brw

A small library for generating conditioned branching random walks

Maximilian Hofer

max.jakob.hofer@gmail.com

23rd September 2023

The software in this repository allows generating unbiased realizations of branching random walks conditioned to having the lead particle arrive at a predefined position after a predefined number of time steps. Motivation and details of the underlying algorithm are provided in my master thesis which is also in the repository. The present document is a short guide on how to use the program and library. Complementary to this, an automatically generated documentation of the library can be found separately.

Contents

1 File overview **1**

2 Usage of `main.cpp` **2**

3 Main functionality of the `brw` library **3**
3.1 `class prob_fields` 3
3.2 `template<class RandEng> class condBRW<RandEng>` 5

4 Utility functions **6**
4.1 `class stopwatch` 6
4.2 `class progress_monitor` 7
4.3 `class ppm` 7
4.4 `template<typename IntType> class multinomial_distribution<IntType>` 7

1 File overview

This section lists and relates the files in the repository.

`manual.pdf`	The document you are reading. It contains explanations and usage examples of the program `main.cpp` and the classes in `brw.hpp` / `brw.cpp`.
`refman.pdf`	Automatically generated doxygen documentation of `brw.hpp`.
`Masterarbeit.pdf`	My master's thesis covering in detail the objects and algorithm underlying this library.
`brw.hpp`, `brw.cpp`	The actual `brw` library.
`main.cpp`	A command-line program encapsulating the library functionality.

There are more files in the repository coming from some tests and tries which are not officially part of this library and shall therefore not be documented.

2 Usage of `main.cpp`

The program in `main.cpp` combines all the library functionality in a command-line program for generating conditioned branching random walks (BRWs). It requires the boost program options library. On my setup (Linux Manjaro, boost library installed via `pamac`, `g++ 13.2.1`), the following line compiles the program:

```
$ g++ main.cpp brw.cpp -o main -lboost_program_options
```

Configuration via command line. The programme can be run via

```
$ ./main {--help | --y_values x0 x1 ...} [additional options]
```

where the list of possible options can be printed via the `--help` option:

```
$ ./main --help
```

Aside from `--help`, each option corresponds to an argument/parameter of the generation. For example, `--ntimesteps` (or equivalently `-T`) specifies the number of time steps of the simulation. The help menu also shows the default parameters of each simulation, e.g. `-T [--ntimesteps] arg (=1000)`. The only parameter without default value is `--yvalue`, so for this option values **must be provided manually** (but this can also be done in the config file, see below). For example, the program can be run with the command

```
$ ./main --y_values 100 200 300
```

Configuration via config file (recommended). Probably more convenient is the usage of a configuration file. The latter must be named `config.txt`, and it could look as follows:

```
ncores = 2
lambda = 0.01
ntimesteps = 2000
X-mT = 600
y_values = 200
y_values = 500
y_values = 800
nsites = 1000
savingperiod = 1
nevents = 1
tol = 1e-5
seed = 1
```

Notice that `y_values=` has to be written for every parameter separately. Command line options can also be used in the presence of a configuration file and overwrite the respective setting from `config.txt`.

Output. The program produces three files:

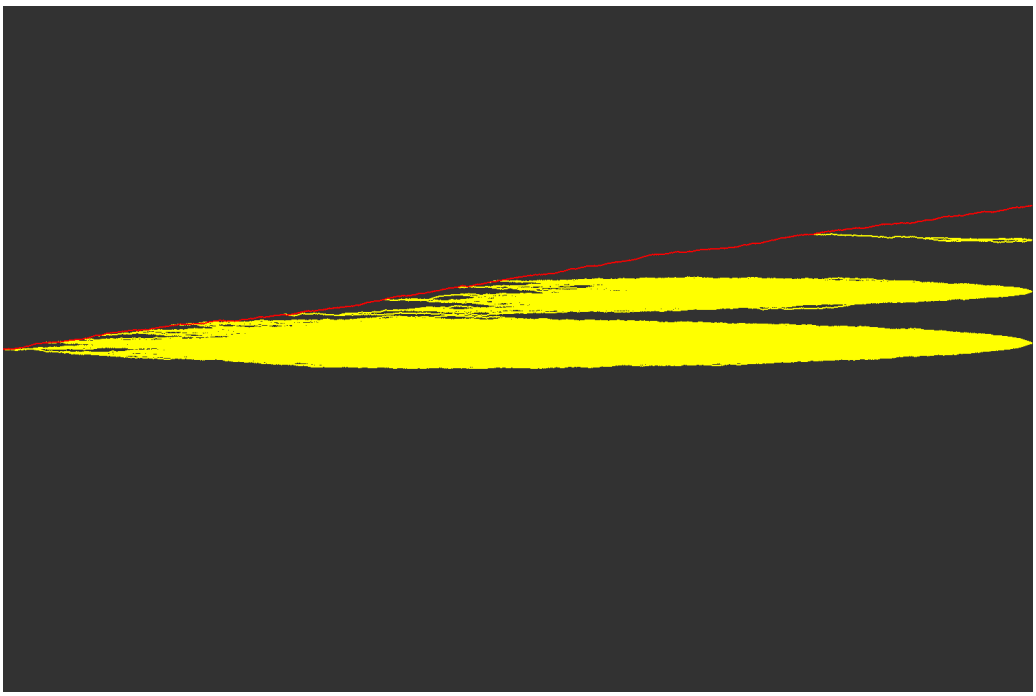

Figure 1: `output.ppm` for the configuration in Section 1

`seeds_s1_to_s2.csv` Here, `s1` and `s2` are numbers determined from the `seed` option. This file contains in the first line the x coordinates of the red (first entry) and yellow particles (remaining entries) as defined by the user through the parameters `-X-mT` and `-y_values`. Afterwards, for each integer number (the seed) in the closed interval [`s1`, `s2`], there is a row containing the particle numbers at the respective coordinates at the final time T (from the parameter `-ntimesteps` or `-T`).

`all.txt` Can usually be ignored. This file contains similar information to the one before, but not just for the user defined sites (provided by `-y_values`). In theory, there shouldn't be particles at other side, but because of rounding errors there are (but only very few). The purpose of this file was just to check that the numbers are really negligible.

`output.ppm` A picture of the process with lowest seed number in `ppm` format. The option *Save as...* of any standard image viewer should be able to transfer the file to more common formats. Using the configuration above, the output is shown in Figure 1.

3 Main functionality of the `brw` library

This section discusses the classes `prob_fields` and `condBRW`. The remaining classes are explained in the next section.

3.1 `class prob_fields`

The `device_size` parameter. This class holds representations of the variables u and y which are explained in my master thesis. Because in the considered model we always start

with a single particle and because particles move on every time step, only x coordinates of the same parity as the corresponding t coordinate are relevant. The internal representation uses therefore $i = \frac{t+x}{2}$ as a coordinate instead of x. For each time t, only a certain range of i coordinates are saved, namely those from and including `lower_scaling_region_bound(t)` up to and excluding `upper_scaling_region_bound(t)`. The lower bound is determined by a tolerance parameter tol (provided during filling, see below) such that u is only stored for i where $u[t,i] < 1 - \text{tol}$. The upper bound is computed by adding the parameter `device_size` (provided during construction) to the lower bound. `device_size` should approximately be chosen of the order of $\sim 3\sqrt{T}$. In general, the higher this parameter, the larger the computation time, but if it is chosen too low, this introduces an effective cut-off to u and y which may compromise the results.

The `saving_period` parameter. Apart from only saving u and y in a certain range of spatial coordinates, they are also only saved for certain time steps. The `saving_period` parameter (provided during construction) specifies that *rows* $u[t,\cdot]$ (and $y[t,\cdot]$) should be saved `saving_period` divides t. These t are called *checkpoints* and the function `fill_checkpoints` allows calculating and storing these up to a time T. For the generation of conditioned BRWs we need $u(X-x, T-t)$ rather than $u(x,t)$. Since the recursion defining u is forward in time, this means that we need to compute u up to time T before starting the simulation. For large T we cannot store all values $t \in [0,T] \cap \mathbb{Z}$, so the idea is to first fill in only the checkpoints and then loop back from $t = T$ to $t = 0$ while only filling the rows between two adjacent checkpoints. This can be achieved with the `fill_between` method. Rows that are not needed any more can be freed with the `erase` method.

Example 1. The following program illustrates the basic usage of `prob_fields`. Without particle generation there is however not much to do with u and y.

```cpp
#include "brw.hpp"
#include <iostream>

using namespace brw;

int main()
{
    double lambda = 0.01;
    int saving_period = 100; //only every 100th row will be saved
    int device_size = 1000;
    int T = 1000;
    std::vector<int> y_values = {100, 200, 300};

    prob_fields uy(int(1000000 * lambda),
                   y_values.begin(), //provide y by iterator
                   y_values.end(),
                   saving_period,
                   device_size);

    uy.fill_checkpoints(T); //fills rows 0, 100, 200,..., 1000
    uy.fill_between(T-saving_period+1, T); //fills rows 901, 902,...,999
    uy.erase(T-10, T); //Frees rows 990, 991,..., 999

    int x = 123;

    // The following is fine: row 900 is filled
    std::cout << "u(t=900, x=" << x << ") = " << uy.u(900, x) << std::endl;

```

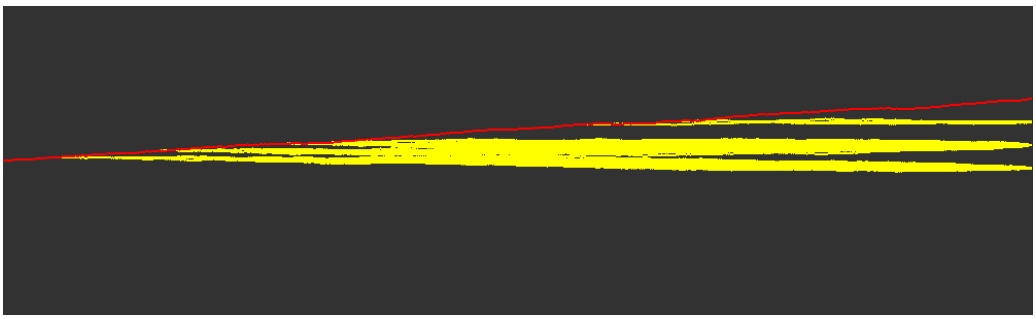

Figure 2: Output of Example 2.

```
    /* The following is error-prone: row 990 is not filled
    * If x is even, an std::out_of_range error is thrown.
    * If x is odd, the function returns 0 without error!
    * (this is no bug, because y is indeed 0 if x+t is odd)
    */
    std::cout << "y(t=990, x=" << x << ") = " << uy.y(990, x) << std::endl;
    return 0;
}
```

3.2 `template<class RandEng> class condBRW<RandEng>`

This class template stores a conditioned BRW and provides functionality to evolve it. The template parameter `RandEng` should be some random engine like for example `std::mt19337`. The class constructor requires a reference to a `prob_fields` object from which it takes the u and y fields (i.p. `prob_fields` can be used by many condBRW objects simultaneously), a random seed and the values of X and T. Afterwards, the state can be evolved using `evolve` which invokes `evolve_one_step` which is the only non-trivial function of the class. The internal state consists of an integer time `t`, a `std::vector<int> red_locations` storing the coordinates of each red particle and a `std::map<int, ptcl_n> n_yellow` storing the number of yellow particles on occupied sites. Here, `ptcl_n` is a type alias of `double` which could also be replaced by `double long` for a higher precision (similarly there is an alias called `real_t` in the `prob_fields` class).

Example 2. The following example produces an image of a conditioned BRW for small T, displayed in Figure 2. The filling of `prob_fields uy` becomes trivial because `saving_period=1`. For large T the filling becomes more sophisticated, as seen in `main.cpp`. The example also demonstrates the use of the `image`, which will be introduced in the next section.

```
#include "brw.hpp"
#include <iostream>

using namespace brw;

int main()
{
    int T = 1000; //target simulation time steps
    int X = 400; //destination of red particle
    //destinations of yellow particles:
    std::vector<int> y_values = {150, 300, 450};

    prob_fields uy(10000, y_values.cbegin(), y_values.cend(), 1, 1000);
    uy.fill_checkpoints(T); //fills all rows up to T

```

```
    condBRW<std::mt19937> state(uy, // prob_fields with information on u,y
                                0,  // random seed
                                X,  // destination of red particle ...
                                T); // ...at time T.

    ppm image(1000,300,50); //see Section 3
    for (int t = 0; t != T; ++t)
    {
        //plotting of yellow particles
        for (auto it = state.cbegin_y(); it != state.cend_y(); ++it)
        {
            unsigned row = (-it->first + T) / (2. * T) * (image.height() - 1);
            unsigned col = t / ((double) T) * (image.width() - 1);
            if (row+1 < image.height())
            {
                image.pixel(row, col, {255, 255, 0});
                image.pixel(row+1, col, {255, 255, 0});
            }

        }
        //plotting of red particles;
        for (auto it = state.cbegin_r(); it != state.cend_r(); ++it)
        {
            unsigned row = (-*it + T) / (2. * T) * (image.height() - 1);
            unsigned col = t / ((double) T) * (image.width() - 1);
            if (row+1 < image.height())
            {
                image.pixel(row, col, {255, 0, 0});
                image.pixel(row+1, col, {255, 0, 0});
            }
        }

        state.evolve(); //evolves the state by one step
    }
    image.save("image_output.ppm");
    return 0;
}
```

4 Utility functions

This section outlines the usage of the utility classes that are part of the `brw` library.

4.1 class stopwatch

It is really straightforward:

```
stopwatch clock; //the construction starts the clock
[do task1]
unsigned long long duration1 = clock.time(); //time that task1 took in
    milliseconds
clock.reset(); //this sets the clock again to zero
[do task2]
std::cout << "Task 2 took " << clock;
//Ouput: "Task 2 took HH h MM min SS s XXX ms
```

4.2 class progress_monitor

The purpose of this class is to keep track of how much progress was achieved at which time in order to estimate how much time will still be needed until completion. The user has to define "progress" himself in terms of number between 0 (no progress) and 1 (completion). At any time, the user can then invoke the function `add_datapoint` to pass the current program. The function will save this data point together with a time stamp in a buffer of constant size n (only the n most recent data points are stored). The remaining time is then estimated from a simple linear extrapolation. Thus, for a reliable prediction can for example be expected if a task is repeated often without a significant change in complexity. Then the class can be used like this:

```
//Construction already starts the internal clock!!
progress_monitor pm(20); //20=number of stored points
for (int i = 0; i != N; ++i)
{
    [task with complexity independent of i]
    pm.add_datapoint(i/N);
    std::cout << "\x1B[2J\x1B[H"  //clears console
                 "Progress: "
              << pm << std::endl; //prints progress and remaining time
}
```

For filling u and y, the estimated time is quite good. For the particle generation it does not work well, because the task of evolving `condBRW` becomes more computationally intensive as the particle number increases over time. Here it would be nice to replace the linear regression by a polynomial of higher order.

4.3 class ppm

This class allows creating, modifying and saving a PPM image (not reading). It is internally not more than a 2D-array of RGB pixels with $256 \times 256 \times 256$ values. The constructor takes a width, a height and optionally a default value `background_brightness` (only greyscale, e.g. 0 for white, 255 for black). The function `pixel` can then be used to read pixel colours (if only row and column are provided) and set pixel colours (if the new colour is provided as a third argument). For a usage example see Example 2 from the previous section.

4.4 template<typename IntType> class multinomial_distribution<IntType>

This class can be used like the standard library random number distributions but for multinomial distributions. Internally, it relies on `std::binomial_distribution`.

Example 3. Consider a game in which n dice are thrown. For each die showing 1,2 or 3 eyes the player shall pay CHF 1, for each die showing 4 eyes the player shall do nothing and for each die showing 5 or 6 eyes the player receives CHF 1. To generate realizations of this game, it is convenient to generate random variables using `multinomial_distribution`:

```
double p123 = 0.5; //prob. that a die shows 1,2 or 3 eyes
double p4 = 1./6;
double p56 = 1./3;
int n = 5; //number of dies to throw
int seed = 0;
std::mt19337 engine(seed);
multinomial_distribution<int> dist(n, {p123, p4, p56});
```

```
8  auto rvec = dist(engine); //draws a random vector
9  std::cout << "n123 = "  << rvec[0]
10           << ", n4 = "  << rvec[1]
11           << ", n56 = " << rvec[2]
12           << std::endl;
13 std::cout << "Winnings: CHF " << rvec[2]-rvec[0] << std::endl;
```