Getting Started with GLinvCI

Hao Chi Kiang <hello@hckiang.com>

November, 2021

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 GLInv family, 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 (Φ_k, w_k, V_k) parameter space.

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.

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

With the above material, we are ready to make a model object. Note that we do not need an actual trait values in order to construct a model object. We use OU as an example. Here we restrict H to be a positively definite matrix using the `get_restricted_ou` function.

```
repar   = get_restricted_ou(H='logspd', theta=NULL, Sig=NULL, lossmiss=NULL)
mod     = glinv(tr, x0, NULL,
                pardims = repar$nparams(k),
                parfns  = repar$par,
                parjacs = repar$jac,
                parhess = repar$hess)
print(mod)
```

```
A GLInv model with 1 regimes and 8 parameters in total, all of which are
    associated to the only one existing regime, which starts from the root.
The phylogeny has 300 tips and 299 internal nodes.
```

Let's take an arbitrary set of parameters as an example. The following are supposed to be the ground truth parameters. Using these parameters, we will later simulate a set of trait values from the model and estimate a confidence interval.

```
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 out 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. GLInv always accept the variance-covariance matrix of the Brownian motion term in this form. The Cholesky decomposition ensures that, during numerical optimisation in the model fitting, the diagonals 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 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. We will discuss this later in this document.

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 mod object. 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]])
```

Nonetheless, 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
 0.000  0.000  0.000  0.000  0.000 -0.347  0.000 -0.347
Likelihood:
[1] -400
Gradient:
[1]   4.33  -6.24 -27.71  -9.90 -14.95 -12.85  -5.94  35.22
Hessian:
       [,1]    [,2]    [,3]   [,4]   [,5]    [,6]    [,7]    [,8]
[1,] -545.0  -29.36    0.00   99.0    0.0  432.20   13.76    0.00
[2,]  -29.4 -288.98  -23.12  -13.0   49.5    2.74  328.27    9.73
[3,]    0.0  -23.12 -546.88    0.0  -26.0    0.00    3.88  496.29
[4,]   99.0  -12.99    0.00 -504.7    0.0   19.80   21.14    0.00
[5,]    0.0   49.51  -25.98    0.0 -504.7    0.00   14.00   29.89
[6,]  432.2    2.74    0.00   19.8    0.0 -574.31    5.94    0.00
[7,]   13.8  328.27    3.88   21.1   14.0    5.94 -574.31   11.87
[8,]    0.0    9.73  496.29    0.0   29.9    0.00   11.87 -670.44
```

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
-0.02465 -0.12193 -0.00579 -0.03501 -0.04851 -0.38600 -0.07831 -0.29657

$score
       H1        H2        H3    theta1    theta2    sig_x1    sig_x2    sig_x3
-1.17e-03 -9.89e-05 -2.54e-05  3.58e-04  4.61e-04  1.75e-04 -5.11e-05  1.09e-04

$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=0.95))
```

```
        Lower   Upper
H1     -0.166  0.1168
H2     -0.346  0.1019
H3     -0.173  0.1615
theta1 -0.136  0.0656
theta2 -0.149  0.0517
sig_x1 -0.516 -0.2560
sig_x2 -0.228  0.0712
sig_x3 -0.443 -0.1501
```

Compare these to the truth. Does the confidence interval cover the truth?

5 High-level interface example #2: Brownian Motion

Let's assume we have the same data k, tr, X, x0 as generated before. To fit a standard 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]],
                  pardims = repar_brn$nparams(k),
                  parfns  = repar_brn$par,
                  parjacs = repar_brn$jac,
                  parhess = repar_brn$hess)
print(mod_brn)
```

```
A GLInv model with 1 regimes and 3 parameters in total, all of which are
    associated to the only one existing regime, which starts from the root.
The phylogeny has 300 tips and 299 internal nodes.
```

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 will throw an error if theta is not zero. In other words, for Brownian motion we always have H='zero', theta='zero'. Also note that this time we have do not need to call set_tips because the trait data are already passed to the glinv function call.

The following calls demonstrates how to compute the likelihood:

```
par_brn = c(sig_x=sig_x)
cat('Likelihood:\n')
print(lik(mod_brn)(par_brn))
```

```
Likelihood:
[1] -536
```

The user can obtain the an MLE fit by calling `fitted_brn = fit(mod_brn, par_brn)`. The marginal CI and the estimator's variance can be obtained in exactly the same way as in the OU example. But in this case, keep in mind that the truth process is an OU process, while you are fitting a Brownian motion model on it.

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 two evolutionary regimes are present: one starts at the root with Brownian motion, and another one starts at internal node number 390 with an OU process in which the drift matrix is restricted to positively definite diagonal matrices. In a binary tree with N tips, the node number of the root is always $N + 1$; in other words, in our case, the root node number is $300 + 1$.

```
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]],
                   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)))
print(mod_tworeg)
```

```
A GLInv model with 2 regimes and 10 parameters in total, among which;
    the 1~7-th parameters are asociated with regime no. {2};
    the 8~10-th parameters are asociated with regime no. {1},
where
    regime #1 starts from node #301, which is the root;
    regime #2 starts from node #390.
The phylogeny has 300 tips and 299 internal nodes.
```

In the above, we have defined two regimes and two stochastic processes. The `pardims`, `parfns`, `parjacs`, and `parhess` 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 processes. 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 `pardims`, `parfns`, `parjacs`, and `parhess`. 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 take note of the input parameters' format. When `parfns` etc. have 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)
print(lik(mod_tworeg)(c(logdiagH, theta, Sig_x_ou, Sig_x_brn)))
```

```
[1] -636
```

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

To write custom models, it is important to note that 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. Rather than writing all these from scratch by yourself, a much easier way 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. 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 later. Mathematically, `parfns`, `parjacs`, and `parhess` maps the OU process parameters to the (Φ_k, w_k, V_k). All of the three accepts the same format of four parameters. As an example:

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

```
[1] 0.9048 0.0000 0.0000 0.9048 0.0000 0.0000 0.0906 0.0000 0.0906
```

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 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)`.
2. The second argument is the branch length leading to a node.
3. The third argument is a vector of factors or string with three levels `OK`, `LOST`, and `MISS`, indicating which dimensions are missing or lost in the mother of this node. In our case, the length of this vector is two because the we have two trait dimensions; two `OK` 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 this node. The format should be the same as the third argument.
5. The return value is a concatenation of (Φ, w, V), flattened in column-major order, which is the R default. This means that Φ is $0.9048374 * I$ where I is the identity matrix; w is the 2D zero vector and V is $0.6697043 * I$.

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,] -0.09048  0.00000 0.0000 0.0000 0.000 0.0000 0.000
 [2,]  0.00000  0.00000 0.0000 0.0000 0.000 0.0000 0.000
 [3,]  0.00000  0.00000 0.0000 0.0000 0.000 0.0000 0.000
 [4,]  0.00000 -0.09048 0.0000 0.0000 0.000 0.0000 0.000
 [5,]  0.00000  0.00000 0.0952 0.0000 0.000 0.0000 0.000
 [6,]  0.00000  0.00000 0.0000 0.0952 0.000 0.0000 0.000
 [7,] -0.00876  0.00000 0.0000 0.0000 0.181 0.0000 0.000
 [8,]  0.00000  0.00000 0.0000 0.0000 0.000 0.0906 0.000
 [9,]  0.00000 -0.00876 0.0000 0.0000 0.000 0.0000 0.181
```

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$hess` returns a list containing three elements, `V`, `w`, and `Phi`, each being a three-dimensional array. They contains all the second-order partial derivatives of the `repar$par` function, with `tmp$V[m,i,j]` containing $\partial^2 V_m/\partial\eta_i\partial\eta_j$, `tmp$w[m,i,j]` contains $\partial^2 w_m/\partial\eta_i\partial\eta_j$ and `tmp$Phi[m,i,j]` contains $\partial^2\Phi_m/\partial\eta_i\partial\eta_j$, where η denotes parameter vector 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. 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—users who don't understand this can pretty much assume that it is magically there. Now let's define the Jacobian function:

```
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 for illustration
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)
```

```
A GLInv model with 1 regimes and 9 parameters in total, all of which are
    associated to the only one existing regime, which starts from the root. The
    phylogeny has 800 tips and 799 internal nodes.
```

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
 0.14669  0.33743 -1.86618  0.91532 -0.04066 -0.00391 -0.22907  0.41071
  sig_e2
 0.92757

$loglik
[1] -2614

$fn.reduction
[1] 498

$iter
[1] 248

$feval
[1] 249

$convergence
[1] 0

$message
[1] "Successful convergence"

$cpar
method      M
     2     50
```

```
$score
[1] -2.84e-05 -7.46e-05 -4.63e-04 -3.06e-06 -4.14e-05  6.01e-06  6.08e-05
[8]  1.86e-05  3.32e-05

-- CONF. INTERVALS --
         Lower  Upper
H1      0.0703  0.223
H2      0.1357  0.539
theta1 -3.0609 -0.672
theta2  0.6249  1.206
sig_x1 -0.1884  0.107
sig_x2 -0.1089  0.101
sig_x3 -0.5641  0.106
sig_e1  0.2498  0.572
sig_e2  0.6854  1.170
```

Notice that in the `fit` call, we needed to supply the lower bound of the measurement errors `sig_e1` and `sig_e2` because they should be positive; on the other hand, `sig_x1` and `sig_x2` does not need to be bounded because they are assumed to be on log-scale by default. Of course, the users are free to choose how to parameterize the two new parameters. If log-scale `sig_e1` and `sig_e2` are desired, he or she only need to change `my_par`, `my_jac`, and `my_hess` accordingly.