pdmphmc - numerical generalized randomized HMC processes for R

Tore Selland Kleppe

2023-07-13

Contents

1 About **7**
- 1.1 What is `pdmphmc`? . . . 7
- 1.2 Why `pdmphmc`? . . . 7
- 1.3 Prerequisites . . . 8

2 `pdmphmc` "Hello World" **9**
- 2.1 Requirements . . . 9
- 2.2 Check your installation . . . 9

3 The model specification **11**
- 3.1 A simple worked example . . . 11

4 Model specification details **15**
- 4.1 `DATA_*` statements . . . 15
- 4.2 The `preProcess()` function . . . 16
- 4.3 The `operator()()` function . . . 16

5 Constraints **23**
- 5.1 Specifying constraints . . . 23
- 5.2 Special constraints . . . 24
- 5.3 Non-linear constraints . . . 25
- 5.4 Worked example . . . 26

6 The `amt`-library **33**
- 6.1 Utilities for symmetric positive definite matrices . . . 34
- 6.2 Univariate continuous distributions . . . 35
- 6.3 Univariate discrete distributions . . . 36
- 6.4 Distributions on unbounded vectors . . . 36
- 6.5 Distributions on symmetric positive definite matrices . . . 37

7 The sampling algorithms **39**
- 7.1 Fixed mass sampler . . . 39
- 7.2 Riemann manifold sampler . . . 40
- 7.3 ODE solvers . . . 41

7.4 Scaling adaptation 41

7.5 Event intensity 42

8 Worked examples **43**

Chapter 1

About

This document provides documentation of the `pdmphmc`(piecewise deterministic Markov process HMC) package.

The package is available on github and is most easily installed via `devtools::install_github("https://github.com/torekleppe/pdmphmc")` (requires the `devtools` package)

The package implements, with some modifications and substantial additions, the methodology of Kleppe [2022a], Kleppe [2022b].

1.1 What is `pdmphmc`?

`pdmphmc` is a system for carrying out probability computations, typically associated with Bayesian statistical modelling. `pdmphmc` consist of

- A very flexible system for specifying Bayesian statistical models using C++ classes.
- An implementation of numerical generalized randomized Hamiltonian Monte Carlo samplers for MCMC-like computations for models specified in the above mentioned system.

1.2 Why `pdmphmc`?

Many packages provides computational methods for Bayesian statistical models. `pdmphmc` is designed to be *computationally fast and stable, even for high-dimensional models and/or models where the posterior distribution exhibits complicated non-linear dependence structures*. Particular emphasis has been on developing and implementing methodology suitable for fitting non-linear hierarchical models, and also models involving hard constraints/restricted domains.

1.3 Prerequisites

- This documentation assumes basic familiarity with the C++ programming language.
- This documentation assumes modest familiarity with the Eigen C++ library, see e.g. https://eigen.tuxfamily.org/dox/group___QuickRefPage.html
- The package relies on the Stan Math Library (https://mc-stan.org/users/interfaces/math) for automatic differentiation. In addition, the complete Stan Math Library (including all probability density functions etc) are also available within the modeling facilities of this package. Hence, some familiarity with the probability densities etc documented in https://mc-stan.org/docs/functions-reference/index.html may be useful.

Chapter 2

pdmphmc "Hello World"

This section assumes that pdmphmc R-package has already been installed.

2.1 Requirements

Other than a working installation of R and R-package rstan (with dependencies), the pdmphmc package requires a working C++ compiler.

- On mac, linux, this is typically available as g++ on the command line. If you get an error message when running e.g. g++ -v in the command line, you need to get the g++ for your system.
- On Windows, testing of pdmphmc is done using the GCC 10/MinGW-w64 compiler toolchain that comes with the Rtools (https://cran.r-project.org/bin/windows/Rtools/), but it should also be noted that it should be possible to use different compilers.

2.2 Check your installation

The simplest way to check your installation is to try to run the testSystem()

```
success <- pdmphmc::testSystem()
```

```
## model name : model
## process type : HMCProcessConstr
## Runge Kutta step type : RKBS32
## Transport map type : diagLinearTM_VARI
## compilation exited successfully
```

```
## model ran successfully
```

If the last line printed reads "model ran successfully", you are good to go.

If not, there is something wrong with the setup for your system. A good starting point for resolving compilation issues is first check that your installation of package `rstan` is really working, see https://github.com/stan-dev/rstan/wiki/RStan-Getting-Started for details.

Chapter 3

The model specification

Models are specified as a C++ `class`/`struct` that should have the following signature

```
struct model{
  // data statements
  void preProcess(){
    // preprocess (called only once) here
  }

  template < class varType, class tensorType, bool storeNames>
  void operator()(amt::amtModel<varType,tensorType,storeNames> &model__){
    // specifcation of model target density ++ here
  }
}; // end of struct
```

The name of the `class`/`struct` is arbitrary

3.1 A simple worked example

Now consider specifying the model $y_i \sim$ iid $N(\mu, \exp(\lambda))$, $i = 1, \ldots, n$ (with flat priors on μ, λ) for a given data set $\mathbf{y}$. We wish to sample from the posterior distribution of (μ, λ), and in addition, get samples from the posterior distribution of $\sigma = \exp(0.5\lambda)$.

The model class would look something like

```
using namespace amt;
struct model{

  DATA_VECTOR(y); // data to be passed from R
```

```
  void preProcess(){} // not used in this example

  template < class varType, class tensorType, bool storeNames>
  void operator()(amt::amtModel<varType,tensorType,storeNames> &model__){

    PARAMETER_SCALAR(mu); // parameter (sampled quantity)
    PARAMETER_SCALAR(lambda); // parameter (sampled quantity)
    // note; all variables depending on the parameters must be of type varType
    varType sigma = exp(0.5*lambda);
    // add data likelihood to the model object
    model__+=normal_ld(y,mu,sigma);
    // add sigma as a quantity to produce samples of
    model__.generated(sigma,"sigma");
  }
}; // end of struct
```

When stored in file `basic_model.cpp`, the above model specification may be compiled using

```
model <- pdmphmc::build("basic_model.cpp")
```

```
## model name : model

## process type : HMCProcessConstr

## Runge Kutta step type : RKDP54

## Transport map type : diagLinearTM_VARI

## compilation exited successfully
```

Then let's simulate some data and run the model:

```
set.seed(123)
y <- rnorm(10) # y is iid N(0,1) with n=10
fit <- pdmphmc::run(model,data=list(y=y))
```

Finally, get a summary of the sampled- and generated quantities, based on `rstan::monitor`:

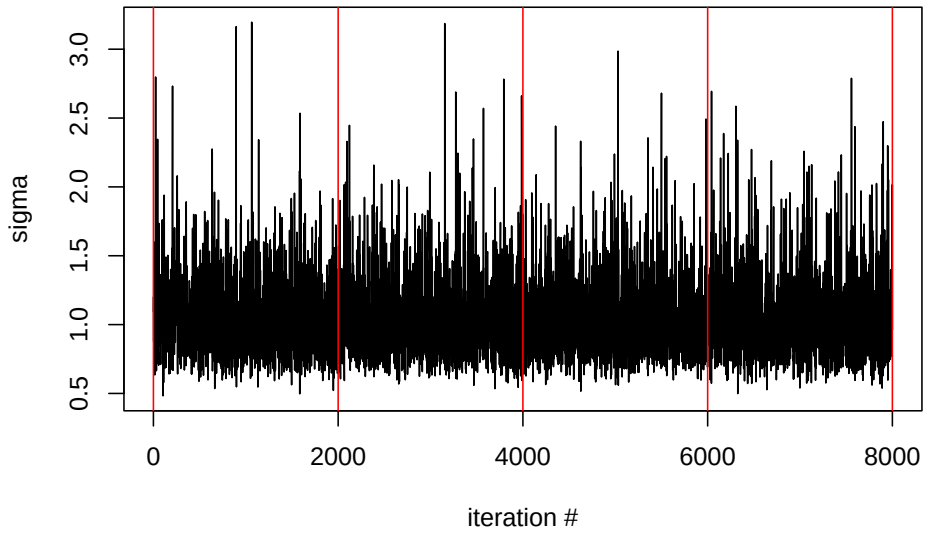
```
fit
```

```
## run output for model: model

## # of chains : 4

## Summary based on discrete samples:

##         mean se_mean    sd n_eff  Rhat
## mu     0.079   0.006 0.340  2804 1.001
## lambda 0.025   0.010 0.511  2746 1.003
```

```
## sigma  1.048   0.006 0.293  2763 1.003

## summary based on integrated samples:

##    estimate se_estimate n_eff  Rhat
## V1    1.049       0.004  1794 1.001

## NOTE: integrated samples do NOT reflect the complete target distibution, only indicated moment
```

Further functions exist for inspecting the output, e.g. trace plots

```
pdmphmc::trace.plot(fit,"sigma")
```



In the next chapter, a more detailed summary of the possibilities of the model specification is given.

Chapter 4

Model specification details

4.1 `DATA_*` statements

The facility for passing data from `R` to the model is based on the `DATA_*` preprocessor macros. Currently there are 6 types of these, corresponding to different types of C++ storage in the model:

```
DATA_DOUBLE(<dataname>); // C++ storage: double
DATA_INT(<dataname>); // C++ storage: int
DATA_VECTOR(<dataname>); // C++ storage: Eigen::VectorXd
DATA_IVECTOR(<dataname>); // C++ storage: Eigen::VectorXi
DATA_MATRIX(<dataname>); // C++ storage: Eigen::MatrixXd
DATA_IMATRIX(<dataname>); // C++ storage: Eigen::MatrixXi
```

After putting one or multiple `DATA_*` statements at the **start of the model class**, a object with name `<dataname>` and the indicated type will be available throughout the class.

The objects initiated with `DATA_*` statements are filled using the `data` argument of the `pdmphmc::run` function. E.g. if the model class contains

```
struct model{
  DATA_DOUBLE(dd);
  DATA_MATRIX(mat);
  // rest of model spec here
};
```

then calls to `pdmphmc::run` must be done with something like

```
# assuming the model has been built into R-variable "model"
fit <- run(model,data=list(dd=1.23,mat=matrix(rnorm(4),2,2)))
```

The data are read into the C++ program before any sampling etc is done.

4.2 The `preProcess()` function

The `preProcess()` is intended for restructuring data, reading data directly from a file etc.

The pre-process function gets called after data are read into C++/is available as objects `<dataname>`. E.g. continuing the example above, e.g.

```
struct model{
  DATA_DOUBLE(dd);
  DATA_MATRIX(mat);
  double ddHalf_;
  void preProcess(){
    ddHalf_ = 0.5*dd;
  }
  // rest of model spec here
};
```

the variable `ddHalf_` would contain `0.5*1.23` at the time of sampling if the same data as above are provided.

4.3 The `operator()()` function

The `operator()()` function is used for specifying (in a broad sense) the target distribution, say $\pi(\theta)$.

It should have the signature:

```
template < class varType, class tensorType, bool storeNames>
  void operator()(amt::amtModel<varType,tensorType,storeNames> &model__){
  }
```

The particular meaning of the templates etc is not important, but the user should be aware that all parameters, and all quantities depending on parameters should be of type `varType`[1]. `varTypes` could be used more or less as `double` types in C++.

Further, the user should be aware of the `amt::amtModel` object by default named `model__`[2]. The `amt::amtModel` object keeps track of the names and dimensions of the parameters etc, the value of log-target distribution and so on.

`operator()` typically consist of three parts:

- specifying the sampled quantities θ using `PARAMETER_*` macros.
- specifying the shape of the target distribution $\pi(\theta)$.

[1] So far, `varType` is either of type `stan::math::var` or `amt::amtVar` (see header `include/amt/amtVar.hpp`).

[2] In principle it could be named something else, but this will break the `PARAMETER_*`-macros discussed below. Hence, if the argument to `operator()` is named something else, the user should provide calls similar to those defined by these macros (see `include/amt.hpp`)

- specifying other quantities of interest for which samples are recorded.

In what follows, each of these point are discussed in more detail:

4.3.1 The PARAMETER_* macros

The sampled quantities (e.g. parameters, latent variables and so on) are specified using the PARAMETER_* macros:

```
PARAMETER_SCALAR(<name>,...); // C++ storage: varType
PARAMETER_VECTOR(<name>,dim,...);
// C++ storage: Eigen::Matrix<varType,Eigen::Dynamic,1>
PARAMETER_MATRIX(name,dim1,dim2,...);
// C++ storage: Eigen::Matrix<varType,Eigen::Dynamic,Eigen::Dynamic>
PARAMETER_SPD_MATRIX(name,dim,...);
// C++ storage: amt::SPDmatrix<varType>
```

All of these macros presume that the argument to the operator()-function is named model__. After a call to e.g. PARAMETER_VECTOR(<name>,dim,...);, a vector of varTypes of dimension dim and with name <name> will be available subsequently.

The SPDmatrixclass template is further explained in Chapter 6.1

An optional argument - the default value - may be passed to each of the PARAMETER_* macros. The default value is used when initiating the sampling, and **it is important that $\pi(\theta)$ is defined when θ corresponds to the default values**. If no default values is provided, the default value is implicitly set to 0.

(Non-zero) default values could be either double scalars which are repeated to fill non-scalar variables, e.g.

```
PARAMETER_VECTOR(mu,4,1.0);// mu is intiated at (1.0,1.0,1.0,1.0)
```

Otherwise, the default values may be vectors/matrices of the same dimension(s) as the parameter, e.g.

```
Eigen::Vector muDefault(4);
muDefault.setZero();
muDefault(3) = 2.0;
PARAMETER_VECTOR(mu,muDefault.size(),muDefault);
// mu is intiated at (0.0,0.0,0.0,2.0)
```

One may even pass the dimension and default values of the parameters using the DATA_*:

```
struct model{
  DATA_VECTOR(dvec);
  void preProcess(){}
```

```
  template < class varType, class tensorType, bool storeNames>
  void operator()(amt::amtModel<varType,tensorType,storeNames> &model__){
    PARAMETER_VECTOR(d,dvec.size(),dvec);
    // d will have the dimension of dvec, and default values=dvec
    // rest of model spec here
  }
};
```

Note that there are no facilities for specifying that certain elements θ to be restricted to a certain range (e.g. positive). Hence the user is responsible for ensuring that suitable transformations are made to avoid numerical problems. See e.g. the log-transformation of the standard deviation in the example in Section 3.1

4.3.2 Specifying the shape of $\pi(\theta)$.

Specifying the shape of the target distribution is done using various overloaded `+=` operators applied to the `amt::amtModel` object. It is assumed that $\log \pi(\theta)$ may be written as the sum of several log-densities.

Two sets of such log-densities are available:

- For fixed metric samplers and Riemannian samplers (see Chapter (ref:the-samplers) for details), a library of log-densities with names ending with `_ld` or `_lm` are available under namesspace `amt`. These are further documented in Chapter 6
- For fixed metric samplers only, the complete set of `stan::math` functions with names ending with `_lpdf` or `_lpmf` (see https://mc-stan.org/docs/functions-reference/index.html) are available

Below are two examples, both specifying a 4-dimensional standard normal target distribution:

```
//...
  template < class varType, class tensorType, bool storeNames>
  void operator()(amt::amtModel<varType,tensorType,storeNames> &model__){
    PARAMETER_VECTOR(x,4);
    model__ += normal_ld(x,0.0,1.0); // amt version
  }
//...
```

```
//...
  template < class varType, class tensorType, bool storeNames>
  void operator()(amt::amtModel<varType,tensorType,storeNames> &model__){
    PARAMETER_VECTOR(x,4);
    model__ += stan::math::normal_lpdf(x,0.0,1.0); // stan::math version
  }
```

```
//...
```

Further, more elaborate examples of model specifications are provided in Chapter 8.

4.3.3 Generated quantities

Generated quantities are used in two ways:

- It is often the case that one is interested in the posterior distribution of some auxiliary quantities which depend on the parameter θ.
- The `pdmphmc` package has the capability to compute "time-integrated" samples (see e.g. Kleppe [2022a], Equation 7 and Figure 3), which in certain cases may be much more efficient for estimating certain moments. Time integrated samples are computed for all generated quantities (and not the parameters, so if you want time-integrated samples of the parameters, pass them as generated quantities).

To record such quantities, the `generated()` function (member of `amt::amtModel`) is used. There are several overloads of `generated()` allowing several types (scalars, vectors, matrices etc.) to be passed.

The `generated()` functions have signatures similar to `inline void generated(const <SOME TYPE>& value,const std::string& name)`, where `<SOME TYPE>` is a C++ type/template, such as e.g. `double`, `Eigen::VectorXd` or `varType`. `name` is the name used for the generated quantity in the output.

Below is a complete example of the usage. First, the content of file `generated_model.cpp`:

```
using namespace amt;

double quadNorm(const Eigen::VectorXd& x){
  return(x.dot(x));
}

struct model{
  void preProcess(){} // not used in this example
  template < class varType, class tensorType, bool storeNames>
  void operator()(amt::amtModel<varType,tensorType,storeNames> &model__){

    PARAMETER_VECTOR(x,4); // parameter (sampled quantity)
    model__+=normal_ld(x,0.0,1.0); // add standard normal log-density


    // make the whole parameter a generated
    model__.generated(asDouble(x),"x_gen");
    // different transformations of x to be recored:
```

```
    model__.generated(std::pow(asDouble(x(0)),3),"x1_cube");
    model__.generated(exp(asDouble(x(1))),"x2_exp");

    double qn = quadNorm(asDouble(x));
    model__.generated(qn,"quadraticNorm");

  }
}; // end of struct
```

Note that it is good practice to provide `double` values as the first argument to `generated()`. The double value of AD types `varType` obtains using the `asDouble()` function.

Then build and run the model:

```
model <- pdmphmc::build("generated_model.cpp")
```

```
## model name : model

## process type : HMCProcessConstr

## Runge Kutta step type : RKDP54

## Transport map type : diagLinearTM_VARI

## compilation exited successfully
```

```
fit <- pdmphmc::run(model)
pdmphmc::clean.model(model,remove.all = TRUE)
```

```
## [1] TRUE
```

```
fit
```

```
## run output for model: model

## # of chains : 4

## Summary based on discrete samples:

##                  mean se_mean    sd n_eff  Rhat
## x[1]           -0.003   0.017 0.977  3301 1.004
## x[2]           -0.007   0.018 1.035  3495 1.003
## x[3]           -0.001   0.017 1.014  3640 0.999
## x[4]            0.002   0.017 0.973  3397 1.002
## x_gen[1]       -0.003   0.017 0.977  3301 1.004
## x_gen[2]       -0.007   0.018 1.035  3495 1.003
## x_gen[3]       -0.001   0.017 1.014  3640 0.999
## x_gen[4]        0.002   0.017 0.973  3397 1.002
## x1_cube        -0.001   0.063 3.797  3599 1.005
## x2_exp          1.697   0.040 2.345  3415 1.002
## quadraticNorm   4.000   0.070 2.845  1661 1.002
```

```
## summary based on integrated samples:

##               estimate se_estimate n_eff  Rhat
## x_gen[1]         0.005       0.005  6235 1.004
## x_gen[2]         0.003       0.005  5887 1.002
## x_gen[3]         0.004       0.005  6369 1.003
## x_gen[4]        -0.002       0.005  5919 1.000
## x1_cube          0.016       0.018  6310 1.007
## x2_exp           1.708       0.037   982 1.003
## quadraticNorm    4.020       0.068   826 1.003

## NOTE: integrated samples do NOT reflect the complete target distibution, only indicated moment
```

It is seen that discrete time samples (which reflect target distribution) are recorded (by default) for the parameters (in this case `x[1]`,…,`x[4]`) and the generated quantities (here `x_gen[1]`,…,`x_gen[4]`,`x1_cube`,`x2_exp`,`quadraticNorm`).

Integrated samples for calculating moments with respect to the target distribution are recorded for the generated quantities only.

Some minor comments related to performance are in order here:

- As seen in the example above, `double` values (or vectors or matrices thereof) should ideally be passed to the `generated()` functions. This is in order of avoiding unnecessary AD computations. In particular, if the generated function is a complicated function of the parameters (for which function `quadNorm()` may serve as a placeholder), the computations leading up to the generated quantity should be done in double-variables.
- In models where θ is high-dimensional and parts of θ are of less interest, the default behavior of storing samples of all of θ may be overridden by using the `store.pars` argument in `pdmphmc::run()`. Reducing the number of stored quantities leads to both memory savings and savings in terms of computing time.

Chapter 5

Constraints

In a recent update of `pdmphmc`, constraints are included as a part of the model specification.

The parameters of the model may be specified to be constrained to on a specific domain, i.e. so that sampling is performed with stationary distribution being

$$\propto \pi(\theta) \text{ such that } c_1(\theta) > 0, \dots, c_B(\theta) > 0$$

where $c_b(\theta),\ b = 1, \dots, B$ are scalar constraint functions.

Further details regarding the numerical details may be found in a paper which will be published very soon.

Constraints are so far only available for the fixed mass sampler, and requires that the `RKBS32` integrator is used (see Section 7.3, obtained using `step.type = "RKBS32"` in `build()`) .

5.1 Specifying constraints

There are several different types of constraints, all of which are specified by member-functions of the `amt::amtModel` class (see examples below).

In many cases, a given constraint may be specified in multiple ways, and in general, always using the non-linear constraint specification described at the end of this subsection. However, *if the constraint in question has some special structure, this may be exploited in the numerical implementation to obtain faster sampling*, and should therefore be used. Below, first the different such special structures are described, before general non-linear constraints are discussed.

5.2 Special constraints

The following special constraints are available:

5.2.1 Dense Linear constraints

may be written as

$$\mathbf{a}^T\theta - b > 0$$

where $\mathbf{a}$ is a constant (i.e. not depending on θ) vector and b is a constant scalar. Specified using `linConstraint()`. The user is only responsible for providing code for $\mathbf{a}^T\theta - b$. E.g.

```
template < class varType, class tensorType, bool storeNames>
  void operator()(amt::amtModel<varType,tensorType,storeNames> &model__){
    PARAMETER_VECTOR(theta,2); // parameter (sampled quantity)
    /* ... */
    varType s = 0.5*theta(0)-2.0+theta(1); // should be positive
    model__.linConstraint(s); // specify the constraint
  }
```

ensures that sampling only occur for values of θ so that $s = \theta_1/2 + \theta_2 - 2 > 0$.

5.2.2 Sparse Linear constraints

are essentially the same as dense linear constraints, but where $\mathbf{a}$ is assumed to contain many zeros. Specified using `sparseLinConstraint()`.

5.2.3 Sparse Linear L^1 constraints

may be expressed as

$$\| \mathbf{A}\theta - \mathbf{b} \|_1 < c$$

for constant matrix $\mathbf{A}$, constant vector b and constant scalar c. Specified using `sparseLinL1Constraint()` The user is only responsible for code evaluating $\mathbf{A}\theta - \mathbf{b}$ and c. E.g.

```
template < class varType, class tensorType, bool storeNames>
  void operator()(amt::amtModel<varType,tensorType,storeNames> &model__){
    PARAMETER_VECTOR(theta,2); // parameter (sampled quantity)
    /* ... */
    VectorXv t(3); // should have L1-norm less than 2.0
    t(0) = theta(0) + 1.0;
    t(1) = theta(1)-theta(0);
    t(2) = theta(0) + 1.0;
    model__.sparseLinL1Constraint(t,2.0); // specify the constraint
  }
```

It is assumed that matrix $\mathbf{A}$ is a sparse matrix.

5.2.4 Sparse Linear L^2 constraints

Similar as above, but for L^2 norm, i.e.

$$\| \mathbf{A}\theta - \mathbf{b} \|_2 < c.$$

Specified using `sparseLinL2Constraint()`.

5.3 Non-linear constraints

Suppose the the constraint may be written as

$$F(\mathbf{A}\theta - \mathbf{b}) > 0$$

for some non-linear function $F : \mathbb{R}^p \mapsto \mathbb{R}$, a constant matrix $\mathbf{A}$ with p rows and a constant vector $\mathbf{b} \in \mathbb{R}^p$. Further, it is assumed that the gradient of F may be computed, either analytically or at least be found efficiently using automatic differentiation.

Note that A may in principle be the identity matrix, but it is often the case that p is much smaller than the dimension of θ. Such constraints should be specified via a functor class and the `sparseLinFunConstraint()`-function.

Technical details regarding the constraint functor are found below in Section 5.3.1. First, a working example of the functor representing $F(\mathbf{z}) = 1 - \frac{1}{2}\mathbf{z}^T\mathbf{z}$ is considered. The functor should both include code evaluating F itself and also F and the gradient of F:

```
class L2fun : public constraintFunctor {
public:
  // evaluates F only
  inline double operator()(const Eigen::VectorXd& arg) const {
    return(1.0-0.5*arg.squaredNorm());
  }
  // evaluates F and the gradient of F stored in grad
  inline double operator()(const Eigen::VectorXd& arg,
                           Eigen::VectorXd& grad) const {
    grad = -arg; // gradient of F
    return(operator()(arg)); // return value of F itself
  }
  // arbitrary name
  inline std::string name() const {return "L2fun__";}
};
```

Further, the model specification should include an instance of the functor, and a call to the model-member function `sparseLinFunConstraint()`, e.g.

```
using namespace amt;
struct model{
  L2fun functor;
  /* ... */
  template < class varType, class tensorType, bool storeNames>
  void operator()(amt::amtModel<varType,tensorType,storeNames> &model__){
    PARAMETER_VECTOR(x,5);

    /* model etc... */

    // ensures that 0.5*sqrt(sum(x[0:2]^2))<1.0
    model__.sparseLinFunConstraint(x.head(3),functor);
  }
}
```

5.3.1 Details of the constraint functor

The functor representing F **must** inherit virtual base class

```
class constraintFunctor{
public:
  virtual double operator()(const Eigen::VectorXd& arg) const = 0;
  virtual double operator()(const Eigen::VectorXd& arg,
                            Eigen::VectorXd& grad) const  = 0;
  virtual inline std::string name() const {return "unnamed constraint functor";}
};
```

Additional member functions/non-trivial constructors may be added as necessary.

The `pdmphmc` package comes with a simple forward mode automatic differentiation library `FMVAD` which may be used to calculate the required gradient if the gradient is complicated to code. The library is yet to be documented.

5.4 Worked example

The following code is contained in file `constraint_model.cpp` :

```
class nonLinFun : public constraintFunctor {
public:
  // evaluates F only
  inline double operator()(const Eigen::VectorXd& arg) const {
    return(1.0 - arg(0)*arg(1));
  }
  // evaluates F and the gradient of F stored in grad
  inline double operator()(const Eigen::VectorXd& arg,
```

```
                          Eigen::VectorXd& grad) const {
    if(grad.size()!=arg.size()) grad.resize(arg.size());
    grad.setZero();
    grad(0) = -arg(1);
    grad(1) = -arg(0); // gradient of F
    return(operator()(arg)); // return value of F itself
  }
  // arbitrary name
  inline std::string name() const {return "some_constraint_functor";}
};


using namespace amt;
struct model{

  nonLinFun functor_;

  DATA_INT(constrType); // which constraint type to use, passed from R

  void preProcess(){} // not used in this example

  template < class varType, class tensorType, bool storeNames>
  void operator()(amt::amtModel<varType,tensorType,storeNames> &model__){


    PARAMETER_VECTOR(x,2); // parameter (sampled quantity)
    // x is bivariate normally distributed with correlation rho
    // before any constraints are imposed
    double rho = 0.7;
    model__+=normal_ld(x(0),0.0,1.0);
    model__+=normal_ld(x(1),rho*x(0),sqrt(1.0-rho*rho));

    // now add the different constraints
    if(constrType==1){
      model__.linConstraint(1.0-x(0)-x(1));
    } else if(constrType==2){
      model__.sparseLinConstraint(1.0-x(0)-x(1));
    } else if(constrType==3){
      VectorXv z = x;
      z(0)-=0.5;
      z(1)-=0.25;
      model__.sparseLinL1Constraint(z,2.0);
    } else if(constrType==4){
      VectorXv z(x.size());
      z(0) = x(0) + 0.5;
```

```
      z(1) = 2.0*x(1) - 0.5*x(0) - 0.25;
      model__.sparseLinL2Constraint(z,2.0);
    } else if(constrType==5){
      model__.sparseLinFunConstraint(x,functor_);
    }
  }
}; // end of struct
```

The model is built and run without constraints using (note `step.type = "RKBS32"`)

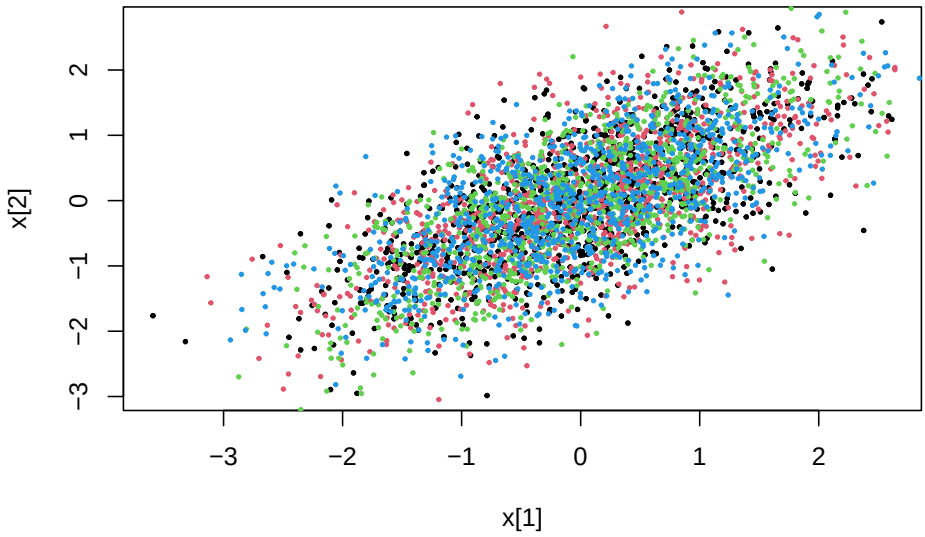
```
mdl <- pdmphmc::build("constraint_model.cpp",step.type = "RKBS32")
```

```
## model name : model

## process type : HMCProcessConstr

## Runge Kutta step type : RKBS32

## Transport map type : diagLinearTM_VARI

## compilation exited successfully
```

```
# unconstrained
fit <- pdmphmc::run(mdl,data=list(constrType=0L))
```

```
## Note: no integrated samples recorded
```

```
pdmphmc::pair.plot(fit,"x[1]","x[2]")
```

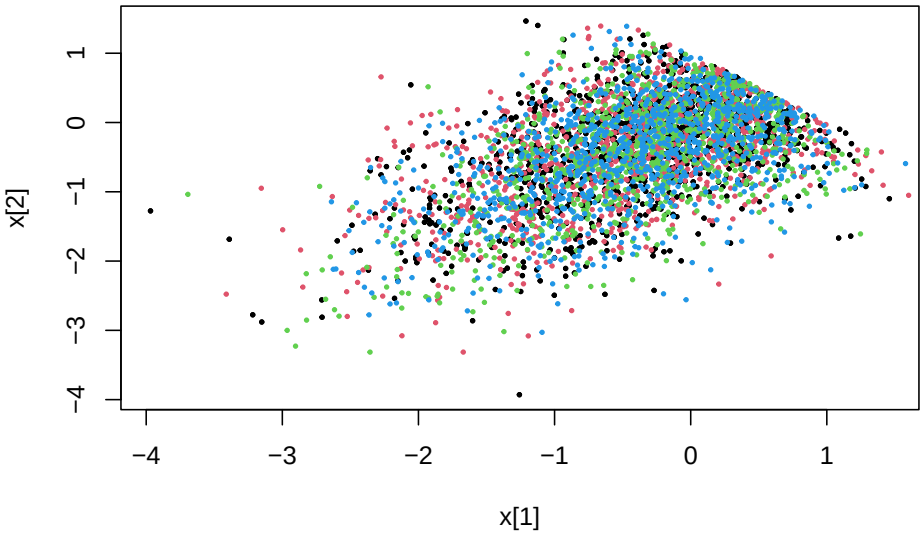


```
# (dense) linear constraint 1.0-x(0)-x(1)>0
fit <- pdmphmc::run(mdl,data=list(constrType=1L))
```

3.4.1 Linear constraints (dense and sparse)

```
## Note: no integrated samples recorded
```

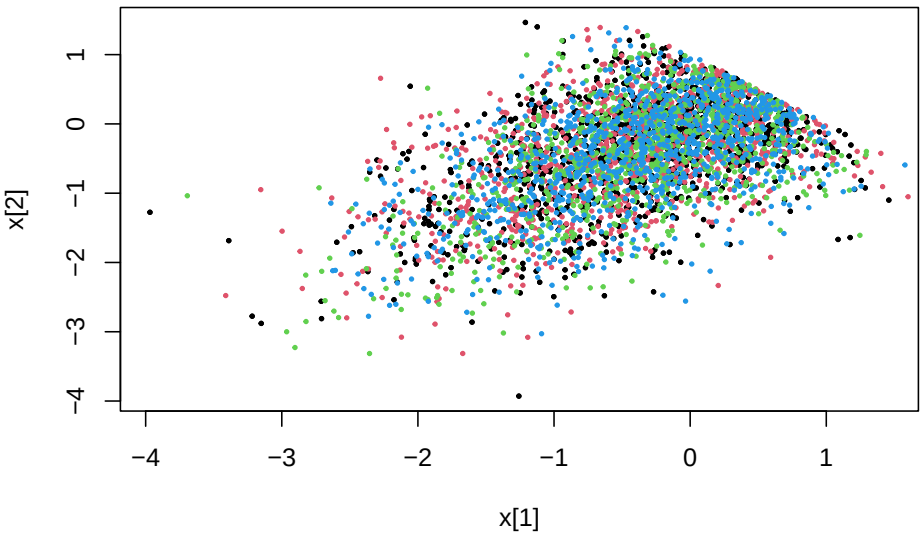
```
pdmphmc::pair.plot(fit,"x[1]","x[2]")
```




```
# linear constraint 1.0-x(0)-x(1)>0
fit <- pdmphmc::run(mdl,data=list(constrType=2L))
```

```
## Note: no integrated samples recorded
```

```
pdmphmc::pair.plot(fit,"x[1]","x[2]")
```

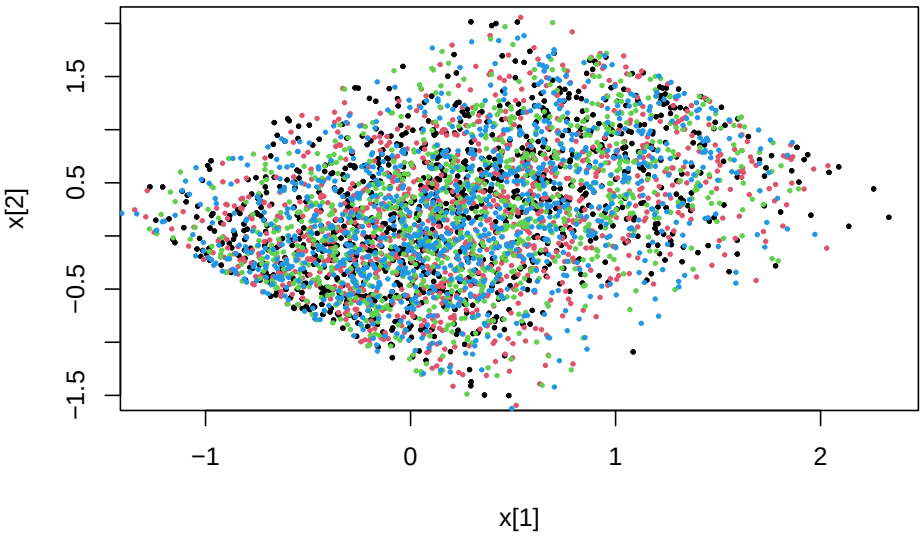


5.4.2 L^1 constraint

```
# L1 norm of (x(0)-0.5,x(1)-0.25) smaller than 2.0
fit <- pdmphmc::run(mdl,data=list(constrType=3L))
```

```
## Note: no integrated samples recorded
```

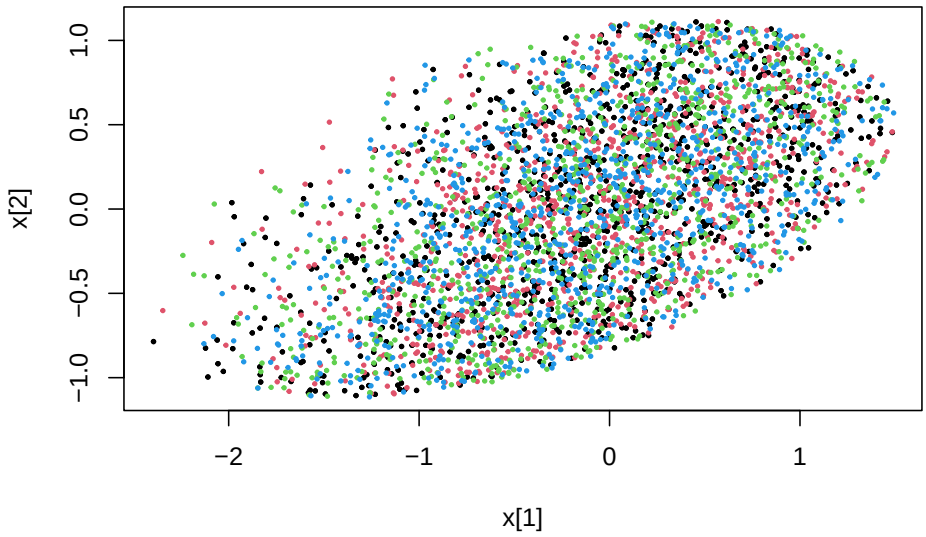
```
pdmphmc::pair.plot(fit,"x[1]","x[2]")
```



```
# L2 norm of (x(0) + 0.5, 2.0*x(1) - 0.5*x(0) - 0.25) smaller than 2.0
fit <- pdmphmc::run(mdl,data=list(constrType=4L))
```

```
## Note: no integrated samples recorded
```

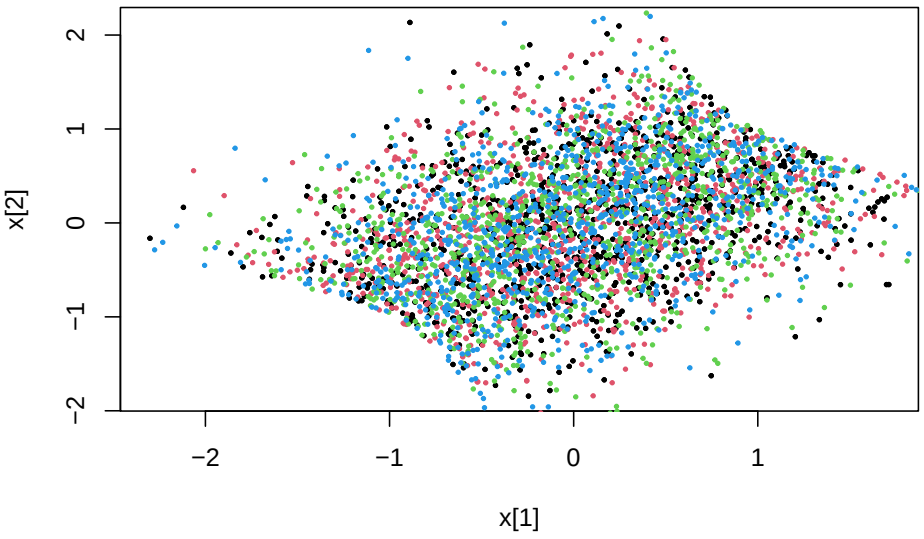
```
pdmphmc::pair.plot(fit,"x[1]","x[2]")
```




```
# nonlinear;
fit <- pdmphmc::run(mdl,data=list(constrType=5L))
```

```
## Note: no integrated samples recorded
```

```
pdmphmc::pair.plot(fit,"x[1]","x[2]")
```


x[2]
2
1
0
−1
−2
−2
−1
0
1
x[1]

Chapter 6

The amt-library

This section gives an overview of the different probability density/mass functions, in addition to some utility functions available in the `amt`-library. The library provides an automatic implementation of the methodology of Kleppe [2022b] for automatically computing a "metric tensors" suitable Riemannian sampling algorithms for arbitrary models.

The design of the library allows the user to use the same model specification for both Riemannian and regular fixed metric sampling (see Chapter 7).

The methodology of Kleppe [2022b] generally requires that all probability densities involved in the model where the argument depends on θ (generally the conditional densities of priors and latent variables) to have continuous first order derivatives. To achieve such continuity, many of the below distributions are known probability distributions that have been transformed to have support on the whole real line in such a way that the resulting probability distribution is sufficiently smooth. These transformations are also desirable from a numerical point of view.

As a basic example of such transformations involves the *ExpGamma*-distribution (see below for proper definition):

```
// inside operator()()
PARAMETER_SCALAR(logSigma,0.0);
model__+=expGamma_ld(logSigma,1.0,1.0);
varType sigma=exp(logSigma);
model__.generated(asDouble(sigma),"sigma");
```

In the above example, `sigma` will be exponentially distributed with mean 1 (until further conditioning is done) and will be recorded as a generated quantity.

6.1 Utilities for symmetric positive definite matrices

The `amt`-package provides a set of utilities for representing symmetric positive definite (SPD) matrices. Internally, this class template `SPDmatrix` represents a $d \times d$ SPD matrix as an "internal" vector of $d(d+1)/2$ `varTypes`.

`SPDmatrix` objects may be used as e.g. covariance- or precision matrices in multivariate normal distributions, and it also possible to assign e.g. Wishart distributions on such objects (though in practice, such assignments are really distributions on the internal vector, consistent with the matrix-valued distribution).

The easiest way to construct a `SPDmatrix` is via the `PARAMETER_SPD_MATRIX(name,dim,...)` macro. Typical usage involves something like:

```
// inside operator()() :
PARAMETER_SPD_MATRIX(P,3);
// Note: PARAMETER_SPD_MATRIX(P,3) really makes a parameter vector
// named P_internal, and subsequently constructs SPDmatrix<varType> P
Eigen::VectorXd scaleDiag; scaleDiag.setConstant(3,4.0);
model__ += wishartDiagScale_ld(P,scaleDiag,10.0);
// P has a Wishart(4*I,10.0) distribution a priori
model__.generated(P,"P"); // full matrix stored
model__.generated(P.coeff(0,2),"P13"); // single element stored
```

A complete class where the above code block is the body of the `operator()()`-method is stored in the file `wishart_example.cpp`, which is built an run via:

```
mdl <- pdmphmc::build("wishart_example.cpp")
```

```
## model name : model

## process type : HMCProcessConstr

## Runge Kutta step type : RKDP54

## Transport map type : diagLinearTM_VARI

## compilation exited successfully
```

```
fit <- pdmphmc::run(mdl)
pdmphmc::clean.model(mdl,remove.all = TRUE)
```

```
## [1] TRUE
```

```
fit
```

```
## run output for model: model

## # of chains : 4
```

```
## Summary based on discrete samples:

##                  mean se_mean     sd n_eff  Rhat
## P_internal[1]  3.578   0.010  0.485  2492 1.005
## P_internal[2]  3.466   0.010  0.497  2431 1.003
## P_internal[3]  3.330   0.009  0.539  3209 1.002
## P_internal[4]  0.000   0.007  0.370  3153 1.002
## P_internal[5] -0.009   0.007  0.365  3057 1.001
## P_internal[6]  0.006   0.008  0.378  2498 1.001
## P[1,1]        39.964   0.364 18.438  2565 1.005
## P[2,1]         0.124   0.236 13.323  3176 1.002
## P[3,1]        -0.229   0.236 13.037  3037 1.000
## P[2,2]        40.288   0.353 17.848  2559 1.005
## P[3,2]         0.341   0.245 12.695  2680 1.001
## P[3,3]        40.141   0.312 18.027  3330 1.001
## P13           -0.229   0.236 13.037  3037 1.000

## summary based on integrated samples:

##        estimate se_estimate n_eff  Rhat
## P[1,1]   39.673       0.190  1690 1.006
## P[2,1]    0.025       0.054  7448 1.040
## P[3,1]    0.052       0.054  6768 1.033
## P[2,2]   40.231       0.182  1733 1.004
## P[3,2]    0.156       0.110  1972 1.003
## P[3,3]   40.250       0.166  2207 1.002
## P13       0.052       0.054  6768 1.033

## NOTE: integrated samples do NOT reflect the complete target distibution, only indicated moment
```

Notice that by default, the "internal" parameter vector `P_internal` is stored[1] in addition to the generated quantities. Note, in this case, we would expect $E(\mathbf{P}) = 40\mathbf{I}$.

6.2 Univariate continuous distributions

6.2.1 `expGamma_ld(arg,shape,scale)`

The distribution obtained by applying the (natural) logarithm to a gamma-distributed random variable with shape parameter $\alpha =$`shape` and scale parameter $\beta =$`scale`. Argument of resulting PDF $x =$`arg`. Log-density:

$$\log p(x|\alpha,\beta) = -a\log(b) - \log(\Gamma(a)) + ax - \exp(x)/b.$$

[1] This may be avoided by using the `store.pars`-option in `pdmphmc::run()`.

6.2.2 invLogitBeta_ld(x,a,b)

The distribution obtained by applying the logit-function to a Beta distributed random variable with shape parameters `a` and `b`. Argument of resulting PDF: $x =$`x`. Log-density:

$$\log p(x|a,b) = \log(\Gamma(a+b)) - \log(\Gamma(a)\Gamma(b)) + a \log\left(\frac{\exp(x)}{1+\exp(x)}\right) - b\log(1+\exp(x)).$$

6.2.3 invLogitUniform_ld(x)

The distribution obtained by applying the logit-function to a uniform(0,1); same as the standard logistic distribution. Argument of resulting PDF: $x =$`x`. Log-density:

$$\log p(x) = x - 2\log(1+\exp(x))$$

6.2.4 normal_ld(arg,mean,sd)

Normal/Gaussian distribution with mean=`mean` and standard deviation=`sd`. Argument of PDF: $x =$`arg`.

6.3 Univariate discrete distributions

6.3.1 bernoulli_logit_lm(y,alpha)

Bernoulli-distribution with $P(y = 1) = \exp(\alpha)/(1+\exp(\alpha))$. Argument y is of integer type.

6.3.2 poisson_log_lm(y,eta)

Poisson distribution with mean equal to $\exp(\eta)$. Argument y is of integer type.

6.3.3 ziPoisson_log_lm(y,eta,g)

Zero-inflated Poisson distribution. A mixture of point-mass at $y = 0$ (with weight p) and a regular Poisson distribution with mean $\exp(\eta)$ (with weight $1-p$). $g = \text{logit}(p)$.

6.4 Distributions on unbounded vectors

6.4.1 multi_normal_prec_ld(arg,mu,Prec)

Multivariate normal distribution where the **precision** matrix `Prec` is a `SPDmatrix` (see Section 6.1). Both the argument `arg` and `mean` are vectors with dimension equal to `Prec.dim()`.

6.4.2 `iid_multi_normal_prec_ld(arg,mu,Prec)`

Same as above, but `arg` is now a matrix, where the columns are iid realizations from $N(\mu, P^{-1})$, where P =`Prec`.

6.4.3 `multi_normal_ld(arg,mu,Sigma)`

Multivariate normal distribution where the **covariance** matrix `Sigma` is a `SPDmatrix` (see Section 6.1). Both the argument `arg` and `mean` are vectors with dimension equal to `Sigma.dim()`.

To do: iid-variant of this.

6.4.4 `normalAR1_ld(arg,mu,phi,sigma)`

Stationary Gaussian first order autoregressive process

$$x_1 \sim N\left(\mu, \frac{\sigma^2}{1-\phi^2}\right), x_t = \mu + \phi(x_{t-1} - \mu) + N(0, \sigma^2), t = 2, \ldots, T$$

for scalar parameters μ, $-1 < \phi < 1$ and $\sigma > 0$. $\mathbf{x}$ =`arg` is a T-dimensional vector.

6.4.5 `normalRW1_ld(x,sigma)`

First order (intrinsic) Gaussian random walk model:

$$x_t \sim N(x_{t-1}, \sigma^2),\ t = 2, \ldots, T,$$

for $\sigma > 0$. $\mathbf{x}$ =`x` is a T-dimensional vector. Note that this model specifies a degenerate probability distribution, and $\mathbf{x}$ cannot be sampled without further conditioning.

6.5 Distributions on symmetric positive definite matrices

The notation for the Wishart distribution is $\mathcal{W}(\mathbf{M}, \nu)$ where $\mathbf{M}$ is the scale matrix and ν is the degrees of freedom parameter. Consequently, if $\mathbf{P} \sim \mathcal{W}(\mathbf{M}, \nu)$, then $E(\mathbf{P}) = \nu\mathbf{M}$.

6.5.1 `wishartDiagScale_ld(arg,scaleDiag,df)`

The Wishart distribution $\mathbf{P} \sim \mathcal{W}(\mathbf{M}, \nu)$ where $\mathbf{P}$ =`arg` is a `SPDmatrix`, vector diag($\mathbf{M}$)=`scaleDiag` and scalar ν =`df`.

6.5.2 `wishartRW1_ld(arg,mean,nu)`

The Wishart distribution $\mathbf{P} \sim \mathcal{W}(\nu^{-1}\mathbf{Q}, \nu)$. $\mathbf{P}$ =`arg` is a `SPDmatrix`, $E(\mathbf{P}) = \mathbf{Q}$ =`mean` is a `SPDmatrix` and scalar ν =`nu`.

Chapter 7

The sampling algorithms

As the name of the package indicates, the samplers in employed in this package are based on piecewise deterministic Markov processes (see e.g. Fearnhead et al. [2018]) with Hamiltonian dynamic between events (see Kleppe [2022a]).

Two distinct such samplers are so far implemented: fixed mass (section 7.1) and Riemannian manifold (Section 7.2). The Hamiltonians, which are fully specifies the between event dynamics are provided below.

The same set of numerical differential equation solvers (Section 7.3) are used for both solvers. Finally, methods for adapting the scaling of the differential equations (Section 7.4) and the different available event intensities (Section 7.5) are described below.

The sampling algorithm to be used are chosen prior to the compilation `pdmphmc::build()`. By default, a fixed mass sampler is used.

Whether to use the RM sampler or fixed metric sampler is inherently problem-specific. As a rule of thumb, if your target distribution involves complicated non-linear dependencies, or the scale of some subset of θ depends strongly on some other subset of θ, the RM sampler is often a good choice. However, the differential equations associated with the RM sampler are more expensive to compute than for the fixed metric sampler. Hence, if it is possible to rewrite your model so that the fixed metric sampler gives good results (see e.g. Kleppe [2019] or Osmundsen et al. [2021]), this approach is often a good choice.

7.1 Fixed mass sampler

Suppose the target density is given by $\pi(\theta)$ and that the model specification admits the evaluation $\bar{\pi}(\theta) \propto \pi(\theta)$. The Hamiltonian $\mathcal{H}(\mathbf{q}, \mathbf{p})$ used to define

the deterministic dynamics for the fixed mass sampler is

$$\mathcal{H}(\mathbf{q}, \mathbf{p}) = -\log \pi(\mathbf{m} + \mathbf{S}\mathbf{q}) + \frac{1}{2}\mathbf{p}^T\mathbf{p}.$$

Here $\mathbf{p}$ is the (fictitious) momentum variable introduced to construct a dynamical system with Boltzmann-Gibbs distribution $\pi(\mathbf{q}, \mathbf{p}) \propto \exp(-\mathcal{H}(\mathbf{q}, \mathbf{p}))$.

Samples are subsequently recorded for $\theta = \mathbf{m} + \mathbf{S}\mathbf{q}$ which will be distributed according to $\pi(\theta)$. Here vector $\mathbf{m}$ and diagonal positive definite matrix $\mathbf{S}$ are adapted during the warmup process (see Chapter 7.4)

The vector $\mathbf{m}$ should reflect the mean of the target distribution, and the diagonal elements of $\mathbf{S}$ should reflect the scale properties of θ under the target distribution.

The reasoning behind introducing the re-parameterization $\mathbf{q} \leftrightarrow \theta$ (rather than to account for different scales by introducing a non-identity mass matrix in the kinetic energy term of $\mathcal{H}(\mathbf{q}, \mathbf{p})$) is based on the desire for obtaining well-scaled Hamilton's equations

$$\dot{\mathbf{q}}(t) = \mathbf{p}(t)$$

$$\dot{\mathbf{p}}(t) = \mathbf{S}\mathbf{g}(\mathbf{m} + \mathbf{S}\mathbf{q}(t)),\ \mathbf{g}(\theta) = \nabla_\theta \log \pi(\theta)$$

suitable for numerical integration. I.e. for suitably chosen $\mathbf{S}$ the force term in both equations should have order 1 (as $Var(\mathbf{p}) = \mathbf{I}$ under the Boltzmann-Gibbs distribution).

The sampler is chosen explicitly by selecting `process.type = "HMCProcess"` in the call to `pdmphmc::build()`, but as mentioned is already the default option.

7.2 Riemann manifold sampler

The Riemann manifold (RM) sampler is based on the availability of a "metric tensor" $\mathbf{G}(\theta)$. Namely for each θ, then $\mathbf{G}(\theta)$ is a symmetric positive definite matrix that may be interpreted as the "local precision matrix" of the target distribution.

The `amt`-library provides an automatic methodology for computing metric tensors from a given model specification (see Kleppe [2022b] for details), and by default this methodology is used if the RM sampler is selected via `process.type = "RMHMCProcess"` in the call to `pdmphmc::build()`.

The Hamiltonian for the RM sampler is given by

$$\mathcal{H}(\mathbf{q}, \mathbf{p}) = -\log \bar{\pi}(\mathbf{m} + \mathbf{S}\mathbf{q}) + \frac{1}{2}\log(|\bar{\mathbf{G}}(\mathbf{q})|) + \frac{1}{2}\mathbf{p}^T[\bar{\mathbf{G}}(\mathbf{q})]^{-1}\mathbf{p}$$

where

$$\bar{\mathbf{G}}(\mathbf{q}) = \mathbf{S}^T\mathbf{G}(\mathbf{m} + \mathbf{S}\mathbf{q})\mathbf{S}.$$

It is seen that the interpretation of **m** and **S** are the same also for the RM sampler, i.e. **m** should reflect the center/mean of the distribution, and **S** should reflect the scale of each element in θ.

7.2.1 Metric tensor storage

Two storage schemes for the metric tensor are available

- sparse storage using on the `ldl` Cholesky factorization (Davis [2005])
- dense storage using the Cholesky factorization of the Stan math library.

One should use the sparse storage scheme if the metric tensor is very sparse, which is often the case for hierarchical models. The metric tensor scheme is chosen during the `pdmphmc::build()`-process by selecting `metric.tensor.type = c("Sparse", "Dense")`.

Note that for the sparse storage scheme, the ordering of the parameters may play a role for the performance of the Cholesky factorization (see Davis [2006]). The ordering of the parameters are determined by the ordering of the `PARAMETER_*` statements in the model specification.

7.2.2 Advanced

It is also possible to specify ones own metric tensor. One turns off the default `amt`-library metric tensor by choosing the option `amt=FALSE` in the call to `pdmphmc::build()`. This process will be documented in more detail soon.

7.3 ODE solvers

The ODE solver is a bespoke Runge Kutta method with a PI controller for adaptive time step sizes. The Runge Kutta type steps (including dense formulas) implemented so far are

- The order 5 (4) pair of Dormand and Prince [1980] (obtained by choosing `step.type = "RKDP54"` in `build()`).
- The order 3 (2) pair of Bogacki and Shampine [1989] (obtained by choosing `step.type = "RKBS32"` in `build()`)

The latter is generally preferred only for models with constraints (see Section 5) or when running with very lax integration tolerances.

7.4 Scaling adaptation

So far, only square-roots of estimates of the diagonal elements of $Var(\theta)$ are available as the diagonal elements of **S**. See Kleppe [2022a] for details

7.5 Event intensity

So far, only constant event intensities are available. See Kleppe [2022a] for details on on the adaptive choice of this constant event intensity.

Chapter 8

Worked examples

More to come here.

Bibliography

P. Bogacki and L.F. Shampine. A 3(2) pair of Runge - Kutta formulas. *Applied Mathematics Letters*, 2(4):321–325, 1989. ISSN 0893-9659. doi: https://doi.org/10.1016/0893-9659(89)90079-7. URL https://www.sciencedirect.com/science/article/pii/0893965989900797.

T. A. Davis. Algorithm 849: A concise sparse cholesky factorization package. *ACM Transactions Mathematical Software*, 31:587–591, 2005.

Timothy A. Davis. *Direct Methods for Sparse Linear Systems*, volume 2 of *Fundamentals of Algorithms*. SIAM, 2006.

J.R. Dormand and P.J. Prince. A family of embedded runge-kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19–26, 1980. ISSN 0377-0427. doi: https://doi.org/10.1016/0771-050X(80)90013-3. URL https://www.sciencedirect.com/science/article/pii/0771050X80900133.

Paul Fearnhead, Joris Bierkens, Murray Pollock, and Gareth O. Roberts. Piecewise deterministic Markov processes for continuous-time monte carlo. *Statist. Sci.*, 33(3):386–412, 08 2018. doi: 10.1214/18-STS648. URL https://doi.org/10.1214/18-STS648.

Tore Selland Kleppe. Dynamically rescaled Hamiltonian Monte Carlo for Bayesian hierarchical models. *Journal of Computational and Graphical Statistics*, 28(3):493–507, 2019. doi: 10.1080/10618600.2019.1584901. URL https://doi.org/10.1080/10618600.2019.1584901.

Tore Selland Kleppe. Connecting the dots: Numerical randomized Hamiltonian Monte Carlo with state-dependent event rates. *Journal of Computational and Graphical Statistics*, 2022a. forthcoming.

Tore Selland Kleppe. Log-density gradient covariance and automatic metric tensors for riemann manifold monte carlo methods, 2022b. Working paper.

Kjartan Kloster Osmundsen, Tore Selland Kleppe, and Roman Liesenfeld. Importance sampling-based transport map Hamiltonian Monte Carlo for Bayesian hierarchical models. *Journal of Computational and Graphical Statistics*, 30(4):906–919, 2021.