Getting Started with GLinvCI

Hao Chi Kiang <hello@hckiang.com>

March, 2022

1 Introduction

GLInvCI is a package that provides a framework for computing the maximum-likelihood estimates and asymptotic confidence intervals of a class of continuous-time Gaussian branching processes, including the Ornstein-Uhlenbeck branching process, which is commonly used in phylogenetic comparative methods. The framework is designed to be flexible enough that the user can easily specify their own parameterisation and obtain the maximum-likelihood estimates and confidence intervals of their own parameters.

The model in concern is the family of continuous-trait continuous-time Gaussian evolution processes along a known phylogeny, in which each species' traits evolve independently of each others after branching off from their common ancestor and for every non-root node. Let k be a child node of i, and z_k, z_i denotes the corresponding multivariate traits. We assume that $z_k|z_i$ is a Gaussian distribution with expected value $w_k + \Phi_k z_i$ and variance V_k, where the matrices (Φ_k, w_k, V_k) are parameters independent of z_k but can depend other parameters including t_k. The traits z_k and z_i can have different number of dimension.

2 Installation

The following command should install the latest version of the package:

```
install.packages('devtools')
devtools::install_url(
  'https://git.sr.ht/~hckiang/glinvci/blob/latest-tarballs/glinvci_latest_main.tar.gz')
```

3 High-level and low-level interface

The package contains two levels of user interfaces. The high-level interface, accessible through the `glinv` function, provides facilities for handling missing traits, lost traits, multiple evolutionary regimes, and most importantly, the calculus chain rule. The lower-level interface, accessible through the `glinv_gauss` function, allows the users to operate purely in the $(\Phi_k, w_k, V_k)_k$ parameter space. The latter parameter model obviously has huge number of parameters because k corresponds to all the nodes and tips (except the root).

Most users should be satisfied with the high-level interface, even if they intend to write their own custom models.

4 High-level interface example #1: OU Models

To fit a model using this package, generally you will need two main pieces of input data: a rooted phylogenetic tree and a matrix of trait values. The phylogenetic tree can be non-ultrametric and can potentially contain multifurcation. The matrix of trait values should have the same number of columns as the number of tips.

For the purpose of demonstrating the functionality of the package, we will first generate a random tree.

```
library(glinvci)
set.seed(1)
ntips = 300               # No. of tips
k     = 2                 # No. of trait dimensions
tr    = ape::rtree(ntips) # Random non-ultrametric tree
x0    = rnorm(k)          # Root value
```

With the above material, we are ready to make a model object. We use the OU model as an example. The OU model is generally parameterised in three matrix parameters: the drift matrix H, the evolutionary optimum θ, and the symmetric positively definite Brownian motion covariance matrix Σ. In this example, we restrict H to be a symmetric positively definite matrix while leaving θ and Σ unrestricted:

```
repar = get_restricted_ou(H='logspd', theta=NULL, Sig=NULL, lossmiss=NULL)
mod   = glinv(tr, x0, X=NULL, repar = repar)
print(mod)
# An alternative way to make the model is:
# mod   = glinv(tr, x0, X=NULL,
#               pardims = repar$nparams(k),
#               parfns  = repar$par,
#               parjacs = repar$jac,
#               parhess = repar$hess)
```

```
A GLInv model with 1 regimes and 8 parameters divided into 1 blocks,
all of which are associated to the only one existing regime.
The phylogeny has 300 tips and 299 internal nodes. Tip values are empty,
meaning that `fit()` and `lik()` etc. will not work. See `?set_tips`.
```

The first line above constructs a "re-parameterization object" `repar`, in which `'logspd'` means that H should be symmetric positively definite with its diagonals parametrized in its log scale. There are 8 parameters because the symmetric positive definite matrix H corresponds to 3 parameters (instead of 4 because of symmetry), θ has 2 elements, and Σ has 3 parameters, again, because of symmetry.

Notice that we don't have any trait vectors yet, and this is why the package says you cannot fit the model nor compute the likelihood of the model. To be able to fit the model, of course, we need some trait values as our data, and now we will generate some random trait vectors from the model and use it to fit the model.

But before we can generate data, we need to make some ground-truth parameters:

```
H     = matrix(c(1,0,0,1), k)
theta = c(0,0)
sig   = matrix(c(0.5,0,0,0.5), k)
sig_x = t(chol(sig))
diag(sig_x) = log(diag(sig_x))          # Pass the diagonal to log
sig_x = sig_x[lower.tri(sig_x,diag=T)]  # Trim away upper-tri. part and flatten.
```

In the above, the first three lines defines the actual parameters that we want, but notice that we performed a Cholesky decomposition on `sig_x` and took the logarithm of the diagonal. The package always accept the variance-covariance matrix of the Brownian motion term in this form, unless it is restricted to be a diagonal matrix. The Cholesky decomposition ensures that, during numerical optimisation in the model fitting, the matrix remain positively definite; and logarithm further constrain the diagonal of the Cholesky factor to be positive, hence eliminating multiple optima.

Because we have also constrained `H` to be symmetric positively definite (by passing `H='logspd'` to `get_restricted_ou`), we need to transform `H` in the same manner:

```
H_input = t(chol(H))
diag(H_input) = log(diag(H_input))
H_input = H_input[lower.tri(H_input,diag=T)]
```

This transformation depends on how you restrict your `H` matrix. For example, if you do not put any constrains on `H`, by passing `H=NULL` to `get_restricted_ou`, the above transformation is not needed.

Then we need to concatenate all parameters into a single vector par_truth. All OU-related functions in the package assume that the parameters are concatenated in the `(H, theta, sig_x)` order, as follows:

```
par_truth = c(H=H_input,theta=theta,sig_x=sig_x)
```

Now let's simulate the some trait data and add the these trait data into the model object `mod`. The first line below simulates a set of trait data using the parameters par_truth. The second line adds the simulated data into the mod object.

```
X = rglinv(mod, par_truth, Nsamp=1)
set_tips(mod, X[[1]])
print(mod)
```

```
A GLInv model with 1 regimes and 8 parameters divided into 1 blocks,
all of which are associated to the only one existing regime.
The phylogeny has 300 tips and 299 internal nodes. Tip values are already set.
```

As you can see, after calling `set_tips`, the warning that `fit` etc. will not work is gone.

Now let's compute the likelihood, gradient, and Hessian of this model.

```
cat('Ground-truth parameters:\n')
print(par_truth)
cat('Likelihood:\n')
print(lik(mod)(par_truth))
cat('Gradient:\n')
print(grad(mod)(par_truth))
cat('Hessian:\n')
print(hess(mod)(par_truth))
```

```
Ground-truth parameters:
    H1     H2     H3 theta1 theta2 sig_x1 sig_x2 sig_x3
  0.00   0.00   0.00   0.00   0.00  -0.35   0.00  -0.35
Likelihood:
[1] -400
Gradient:
[1]   4.3  -6.2 -27.7  -9.9 -14.9 -12.8  -5.9  35.2
Hessian:
     [,1]   [,2]   [,3] [,4] [,5]   [,6]   [,7]   [,8]
```

```
[1,] -545  -29.4    0.0   99    0  432.2   13.8    0.0
[2,]  -29 -289.0  -23.1  -13   50    2.7  328.3    9.7
[3,]    0  -23.1 -546.9    0  -26    0.0    3.9  496.3
[4,]   99  -13.0    0.0 -505    0   19.8   21.1    0.0
[5,]    0   49.5  -26.0    0 -505    0.0   14.0   29.9
[6,]  432    2.7    0.0   20    0 -574.3    5.9    0.0
[7,]   14  328.3    3.9   21   14    5.9 -574.3   11.9
[8,]    0    9.7  496.3    0   30    0.0   11.9 -670.4
```

The maximum likelihood estimates can be obtained by calling the `fit.glinv` method. We use the zero vector as the optimisation routine's initialisation:

```
par_init   = par_truth
par_init[] = 0.
fitted = fit(mod, par_init)
print(fitted)
```

```
$mlepar
     H1      H2      H3  theta1  theta2  sig_x1  sig_x2  sig_x3
-0.0247 -0.1219 -0.0058 -0.0350 -0.0485 -0.3860 -0.0783 -0.2966

$score
       H1        H2        H3    theta1    theta2    sig_x1    sig_x2    sig_x3
-0.001171 -0.000099 -0.000025  0.000358  0.000461  0.000175 -0.000051  0.000109

$loglik
[1] -398

$counts
[1] 31 31

$convergence
[1] 0

$message
[1] "CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH"
```

Once the model is fitted, one can estimate the variance-covariance matrix of the maximum-likelihood estimator using `varest`.

```
v_estimate = varest(mod, fitted)
```

The marginal confidence interval can be obtained by calling `marginal_ci` on the object returned by `varest`.

```
print(marginal_ci(v_estimate, lvl=0.99))
```

```
       Lower  Upper
H1     -0.21  0.161
H2     -0.42  0.172
H3     -0.23  0.214
theta1 -0.17  0.097
theta2 -0.18  0.083
sig_x1 -0.56 -0.215
sig_x2 -0.27  0.118
sig_x3 -0.49 -0.104
```

It seems that these confidence intervals is indeed covering the ground truth.

5 High-level interface example #2: Brownian Motion

Let's assume we have the same data `k`, `tr`, `X`, `x0` as generated before but we fit a simple Brownian motion model instead. To make a Brownian motion model, we can call the following:

```
repar_brn = get_restricted_ou(H='zero', theta='zero', Sig=NULL, lossmiss=NULL)
mod_brn   = glinv(tr, x0, X[[1]], repar=repar_brn)
print(mod_brn)
```

```
A GLInv model with 1 regimes and 3 parameters divided into 1 blocks,
all of which are associated to the only one existing regime.
The phylogeny has 300 tips and 299 internal nodes. Tip values are already set.
```

As you may might have already guessed, `H='zero'` above means that we restrict the drift matrix term of the OU to be a zero matrix. In this case, `theta`, the evolutionary optimum, is meaningless. In this case, the package has a convention that in a Brownian motion we always have `H='zero', theta='zero'`.

The following calls demonstrates how to compute the likelihood:

```
par_init_brn = c(sig_x=sig_x)
print(c(likelihood=lik(mod_brn)(par_init_brn)))
```

```
likelihood
      -536
```

The user can obtain the an MLE fit by calling `fit(mod_brn, par_init_brn)`. The marginal CI and the estimator's variance can be obtained in exactly the same way as in the OU example. But just in this example, keep in mind that we have previously generated `X` using an OU process rather than from a Brownian motion.

6 High-level interface example #3: Multiple regimes, missing data, and lost traits.

Out of box, the package allows missing data in the tip trait matrix, as well as allowing multiple revolutionary regimes.

A 'missing' trait refers to a trait value whose data is missing due to data collection problems. Fundamentally, they evolves in the same manner as other traits. An `NA` entry in the trait matrix `X` is deemed 'missing'. A lost trait is a trait dimension which had ceased to exists during the evolutionary process. An `NaN` entry in the data indicates a 'lost' trait. The package provides two different ways of handling lost traits. For more details about how missingness is handled, the users should read `?ou_haltlost`.

In this example, we demonstrate how to fit a model with two regimes, and some missing data and lost traits. Assume the phylogeny is the same as before but some entries of `X` is `NA` or `NaN`. First, let's arbitrarily set some entries of `X` to missingness, just for the purpose of demonstration.

```
X[[1]][2,c(1,2,80,95,130)] = NA
X[[1]][1,c(180:200)] = NaN
```

The following call constructs a model object in which three evolutionary regimes are present: the first regime starts at the root and it has a Brownian motion evolution; the second regime starts at the beginning of the edge that leads to internal node number 390 and it has an OU

process; and the third regime starts at the beginning of the edge that leads to internal node number 502 and it has an OU process that shares the same underlying ground-truth parameters as the second regime. In other words, there are three blocks of parameters. In a binary tree with N tips, the root's node number is $N + 1$; in other words, in our case, the root node number is 301. The following code constructs such a model:

```
repar_a = get_restricted_ou(H='logdiag', theta=NULL,   Sig=NULL, lossmiss='halt')
repar_b = get_restricted_ou(H='zero',    theta='zero', Sig=NULL, lossmiss='halt')
mod_tworeg = glinv(tr, x0, X[[1]], repar=list(repar_a, repar_b),
                   regimes = list(c(start=301, fn=2),
                                  c(start=390, fn=1),
                                  c(start=502, fn=1)))
# A long-form alternative way to do the same is this:
# mod_tworeg = glinv(tr, x0, X[[1]],
#                    pardims = list(repar_a$nparams(k), repar_b$nparams(k)),
#                    parfns  = list(repar_a$par,     repar_b$par),
#                    parjacs = list(repar_a$jac,     repar_b$jac),
#                    parhess = list(repar_a$hess,    repar_b$hess),
#                    regimes = list(c(start=301, fn=2),
#                                   c(start=390, fn=1),
#                                   c(start=502, fn=1)))
print(mod_tworeg)
```

```
A GLInv model with 3 regimes and 10 parameters divided into 2 blocks, whose
    1-7th parameters are asociated with regime no. {2,3};
    8-10th parameters are asociated with regime no. {1},
where
    regime #1 starts from the branch that leads to node #301, which is the root;
    regime #2 starts from the branch that leads to node #390;
    regime #3 starts from the branch that leads to node #502.
The phylogeny has 300 tips and 299 internal nodes. Tip values are already set.
```

In the above, we have defined three regimes and two stochastic processes. The `repar` argument, or alternatively the `pardims`, `parfns`, `parjacs`, and `parhess` argument, specifies the two stochastic processes and the `regime` parameter can be thought of as 'drawing the lines' to match each regime to a seperately defined stochastic process. The `start` element in the list specifies the node number at which a regime starts, and the `fn` element is an index to the list passed to `repar`. In this example, the first regime starts at the root and uses `repar_b`. If multiple regimes share the same `fn` index then it means that they shares both the underlying stochastic process and the parameters. `lossmiss='halt'` specifies how the lost traits (the `NaN`) are handled.

To compute the likelihood and initialize for optimisation, one need to notice the input parameters' format. When `repar` (or alternatively `parfns` etc.) has more than one elements, the parameter vector that `lik` and `fit` etc. accept is simply assumed to be the concatenation of all of its elements' parameters. The following example should illustrate this.

```
logdiagH = c(0,0)    # Meaning H is the identity matrix
theta    = c(1,0)
Sig_x_ou  = c(0,0,0) # Meaning Sigma is the identity matrix
Sig_x_brn = c(0,0,0) # The Brownian motion's parameters
print(lik(mod_tworeg)(c(logdiagH, theta, Sig_x_ou, Sig_x_brn)))
```

```
[1] -623
```

7 High-level interface example #4: Custom model with measurement error

To write custom models, we cannot use the `glinv(..., repar=repar)` shortcut anymore. Instead, one should use the `parfns`, `parjacs`, and `parhess` arguments.

It is important to note that the `parfns`, `parjacs`, and `parhess` arguments to `glinv()` are simply R functions, which the user can either create themselves or obtain from calling `get_restricted_ou()`, which is simply a convenient helper function for making the likelihood, Jacobian, Hessian functions automatically. Rather than writing all these from scratch by yourself, it is often possible to customize a model is to take the functions returned by `get_restricted_ou()` and extending them. In this example, we familiarize ourselves with these functions's input and output format and write a custom OU model with diagonal drift matrix and a diagonal additive measurement error on each tips. The additive measurement error could be estimated together if one wish (but estimating all these together may require a fairly large tree).

Nonetheless, to investigate the format of the mentioned three functions, first, we obtain the reparametrization likelihood, Jacobian, Hessian, and number of parameter functions using `get_restricted_ou()`:

```
repar = get_restricted_ou(H='diag', theta=NULL, Sig=NULL, lossmiss='halt')
print(sapply(repar, class))
```

```
       par        jac       hess    nparams
"function" "function" "function" "function"
```

We will deal with nparams later and let's look at the other three first. Recall that in the long-form calls in the previous example, `repar$par` is always passed to `parfns`, `repar$jac` always to `parjacs` and `repar$hess` to `parhess`.Mathematically, the three functions map the OU process parameters to $(\Phi_k, w_k, V_k)_k$, where k are the nodes. The input format of all three of them are the same. They accepts four parameters, and as an example, we could call

```
print(repar$par(c(1,1,0,0,0,0,0,0), 0.1, c('OK','OK'), c('OK','OK')))
```

```
[1] 0.905 0.000 0.000 0.905 0.000 0.000 0.091 0.000 0.091
```

In the call above:

1. The first argument passed to `repar$par` is the parameters of the OU model, with H being the identity matrix, represented by (1,1), the evolutionary optimum θ being the 2D zero vector, represented by (0,0), and Σ being the identity matrix, represented by (0,0,0). Therefore concatenated together we have `c(1,1,0,0,0,0,0,0)`.

2. The second argument is the branch length leading to node k.

3. The third argument is a vector of factors or string with three levels `OK`, `LOST`, `MISSING`, indicating which dimensions are missing or lost in the mother node of node k. In our case, the length of this vector is two because the we have two trait dimensions; two `OK`'s means that both the traits are "normal", neither missing nor lost.

4. The fourth argument is a vector of factors or string with the same three levels indicating the missingness of the node k. The format is the same as the third argument.

5. The return value is a concatenation of (Φ_k, w_k, V_k), flattened in column-major order, which is the R default. This means that Φ is 0.905 times the identity matrix; w is the 2D zero vector and V is 0.091 times the identity matrix.

The `repar$jac` function simply returns the Jacobian matrix of `repar$par`:

```
print(repar$jac(c(1,1,0,0,0,0,0), 0.1, c('OK','OK'), c('OK','OK')))
```

```
          [,1]    [,2]  [,3]  [,4] [,5]  [,6] [,7]
 [1,] -0.0905  0.0000 0.000 0.000 0.00 0.000 0.00
 [2,]  0.0000  0.0000 0.000 0.000 0.00 0.000 0.00
 [3,]  0.0000  0.0000 0.000 0.000 0.00 0.000 0.00
 [4,]  0.0000 -0.0905 0.000 0.000 0.00 0.000 0.00
 [5,]  0.0000  0.0000 0.095 0.000 0.00 0.000 0.00
 [6,]  0.0000  0.0000 0.000 0.095 0.00 0.000 0.00
 [7,] -0.0088  0.0000 0.000 0.000 0.18 0.000 0.00
 [8,]  0.0000  0.0000 0.000 0.000 0.00 0.091 0.00
 [9,]  0.0000 -0.0088 0.000 0.000 0.00 0.000 0.18
```

The `repar$hess` function also accepts the same argument but its return values have a slightly different format:

```
tmp = repar$hess(c(1,1,0,0,0,0,0), 0.1, c('OK','OK'), c('OK','OK'))
print(names(tmp))
```

```
[1] "V"   "w"   "Phi"
```

```
print(sapply(tmp, dim))
```

```
     V w Phi
[1,] 3 2   4
[2,] 7 7   7
[3,] 7 7   7
```

Notice that `repar$hess` returns a list containing three elements, `V`, `w`, and `Phi`, each being a three-dimensional array. They contain all the second-order partial derivatives of the `repar$par` function, with `tmp$V[m,i,j]` containing $\partial^2 V_m / \partial\eta_i \partial\eta_j$, `tmp$w[m,i,j]` containing $\partial^2 w_m / \partial\eta_i \partial\eta_j$ and `tmp$Phi[m,i,j]` containing $\partial^2 \Phi_m / \partial\eta_i \partial\eta_j$, where η denotes the vector of parameters that `repar$par` accepts and m means the index of the matrices but not the node numbers. For example, in our situation, `tmp$w[2,3,4]` contains $\partial^2 w_2 / \partial\theta_1 \partial\theta_2$ and `tmp$Phi[3,2,3]` is $\partial^2 \Phi_{21} / \partial H_{22} \partial\theta_1$.

Having understood their input and output, we are now ready to make a custom model. In this custom model, we assume that all species evolve exactly the same as specified in repar, but we cannot measure the traits at the tip accurately. To take into account this measurement error, we add an uncorrelated Gaussian error at each tip. We assume that the tree has a 800 tips in this example for simplicity. First, we extend `repar$par` to accept our additional parameters:

```
my_par = function (par, ...) {
    phiwV = repar$par(par[1:7], ...)
    if (INFO__$node_id > 800)  # If not tip just return the original
        return(phiwV)
    Sig_e = diag(par[8:9])     # Our measurement error matrix
    phiwV[7:9] = phiwV[7:9] + Sig_e[lower.tri(Sig_e, diag=T)]
    phiwV
}
```

Note that we have accessed the node ID using `INFO__$node_id`. In our package, "node IDs" means the same thing as the node numbers in the ape package, hence the nodes with ID 1-300 are the tips and the rest are the internal nodes. The `INFO__` object is neither a global variable nor an argument but a variable that lives in function's enclosing environment. Now let's define the Jacobian function, which contains the same Jacobian as the original no-measurement-error Jacobian, but with some extra entries that are either one or zero.

```
my_jac = function (par, ...) {
    new_jac = matrix(0.0, 9, 9)
    new_jac[,1:7] = repar$jac(par[1:7], ...)
    if (INFO__$node_id <= 800)
        new_jac[7,8] = new_jac[9,9] = 1.0
    new_jac
}
```

The Hessian matrix of our modified model is actually unchanged except that there are more zero entries, because the new parameters are simply a linear sum.

```
my_hess = function (par, ...)
    lapply(repar$hess(par[1:7], ...), function (H) {
        newH = array(0.0, dim=c(dim(H)[1], 9, 9))
        newH[,1:7,1:7] = H[,,]   # Copy the original part
        newH                     # Other entries are just zero
    })
```

Finally, we actually do not need to write our own `repar$nparams`, which accepts the number of trait dimensions and returns the number of parameters, beacuase we know exactly we have 9 parameters in our example. Now we can construct our custom model:

```
# Simulate a tree with 800 tips
set.seed(777)
tr = ape::rtree(800)
mod_measerr = glinv(tr, x0, NULL,
                    pardims = 9,
                    parfns  = my_par,
                    parjacs = my_jac,
                    parhess = my_hess)
print(mod_measerr)
```

Now let's make a ground truth parameter value, generate some random data and fit the model:

```
par_measerr_truth = c(H1=0.2, H2=0.5,
                      theta1=-1, theta2=1,
                      sig_x1=0, sig_x2=0, sig_x3=0,
                      sig_e1=0.4, sig_e2=0.8)
set.seed(999)
X_measerr = rglinv(mod_measerr, par_measerr_truth, Nsamp=1)
set_tips(mod_measerr, X_measerr[[1]])
## Fit the model
par_measerr_init = par_measerr_truth
par_measerr_init[] = 1.0
fitted_measerr  = fit(mod_measerr, par_measerr_init,
                      method='BB',        ## Try out different opt. methods
                      lower=c(H1=-Inf, H2=-Inf,
                              theta1=-Inf, theta2=-Inf,
                              sig_x1=-Inf, sig_x2=-Inf, sig_x3=-Inf,
                              sig_e1=1e-9, sig_e2=1e-9))
```

```
vest_measerr = varest(mod_measerr, fitted_measerr$mlepar)
confint = marginal_ci(vest_measerr, lvl=0.95)
cat('-- ESTIMATES --\n')
print(fitted_measerr)
cat('-- CONF. INTERVALS --\n')
print(confint)
```

```
-- ESTIMATES --
$mlepar
     H1      H2  theta1  theta2  sig_x1  sig_x2  sig_x3  sig_e1  sig_e2
 0.1467  0.3374 -1.8662  0.9153 -0.0407 -0.0039 -0.2291  0.4107  0.9276

$loglik
[1] -2614

$fn.reduction
[1] 498

$iter
[1] 207

$feval
[1] 208

$convergence
[1] 0

$message
[1] "Successful convergence"

$cpar
method      M
     2     50

$score
[1] -0.000042 -0.000207 -0.000413  0.000126 -0.000024  0.000062 -0.000058
[8]  0.000023 -0.000348

-- CONF. INTERVALS --
       Lower Upper
H1      0.07  0.22
H2      0.14  0.54
theta1 -3.06 -0.67
theta2  0.62  1.21
sig_x1 -0.19  0.11
sig_x2 -0.11  0.10
sig_x3 -0.56  0.11
sig_e1  0.25  0.57
sig_e2  0.69  1.17
```