International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Volume XXXIX-B3, 2012
XXII ISPRS Congress, 25 August – 01 September 2012, Melbourne, Australia

LIBRJMCMC: AN OPEN-SOURCE GENERIC C++ LIBRARY FOR STOCHASTIC OPTIMIZATION

Mathieu Brédif[1] **and Olivier Tournaire**[23]

[1]IGN, MATIS, 73 avenue de Paris, 94160 Saint-Mandé, France ; Université Paris-Est
[2]CSTB, 84 avenue Jean Jaurès, 77447 Marne la Vallée, France
[3]Imagine/LIGM, Université Paris-Est, 6 avenue Blaise Pascal, 77455 Marne la Vallée, France

Commission III/3

KEY WORDS: Stochastic Optimization, RJ-MCMC, Simulated Annealing, Generic C++ Library, Open-Source

ABSTRACT:

The `librjmcmc` is an open source C++ library that solves optimization problems using a stochastic framework. The library is primarily intended for but not limited to research purposes in computer vision, photogrammetry and remote sensing, as it has initially been developed in the context of extracting building footprints from digital elevation models using a marked point process of rectangles. It has been designed to be both highly modular and extensible, and have computational times comparable to a code specifically designed for a particular application, thanks to the powerful paradigms of metaprogramming and generic programming. The proposed stochastic optimization is built on the coupling of a stochastic Reversible-Jump Markov Chain Monte Carlo (RJMCMC) sampler and a simulated annealing relaxation. This framework allows, with theoretical guarantees, the optimization of an unrestricted objective function without requiring any initial solution.
The modularity of our library allows the processing of any kind of input data, whether they are 1D signals (e.g. LiDAR or SAR waveforms), 2D images, 3D point clouds... The library user has just to define a few modules describing its domain specific context: the encoding of a configuration (e.g. its object type in a marked point process context), reversible jump kernels (e.g. birth, death, modifications...), the optimized energies (e.g. data and regularization terms) and the probabilized search space given by the reference process. Similar to this extensibility in the application domain, concepts are clearly and orthogonally separated such that it is straightforward to customize the convergence test, the temperature schedule, or to add visitors enabling visual feedback during the optimization. The library offers dedicated modules for marked point processes, allowing the user to optimize a Maximum A Posteriori (MAP) criterion with an image data term energy on a marked point process of rectangles.

1 INTRODUCTION

Optimization of an objective function over a given search space is a very important and wide research topic. Optimization problems come however in very different flavors: the objective function may or may not exhibit nice properties like convexity or the search space may be a simple compact of $\mathbb{R}^n$, a (possibly infinite) set of discrete values, or even a combination of both: a (possibly infinite) union of spaces embeddable in $\mathbb{R}^n$. This article proposes an implementation of a stochastic optimization framework for optimizing arbitrary objective functions over the latter and more complex search spaces. Given the difficulty of these problems that mix combinatorial and variational aspects and the genericity of the proposed library, it does not provide a one-size-fits-all solution. Instead, it implements a theoretically sound stochastic framework that is easily and highly customizable for rapid prototyping and tuning of the optimization process. The `librjmcmc` is a generic C++ library (Abrahams and Gurtovoy, 2004) designed to be both heavily optimized and highly modular.

In this paper, we first briefly remind the mathematical foundations involved in the chosen stochastic framework (section 2). We then present our implementation choices to build a computationally efficient and generic library (section 3). Before concluding, we demonstrate sample uses of the library, including the discussion of the implementation of the building footprint extraction from satellite imagery proposed in (Tournaire et al., 2010) (section 4).

2 STOCHASTIC OPTIMIZATION

`librjmcmc` 's optimization is performed using a coupling of a simulated annealing relaxation (Salamon et al., 2002) with a Reversible Jump Markov Chain Monte Carlo (RJMCMC) sampler (Hastings, 1970; Green, 1995). This framework enables the stochastic minimization of a large class of energies over complex search spaces, only requiring the evaluation of the objective function, rather than *e.g.* derivatives or convexity properties (Descombes, 2011).

2.1 Reversible-Jump Markov Chain Monte Carlo

A Markov Chain Monte Carlo (MCMC) sampler provides a series of samples according to a given unnormalized probability distribution function. RJMCMC is an extension of MCMC that allows the configuration space Ω to be the union of spaces of varying dimensions $\Omega = \bigcup_n \Omega_n$. Algorithm 1 (Green, 1995) consists in repeating stochastic proposition and acceptance steps to build a series of configurations (*i.e.* elements of the search space Ω) which stationary distribution is the desired target distribution π. Note that after a sufficient number of iterations, the running configuration X_t is independent of the initial configuration X_0.

A well-known property of interest is that the stationary distribution π is not only never sampled directly (as it is the purpose of the algorithm) but also only evaluated up to a multiplicative constant, as it only appear in the term $\frac{\pi(X'_t)}{\pi(X_t)}$ of the Green ratio. This thus enables the sampling of a probability π defined as the normalization of a non-negative integrable function f: $\pi(x) = \frac{f(x)}{\int_\Omega f}$.

Algorithm 1: RJMCMC sampler: Metropolis-Hastings-Green

• Initialize the configuration X_0 such that $\pi(X_0) \neq 0$
repeat
  • Sample $i \sim q(\cdot|X_t)$; `// Select a reversible kernel` Q_i
  • Sample $X'_t \sim Q_i(\cdot|X_t)$; `// Sample the kernel` Q_i
  • $R \leftarrow \frac{\pi(X'_t)}{\pi(X_t)} \frac{Q_i(X'_t|X_t)}{Q_i(X_t|X'_t)}$; `// Green ratio`
  • $\alpha \leftarrow \min(1, R)$; `// Acceptance rate`
  • With probability $\begin{cases} \alpha & , X_{t+1} \leftarrow X'_t \\ (1-\alpha) & , X_{t+1} \leftarrow X_t \end{cases}$
until *convergence* ;

Algorithm 1 requires a set of reversible kernels (Q_i) associated with normalized probabilities $q(i|X_t)$. Each reversible kernel Q_i models a simple stochastic modification of the current configuration, based only on the current configuration X_t: *i.e.* $Q_i(\cdot|X_t)$ defines a probability distribution over Ω. For the correctness of the algorithm πQ_i must have continuity properties and Q_i must be reversible: $Q_i(X|Y) > 0 \Rightarrow Q_i(Y|X) > 0$ (Descombes, 2011). We will consider reversible kernels defined using the following elements (Green, 1995):

- A probability $p_i(n'|n)$ of applying the reversible kernel, with n, n' such that $X_t \in \Omega_n$, $X'_t \in \Omega_{n'}$.
- A pair of forward and backward views $\theta = \psi_i(X_t) \in \mathbb{R}^M$, $\theta' = \psi'_i(X'_t) \in \mathbb{R}^{M'}$ that provide a probabilized set of concurrent extractions of a fixed length vector representation θ of the configuration. For instance it may provide one of many redundant parameterization of the configuration, or a parameterization of a stochastically chosen element of the configuration.
- A pair of forward and backward distributions ϕ_i, ϕ'_i defined over respectively $\mathbb{R}^N$ and $\mathbb{R}^{N'}$. That completes the vectors θ and θ' to match the dimension of the bijective transform.
- A bijective transform $T_i : \mathbb{R}^N \rightarrow \mathbb{R}^P$, where $P = M + N = M' + N'$, such that T'_i, the backward transform is T_i^{-1}

The kernel contribution to the Green ratio is then:

$$\frac{Q_i(X'_t|X_t)}{Q_i(X_t|X'_t)} = \frac{p_i(n'|n)}{p_i(n|n')} \frac{\phi'_i(u')}{\phi_i(u)} \left| \frac{\partial T_i(\theta, u)}{\partial(\theta, u)} \right| \quad (1)$$

2.2 Marked Point Process

A (possibly *multi*-) Marked Point Process (MPP) (van Lieshout, 2000) is a stochastic unordered set of geometric objects of one or many types (e.g. points, segments, circles, ellipses or rectangles in 2D, ellipsoids or spheres in 3D). The **marked point** denomination arises from the usual decomposition of the object parameterization into a **point** of its embedding space (usually its center) and a vector of values (the **marks**) that completes the geometric parameterization of the object (e.g. orientation, length or radius parameters), hence the name **Marked Point Process**.

The `librjmcmc` currently focuses on (multi) MPP in both sampling and optimization contexts. When direct sampling of the MPP is not possible, as when its probability distribution function (PDF) is not samplable and not normalized, the RJMCMC algorithm 1 may be used to build a Markov Chain which distribution converges to the desired target stationary distribution.

A MPP is defined by a probabilized space (Ω, π). If K denotes the set of possible values of a single object, elements of the *configuration* space $\Omega = \bigcup_{n=0}^{\infty} K^n$ are unordered sets of a varying number n of elements in this set K. If the process is multi-marked, Ω is then a Cartesian product of such spaces. A simple probabilization of this configuration space may be given by a probabilization of the set K of each object type and a discrete probabilities defined over the natural numbers $\mathbb{N}$ for each object type that samples the number of object n of each type. This probabilization allows a direct sampling of the configuration space Ω by sampling both the object counts, and then each object independently. In our context, we will consider more complex probabilizations π of the *configuration* space Ω, which sampling will require the more advanced RJMCMC framework.

2.3 Simulated Annealing

Simulated annealing is a physical process inspired by annealing in metallurgy and is widely used in many communities where global optimization is of importance (Salamon et al., 2002). At each step of the algorithm, the stationary distribution is replaced by a similar stationary distribution that is increasingly more selective around the maxima. Coupled with the RJMCMC framework, it enables the global optimization of an extremely large class of energy functions over complex search spaces of varying dimensions. The goal of the simulated annealing process is to drive the initial RJMCMC sampler from an initial probability distribution function given by an energy-agnostic probabilization of the search space (denoted as the reference process) to a target probability distribution function which support is exactly the set of global minima of the energy. To interpolate the PDF between the initial reference PDF and the final PDF, a Boltzmann distribution is usually used (see section 3.2.2), parameterized by a temperature T which decreases from $+\infty$ to zero. The stationary RJMCMC PDF then converges in theory to a mixture of Dirac masses at the global minima of the energy. More informally energy increases are allowed which avoid being trapped in a local minima. But as the temperature T decreases, the maximum allowable energy increase decreases. Depending on the temperature decrease rate and its schedule, and its adequacy to the energy landscape, a solution close to optimal is found in practice.

3 GENERIC LIBRARY DESIGN

3.1 Generic Programming

Generic Programming is a paradigm that offers compile-time polymorphism through templates. It is not as flexible as object-oriented programming but is much more compiler-friendly as it exposes types at compile-time rather than at runtime. Therefore the compiler is given the opportunity to optimize and produce machine code of similar performance than a special-purpose code, while keeping the object-oriented advantage of segregating orthogonal concepts to modularize the library. This even lets us provide various implementations of each orthogonal concept, which are denoted models of these concepts. A concept may be seen as the documentation of the requirements and behavior of a particular template parameter. Concepts have been proposed but not yet accepted as a C++ extension, which would simplify both the documentation and the compiling error reporting of generic libraries.

3.2 RJMCMC Concepts

The RJMCMC sampler is implemented in the class `sampler`, which can be customized using the template parameters: `Density`, `Acceptance` and a list of `Kernels`. It offers access to statistics

(such as the acceptance rate of each kernel) and is used through the following member function that modifies the current configuration in place (preventing useless copies), performing one iteration of algorithm 1.

```
template<typename Configuration>
void sampler::operator()(Configuration &c, double temp);
```

The `Configuration` type and its `Modification` associated type is highly problem dependent, section 2.2 will detail them in the MPP context. The sampler does only access them through the `Kernels`, the reference `Density` and the evaluation of the energy difference ΔU they offer.

3.2.1 `Density` Strategy This strategy provides the reference distribution, by exposing a `double pdf_ratio(const Configuration&, const Modification&)` member function that evaluates the distribution ratio $\pi(X'_t)/\pi(X_t)$, where X_t is provided as a `Configuration` and X'_t as a proposed `Modification` of the current configuration X_t. The `poisson_distribution` and `uniform_distribution` are discrete probabilities over the set of non-negative integers, which are handy as basic blocks to build a `Density`. Discrete distributions expose the following member functions:

```
double pdf_ratio(int n0, int n1) const; // PDF ratio evaluation
double pdf(int n) const; // PDF evaluation
int operator()() const; // PDF sampling
```

3.2.2 `Acceptance` Strategy It encodes how the simulated annealing temperature T is used to introduce the energy bias (through the ΔU term) into the reference PDF, R_∞ denoting the Green ratio of the reference process. `metropolis_acceptance` is the most common choice, using the unnormalized target distribution $\pi e^{-\frac{U}{T}}$. Other acceptance rules depart from the formulation of a well-defined target distribution (Salamon et al., 2002).

Available `Acceptance` Models, $c(x)$ is x clamped to $[0,1]$	
`metropolis_acceptance`	$R_\infty e^{-\frac{\Delta U}{T}}$
`szu_hartley_acceptance`	$R_\infty/(1+e^{\frac{\Delta U}{T}})$
`franz_hoffmann_acceptance`	$R_\infty\left(c\left(1-\frac{1-q}{2-q}\frac{\Delta U}{T}\right)\right)^{\frac{1}{1-q}}$
`tsallis_tsariolo_acceptance`	$R_\infty\left(c\left(1-(1-q)\frac{\Delta U}{T}\right)\right)^{\frac{1}{1-q}}$
`dueck_scheuer_acceptance`	$\begin{cases} 1 & \texttt{if } R_\infty e^{-\frac{\Delta U}{T}} \geq R_0 \\ 0 & \texttt{otherwise} \end{cases}$

3.2.3 `Kernel` Strategy Pairs of reversible kernels are provided together using the following class, the forward and backward kernel being defined by swaping roles of `ViewS`, of `VariateS` and inversing the `Transform`:

```
template<typename View0, typename View1,
         typename Variate0, typename Variate1,
         typename Transform> class kernel;
```

A kernel is used to propose atomic moves to explore the configuration space.

`Variate` Concept It provides the distribution ϕ_i that samples a fixed-length vector and evaluates its probability density.

`View` Concept It encodes the combinatorial aspects of the kernel and the function ψ_i that extract an iterator over θ values when applied forwards and constructs the new configuration based on the vector (θ', u') resulting from the `Transform` when applied backwards.

`Transform` Concept Transformations encode the bijective differentiable mapping T_i between spaces of equal dimensions. They are thus used to describe the bijection applied during a kernel proposal. Transformations must provide `apply`, `inverse` and `abs_jacobian` functions, which are used respectively in the forward kernel, the backward kernel and in the Green ratio computation.

Available `Transform` Models	
`linear_transform`	$A.X$
`affine_transform`	$A.X+B$
`diagonal_linear_transform`	$diag(a_1 \dots a_n).X$
`diagonal_affine_transform`	$diag(a_1 \dots a_n).X+B$

3.3 MPP Concepts

3.3.1 MPP `Configurations` MPPs are commonly used to drive a stochastic exploration in a optimization context where the objective function, hereafter called an `Energy` is composed as the sum of per-object unary terms and per-pair of objects binary terms. For efficiency reasons, `Configurations` are thus tightly coupled with `Energies` in this library. A `Configuration` is a wrapper around a container of objects. The library offers `vector_configuration` and `graph_configuration` `Configuration` models. The former is based on a simple `std::vector` while the latter embeds a `boost::graph` that allows the caching of unary and non-zero binary energy terms, speeding-up ΔU computations.

The only requirement on the object type is that it is able to provide an iterator over a parameterisation vector. Multi-Marked Point Processes (*i.e.* processes of heterogeneous object types) are supported by supplying a `boost::variant` over the types of objects as the `Object` template argument of the configuration. In this context, all library functions that handle explicitly the objects have them first dispatched through the `boost::variant` dispatching function so that functors like `Energies` are `boost::variant`-agnostic.

3.3.2 MPP Energies The library currently supports energies that are defined as sums of unary (U_1) and binary (U_2) energies:

$$U(X_t) = \sum_{x \in X_t} U_1(x) + \sum_{x,y \in X_t, x \neq y} U_2(x,y) \qquad (2)$$

Available `Energy` Models (Tournaire et al., 2010)	
`constant_energy`	(unary and binary)
`image_gradient_unary_energy`	Integrated vector flow
`intersection_area_binary_energy`	Area of intersection
Available Operator `Energy` Models	
`negate_energy`$<E_0>$	$-E_0$
`plus_energy`$<E_0,E_1>$	E_0+E_1
`minus_energy`$<E_0,E_1>$	E_0-E_1
`multiplies_energy`$<E_0,E_1>$	$E_0.E_1$
`divides_energy`$<E_0,E_1>$	E_0/E_1
`modulus_energy`$<E_0,E_1>$	$E_0 \% E_1$

By overloading the C++ operators, the operator `Energy` models allow to build an energy such as $U_2(x,y) = 100 - area(x \cap y)$ by intuitively writing the code `100-intersection_area_binary_energy()`, resulting in a binary energy of type:
`minus_energy<constant_energy,intersection_area_binary_energy>`

3.3.3 MPP `Density` The reference MPP distribution is defined using the class `marked_point_process::direct_sampler`, which combines a discrete distribution (section 3.2.1) for each object type, probabilizing the number of objects and a uniform object sampler that is used to get independent object samples.

3.3.4 `Accelerator` Concept When queried with a configuration `c` and an object `x`, an accelerator provides a subset of the set of objects in `c` that interact with `x` (i.e. that have a non-zero binary energy with it). If the binary energy becomes null or negligible when objects are further apart than a given threshold, an accelerator may only report objects that intersect a suitable ball centered on the query object `x`. Further releases of the library may include accelerators based for instance on quad-trees, uniform grids or Kd-trees. However, the only model of this concept is currently the `trivial_accelerator` that returns the whole range of objects of the query configuration.

3.3.5 MPP `Kernels` For efficiency, the energy evolution ΔU of the modifications proposed by the kernels must be easy to compute. Thus MPP Kernels usually only delete and/or insert a small number of objects. A `Modification` class thus keeps a vector of iterators to the death-proposed objects and a vector of birth-proposed new objects. The simplest MPP kernels that are sufficient for convergence are provided: `RJMCMC::uniform_birth_kernel` and its reverse kernel `RJMCMC::uniform_death_kernel`.

3.4 Simulated Annealing Concepts

The main simulated annealing function is the following free function that performs the optimization loop until the convergence `EndTest` succeeds, advancing the temperature `Schedule` and calling the `Visitor` functor at each iteration:

```
template< class Configuration, class Sampler,
          class Schedule, class EndTest, class Visitor >
void simulated_annealing::optimize(
  Configuration& config, Sampler& sampler,
  Schedule& schedule, EndTest& end_test, Visitor& vis)
{
  double t = *schedule;
  visitor.begin(config,sampler,t);
  for(; !end_test(config,sampler,t); t = *(++schedule))
  {
     sampler(config,t);
     visitor.visit(config,sampler,t);
  }
  visitor.end(config,sampler,t);
}
```

3.4.1 `Schedule` concept Models of the `Schedule` concept are responsible for providing the evolution of the temperature throughout the simulated annealing process. The `Schedule` concept is a refinement of the InputIterator concept of the C++ Standard Template Library.

Available `Schedule` Models	
`geometric_schedule`	$T_t = \alpha^t T_0$
`logarithmic_schedule`	$T_t = \frac{T_0}{\log_2(2+t)}$
`inverse_linear_schedule`	$\frac{1}{T_t} = \frac{1}{T_0} + t\Delta T$
`step_schedule`$<T'>$	$T_t = T'_{\lfloor t/\tau \rfloor}$

Whereas the `logarithmic_schedule` is the one that ensures convergence (Descombes, 2011), the `geometric_schedule` is usually used instead in practice for computational efficiency. The `step_schedule` wraps another schedule to offer constant temperature periods of τ iterations, enabling statistics computation through a `Visitor`.

The evaluation of a suitable initial temperature is not trivial. On the one hand, too high a temperature will unnecessarily prolong the initial phase of the simulated annealing where the sampling is unbiased by the target distribution. On the other hand, too low a temperature prevents the exploration of the whole configuration space and results in the convergence to a local minimum only. (Salamon et al., 2002) suggests considering an estimation of the variance of the energy of configurations sampled according to the reference process to estimate the initial temperature. The corresponding `salamon_initial_schedule` T_0 estimation function is provided, which performs well in the absence of hardcore energy terms.

3.4.2 `EndTest` concept Models of the `EndTest` concept provide a simple predicate that informs the simulated annealing framework that the process has converged or whatever reason requiring to stop the simulated annealing iterations, such as a user-issued cancellation. The only requirements of this concept is to provide the following member function:

```
template<typename Configuration, typename Sampler>
bool operator()(const Configuration& configuration,
        const Sampler& sampler, double temperature);
```

Provided models are `max_iteration_end_test` that fails after a given number of iterations and `delta_energy_end_test` that fails after a given number of iterations without energy change. They may be combined using the `composite_end_test` model.

Available `EndTest` Models	
`max_iteration_end_test`	$t \geq t_{max}$
`delta_energy_end_test`	$\forall i > t - \tau, \Delta U_i = 0$
`composite_end_test`$<E_0, E_1, ...>$	or$(E_0(), E_1(), ...)$

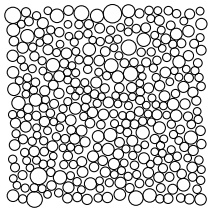
3.4.3 `Visitor` concept `Visitors` are highly customizable objects passed to the simulated annealing process. They may be used to visualize or to gather statistics on the current state of the optimization. Available models offer console (figure 1) or GUI (based on Gilviewer and WxWidgets, figure 2) feedback, or shapefile saving of the current configuration. They may be combined using a `composite_visitor`. This extensibility mechanism offer the possibility to compile the same RJMCMC code with a GUI to build intuition and monitor the running algorithm and without for batch timing purposes.

```
// called once at startup
template<typename Configuration, typename Sampler>
void begin(const Configuration& configuration,
           const Sampler& sampler, double temperature);
// called once per iteration
template<typename Configuration, typename Sampler>
void visit(const Configuration& configuration,
           const Sampler& sampler, double temperature);
// called once at the end
template<typename Configuration, typename Sampler>
void end(const Configuration& configuration,
         const Sampler& sampler, double temperature);
```

4 SAMPLES

4.1 Circle packing

This section details a simple example use of the `librjmcmc` library. The goal is to find a set of circles that minimizes a simple objective function. Each circle center lie in the unit square and its radius is constrained in an interval $[r_{min}, r_{max}]$. Notice that the cardinality of set itself is unknown. The minimized function awards a negative constant value for each circle and penalizes every pair of overlapping circles by its intersection area. Basically, it tries to pack as many circles as possible into the unit square while minimizing their overlap. First, one needs to define the object type of the MPP, with all its geometric methods. Then, the object container is defined as well as an objective function as the sum of a unary term for each object and a binary term for each pair of objects. The set of objects is then probabilized using a Poisson reference process and uniform birth. A basic birth and death RJMCMC sampler is then defined to enable the sampling of the MPP relative to the Poisson reference process, modulated by the score of the objective function and a temperature parameter. Then header files relative to the optimization of the objective function are included. A sufficiently slow temperature decrease morphs the probability distribution function from the one of the reference process to a combination of Dirac masses at the global mimima of the objective function, thus achieving its minimization. A geometric scheduling of the temperature decrease is suboptimal but is more practical and thus more commonly used.



Iteration	Objects	Pkernel	Akernel	Pkernel	Akernel	Accept	Time(ms)	Temp	U_1	U_2	U
1000000	3	50.0251	6.91633	49.9749	6.92748	6.9219	250	73.5759	-3	-7.10543e-14	-3
2000000	4	50.0137	6.9189	49.9863	6.9251	6.922	250	27.0671	-4	-1.13687e-13	-4
3000000	3	50.0744	6.70802	49.9256	6.73041	6.7192	260	9.95741	-3	-1.10134e-13	-3
4000000	4	49.9794	6.67955	50.0206	6.67465	6.6771	250	3.66312	-4	-1.04805e-13	-4
5000000	5	50.0122	6.64158	49.9878	6.64542	6.6435	270	1.34759	-5	-1.04805e-13	-5
6000000	13	49.9979	6.48967	50.0021	6.48753	6.4886	290	0.495749	-13	-1.04583e-13	-13
7000000	38	50.0199	5.25131	49.9801	5.25049	5.2509	430	0.182376	-38	-1.04305e-13	-38
8000000	286	49.9434	1.48088	50.0566	1.42798	1.4544	1700	0.0670923	-286	1.84745	-284.153
9000000	333	49.9631	0.0140103	50.0369	0.00459661	0.0093	3490	0.0246819	-333	8.72057	-324.279
10000000	356	50.0272	0.00559696	49.9728	0.00100054	0.0033	3890	0.00907995	-356	18.3511	-337.649

Figure 1: Output of LaTeXvisitor (left) and the console visitor (right) during a 11s optimization code of section 4.1 (356 objects)

```
using namespace rjmcmc;
using namespace marked_point_process;
using namespace simulated_annealing;

// Objects are circles
#include "geometry/geometry.hpp"
#include "geometry/Circle_2.hpp"
#include "geometry/intersection/Circle_2_intersection.hpp"
#include "geometry/coordinates/Circle_2_coordinates.hpp"
typedef geometry::Simple_cartesian<double> K;
typedef K::Point_2 Point_2;
typedef geometry::Circle_2<K> Circle_2;
typedef Circle_2 object;

// Objective function
#include "rjmcmc/energy/constant_energy.hpp"
#include "rjmcmc/energy/energy_operators.hpp"
#include "mpp/energy/intersection_area_binary_energy.hpp"
#include "mpp/configuration/graph_configuration.hpp"
typedef constant_energy<>                          energy1;
typedef intersection_area_binary_energy<>  area;
typedef multiplies_energy<energy1,area>    energy2;
typedef graph_configuration<object, energy1, energy2>
  configuration;

// Reference process
#include "rjmcmc/distribution/poisson_distribution.hpp"
#include "mpp/kernel/kernel.hpp"
#include "mpp/direct_sampler.hpp"
typedef poisson_distribution  distribution;
typedef uniform_birth<object> uniform_birth;
typedef direct_sampler<distribution,uniform_birth>
 reference_process;

// RJMCMC sampler
#include "rjmcmc/acceptance/metropolis_acceptance.hpp"
#include "rjmcmc/sampler/sampler.hpp"
typedef metropolis_acceptance acceptance;
typedef rjmcmc::sampler<reference_process,acceptance,
 result_of_make_uniform_birth_death_kernel<object>::type>
 sampler;

// Simulated annealing
#include "simulated_annealing.hpp"
#include "schedule/geometric_schedule.hpp"
#include "end_test/max_iteration_end_test.hpp"
#include "visitor/ostream_visitor.hpp"
#include "visitor/tex_visitor.hpp"
#include "visitor/composite_visitor.hpp"

int main(int argc , char** argv)
{
  //parameter parsing
  int i=0;
  double energy = (++i<argc) ? atof(argv[i]): -1.;
  double surface= (++i<argc) ? atof(argv[i]): 10000.;
  double rmin   = (++i<argc) ? atof(argv[i]): 0.02;
  double rmax   = (++i<argc) ? atof(argv[i]): 0.1;
  double poisson= (++i<argc) ? atof(argv[i]): 200.;
  double pbirth = (++i<argc) ? atof(argv[i]): 0.5;
  double pdeath = (++i<argc) ? atof(argv[i]): 0.5;
  int nbiter    = (++i<argc) ? atoi(argv[i]): 10000001;
  double temp   = (++i<argc) ? atof(argv[i]): 200.;
  double deccoef= (++i<argc) ? atof(argv[i]): 0.999999;
  int nbdump    = (++i<argc) ? atoi(argv[i]): 1000000;
  int nbsave    = (++i<argc) ? atoi(argv[i]): 1000000;

  // Reference process
  distribution dpoisson(poisson);
  uniform_birth birth( Circle_2(Point_2(0,0),rmin),
                       Circle_2(Point_2(1,1),rmax) );
  reference_process reference_pdf( dpoisson, birth );

  // Empty configuration linked with the minimized energy
  configuration c(energy, surface*area());
```

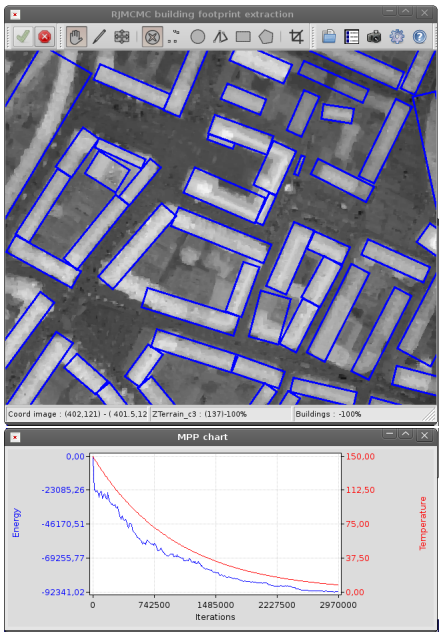


Figure 2: Gilviewer-based wxWidget visitors enabling a runtime visualization of the configuration (top), and of the temperature and energy charts (bottom).

```
  // Optimization
  sampler samp( reference_pdf, acceptance(),
   make_uniform_birth_death_kernel(birth, pbirth, pdeath)
  );

  geometric_schedule<double> sch(temp,deccoef);
  max_iteration_end_test     end(nbiter);

  ostream_visitor osvisitor;
  tex_visitor texvisitor("quickstart_");
  composite_visitor<ostream_visitor,tex_visitor>
    visitor(osvisitor,texvisitor);
  visitor.init(nbdump,nbsave);

  optimize(c,samp,sch,end,visitor);

  return 0;
}
```

The main entry point starts by parsing the input parameters, providing default values. Once the reference Poisson process is instantiated, an empty configuration is created and initialized with the energy objects. Then, the RJMCMC sampler object `samp` is constructed. Finally, the `simulated_annealing` objects are created and the optimization is performed in place on the `configuration` instance `c`. Figure 1 shows the output of the program, and the resulting circle packing. Its runtime is 11s for 356 circles and 10^7 iterations.

4.2 Building extraction

The method introduced in (Tournaire et al., 2010) has been implemented within the `librjmcmc` library, which extracts buildings

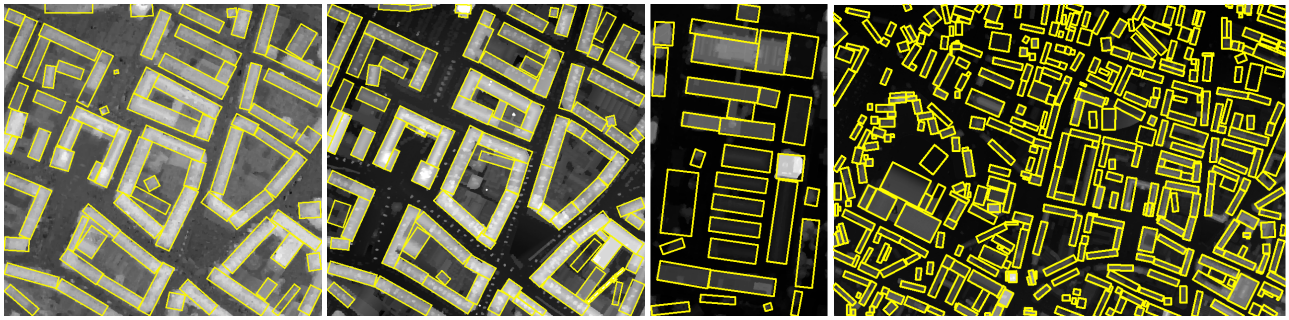


Figure 3: From left to right (a), (b), (c) and (d). Building extraction results using a MPP of rectangles (Tournaire et al., 2010).

from a Digital Surface Model (DSM) using a MPP of rectangles. This implementation, which has been included in the `librjmcmc` to foster reproducible research, amounts to provide the problem-dependent object: the rectangle type, the energies and the kernels. We show here results of this implementation on 4 different DSMs (figure 3), from 10 cm to 50cm Ground Sample Distance (GSD), acquired using photogrammetry or Lidar and of various sizes. Figures 3.a and b are on the same area which is also contained in figure 3.d. The size of the dataset 3.d shows the scalability of the proposed approach, even in the absence of a proper acceleration (such as a quad tree).

Input DSM				MPP	
Figure	Type	GSD	Image size	Rectangles	Time
3.a	Photo	50cm	650× 650	76	177s
3.b	Lidar	50cm	650× 650	109	179s
3.c	Photo	10cm	800× 1400	31	104s
3.d	Photo	50cm	3634× 2502	426	480s

5 CONCLUSIONS AND FUTURE WORK

Generic programming is very efficient at run-time and this library showcases how modular high-level code performs well at runtime (Abrahams and Gurtovoy, 2004). Once accustomed to the necessary typedef clauses, using the library reduces to simply describing the problem at a high-level rather. Its development is however quite cumbersome, due to the extra burden allocated to the compiler, and its lack of native language support. Until concepts are accepted as part of C++, there are preliminary ways to better document them and to get more legible error reporting when the library is misused. `librjmcmc` now only checks the validity of the Schedule concept as an input iterator. Future work would be to systematically check template parameters for their intended concept.

Implementations of close relatives of the RJMCMC algorithm, such as jump diffusion or multiple birth and death (Descombes, 2011) would share many code parts. It would be interesting to implement them in a common library to assess their comparative strengths.

In the `librjmcmc` , complex binary or unary energy terms may be built using expressions on simpler energies. This technique is called a Domain-Specific Embedded Language (DSEL). It would be nice to extend its use the whole energy formulation, kernels, distributions, and even transforms so that, for instance, complex transform would be able to generate their Jacobian computation code at compile-time. This would help hiding the complex type names produced by the generic programming and help offering a more user-friendly interface. boost::proto is a framework to define such DSELs, but its ease of use and runtime costs will have to be assessed.

Finally, the `librjmcmc` has now only be used within a 2D MPP context. Future interesting work would be to implement other Configuration, Kernel and Energy models, in embedding spaces of other dimensions (1D signals (Hernandez-Marin et al., 2007; Mallet et al., 2010), 3D ellipses (Perrin et al., 2006)...) or even in a non-MPP context, for instance to optimize a 3D triangulation over the noisy heightfield of photogrammetric Digital Surface Model. Our wish is that the open-source license of the `librjmcmc` library will enable reproducible research.

References

Abrahams, D. and Gurtovoy, A., 2004. C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond. C++ in-depth series, Addison-Wesley professional.

Descombes, X., 2011. Stochastic Geometry for Image Analysis. Publ. Wiley/Iste, Hoboken, USA.

Gilviewer, 2012. A C++ library for developping applications with 2D raster and vector viewer functionalities. http://code.google.com/p/gilviewer/.

Green, P. J., 1995. Reversible Jump Markov Chain Monte Carlo computation and Bayesian model determination. Biometrika vol. 82(4), pp. 711–732.

Hastings, W. K., 1970. Monte Carlo sampling using Markov chains and their applications. Biometrica 57(1), pp. 97–109.

Hernandez-Marin, S., Wallace, A. M. and Gibson, G. J., 2007. Bayesian analysis of lidar signals with multiple returns. IEEE Trans. Pattern Anal. Mach. Intell. vol. 29(12), pp. 2170–2180.

librjmcmc, 2012. A generic C++ library for stochastic optimization. http://librjmcmc.ign.fr.

Mallet, C., Lafarge, F., Roux, M., Soergel, U., Bretar, F. and Heipke, C., 2010. A Marked Point Process for modeling lidar waveforms. IEEE Transactions on Image Processing vol. 19 (12), pp. 3204–3221.

Perrin, G., Descombes, X. and Zerubia, J., 2006. 2D and 3D vegetation resource parameters assessment using marked point processes. In: Proc. of the International Conference on Pattern Recognition, Hong-Kong, China.

Salamon, P., Sibani, P. and Frost, R., 2002. Facts, conjectures and improvements for simulated annealing. Society for Industrial and Applied Mathematics, Philadelphia, USA.

Tournaire, O., Brédif, M., Boldo, D. and Durupt, M., 2010. An efficient stochastic approach for building footprint extraction from digital elevation models. ISPRS Journal of Photogrammetry and Remote Sensing vol. 65(4), pp. 317–327.

van Lieshout, M. N. M., 2000. Markov point processes and their applications. Imperial College Press, London, England.