Concave-convex PDMP

Description

This package implements the concave-convex PDMP algorithm to facilitate sampling from distributions of interest (particularly Bayesian distributions).

These samplers move a particle with a state and velocity using deterministic dynamics. At random event times the velocity is updated and then the particle continues. The trajectories returned by the sampler define a Markov process sampling the distribution of interest.

Install instructions

Click on the .Rproj file and compile the package. Code is provided in the experiments section to reproduce the figures and tables of the results section from the paper.

Getting started

The cc-pdmp package allows for simulation of interesting Bayesian posteriors when the PDMP rate function can be bounded by a concave-convex decomposition. We assume linear dynamics for the PDMP sampler. The simplest way to start using the package is when the PDMP rate function can be bounded by a polynomial function. If this is the case then the concave-convex PDMP approach will be able to simulate using the concave-convex polynomial decomposition from Proposition 1 of the paper.

Simulating when the PDMP rate is a polynomial

Consider the Banana distribution,

$\pi(x_1, x_2) \propto \exp(-(x_1 - 1)^2 + \kappa(x_2 - x_1^2)^2)$

Taking the derivatives of the negative log gives the function shown below:

```
library(ccpdmp)
kappa <- 1 ## Arbitrary choice

dnlogpi <- function(x, index){
  x1 <- x[1]; x2 <- x[2]
  grad <- c(2*(x1-1) + 4*kappa*(x1^2-x2)*x1,  ## partial x_1
            2*kappa*(x2-x1^2))                ## partial x_2
  return(grad[index])
}
```

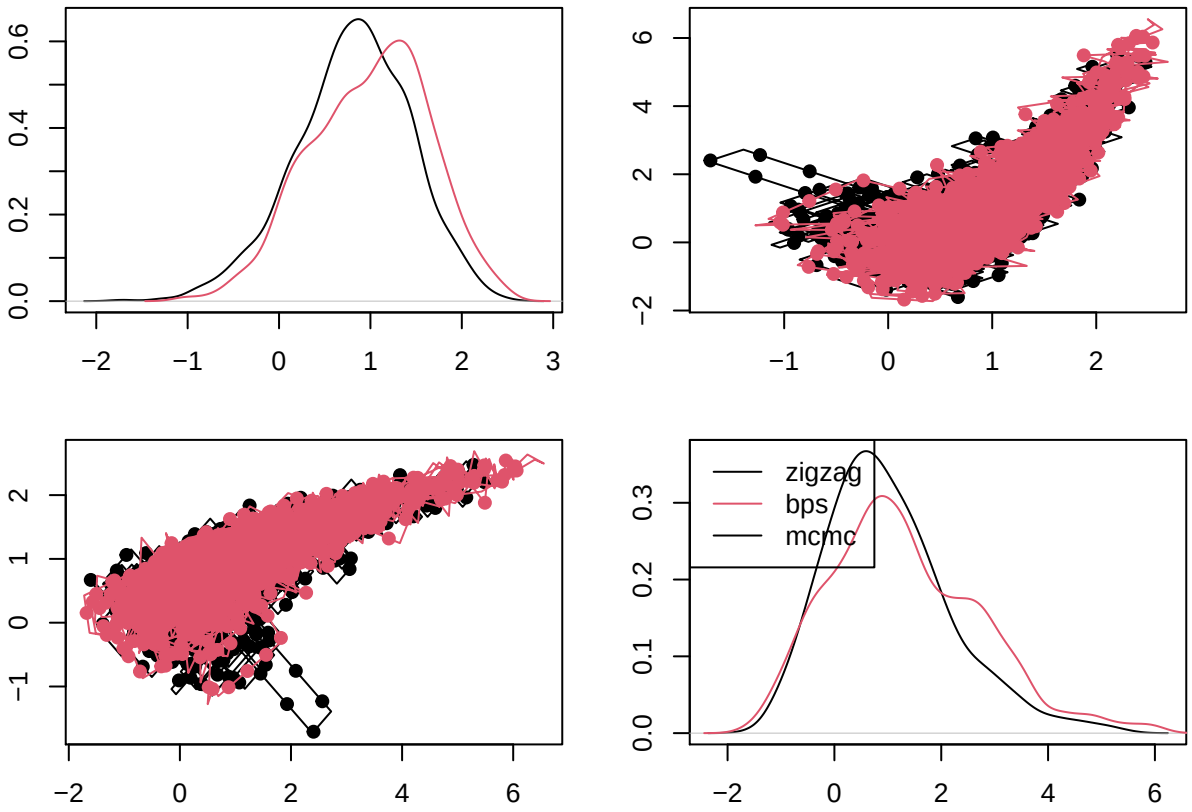
the rate function for the PDMP will depend on the terms

$f_1(t) = -v_1\partial_1 \log \pi(x + t) = v_1(2((x_1 + tv_1) - 1) + 4\kappa((x_1 + tv_1)^2 - (x_2 + tv_2))(x_1 + tv_1))$

$f_2(t) = -v_2\partial_2 \log \pi(x + t) = v_2(2\kappa((x_2 + tv_2) - (x_1 + tv_1)^2)$

So, f_1 is a polynomial of order 3 and f_2 is a polynomial of order 2. Rather than finding these terms exactly the ccpdmp package can evaluate these polynomials at 4 points $t \in [0, \tau_{\max})$ and interpolate to find the coefficients. The user just needs to specify a maximum polynomial order.

```
set.seed(0)
z <- zigzag(1e3, dnlogpi, x0 = c(0,0), poly_order = 3)
b <- bps(1e3, dnlogpi, x0 = c(0,0), poly_order = 3)
plot_pdmp_multiple(list(zigzag=z, bps=b), nsamples = 1e3)
```



Both the Zig-Zag and BPS can be simulated using the same specification.

Simulating when the PDMP rate is bounded by a polynomial

In most situations the PDMP rate will not be exactly polynomial but can be bounded by a polynomial. Consider logistic regression where the gradient function is

```
# Return the partial derivative for a logistic regression
get_grad <- function(x, index){
  expa <- exp(X%*%x)
  phi_1 <- expa/(1+expa) - y

  grad <- rep(0, length(index))
  for(i in 1:length(index)){
    grad[i] <- sum(phi_1*X[,index[i]])
  }
  return(grad)
}
```

Following the notation from the paper this can be simulated using the bound $f(t) \leq f(0) + f'(0)t + f''(0)/2 + M\tau_{\max}^3/6$ where $f'''(t) \leq |M|$. The paper gives an explicit form for this here we show how this polynomial may be used in simulating the logistic regression example. In addition to supplying the gradient function the

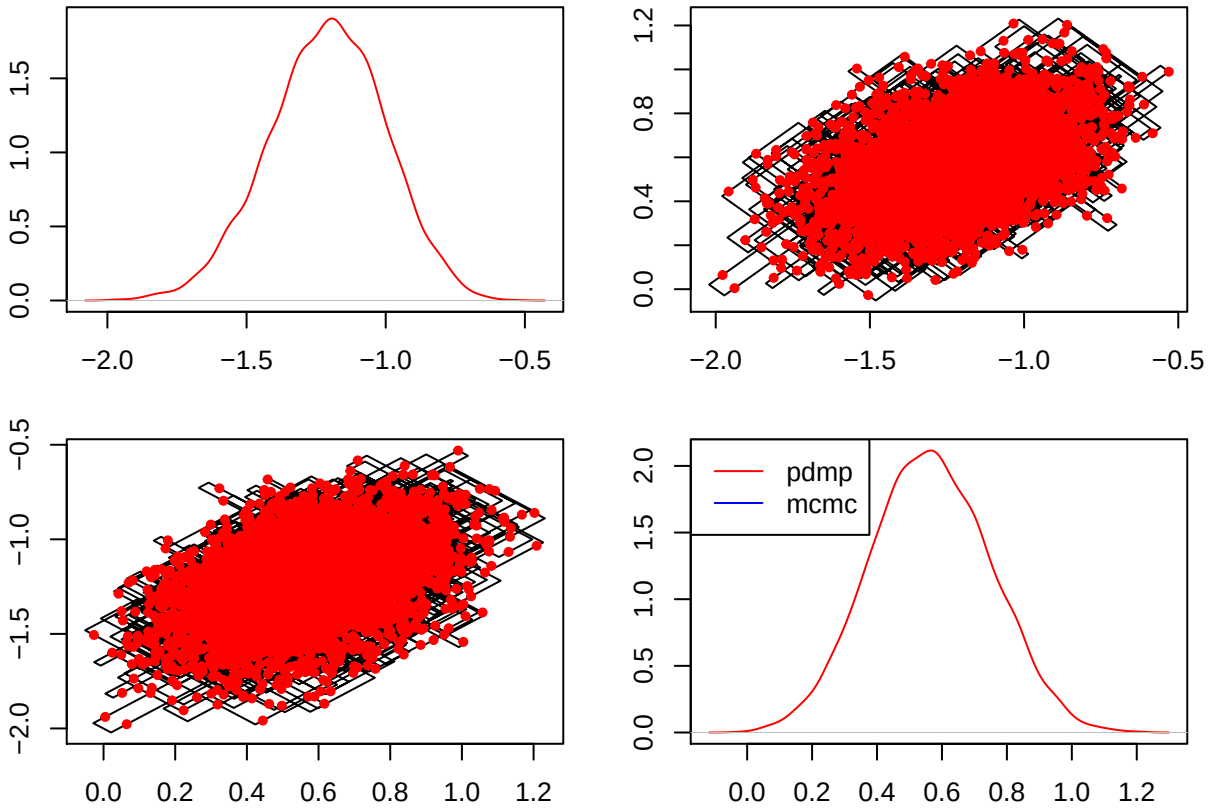
user must also specify a function that evaluates the polynomial upper-bounding rate.

```
## Return the rates for an upper-bounding 3rd order polynomial
return_rates <- function(x, theta, tau_grid, dnlogpi, rate_updates){

  tau_length <- length(tau_grid);
  n_rates <- length(rate_updates)

  rates_eval <- matrix(0, n_rates, tau_length)
  a <- X%*%x
  expa <- exp(a)
  da_dt <- X%*%theta

  # First 3 derivatives of phi(a) = log(1+exp(a)) - ya
  phi_1 <- expa/(1+expa) - y
  phi_2 <- expa/(1+expa)^2
  phi_3 <- -expa*(expa -1)/(expa+1)^3
  phi_4_bound <- 1/8

  for(i in 1:n_rates){
    ## Calculate f(t) and derivatives of f(t) for Taylor expansion
    partial_i <- rate_updates[i]
    f <- theta[partial_i]*sum(phi_1*X[,partial_i])
    f_1 <- theta[partial_i]*sum(phi_2*X[,partial_i]*da_dt)
    f_2 <- theta[partial_i]*sum(phi_3*X[,partial_i]*da_dt^2)
    f_3 <- sum(phi_4_bound*abs(X[,partial_i]*da_dt^3))

    # Evaluate f(t) < f(t) + f'(t) + f''(t)/2 + M/6 at all tau_grid points
    poly <- c(f_3/factorial(3), f_2/factorial(2), f_1, f)

    rates_eval[i,] <- pracma::polyval(poly, tau_grid)
  }
  return(rates_eval)
}
```

Example run:

```
generate.logistic.data <- function(beta, n.obs, siginv) {
  p <- length(beta)
  dataX <- mvtnorm::rmvnorm(n=n.obs*p,sigma = solve(siginv))
  vals <- dataX%*%beta
  generateY <- function(p) { rbinom(1, 1, p)}
  dataY <- sapply(1/(1 + exp(-vals)), generateY)
  return(list(dataX = dataX, dataY = dataY))
}
set.seed(1)
n_ev <- 1e3; p <- 3; n <- 100
beta <- c(c(-1.25, 0.5), rep(-0.4,p-2))
siginv <- diag(1, p,p)
siginv[1,2] <- siginv[2,1] <- 0.8
data <- generate.logistic.data(beta, n, siginv)
X <- data$dataX
y <- data$dataY

z <- zigzag(5e3, dnlogpi = get_grad, return_rates = return_rates,x0 = beta, poly_order = 3)
```

```
plot_pdmp(z, nsamples = 5e3, inds = 1:5e3)
```



Local methods and efficient C++ implementations

More efficient versions of the samplers are also available though they require specifying the concave-convex decomposition using a C++ implementation. The code below defines a cpp function which takes in the current position and velocity (x, theta) data y and any additional data from a list and returns a matrix with concave convex decomposition. In this case the distribution is a Gaussian.

```
library(RcppXPtrUtils)
```

```
## Warning: package 'RcppXPtrUtils' was built under R version 4.0.5
```

```
local_rate <- cppXPtr("arma::vec get_rate(double t, arma::vec& t_old, arma::vec& x, arma::vec& theta,
                      const arma::vec& y, const List& Data, arma::uvec rate_inds, bool grad) {

                    // Standard header //
                    int num_terms = rate_inds.size(), rsize = 5;
                    double x_t, grd;
                    if(grad){
                       rsize += num_terms;
                    }
                    arma::vec res = arma::zeros(rsize);

                    // Evaluate the rate: (f_u, f_n, f_n', p, f) for the factor
                    // Return additional gradient terms if grad = TRUE
```

```
                    for(int i = 0; i < num_terms; i++){
                         int partial = rate_inds(i);
                         x_t = x(partial) + theta(partial)*(t-t_old(partial));
                         grd = x_t;
                         res(0) += theta(partial)*grd;

                         if(grad){
                         res(5 + i) = grd;
                         }
                    }
                    res(4) = res(0) + res(1) + res(3);
                    return(res);
                    }", depends = c("RcppArmadillo"))
```

Once this has been set up the practitioner may specify a choice of Factors wich partition the parameter set and local updates indicating which factors need re-simulation once and event for the corresponding factor is triggered.

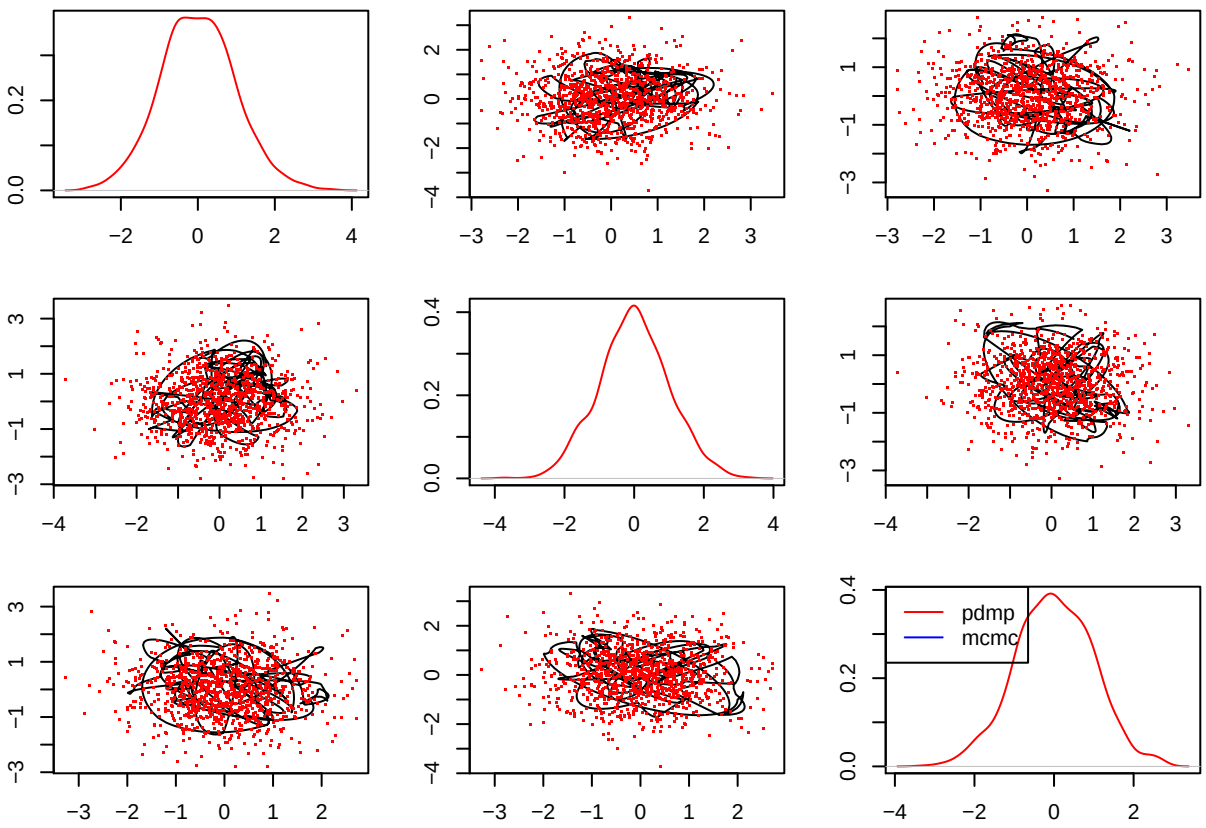
```
set.seed(0)
d <- 1000
x0<- rnorm(d)
theta0 <- rnorm(d)
nmax <- 10^5
ref_rate <- 0.05
Datann <- list()

## Standard BPS -- A single factor consisting of all variables
Factors <- list(c(0:(d-1)))
Neighbourhoods <- list(c(0)) # just resimulate the Factor 0 (the only factor)

system.time({set.seed(1);bps_global <- bps_cpp(maxTime = 2, # Run time
                                               trac_coords = c(0,1,d-1), # Coords to track
                                               rate_f = local_rate, # Cpp function
                                               factors = Factors, # List of factors
                                               local_updates = Neighbourhoods, # List of factors to upd
                                               Data = Datann, # Possible extra parameters
                                               y = y,         # Possible extra data
                                               nmax = d*3*10^2, # maximum number of events
                                               x0 = x0, # initial location
                                               theta0 = theta0, # initial velocity
                                               ref_rate = 1)})
```

```
##    user  system elapsed
##    1.95    0.00    2.01
```

```
length(bps_global$times)
```

```
## [1] 108786
```

```
plot_pdmp(bps_global, pch = '.',coords = c(1,2,3), nsamples = 1e3, inds = 1:1e3)
```



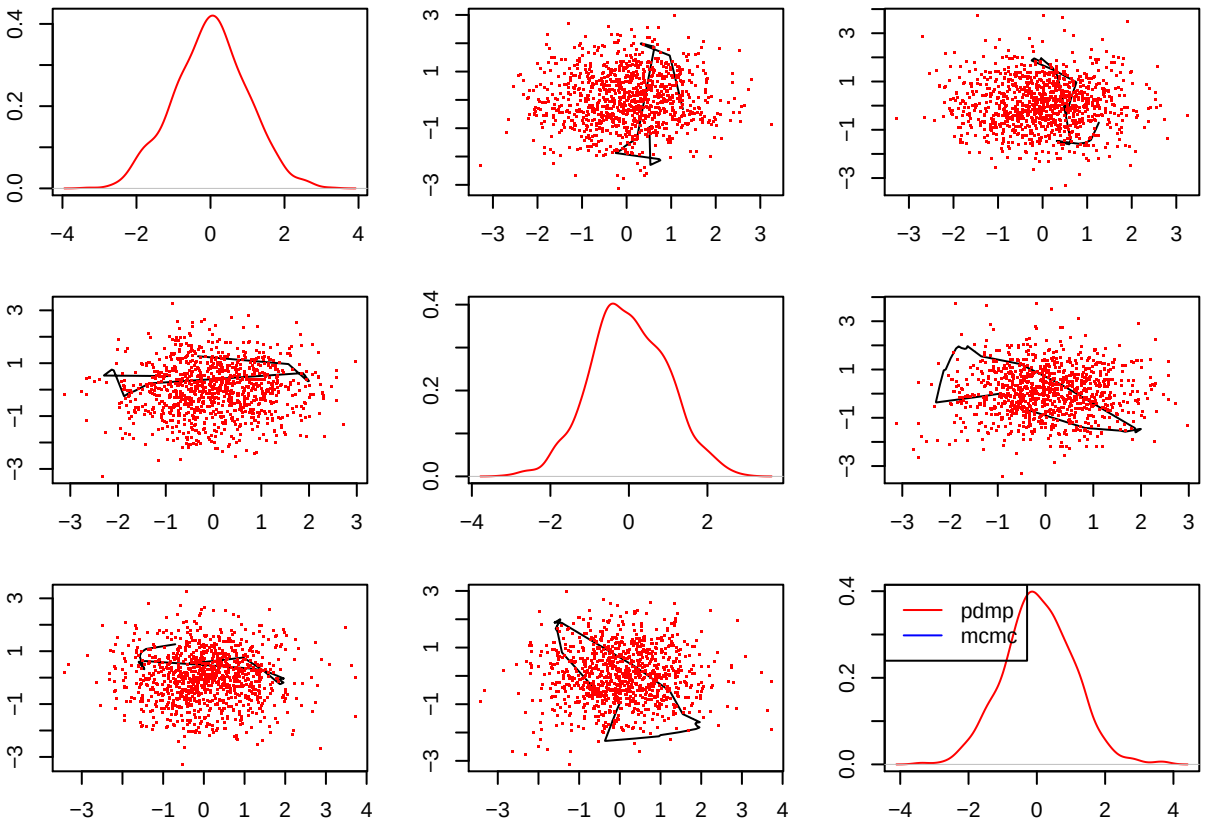

```
## local BPS -- 10 factors consisting of d/10 variables each
Factors <- lapply(0:(d/10-1), function(i) seq(from = 10*i, to = 10*i+9))
Neighbourhoods <- lapply(0:(d/10-1), function(i) c(i))

system.time({set.seed(1);bps_local <- bps_cpp(maxTime = 2, # Run time
                                              trac_coords = c(0,1,d-1), # Coords to track
                                              rate_f = local_rate, # Cpp function
                                              factors = Factors, # List of factors
                                              local_updates = Neighbourhoods, # List of factors to upd
                                              Data = Datann, # Possible extra parameters
                                              y = y,         # Possible extra data
                                              nmax = d*3*10^2, # maximum number of events
                                              x0 = x0, # initial location
                                              theta0 = theta0, # initial velocity
                                              ref_rate = 1)})
```

```
##    user  system elapsed
##    0.53    0.00    0.53
```

```
length(bps_local$times)
```

```
## [1] 300000
```

```
plot_pdmp(bps_local, pch = '.',coords = c(1,2,3), nsamples = 1e3, inds = 1:1e3)
```


pdmp
mcmc