My Project

Generated by Doxygen 1.9.8

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Class Documentation

3.1 branching_random_walk Class Reference

Public Member Functions

- **branching_random_walk** (u_field *u_ptr, unsigned random_seed, int X, int T)
- void **evolve** (long n_steps=1, unsigned long det_thr=1<< 20)
- auto **cbegin** () const
- auto **cend** () const
- auto **cbegin_y** () const
- auto **cend_y** () const
- auto **size** () const
- auto **size_y** () const
- auto **y_at** (int x) const

Protected Types

- using **ptcl_n** = double

Protected Member Functions

- void **evolve_one_step** (unsigned long det_thr)

Protected Attributes

- u_field * **ptr_u**
- std::vector< int > **red_locations**
- std::map< int, ptcl_n > **n_yellow**
- int **t**
- int **X**
- int **T**
- std::mt19937 **engine**

The documentation for this class was generated from the following file:

- cubrw.hpp

3.2 BRW Class Reference

Public Member Functions

- **BRW** (double lambda)
- void **evolve** (long n_steps=1, unsigned long det_thr=1<< 20)
- long **t** () const
- double **n_at** (long x) const
- auto **cbegin** () const
- auto **cend** () const
- const std::vector< long > & **rightmost_track** () const

The documentation for this class was generated from the following file:

- cubrw.hpp

3.3 image Class Reference

Public Types

- using **rgb** = std::array< unsigned char, 3 >

Public Member Functions

- **image** (unsigned width, unsigned height, unsigned char background_brightness=255)
- void **save** (const std::string &filename)
- rgb **pixel** (unsigned row, unsigned col) const
- void **pixel** (unsigned row, unsigned col, rgb colors)
- unsigned **width** () const
- unsigned **height** () const

The documentation for this class was generated from the following file:

- cubrw.hpp

3.4 model Class Reference

Public Types

- using **idx** = unsigned
- using **real_t** = double

Public Member Functions

- **model** (long T, long X, unsigned saving_period, double tol=0.)
- **model** (long T, unsigned saving_period, double tol=0.)
- void **fill_u** ()
- void **fill_logr** ()
- idx **m_t** (idx i)
- real_t **u** (idx i, idx j)
- real_t **w** (idx i, idx j)
- real_t **logw** (idx i, idx j)
- real_t **logq** (idx i, idx j)
- real_t **logr** (idx i, idx j)
- real_t **logp** (idx i, idx j)
- real_t **pplus** (idx i, idx j)
- const real_t & **lambda** () const
- const real_t & **v** () const
- const real_t & **gamma** () const
- const unsigned & **out_period** () const
- long **X** () const
- long **T** () const
- idx **I** () const
- idx **J** () const
- void **print** (std::function< double(model::idx, model::idx)> field, std::ostream &out, const unsigned &digits=3, long window_size=20)

Public Attributes

- bool **approximate_u**
- const unsigned **saving_period**

Static Public Attributes

- static std::vector< idx > **lower_approx_bound**
- static std::vector< idx > **upper_approx_bound**

Friends

- std::ostream & **operator**<< (std::ostream &out, const model &M)
- std::istream & **operator**>> (std::istream &in, const model &M)

The documentation for this class was generated from the following file:

- cubrw.hpp

3.5 multinomial_distribution< IntType > Class Template Reference

Public Member Functions

- **multinomial_distribution** (IntType n_trials, std::initializer_list< double > p_list)
- **multinomial_distribution** (IntType n_trials, std::initializer_list< double > p_list, IntType threshold)
- template<class Generator >
  std::vector< IntType > **operator()** (Generator &engine)

Static Public Attributes

- static unsigned **n_calls_to_binom_dist** = 0

The documentation for this class was generated from the following file:

- cubrw.hpp

3.6 progress_monitor Class Reference

Public Member Functions

- **progress_monitor** (unsigned size=10)
- void **reset** ()
- unsigned **time_remaining** () const
- void **add_datapoint** (double percent_progress)

Public Attributes

- unsigned **size**

Friends

- std::ostream & **operator**<< (std::ostream &out, const progress_monitor &pm)

The documentation for this class was generated from the following file:

- cubrw.hpp

3.7 timer Class Reference

Public Member Functions

- void **reset** ()
- unsigned long long **time** () const

Friends

- std::ostream & **operator**<< (std::ostream &out, const timer &t)

The documentation for this class was generated from the following file:

- cubrw.hpp

3.8 u_field Class Reference

Public Types

- using **sidx** = int
- using **idx** = unsigned int
- using **real_t** = double

Public Member Functions

- **u_field** (unsigned lambda_pμ, unsigned saving_period=1)
- template<class It >
  u_field (unsigned lambda_pμ, It x_y0_begin, It x_y0_end, unsigned saving_period=1)
- **u_field** (unsigned lambda_pμ, std::initializer_list< int > x_y0, unsigned saving_period=1)
- real_t **operator()** (idx t, sidx x) const
- real_t **y** (idx t, sidx x) const
- void **fill_checkpoints** (idx T, unsigned device_size, double tol=1e-5)
- void **fill_between** (idx T1, idx T2, unsigned device_size, double tol=1e-5)
- unsigned long **estimate_memory** (bool rough=true) const
- double **lambda** () const
- unsigned **saving_period** () const
- double **velocity** () const
- double **gamma0** () const
- real_t **avg_R** (idx t) const
- auto **cbegin** (idx t) const
- auto **cend** (idx t) const
- auto **cbegin_y** (idx t) const
- auto **cend_y** (idx t) const
- void **erase** (idx T1, idx T2)
- idx **lower_scaling_region_bound** (idx t) const
- idx **upper_scaling_region_bound** (idx t) const
- void **print** (std::ostream &out, unsigned digits=3, idx window_size=20)

Protected Member Functions

- void **compute_velocity** ()
- void **compute_row** (idx t, double tol=0., bool save=false)
- real_t **u_ti** (idx t, idx i) const
- real_t **y_ti** (idx t, idx i) const

Protected Attributes

- double **_lambda**
- unsigned long **_lambda_pμ**
- bool **_compute_y**
- unsigned **_saving_period**
- double **_velocity**
- double **_gamma0**
- std::map< idx, std::pair< idx, HOST_VEC > > **u_map**
- DEV_VEC **prev_u**
- DEV_VEC **next_u**
- std::map< idx, HOST_VEC > **y_map**

- HOST_VEC **y0**
- DEV_VEC **prev_y**
- DEV_VEC **next_y**
- std::map< idx, HOST_VEC > **prss_map**
- std::map< idx, HOST_VEC > **plss_map**
- std::map< idx, HOST_VEC > **prs2s_map**
- std::map< idx, HOST_VEC > **pls2s_map**
- idx **lsrb**

Friends

- std::ostream & **operator**<< (std::ostream &out, const u_field &u)
- std::istream & **operator**>> (std::istream &in, u_field &u)

The documentation for this class was generated from the following file:

- cubrw.hpp

3.9 u_recursion Class Reference

Public Member Functions

- **u_recursion** (double lambda)
- __host__ __device__ void **operator()** (u_field::real_t &u, const u_field::real_t &uti, const u_field::real_t &uti1) const
- __host__ __device__ void **operator()** (u_field::real_t &u, const u_field::real_t &uti, const u_field::real_t &uti1, u_field::real_t &y, const u_field::real_t &yti, const u_field::real_t &yti1) const

The documentation for this class was generated from the following file:

- cubrw.hpp

Chapter 4

File Documentation

4.1 cubrw.hpp

```
00001 #ifndef cuBRW_H
00002 #define cuBRW_H
00003
00004 #include <map>
00005 #include <random>
00006 #include <vector>
00007 #include <deque>
00008 #include <numeric>
00009 #include <functional>
00010 #include <chrono>
00011 #include <fstream>
00012 #include <initializer_list>
00013
00014 // #define GPU_SUPPORT
00015
00016 #ifdef GPU_SUPPORT
00017     #include "thrust/device_vector.h"
00018     #include "thrust/host_vector.h"
00019     #define DEV_VEC thrust::device_vector<real_t>
00020     #define HOST_VEC thrust::host_vector<real_t>
00021     #define COPY thrust::copy
00022     #define REDUCE thrust::reduce
00023     #define SWAP thrust::swap
00024     #define PLUS thrust::plus<real_t>
00025 #else
00026     #define DEV_VEC std::vector<real_t>
00027     #define HOST_VEC std::vector<real_t>
00028     #define COPY std::copy
00029     #define REDUCE std::reduce
00030     #define SWAP std::swap
00031     #define PLUS std::plus<real_t>
00032 #endif
00033
00034 class BRW
00035 {
00036     private:
00037         typedef unsigned long particle_number;
00038         double lam;
00039         std::map<long, particle_number> n;
00040         std::vector<long> rightmost;
00041         long time = 0;
00042         static std::mt19937 engine;
00043         static const unsigned seed = 5;
00044         void evolve_one_step(unsigned long det_thr);
00045     public:
00046         BRW(double lambda) : lam(lambda) { n[0] = 1; }
00047         void evolve(long n_steps = 1, unsigned long det_thr = 1«20);
00048         inline long t() const { return time; };
00049         inline double n_at(long x) const;
00050         inline auto cbegin() const { return n.cbegin(); }
00051         inline auto cend() const { return n.cend(); }
00052         inline const std::vector<long>& rightmost_track() const { return rightmost; }
00053 };
00054
00055 inline double BRW::n_at(long x) const
00056 {
00057     auto nx_it = n.find(x);
00058     if (nx_it == n.end()) return 0.;
```

```
00059     return nx_it->second;
00060 }
00061
00062 class progress_monitor
00063 {
00064     private:
00065         std::chrono::_V2::steady_clock::time_point starting_time;
00066         std::deque<double> progress; //values between 0 and 1
00067         std::deque<unsigned long> times; //in milliseconds and counted since start
00068     public:
00069         unsigned size;
00070         /*Helper class to store the progress and estimate the time that is left through linear
      extrapolation.
00071
00072         \texttt{size} ... Number of data points that are saved and used for the linear
      extrapolation.*/
00073         progress_monitor(unsigned size = 10) : size(size) { reset(); }
00074         //Deletes all data points saved so far and resets the internal clock. After calling this,
      \texttt{*this} is in the same state as a freshly constructed \texttt{progress_monitor}
00075         void reset();
00076         //Estimated remaining time in milliseconds.
00077         unsigned time_remaining() const;
00078         //Takes a floating point number in [0,1] indicating the progress of some task. Automatically
      saves the time relative to the last reset.
00079         void add_datapoint(double percent_progress);
00080         //Prints the progress and estimated the time in a formatted way to \texttt{out}.
00081         friend std::ostream& operator<<(std::ostream& out, const progress_monitor& pm);
00082 };
00083
00084 class timer
00085 {
00086     private:
00087         std::chrono::_V2::steady_clock::time_point starting_time;
00088     public:
00089         //A simple clock starting to count upon construction.
00090         timer() { reset(); }
00091         //Restarts the clock at zero.
00092         void reset() {starting_time = std::chrono::steady_clock::now(); }
00093         //Returns passed time since last reset or construction in milliseconds.
00094         unsigned long long time() const { return
      std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::steady_clock::now()-starting_time).count();
      }
00095         //Prints time since last reset.
00096         friend std::ostream& operator<<(std::ostream& out, const timer& t);
00097 };
00098
00099 class image
00100 {
00101     private:
00102         unsigned w, h;
00103         std::vector<unsigned char> pixels;
00104     public:
00105         using rgb = std::array<unsigned char, 3>;
00106         //Creates a ppm image of format P6 with values 0-255 for each color rgb.
00107         image(unsigned width, unsigned height, unsigned char background_brightness = 255) : w(width),
      h(height), pixels(3*width*height, background_brightness) {}
00108         //Saves the file in a binary format.
00109         void save(const std::string& filename) {
00110             std::ofstream file(filename, std::ios::binary);
00111             file << "P6\n" << w << " " << h << "\n255\n";
00112             file.write(reinterpret_cast<const char *>(&pixels[0]), pixels.size());
00113         }
00114         //Reads the color values of a given pixel.
00115         rgb pixel(unsigned row, unsigned col) const {
00116             std::size_t start = 3*row*w + 3*col;
00117             return { pixels.at(start), pixels.at(start+1), pixels.at(start+2) };
00118         }
00119         /*Sets the color values of a given pixel.
00120         \texttt{rgb} is an alias for \texttt{std::array<unsigned char, 3>}
00121         */
00122         void pixel(unsigned row, unsigned col, rgb colors) {
00123             std::size_t start = 3*row*w + 3*col;
00124             pixels.at(start) = std::get<0>(colors);
00125             pixels.at(start+1) = std::get<1>(colors);
00126             pixels.at(start+2) = std::get<2>(colors);
00127         }
00128         unsigned width() const { return w; }
00129         unsigned height() const { return h; }
00130 };
00131
00132 class u_field
00133 {
00134     public:
00135         using sidx = int; //signed index type
00136         using idx = unsigned int;
00137         using real_t = double;
00138         //This class holds the values of u in the scaling region for a given value of lambda. The
```

```
saving_period is the distance between two checkpoints.
00139         u_field(unsigned lambda_pu, unsigned saving_period = 1);
00140         /*This class holds the values of u in the scaling region for a given value of lambda. The
saving_period is the distance between two checkpoints.
00141         Using this constructor one also has to pass by iterator the x-values of the sites at which the
function y(0,x) is non-zero.*/
00142         template<class It>
00143         u_field(unsigned lambda_pu, It x_y0_begin, It x_y0_end, unsigned saving_period = 1) :
u_field(lambda_pu, saving_period)
00144         {
00145             _compute_y = true;
00146             if (x_y0_end-x_y0_begin == 1) throw std::runtime_error("The initializer_list<int> x_y0
must be non-empty if it is provided.");
00147             int xmin = *std::min_element(x_y0_begin, x_y0_end);
00148             int xmax = *std::max_element(x_y0_begin, x_y0_end);
00149             if (xmin <= 0) throw std::domain_error("The sites x at which y(0,x) != 0 must all be
strictly positive.");
00150             for (auto it = x_y0_begin; it != x_y0_end; ++it)
00151             {
00152                 if (*it % 2 == 0)
00153                 {
00154                     y0.resize(*it/2 + 1);
00155                     y0[*it/2] = 1;
00156                 }
00157             }
00158         }
00159         /*This class holds the values of u in the scaling region for a given value of lambda. The
saving_period is the distance between two checkpoints.
00160         Using this constructor one also has to pass the x-values of the sites at which the function
y(0,x) is non-zero.*/
00161         u_field(unsigned lambda_pu, std::initializer_list<int> x_y0, unsigned saving_period = 1) :
u_field(lambda_pu, x_y0.begin(), x_y0.end(), saving_period) {}
00162         //Returns u(x,t) assuming step sizes dx == dt == 1
00163         real_t operator()(idx t, sidx x) const;
00164         //Returns y(x,t) assuming step sizes dx == dt == 1
00165         real_t y(idx t, sidx x) const;
00166         /*Fills in all checkpoints up to and including time T starting from the last row that is
saved. If tol = 0., all non-zero values of u are computed.
00167         device_size is the number of entries that are saved.*/
00168         void fill_checkpoints(idx T, unsigned device_size, double tol = 1e-5);
00169         /*
00170         Fills all rows from inclusively T1 to exclusively T2. Existing rows are overwritten.
00171         If T1 != 0, row T1-1 must be filled beforehand.
00172         device_size is the number of entries that are saved.
00173         */
00174         void fill_between(idx T1, idx T2, unsigned device_size, double tol = 1e-5);
00175         /*Writes the object to "out". The first row contains lambda and the saving_period.
00176         The second and third rows contain all values of lower_approx_bound and upper_approx_bound.
00177         The remaining lines start with the time index and afterwards the entries of u at this time
index.
00178         Only the values that are saved are outputted.*/
00179         friend std::ostream& operator<<(std::ostream& out, const u_field& u);
00180         //Reads in the object from "in", following the format outputted by operator<<. this->u_map gets
cleared and overwritten.
00181         friend std::istream& operator>>(std::istream& in, u_field& u);
00182         //Output the approximate memory used to save u. If rough == true, it is assumed that all rows
contain the same number of elements as the latest row.
00183         unsigned long estimate_memory(bool rough = true) const;
00184         double lambda() const { return _lambda; }
00185         unsigned saving_period() const { return _saving_period; }
00186         double velocity() const { return _velocity; }
00187         double gamma0() const { return _gamma0; }
00188         real_t avg_R (idx t) const { return 2*REDUCE(u_map.at(t).second.crbegin(),
u_map.at(t).second.crend(), (real_t) lower_scaling_region_bound(t), PLUS()) - (real_t) t; }
00189         auto cbegin(idx t) const { return u_map.at(t).second.cbegin(); }
00190         auto cend(idx t) const { return u_map.at(t).second.cend(); }
00191         auto cbegin_y(idx t) const { return y_map.at(t).cbegin(); }
00192         auto cend_y(idx t) const { return y_map.at(t).cend(); }
00193         //Erases the memory of the rows from inclusively T1 to exclusively T2.
00194         void erase(idx T1, idx T2) {
00195             for (unsigned t = T1; t != T2; ++t)
00196             {
00197                 u_map.erase(t);
00198                 y_map.erase(t);
00199                 prss_map.erase(t);
00200                 plss_map.erase(t);
00201                 prs2s_map.erase(t);
00202                 pls2s_map.erase(t);
00203             }
00204         }
00205         idx lower_scaling_region_bound(idx t) const { return u_map.at(t).first; }
00206         idx upper_scaling_region_bound(idx t) const { return lower_scaling_region_bound(t) +
u_map.at(t).second.size(); }
00207         void print(std::ostream& out, unsigned digits = 3, idx window_size = 20);
00208     protected:
00209         double _lambda;
00210         unsigned long _lambda_pu;
```

```
00211         bool _compute_y;
00212         unsigned _saving_period;
00213         double _velocity;
00214         double _gamma0;
00215         std::map<idx, std::pair<idx, HOST_VEC» u_map;
00216         DEV_VEC prev_u;
00217         DEV_VEC next_u;
00218         std::map<idx, HOST_VEC> y_map;
00219         HOST_VEC y0;
00220         DEV_VEC prev_y;
00221         DEV_VEC next_y;
00222         std::map<idx, HOST_VEC> prss_map;
00223         std::map<idx, HOST_VEC> plss_map;
00224         std::map<idx, HOST_VEC> prs2s_map;
00225         std::map<idx, HOST_VEC> pls2s_map;
00226         idx lsrb;
00227         void compute_velocity();
00228         /*Compute row t assuming that row t-1 has been computed (i.p. call illegal for t==0).
00229         The row is only saved in u_map if save==true. Otherwise it is only temporarily saved until the
      next call to compute_row.
00230         Values for which 1-u < tol are not saved.*/
00231         void compute_row(idx t, double tol = 0., bool save = false);
00232         //Returns u given (t,i)-coordinates. Only rows that have been filled beforehand can be
      accessed.
00233         real_t u_ti(idx t, idx i) const;
00234         //Returns y given (t,i)-coordinates. Only rows that have been filled beforehand can be
      accessed.
00235         real_t y_ti(idx t, idx i) const;
00236 };
00237
00238 class u_recursion
00239 {
00240         u_field::real_t lambda;
00241     public:
00242         u_recursion(double lambda) : lambda(lambda) {}
00243
00244         __host__ __device__
00245         void operator()(u_field::real_t& u, const u_field::real_t& uti, const u_field::real_t& uti1)
      const {
00246             u_field::real_t result = (1. + lambda) / 2. * (uti1 + uti) - lambda / 2. * (uti1 * uti1 +
      uti * uti);
00247             if (result < 1e-320) u = 0.;
00248             else u = result;
00249         };
00250
00251         __host__ __device__
00252         void operator()(u_field::real_t& u, const u_field::real_t& uti, const u_field::real_t& uti1,
00253                         u_field::real_t& y, const u_field::real_t& yti, const u_field::real_t& yti1)
      const {
00254             u_field::real_t u_result = (1. + lambda) / 2. * (uti1 + uti) - lambda / 2. * (uti1 * uti1
      + uti * uti);
00255             u_field::real_t y_result = ((1. + lambda) / 2. - lambda * uti) * yti + ((1. + lambda) / 2.
      - lambda * uti1) * yti1
00256                                        - lambda / 2. * (yti * yti + yti1 * yti1);
00257             if (u_result < 1e-320) u = 0.;
00258             else u = u_result;
00259             if (y_result < 1e-320) y = 0.;
00260             else y = y_result;
00261         };
00262 };
00263
00264 /*class prob_recursion
00265 {
00266         u_field::real_t lambda;
00267     public:
00268         u_recursion(double lambda) : lambda(lambda) {}
00269
00270         __host__ __device__
00271         void operator()(u_field::real_t& u, const u_field::real_t& uti, const u_field::real_t& uti1)
      const {
00272             u_field::real_t result = (1. + lambda) / 2. * (uti1 + uti) - lambda / 2. * (uti1 * uti1 +
      uti * uti);
00273             if (result < 1e-320) u = 0.;
00274             else u = result;
00275         };
00276
00277         __host__ __device__
00278         void operator()(u_field::real_t& u, const u_field::real_t& uti, const u_field::real_t& uti1,
00279                         u_field::real_t& y, const u_field::real_t& yti, const u_field::real_t& yti1)
      const {
00280             u_field::real_t u_result = (1. + lambda) / 2. * (uti1 + uti) - lambda / 2. * (uti1 * uti1
      + uti * uti);
00281             u_field::real_t y_result = ((1. + lambda) / 2. - lambda * uti) * yti + ((1. + lambda) / 2.
      - lambda * uti1) * yti1
00282                                        - lambda / 2. * (yti * yti + yti1 * yti1);
00283             if (u_result < 1e-320) u = 0.;
00284             else u = u_result;
```

```
00285             if (y_result < 1e-320) y = 0.;
00286             else y = y_result;
00287         };
00288 };*/
00289
00290 template<typename IntType>
00291 class multinomial_distribution
00292 {
00293     private:
00294         IntType N;
00295         std::vector<double> p; //contains the effective probabilities, p_list.size()-1 in number
00296         IntType threshold;
00297         bool approximate;
00298     public:
00299         /*Defines a multinomial distribution using the probabilities in p_list which need to sum to
    one.
00300         The last value is not used but deduced from this assumption.
00301         */
00302         multinomial_distribution(IntType n_trials, std::initializer_list<double> p_list) : N(n_trials)
    {
00303             p.push_back(*(p_list.begin()));
00304             double normalization = 1. - p.back();
00305             for (auto p_it = p_list.begin()+1; p_it != p_list.end()-1; ++p_it)
00306             {
00307                 p.push_back(*p_it / normalization);
00308                 normalization -= *p_it;
00309             }
00310         }
00311         /*Defines a multinomial distribution using the probabilities in p_list which need to sum to
    one.
00312         The last value is not used but deduced from this assumption.
00313
00314         Binomially distributed (n,p) random variables occuring in intermediate steps with n*p >
    threshold are just taken to be [n*p].
00315         */
00316         multinomial_distribution(IntType n_trials, std::initializer_list<double> p_list, IntType
    threshold) : multinomial_distribution(n_trials, p_list) {
00317             this->threshold = threshold;
00318             approximate = true;
00319         }
00320
00321         static unsigned n_calls_to_binom_dist;
00322
00323         template<class Generator>
00324         std::vector<IntType> operator()(Generator& engine);
00325 };
00326
00327 template<typename IntType>
00328 unsigned multinomial_distribution<IntType>::n_calls_to_binom_dist = 0;
00329
00330 template<typename IntType>
00331 template<class Generator>
00332 std::vector<IntType> multinomial_distribution<IntType>::operator()(Generator& engine)
00333 {
00334     std::vector<IntType> result;
00335     IntType remaining = N;
00336     IntType estimated;
00337     for (auto it = p.begin(); it != p.end(); ++it)
00338     {
00339         if (approximate && (estimated = remaining * *it) > threshold) result.push_back(estimated);
00340         else
00341         {
00342             ++n_calls_to_binom_dist;
00343             result.push_back(std::binomial_distribution<IntType>(remaining, *it)(engine));
00344         }
00345         remaining -= result.back();
00346         if (remaining == 0) break;
00347     }
00348     result.push_back(remaining);
00349     result.resize(p.size()+1);
00350     return result;
00351 }
00352
00353 class branching_random_walk
00354 {
00355     protected:
00356         using ptcl_n = double;
00357         u_field* ptr_u;
00358         std::vector<int> red_locations;
00359         std::map<int, ptcl_n> n_yellow;
00360         int t;
00361         int X, T;
00362         std::mt19937 engine;
00363         void evolve_one_step(unsigned long det_thr);
00364     public:
00365         branching_random_walk(u_field* u_ptr, unsigned random_seed, int X, int T) : ptr_u(u_ptr),
    engine(random_seed), T(T), X(X), t(0), red_locations{0} { }
```

```
00366         void evolve(long n_steps = 1, unsigned long det_thr = 1«20);
00367         auto cbegin() const { return red_locations.cbegin(); }
00368         auto cend() const { return red_locations.cend(); }
00369         auto cbegin_y() const { return n_yellow.cbegin(); }
00370         auto cend_y() const { return n_yellow.cend(); }
00371         auto size() const { return red_locations.size(); }
00372         auto size_y() const { return n_yellow.size(); }
00373         auto y_at(int x) const { if (std::map<int, branching_random_walk::ptcl_n>::const_iterator
      search = n_yellow.find(x); search != n_yellow.end()) return search->second; else return
      (branching_random_walk::ptcl_n) 0; }
00374 };
00375
00376 #ifdef GPU_SUPPORT
00377 class red_orange_brw
00378 {
00379     protected:
00380         using ptcl_n = double;
00381         u_field* ptr_u;
00382         std::vector<int> red_locations;
00383         thrust::device_vector<ptcl_n> curr_orange;
00384         thrust::device_vector<ptcl_n> next_orange;
00385         size_t orange_size;
00386         int t;
00387         int X, T, Delta;
00388         std::mt19937 engine;
00389         void evolve_one_step(unsigned long det_thr);
00390     public:
00391         red_orange_brw(u_field* u_ptr, unsigned random_seed, int X, int T, int Delta) : ptr_u(u_ptr),
      engine(random_seed), T(T), X(X), t(0), Delta(Delta), red_locations{0} {
00392             orange_size = u_ptr->cend(0) - u_ptr->cbegin(0);
00393             curr_orange = thrust::device_vector<ptcl_n>(orange_size, 0);
00394             next_orange = thrust::device_vector<ptcl_n>(orange_size, 0);
00395         }
00396         void evolve(long n_steps = 1, unsigned long det_thr = 1«20) { for (unsigned i = 0; i !=
      n_steps; ++i) { evolve_one_step(det_thr); ++t; } }
00397         auto cbegin_r() const { return red_locations.cbegin(); }
00398         auto cend_r() const { return red_locations.cend(); }
00399         auto cbegin_o() const { return curr_orange.cbegin(); }
00400         auto cend_o() const { return curr_orange.cend(); }
00401         auto size_r() const { return red_locations.size(); }
00402         auto size_o() const { return orange_size; }
00403 };
00404 #endif
00405
00406 class model
00407 {
00408     public:
00409         using idx = unsigned;
00410         using real_t = double;
00411     private:
00412         const unsigned lambda_pm = 10;
00413         const double approx_tol;
00414         const real_t _lambda;
00415         const real_t logw_plus;
00416         const real_t logw_minus;
00417         inline static real_t velocity;
00418         inline static real_t gamma0;
00419         unsigned output_period;
00420         inline static std::map<idx, std::vector<real_t» u_map;
00421         inline static std::map<std::pair<idx, idx>, real_t> logq_map;
00422         std::map<idx, std::deque<real_t» logr_map;
00423         void compute_velocity();
00424         #ifdef PRINT_W_MEMORY
00425         void print_w_map();
00426         bool w_changed = false;
00427         #endif
00428         long _X, _T;
00429         idx _I, _J;
00430         progress_monitor pm_u;
00431         progress_monitor pm_r;
00432         void fill_u_row(idx i);
00433         void fill_logr_row(idx i);
00434         long u_size();
00435         long r_size();
00436     public:
00437         inline static std::vector<idx> lower_approx_bound;
00438         inline static std::vector<idx> upper_approx_bound;
00439         bool approximate_u;
00440         const unsigned saving_period;
00441         model(long T, long X, unsigned saving_period, double tol = 0.);
00442         model(long T, unsigned saving_period, double tol = 0.);
00443         void fill_u();
00444         void fill_logr();
00445         idx m_t(idx i); // gives the index j such that [i,j] corresponds to the coordinates (t,m_t)
00446         real_t u(idx i, idx j);
00447         real_t w(idx i, idx j);
00448         real_t logw(idx i, idx j);
```

```
00449         real_t logq(idx i, idx j);
00450         real_t logr(idx i, idx j);
00451         real_t logp(idx i, idx j);
00452         real_t pplus(idx i, idx j);
00453         inline const real_t& lambda() const { return _lambda; }
00454         inline const real_t& v() const { return velocity; }
00455         inline const real_t& gamma() const { return gamma0; }
00456         inline const unsigned& out_period() const { return output_period; }
00457         inline long X() const { return _X; }
00458         inline long T() const { return _T; }
00459         inline idx I() const { return _I; }
00460         inline idx J() const { return _J; }
00461         friend std::ostream& operator«(std::ostream& out, const model& M);
00462         friend std::istream& operator»(std::istream& in, const model& M);
00463         void print(std::function<double(model::idx, model::idx)> field, std::ostream& out, const
      unsigned& digits = 3, long window_size = 20);
00464 };
00465
00466 std::string d_to_str(double x, int precision = 2);
00467
00468 //A helper function returning the parameters k,d such that k*x+d is the best linear fit of y.
00469 template<typename T>
00470 std::pair<T, T> linear_fit(const std::vector<T>& x, const std::vector<T>& y)
00471 {
00472     auto x_it = x.begin();
00473     auto y_it = y.begin();
00474     T sum_xy = 0;
00475     T sum_xx = 0;
00476     T sum_x = 0;
00477     T sum_y = 0;
00478     unsigned n = x.size();
00479     while (y_it != y.end() && x_it != x.end())
00480     {
00481         sum_xy += *x_it * *y_it;
00482         sum_xx += *x_it * *x_it;
00483         sum_x += *x_it;
00484         sum_y += *y_it;
00485         ++y_it;
00486         ++x_it;
00487     }
00488     T k = (sum_xy - sum_x * sum_y / n) / (sum_xx - sum_x * sum_x / n);
00489     T d = (sum_y - k * sum_x) / n;
00490     return {k,d};
00491 }
00492
00493 #endif
```

Index