MCMCLib

Keith O'Hara

Feb 08, 2024

GUIDE

MCMCLib is a lightweight C++ library of Markov Chain Monte Carlo (MCMC) methods.

Features:

- A C++11/14/17 library of well-known MCMC algorithms.
- Parallelized samplers designed for multi-modal distributions, including:
  - Differential Evolution (DE); and
  - Adaptive Equi-Energy Sampler (AEES).
- For fast and efficient matrix-based computation, MCMCib supports the following templated linear algebra libraries:
  - Armadillo
  - Eigen (version >= 3.4.0)
- Automatic differentiation functionality is available through use of the Autodiff library
- OpenMP-accelerated algorithms for parallel computation.
- Straightforward linking with parallelized BLAS libraries, such as OpenBLAS.
- Available as a header-only library, or as a compiled shared library.
- Released under a permissive, non-GPL license.

Author: Keith O'Hara

License: Apache Version 2.0

CHAPTER

ONE

INSTALLATION

The library can be installed on Unix-alike systems via the standard `./configure && make` method.

See the installation page for *detailed instructions*.

CHAPTER

TWO

ALGORITHMS

A list of currently available algorithms includes:

- *Adaptive Equi-Energy Sampler (AEES)*
- *Differential Evolution (DE-MCMC)*
- *Hamiltonian Monte Carlo (HMC)*
- *Metropolis-adjusted Langevin algorithm (MALA)*
- *No-U-Turn Sampler (NUTS)*
- *Random Walk Metropolis-Hastings (RWMH)*
- *Riemannian Manifold Hamiltonian Monte Carlo (RM-HMC)*

CHAPTER

THREE

CONTENTS

3.1 Installation

MCMCLib is available as a compiled shared library, or as header-only library, for Unix-alike systems only (e.g., popular Linux-based distros, as well as macOS). Note that use of this library with Windows-based systems, with or without MSVC, **is not supported**.

3.1.1 Requirements

MCMCLib requires either the Armadillo or Eigen C++ linear algebra libraries. (Note that Eigen version 3.4.0 requires a C++14-compatible compiler.)

The following options should be declared **before** including the MCMCLib header files.

- OpenMP functionality is enabled by default if the `_OPENMP` macro is detected (e.g., by invoking `-fopenmp` with GCC or Clang).
  - To explicitly enable OpenMP features, use:

```
#define MCMC_USE_OPENMP
```

  - To explicitly disable OpenMP functionality, use:

```
#define MCMC_DONT_USE_OPENMP
```

- To use MCMCLib with Armadillo or Eigen:

```
#define MCMC_ENABLE_ARMA_WRAPPERS
#define MCMC_ENABLE_EIGEN_WRAPPERS
```

  Example:

```
#define MCMC_ENABLE_EIGEN_WRAPPERS
#include "mcmc.hpp"
```

- To use MCMCLib with RcppArmadillo:

```
#define MCMC_USE_RCPP_ARMADILLO
```

  Example:

```
#define MCMC_USE_RCPP_ARMADILLO
#include "mcmc.hpp"
```

3.1.2 Installation Method 1: Shared Library

The library can be installed on Unix-alike systems via the standard `./configure && make` method.

The primary configuration options can be displayed by calling `./configure -h`, which results in:

```
$ ./configure -h

MCMCLib Configuration

Main options:
-c    Code coverage build
        (default: disabled)
-d    Developmental build
        (default: disabled)
-f    Floating-point number type
        (default: double)
-g    Debugging build (optimization flags set to -O0 -g)
        (default: disabled)
-h    Print help
-i    Install path (default: current directory)
        Example: /usr/local
-l    Choice of linear algebra library
        Examples: -l arma or -l eigen
-m    Specify the BLAS and Lapack libraries to link against
        Examples: -m "-lopenblas" or -m "-framework Accelerate"
-o    Compiler optimization options
        (default: -O3 -march=native -ffp-contract=fast -flto -DARMA_NO_DEBUG)
-p    Enable OpenMP parallelization features
        (default: disabled)

Special options:
--header-only-version    Generate a header-only version of MCMCLib
```

If choosing a shared library build, set (one) of the following environment variables *before* running *configure*:

```
export ARMA_INCLUDE_PATH=/path/to/armadillo
export EIGEN_INCLUDE_PATH=/path/to/eigen
```

Then, to set the install path to `/usr/local`, use Armadillo as the linear algebra library, and enable OpenMP features, we would run:

```
./configure -i "/usr/local" -l arma -p
```

Following this with the standard `make && make install` would build the library and install into `/usr/local`.

3.1.3 Installation Method 2: Header-only Library

MCMCLib is also available as a header-only library (i.e., without the need to compile a shared library). Simply run `configure` with the `--header-only-version` option:

```
./configure --header-only-version
```

This will create a new directory, `header_only_version`, containing a copy of MCMCLib, modified to work on an inline basis. With this header-only version, simply include the header files (`#include "mcmc.hpp`) and set the include path to the `head_only_version` directory (e.g.,``-I/path/to/mcmclib/header_only_version``).

3.2 Examples and Tests

3.2.1 API

The MCMCLib API follows a relatively simple convention, with most algorithms called using the following syntax:

```
algorithm_id(<initial values>, <log posterior kernel function of the target distribution>
↪, <storage for posterior draws>, <additional data for the log posterior kernel␣
↪function>);
```

The inputs, in order, are:

- A vector of initial values used to define the starting point of the algorithm.
- A user-specified function that returns the log posterior kernel value of the target distribution.
- An array to store the posterior draws.
- The final input is optional: it is any object that contains additional data necessary to evaluate the log posterior kernel function.

For example, the RWMH algorithm is called using

```
rwmh(const ColVec_t& initial_vals, std::function<fp_t (const ColVec_t& vals_inp, void*␣
↪target_data)> target_log_kernel, Mat_t& draws_out, void* target_data);
```

3.2.2 Example

The code below uses the RWMH algorithm generate draws of the mean of a Gaussian likelihood function.

```
#define MCMC_ENABLE_EIGEN_WRAPPERS
#include "mcmc.hpp"

inline
Eigen::VectorXd
eigen_randn_colvec(size_t nr)
{
    static std::mt19937 gen{ std::random_device{}() };
    static std::normal_distribution<> dist;

    return Eigen::VectorXd{ nr }.unaryExpr([&](double x) { (void)(x); return dist(gen); }
↪);
}

struct norm_data_t {
    double sigma;
    Eigen::VectorXd x;

    double mu_0;
    double sigma_0;
};

double ll_dens(const Eigen::VectorXd& vals_inp, void* ll_data)
{
    const double pi = 3.14159265358979;

    //

    const double mu = vals_inp(0);

    norm_data_t* dta = reinterpret_cast<norm_data_t*>(ll_data);
    const double sigma = dta->sigma;
    const Eigen::VectorXd x = dta->x;

    const int n_vals = x.size();

    //

    const double ret = - n_vals * (0.5 * std::log(2*pi) + std::log(sigma)) - (x.array() -
↪ mu).pow(2).sum() / (2*sigma*sigma);

    //

    return ret;
}

double log_pr_dens(const Eigen::VectorXd& vals_inp, void* ll_data)
{
    const double pi = 3.14159265358979;
```

(continues on next page)

(continued from previous page)

```
    //

    norm_data_t* dta = reinterpret_cast< norm_data_t* >(ll_data);

    const double mu_0 = dta->mu_0;
    const double sigma_0 = dta->sigma_0;

    const double x = vals_inp(0);

    const double ret = - 0.5*std::log(2*pi) - std::log(sigma_0) - std::pow(x - mu_0,2) /␣
↪(2*sigma_0*sigma_0);

    return ret;
}

double log_target_dens(const Eigen::VectorXd& vals_inp, void* ll_data)
{
    return ll_dens(vals_inp,ll_data) + log_pr_dens(vals_inp,ll_data);
}

int main()
{
    const int n_data = 100;
    const double mu = 2.0;

    norm_data_t dta;
    dta.sigma = 1.0;
    dta.mu_0 = 1.0;
    dta.sigma_0 = 2.0;

    Eigen::VectorXd x_dta = mu + eigen_randn_colvec(n_data).array();
    dta.x = x_dta;

    Eigen::VectorXd initial_val(1);
    initial_val(0) = 1.0;

    //

    mcmc::algo_settings_t settings;

    settings.rwmh_settings.par_scale = 0.4;
    settings.rwmh_settings.n_burnin_draws = 2000;
    settings.rwmh_settings.n_keep_draws = 2000;

    //

    Eigen::MatrixXd draws_out;
    mcmc::rwmh(initial_val, log_target_dens, draws_out, &dta, settings);

    //
```

(continues on next page)

(continued from previous page)

```
    std::cout << "de mean:\n" << draws_out.colwise().mean() << std::endl;
    std::cout << "acceptance rate: " << static_cast<double>(settings.rwmh_settings.n_
↪accept_draws) / settings.rwmh_settings.n_keep_draws << std::endl;

    //

    return 0;
}
```

On x86-based computers, this example can be compiled using:

```
g++ -Wall -std=c++14 -O3 -mcpu=native -ffp-contract=fast -I$EIGEN_INCLUDE_PATH -I./../../
↪include/ rwmh_normal_mean.cpp -o rwmh_normal_mean.out -L./../.. -lmcmc
```

3.2.3 Test suite

You can build the test suite as follows:

```
# compile tests
cd ./tests
./setup
cd ./examples
./configure -l eigen
make
./rwmh.test
```

3.3 MCMC Settings

3.3.1 Main

An object of type `algo_settings_t` can be used to control the behavior of the MCMC routines. Each algorithm page details the relevant parameters for that methods, but we list the full settings here for completeness.

```
struct algo_settings_t
{
    // RNG seeding

    size_t rng_seed_value = std::random_device{}();

    // bounds

    bool vals_bound = false;

    ColVec_t lower_bounds;
    ColVec_t upper_bounds;

    // AEES
    aees_settings_t aees_settings;

    // DE
    de_settings_t de_settings;

    // HMC
    hmc_settings_t hmc_settings;

    // RM-HMC
    rmhmc_settings_t rmhmc_settings;

    // MALA
    mala_settings_t mala_settings;

    // RWMH
    rwmh_settings_t rwmh_settings;
};
```

Description:

- `rng_seed_value` seed value used for random number generators.
- `vals_bound` whether the search space of the algorithm is bounded.
- `lower_bounds` defines the lower bounds of the search space.
- `upper_bounds` defines the upper bounds of the search space.

Algorithm-specific data structures are listed in the next section.

3.3.2 By Algorithm

AEES

```
struct aees_settings_t
{
    size_t n_initial_draws = 1E03;
    size_t n_burnin_draws = 1E03;
    size_t n_keep_draws = 1E03;

    int omp_n_threads = -1; // numbers of threads to use

    fp_t par_scale = 1.0;
    Mat_t cov_mat;

    size_t n_rings = 5; // number of energy rings
    fp_t ee_prob_par = 0.10; // equi-energy probability parameter
    ColVec_t temper_vec; // temperature vector
};
```

DE

```
struct de_settings_t
{
    bool jumps = false;

    size_t n_pop = 100;
    size_t n_burnin_draws = 1E03;
    size_t n_keep_draws = 1E03;

    int omp_n_threads = -1; // numbers of threads to use

    fp_t par_b = 1E-04;
    fp_t par_gamma = 1.0;
    fp_t par_gamma_jump = 2.0;

    ColVec_t initial_lb; // this will default to -0.5
    ColVec_t initial_ub; // this will default to  0.5

    size_t n_accept_draws; // will be returned by the algorithm
};
```

HMC

```
struct hmc_settings_t
{
    size_t n_burnin_draws = 1E03;
    size_t n_keep_draws = 1E03;

    int omp_n_threads = -1; // numbers of threads to use

    size_t n_leap_steps = 1; // number of leap frog steps
    fp_t step_size = 1.0;
    Mat_t precond_mat;

    size_t n_accept_draws; // will be returned by the function
};
```

RM-HMC

```
struct rmhmc_settings_t
{
    size_t n_burnin_draws = 1E03;
    size_t n_keep_draws = 1E03;

    int omp_n_threads = -1; // numbers of threads to use

    size_t n_leap_steps = 1; // number of leap frog steps
    fp_t step_size = 1.0;
    Mat_t precond_mat;

    size_t n_fp_steps = 5; // number of fixed point iteration steps

    size_t n_accept_draws; // will be returned by the function
};
```

MALA

```
struct mala_settings_t
{
    size_t n_burnin_draws = 1E03;
    size_t n_keep_draws = 1E03;

    int omp_n_threads = -1; // numbers of threads to use

    fp_t step_size = 1.0;
    Mat_t precond_mat;

    size_t n_accept_draws; // will be returned by the function
};
```

RWMH

```
struct rwmh_settings_t
{
    size_t n_burnin_draws = 1E03;
    size_t n_keep_draws = 1E03;

    int omp_n_threads = -1; // numbers of threads to use

    fp_t par_scale = 1.0;
    Mat_t cov_mat;

    size_t n_accept_draws; // will be returned by the function
};
```

3.4 Automatic Differentiation

Gradient-based MCMC methods in MCMCLib (such as HMC and MALA) require a user-defined function that returns a gradient vector at each function evaluation. While this is best achieved by knowing the gradient in closed form, MCMCLib also provides **experimental support** for automatic differentiation with Eigen-based builds via the autodiff library.

Requirements: an Eigen-based build of MCMCLib, a copy of the `autodiff` header files, and a C++17 compatible compiler.

3.4.1 Example

The example below uses forward-mode automatic differentiation to compute the gradient of the Gaussian likelihood function, and the HMC algorithm to sample from the posterior distribution of the mean and variance parameters.

```
/*
 *  Sampling from a Gaussian distribution using HMC and Autodiff
 */

#define MCMC_ENABLE_EIGEN_WRAPPERS
#include "mcmc.hpp"

#include <autodiff/forward/real.hpp>
#include <autodiff/forward/real/eigen.hpp>

inline
Eigen::VectorXd
eigen_randn_colvec(size_t nr)
{
    static std::mt19937 gen{ std::random_device{}() };
    static std::normal_distribution<> dist;
```

(continues on next page)

(continued from previous page)

```
    return Eigen::VectorXd{ nr }.unaryExpr([&](double x) { (void)(x); return dist(gen); }
↪);
}

struct norm_data_t {
    Eigen::VectorXd x;
};

double ll_dens(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void* ll_data)
{
    const double pi = 3.14159265358979;

    norm_data_t* dta = reinterpret_cast<norm_data_t*>(ll_data);
    const Eigen::VectorXd x = dta->x;

    //

    autodiff::real u;
    autodiff::ArrayXreal xd = vals_inp.eval();

    std::function<autodiff::real (const autodiff::ArrayXreal& vals_inp)> normal_dens_log_
↪form \
    = [x, pi](const autodiff::ArrayXreal& vals_inp) -> autodiff::real
    {
        autodiff::real mu    = vals_inp(0);
        autodiff::real sigma = vals_inp(1);

        return - x.size() * (0.5 * std::log(2*pi) + autodiff::detail::log(sigma)) - (x.
↪array() - mu).pow(2).sum() / (2*sigma*sigma);
    };

    //

    if (grad_out) {
        Eigen::VectorXd grad_tmp = autodiff::gradient(normal_dens_log_form,␣
↪autodiff::wrt(xd), autodiff::at(xd), u);

        *grad_out = grad_tmp;
    } else {
        u = normal_dens_log_form(xd);
    }

    //

    return u.val();
}

double log_target_dens(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void*␣
↪ll_data)
{
    return ll_dens(vals_inp,grad_out,ll_data);
}
```

(continues on next page)

(continued from previous page)

```
int main()
{
    const int n_data = 1000;

    const double mu = 2.0;
    const double sigma = 2.0;

    norm_data_t dta;

    Eigen::VectorXd x_dta = mu + sigma * eigen_randn_colvec(n_data).array();
    dta.x = x_dta;

    Eigen::VectorXd initial_val(2);
    initial_val(0) = mu + 1; // mu
    initial_val(1) = sigma + 1; // sigma

    mcmc::algo_settings_t settings;

    settings.hmc_settings.step_size = 0.08;
    settings.hmc_settings.n_burnin_draws = 2000;
    settings.hmc_settings.n_keep_draws = 2000;

    //

    Eigen::MatrixXd draws_out;
    mcmc::hmc(initial_val, log_target_dens, draws_out, &dta, settings);

    //

    std::cout << "hmc mean:\n" << draws_out.colwise().mean() << std::endl;
    std::cout << "acceptance rate: " << static_cast<double>(settings.hmc_settings.n_
↪accept_draws) / settings.hmc_settings.n_keep_draws << std::endl;

    //

    return 0;
}
```

This example can be compiled using:

```
g++ -Wall -std=c++17 -O3 -march=native -ffp-contract=fast -I/path/to/eigen -I/path/to/
↪autodiff -I/path/to/mcmc/include hmc_normal_autodiff.cpp -o hmc_normal_autodiff.cpp -L/
↪path/to/mcmc/lib -lmcmc
```

3.5 Adaptive Equi-Energy Sampler

Table of contents

3.5.1 Algorithm Description

The Adaptive Equi-Energy Sampler (AEES) algorithm is a Markov Chain Monte Carlo method designed to generate samples from multi-modal target distributions. See Kou, Zhou, Wong (2006) for details of the standard equi-energy sampler, and Schreck, Fort, Moulines (2013) for the adaptive version (presented here).

Let $\theta_k^{(i)}$ denote a d-dimensional vector of stored values at stage i of the algorithm, drawn from target distribution π_k, where $k \in \{0, 1, \ldots, K\}$ and K denotes the number of energy rings. We will use the following notation to define a tempered target distribution:

$$\pi_k(\theta) \propto \exp(-H(\theta|X)/T_k)$$

where T_k denotes the temperature (with $T_0 = 1$) and H denotes the energy function (i.e., the negative of the log-posterior kernel function).

The AEES algorithm proceeds as follows.

1. Sample $\theta_K^{(i+1)} \sim \pi_K$ using Metropolis-Hastings.
2. **for** $k \in \{K-1, K-2, \ldots, 0\}$ **do**: if $i > (K-k) \times$ (`n_initial_draws` + `n_burnin_draws`):
   i. Sample $z \sim U(0,1)$
   ii. (Local move) if $z >$ `ee_prob_par`, sample $\theta_k^{(i+1)} \sim \pi_k$ using Metropolis-Hastings.
   iii. (Equi-energy move) if $z \leq$ `ee_prob_par`:
   - construct `n_rings` number of evenly spaced energy rings using previous draws from π_{k+1}: $\{\theta_{k+1}^{(0)}, \ldots, \theta_{k+1}^{(i)}\}$.

     $$\alpha = \min\left\{1, \frac{\pi_k(\theta_k^{(*)})}{\pi_{k+1}(\theta_k^{(*)})} \frac{\pi_{k+1}(\theta_k^{(i)})}{\pi_k(\theta_k^{(i)})}\right\}$$

     where

     $$\theta_k^{(i+1)} = \begin{cases} \theta_k^{(*)} & \text{with probability } \alpha \\ \theta_k^{(i)} & \text{else} \end{cases}$$

The algorithm stops when the number of draws reaches `n_initial_draws` + `n_burnin_draws` + `n_keep_draws`, and returns the final `n_keep_draws` number of draws.

3.5.2 Function Declarations

bool **aees**(const ColVec_t &initial_vals, std::function<fp_t(const ColVec_t &vals_inp, void *target_data)> target_log_kernel, Mat_t &draws_out, void *target_data)

The Adaptive Equi-Energy MCMC Algorithm.

Parameters

- **initial_vals** – a column vector of initial values.
- **target_log_kernel** – the log posterior kernel function of the target distribution, taking two arguments:
  - `vals_inp` a vector of inputs; and
  - `target_data` additional data passed to the user-provided function.
- **draws_out** – a matrix of posterior draws, where each row represents one draw.
- **target_data** – additional data passed to the user-provided function.

Returns

a boolean value indicating successful completion of the sampling algorithm.

bool **aees**(const ColVec_t &initial_vals, std::function<fp_t(const ColVec_t &vals_inp, void *target_data)> target_log_kernel, Mat_t &draws_out, void *target_data, algo_settings_t &settings)

The Adaptive Equi-Energy MCMC Algorithm.

Parameters

- **initial_vals** – a column vector of initial values.
- **target_log_kernel** – the log posterior kernel function of the target distribution, taking two arguments:
  - `vals_inp` a vector of inputs; and
  - `target_data` additional data passed to the user-provided function.
- **draws_out** – a matrix of posterior draws, where each row represents one draw.
- **target_data** – additional data passed to the user-provided function.
- **settings** – parameters controlling the sampling algorithm.

Returns

a boolean value indicating successful completion of the sampling algorithm.

3.5.3 Control Parameters

The basic control parameters are:

- `size_t aees_settings.n_initial_draws`: number of initial draws.
- `size_t aees_settings.n_burnin_draws`: number of burn-in draws.
- `size_t aees_settings.n_keep_draws`: number of draws to keep (post sample burn-in period).
- `bool vals_bound`: whether the search space of the algorithm is bounded. If `true`, then
  - `ColVec_t lower_bounds`: defines the lower bounds of the search space.
  - `ColVec_t upper_bounds`: defines the upper bounds of the search space.

Additional settings:

- `int aees_settings.omp_n_threads`: the number of OpenMP threads to use.
  - Default value: `-1` (use all available threads divided by 2).
- `fp_t aees_settings.par_scale`: scaling parameter for Metropolis-Hastings draws.
  - Default value: `1.0`.
- `Mat_t aees_settings.cov_mat`: covariance matrix of Metropolis-Hastings draws.
  - Default value: diagonal matrix.
- `size_t aees_settings.n_rings`: the number of energy rings.
  - Default value: `5`.
- `fp_t aees_settings.ee_prob_par`: the equi-energy sampling probability.
  - Default value: `0.10`.
- `ColVec_t aees_settings.temper_vec`: a vector of temperature values.

3.5.4 Examples

Gaussian Mixture Distribution

Code to run this example is given below.

Armadillo (Click to show/hide)

```
#define MCMC_ENABLE_ARMA_WRAPPERS
#include "mcmc.hpp"

struct mixture_data_t {
    arma::mat mu;
    arma::vec sig_sq;
    arma::vec weights;
};

double
gaussian_mixture(const arma::vec& X_vec_inp, const arma::vec& weights, const arma::mat&␣
↪mu, const arma::vec& sig_sq)
{
    const double pi = arma::datum::pi;

    const int n_vals = X_vec_inp.n_elem;
    const int n_mix = weights.n_elem;

    //

    double dens_val = 0;

    for (int i = 0; i < n_mix; ++i) {
        const double dist_val = arma::accu(arma::pow(X_vec_inp - mu.col(i), 2));
```

(continues on next page)

(continued from previous page)

```
        dens_val += weights(i) * std::exp(-0.5 * dist_val / sig_sq(i)) / std::pow(2.0 *␣
↪pi * sig_sq(i), static_cast<double>(n_vals) / 2.0);
    }

    //

    return std::log(dens_val);
}

double
target_log_kernel(const arma::vec& vals_inp, void* target_data)
{
    mixture_data_t* dta = reinterpret_cast<mixture_data_t*>(target_data);

    return gaussian_mixture(vals_inp, dta->weights, dta->mu, dta->sig_sq);
}

int main()
{
    const int n_vals = 2;
    const int n_mix  = 2;

    //

    arma::mat mu = arma::ones(n_vals, n_mix) + 1.0;
    mu.col(0) *= -1.0; // (-2, 2)

    arma::vec weights(n_mix, arma::fill::value(1.0 / n_mix));

    arma::vec sig_sq = 0.1 * arma::ones(n_mix);

    mixture_data_t dta;
    dta.mu = mu;
    dta.sig_sq = sig_sq;
    dta.weights = weights;

    //

    arma::vec T_vec(2);
    T_vec(0) = 60.0;
    T_vec(1) = 9.0;

    // settings

    mcmc::algo_settings_t settings;

    settings.aees_settings.n_initial_draws = 1000;
    settings.aees_settings.n_burnin_draws  = 1000;
    settings.aees_settings.n_keep_draws    = 20000;

    settings.aees_settings.n_rings = 11;
    settings.aees_settings.ee_prob_par = 0.05;
```

(continues on next page)

(continued from previous page)

```
    settings.aees_settings.temper_vec = T_vec;

    settings.aees_settings.par_scale = 1.0;
    settings.aees_settings.cov_mat = 0.35 * arma::eye(n_vals, n_vals);

    //

    arma::mat draws_out;

    mcmc::aees(mu.col(0), target_log_kernel, draws_out, &dta, settings);

    arma::cout << "posterior mean for > 0.1:\n" << arma::mean(draws_out.elem(␣
↪arma::find(draws_out > 0.1) ), 0) << arma::endl;
    arma::cout << "posterior mean for < -0.1:\n" << arma::mean(draws_out.elem(␣
↪arma::find(draws_out < -0.1) ), 0) << arma::endl;

    //

    return 0;
}
```

Eigen (Click to show/hide)

```
#define MCMC_ENABLE_EIGEN_WRAPPERS
#include "mcmc.hpp"

struct mixture_data_t {
    Eigen::MatrixXd mu;
    Eigen::VectorXd sig_sq;
    Eigen::VectorXd weights;
};

double
gaussian_mixture(const Eigen::VectorXd& X_vec_inp, const Eigen::VectorXd& weights, const␣
↪Eigen::MatrixXd& mu, const Eigen::VectorXd& sig_sq)
{
    const double pi = 3.14159265358979;

    const int n_vals = X_vec_inp.size();
    const int n_mix = weights.size();

    //

    double dens_val = 0;

    for (int i = 0; i < n_mix; ++i) {
        const double dist_val = (X_vec_inp - mu.col(i)).array().pow(2).sum();

        dens_val += weights(i) * std::exp(-0.5 * dist_val / sig_sq(i)) / std::pow(2.0 *␣
↪pi * sig_sq(i), static_cast<double>(n_vals) / 2.0);
    }
```

(continues on next page)

(continued from previous page)

```
    //

    return std::log(dens_val);
}

double
target_log_kernel(const Eigen::VectorXd& vals_inp, void* target_data)
{
    mixture_data_t* dta = reinterpret_cast<mixture_data_t*>(target_data);

    return gaussian_mixture(vals_inp, dta->weights, dta->mu, dta->sig_sq);
}

int main()
{
    const int n_vals = 2;
    const int n_mix  = 2;

    //

    Eigen::MatrixXd mu = Eigen::MatrixXd::Ones(n_vals, n_mix).array() + 1.0;
    mu.col(0) *= -1.0; // (-2, 2)

    Eigen::VectorXd weights = Eigen::VectorXd::Constant(n_mix, 1.0 / n_mix);

    Eigen::VectorXd sig_sq = 0.1 * Eigen::VectorXd::Ones(n_mix);

    mixture_data_t dta;
    dta.mu = mu;
    dta.sig_sq = sig_sq;
    dta.weights = weights;

    //

    Eigen::VectorXd T_vec(2);
    T_vec(0) = 60.0;
    T_vec(1) = 9.0;

    // settings

    mcmc::algo_settings_t settings;

    settings.aees_settings.n_initial_draws = 1000;
    settings.aees_settings.n_burnin_draws  = 1000;
    settings.aees_settings.n_keep_draws    = 20000;

    settings.aees_settings.n_rings = 11;
    settings.aees_settings.ee_prob_par = 0.05;
    settings.aees_settings.temper_vec = T_vec;

    settings.aees_settings.par_scale = 1.0;
    settings.aees_settings.cov_mat = 0.35 * Eigen::MatrixXd::Identity(n_vals, n_vals);
```

(continues on next page)

(continued from previous page)

```
    //

    Eigen::MatrixXd draws_out;

    mcmc::aees(mu.col(0), target_log_kernel, draws_out, &dta, settings);

    //

    Eigen::Matrix<bool, Eigen::Dynamic, Eigen::Dynamic> pos_inds = (draws_out.array() >␣
↪0.1);

    Eigen::VectorXd mean_vec = Eigen::VectorXd::Zero(2);

    for (int i = 0; i < n_vals; ++i) {
        for (size_t draw_ind = 0; draw_ind < settings.aees_settings.n_keep_draws; ++draw_
↪ind) {
            if (pos_inds(draw_ind, i)) {
                mean_vec(i) += draws_out(draw_ind, i);
            }
        }
        mean_vec(i) /= pos_inds.col(i).count();
    }

    std::cout << "posterior mean for > 0.1:\n" << mean_vec << std::endl;

    //

    Eigen::Matrix<bool, Eigen::Dynamic, Eigen::Dynamic> neg_inds = (draws_out.array() < -
↪ 0.1);

    mean_vec = Eigen::VectorXd::Zero(2);

    for (int i = 0; i < n_vals; ++i) {
        for (size_t draw_ind = 0; draw_ind < settings.aees_settings.n_keep_draws; ++draw_
↪ind) {
            if (neg_inds(draw_ind, i)) {
                mean_vec(i) += draws_out(draw_ind, i);
            }
        }
        mean_vec(i) /= neg_inds.col(i).count();
    }

    std::cout << "posterior mean for < - 0.1:\n" << mean_vec << std::endl;

    //

    return 0;
}
```

3.6 Differential Evolution

Table of contents

3.6.1 Algorithm Description

The Differential Evolution (DE) sampler is a Markov Chain Monte Carlo variant of the well-known stochastic genetic search algorithm used for global optimization. See Cajo J. F. Ter Braak (2006) for details.

Let $\boldsymbol{\theta}_k^{(i)}$ denote a $N_p \times d$-dimensional array of stored values at stage i of the algorithm, and let $\gamma = 2.38/\sqrt{2d}$. The DE-MCMC algorithm proceeds in two steps.

1. (**Mutation Step**) For random and unique indices j, k, set:

$$\theta^{(*)} = \theta^{(i)} + \gamma \times (\boldsymbol{\theta}^{(i)}(j,:) - \boldsymbol{\theta}^{(i)}(k,:))) + U$$

   where $U \sim \text{Unif}[-b, b]$ and b is set via `de_settings.par_b`.

2. (**Accept/Reject Step**) Define

$$\alpha = \min\left\{1, K(\theta^{(*)}|X)/K(\theta^{(i)}|X)\right\}$$

   where K denotes the posterior kernel. Then

$$\theta^{(i+1)} = \begin{cases} \theta^{(*)} & \text{with probability } \alpha \\ \theta^{(i)} & \text{else} \end{cases}$$

The algorithm stops when the number of draws reaches `n_burnin_draws` + `n_keep_draws`, and returns the final `n_keep_draws` number of draws (in the form a three-dimensional array).

3.6.2 Function Declarations

bool **de**(const ColVec_t &initial_vals, std::function<fp_t(const ColVec_t &vals_inp, void *target_data)> target_log_kernel, Cube_t &draws_out, void *target_data)

The Differential Evolution MCMC Algorithm.

Parameters

- **initial_vals** – a column vector of initial values.
- **target_log_kernel** – the log posterior kernel function of the target distribution, taking two arguments:
  - `vals_inp` a vector of inputs; and
  - `target_data` additional data passed to the user-provided function.
- **draws_out** – a 3-dimensional array posterior draws, where each matrix represents one draw.
- **target_data** – additional data passed to the user-provided function.

Returns
a boolean value indicating successful completion of the sampling algorithm.

bool **de**(const ColVec_t &initial_vals, std::function<fp_t(const ColVec_t &vals_inp, void *target_data)> target_log_kernel, Cube_t &draws_out, void *target_data, algo_settings_t &settings)

The Differential Evolution MCMC Algorithm.

Parameters

- **initial_vals** – a column vector of initial values.
- **target_log_kernel** – the log posterior kernel function of the target distribution, taking two arguments:
  - `vals_inp` a vector of inputs; and
  - `target_data` additional data passed to the user-provided function.
- **draws_out** – a 3-dimensional array posterior draws, where each matrix represents one draw.
- **target_data** – additional data passed to the user-provided function.
- **settings** – parameters controlling the sampling algorithm.

Returns
a boolean value indicating successful completion of the sampling algorithm.

3.6.3 Control Parameters

The basic control parameters are:

- `size_t de_settings.n_pop`: size of population for each set of draws.
  - Default value: `100`.
- `size_t de_settings.n_burnin_draws`: number of burn-in draws.
- `size_t de_settings.n_keep_draws`: number of draws to keep (post sample burn-in period).
- Upper and lower bounds of the uniform distributions used to generate initial values:
  - `ColVec_t de_settings.initial_lb`: defines the lower bounds of the search space.
  - `ColVec_t de_settings.initial_ub`: defines the upper bounds of the search space.
- `bool vals_bound`: whether the search space of the algorithm is bounded. If `true`, then
  - `ColVec_t lower_bounds`: defines the lower bounds of the search space.
  - `ColVec_t upper_bounds`: defines the upper bounds of the search space.

Additional settings:

- `int de_settings.omp_n_threads`: the number of OpenMP threads to use.
  - Default value: `-1` (use all available threads divided by 2).
- `fp_t de_settings.par_b`: support parameter of the uniform distribution used in the Mutation Step.
  - Default value: `1E-04`.

3.6.4 Examples

Gaussian Mean

Code to run this example is given below.

Armadillo (Click to show/hide)

```
#define MCMC_ENABLE_ARMA_WRAPPERS
#include "mcmc.hpp"

struct norm_data_t {
    double sigma;
    arma::vec x;

    double mu_0;
    double sigma_0;
};

double ll_dens(const arma::vec& vals_inp, void* ll_data)
{
    const double pi = arma::datum::pi;

    //

    const double mu = vals_inp(0);

    norm_data_t* dta = reinterpret_cast<norm_data_t*>(ll_data);
    const double sigma = dta->sigma;
    const arma::vec x = dta->x;

    const int n_vals = x.n_rows;

    //

    const double ret = - ((double) n_vals) * (0.5*std::log(2*pi) + std::log(sigma)) -␣
↪arma::accu( arma::pow(x - mu,2) / (2*sigma*sigma) );

    //

    return ret;
}

double log_pr_dens(const arma::vec& vals_inp, void* ll_data)
{
```

(continues on next page)

(continued from previous page)

```
    const double pi = arma::datum::pi;

    //

    norm_data_t* dta = reinterpret_cast< norm_data_t* >(ll_data);

    const double mu_0 = dta->mu_0;
    const double sigma_0 = dta->sigma_0;

    const double x = vals_inp(0);

    const double ret = - 0.5*std::log(2*pi) - std::log(sigma_0) - std::pow(x - mu_0,2) /␣
↪(2*sigma_0*sigma_0);

    return ret;
}

double log_target_dens(const arma::vec& vals_inp, void* ll_data)
{
    return ll_dens(vals_inp,ll_data) + log_pr_dens(vals_inp,ll_data);
}

int main()
{
    const int n_data = 100;
    const double mu = 2.0;

    norm_data_t dta;
    dta.sigma = 1.0;
    dta.mu_0 = 1.0;
    dta.sigma_0 = 2.0;

    arma::vec x_dta = mu + arma::randn(n_data,1);
    dta.x = x_dta;

    arma::vec initial_val(1);
    initial_val(0) = 1.0;

    //

    mcmc::algo_settings_t settings;

    settings.de_settings.n_burnin_draws = 2000;
    settings.de_settings.n_keep_draws = 2000;

    //

    mcmc::Cube_t draws_out;
    mcmc::de(initial_val, log_target_dens, draws_out, &dta, settings);

    //
```

(continues on next page)

(continued from previous page)

```
    std::cout << "de mean:\n" << arma::mean(draws_out.mat(settings.de_settings.n_keep_
↪draws - 1)) << std::endl;
    std::cout << "acceptance rate: " << static_cast<double>(settings.de_settings.n_
↪accept_draws) / (settings.de_settings.n_keep_draws * settings.de_settings.n_pop) <<␣
↪std::endl;

    //

    return 0;
}
```

Eigen (Click to show/hide)

```
#define MCMC_ENABLE_EIGEN_WRAPPERS
#include "mcmc.hpp"

inline
Eigen::VectorXd
eigen_randn_colvec(size_t nr)
{
    static std::mt19937 gen{ std::random_device{}() };
    static std::normal_distribution<> dist;

    return Eigen::VectorXd{ nr }.unaryExpr([&](double x) { (void)(x); return dist(gen); }
↪);
}

struct norm_data_t {
    double sigma;
    Eigen::VectorXd x;

    double mu_0;
    double sigma_0;
};

double ll_dens(const Eigen::VectorXd& vals_inp, void* ll_data)
{
    const double pi = 3.14159265358979;

    //

    const double mu = vals_inp(0);

    norm_data_t* dta = reinterpret_cast<norm_data_t*>(ll_data);
    const double sigma = dta->sigma;
    const Eigen::VectorXd x = dta->x;

    const int n_vals = x.size();

    //

    const double ret = - n_vals * (0.5 * std::log(2*pi) + std::log(sigma)) - (x.array() -
```

(continues on next page)

(continued from previous page)

```
↪ mu).pow(2).sum() / (2*sigma*sigma);

    //

    return ret;
}

double log_pr_dens(const Eigen::VectorXd& vals_inp, void* ll_data)
{
    const double pi = 3.14159265358979;

    //

    norm_data_t* dta = reinterpret_cast< norm_data_t* >(ll_data);

    const double mu_0 = dta->mu_0;
    const double sigma_0 = dta->sigma_0;

    const double x = vals_inp(0);

    const double ret = - 0.5*std::log(2*pi) - std::log(sigma_0) - std::pow(x - mu_0,2) /␣
↪(2*sigma_0*sigma_0);

    return ret;
}

double log_target_dens(const Eigen::VectorXd& vals_inp, void* ll_data)
{
    return ll_dens(vals_inp,ll_data) + log_pr_dens(vals_inp,ll_data);
}

int main()
{
    const int n_data = 100;
    const double mu = 2.0;

    norm_data_t dta;
    dta.sigma = 1.0;
    dta.mu_0 = 1.0;
    dta.sigma_0 = 2.0;

    Eigen::VectorXd x_dta = mu + eigen_randn_colvec(n_data).array();
    dta.x = x_dta;

    Eigen::VectorXd initial_val(1);
    initial_val(0) = 1.0;

    //

    mcmc::algo_settings_t settings;

    settings.de_settings.n_burnin_draws = 2000;
```

(continues on next page)

(continued from previous page)

```
    settings.de_settings.n_keep_draws = 2000;

    //

    mcmc::Cube_t draws_out;
    mcmc::de(initial_val, log_target_dens, draws_out, &dta, settings);

    //

    std::cout << "de mean:\n" << draws_out.mat(settings.de_settings.n_keep_draws - 1).
↪colwise().mean() << std::endl;
    std::cout << "acceptance rate: " << static_cast<double>(settings.de_settings.n_
↪accept_draws) / (settings.de_settings.n_keep_draws * settings.de_settings.n_pop) <<␣
↪std::endl;

    //

    return 0;
}
```

3.7 Hamiltonian Monte Carlo

Table of contents

3.7.1 Algorithm Description

The Hamiltonian Monte Carlo (HMC) algorithm is a Markov Chain Monte Carlo method based on principles of Hamiltonian Dynamics.

Let $\theta^{(i)}$ denote a d-dimensional vector of stored values at stage i of the algorithm. The HMC algorithm proceeds in three steps.

1. (**Initialization**) Sample $p^{(i)} \sim N(0, \mathbf{M})$, and set: $\theta^{(*)} = \theta^{(i)}$ and $p^{(*)} = p^{(i)}$.
2. (**Leapfrog Steps**) **for** $k \in \{1, \ldots,$ `n_leap_steps` $\}$ **do**:
   i. Momentum Update Half-Step.

$$p^{(*)} = p^{(*)} + \epsilon \times \nabla_\theta \ln K(\theta^{(*)}|X)/2$$

where K denotes the posterior kernel function and ϵ is a scaling value set via `hmc_settings.step_size`.

ii. Position Update Step.

$$\theta^{(*)} = \theta^{(*)} + \epsilon \times \mathbf{M}^{-1} p^{(*)}$$

where $\mathbf{M}$ is a pre-conditioning matrix set via `hmc_settings.precond_mat`.

iii. Momentum Update Half-Step.

$$p^{(*)} = p^{(*)} + \epsilon \times \nabla_\theta \ln K(\theta^{(*)}|X)/2$$

3. (**Accept/Reject Step**) Denote the Hamiltonian by

$$H(\theta, p) := \frac{1}{2} \log\left\{(2\pi)^d |\mathbf{M}|\right\} + \frac{1}{2} p^\top \mathbf{M}^{-1} p - \ln K(\theta|X)$$

and define

$$\alpha = \min\left\{1, \exp(H(\theta^{(i)}, p^{(i)}) - H(\theta^{(*)}, p^{(*)}))\right\}$$

Then

$$\theta^{(i+1)} = \begin{cases} \theta^{(*)} & \text{with probability } \alpha \\ \theta^{(i)} & \text{else} \end{cases}$$

The algorithm stops when the number of draws reaches `n_burnin_draws` + `n_keep_draws`, and returns the final `n_keep_draws` number of draws.

3.7.2 Function Declarations

bool **hmc**(const ColVec_t &initial_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *target_data)> target_log_kernel, Mat_t &draws_out, void *target_data)

The Hamiltonian Monte Carlo (HMC) MCMC Algorithm.

Parameters

- **initial_vals** – a column vector of initial values.
- **target_log_kernel** – the log posterior kernel function of the target distribution, taking three arguments:
  - `vals_inp` a vector of inputs; and
  - `grad_out` a vector to store the gradient; and
  - `target_data` additional data passed to the user-provided function.

- **draws_out** – a matrix of posterior draws, where each row represents one draw.
- **target_data** – additional data passed to the user-provided function.

Returns
a boolean value indicating successful completion of the algorithm.

bool **hmc**(const ColVec_t &initial_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *target_data)> target_log_kernel, Mat_t &draws_out, void *target_data, algo_settings_t &settings)

The Hamiltonian Monte Carlo (HMC) MCMC Algorithm.

Parameters

- **initial_vals** – a column vector of initial values.
- **target_log_kernel** – the log posterior kernel function of the target distribution, taking three arguments:
  - `vals_inp` a vector of inputs; and
  - `grad_out` a vector to store the gradient; and
  - `target_data` additional data passed to the user-provided function.
- **draws_out** – a matrix of posterior draws, where each row represents one draw.
- **target_data** – additional data passed to the user-provided function.
- **settings** – parameters controlling the MCMC routine.

Returns
a boolean value indicating successful completion of the algorithm.

3.7.3 Control Parameters

The basic control parameters are:

- `size_t hmc_settings.n_burnin_draws`: number of burn-in draws.
- `size_t hmc_settings.n_keep_draws`: number of draws to keep (post sample burn-in period).
- `bool vals_bound`: whether the search space of the algorithm is bounded. If `true`, then
  - `ColVec_t lower_bounds`: defines the lower bounds of the search space.
  - `ColVec_t upper_bounds`: defines the upper bounds of the search space.

Additional settings:

- `int hmc_settings.omp_n_threads`: the number of OpenMP threads to use.
  - Default value: `-1` (use all available threads divided by 2).
- `size_t hmc_settings.n_leap_steps`: the number of leapfrog steps.
  - Default value: 1.
- `fp_t hmc_settings.step_size`: scaling parameter for the leapfrog step.
  - Default value: `1.0`.
- `Mat_t hmc_settings.precond_mat`: preconditioning matrix for the leapfrog step.
  - Default value: a diagonal matrix.

3.7.4 Examples

Gaussian Distribution

Code to run this example is given below.

Armadillo (Click to show/hide)

```
#define MCMC_ENABLE_ARMA_WRAPPERS
#include "mcmc.hpp"

struct norm_data_t {
    arma::vec x;
};

double ll_dens(const arma::vec& vals_inp, arma::vec* grad_out, void* ll_data)
{
    const double pi = arma::datum::pi;

    const double mu    = vals_inp(0);
    const double sigma = vals_inp(1);

    norm_data_t* dta = reinterpret_cast<norm_data_t*>(ll_data);
    const arma::vec x = dta->x;
    const int n_vals = x.n_rows;

    //

    const double ret = - n_vals * (0.5 * std::log(2*pi) + std::log(sigma)) - arma::accu(␣
↪arma::pow(x - mu,2) / (2*sigma*sigma) );

    //

    if (grad_out) {
        grad_out->set_size(2,1);

        //

        const double m_1 = arma::accu(x - mu);
        const double m_2 = arma::accu( arma::pow(x - mu,2) );

        (*grad_out)(0,0) = m_1 / (sigma*sigma);
        (*grad_out)(1,0) = (m_2 / (sigma*sigma*sigma)) - ((double) n_vals) / sigma;
    }

    //

    return ret;
}

double log_target_dens(const arma::vec& vals_inp, arma::vec* grad_out, void* ll_data)
{
    return ll_dens(vals_inp,grad_out,ll_data);
}
```

(continues on next page)

(continued from previous page)

```
int main()
{
    const int n_data = 1000;

    const double mu = 2.0;
    const double sigma = 2.0;

    norm_data_t dta;

    arma::vec x_dta = mu + sigma * arma::randn(n_data,1);
    dta.x = x_dta;

    arma::vec initial_val(2);
    initial_val(0) = mu + 1; // mu
    initial_val(1) = sigma + 1; // sigma

    mcmc::algo_settings_t settings;

    settings.hmc_settings.step_size = 0.08;
    settings.hmc_settings.n_burnin_draws = 2000;
    settings.hmc_settings.n_keep_draws = 2000;

    arma::mat draws_out;
    mcmc::hmc(initial_val, log_target_dens, draws_out, &dta, settings);

    //

    std::cout << "hmc mean:\n" << arma::mean(draws_out) << std::endl;
    std::cout << "acceptance rate: " << static_cast<double>(settings.hmc_settings.n_
↪accept_draws) / settings.hmc_settings.n_keep_draws << std::endl;

    //

    return 0;
}
```

Eigen (Click to show/hide)

```
#define MCMC_ENABLE_EIGEN_WRAPPERS
#include "mcmc.hpp"

inline
Eigen::VectorXd
eigen_randn_colvec(size_t nr)
{
    static std::mt19937 gen{ std::random_device{}() };
    static std::normal_distribution<> dist;

    return Eigen::VectorXd{ nr }.unaryExpr([&](double x) { (void)(x); return dist(gen); }
↪);
}
```

(continues on next page)

(continued from previous page)

```
struct norm_data_t {
    Eigen::VectorXd x;
};

double ll_dens(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void* ll_data)
{
    const double pi = 3.14159265358979;

    const double mu    = vals_inp(0);
    const double sigma = vals_inp(1);

    norm_data_t* dta = reinterpret_cast<norm_data_t*>(ll_data);
    const Eigen::VectorXd x = dta->x;
    const int n_vals = x.size();

    //

    const double ret = - n_vals * (0.5 * std::log(2*pi) + std::log(sigma)) - (x.array() -
↪ mu).pow(2).sum() / (2*sigma*sigma);

    //

    if (grad_out) {
        grad_out->resize(2,1);

        //

        const double m_1 = (x.array() - mu).sum();
        const double m_2 = (x.array() - mu).pow(2).sum();

        (*grad_out)(0,0) = m_1 / (sigma*sigma);
        (*grad_out)(1,0) = (m_2 / (sigma*sigma*sigma)) - ((double) n_vals) / sigma;
    }

    //

    return ret;
}

double log_target_dens(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void*␣
↪ll_data)
{
    return ll_dens(vals_inp,grad_out,ll_data);
}

int main()
{
    const int n_data = 1000;

    const double mu = 2.0;
    const double sigma = 2.0;
```

(continues on next page)

(continued from previous page)

```
    norm_data_t dta;

    Eigen::VectorXd x_dta = mu + sigma * eigen_randn_colvec(n_data).array();
    dta.x = x_dta;

    Eigen::VectorXd initial_val(2);
    initial_val(0) = mu + 1; // mu
    initial_val(1) = sigma + 1; // sigma

    mcmc::algo_settings_t settings;

    settings.hmc_settings.step_size = 0.08;
    settings.hmc_settings.n_burnin_draws = 2000;
    settings.hmc_settings.n_keep_draws = 2000;

    //

    Eigen::MatrixXd draws_out;
    mcmc::hmc(initial_val, log_target_dens, draws_out, &dta, settings);

    //

    std::cout << "hmc mean:\n" << draws_out.colwise().mean() << std::endl;
    std::cout << "acceptance rate: " << static_cast<double>(settings.hmc_settings.n_
↪accept_draws) / settings.hmc_settings.n_keep_draws << std::endl;

    //

    return 0;
}
```

3.8 Metropolis-adjusted Langevin Algorithm

Table of contents

3.8.1 Algorithm Description

The Metropolis-adjusted Langevin algorithm (MALA) extends the Random Walk Metropolis-Hasting algorithm by generating proposal draws via Langevin diffusions.

Let $\theta^{(i)}$ denote a d-dimensional vector of stored values at stage i of the algorithm. MALA proceeds in two steps.

1. (**Proposal Step**) Let

$$\mu(\theta^{(i)}) := \theta^{(i)} + \frac{\epsilon^2}{2} \times \mathbf{M} \left[\nabla_\theta \ln K(\theta^{(i)}|X) \right]$$

   where K denotes the posterior kernel function; ∇_θ denotes the gradient operator; $\mathbf{M}$ is a pre-conditioning matrix, set via `mala_settings.precond_mat`; and ϵ is a scaling value, set via `mala_settings.step_size`.

   Generate a proposal draw, $\theta^{(*)}$. using:

$$\theta^{(*)} = \mu(\theta^{(i)}) + c \times \mathbf{M}^{1/2} W$$

   where $W \sim N(0, I_d)$.

2. (**Accept/Reject Step**) Denote the proposal density by $q(\theta^{(*)}|\theta^{(i)}) := \phi(\theta^{(*)}; \mu(\theta^{(i)}), \epsilon^2 \mathbf{M})$ and let

$$\alpha = \min \left\{ 1, [K(\theta^{(*)}|X) q(\theta^{(i)}|\theta^{(*)})] / [K(\theta^{(i)}|X) q(\theta^{(*)}|\theta^{(i)})] \right\}$$

   Then

$$\theta^{(i+1)} = \begin{cases} \theta^{(*)} & \text{with probability } \alpha \\ \theta^{(i)} & \text{else} \end{cases}$$

The algorithm stops when the number of draws reaches `n_burnin_draws` + `n_keep_draws`, and returns the final `n_keep_draws` number of draws.

3.8.2 Function Declarations

bool **mala**(const ColVec_t &initial_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *target_data)> target_log_kernel, Mat_t &draws_out, void *target_data)

The Metropolis-adjusted Langevin Algorithm (MALA)

Parameters

- **initial_vals** – a column vector of initial values.
- **target_log_kernel** – the log posterior kernel function of the target distribution, taking three arguments:
  - `vals_inp` a vector of inputs; and
  - `grad_out` a vector to store the gradient; and
  - `target_data` additional data passed to the user-provided function.
- **draws_out** – a matrix of posterior draws, where each row represents one draw.

- **target_data** – additional data passed to the user-provided function.

Returns
a boolean value indicating successful completion of the algorithm.

bool **mala**(const ColVec_t &initial_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *target_data)> target_log_kernel, Mat_t &draws_out, void *target_data, algo_settings_t &settings)

The Metropolis-adjusted Langevin Algorithm (MALA)

Parameters

- **initial_vals** – a column vector of initial values.
- **target_log_kernel** – the log posterior kernel function of the target distribution, taking three arguments:
  - `vals_inp` a vector of inputs; and
  - `grad_out` a vector to store the gradient; and
  - `target_data` additional data passed to the user-provided function.
- **draws_out** – a matrix of posterior draws, where each row represents one draw.
- **target_data** – additional data passed to the user-provided function.
- **settings** – parameters controlling the MCMC routine.

Returns
a boolean value indicating successful completion of the algorithm.

3.8.3 Control Parameters

The basic control parameters are:

- `size_t mala_settings.n_burnin_draws`: number of burn-in draws.
- `size_t mala_settings.n_keep_draws`: number of draws to keep (post sample burn-in period).
- `bool vals_bound`: whether the search space of the algorithm is bounded. If `true`, then
  - `ColVec_t lower_bounds`: defines the lower bounds of the search space.
  - `ColVec_t upper_bounds`: defines the upper bounds of the search space.

Additional settings:

- `int mala_settings.omp_n_threads`: the number of OpenMP threads to use.
  - Default value: `-1` (use all available threads divided by 2).
- `fp_t mala_settings.step_size`: scaling parameter for the proposal step.
  - Default value: `1.0`.
- `Mat_t mala_settings.precond_mat`: preconditioning matrix for the proposal step.
  - Default value: diagonal matrix.

3.8.4 Examples

Gaussian Distribution

Code to run this example is given below.

Armadillo (Click to show/hide)

```
#define MCMC_ENABLE_ARMA_WRAPPERS
#include "mcmc.hpp"

struct norm_data_t {
    arma::vec x;
};

double ll_dens(const arma::vec& vals_inp, arma::vec* grad_out, void* ll_data)
{
    const double pi = arma::datum::pi;

    const double mu    = vals_inp(0);
    const double sigma = vals_inp(1);

    norm_data_t* dta = reinterpret_cast<norm_data_t*>(ll_data);
    const arma::vec x = dta->x;
    const int n_vals = x.n_rows;

    //

    const double ret = - n_vals * (0.5 * std::log(2*pi) + std::log(sigma)) - arma::accu(␣
↪arma::pow(x - mu,2) / (2*sigma*sigma) );

    //

    if (grad_out) {
        grad_out->set_size(2,1);

        //

        const double m_1 = arma::accu(x - mu);
        const double m_2 = arma::accu( arma::pow(x - mu,2) );

        (*grad_out)(0,0) = m_1 / (sigma*sigma);
        (*grad_out)(1,0) = (m_2 / (sigma*sigma*sigma)) - ((double) n_vals) / sigma;
    }

    //

    return ret;
}

double log_target_dens(const arma::vec& vals_inp, arma::vec* grad_out, void* ll_data)
{
    return ll_dens(vals_inp,grad_out,ll_data);
}
```

(continues on next page)

(continued from previous page)

```
int main()
{
    const int n_data = 1000;

    const double mu = 2.0;
    const double sigma = 2.0;

    norm_data_t dta;

    arma::vec x_dta = mu + sigma * arma::randn(n_data,1);
    dta.x = x_dta;

    arma::vec initial_val(2);
    initial_val(0) = mu + 1; // mu
    initial_val(1) = sigma + 1; // sigma

    //

    mcmc::algo_settings_t settings;

    settings.mala_settings.step_size = 0.08;
    settings.mala_settings.n_burnin_draws = 2000;
    settings.mala_settings.n_keep_draws = 2000;

    //

    arma::mat draws_out;
    mcmc::mala(initial_val, log_target_dens, draws_out, &dta, settings);

    //

    std::cout << "mala mean:\n" << arma::mean(draws_out) << std::endl;
    std::cout << "acceptance rate: " << static_cast<double>(settings.mala_settings.n_
↪accept_draws) / settings.mala_settings.n_keep_draws << std::endl;

    //

    return 0;
}
```

Eigen (Click to show/hide)

```
#define MCMC_ENABLE_EIGEN_WRAPPERS
#include "mcmc.hpp"

inline
Eigen::VectorXd
eigen_randn_colvec(size_t nr)
{
    static std::mt19937 gen{ std::random_device{}() };
    static std::normal_distribution<> dist;
```

(continues on next page)

(continued from previous page)

```
    return Eigen::VectorXd{ nr }.unaryExpr([&](double x) { (void)(x); return dist(gen); }
↪);
}

struct norm_data_t {
    Eigen::VectorXd x;
};

double ll_dens(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void* ll_data)
{
    const double pi = 3.14159265358979;

    const double mu    = vals_inp(0);
    const double sigma = vals_inp(1);

    norm_data_t* dta = reinterpret_cast<norm_data_t*>(ll_data);
    const Eigen::VectorXd x = dta->x;
    const int n_vals = x.size();

    //

    const double ret = - n_vals * (0.5 * std::log(2*pi) + std::log(sigma)) - (x.array() -
↪ mu).pow(2).sum() / (2*sigma*sigma);

    //

    if (grad_out) {
        grad_out->resize(2,1);

        //

        const double m_1 = (x.array() - mu).sum();
        const double m_2 = (x.array() - mu).pow(2).sum();

        (*grad_out)(0,0) = m_1 / (sigma*sigma);
        (*grad_out)(1,0) = (m_2 / (sigma*sigma*sigma)) - ((double) n_vals) / sigma;
    }

    //

    return ret;
}

double log_target_dens(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void*␣
↪ll_data)
{
    return ll_dens(vals_inp,grad_out,ll_data);
}

int main()
{
```

(continues on next page)

(continued from previous page)

```
    const int n_data = 1000;

    const double mu = 2.0;
    const double sigma = 2.0;

    norm_data_t dta;

    Eigen::VectorXd x_dta = mu + sigma * eigen_randn_colvec(n_data).array();
    dta.x = x_dta;

    Eigen::VectorXd initial_val(2);
    initial_val(0) = mu + 1; // mu
    initial_val(1) = sigma + 1; // sigma

    mcmc::algo_settings_t settings;

    settings.mala_settings.step_size = 0.08;
    settings.mala_settings.n_burnin_draws = 2000;
    settings.mala_settings.n_keep_draws = 2000;

    //

    Eigen::MatrixXd draws_out;
    mcmc::mala(initial_val, log_target_dens, draws_out, &dta, settings);

    //

    std::cout << "mala mean:\n" << draws_out.colwise().mean() << std::endl;
    std::cout << "acceptance rate: " << static_cast<double>(settings.mala_settings.n_
↪accept_draws) / settings.mala_settings.n_keep_draws << std::endl;

    //

    return 0;
}
```

3.9 No-U-Turn Sampler

Table of contents

3.9.1 Algorithm Description

The No-U-Turn Sampler (NUTS) is a Hamiltonian Monte Carlo (HMC) method that adaptively chooses the number of leapfrog steps and step size. The description below is a modified and shortened version of Algorithm 6 ('No-U-Turn Sampler with Dual Averaging') in Hoffman and Gelman (2014).

Let $\theta^{(i)}$ denote a d-dimensional vector of stored values at stage i of the algorithm, and denote the Hamiltonian by

$$H(\theta, p) := \frac{1}{2}\log\left\{(2\pi)^d|\mathbf{M}|\right\} + \frac{1}{2}p^\top \mathbf{M}^{-1}p - \ln K(\theta|X)$$

where $\mathbf{M}$ is a pre-conditioning matrix. The NUTS algorithm proceeds in 3 steps.

1. (**Initialization**) Sample $p^{(i)} \sim N(0, \mathbf{M})$,

$$u \sim U(0, \exp(H(\theta^{(i-1)}, p^{(i)}))),$$

and set: $n = 1$, $s = 1$,

$$\theta^{(*)} = \theta^{(+)} = \theta^{(-)} = \theta^{(i-1)},$$

and

$$p^{(*)} = p^{(+)} = p^{(-)} = p^{(i)}$$

2. (**Proposal Step**) **while** $s = 1$ **do**:

   i. Sample a direction: $v \sim R$, where R denotes the standard Rademacher distribution (i.e., v takes values in $\{-1, 1\}$ with equal probability).

   ii. **if** $v = -1$:

   update $\theta^{(*)}$, $\theta^{(-)}$, and $p^{(-)}$ by calling the proposal tree-building function (see Hoffman and Gelman (2014) for details).

   else:

   update $\theta^{(*)}$, $\theta^{(+)}$, and $p^{(+)}$ by calling the proposal tree-building function.

   In addition to the proposal and momentum values, n', s', α, and n_α are also updated.

   (Note that, in the tree building process, each tree takes 2^{depth} leapfrog steps with step size $v\epsilon$.)

   iii. **if** $s' = 1$:

$$\theta^{(i)} = \begin{cases} \theta^{(*)} & \text{with probability } n'/n \\ \theta^{(i-1)} & \text{else} \end{cases}$$

   iv. Set: $n = n + n'$, depth = depth + 1, and

$$s = s' \times \mathbf{1}[(\theta^{(+)} - \theta^{(-)}) \cdot p^{(-)} \geq 0] \times \mathbf{1}[(\theta^{(+)} - \theta^{(-)}) \cdot p^{(+)} \geq 0]$$

3. (**Update Step Size**)

   if $i \leq$ `n_adapt_draws`:

update

$$h_i = \left(1 - \frac{1}{i + t_0}\right) \times h_{i-1} + \frac{1}{i + t_0}\left(\delta - \frac{\alpha}{n_\alpha}\right)$$

where: t_0 is set via `nuts_settings.t0_val`; and δ, the target acceptance rate, is set via `nuts_settings.target_accept_rate`.

Set

$$\ln \epsilon = \mu - \frac{\sqrt{i}}{\gamma} \times h_i$$

where $\mu = \ln(10 \times \epsilon_0)$ and γ is set via `nuts_settings.gamma_val`.

$$\ln \bar{\epsilon}_i = i^{-\kappa} \times \ln \epsilon + (1 - i^{-\kappa}) \times \ln \bar{\epsilon}_{i-1}$$

where κ is set via `nuts_settings.kappa_val`.

else:

Set ϵ equal to $\bar{\epsilon}$ from the last adaptation round.

The algorithm stops when the number of draws reaches `n_burnin_draws` + `n_keep_draws`, and returns the final `n_keep_draws` number of draws.

3.9.2 Function Declarations

bool **nuts**(const ColVec_t &initial_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *target_data)> target_log_kernel, Mat_t &draws_out, void *target_data)

The No-U-Turn Sampler (NUTS) MCMC Algorithm.

Parameters

- **initial_vals** – a column vector of initial values.
- **target_log_kernel** – the log posterior kernel function of the target distribution, taking three arguments:
  - `vals_inp` a vector of inputs; and
  - `grad_out` a vector to store the gradient; and
  - `target_data` additional data passed to the user-provided function.
- **draws_out** – a matrix of posterior draws, where each row represents one draw.
- **target_data** – additional data passed to the user-provided function.

Returns

a boolean value indicating successful completion of the algorithm.

bool **nuts**(const ColVec_t &initial_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *target_data)> target_log_kernel, Mat_t &draws_out, void *target_data, algo_settings_t &settings)

The No-U-Turn Sampler (NUTS) MCMC Algorithm.

Parameters

- **initial_vals** – a column vector of initial values.

- **target_log_kernel** – the log posterior kernel function of the target distribution, taking three arguments:
  - vals_inp a vector of inputs; and
  - grad_out a vector to store the gradient; and
  - target_data additional data passed to the user-provided function.
- **draws_out** – a matrix of posterior draws, where each row represents one draw.
- **target_data** – additional data passed to the user-provided function.
- **settings** – parameters controlling the MCMC routine.

Returns
a boolean value indicating successful completion of the algorithm.

3.9.3 Control Parameters

The basic control parameters are:

- size_t nuts_settings.n_burnin_draws: number of burn-in draws.
- size_t nuts_settings.n_keep_draws: number of draws to keep (post sample burn-in period).
- bool vals_bound: whether the search space of the algorithm is bounded. If true, then
  - ColVec_t lower_bounds: defines the lower bounds of the search space.
  - ColVec_t upper_bounds: defines the upper bounds of the search space.

Additional settings:

- int nuts_settings.omp_n_threads: the number of OpenMP threads to use.
  - Default value: -1 (use all available threads divided by 2).
- size_t nuts_settings.n_adapt_draws: the number of draws to use when adaptively setting the step size (ϵ).
  - Default value: 1000.
- fp_t nuts_settings.target_accept_rate: the target acceptance rate for the MCMC chain.
  - Default value: 0.55.
- size_t nuts_settings.max_tree_depth: maximum tree depth for build tree function.
  - Default value: 10.
- fp_t nuts_settings.gamma_val: the tuning parameter γ, used when updating the step size (ϵ).
  - Default value: 0.05.
- fp_t nuts_settings.t0_val: the tuning parameter t_0, used when updating the step size (ϵ).
  - Default value: 10.
- fp_t nuts_settings.kappa_val: the tuning parameter κ, used when updating the step size (ϵ).
  - Default value: 0.75.
- Mat_t nuts_settings.precond_mat: preconditioning matrix for the leapfrog step.
  - Default value: a diagonal matrix.

3.9.4 Examples

Gaussian Distribution

Code to run this example is given below.

Armadillo (Click to show/hide)

```
#define MCMC_ENABLE_ARMA_WRAPPERS
#include "mcmc.hpp"

struct norm_data_t {
    arma::vec x;
};

double ll_dens(const arma::vec& vals_inp, arma::vec* grad_out, void* ll_data)
{
    const double pi = arma::datum::pi;

    const double mu    = vals_inp(0);
    const double sigma = vals_inp(1);

    norm_data_t* dta = reinterpret_cast<norm_data_t*>(ll_data);
    const arma::vec x = dta->x;
    const int n_vals = x.n_rows;

    //

    const double ret = - n_vals * (0.5 * std::log(2*pi) + std::log(sigma)) - arma::accu(␣
↪arma::pow(x - mu,2) / (2*sigma*sigma) );

    //

    if (grad_out) {
        grad_out->set_size(2,1);

        //

        const double m_1 = arma::accu(x - mu);
        const double m_2 = arma::accu( arma::pow(x - mu,2) );

        (*grad_out)(0,0) = m_1 / (sigma*sigma);
        (*grad_out)(1,0) = (m_2 / (sigma*sigma*sigma)) - ((double) n_vals) / sigma;
    }

    //

    return ret;
}
```

(continues on next page)

(continued from previous page)

```
double log_target_dens(const arma::vec& vals_inp, arma::vec* grad_out, void* ll_data)
{
    return ll_dens(vals_inp,grad_out,ll_data);
}

int main()
{
    const int n_data = 1000;

    const double mu = 2.0;
    const double sigma = 2.0;

    norm_data_t dta;

    arma::vec x_dta = mu + sigma * arma::randn(n_data,1);
    dta.x = x_dta;

    arma::vec initial_val(2);
    initial_val(0) = mu + 1; // mu
    initial_val(1) = sigma + 1; // sigma

    mcmc::algo_settings_t settings;

    settings.nuts_settings.n_burnin_draws = 2000;
    settings.nuts_settings.n_keep_draws = 2000;

    arma::mat draws_out;
    mcmc::nuts(initial_val, log_target_dens, draws_out, &dta, settings);

    //

    std::cout << "nuts mean:\n" << arma::mean(draws_out) << std::endl;
    std::cout << "acceptance rate: " << static_cast<double>(settings.nuts_settings.n_
↪accept_draws) / settings.nuts_settings.n_keep_draws << std::endl;

    //

    return 0;
}
```

Eigen (Click to show/hide)

```
#define MCMC_ENABLE_EIGEN_WRAPPERS
#include "mcmc.hpp"

inline
Eigen::VectorXd
eigen_randn_colvec(size_t nr)
{
    static std::mt19937 gen{ std::random_device{}() };
    static std::normal_distribution<> dist;
```

(continues on next page)

(continued from previous page)

```
    return Eigen::VectorXd{ nr }.unaryExpr([&](double x) { (void)(x); return dist(gen); }
↪);
}

struct norm_data_t {
    Eigen::VectorXd x;
};

double ll_dens(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void* ll_data)
{
    const double pi = 3.14159265358979;

    const double mu    = vals_inp(0);
    const double sigma = vals_inp(1);

    norm_data_t* dta = reinterpret_cast<norm_data_t*>(ll_data);
    const Eigen::VectorXd x = dta->x;
    const int n_vals = x.size();

    //

    const double ret = - n_vals * (0.5 * std::log(2*pi) + std::log(sigma)) - (x.array() -
↪ mu).pow(2).sum() / (2*sigma*sigma);

    //

    if (grad_out) {
        grad_out->resize(2,1);

        //

        const double m_1 = (x.array() - mu).sum();
        const double m_2 = (x.array() - mu).pow(2).sum();

        (*grad_out)(0,0) = m_1 / (sigma*sigma);
        (*grad_out)(1,0) = (m_2 / (sigma*sigma*sigma)) - ((double) n_vals) / sigma;
    }

    //

    return ret;
}

double log_target_dens(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void*␣
↪ll_data)
{
    return ll_dens(vals_inp,grad_out,ll_data);
}

int main()
{
    const int n_data = 1000;
```

(continues on next page)

(continued from previous page)

```
    const double mu = 2.0;
    const double sigma = 2.0;

    norm_data_t dta;

    Eigen::VectorXd x_dta = mu + sigma * eigen_randn_colvec(n_data).array();
    dta.x = x_dta;

    Eigen::VectorXd initial_val(2);
    initial_val(0) = mu + 1; // mu
    initial_val(1) = sigma + 1; // sigma

    mcmc::algo_settings_t settings;

    settings.nuts_settings.n_burnin_draws = 2000;
    settings.nuts_settings.n_keep_draws = 2000;

    //

    Eigen::MatrixXd draws_out;
    mcmc::nuts(initial_val, log_target_dens, draws_out, &dta, settings);

    //

    std::cout << "nuts mean:\n" << draws_out.colwise().mean() << std::endl;
    std::cout << "acceptance rate: " << static_cast<double>(settings.nuts_settings.n_
↪accept_draws) / settings.nuts_settings.n_keep_draws << std::endl;

    //

    return 0;
}
```

3.10 Random Walk Metropolis-Hastings

Table of contents

3.10.1 Algorithm Description

The Random Walk Metropolis-Hastings algorithm is the fundamental Markov Chain Monte Carlo method, generating draws from a target posterior distribution via a random walk proposal.

Let $\theta^{(i)}$ denote a d-dimensional vector of stored values at stage i of the algorithm. The RWMH algorithm proceeds in two steps.

1. (**Proposal Step**) Generate a proposal draw, $\theta^{(*)}$, using:

$$\theta^{(*)} = \theta^{(i)} + c \times \Sigma^{1/2} W$$

   where c is a scalar value set via `rwmh_settings.par_scale`; Σ is a matrix set via `rwmh_settings.cov_mat`; and $W \sim N(0, I_d)$.

2. (**Accept/Reject Step**) Define

$$\alpha = \min \left\{ 1, K(\theta^{(*)}|X) / K(\theta^{(i)}|X) \right\}$$

   where K denotes the posterior kernel. Then

$$\theta^{(i+1)} = \begin{cases} \theta^{(*)} & \text{with probability } \alpha \\ \theta^{(i)} & \text{else} \end{cases}$$

The algorithm stops when the number of draws reaches `n_burnin_draws` + `n_keep_draws`, and returns the final `n_keep_draws` number of draws.

3.10.2 Function Declarations

bool **rwmh**(const ColVec_t &initial_vals, std::function<fp_t(const ColVec_t &vals_inp, void *target_data)> target_log_kernel, Mat_t &draws_out, void *target_data)

The Random Walk Metropolis-Hastings MCMC Algorithm.

Parameters

- **initial_vals** – a column vector of initial values.
- **target_log_kernel** – the log posterior kernel function of the target distribution, taking two arguments:
  - `vals_inp` a vector of inputs; and
  - `target_data` additional data passed to the user-provided function.
- **draws_out** – a matrix of posterior draws, where each row represents one draw.
- **target_data** – additional data passed to the user-provided function.

Returns

a boolean value indicating successful completion of the algorithm.

bool **rwmh**(const ColVec_t &initial_vals, std::function<fp_t(const ColVec_t &vals_inp, void *target_data)> target_log_kernel, Mat_t &draws_out, void *target_data, algo_settings_t &settings)

The Random Walk Metropolis-Hastings MCMC Algorithm.

Parameters

- **initial_vals** – a column vector of initial values.
- **target_log_kernel** – the log posterior kernel function of the target distribution, taking two arguments:
  - `vals_inp` a vector of inputs; and
  - `target_data` additional data passed to the user-provided function.
- **draws_out** – a matrix of posterior draws, where each row represents one draw.
- **target_data** – additional data passed to the user-provided function.
- **settings** – parameters controlling the MCMC routine.

Returns

a boolean value indicating successful completion of the algorithm.

3.10.3 Control Parameters

The basic control parameters are:

- `size_t rwmh_settings.n_burnin_draws`: number of burn-in draws.
- `size_t rwmh_settings.n_keep_draws`: number of draws to keep (post sample burn-in period).
- `bool vals_bound`: whether the search space of the algorithm is bounded. If `true`, then
  - `ColVec_t lower_bounds`: defines the lower bounds of the search space.
  - `ColVec_t upper_bounds`: defines the upper bounds of the search space.

Additional settings:

- `int rwmh_settings.omp_n_threads`: the number of OpenMP threads to use.
  - Default value: `-1` (use all available threads divided by 2).
- `fp_t rwmh_settings.par_scale`: scaling parameter for the proposal step.
  - Default value: `1.0`.
- `Mat_t rwmh_settings.cov_mat`: covariance matrix for the proposal step.
  - Default value: diagonal matrix.

3.10.4 Examples

Gaussian Mean

Code to run this example is given below.

Armadillo (Click to show/hide)

```
#define MCMC_ENABLE_ARMA_WRAPPERS
#include "mcmc.hpp"

struct norm_data_t {
    double sigma;
    arma::vec x;

    double mu_0;
    double sigma_0;
};

double ll_dens(const arma::vec& vals_inp, void* ll_data)
{
    const double pi = arma::datum::pi;

    //

    const double mu = vals_inp(0);

    norm_data_t* dta = reinterpret_cast<norm_data_t*>(ll_data);
    const double sigma = dta->sigma;
    const arma::vec x = dta->x;

    const int n_vals = x.n_rows;

    //

    const double ret = - ((double) n_vals) * (0.5*std::log(2*pi) + std::log(sigma)) -␣
↪arma::accu( arma::pow(x - mu,2) / (2*sigma*sigma) );

    //

    return ret;
}

double log_pr_dens(const arma::vec& vals_inp, void* ll_data)
{
    const double pi = arma::datum::pi;

    //

    norm_data_t* dta = reinterpret_cast< norm_data_t* >(ll_data);

    const double mu_0 = dta->mu_0;
    const double sigma_0 = dta->sigma_0;
```

(continues on next page)

(continued from previous page)

```
    const double x = vals_inp(0);

    const double ret = - 0.5*std::log(2*pi) - std::log(sigma_0) - std::pow(x - mu_0,2) /␣
↪(2*sigma_0*sigma_0);

    return ret;
}

double log_target_dens(const arma::vec& vals_inp, void* ll_data)
{
    return ll_dens(vals_inp,ll_data) + log_pr_dens(vals_inp,ll_data);
}

int main()
{
    const int n_data = 100;
    const double mu = 2.0;

    norm_data_t dta;
    dta.sigma = 1.0;
    dta.mu_0 = 1.0;
    dta.sigma_0 = 2.0;

    arma::vec x_dta = mu + arma::randn(n_data,1);
    dta.x = x_dta;

    arma::vec initial_val(1);
    initial_val(0) = 1.0;

    //

    mcmc::algo_settings_t settings;

    settings.rwmh_settings.par_scale = 0.4;
    settings.rwmh_settings.n_burnin_draws = 2000;
    settings.rwmh_settings.n_keep_draws = 2000;

    //

    arma::mat draws_out;
    mcmc::rwmh(initial_val, log_target_dens, draws_out, &dta, settings);

    //

    std::cout << "rwmh mean:\n" << arma::mean(draws_out) << std::endl;
    std::cout << "acceptance rate: " << static_cast<double>(settings.rwmh_settings.n_
↪accept_draws) / settings.rwmh_settings.n_keep_draws << std::endl;

    //

    return 0;
}
```

Eigen (Click to show/hide)

```cpp
#define MCMC_ENABLE_EIGEN_WRAPPERS
#include "mcmc.hpp"

inline
Eigen::VectorXd
eigen_randn_colvec(size_t nr)
{
    static std::mt19937 gen{ std::random_device{}() };
    static std::normal_distribution<> dist;

    return Eigen::VectorXd{ nr }.unaryExpr([&](double x) { (void)(x); return dist(gen); }
↪);
}

struct norm_data_t {
    double sigma;
    Eigen::VectorXd x;

    double mu_0;
    double sigma_0;
};

double ll_dens(const Eigen::VectorXd& vals_inp, void* ll_data)
{
    const double pi = 3.14159265358979;

    //

    const double mu = vals_inp(0);

    norm_data_t* dta = reinterpret_cast<norm_data_t*>(ll_data);
    const double sigma = dta->sigma;
    const Eigen::VectorXd x = dta->x;

    const int n_vals = x.size();

    //

    const double ret = - n_vals * (0.5 * std::log(2*pi) + std::log(sigma)) - (x.array() -
↪ mu).pow(2).sum() / (2*sigma*sigma);

    //

    return ret;
}

double log_pr_dens(const Eigen::VectorXd& vals_inp, void* ll_data)
{
    const double pi = 3.14159265358979;

    //
```

(continues on next page)

(continued from previous page)

```
    norm_data_t* dta = reinterpret_cast< norm_data_t* >(ll_data);

    const double mu_0 = dta->mu_0;
    const double sigma_0 = dta->sigma_0;

    const double x = vals_inp(0);

    const double ret = - 0.5*std::log(2*pi) - std::log(sigma_0) - std::pow(x - mu_0,2) /␣
↪(2*sigma_0*sigma_0);

    return ret;
}

double log_target_dens(const Eigen::VectorXd& vals_inp, void* ll_data)
{
    return ll_dens(vals_inp,ll_data) + log_pr_dens(vals_inp,ll_data);
}

int main()
{
    const int n_data = 100;
    const double mu = 2.0;

    norm_data_t dta;
    dta.sigma = 1.0;
    dta.mu_0 = 1.0;
    dta.sigma_0 = 2.0;

    Eigen::VectorXd x_dta = mu + eigen_randn_colvec(n_data).array();
    dta.x = x_dta;

    Eigen::VectorXd initial_val(1);
    initial_val(0) = 1.0;

    //

    mcmc::algo_settings_t settings;

    settings.rwmh_settings.par_scale = 0.4;
    settings.rwmh_settings.n_burnin_draws = 2000;
    settings.rwmh_settings.n_keep_draws = 2000;

    //

    Eigen::MatrixXd draws_out;
    mcmc::rwmh(initial_val, log_target_dens, draws_out, &dta, settings);

    //

    std::cout << "hmc mean:\n" << draws_out.colwise().mean() << std::endl;
    std::cout << "acceptance rate: " << static_cast<double>(settings.rwmh_settings.n_
```

(continues on next page)

(continued from previous page)

```
↪accept_draws) / settings.rwmh_settings.n_keep_draws << std::endl;

    //

    return 0;
}
```

3.11 Riemannian Manifold HMC

Table of contents

3.11.1 Algorithm Description

The Riemannian Manifold Hamiltonian Monte Carlo (RM-HMC) algorithm is a Markov Chain Monte Carlo method based on principles of Hamiltonian Dynamics.

Let $\theta^{(i)}$ denote a d-dimensional vector of stored values at stage i of the algorithm, and denote the Hamiltonian by

$$H\left\{\theta, p\right\} := -\ln K(\theta|X) + \frac{1}{2}\log\left\{(2\pi)^d|\mathbf{G}(\theta)|\right\} + \frac{1}{2}p^\top \mathbf{G}^{-1}(\theta)p$$

where K denotes the posterior kernel function and $\mathbf{G}$ denotes the metric tensor function.

The RM-HMC algorithm proceeds in three steps.

1. (**Initialization**) Sample $p^{(i)} \sim N(0, \mathbf{G}(\theta^{(i)}))$, and set: $\theta^{(*)} = \theta^{(i)}$ and $p^{(*)} = p^{(i)}$.
2. (**Leapfrog Steps**) **for** $k \in \{1, \ldots,$ `n_leap_steps` $\}$ **do**:
   i. Initialization. Set $\theta_o^{(*)} = \theta^{(i)}$ and $p_o^{(*)} = p^{(i)}$
   ii. Momentum Update Half-Step. **for** $l \in \{1, \ldots,$ `n_fp_steps` $\}$ **do**:

$$p_h^{(*)} = p_o^{(*)} - \frac{\epsilon}{2} \times \nabla_\theta H\left\{\theta_o^{(*)}, p_h^{(*)}\right\}$$

   where the subscript h denotes a half-step. (Notice that $p_h^{(*)}$ appears on both sides of this expression.)

iii. Position Update Step. **for** $l \in \{1, \ldots,$ `n_fp_steps` $\}$ **do**:

$$\theta^{(*)} = \theta_o^{(*)} + \frac{\epsilon}{2} \times \left[\nabla_p H \left\{ \theta_o^{(*)}, p_h^{(*)} \right\} + \nabla_p H \left\{ \theta^{(*)}, p_h^{(*)} \right\} \right]$$

(Notice that $\theta^{(*)}$ appears on both sides of this expression.)

iv. Momentum Update Half-Step.

$$p^{(*)} = p_h^{(*)} + \epsilon \times \nabla_\theta H \left\{ \theta^{(*)}, p_h^{(*)} \right\}$$

3. (**Accept/Reject Step**) Define

$$\alpha = \min \left\{ 1, \exp \left(H \left\{ \theta^{(i)}, p^{(i)} \right\} - H \left\{ \theta^{(*)}, p^{(*)} \right\} \right) \right\}$$

Then

$$\theta^{(i+1)} = \begin{cases} \theta^{(*)} & \text{with probability } \alpha \\ \theta^{(i)} & \text{else} \end{cases}$$

The algorithm stops when the number of draws reaches `n_burnin_draws` + `n_keep_draws`, and returns the final `n_keep_draws` number of draws.

3.11.2 Function Declarations

bool **rmhmc**(const ColVec_t &initial_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *target_data)> target_log_kernel, std::function<Mat_t(const ColVec_t &vals_inp, Cube_t *tensor_deriv_out, void *tensor_data)> tensor_fn, Mat_t &draws_out, void *target_data, void *tensor_data)

The Riemannian Manifold Hamiltonian Monte Carlo (RM-HMC) MCMC Algorithm.

Parameters

- **initial_vals** – a column vector of initial values.
- **target_log_kernel** – the log posterior kernel function of the target distribution, taking three arguments:
  - `vals_inp` a vector of inputs; and
  - `grad_out` a vector to store the gradient; and
  - `target_data` additional data passed to the user-provided function.
- **tensor_fn** – the manifold tensor function, taking three arguments:
  - `vals_inp` a vector of inputs; and
  - `tensor_deriv_out` a 3-dimensional array to store the tensor derivatives; and
  - `tensor_data` additional data passed to the user-provided function.
- **draws_out** – a matrix of posterior draws, where each row represents one draw.

- **target_data** – additional data passed to the user-provided log kernel function.
- **tensor_data** – additional data passed to the user-provided tensor function.

Returns
a boolean value indicating successful completion of the algorithm.

bool **rmhmc**(const ColVec_t &initial_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *target_data)> target_log_kernel, std::function<Mat_t(const ColVec_t &vals_inp, Cube_t *tensor_deriv_out, void *tensor_data)> tensor_fn, Mat_t &draws_out, void *target_data, void *tensor_data, algo_settings_t &settings)

The Riemannian Manifold Hamiltonian Monte Carlo (RM-HMC) MCMC Algorithm.

Parameters

- **initial_vals** – a column vector of initial values.
- **target_log_kernel** – the log posterior kernel function of the target distribution, taking three arguments:
  - `vals_inp` a vector of inputs; and
  - `grad_out` a vector to store the gradient; and
  - `target_data` additional data passed to the user-provided function.
- **tensor_fn** – the manifold tensor function, taking three arguments:
  - `vals_inp` a vector of inputs; and
  - `tensor_deriv_out` a 3-dimensional array to store the tensor derivatives; and
  - `tensor_data` additional data passed to the user-provided function.
- **draws_out** – a matrix of posterior draws, where each row represents one draw.
- **target_data** – additional data passed to the user-provided log kernel function.
- **tensor_data** – additional data passed to the user-provided tensor function.
- **settings** – parameters controlling the MCMC routine.

Returns
a boolean value indicating successful completion of the algorithm.

3.11.3 Control Parameters

The basic control parameters are:

- `size_t rmhmc_settings.n_burnin_draws`: number of burn-in draws.
- `size_t rmhmc_settings.n_keep_draws`: number of draws to keep (post sample burn-in period).
- `bool vals_bound`: whether the search space of the algorithm is bounded. If `true`, then
  - `ColVec_t lower_bounds`: defines the lower bounds of the search space.
  - `ColVec_t upper_bounds`: defines the upper bounds of the search space.

Additional settings:

- `int rmhmc_settings.omp_n_threads`: the number of OpenMP threads to use.
  - Default value: `-1` (use all available threads divided by 2).

- `size_t rmhmc_settings.n_leap_steps`: the number of leapfrog steps.
  - Default value: `1`.
- `fp_t rmhmc_settings.step_size`: scaling parameter for the leapfrog step.
  - Default value: `1.0`.
- `Mat_t rmhmc_settings.precond_mat`: preconditioning matrix for the leapfrog step.
  - Default value: diagonal matrix.
- `size_t rmhmc_settings.n_fp_steps`: the number of fixed-point steps.
  - Default value: `5`.

3.11.4 Examples

Gaussian Distribution

Code to run this example is given below.

Armadillo (Click to show/hide)

```
#define MCMC_ENABLE_ARMA_WRAPPERS
#include "mcmc.hpp"

struct norm_data_t {
    arma::vec x;
};

double ll_dens(const arma::vec& vals_inp, arma::vec* grad_out, void* ll_data)
{
    const double pi = arma::datum::pi;

    const double mu    = vals_inp(0);
    const double sigma = vals_inp(1);

    norm_data_t* dta = reinterpret_cast<norm_data_t*>(ll_data);
    const arma::vec x = dta->x;
    const int n_vals = x.n_rows;

    //

    const double ret = - n_vals * (0.5 * std::log(2*pi) + std::log(sigma)) - arma::accu(␣
↪arma::pow(x - mu,2) / (2*sigma*sigma) );

    //

    if (grad_out) {
        grad_out->set_size(2,1);

        //

        const double m_1 = arma::accu(x - mu);
```

(continues on next page)

(continued from previous page)

```
        const double m_2 = arma::accu( arma::pow(x - mu,2) );

        (*grad_out)(0,0) = m_1 / (sigma*sigma);
        (*grad_out)(1,0) = (m_2 / (sigma*sigma*sigma)) - ((double) n_vals) / sigma;
    }

    //

    return ret;
}

arma::mat tensor_fn(const arma::vec& vals_inp, mcmc::Cube_t* tensor_deriv_out, void*␣
↪tensor_data)
{
    // const double mu    = vals_inp(0);
    const double sigma = vals_inp(1);

    norm_data_t* dta = reinterpret_cast<norm_data_t*>(tensor_data);

    const int n_vals = dta->x.n_rows;

    //

    const double sigma_sq = sigma*sigma;

    arma::mat tensor_out = arma::zeros(2,2);

    tensor_out(0,0) = ((double) n_vals) / sigma_sq;
    tensor_out(1,1) = 2.0 * ((double) n_vals) / sigma_sq;

    //

    if (tensor_deriv_out) {
        tensor_deriv_out->setZero(2,2,2);

        //

        // tensor_deriv_out->mat(0).setZero();

        tensor_deriv_out->mat(1) = - 2.0 * tensor_out / sigma;
    }

    //

    return tensor_out;
}

double log_target_dens(const arma::vec& vals_inp, arma::vec* grad_out, void* ll_data)
{
    return ll_dens(vals_inp,grad_out,ll_data);
}
```

(continues on next page)

(continued from previous page)

```
int main()
{
    const int n_data = 1000;

    const double mu = 2.0;
    const double sigma = 2.0;

    norm_data_t dta;

    arma::vec x_dta = mu + sigma * arma::randn(n_data,1);
    dta.x = x_dta;

    arma::vec initial_val(2);
    initial_val(0) = mu + 1; // mu
    initial_val(1) = sigma + 1; // sigma

    //

    mcmc::algo_settings_t settings;

    settings.rmhmc_settings.step_size = 0.2;
    settings.rmhmc_settings.n_burnin_draws = 2000;
    settings.rmhmc_settings.n_keep_draws = 2000;

    //

    arma::mat draws_out;
    mcmc::rmhmc(initial_val, log_target_dens, tensor_fn, draws_out, &dta, &dta,␣
↪settings);

    //

    std::cout << "rmhmc mean:\n" << arma::mean(draws_out) << std::endl;
    std::cout << "acceptance rate: " << static_cast<double>(settings.rmhmc_settings.n_
↪accept_draws) / settings.rmhmc_settings.n_keep_draws << std::endl;

    //

    return 0;
}
```

Eigen (Click to show/hide)

```
#define MCMC_ENABLE_EIGEN_WRAPPERS
#include "mcmc.hpp"

inline
Eigen::VectorXd
eigen_randn_colvec(size_t nr)
{
    static std::mt19937 gen{ std::random_device{}() };
    static std::normal_distribution<> dist;
```

(continues on next page)

(continued from previous page)

```
    return Eigen::VectorXd{ nr }.unaryExpr([&](double x) { (void)(x); return dist(gen); }
↪);
}

struct norm_data_t {
    Eigen::VectorXd x;
};

double ll_dens(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void* ll_data)
{
    const double pi = 3.14159265358979;

    const double mu    = vals_inp(0);
    const double sigma = vals_inp(1);

    norm_data_t* dta = reinterpret_cast<norm_data_t*>(ll_data);
    const Eigen::VectorXd x = dta->x;
    const int n_vals = x.size();

    //

    const double ret = - n_vals * (0.5 * std::log(2*pi) + std::log(sigma)) - (x.array() -
↪ mu).pow(2).sum() / (2*sigma*sigma);

    //

    if (grad_out) {
        grad_out->resize(2,1);

        //

        const double m_1 = (x.array() - mu).sum();
        const double m_2 = (x.array() - mu).pow(2).sum();

        (*grad_out)(0,0) = m_1 / (sigma*sigma);
        (*grad_out)(1,0) = (m_2 / (sigma*sigma*sigma)) - ((double) n_vals) / sigma;
    }

    //

    return ret;
}

Eigen::MatrixXd tensor_fn(const Eigen::VectorXd& vals_inp, mcmc::Cube_t* tensor_deriv_
↪out, void* tensor_data)
{
    // const double mu    = vals_inp(0);
    const double sigma = vals_inp(1);

    norm_data_t* dta = reinterpret_cast<norm_data_t*>(tensor_data);
```

(continues on next page)

(continued from previous page)

```
    const int n_vals = dta->x.size();

    //

    const double sigma_sq = sigma*sigma;

    Eigen::MatrixXd tensor_out = Eigen::MatrixXd::Zero(2,2);

    tensor_out(0,0) = ((double) n_vals) / sigma_sq;
    tensor_out(1,1) = 2.0 * ((double) n_vals) / sigma_sq;

    //

    if (tensor_deriv_out) {
        tensor_deriv_out->setZero(2,2,2);

        //

        // tensor_deriv_out->mat(0).setZero();

        tensor_deriv_out->mat(1) = - 2.0 * tensor_out / sigma;
    }

    //

    return tensor_out;
}

double log_target_dens(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void*␣
↪ll_data)
{
    return ll_dens(vals_inp,grad_out,ll_data);
}

int main()
{
    const int n_data = 1000;

    const double mu = 2.0;
    const double sigma = 2.0;

    norm_data_t dta;

    Eigen::VectorXd x_dta = mu + sigma * eigen_randn_colvec(n_data).array();
    dta.x = x_dta;

    Eigen::VectorXd initial_val(2);
    initial_val(0) = mu + 1; // mu
    initial_val(1) = sigma + 1; // sigma

    mcmc::algo_settings_t settings;
```

(continues on next page)

(continued from previous page)

```
    settings.rmhmc_settings.step_size = 0.2;
    settings.rmhmc_settings.n_burnin_draws = 2000;
    settings.rmhmc_settings.n_keep_draws = 2000;

    //

    Eigen::MatrixXd draws_out;
    mcmc::rmhmc(initial_val, log_target_dens, tensor_fn, draws_out, &dta, &dta,␣
↪settings);

    //

    std::cout << "rmhmc mean:\n" << draws_out.colwise().mean() << std::endl;
    std::cout << "acceptance rate: " << static_cast<double>(settings.rmhmc_settings.n_
↪accept_draws) / settings.rmhmc_settings.n_keep_draws << std::endl;

    //

    return 0;
}
```

3.12 Box Constraints

This section provides implementation details for how MCMCLib handles box constraints.

For a parameter θ_j defined on a bounded interval $[a_j, b_j]$, where $a_j < b_j$, we use the generalized logit transform:

$$g(\theta_j) = \phi_j := \ln\left(\frac{x_j - a_j}{b_j - x_j}\right)$$

with corresponding inverse transform

$$\theta_j = g^{-1}(\phi_j) = \frac{a_j + b_j \exp(\phi_j)}{1 + \exp(\phi_j)}$$

Note that the support of ϕ_j is $\mathbb{R}$.

The log posterior kernel function of the $J \times 1$ vector $\boldsymbol{\theta}$ is given by:

$$K(\boldsymbol{\theta}|Y) = \ln L(Y|\boldsymbol{\theta}) + \ln \pi(\boldsymbol{\theta})$$

where L is the likelihood function of the data Y parametrized by $\boldsymbol{\theta}$, and π is the joint prior distribution over $\boldsymbol{\theta}$. The log posterior kernel function of the transformed parameter vector $\boldsymbol{\phi} = g(\boldsymbol{\theta})$ is then computed as

$$K(\boldsymbol{\phi}|Y) = \ln L(Y|g^{-1}(\boldsymbol{\phi})) + \ln \pi(g^{-1}(\boldsymbol{\phi})) + \ln |J(\boldsymbol{\phi})|$$

where $|J|$ is the modulus of the Jacobian determinant matrix J—that is, the determinant of a matrix with (i, j) elements equal to

$$\frac{\partial [g^{-1}(\phi)]_i}{\partial \phi_j} = \frac{\partial \theta_i}{\partial \phi_j}$$

If the parameters $\boldsymbol{\theta}$ are assumed to be *a-priori* independent, then

$$\pi(\boldsymbol{\theta}) = \pi_1(\theta_1) \cdots \pi_J(\theta_J)$$

Given our specification for g, the Jacobian term will be a diagonal matrix with non-negative elements: the log Jacobian adjustment for parameter j is given by

$$\ln J_{j,j} := \ln \left(\frac{d\theta_j}{d\phi_j} \right) = \ln(b_j - a_j) + \phi_j - 2\ln(1 + \exp(\phi_j))$$

As the determinant of a diagonal matrix is the product of its diagonal elements, the final term in can be computed as a sum:

$$\ln |J(\boldsymbol{\phi})| = \sum_{j=1}^{J} \ln |J_{j,j}| = \sum_{j=1}^{J} [\ln(b_j - a_j) + \phi_j - 2\ln(1 + \exp(\phi_j))]$$

INDEX