases in phase 2 is greater than 0, make the change by subtracting the changes from the current value of the parameters.

11. Phase2

11.1 Input

as for *robmon*

11.2 Output

as for *robmon*

11.3 Details

11.3.1 Initialize Phase

Turn off calculation of derivatives. Multiply the gain parameter by 2.

11.3.2 Process subphase

- Initialize
  - Announce subphase

  - Extract max and min number of iterations from values stored from start. (They are calculated at the start to find the length of the progress bar, and it seemed better not to repeat it here!).
  - Divide the gain parameter by 2
  - Initialize the sum (which will become the mean) of thetas with the current value.
  - Create arrays to store products of successive thetas.

- Perform iterations
  - Timing is calculated over the ten iterations 2 to 11. Write frequency is set to 20/(time for these 10 iterations) or 20 if time is too small. It is then prettified. Reporting using progress bar or console is done for each of the first 10 iterations and then at write frequency.
  - Call FRAN and store the deviations returned.
  - The update:
    * if only the diagonal of the derivative matrix is to be used in the update step (flag *x$diag*), calculate maxrat, the maximum ratio of the absolute deviations to their standard deviation (as estimated in Phase 1). If this is greater than the parameter maxmaxrat, set maxrat to maxmaxrat/ maxrat, and record that the values were truncated.
    * if *x$diag*, update is current gain * current deviations * maxrat / diagonal of derivative
    * otherwise, update is current gain * matrix product of current deviations with inverse of derivative matrix.
    * For parameters which must be positive and would not remain so, replace change by 0.5 times current value of parameter,
    * Zero the change for any fixed parameters.
    * update theta by subtracting the change.
    * Add new value of theta to sum of thetas.
  - After each pair of iterations, add the product of the two deviations and the square of the most recent to accumulators, and calculate the ratio of the former to the latter, ac.
  - Check for user interrupts

- Stop the subphase when either the minimum number of iterations has been reached, and the maximum of ac is less then 1e-10 or the maximum number of iterations has been reached; or (the next condition is a rare occurrence but helps a lot when it occurs) at least 50 iterations have been done and the minimum ac is < -0.8 and we have not done the maximum number of subphase repeats already (or user interrupt or error)
- Repeat the subphase if we stopped because of the minimum value of ac, unless we have already done the maximum number (set to 4).

- End of subphase. Replace parameters by the average in the subphase. Report details.

12. Phase3

12.1 Input

as for *robmon*

12.2 Output

as for *robmon*

12.3 Details

12.3.1 Initialize

- Initialize arrays for deviations, scores, derivatives.
- Divide write frequency by the number of parameters for finite differences, 2 for score derivatives. Leave unchanged for maximum likelihood. Then re-prettify.
- Announce Phase

12.3.2 Iterations

- Update progress via console or progress bar for each of first 5 iterations, then at the 10th and then at the write frequency.
- Call FRAN and store the deviations, scores, derivative contributions, and simulation values returned.

- If using finite differences (we revert to the method requested by the user here, even if we altered it in phase 1 because the score method did not work), call FiniteDifferences routine and store the resulting differences.
- After the 10th iteration check for user interrupts each time.

12.3.3 End of Phase 3

- Calculate derivative matrix dfra. Formulae as in Phase 1.
- Create a flag diver for each parameter, set to true if all the parameter values are fixed and the absolute value of the corresponding diagonal of the derivative matrix is less than 1e-6. .
- Calculate the covariance matrix of the simulated deviations, and the autocorrelations between them. (Done in the CalculateDerivative3 routine).
- Report...
- Calculate t-values for deviations from targets, as mean deviation divided by sqrt of diagonal entry of covariance matrix. Use 0 for small values of deviations and 999 for small values of variance.
- Report t-values, and comments on their values.
- For maximum likelihood, report autocorrelations
- Calculate cov, the covariance matrix of the estimates:
  - For maximum likelihood, inverse of [dfra] - variance matrix of deviations (here=scores!). (0 the rows and columns coresponding to fixed parameters, and put 1 on the diagonal, first.)
  - For others, matrix product of dinv, the adjusted covariance matrix of the deviations and the transpose of dinv.
  - Try to invert cov. Report...
  - Update the flag diver to mean: Not all parameters are fixed and this one is fixed or diver was true before (section 12.3.3) or the corresponding entry in the diagonal of the covariance matrix of the estimates is less than 1e-9.
  - set entries in cov to 33* sqrt(diagonal) off diagonal and 999 on diagonal for parameters with diver true.
- Do ScoreTests if any have been requested.

13. FiniteDifferences

13.1 Input

as for *robmon* plus *fra*, the simulated value of the targets.

13.2 Output

as for *robmon*

13.3 Details

For each parameter,

- Call FRAN with epsilon added to this parameter only
- Calculate the deviations from the simulated statistics with no epsilon.
- In first 10 iterations of phase 1: Record if difference is greater than 1e-06
- Store and return these deviations divided by epsilon.

.

14. Score Tests

14.1 Control

- Do general test: call EvaluateTestStatistic with the complete arrays dfra, msf, the covariance matrix of the deviations, and fra the mean deviations.
- The values returned are a chi-squared test and, if the degrees of freedom are 1, a one-sided test.
- If only one test was requested, the two values returned correspond to the results required.
- If more than one test was requested, call EvaluateTestStatistic with data from which all but one of the parameters for which tests are required have been removed. Repeat for each parameter for which tests were requested.

14.2 EvaluateTestStatistic

- Partition dfra into four: R code is clear enough, I hope (drop=FALSE just retains the matrix class even if one of the dimensions is 1)

```
d11 <- dfra[!test,!test,drop=FALSE]
d22 <- dfra[test,test,drop=FALSE]
d21 <- dfra[test,!test,drop=FALSE]
d12 <- t(d21)
```

- Similarly create Σ11, Σ22, Σ12 and Σ21 from msf, and z1 and z2 from fra. Then

  For maximum likelihood

$$\begin{aligned} \texttt{ov} &= -\texttt{z2} \\ \texttt{vav} &= \left(\texttt{d22} - \texttt{d21}\ \texttt{d11}^{-1}\ \texttt{d12}\right)^{-1} \end{aligned}$$

  Otherwise

$$\begin{aligned} \texttt{ov} &= \texttt{z2} - \texttt{d21}\ \texttt{d11}^{-1}\ \texttt{z1} \\ \texttt{vav} &= \left(\Sigma 22 - \texttt{d21}\ \texttt{d11}^{-1}\Sigma 12 - (\Sigma 21 - \texttt{d21}\ \texttt{d11}^{-1}\Sigma 11)\ \texttt{d11}^{-T}\ \texttt{d21}^{T}\right)^{-1} \end{aligned}$$

  then

  test statistic = $\texttt{ov}^T\ \texttt{vav}\ \texttt{ov}$

  and the one-sided one, if appropriate,

$$\texttt{ov}\sqrt{\texttt{vav}}$$

  as vav is then a scalar.