

Getting Started with GLInvCI

Hao Chi Kiang <hello@hckiang.com>

20 August, 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(
  '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 Using the high-level interface: Brownian Motion and 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 = 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. We use OU as an example. Here we restrict H to be a positively definite matrix.

```
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 2 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 demonstrates how to computing the model's likelihood, gradient, and Hessian at an arbitrarily specified parameter:

```
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 constraints 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]      [,7]
[1,] -1787.1716  62.9472    0.0000  -25.5207    0.0000  1342.9275  -88.6955
[2,]   62.9472 -877.1531   17.3265  -63.7235  -12.7604  -28.5241   936.3116
[3,]    0.0000  17.3265 -1759.0064    0.0000 -127.4470    0.0000  -40.3392
[4,]   -25.5207  -63.7235    0.0000 -367.6268    0.0000   15.2975   83.0685
[5,]    0.0000  -12.7604  -127.4470    0.0000 -367.6268    0.0000   10.8170
[6,]  1342.9275  -28.5241    0.0000   15.2975    0.0000 -2988.2517  -158.7206
[7,]  -88.6955   936.3116  -40.3392   83.0685   10.8170  -158.7206 -2988.2517
[8,]    0.0000  -62.7172  1305.3618    0.0000  117.4766    0.0000  -317.4413
      [,8]
[1,]  0.0000
[2,] -62.7172
[3,] 1305.3618
[4,]  0.0000
[5,]  117.4766
[6,]  0.0000
[7,] -317.4413
[8,] -2557.5913
```

The maximum likelihood estimates can be obtained by calling the `fit.glmv` 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
1.7473088 3.0687662 0.9462338 -0.0417478 -0.0967537 2.1723982 4.3828934
      sig_x3
1.3741767

$loglik
[1] 599.386
```

```

$counts
[1] 852 345

$convergence
[1] 0

$message
[1] "Rcgmin seems to have converged"

$score
[1] 0.0010458539 0.0012164202 0.0019767975 0.0000223676 0.0001637558
[6] -0.0000594491 0.0027448292 0.0013087048

```

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	-0.986260	4.4808777
H2	-8.798587	14.9361198
H3	0.122600	1.7698674
theta1	-0.194455	0.1109592
theta2	-0.247122	0.0536148
sig_x1	-0.546810	4.8916064
sig_x2	-13.162203	21.9279898
sig_x3	0.549033	2.1993203

Notice that some of the parameters have fairly large confidence intervals. This suggests that perhaps we do not have enough data to precisely estimate all the parameters.