*Note that we have moved our git depository since Milestone2 by TF's comments. New repo: https://github.com/CSExponentials/cs207-FinalProject.git

Introduction

Automatic Differentiation (AD) is a set of techniques to numerically evaluate derivatives of functions. AD is superior to classical computer-based differentiation procedures such as finite difference method in that AD is accurate up to machine precision and does not require the user to "tune" parameters such as step-size. AD is also much more efficient than symbolic differentiation.

If f(x) is a function, then its derivative f′(a) at point a, measures the sensitivity of f to small changes in x around a. Therefore, derivatives help us understand how a function changes. Since many scenarios in the world can be modelled as functions, derivatives help us understand these models. Since AD is a very efficient way to calculate derivatives, it is widely used.

This project implements the forward mode of AD using elementary operator override methods. In particular, our library allows users to evaluate the gradient/Jacobian of multivariable scalar/vector functions using AD up to machine precision. We also provide flexibility in computing either the entire Jacobian matrix J or its action on a particular vector p, i.e. Jp. The latter option is beneficial when the entire J is not required in application: computation of Jp is much more efficient due to the way we implement AD.

Along with the AD implementation, we also provide a unique feature: two gradient-based MCMC algorithms that use AD to compute the gradient of the target distribution. The first MCMC algorithm is Metropolis-Adjusted-Langevin Algorithm (MALA) that generates proposal at the direction of the gradient and the second MCMC algorithm is Hamiltonian Monte Carlo, which requires computing gradient when solving the Hamiltonian ODEs.

Background

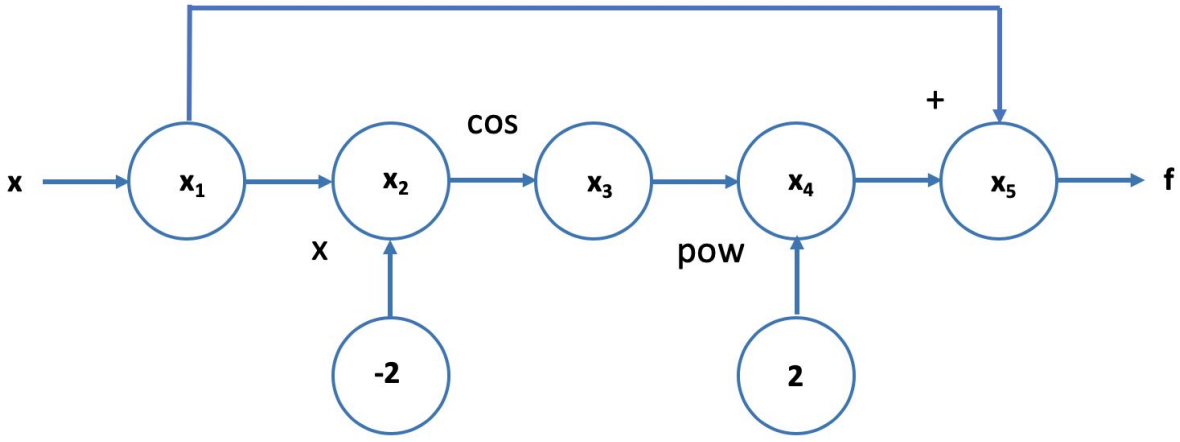
Automatic differentiation (AD) is a method for evaluating derivatives of functions. The method is fundamentally different from other computational differentiation approaches which use approximative finite difference methods or symbolic differentiation, and it overcomes limitations of both approaches. AD requires shorter computation times than

symbolic differentiation and computes derivatives up to machine precision while finite difference methods are prone to rounding errors or have limited robustness.

AD has a forward mode and a reverse mode. Here we will only describe the forward mode.

Let the function of interest be $f : R^n \rightarrow R^m$, consisting of elementary functions ϕ_i. Then, forward AD computes $J_f(c) \cdot p$ where $J_f(c)$ (abbr. J) is the Jacobian matrix of function f evaluated at c and p is arbitrary vectors. Note that the entire Jacobian J may be evaluated by computing the i-th column of J as $J_f(c) \cdot e_i$ where e_i denotes a vector with i-th entry set to 1 and 0 otherwise.

Essentially, f is understood as a composition of elementary functions such as `sin`, `cos`, `exp`. The procedure may be understood conceptually in terms of a computational graph. The figure below demonstrates how a computational graph is constructed for the example $f = x + cos^2(-2x)$.



The program stores the analytic derivative of these elementary functions and the differentiation of f is obtained by applying the chain rule successively. Each node of the graph stores two real scalars: value and derivative. This allows to generate the evaluation trace (shown below) and compute the derivative of the function. Here we calculate the derivative of the function at $f'(\pi/2)$ where $f'(x) = \partial f/\partial x$.

	value	derivative	numerical
x_1	π/2	1	val: π/2, der: 1

x_2	$-2x_1$	$-2\dot{x}_1$	val: -π, der: -2
x_3	$\cos(x_2)$	$-\sin(x_2)\dot{x}_2$	val:-1, der: 0
x_4	x_3^2	$2x_3\dot{x}_3$	val: 1, der: 0
x_5	x_1+x_4	$\dot{x}_1+\dot{x}_4$	val: 1+π/2, der: 1

The forward AD initiates at n nodes, each representing a variable in f: the i-th node's value and derivative are set to be `c[i]` and `p[i]`. New nodes are then "composed" in exact order the sub-terms of f are composed, starting from n variables, from elementary functions.

To illustrate, let's denote a new node as s_{k+1} and old nodes as $s^{(i)}{}_k$. The value of a new node is simply the elementary function applied to value of old node(s) :

$s_{k+1}[value] \leftarrow \phi_i(s_k[value])$ or

$s_{k+1}[value] \leftarrow \phi_i(s_k^{(1)}[value],\ s_k^{(2)}[value])$

Whereas the derivative is calculated by applying the chain rule to the accumulated value and derivative fields

$s_{k+1}[derivative] \leftarrow \phi_i'(s_k[value])s_k[derivative]$ or

$s_{k+1}[derivative] \leftarrow \partial/\partial x\ \phi_i(s_k^{(1)}[value],\ s_k^{(2)}[value])\cdot s_k^{(1)}[derivative]]$

$+\ \partial/\partial y\ \phi_i(s_k^{(1)}[value],\ s_k^{(2)}[value])\cdot s_k^{(2)}[derivative]]$

New nodes are generated automatically by virtue of each elementary function and overridden to evaluate the value and derivative as above; and the program should "trace out" the entire computational graph by simply evaluating f in the natural arithmetic order.

How to Use AutoJac

How to use your package

Installation

If the user is interested to install the entire package that includes the unique MCMC features, then they should run the following command:

```
pip install -i https://test.pypi.org/simple/ AutoDiffMCMC_ExponentialsFinal
```

If the user intends to use only the AD functionality (without unique features), they can install the following version:

```
pip install -i https://test.pypi.org/simple/ AutoDiff_Exponentials
```

The following are the steps needed to install our source code, including our advanced feature:

1. Clone our repository by running:

   ```
   git clone https://github.com/CSExponentials/cs207-FinalProject.git
   ```

2. Install our dependencies by running:

   ```
   pip install -r requirements.txt
   ```

Demos

If the user wishes to git clone and run our source code, the easiest way for them to do so is to go to the Demos folder under the root directory and run the AD_Demo.py (Demo for AD functionality) and the MCMC_Demo.py.

Otherwise, after you install the package, you can first import the modules:

```
>>> import AD

>>> import AD.ElemFunc as EF
```

```
>>> from AD.ADiff import ADiff

>>> import pprint as pp
```

And then, users will define the function of interest as follows:

```
>>> def myfunc(x,y):
...     f=[x*y+EF.sin(x),x+y+EF.sin(x*y)]
...     return f
```

where `x,y` can be any other variable names, number of variables and dimension of vector function is arbitrary.

Note that the program is robust enough for function handles of various forms (e.g. input below is completely equivalent to input above):

```
>>>def myfunc(x,y):
...     f1=x
...     f2=EF.sin(x)
...     f3=10
...     f4=x+y+EF.sin(x*y)+10
...     return [y*f1+f2, -(f3-f4)]
```

Users will instantiate a `ADiff` object (`ADiff` is the interface class under module `AD`):

```
>>> f_obj = ADiff(myfunc) # Instantiate the interface
```

Then users will have the option to evaluate the entire Jacobian J at any point c:

```
>>> c = [1,2] # Jacobian evaluated at c=[1,2]
>>> res = f_obj.Jac(c)
```

The output `res` is a dictionary with keys: `val, diff. val` is the value of the function evaluated at `c, diff` is Jacobian matrix:

```
>>> res
{
'value': [2.841, 3.909]
'diff': [[2.540, 1.000],
         [0.168, 0.584]],
}
```

Or they can just evaluate `Jp` for any vector `p`. This method is more efficient than computing `J` and compute `Jp` again via matrix multiplication:

```
>>> p = [2,3]
>>> res = f_obj.pJac(c, p) % Outputs the vector Jp and value of function at c
```

The output is a dictionary where `val` is the value of function at `c` and `diff` is `Jp`:

```
>>> res
{
'value': [2.841, 3.909]
'diff':  [8.081, 2.087]
}
```

Demo for our unique features: MCMC

In case the user has already downloaded the source code from Github, you can go to the Demos folder and run the MCMC_Demo.py.

If the user just installs the package from pip:

```
pip install -i https://test.pypi.org/simple/ AutoDiffMCMC_ExponentialsFinal
```

then they may do a simple testing by running the following commmand

```
import AD

import MCMC

import MCMC.HMC as HMC

import MCMC.MALA as MALA

import numpy as np

import AD.ElemFunc as EF

# Used for diagonosit output

def diagfun(samples):

    return np.mean(samples,1)

# Distribution to sample from (target distribution)

def target(x,y):

   return EF.exp(-(1-x)**2-10*(y-x**2)**2)

# Print diagnostics
```

```
def printVals(sampler,samples):

    print(sampler.getAcceptRatio())

    print(sampler.getAvgMovesize())

    print(sampler.getVarMovesize())

    sampler.plotSamples(samples)


# Sampling using HMC

sampler=HMC.HMCSampler(target, ep=0.05, L=100)

samples=sampler.sample(steps_=2000,  X0=np.zeros(2), liveoutput=200)

printVals(sampler,samples)

# Sampling using MALA

sampler=MALA.MALASampler(target, tau=0.02)

samples=sampler.sample(steps_=100000, X0=np.zeros(2), liveoutput=2000,
diagfun=diagfun)

printVals(sampler,samples)
```

Software Organization

Directory Structure

```
AutoJac/
       AD/
         __init__.py
         AutoDiff.py
         ADiff.py
         ElemFunc.py
       MCMC/
         HMC.py
         MALA.py
         Sampler.py
       Demos/
```

```
  AD_Demo.py
  MCM_Demo.py
.travis.yml
LICENSE.txt
requirements.txt
README.md
setup.py
tests/
     test_AD.py
     test_MCMC
```

At the root of the project, we will include general files such as `README.md, .travis.yml, requirements.txt`, and packaging and distribution files.

The AD sub-directory will include the files that implement automatic differentiation:

- `ADiff` module: implement `ADiff` class, which is user-interface
- `AutoDiff` module: implement `AutoDiff` class, which overloads built-in python functions
- `ElemFunc` module: implement elementary functions such as `sin, cos, exp`

A sub-directory tests contains the files that test the rest of the code.

Modules and Basic Functionalities

AutoDiff class overloads python built-in functions. Other elementary functions `sin, sqrt, log,` and `exp` are defined in ElemFunc class. We also require an ADiff class, which serves as a user interface: it receives and stores user inputs, and assembles results from AutoDiff, ElemFunc to form output of various sorts (e.g. entire Jacobian J, or Jp or function values).

Testing

Our test suite will live under a subdirectory `tests` . We will indeed use continuous integration tools such as TravisCI and CodeCov. To manually run the tests, the user must first install the package (refer to “How to installAutoJav” above). Once installed, the user can run the following command from the `cs207-FinalProject` directory to run all of the tests, see their coverage, and the missing lines:

```
pytest --cov --cov-report term-missing
```

Implementation

Use Cases

We support vector or scalar functions as input. Our interface provides choices of outputs as either an entire Jacobian matrix or the product of a Jacobian matrix and a vector where the latter is computationally efficient. Note that the implementation is based on the idea of a seed vector p at the end of lecture 10.

Core Data Structure

We use only Python Lists, Dictionaries and Class structures for the development of the forward mode:

- We use List object to store matrices and vectors instead of third party data structures such as numpy array. Since no matrix computation is involved in the forward mode, we opt to avoid unnecessary dependencies on external libraries.
- We define the output of our computation as a Dictionary where we store the value of the function and derivative/gradient/Jacobian of the function.
- See next section for our use of class structure and their purposes.

Classes to implement, Core Flow

AutoDiff class overloads python built-in functions. Other elementary functions `sin`, `sqrt`, `log`, and `exp` are defined in ElemFunc class. We also require a ADiff class, which serves as a user interface: it receives and stores user inputs, and assembles results from AutoDiff, ElemFunc to form output of various sorts (e.g. entire Jacobian J, or Jp or function values).

The core flow of the program (implemented within interface `ADiff.pJac`) to evaluate `Jp` is as follows (entire `J` is easily implemented by calling this procedure with `p` set as `[1,0,0,..], [0,1,0,0,...]...`):

Step 1. Receive function of interest from user:

```
>>> def myfunc(x,y):
...     f=[x*y+EF.sin(x),x+y+EF.sin(x*y)]
...     return f
```

Step 2. Receive `c, p` from user

Step 3. Count number of variables of the function using

```
>>> varNum = myfunc.__code__.co_argcount
```

Step 4. Construct `varNum` of `AutoDiff` instances and put them in a list, each initialized with `p[i], c[i]`

```
>>> varList=[AutoDiff(c[i], p[i]) for i in range (varNum)]
```

Step 5. Pass this list as arguments to `myfunc`.

```
>>> myfunc(*varList)
```

Notice that we simply construct AutoDiff class instances based on number and order of inputs of myfunc. It is more natural to let user pass their function as a function handle instead of string. Also, this design saves the trouble of parsing variable names and associated exception handling.

Attributes for Classes

AutoDiff Class

The `AutoDiff` class is analogous to `AutoDiffToy` class in HW4. The idea is that each (i-th) variable in the input function is to be initialized as an instance of the AutoDiff class with seed `p[i]` and variable values `c[i]`.

The constructor of `AutoDiff` receives two inputs, which are primary instance variables of this class:

- `val`: a **scalar** real number initialized as `c[i]`, representing the "value column" in the evaluation trace;
- `der`: a **scalar** real number initialized as `p[i]`, representing the "directional derivative column" in the evaluation trace.

Suppose we are interested in the directional derivative `Jp` of a function for any vector `p` evaluated at `c`. Function variables `x, y` are initialized as instances of this class as follows.

```
>>> p=[2,3]
>>> c=[1,2]

>>> x=AutoDiff(c[0],p[0])
>>> y=AutoDiff(c[1],p[1])
```

This is exactly what happens in Step 3 in `ADiff.pJac` outlined above. Indeed, the i-th entry in `f_obj` below is an `AutoDiff` instance with `val` being the i-th entry of the function value at `c` and `der` being the i-th entry of `Jp`.

```
>>> f_obj=[x*y+EF.sin(x),x+y+EF.sin(xy)]
```

Class methods of `AutoDiff` class are Dunder methods overloading python built-in elementary operations. In particular, each of these methods **returns a new `AutoDiff` instance** with the appropriate `val, der` computed according to the following table, which are just elementary derivatives with the chain rule. **Note that the format of the pseudo-code below is: `AutoDiff(val, der)` where `val, der` are the appropriate `val, der` fields based on input `self, other` for each overloaded method to return, analogous to HW4.** We differentiate implementation of input of AutoDiff type or real scalar type. This is done by the Duck Typing method covered in lecture and HW4.

	Return when other is AutoDiff type	Return when other is real scalar
`__add__(self, other)`	`AutoDiff(self.val+other.val, self.der+other.der)`	`AutoDiff(self.val+other, self.der)`
`__radd__(self, other)`	N/A	”
`__sub__(self, other)`	`AutoDiff(self.val-other.val, self.der-other.der)`	`AutoDiff(self.val-other, self.other)`
`__rsub__(self,other)`	N/A	`AutoDiff(other-self.val,-self.der)`
`__mul__(self, other)`	`AutoDiff(self.val*other.val, self.val*other.der+self.der*other.val)`	`AutoDiff(self.val*other, self.der*other)`
`__rmul__(self,other)`	N/A	”
`__truediv__(self, other)`	`AutoDiff(self.val/other.val, (self.der*other.val-other.der*self.val)/(other.val**2))`	`AutoDiff(self.val/other, self.der/other)`
`__rtruediv__(self, other)`	N/A	`AutoDiff(other/self.val,-other*self.der/(self.val**2))`

*Note that this table is not complete: we have omitted more complicated expressions such as __pow__(), __rpow__(). The idea is the same.

ElemFunc Class

The other elementary functions such as `exp, sin, cos` are implemented in ElemFunc Classes -- all these functions return a new `AutoDiff` object same as the Dunder methods, with corresponding `val` and `der`. They can handle either real or AutoDiff type input with duck typing, similar to Duder methods.

	x is AutoDiff object	x is real number
exp(x)	AutoDiff(math.exp(x.val), math.exp(x.val)*x.der)	AutoDiff(math.exp(x),0)
exp_base(y,x)	AutoDiff(math.pow(y,x.val), math.pow(y,x.val)*x.der)	AutoDiff(math.pow(y,x),0)
log(x)	AutoDiff(math.log(x.val), (1/(x.val)*x.der)	AutoDiff(math.log(x),0)
Trigonometric functions		
sin(x)	AutoDiff(math.sin(x.val), math.cos(x.val)*x.der)	AutoDiff(math.sin(x),0)
cos(x)	AutoDiff(math.cos(x.val), -math.sin(x.val)*x.der)	AutoDiff(math.cos(x),0)
tan(x)	AutoDiff(math.tan(x.val), (2/(math.cos(x.val*2)+1))*x.der)	AutoDiff(math.tan(x),0)
cot(x)	AutoDiff(math.cos(x.val)/math.sin(x.val), (2/(math.cos(x.val*2)-1))*x.der)	AutoDiff(math.cos(x.val)/math.sin(x.val),0)
sec(x)	AutoDiff(1/(math.cos(x.val)), (math.tan(x.val)*1/(math.cos(x.val))*x.der))	AutoDiff(1/math.cos(x),0)
csc(x)	AutoDiff(1/math.sin(x.val), (-1/math.sin(x.val)*math.cos(x.val)/math.sin(x.val))*x.der)	AutoDiff(1/math.sin(x),0)
arcsin(x)	AutoDiff(math.asin(x.val), (1/math.sqrt(1-(x.val)*(x.val)))*x.der)	AutoDiff(math.asin(x),0)

arccos(x)	AutoDiff(math.acos(x.val), (-1/math.sqrt(1-(x.val)*(x.val)))*x.der)	AutoDiff(math.acos(x),0)
arctan(x)	AutoDiff(math.atan(x.val), (-1/(1+(x.val)*(x.val))*x.der))	AutoDiff(math.atan(x),0)
Hyperbolic functions		
cosh(x)	AutoDiff(((math.exp(x.val)+math.exp(-x.val))/2), ((math.exp(x.val)-math.exp(-x.val))/2)*x.der)	AutoDiff(((math.exp(x)+math.exp(-x))/2),0)
sinh(x)	AutoDiff(((math.exp(x.val)-math.exp(-x.val))/2), ((math.exp(x.val)+math.exp(-x.val))/2)*x.der)	AutoDiff(((math.exp(x)-math.exp(-x))/2),0)
tanh(x)	AutoDiff(((math.exp(x.val)-math.exp(-x.val))/2)/((math.exp(x.val)+math.exp(-x.val))/2), ((1/((math.exp(x.val)+math.exp(-x.val))/2)*((math.exp(x.val)+math.exp(-x.val))/2))*x.der))	AutoDiff(((math.exp(x)-math.exp(-x))/2)/((math.exp(x)+math.exp(-x))/2),0)

ADiff Class

This class mainly serves as a user-interface. The constructor receives one input, which are instance variable of this class:

- `func`: a user defined function

```
>>> def myfunc(x,y):
...     f=[x*y+EF.sin(x),x+y+EF.sin(x*y)]
...     return f
>>> f.obj = ADiff(myfunc)
```

There are two methods in this class:

- `Jac(c)`: Return a dictionary with value of the function at `c` and entire Jacobian `J` at `c`
- `pJac(c,p)`: Return a dictionary with value of function at `c` and Jacobian acting on `p`, i.e. `Jp`

The implementation of `pJac` is given in previous section (Classes to implement, core flow). Jac fills i-th column of Jacobian matrix by calling upon `pJac` with `p` set to be a vector with 1 at i-th entry and 0 otherwise.

External Dependencies

We will rely on math and pytest (for testing). For advanced features, we will likely use numpy and are likely to evaluate our results to those implemented in standard library such as scikit-learn.

Implementing elementary functions like `sin, sqrt, log,` and `exp`

The Python built-in arithmetic operators such as “+”, “-”, “**” are implemented by overriding corresponding Dunder methods. Others such as `sin, sqrt, log,` and `exp` are implemented in Elemfunc Class. See previous sections for detail.

The Advanced Feature: Gradient-Based Markov Chain Monte Carlo (MCMC) algorithms using AD

Our extension and its purpose

Along with the AD package, we include implementation of two of the most popular gradient-based MCMC algorithms: Metropolis-adjusted Langevin Algorithm (MALA) and Hamiltonian Monte Carlo (HMC). Both of these methods serve essentially the same purpose of drawing samples from complicated, high-dimensional distributions such as Bayesian posterior and Ising models. Let us denote target distribution as $\pi(x) = \pi_u(x)/Z,\ x\ \in R^d$ where Z is a usually unknown normalizing constant. We assume that we are able to evaluate $\pi_u(x)$ for both algorithms. Gradient-based MCMC such as MALA and HMC uses first-order derivative of $\log \pi_u(x)$ as they determine “a proposed move” and thus is able to outperform more basic MCMC algorithms such as Metropolis-Hastings where such proposed move does not take into consideration of the local information of $\pi_u(x)$. In most practical use cases, analytic form of $\log \pi_u(x)$ is not available and a finite-difference method is often used to approximate it. Therefore, an AD based sampling package may be appealing to prospective users given all computational advantage of automatic differentiation we have discussed.

Metropolis-Adjusted Langevin Algorithm (MALA)

Metropolis-Adjusted Langevin Algorithm (MALA) is essentially a discretized process based on the Langevin diffusion for a multivariate distribution $\pi_u(x)$. In short, a Langevin diffusion for a multivariate distribution $\pi_u(x)$ is a non-explosive diffusion which is reversible to π --it makes use of the gradient of π to move more often in directions in which π is increasing. Thus, a discrete approximation to a Langevin diffusion should have a higher acceptance probability than for random-walk proposals as is the case for Metropolis Hastings.

The algorithm may be summarized as follows:

At step t of the algorithm, given state from previous state X_{t-1}, the Markov kernel samples X_t as follows:

1. Sample $X^* \sim N(X_{t-1} + \tau \nabla \log \pi_u(X_{t-1}),\ 2\tau I)$
2. Compute acceptance ratio $\alpha = \min(1,\ \frac{\pi(X^*)q(X_{t-1}|X^*)}{\pi(X_{k-1})q(X^*|X_{t-1})})$ where $q(x|x') = \exp(-\frac{\|x'-x'\tau \nabla \log \pi(x)\|_2^2}{4\tau})$
3. Let U be uniformly distributed in $(0, 1)$. If $U < \alpha$, then set $X_t = X^*$, otherwise set $X_t = X_{t-1}$.

Note that the values of τ is the tuning parameter. $X_t,\ t = 1, \ldots T$ are the samples we are supposed to collect from our target $\pi(x)$.

Hamiltonian Monte Carlo

To give some intuition regarding this algorithm, $H(x, p) = -\log \pi_u(x) + 1/2\ p^T M^{-1} p$ here corresponds to the "Hamiltonian" in classical mechanics, which is the sum of potential ($-\log \pi_u(x)$) and kinetic energy ($1/2\ p^T M^{-1} p$). Indeed, we regard each sample draw X_t as a moving particle that abides by the law of motion referred to as Hamiltonian dynamics. The theory then suggests that the position of the particle X_t at each t, as dictated by the Hamiltonian ODEs, would produce "asymptotically correct" samples of $\pi(x)$. The second step of the algorithm is essentially solving the trajectory of this particle via the

so-called “leap-frog” (or Stormer-Verlet) scheme which is where the gradient (and thus automatic differentiation) comes in.

Unlike more basic Monte Carlo algorithms such as Metropolis Hastings or Gibbs sampler, it also requires that we are able to evaluate $\nabla \log \pi_u(x)$. The second requirement is often computationally challenging since $\pi_u(x)$ is typically of high dimension (i.e. d is large) and that the precision at which we are able to evaluate the gradient of $\log \pi_u(x)$ has a significant impact on the performance of the algorithm. Automatic differentiation is thus a potentially suitable solution.

HMC proceeds as follows (roughly):

Define $H(x,p) = -\log \pi_u(x) + 1/2\, p^T M^{-1} p$ for semidefinite positive matrix M. At step t of the algorithm, given state from previous state X_{t-1}, the Markov kernel samples X_t as follows:

4. Set $Q(0) = X_{t-1}$, sample $P(0) \sim N(0, M)$
5. Leap-frog ODE solver that solves Hamiltonian flow numerically
    a. For $l = 0, ..., L-1$,
        i. $P(l+1/2) = P(l) + \varepsilon/2\ \nabla \log \pi_u(Q(l))$.
        ii. $Q(l+1) = Q(l) + \varepsilon M^{-1} P(l+1/2)$.
        iii. $P(l+1) = P(l+1/2) + \varepsilon/2\ \nabla \log \pi_u(Q(l+1))$.
6. Set $X^* = Q(L)$ and $P^* = P(L)$, and compute $\Delta^* = H(Q(0),\ P(0)) - H(X^*,\ P^*)$.
7. Let U be uniformly distributed in $(0, 1)$. If $\log(U) < \Delta^*$, then set $X_t = X^*$, otherwise set $X_t = X_{t-1}$.

Note that the values of $\varepsilon,\ M$ and L are tuning parameters. $X_t,\ t = 1, ...T$ are the samples we are supposed to collect from our target $\pi(x)$.

Structure of the extension and Implementation

The implementation will be incorporated in a separate directory “MCMC” and it will be shipped with the forward AD module we have already implemented. The user can install our package and simply import MCMC module (See “How to Use”). The structure is such that there is a parent class Sampler and two child classes for each of the two samplers (HMC, MALA) that inherits it.

The constructor for both sampling methods requires user to specify:

- target: the target distribution as a function
- Parameters associated with the sampler: τ for MALA, ε, L, M for HMC

Each sampler class has the `sampling` method where it accepts the following parameters

- `steps_`: total number of samples to obtain
- `X0`: the initial state for the MCMC algorithm
- `burnin`: the number of initial steps to discard
- `liveoutput`: negative input indicates no live output of the status of the sampler; positive input indicates number of steps per update
- `diagfun`: the diagnostic function to use for the live output (See demo)

The output `XmpHa` contains generated samples.

Refer to “How to Use” section for a clear demonstration of how to interact with the sampler.

Future Directions

Regarding to Automatic Differentiation, a clear direction would be to also develop the backward mode and apply it to our unique features.

Additional directions that can be taken to further develop of our unique feature are:

- More gradient-based MCMC algorithm such as the class of “stochastic gradient MCMC“, which are now quite popular with large data sampling
- Provide the users with more informative statistics such as the autocorrelation of the sampled points
- Add additional features such as a convergence criteria to complete the sampling.
- Create a user interface so that the user can adjust the parameters on-the-fly depending on the printed diagnostics. This corresponds to a type of adaptive MCMC but based on human judgement, which can often be valuable for practical use scenarios.