Getting Started with GLinvCI

Hao Chi Kiang <hello@hckiang.com>

September, 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

Beside installing from CRAN, you can use alternatively use the following command to install the latest release of the package:

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

3 Example #1: Ornstein–Uhlenbeck 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 = 200
k     = 2                                  # No. of trait dimensions
tr    = ape::rtree(ntips)
```

```
X     = matrix(rnorm(k*ntips), k, ntips) # Trait matrix
x0    = rnorm(k)                         # Root value
```

With the above material, we are ready to make a model object. Here we restrict our drift matrix H to be a positively definite matrix, while leaving the optimum vector and Brownian motion covariance matrix unrestricted.

```
repar = get_restricted_ou(H='logspd', theta=NULL, Sig=NULL, lossmiss=NULL)
mod   = glinv(tr, x0, X,
              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 200 tips and 199 internal nodes.
```

Let's take an arbitrary parameters as an example: The following code demostrates how to computing the model's likelihood, gradient, and Hessian at an arbitrarily specified pararmenter:

```
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.

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

```
par_init = c(H=H_input,theta=theta,sig_x=sig_x)
cat('Initial parameters:\n')
print(par_init)
cat('Likelihood:\n')
print(lik(mod)(par_init))
cat('Gradient:\n')
print(grad(mod)(par_init))
cat('Hessian:\n')
print(hess(mod)(par_init))
```

```
Initial parameters:
       H1        H2        H3    theta1    theta2    sig_x1    sig_x2
          sig_x3
 0.000000  0.000000  0.000000  0.000000  0.000000 -0.346574  0.000000
     -0.346574
Likelihood:
[1] -1451.43
Gradient:
[1] -519.02035   45.62066 -500.23748   -7.64877  -58.73828 1294.12584
    158.72065
[8] 1078.79563
Hessian:
           [,1]      [,2]       [,3]      [,4]      [,5]       [,6]
[1,] -1787.1716   62.9472     0.0000  -25.5207    0.0000  1342.9275
[2,]    62.9472 -877.1531    17.3265  -63.7235  -12.7604   -28.5241
[3,]     0.0000   17.3265 -1759.0064    0.0000 -127.4470     0.0000
[4,]   -25.5207  -63.7235     0.0000 -367.6268    0.0000    15.2975
[5,]     0.0000  -12.7604  -127.4470    0.0000 -367.6268     0.0000
[6,]  1342.9275  -28.5241     0.0000   15.2975    0.0000 -2988.2517
[7,]   -88.6955  936.3116   -40.3392   83.0685   10.8170  -158.7206
[8,]     0.0000  -62.7172  1305.3618    0.0000  117.4766     0.0000
           [,7]       [,8]
[1,]   -88.6955     0.0000
[2,]   936.3116   -62.7172
[3,]   -40.3392  1305.3618
[4,]    83.0685     0.0000
[5,]    10.8170   117.4766
[6,]  -158.7206     0.0000
[7,] -2988.2517  -317.4413
[8,]  -317.4413 -2557.5913
```

The maximum likelihood estimates can be obtained by calling the `fit.glinv` method. We use the `parinit` which we have constructed before as the optimisation routine's initialisation:

```
fitted = fit(mod, par_init)
print(fitted)
```

```
$mlepar
        H1         H2         H3     theta1     theta2     sig_x1     sig_x2
 1.7883480  3.2584507  0.9489094 -0.0417397 -0.0967488  2.2131622  4.6633817
    sig_x3
 1.3769953

$score
          H1           H2           H3       theta1       theta2       sig_x1
 0.000923787 -0.001891418  0.000803219 -0.000420278  0.000491690 -0.001620165
      sig_x2       sig_x3
 0.000240675 -0.002768440

$loglik
[1] 599.385

$counts
[1] 493 493

$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      -2.564415  6.1411108
H2     -16.193767 22.7106680
H3       0.103568  1.7942504
theta1  -0.194444  0.1109650
theta2  -0.247124  0.0536267
sig_x1  -2.115801  6.5421255
sig_x2 -24.044181 33.3709443
sig_x3   0.528665  2.2253252
```

Notice that some of the parameters have fairly large confidence intervals and the likelihood surface is quite flat in some directions. This suggests that perhaps we do not have enough data to precisely estimate all the parameters, or the data does not allow us to estimate the parameters well. This makes sense because we have generated the data in a way that is completely unrelated to the evolutionary model.

4 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,
                  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 200 tips and 199 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'`.

The following calls demonstrates how to compute the likelihood:

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

```
Likelihood:
[1] -1158.31
```

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.

5 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[2,c(1,2,80,95,130)] = NA
X[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 290 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 $200 + 1$.

```
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,
                   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=201, fn=2),
                                  c(start=290, 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 #201, which is the root;
    regime #2 starts from node #290.
The phylogeny has 200 tips and 199 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] -760.301
```

6 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, as before, we generate some random data:

```
library(glinvci)
set.seed(1)
ntips = 200
k     = 2                                # No. of trait dimensions
tr    = ape::rtree(ntips)
X     = matrix(rnorm(k*ntips), k, ntips) # Trait matrix
x0    = rnorm(k)                         # Root value
```

Then we obtain the 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. 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] 0.904837 0.000000 0.000000 0.904837 0.000000 0.000000 0.090635 0.000000
[9] 0.090635
```

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 the 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 the 2-by-2 square matrix $0.9048374 * I$ where I is the identity matrix; w is the 2D zero vector and V is $0.090635 * I$. There are only three numbers representing V because only the lower-triangular part is needed.

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,] -0.0904837  0.0000000 0.000000 0.000000 0.00000 0.000000 0.00000
 [2,]  0.0000000  0.0000000 0.000000 0.000000 0.00000 0.000000 0.00000
 [3,]  0.0000000  0.0000000 0.000000 0.000000 0.00000 0.000000 0.00000
 [4,]  0.0000000 -0.0904837 0.000000 0.000000 0.00000 0.000000 0.00000
 [5,]  0.0000000  0.0000000 0.095163 0.000000 0.00000 0.000000 0.00000
 [6,]  0.0000000  0.0000000 0.000000 0.095163 0.00000 0.000000 0.00000
 [7,] -0.0087615  0.0000000 0.000000 0.000000 0.18127 0.000000 0.00000
 [8,]  0.0000000  0.0000000 0.000000 0.000000 0.00000 0.090635 0.00000
 [9,]  0.0000000 -0.0087615 0.000000 0.000000 0.00000 0.000000 0.18127
```

The `repar$hess` function stiill 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 $\partial V_m / \partial \eta_i \partial \eta_j$, `tmp$w[m,i,j]` contains $\partial w_m / \partial \eta_i \partial \eta_j$ and

`tmp$Phi[m,i,j]` contains $\partial\Phi_m/\partial\eta_i\partial\eta_j$, 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 $\partial w_2/\partial\theta_1\partial\theta_2$ and `tmp$Phi[3,2,3]` is $\partial\Phi_{21}/\partial H_{22}\partial\theta_1$.

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. First, we extend `repar$par` to accept our additional parameters:

```
my_par = function (par, ...) {
    phiwV = repar$par(par[1:7], ...)
    if (INFO__$node_id > 200)  # 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 1-200 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 <= 200)
        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:

```
mod = glinv(tr, x0, X,
            pardims = 9,
            parfns  = my_par,
            parjacs = my_jac,
            parhess = my_hess)
print(mod)
```

```
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 200 tips and 199 internal nodes.
```

Now the user can fit the model as usual.

```
fitted  = fit(mod, par_init=c(1,1,0,0,0,0,0,0.5,0.5))
confint = marginal_ci(varest(mod, fitted$mlepar), lvl=0.95)
```