EWF User Manual

Contents

1 Release notes

1. Initial release version 1.0, 28th December 2022
2. Release version 1.1, 26th March 2024
3. Release version 1.2, 4th June 2024
4. Release version 1.3, 12th September 2024
5. Release version 2.0, 20th October 2025

1.1 Release 2.0

Release 2.0 allows for piecewise constant demographies to be accounted for in the exact simulation routines. For precise details on how this is achieved, please consult the following article, particularly the Supplementary Information.

This latest release also features an overhaul of the previous `EWF` constructor syntax - please see Section 5 below for details on the new syntax. Please note that earlier versions of `EWF` (i.e. any one of the 1.x releases) still operates with the old constructor - it is strongly recommended to use the newest version of `EWF`, however if you still wish to use the older versions, please refer to the respective user manuals.

Finally, please check out the new section below about installing `EWF` within a `venv` environment.

2 Installing `EWF` and dependencies within a `venv` environment

It is strongly recommended to install `EWF` and its dependencies within a python virtual environment (`venv`), as `pip` is now an externally managed environment and thus commands such as `pip install x` won't normally execute in shell.

To this end you need to create a python `venv` by typing in command line

```
$ python3 -m venv NAME
$ source NAME/bin/activate
```

where the first command creates a new `venv` called `NAME` in your current directory, and the second activates the `venv`. Once the `venv` is activated, you can proceed to install all remaining dependencies as well as `EWF` itself as detailed below.

3 Installing Dependencies

`EWF` requires the following components to be run

1. `g++` compiler
2. `boost` library
3. `python` together with `pip`/`pip3`
4. `CMake` together with `Ninja`
5. `pybind11`

The following instructions are meant to illustrate how to install the above, and are functional as of the 6th June 2024. Please note that new releases of the above software might require different installation procedures to the ones below, and thus there is no guarantee that these instructions are up to date nor correct for your specific platform!

3.1 Installing the `g++` compiler

3.1.1 On Windows

If `g++` is not present on your distribution (you can check this by typing `g++ -v` in Command Prompt, which will return all the information regarding the installed `g++` compiler together with its location if `g++` is present, otherwise an error will be returned):

1. Download the latest MinGW (mingw-get-setup.exe) from https://osdn.net/projects/mingw/releases/
2. Follow the installation prompts, note down where MinGW is installed (typically this would be `C:\MinGW`) and in the package selection menu choose the option `mingw32-gcc-g++-bin`. From the "Installation" drop down menu click on "Apply Changes"
3. Once installation is complete, go to System Properties and under the Advanced tab click on "Environment Variables". Click on the Path field and edit it to include the location where the MinGW bin file was installed to (under a typical installation this would be `C:\MinGW\bin`)

To ensure that `g++` was installed as necessary, re-run `g++ -v` within Command Prompt which should now print out the location and further information regarding the compiler you installed.

NB: If you already have Microsoft Visual Studio Code installed on your platform, then we would suggest using the default compiler `MVSC` rather than installing any other compiler, as `pip` defaults to using `MVSC` if the latter is present!

3.1.2 On Mac

If `g++` is not present on your platform (you can check this by running `g++ -dumpversion` within terminal), download the latest version of `Xcode` from the Mac App Store (this might take a while!) and install following the prompts. To check that `g++` was installed as necessary, re-run `g++ -dumpversion` within terminal.

3.1.3 On Linux

If `g++` is absent from your distribution (within terminal type `g++ --version`), then from terminal run `sudo apt update` followed by `sudo apt install build-essential`. To check that installation was successful, re-run `g++ --version` in terminal. This should work on most Linux distributions, but if it does not please search online for an installation procedure for your specific distribution!

3.2 Installing the `boost` library

3.2.1 On Windows and Linux

From `https://www.boost.org/users/download/` download the latest version of `boost` (please note that `EWF` was written using `boost` version 1.84.0 and thus we cannot ensure that certain features are not superseded or deprecated in more up to date version of `boost` - if the latest version produces error on compiling, please download and install version 1.84.0). Once downloaded, extract the files to your desired location on your platform.

3.2.2 On MacOS

Download "HomeBrew" from `brew.sh` (installation instruction provided on webpage), and in terminal run `brew install boost` for the latest version of `boost`.

3.3 Installing `python` and `pip`/`pip3`

On Windows/Mac OS: Download the latest version of `python` from `https://www.python.org/downloads/` and run the installer.

On Linux: Run `sudo apt install python` in terminal.

NB: If you have a recent version of python, then `pip` is probably already present on your platform. If not, you can run `python3 get-pip.py` on Windows/Mac OS, or `sudo apt install python3-pip`.

NB: If you are using `python3`, then you should instead use `pip3` in any command line instructions (i.e. you should run `pip3 install .` rather than `pip install .`)

3.4 Installing `CMake` and `Ninja`

Prior to installing `CMake`, make sure `Ninja` is present on your system - if not you can download the relevant binaries from the Github release (see the corresponding GitHub page and Ninja webpage for more details)

On Linux/Windows: Run `pip install cmake` in terminal/Command Prompt.

On MacOS: Run `brew install cmake` in terminal.

3.5 Installing the `pybind11` module in python

On Linux/Windows: Run `pip install pybind11` in terminal/Command Prompt.

On Mac: Run `brew install pybind11` in terminal.

4 Installing `EWF`

NB: if using Windows, please ensure that your `PATH` variable (part of your environment variables) is pointing at the directories containing `g++`, `boost`, `python`, `pip`, `pybind11` and `CMake`. You can find out where `pybind11` and `CMake` are by running `python3 -v` and entering `import PACKAGE_NAME`.

To install `EWF`, run the following in terminal

```
$ mkdir build
$ cd build
$ cmake ..
$ cmake --build .
$ cd ..
$ pip install .
```

Provided all steps are followed and no errors are thrown, then you can test that `EWF` was run correctly by running the provided test cases found in the `examples` directory, which should print some information to terminal, and create two texts files together with a `png` file.

NB: To run the example script, you will need to have `numpy` and `matplotlib` installed in `python` (simply run `pip install PACKAGE` in terminal).

5 Calling `EWF` in python

To call `EWF` from within python, simply add `import EWF_pybind` at the start of your python script. This allows you to invoke the following functions:

1. `DiffusionRunner` - which allows you to generate draws from the law of a Wright–Fisher diffusion
2. `DiffusionRunnerVector` - which allows you to generate draws from the law of a Wright–Fisher diffusion for a vector of starting points

3. `DiffusionTrajectoryVector` - which allows you to generate draws from the law of a Wright–Fisher diffusion for a vector of sampling times

4. `BridgeDiffusionRunner` - which allows you to generate draws from the law of a Wright–Fisher diffusion bridge

5. `DiffusionDensityCalculator` - which allows you to evaluate the transition density of a Wright–Fisher diffusion

6. `BridgeDiffusionDensityCalculator` - which allows you to evaluate the transition density of a Wright–Fisher diffusion bridge

A detailed explanation of the above functions can be found below, whilst example scripts for running both simulation and pointwise evaluation of the transition densities for both diffusion and bridge diffusion cases can be found in the `example` directory.

We first give an example of how to initialise the `WrightFisher` class:

5.1 The `WrightFisher` class

Initialise a `WrightFisher` class through which both diffusion and bridge simulation functions can be invoked.

Parameters:

- `changepoints` (*array float*) - an array of floats specifying the left endpoint (i.e. the start) of an epoch
- `mutation` (*2D array float*) - an array of arrays, where each inner array contains two floats specifying the mutation rates $\theta_1(t), \theta_2(t)$ during epoch t. The size of the outer array must match the number of entries in `changepoints`
- `non_neutral` (*bool*) - a boolean specifying whether we want a neutral or non-neutral Wright–Fisher diffusion
- `sigma` (*array float*) - an array of floats specifying the coefficient $\sigma(t)$ in (1) during epoch t
- `selectionSetup` (*int*) - an integer specifying whether the users desires genic selection (`selectionSetup = 0`), diploid selection (`selectionSetup = 1`), or polynomial selection (`selectionSetup = 2`). **NB**: Any other value for this parameter will return an error!
- `dominance_parameter` (*float*) - a float specifying the parameter h in the case of diploid selection (see (1) and subsequent discussion for more details).
- `selectionPolynomialDegree` (*int*) - an integer specifying the degree of the polynomial desired by the user.
- `selectionCoefficients` (*array float*) - an array of floats of size `selectionPolynomialDegree+1`, where each float entry specifies the coefficient a_i for the polynomial selection function the users desires to make use of. The entries should be provided in increasing order of power, so that the first entry corresponds to the constant, the second entry correspond to the first order coefficient, etc.

NB: All of the following functions return `void`, and all relevant output is to be found in the corresponding output file (the name of which is provided as input by the user)!

5.2 `DiffusionRunner`

Generate `nSim` draws from the law $\mathbb{WF}^{(x)}_{\sigma,\boldsymbol{\theta}}$ sampled at time `endT`.

NB: This function will assume that the values of $\sigma, \boldsymbol{\theta}$ you want to generate from correspond to those of the epoch to which `startT` belongs! If you want to generate a WF diffusion spanning multiple epochs, please use the function `DiffusionVectorTrajectory`!!

Parameters:

- `nSim` (*int*) - the number of simulated points desired
- `x` (*float*) - the starting point for the diffusion
- `startT` (*float*) - the starting time (in diffusion time units!) for the diffusion
- `endT` (*float*) - the desired sampling time (in diffusion time units!) for the diffusion
- `Absorption` (*bool*) - boolean dictating whether diffusion is conditioned on non-absorption at the boundary. This quantity is only relevant in cases when the mutation rate is zero at a boundary, and in particular is ignored if the mutation rate is strictly positive.
- `Filename` (*string*) - specifies the name of the file where the user desires to save their output
- `diffusion_threshold` (*float*, optional) - threshold below which Gaussian approximations are used. Unless specified, default is `diffusion_threshold` $= 0.1$

Returns: Resulting `nSim` draws are printed to file in `Filename`

5.3 `DiffusionRunnerVector`

Generate `nSim` $\times$ `len(x)` draws sampled at time `endT` from the law $\mathbb{WF}^{(x_i)}_{\sigma,\boldsymbol{\theta}}$ where $\mathtt{x}_i$ is the i^{th} of the vector of starting points `x`.

NB: This function will assume that the values of $\sigma, \boldsymbol{\theta}$ you want to generate from correspond to those of the epoch to which `startT` belongs! If you want to generate a WF diffusion spanning multiple epochs, please use the function `DiffusionVectorTrajectory`!!

Parameters:

- `nSim` (*int*) - the number of simulated points desired
- `x` (*array float*) - an array of floats denoting the desired starting points for the diffusion
- `startT` (*float*) - the starting time (in diffusion time units!) for the diffusion
- `endT` (*float*) - the desired sampling time (in diffusion time units!) for the diffusion
- `Absorption` (*bool*) - boolean dictating whether diffusion is conditioned on non-absorption at the boundary. This quantity is only relevant in cases when the mutation rate is zero at a boundary, and in particular is ignored if the mutation rate is strictly positive.
- `Filename` (*string*) - specifies the name of the file where the user desires to save their output
- `diffusion_threshold` (*float*, optional) - threshold below which Gaussian approximations are used. Unless specified, default is `diffusion_threshold` $= 0.1$

Returns: Resulting `nSim` $\times$ `len(x)` draws are printed to file in `Filename`, with each row i corresponding to the array of samples coming from starting point x_i.

5.4 `DiffusionTrajectoryVector`

Generate `nSim` × `(len(times)-1)` draws from the law $\mathbb{WF}^{(x)}_{\sigma,\boldsymbol{\theta}}$, sampled at each time t_i for t_i the i^{th} entry of `times`.

NB: This function will automatically detect the correct values of $\sigma, \boldsymbol{\theta}$ to use based on the sampling times supplied in `times`. If the consecutive entries in `times` belong to adjacent epochs, the function automatically samples a point at the intermediate time at which the new epoch starts, and uses this to simulate the draw at the desired sample time. Parameters:

- `nSim` (*int*) - the number of simulated points desired
- `x` (*float*) - a float denoting the desired starting point for the diffusion
- `times` (*array float*) - the desired sampling times (in diffusion time units!) for the diffusion. The start time `times[0]` denotes the first sampling time.
- `Absorption` (*bool*) - boolean dictating whether diffusion is conditioned on non-absorption at the boundary. This quantity is only relevant in cases when the mutation rate is zero at a boundary, and in particular is ignored if the mutation rate is strictly positive.
- `Filename` (*string*) - specifies the name of the file where the user desires to save their output
- `diffusion_threshold` (*float*, optional) - threshold below which Gaussian approximations are used. Unless specified, default is `diffusion_threshold` = 0.1

Returns: Resulting `nSim` × `(len(times)-1)` draws are printed to file in `Filename`, with each row i corresponding to a sample path with sampling times given by `times`.

5.5 `BridgeDiffusionRunner`

Generate `nSim` draws from the law $\mathbb{WF}^{(t,x,z)}_{\sigma,\boldsymbol{\theta}}$ sampled at time `sampleT`.

NB: This function automatically detects whether `startT` and `endT` lie in separate epochs, however returns an error if they are not in adjacent epochs! Parameters:

- `nSim` (*int*) - the number of simulated points desired
- `x` (*float*) - the starting point for the diffusion bridge
- `z` (*float*) - the ending point for the diffusion bridge
- `startT` (*float*) - the starting time (in diffusion time units!) for the diffusion bridge
- `endT` (*float*) - the ending time (in diffusion time units!) for the diffusion bridge
- `sampleT` - the sampling time (in diffusion time units!) for the diffusion bridge
- `Absorption` (*bool*) - boolean dictating whether diffusion is conditioned on non-absorption at the boundary. This quantity is only relevant in cases when the mutation rate is zero at a boundary, and in particular is ignored if the mutation rate is strictly positive.
- `Filename` (*string*) - specifies the name of the file where the user desires to save their output
- `diffusion_threshold` (*float*, optional) - time threshold below which Gaussian approximations are used. Unless specified, default is `diffusion_threshold` = 0.1.
- `bridge_threshold` (*float*, optional) - time threshold below which a diffusion approximation and linear interpolation are used. Unless specified, default is `diffusion_threshold` = 0.04.

Returns: Resulting `nSim` draws are printed to file in `Filename`

5.6 `DiffusionDensityCalculator`

Compute the transition density for a *neutral* Wright–Fisher diffusion by appropriately truncating the infinite sums. Running on a non-neutral Wright–Fisher diffusion is not supported and will throw an error!

Parameters:

- `x` (*float*) - the starting point for the diffusion
- `startT` (*float*) - the starting time (in diffusion time units!) for the diffusion
- `endT` (*float*) - the desired sampling time (in diffusion time units!) for the diffusion
- `Absorption` (*bool*) - boolean dictating whether diffusion is conditioned on non-absorption at the boundary. This quantity is only relevant in cases when the mutation rate is zero at a boundary, and in particular is ignored if the mutation rate is strictly positive.
- `Filename` (*string*) - specifies the name of the file where the user desires to save their output
- `diffusion_threshold` (*float*, optional) - threshold below which Gaussian approximations are used. Unless specified, default is `diffusion_threshold` = 0.1

Returns: Resulting transition density evaluations are printed to file in `Filename`

5.7 `BridgeDiffusionDensityCalculator`

Compute the transition density for a *neutral* Wright–Fisher diffusion bridge by appropriately truncating the infinite sums. Running on a non-neutral Wright–Fisher diffusion is not supported and will throw an error!

Parameters:

- `x` (*float*) - the starting point for the diffusion bridge
- `z` (*float*) - the ending point for the diffusion bridge
- `startT` (*float*) - the starting time (in diffusion time units!) for the diffusion bridge
- `endT` (*float*) - the ending time (in diffusion time units!) for the diffusion bridge
- `sampleT` - the sampling time (in diffusion time units!) for the diffusion bridge
- `Absorption` (*bool*) - boolean dictating whether diffusion is conditioned on non-absorption at the boundary. This quantity is only relevant in cases when the mutation rate is zero at a boundary, and in particular is ignored if the mutation rate is strictly positive.
- `Filename` (*string*) - specifies the name of the file where the user desires to save their output
- `diffusion_threshold` (*float*, optional) - time threshold below which Gaussian approximations are used. Unless specified, default is `diffusion_threshold` = 0.1.
- `bridge_threshold` (*float*, optional) - time threshold below which a diffusion approximation and linear interpolation are used. Unless specified, default is `diffusion_threshold` = 0.04.

Returns: Resulting transition density evaluations are printed to file in `Filename`

6 Parameter configuration

Recall that `EWF` returns paths distributed according to the law of a diffusion or diffusion bridge satisfying the following stochastic differential equation

$$dX_t = \frac{1}{2}\left[\sigma(t)X_t(1-X_t)\eta(X_t) - \theta_2(t)X_t + \theta_1(t)(1-X_t)\right]dt + \sqrt{X_t(1-X_t)}dW_t \tag{1}$$

for $t \geq 0$ with $X_0 \in [0,1]$. Note that we assume that

$$\theta_i(t) = 2N_e(t)\mu_i, \qquad \sigma(t) = 2N_e(t)s,$$

for $i = 1, 2$, μ_i the per base-pair per generation mutation rate, s the per generation selection coefficient, a_j the j^{th} selection function coefficient for $j = 0, \ldots, n$, and $N_e(t)$ denoting the effective population size during epoch t. Our implementation allows for any piecewise constant function $N_e(t)$, such that the resulting population rescaled parameters $\sigma(t), \theta_i(t)$ for $i = 1, 2$ are also piecewise constant.

Note further that $\eta(x)$ is a finite degree polynomial in x, i.e. $\eta(x) = \sum_{i=0}^{n} a_i x^i$ for $n \in \mathbb{N}$. Such a formulation allows for a wide class of non-neutral regimes including the case of:

1. Genic selection - here $\eta(x)$ is set to be the constant function 1, such that the contribution from selection to the drift component in (1) becomes $\sigma(t)X_t(1-X_t)$ and $\sigma(t)$ is the only free parameter.

2. Diploid selection - here $\eta(x)$ is typically formulated as $\eta(x) = h + x(1-2h)$ with the free parameter $h \in \mathbb{R}$ determining the relative fitness of the heterozygotes and commonly termed the dominance parameter or degree of dominance. Thus in this case selection contributes a factor of $\sigma(t)X_t(1-X_t)(h + X_t(1-2h))$ to the drift in (1), where we now have 2 free parameters $\sigma(t)$ and h.

In the general case where $\eta(x)$ is a polynomial with degree $n \in \mathbb{N}$, the selection contribution becomes $\sigma(t)X_t(1-X_t)\sum_{i=0}^{n} a_i {X_t}^i$ and we now have $n+1$ free parameters: $\sigma(t)$ and $\{a_i\})_{i=0}^{n}$.

The presence or absence of the mutation parameter $\boldsymbol{\theta}(t) = (\theta_1(t), \theta_2(t))$ dictates whether the boundary points $\{0,1\}$ are absorbing or entrance/reflecting. Furthermore, whenever $\theta_1(t), \theta_2(t) > 0$, the corresponding boundary is reflecting and attainable if $\theta_i(t) < 1$ or entrance (and therefore unattainable) if $\theta_i(t) \geq 1$.

7 Example python scripts

In the examples directory there are two python scripts detailing how to use `EWF` from within python. After setting the desired Wright–Fisher diffusion parameters, the corresponding class is instantiated, and the simulator is invoked to generate 10,000 draws from the law of the diffusion. Subsequently the neutral transition density is estimated through the transition density calculator function, with the resulting estimate for the transition density being plotted on top of the histogram obtained from the previously simulated draws.

We point out that any further function users might wish to expose from the `EWF` C++ codebase to python can be included in the `EWF_python_bindings.cpp` file, but users must re-run `run.sh` for the implemented changes to be reflected in the `EWF` python module.

To run the example scripts, please ensure your `python` has the `numpy` and `matplotlib` packages installed.

8 Bugs, queries, suggestions, comments

If you spot any bugs, or have any queries, suggestions or comments please do not hesitate to get in touch on jaromir.sant@gmail.com!