1. Prepare Directory

Before you start, make sure git is installed on your PC. If not, you can skip this step and just manually transfer the files into your current working directory.

Transfer the files with git to your computer.

```
!git clone https://github.com/diogo-s-ribeiro/delta-statistic
```

Move to the correct directory. Again, you can just manually open python directly in the "Delta-Python" folder.

```
!cd ./delta-statistic/Delta-Python/
```

2. Install and import Packages that are needed in this new

You can now install packages and libraries specific to this environment using "pip". If you only want to run a specific part of the code, you can install it one-by-one, otherwise it's simpler to run in the terminal:

```
pip install -r requirements.txt
```

Make sure your python version is up-to-date.

```
!python --version
```

Now simply import the needed packages to run the code:

```
import os, math, re
import numpy as np
import pandas as pd
from scipy.stats import entropy
from numba import njit, float64, int64
```

3. Import the delta-statistic (python) code

The delta function calculates the delta-statistic after an MCMC step. It uses the emcmc function to obtain Markov chain samples and then calculates the delta statistic. The **@njit** decorator is used for just-in-time compilation, which can improve the performance of the code.

To run it, first import the delta-statistic code available in Python. This can be done either by:

References

Borges, R. et al. (2019). Measuring phylogenetic signal between categorical traits and phylogenies. Bioinformatics, 35, 1862-1869.

Diogo, R. (Github). Assessing traits and phylogenetic signal to unravel the tempo and mode of phenotypic evolution.

3A. Importing from file

Importing the provided delta-statistic file, if it is in the same working directory;

```
from delta_functs import delta
```

3B. Directly (Code)

Copying all of the delta-statistic related functions.

```
# Metropolis-Hastings step for alpha parameter
@njit(float64(float64, float64, float64[::1], float64, float64))
def mhalpha(a,b,x,l0,se):
    '''a = The current value of the alpha parameter.
    b    = The current value of the beta parameter.
    x    = An array of data points used in the acceptance ratio computations, after uncertainty is calculated.
    l0   = A constant value used in the acceptance ratio computations.
    se   = The standard deviation used for the random walk in the Metropolis-Hastings algorithm.'''

    a1   = np.exp(np.random.normal(np.log(a),se, 1))[0]
    lp_a = np.exp( (len(x)*(math.lgamma(a1+b)-math.lgamma(a1)) - a1*(l0-np.sum(np.log(x)))) -
(len(x)*(math.lgamma(a+b)-math.lgamma(a)) - a*(l0-np.sum(np.log(x)))) )
    r    = min( 1, lp_a )
```

```
    # Repeat until a valid value is obtained
    while (np.isnan(lp_a) == True):
        a1   = np.exp(np.random.normal(np.log(a),se, 1))[0]
        lp_a = np.exp( (len(x)*(math.lgamma(a1+b)-math.lgamma(a1)) - a1*(l0-np.sum(np.log(x)))) -
(len(x)*(math.lgamma(a+b)-math.lgamma(a)) - a*(l0-np.sum(np.log(x)))) )
        r    = min( 1, lp_a )

    # Accept or reject based on the acceptance ratio
    if np.random.uniform(0,1) < r:
        return a1
    else:
        return a


# Metropolis-Hastings step for beta parameter
@njit(float64(float64, float64, float64[::1], float64, float64))
def mhbeta(a,b,x,l0,se):
    '''a = The current value of the alpha parameter.
    b    = The current value of the beta parameter.
    x    = An array of data points used in the acceptance ratio computations, after uncertainty is calculated.
    l0   = A constant value used in the acceptance ratio computations.
    se   = The standard deviation used for the random walk in the Metropolis-Hastings algorithm.'''

    b1   = np.exp(np.random.normal(np.log(b),se,1))[0]
    lp_b = np.exp( (len(x)*(math.lgamma(a+b1)-math.lgamma(b1)) - b1*(l0-np.sum(np.log(1-x)))) -
(len(x)*(math.lgamma(a+b)-math.lgamma(b)) - b*(l0-np.sum(np.log(1-x)))) )
    r    = min( 1, lp_b )

    # Repeat until a valid value is obtained
    while (np.isnan(lp_b) == True):
        b1   = np.exp(np.random.normal(np.log(b),se,1))[0]
        lp_b = np.exp( (len(x)*(math.lgamma(a+b1)-math.lgamma(b1)) - b1*(l0-np.sum(np.log(1-x)))) -
(len(x)*(math.lgamma(a+b)-math.lgamma(b)) - b*(l0-np.sum(np.log(1-x)))) )
        r    = min( 1, lp_b )

    # Accept or reject based on the acceptance ratio
    if np.random.uniform(0,1) < r:
```

```
        return b1
    else:
        return b


# Metropolis-Hastings algorithm using alpha and beta
@njit(float64[:, ::1](float64, float64, float64[::1], float64, float64, int64, int64, int64))
def emcmc(alpha,beta,x,l0,se,sim,thin,burn):
    '''alpha = The initial value of the alpha parameter.
    beta     = The initial value of the beta parameter.
    x        = An array of data points used in the acceptance ratio computations, after uncertainty is calculated.
    l0       = A constant value used in the acceptance ratio computations.
    se       = The standard deviation used for the random walk in the Metropolis-Hastings algorithm.
    sim      = The number of total iterations in the Markov Chain Monte Carlo (MCMC) simulation.
    thin     = The thinning parameter, i.e., the number of iterations to discard between saved samples.
    burn     = The number of burn-in iterations to discard at the beginning of the simulation.'''

    n_size = np.linspace(burn, sim, int((sim - burn) / thin + 1))
    usim   = np.round(n_size, 0, np.empty_like(n_size))
    gibbs  = []
    p      = 0

    for i in range(sim+1):
        alpha = mhalpha(alpha,beta,x,l0,se)
        beta  = mhbeta(alpha,beta,x,l0,se)

        if i == usim[p]:
            gibbs.append((alpha, beta))
            p += 1

    gibbs = np.asarray(gibbs)
    return gibbs


# Calculate uncertainty using different types
def entropy_type(prob, ent_type):
    '''prob  = A matrix of ancestral probabilities.
    ent_type = A string indicating the type of entropy calculation. (options: 'LSE', 'SE', or any other value for
```

```
Gini impurity).'''

    # Linear Shannon Entropy
    if ent_type == 'LSE':
        k    = np.shape(prob)[1]
        prob = np.asarray(np.where(prob<=(1/k), prob, prob/(1-k) - 1/(1-k)))
        tent = np.sum(prob, 1)

        # Ensure absolutes
        tent = np.asarray(np.where(tent != 0, tent, tent + np.random.uniform(0,1,1)/10000))
        tent = np.asarray(np.where(tent != 1, tent, tent - np.random.uniform(0,1,1)/10000))

        return tent

    # Shannon Entropy
    elif ent_type == 'SE':
        k    = np.shape(prob)[1]
        tent = entropy(prob, base=k, axis=1)

        # Ensure absolutes
        tent = np.asarray(np.where(tent != 0, tent, tent + np.random.uniform(0,1,1)/10000))
        tent = np.asarray(np.where(tent != 1, tent, tent - np.random.uniform(0,1,1)/10000))

        return tent

    # Ginni Impurity
    else:
        k    = np.shape(prob)[1]
        tent = ((1 - np.sum(prob**2, axis=1))*k)/ (k - 1)

        # Ensure absolutes
        tent = np.asarray(np.where(tent != 0, tent, tent + np.random.uniform(0,1,1)/10000))
        tent = np.asarray(np.where(tent != 1, tent, tent - np.random.uniform(0,1,1)/10000))

        return tent


# Calculate delta-statistic after an MCMC step
```

```
def delta(x,lambda0,se,sim,thin,burn,ent_type):
    '''x     = A matrix of ancestral probabilities.
    lambda0  = A constant value used in the acceptance ratio computations.
    se       = The standard deviation used for the random walk in the Metropolis-Hastings algorithm.
    sim      = The number of total iterations in the Markov Chain Monte Carlo (MCMC) simulation.
    thin     = The thinning parameter, i.e., the number of iterations to discard between saved samples.
    burn     = The number of burn-in iterations to discard at the beginning of the simulation.
    ent_type = A string specifying the type of entropy calculation (options: 'LSE', 'SE', or any other value for Gini
impurity).'''

    mc1    = emcmc(np.random.exponential(),np.random.exponential(),entropy_type(x,
ent_type),lambda0,se,sim,thin,burn)
    mc2    = emcmc(np.random.exponential(),np.random.exponential(),entropy_type(x,
ent_type),lambda0,se,sim,thin,burn)
    mchain = np.concatenate((mc1,mc2), axis=0)

    deltaA = (np.mean(mchain[:,1]))/(np.mean(mchain[:,0]))

    return deltaA
```

4. Ancestral Probabilities

The ancestral probabilities are needed before calculating their respective delta-statistic. This can be done either through Maximum Likelihood or Bayesian inference. Multiple software packages are currently available that can perform ancestral state reconstruction. Choose the best suited for your needs!

Since only the matrix with ancestral probabilities is needed, you can:

4A. Input them directly from a file, after calculating them through an external source

```
def read_file_ace(file_path, separator=',', rmv_col_name=True, rmv_row_name=True):
    ''' file_path = Represents the path to the file that you want to read.
    separator     (default: ',')  = Specifies the separator used in the file to separate the values.
    rmv_col_name  (default: True) = Boolean value that determines whether the first row (column names) should be
removed from the array.
```

```
    rmv_row_name  (default: True) = Boolean value that determines whether the first column (index) should be removed
from the array.'''

    # Read the data from the file using numpy's genfromtxt function
    array = np.genfromtxt(file_path, delimiter = separator)

    # Check if the column name should be removed
    if rmv_col_name == True:
        array = np.delete(array, 0, axis=0)     # Remove the first row (Trait Name)

    # Check if the row name should be removed
    if rmv_row_name == True:
        array = np.delete(array, 0, axis=1)     # Remove the first column (Entity Name)

    return array
```

4B. Or use a package to calculate them directly in Python (e.g.)

4B.1. PastML

A good package for Ancestral Character Estimation (through Maximum Likelihood) in Python is PastML. If you intend to run PastML don't forget to install the package.

References

Ishikawa, S. A. et al. (2019). A fast likelihood method to reconstruct and visualize ancestral scenarios. Molecular Biology and Evolution, 36, 2069-2085.


```
from pastml.tree import read_tree, name_tree
from pastml.acr import acr
from pastml.annotation import preannotate_forest
from pastml import col_name2cat
from collections import defaultdict, Counter
```

```
def _validate_input(tree_nwk, data, data_sep=',', single_tree_file=False):
    '''tree_nwk      = Represents the path to the Newick file containing the tree or a string with the tree itself.
    data             = Represents the path to the data file or DataFrame used for annotation with leaf states.
    data_sep         (default: ',')   = Separator used in the data file.
    single_tree_file (default: False) = Boolean value that specifies whether the input tree is provided as a single file.'''

    if single_tree_file==False:
        with open(tree_nwk, 'r') as f:                                  # Reads the tree from a Newick file and returns its roots
            nwks = f.read().replace('\n', '')
        roots = [read_tree(tree_nwk)]
    else:
        roots = [read_tree(tree_nwk)]                                   # Reads the newick tree and returns its roots

    column2annotated = Counter()                                        # Counter to keep track of the number of times each column is annotated
    column2states    = defaultdict(set)                                 # Dictionary to store the unique states for each column

    # Read the data as a pandas DataFrame
    df         = pd.read_csv(data, sep=data_sep, index_col=0, header=0, dtype=str)
    df.index   = df.index.map(str)
    df.columns = [col_name2cat(column) for column in df.columns]
    columns    = df.columns

    node_names     = set.union(*[{n.name for n in root.traverse() if n.name} for root in roots])     # Get the names of the nodes in the tree
    df_index_names = set(df.index)                                      # Get the index names from the DataFrame
    common_ids     = list(node_names & df_index_names)                  # Find the common IDs between node names and
DataFrame index names

    # strip quotes if needed
    if not common_ids:
        node_names = {_.strip("'").strip('"') for _ in node_names}
        common_ids = node_names & df_index_names
        if common_ids:
            for root in roots:
                for n in root.traverse():
```

```
            n.name = n.name.strip("'").strip('"')

# Preannotate the forest with the DataFrame
preannotate_forest(roots, df=df)

# Populate the column2states dictionary with unique states for each column
for c in df.columns:
    column2states[c] |= {_ for _ in df[c].unique() if pd.notnull(_) and _ != ''}

num_tips = 0

# Count the number of annotated columns for each node
column2annotated_states = defaultdict(set)
for root in roots:
    for n in root.traverse():
        for c in columns:
            vs = getattr(n, c, set())
            column2states[c] |= vs
            column2annotated_states[c] |= vs
            if vs:
                column2annotated[c] += 1
        if n.is_leaf():
            num_tips += 1

if column2annotated:
    c, num_annotated = min(column2annotated.items(), key=lambda _: _[1])
else:
    c, num_annotated = columns[0], 0

# Calculate the percentage of unknown tip annotations
percentage_unknown = (num_tips - num_annotated) / num_tips
if percentage_unknown >= .9:
    raise ValueError('{:.1f}% of tip annotations for character "{}" are unknown, '
                     'not enough data to infer ancestral states. '
```

```
                    '{}'
                    .format(percentage_unknown * 100, c,
                            'Check your annotation file and if its ids correspond to the tree tip/node names.'
                            if data
                            else 'You tree file should contain character state annotations, '
                                 'otherwise consider specifying a metadata file.'))
    c, states = min(column2annotated_states.items(), key=lambda _: len(_[1]))

    # Check if the number of unique states is too high for the given number of tips
    if len(states) > num_tips * .75:
        raise ValueError('Character "{}" has {} unique states annotated in this tree: {}, '
                         'which is too much to infer on a {} with only {} tips. '
                         'Make sure the character you are analysing is discrete, and if yes use a larger tree.'
                         .format(c, len(states), states, 'tree' if len(roots) == 1 else 'forest', num_tips))


    # Convert column2states to numpy arrays and sort the states
    column2states = {c: np.array(sorted(states)) for c, states in column2states.items()}

    # Name the trees in the forest
    for i, tree in enumerate(roots):
        name_tree(tree, suffix='' if len(roots) == 1 else '_{}'.format(i))

    return roots, columns, column2states


def marginal(tree, data, prediction_method='MPPA', model='F81', threads=0, single_tree_file=False):
    '''tree           = Represents the path to the Newick file containing the tree or a string with the tree itself.
    data              = Represents the path to the data file or DataFrame used for annotation with leaf states.
    prediction_method (default: 'MPPA') = Specifies the ancestral character prediction method.
    model             (default: 'F81')  = Specifies the evolutionary model used for reconstruction.
    threads           (default: 0)      = Specifies the number of threads to use for the analysis.
    single_tree_file  (default: False)  = Boolean value that specifies whether the input tree is provided as a single file.'''
```

```
    # Set the number of threads based on the available CPU cores
    if threads < 1:
        threads = max(os.cpu_count(), 1)

    # Validate the input and get the roots, columns, and column2states
    roots, columns, column2states = \
        _validate_input(tree_nwk=tree, data=data, data_sep=',', single_tree_file=single_tree_file)

    # Perform the ancestral character reconstruction (ACR) analysis
    acr_results = acr(forest=roots, columns=columns, column2states=column2states, prediction_method=prediction_method, model=model,
threads=threads)

    # Get the leaf names from the tree
    leaf_names = read_tree(tree).get_leaf_names()

    # Get the marginal probabilities and exclude the leaf nodes
    marginal   = np.asarray( acr_results[0]['marginal_probabilities'].drop(leaf_names) )

    return marginal
```

4B.2. rpy2:ape

Another popular option to calculate the marginal probabilities would be to use the ape package in Python through rpy2.
Once again, don't forget to install the necessary packages.

PS. If the ancestral probabilities matrix can't properly be calculated, resulting in a matrix with multiple **-NaN-** values, there might be a problem related to the tree branch lengths. Try uncommenting the "tree$edge.length" line and rerun the code.

References

Paradis, E. and Schliep, K. (2019). ape 5.0: an environment for modern phylogenetics and evolutionary analyses in R. Bioinformatics, 35, 526-528.

```
from rpy2.robjects import r
```

```
def to_ape(dir_tree, dir_data, instal_ape=False, tree_dir=False):
    '''dir_tree = Represents the path to the Newick file containing the tree or a string with the tree itself.
    dir_data    = Represents the path to the data file or DataFrame used for annotation with leaf states.
    instal_ape  (default: False) = Boolean value that determines whether to install the 'ape' package or not.
    tree_dir    (default: False) = Boolean value that specifies whether the dir_tree variable should be used to load the tree data from a file.'''

    traits_df = pd.read_csv(dir_data, sep=',', index_col=0, header=0, dtype=str).sort_index().iloc[:,0]                          # Reading traits data
from a CSV file and sorting it
    traits    = tuple(  traits_df.replace( np.sort(np.unique(traits_df)), list(range( 1, 1+len(np.unique(traits_df))))  )        # Replacing unique traits
with numerical values

    # Loading tree data from a file
    if tree_dir==True:
        dir_tree = np.loadtxt( dir_tree , dtype=str)

    # The R code installs the 'ape' package if "instal_ape"=True
    if instal_ape == True:
        instal_ape = ''
    else:
        instal_ape = '#'

    marginal_prob  = r(
        """
        # Import Ape
        {instal_ape}install.packages('ape')
        library('ape')

        # Tree Import and transform
        tree_newick <- read.tree(text="{dir_tree}")
        tree <- multi2di( tree_newick )
        tree$edge.length <- tree$edge.length + runif(tree$edge.length, 0, 1e-7)
```

```
    # Trait vector import and order vector correctly depending on the tree
    trait <- c{traits}
    trait <- trait[ rank( tree$tip.label ) ]

    # Marginal Probabilities calculate
    ar <- ace(trait,tree,type="discret",method="ML",model="ARD")$lik.anc

    """.format(instal_ape=instal_ape, dir_tree=dir_tree, traits=traits)
  )

  return marginal_prob
```

5. Calculate Delta-Statistic

```
lambda0  = 0.1                      # rate parameter of the proposal
se       = 0.5                      # standard deviation of the proposal
sim      = 100000                   # number of iterations
thin     = 10                       # Keep only each xth iterate
burn     = 100                      # Burned-in iterates

ent_type = 'LSE'                    # Linear Shannon Entropy
```

It is also possible to calculate uncertainty using a normalized version of both:

1) The Shannon Entropy: A widely used measure of information content or uncertainty in a random variable;
2) The Gini impurity: Measure that can be used to quantify the impurity or disorder in a set of class labels.

ACE (see above "4. Ancestral Probabilities") and Delta Calculation

Before starting, make sure that:

1) The necessary files are in the correct directory;
2) The functions of the preferred ACE and delta calculation have already run.

A1) (Single) Input directly

```
path_ap      = r"./input/Simplified/Ancestral_Probabilities/FILE"   # Path to (ap: ancestral probabilities) files
file         = "Simplified_AP_1.txt"                                # File with Ancestral Probabilities
file         = path_ap.replace('FILE', file)                        # Path + file

Ancest_Prob  = read_file_ace(file_path=file, separator=',', rmv_col_name=True, rmv_row_name=True)
Delta_Final  = delta(x=Ancest_Prob, lambda0=lambda0, se=se, sim=sim, burn=burn, thin=thin, ent_type='LSE')
print( Delta_Final )
```

A2) (Multiple) Input directly

```
def multiple_files_ap( dir=None, separator=',', rmv_col_name=True, rmv_row_name=True, lambda0=0.1, se=0.5,
sim=100000, thin=10, burn=100, ent_type='LSE' ):
    '''dir       (default: None)    = Represents the directory path with the ancestral probabilities matrices. If no
value is provided when calling the function, the user will be prompted to input the directory path.
    separator    (default: ',')     = Determines the separator used in the matrices when reading the files.
    rmv_col_name (default: True)    = Boolean value that indicates whether the column names should be removed when
reading the files.
    rmv_row_name (default: True)    = Boolean value that indicates whether the row names should be removed when
reading the files.
    lambda0      (default: 0.1)     = A constant value used in the acceptance ratio computations.
    se           (default: 0.5)     = The standard deviation used for the random walk in the Metropolis-Hastings
algorithm.
    sim          (default: 100000)  = The number of total iterations in the Markov Chain Monte Carlo (MCMC)
simulation.
    thin         (default: 10)      = The thinning parameter, i.e., the number of iterations to discard between saved
samples.
    burn         (default: 100)     = The number of burn-in iterations to discard at the beginning of the simulation.
    ent_type     (default: 'LSE')   = A string specifying the type of entropy calculation (options: 'LSE', 'SE', or
any other value for Gini impurity).'''
    if dir == None:
        dir = str(input( 'What is the directory path with the ancestral probabilities matrices?' ))

    dic_files = set( os.listdir(dir) )
    dic_delta = set()
    for file in dic_files:
        file = dir + file
```

```
        ace = read_file_ace( file, separator, rmv_col_name, rmv_row_name )
        dic_delta.add( delta(x=ace, lambda0=lambda0, se=se, sim=sim, burn=burn, thin=thin, ent_type=ent_type) )

    return dict(zip( dic_files, dic_delta ))
```

```
Delta_Final_dic = multiple_files_ap( dir = './input/Simplified/Ancestral_Probabilities/' )
print( Delta_Final_dic )
print( Delta_Final_dic.values() )
```

B1) (Single) PastML

```
path_data  = r"./input/3Class/3C_States.txt"          # File containing tip/node annotations, in csv or tab format
path_tree  = r"./input/3Class/Trees/3C_Trees_1.txt"   # File containing tip/node annotations, in csv or tab format

method     = "MPPA"                                   # MPPA, MAP
model      = "F81"                                    # F81, JC, EFT

Ancest_Prob = marginal(path_tree, path_data)
Delta_Final = delta(x=Ancest_Prob, lambda0=lambda0, se=se, sim=sim, burn=burn, thin=thin, ent_type='LSE')
print(Delta_Final)
```

B2) (Multiple) PastML

```
def multiple_files_PastML( dir_tree=None, dir_data=None, single_tree_file=False, threads=0, lambda0=0.1, se=0.5,
sim=100000, thin=10, burn=100, ent_type='LSE' ):
    '''dir_tree      (default: None)    = Represents the directory path to the phylogenetic tree(s) file(s). If no
value is provided when calling the function, the user will be prompted to input the directory path.
    dir_data         (default: None)    = Represents the directory path to the observed states file(s). If no value is
provided when calling the function, the user will be prompted to input the directory path.
    single_tree_file (default: False)   = Boolean value that indicates whether the phylogenetic tree(s) file(s) are
stored in a single file (True) or multiple files (False).
    threads          (default: None)    = This variable represents the number of threads used in the marginal
function.
    lambda0          (default: 0.1)     = A constant value used in the acceptance ratio computations.
    se               (default: 0.5)     = The standard deviation used for the random walk in the Metropolis-Hastings
algorithm.
```

```
    sim              (default: 100000) = The number of total iterations in the Markov Chain Monte Carlo (MCMC)
simulation.
    thin             (default: 10)     = The thinning parameter, i.e., the number of iterations to discard between
saved samples.
    burn             (default: 100)    = The number of burn-in iterations to discard at the beginning of the
simulation.
    ent_type         (default: 'LSE')  = A string specifying the type of entropy calculation (options: 'LSE', 'SE',
or any other value for Gini impurity).'''

    if dir_tree == None:
        dir_tree = str(input( 'What is the directory path to the phylogenetic tree(s) file(s)?' ))
    if dir_data == None:
        dir_data = str(input( 'What is the directory path to the observed states file(s)?' ))
    if single_tree_file==False:
        dic_files = set( os.listdir( dir_tree ) )
        dic_delta = set()
        for phylo_tree in dic_files:
            Ancest_Prob  = marginal( dir_tree+'/'+phylo_tree, dir_data, single_tree_file=single_tree_file,
threads=threads )
            dic_delta.add( delta(x=Ancest_Prob, lambda0=lambda0, se=se, sim=sim, burn=burn, thin=thin,
ent_type=ent_type) )
        dic_final = dict(zip( dic_files, dic_delta ))
    else:
        dic_final = {}
        arr       = np.loadtxt( dir_tree , dtype=str)
        indx      = 0
        for phylo_tree in arr:
            Ancest_Prob = marginal( phylo_tree, dir_data, single_tree_file=single_tree_file, threads=threads )
            delta_value = delta(x=Ancest_Prob, lambda0=lambda0, se=se, sim=sim, burn=burn, thin=thin,
ent_type=ent_type)
            phylo_tree  = 'Tree:' + str(indx) + '_' + phylo_tree
            dic_final[phylo_tree] = delta_value

            indx += 1
    return dic_final
```

```
path_tree  = r"./input/3Class/Trees"
```

```
Delta_Final = multiple_files_PastML(path_tree, path_data, single_tree_file=False)
print( Delta_Final )

path_tree  = r"./input/3Class/3C_Multiple_Trees.txt"
Delta_Final = multiple_files_PastML(path_tree, path_data, single_tree_file=True)
print( Delta_Final )
print( Delta_Final.values() )
```

PS. Before starting don't forget to install ape with the variable "install_ape = True" and select a CRAN mirror.

ATENTION:

You can select any "Secure CRAN mirror" but it is a good practice to choose a mirror closer to your location or one that is known to be reliable. CRAN (Comprehensive R Archive Network) is a network of servers worldwide that distribute R packages.

C1) (Single) rpy2:ape

```
path_tree = "./input/2Class/Trees/2C_Trees_1.txt"
path_data = "./input/2Class/2C_States.txt"

Ancest_Prob = np.asarray( to_ape(path_tree, path_data, instal_ape=True, tree_dir=True) )
Delta_Final = delta(x=Ancest_Prob, lambda0=lambda0, se=se, sim=sim, burn=burn, thin=thin, ent_type='LSE')
print( Delta_Final )
```

C2) (Multiple) rpy2:ape

```
def multiple_files_Ape( dir_tree=None, dir_data=None, single_tree_file=False, lambda0=0.1, se=0.5, sim=100000,
thin=10, burn=100, ent_type='LSE', install_ape=False ):
    '''dir_tree      (default: None)   = Represents the directory path to the phylogenetic tree(s) file(s). If no
value is provided when calling the function, the user will be prompted to input the directory path.
    dir_data         (default: None)   = Represents the directory path to the observed states file(s). If no value is
provided when calling the function, the user will be prompted to input the directory path.
    single_tree_file (default: False)  = Boolean value that indicates whether the phylogenetic tree(s) file(s) are
stored in a single file (True) or multiple files (False).
    lambda0          (default: 0.1)    = A constant value used in the acceptance ratio computations.
```

```
    se               (default: 0.5)    = The standard deviation used for the random walk in the Metropolis-Hastings
algorithm.
    sim              (default: 100000) = The number of total iterations in the Markov Chain Monte Carlo (MCMC)
simulation.
    thin             (default: 10)     = The thinning parameter, i.e., the number of iterations to discard between
saved samples.
    burn             (default: 100)    = The number of burn-in iterations to discard at the beginning of the
simulation.
    ent_type         (default: 'LSE')  = A string specifying the type of entropy calculation (options: 'LSE', 'SE',
or any other value for Gini impurity).
    instal_ape       (default: False)  = Boolean value that determines whether to install the 'ape' package or
not.'''

    if dir_tree == None:
        dir_tree = str(input( 'What is the directory path to the phylogenetic tree(s) file(s)?' ))
    if dir_data == None:
        dir_data = str(input( 'What is the directory path to the observed states file(s)?' ))

    if install_ape==True:
        r(
        """
        # Instal Ape
        install.packages('ape')
        """)

    if single_tree_file==False:
        dic_files = set( os.listdir( dir_tree ) )
        dic_delta = set()
        for phylo_tree in dic_files:
            dir_file     = np.loadtxt( dir_tree + '/' + phylo_tree, dtype=str )
            Ancest_Prob  = np.asarray( to_ape( dir_file, dir_data, instal_ape=False ) )
            dic_delta.add( delta(x=Ancest_Prob, lambda0=lambda0, se=se, sim=sim, burn=burn, thin=thin,
ent_type=ent_type) )
        dic_final = dict(zip( dic_files, dic_delta ))

    else:
        dic_final = {}
        arr       = np.loadtxt( dir_tree , dtype=str)
```

```
        indx      = 0
        for phylo_tree in arr:
            Ancest_Prob = np.asarray( to_ape( phylo_tree, dir_data, instal_ape=False ) )
            delta_value = delta(x=Ancest_Prob, lambda0=lambda0, se=se, sim=sim, burn=burn, thin=thin,
ent_type=ent_type)

            phylo_tree  = 'Tree:' + str(indx) + '_' + phylo_tree
            dic_final[phylo_tree] = delta_value

            indx += 1

    return dic_final
```

```
path_tree = './input/2Class/2C_Multiple_Trees.txt'
Delta_Final = multiple_files_Ape( path_tree, path_data, single_tree_file=True)
print( Delta_Final )
print( Delta_Final.values() )

path_tree = './input/2Class/Trees/'
Delta_Final = multiple_files_Ape( path_tree, path_data, single_tree_file=False)
print( Delta_Final )
print( Delta_Final.values() )
```