anno-domini Documentation

Release 0.0.1

simon,kaela,yoon,david

Dec 10, 2019

Contents

CHAPTER 1

Introduction

Calculating the derivative and gradients of functions is essential to many computational and mathematical fields. In particular, this is useful in machine learning because these ML algorithms are centered around minimizing an objective loss function. Traditionally, scientists have used numerical differentiation methods to compute these derivatives and gradients, which potentially accumulates floating point errors in calculations and penalizes accuracy.

Automatic differentiation is an algorithm that can solve complex derivatives in a way that reduces these compounding floating point errors. The algorithm achieves this by breaking down functions into their elementary components and then calculates and evaluates derivatives at these elementary components. This allows the computer to solve derivatives and gradients more efficiently and precisely. This is a huge contribution to machine learning, as it allows scientists to achieve results with more precision.

CHAPTER 2

Background

In automatic differentiation, we can visualize a function as a graph structure of calculations, where the input variables are represented as nodes, and each separate calculation is represented as an arrow directed to another node. These separate calculations (the arrows in the graph) are the function's elementary components.

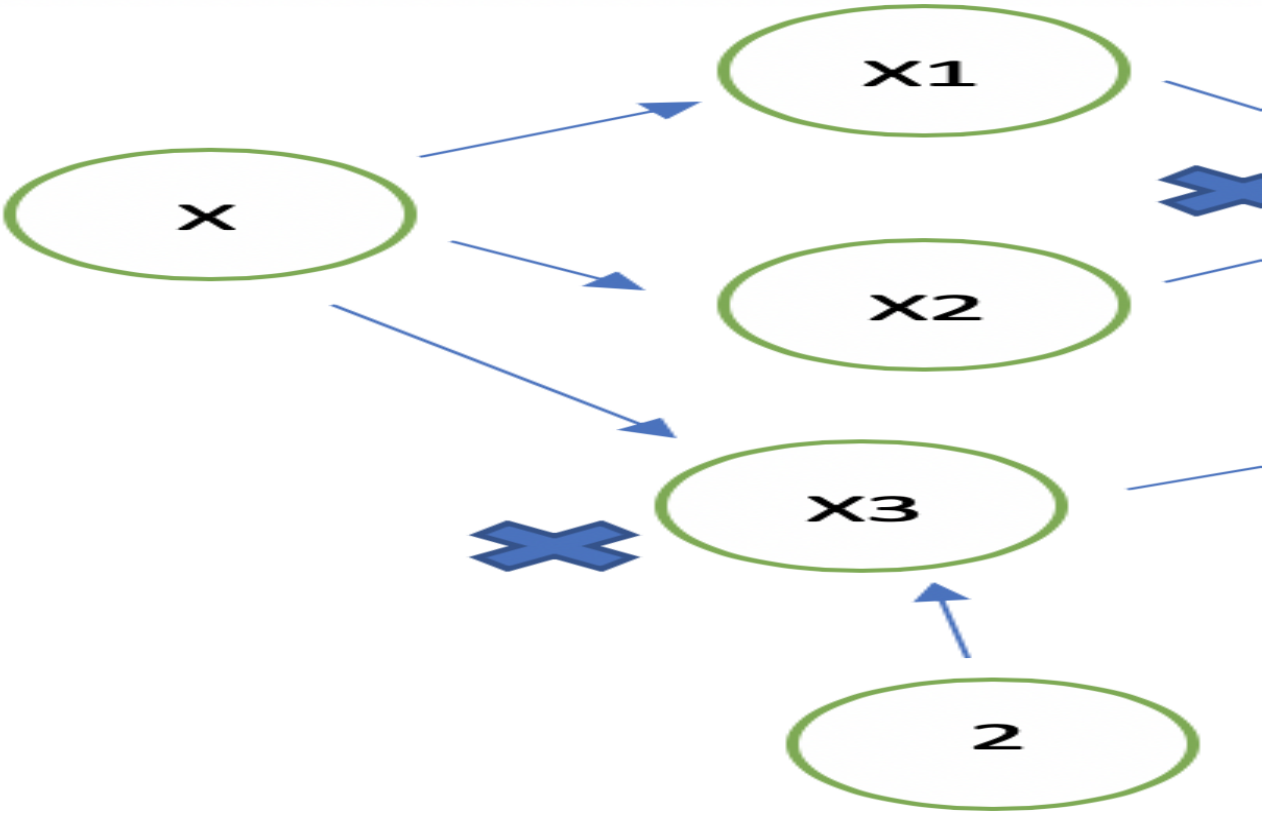
We then are able to compute the derivatives through a process called the forward mode. In this process, after breaking down a function to its elementary components, we take the symbolic derivative of these components via the chain rule. For example, if we were to take the derivative of $\sin(x)$, we would have that $\frac{d}{dx}\sin(x) = \sin'(x)x'$, where we treat "x" as a variable, and x prime is the symbolic derivative that serves as a placeholder for the actual value evaluated here. We then calculate the derivative (or gradient) by evaluating the partial derivatives of elementary functions with respect to each variable at the actual value.

For further visualization automatic differentiation, consider the function, $x^2 + 2x + 1$. The computational graph for this function looks like:

The corresponding evaluation trace looks like:

Trace	Elementary Function	Current Function value	Elementary Function Derivative	$f(1)$
x_1	x_1	x	1	1
x_2	x_2	x	1	1
x_3	x_3	x	1	1
x_4	x_1x_2	x^2	$\dot{x_1}x_2 + x_1\dot{x_2}$	2
x_5	$x_4 + 2x_3$	$x^2 + 2x$	$\dot{x_4} + 2\dot{x_3}$	4
x_6	$x_5 + 1$	$x^2 + 2x + 1$	$\dot{x_5}$	4

For the single output case, what the forward model is calculating the product of gradient and the initializaed vector p, represented mathematically as $D_p x = \Delta x \cdot p$. For the multiple output case, the forward model calculates the product of Jacobian and the initialized vector p: $D_p x = J \cdot p$. We can obtain the gradient or Jacobian matrix of the function through different seeds of the vector p.


x1
x
x2
x3
2

CHAPTER 3

How to use Anno Domini

3.1 How to Install

Internal Note: How to Publish to Pip

```
$ python setup.py sdist
$ twine upload dist/*
```

Install via Pip:

```
pip install AnnoDomini
```

Install in a Virtual Environment:

```
$ pip install virtualenv # If Necessary
$ virtualenv venv
$ source venv/bin/activate
$ pip install numpy
$ pip install AnnoDomini
$ python
>>> import AnnoDomini.AutoDiff as AD
>>> x = AD.AutoDiff(3.0)
>>> print(x)
====== Function Value(s) ======
3.0
===== Derivative Value(s) =====
1.0
>>> quit()
$ deactivate
```

Note: For using additional features, SciPy and tqdm packages are also required.

3.2 Basic Demos

3.2.1 1. Single Variable, Single Function ($\mathbb{R}^1 \rightarrow \mathbb{R}^1$)

Suppose we want to find the derivative of $x^2 + 2x + 1$. We can utilize the AnnoDomini package as follows:

```
>>> x = AD.AutoDiff(1.5)
>>> print(x)
====== Function Value(s) ======
1.5
===== Derivative Value(s) =====
1.0
>>> f = x**2 + 2*x + 1
>>> print(f)
====== Function Value(s) ======
6.25
===== Derivative Value(s) =====
5.0
```

We can access only the value or derivative component as follows:

```
>>> print(f.val)
6.25
>>> print(f.der)
5.0
```

Other elementary functions can be used in the same way. For instance, we may evaluate the derivative of $log_2(x) + arctan(3x + 5)$ at $x = 10.0$ as follows:

```
>>> x = AD.AutoDiff(10.0)
>>> f = x.log(2) + np.arctan(3 * x + 5)
>>> print(f)
====== Function Value(s) ======
4.864160763843499
===== Derivative Value(s) =====
0.14671648614436125
```

Note: For the single variable case, we do not need to input the scalar number in the form of a list (i.e. using brackets); the AutoDiff class is smart enough to handle the scalar form as appropriate.

3.2.2 2. Multiple Variables, Single Function ($\mathbb{R}^m \rightarrow \mathbb{R}^1$)

Consider the case where the user would like to input the function, $f = xy$. Then, the derivative of this would be represented in a Jacobian matrix, $J = [\frac{df}{dx}, \frac{df}{dy}] = [y, x]$.

```
>>> x = AD.AutoDiff(3., [1., 0.])
>>> y = AD.AutoDiff(2., [0., 1.])
>>> f = x*y
>>> print(f)
====== Function Value(s) ======
6.0
===== Derivative Value(s) =====
[2. 3.]
```

3.2.3 3. Single Variable, Multiple Functions ($\mathbb{R}^1 \rightarrow \mathbb{R}^n$)

Consider the case where the user would like to input the two functions, $F = [x^2, 2x]$. Then, the derivative of this would be represented in a Jacobian matrix, $J = [\frac{df_1}{dx}, \frac{df_2}{dx}] = [2x, 2]$.

```
>>> x = AD.AutoDiff(3., 1.)
>>> f1 = x**2
>>> f2 = 2*x
>>> print(AD.AutoDiff([f1, f2]))
====== Function Value(s) ======
[9. 6.]
===== Derivative Value(s) =====
[6. 2.]
```

Note: For evaluating multiple functions, the AutoDiff class expects the functions to be input as a Python list (i.e. using brackets); other data structures (e.g., NumPy array) are not supported.

3.2.4 4. Multiple Variables, Multiple Functions ($\mathbb{R}^m \rightarrow \mathbb{R}^n$)

Consider the case where the user would like to input the two functions, $F = [x + y, xy]$. Then, the derivative of this would be represented in a Jacobian matrix, $J = [[\frac{df_1}{dx}, \frac{df_1}{dy}], [\frac{df_2}{dx}, \frac{df_2}{dy}]] = [[1, 1], [y, x]]$.

```
>>> x = AD.AutoDiff(3., [1., 0.])
>>> y = AD.AutoDiff(2., [0., 1.])
>>> f1 = x+y
>>> f2 = x*y
>>> print(AD.AutoDiff([f1, f2]))
====== Function Value(s) ======
[5. 6.]
===== Derivative Value(s) =====
[[1. 1.]
 [2. 3.]]
```

CHAPTER 4

Software Organization

4.1 Directory Structure

```
AnnoDomini/
  AutoDiff.py
  BFGS.py
  DFP.py
  hamilton_mc.py
  newtons_method.py
  steepest_descent.py
docs/
  source/
    .index.rst.swp
    conf.py
    index.rst (documentation file)
  Makefile
  make.bat
  milestone1.md
tests/
  test_AutoDiff.py
  test_BFGS.py
  test_DFP.py
  test_hmc.py
  test_newton.py
  test_steepest_descent.py
.gitignore
.travis.yml
LICENSE
README.md
```

4.2 Basic Modules

- AutoDiff.py
  - Contains implementation of the master class and its methods for calculating derivatives of elementary functions (list of methods shown in **Core Classes** section below).
- newtons_method.py
  - Contains implementation of the root finding algorithm, Newton's Method
- steepest_descent.py
  - Contains implementation of the optimization method, Steepest Descent
- BFGS.py
  - Contains implementation of the optimization method, BFGS
- DFP.py
  - Contains implementation of the optimization method, BFGS
- hamilton_mc.py
  - Contains the simulation method, Hamiltonian Monte Carlo

4.3 Testing

Our tests are contained in `tests/` directory. - `test_AutoDiff` test_AutoDiff.py is used to test the functions in the AutoDiff Class. It includes tests for both scaler and vector inputs and outputs to ensure our core implementation is correct for general cases. - `test_BFGS` tests BFGS converges on the Rosenbock function - `test_BFGS` tests DFP converges - `test_hmc` tests the Hamiltonian Monte Carlo method - `test_newton` tests Newton's root-finding method - `test_steepestDescent` tests that steepest descent optimization converges

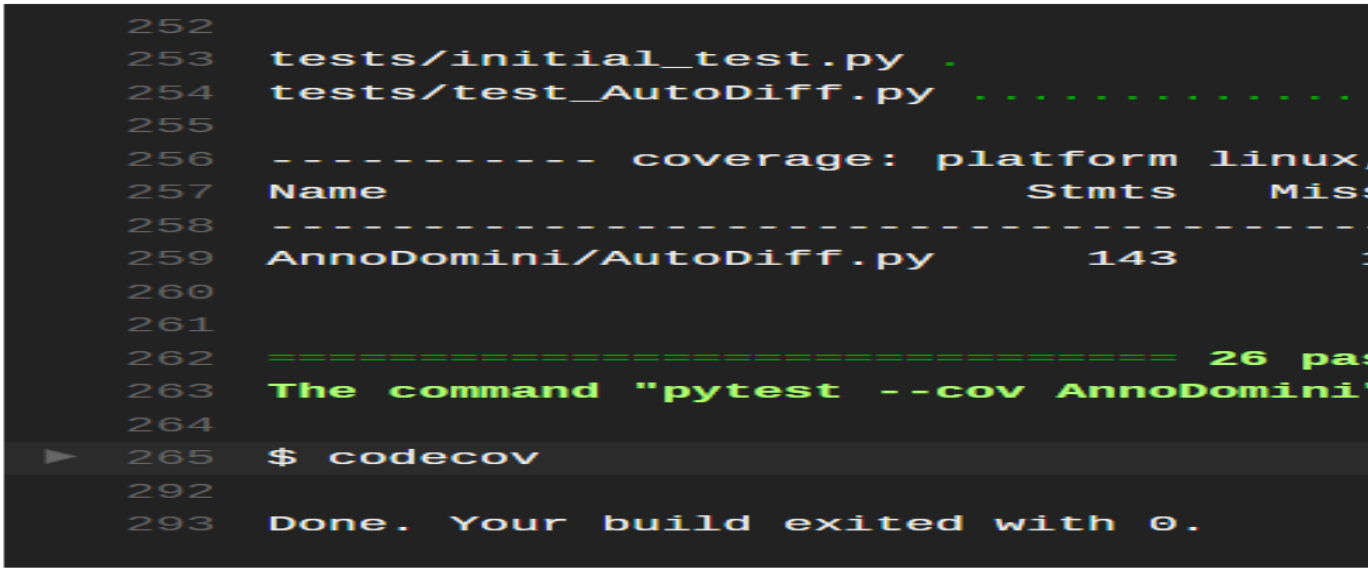
Our test suites are hosted through TravisCI and CodeCov. We run TravisCI first to test the accuracy and CodeCov to test the test coverage. The results can be inferred via the README section.

Our tests are integrated via the TravisCI. that is, call ask TravisCI to CodeCov after completion.

4.4 Packaging

Details on how to install our package are included in the section, How to use Anno Domini.

We use Git to develop the package; we follow instructions :here to package our code and distribute it on PyPi. Instead of using a framework such as PyScaffold, we will adhere to the proposed directory structure. We provide necessary documentation via .rst files (rendered through Sphinx) to provide a clean, readable format on Github.


252
253 tests/initial_test.py .
254 tests/test_AutoDiff.py .....
255
256 ---------- coverage: platform linux,
257 Name Stmts Miss
258 ----------
259 AnnoDomini/AutoDiff.py 143
260
261
262 ========== 26 pas
263 The command "pytest --cov AnnoDomini"
264
265 $ codecov
292
293 Done. Your build exited with 0.

CHAPTER 5

Implementation Details

5.1 AutoDiff

The *AutoDiff* class, which is the core class for the *AnnoDomini* package, takes as input the value to evaluate the function at. It contains two important attributes, `val` and `der`, that respectively store the evaluated function and derivative values at each stage of evaluation.

For instance, consider the case where we want to evaluate the derivative of $x^2 + 2x + 1$ evaluated at $x = 5.0$. This is achieved by first setting `x = AD.AutoDiff(5.0)`, which automatically stores function and derivative values of x evaluated at $x = 5.0$ (i.e. `x.val` = 5.0 and `x.der` = 1.0). When we pass the desired expression (i.e. $x^2 + 2x + 1$) to Python, x is raised to the power of 2, which is carried through *AutoDiff*'s `__pow__` method that returns a new *AutoDiff* object with the updated `val` and `der` values. Specifically, the new returned object in this case has `val` = 25.0 and `der` = 10.0, which are the function and derivative values of x^2 evaluated at $x = 5.0$. A similar process occurs for the other operation steps (i.e. multiplication and addition), and we eventually obtain the *AutoDiff* object with the desired `val` and `der` values (i.e. function and derivative values of $x^2 + 2x + 1$ evaluated at $x = 5.0$).

The *AutoDiff* class has its own methods that define its behavior for common elementary functions such as addition and multiplication. Specifically, the class has the following methods implemented:

```
__add__
__radd__
__sub__
__rsub__
__mul__
__rmul__
__truediv__
__rtruediv__
__pow__
__rpow__
__neg__
sqrt
sin
cos
tan
```

(continues on next page)

(continued from previous page)

```
arcsin
arccos
arctan
sinh
cosh
log
exp
logistic
```

Note: Note that many methods in the *AutoDiff* class, such as *cos* and *exp*, utilize their counterparts in NumPy (e.g., *numpy.cos* and *numpy.exp*). Furthermore, the *AutoDiff* class heavily relies on NumPy arrays and their effective vectorized operations to handle cases with multiple variables and/or multiple functions. Hence, NumPy is an important external dependency that provides a core data structure for the *AnnoDomini* package.

CHAPTER 6

Additional features

Our package comes with a library package containing the following methods:

- Newton's Root Finding
- Steepest Descent
- Broyden–Fletcher–Goldfarb–Shanno (BFGS)
- Davidon–Fletcher–Powell formula (DFP)
- Hamiltonian Monte Carlo(HMC)

All of these methods' implementations use our AnnoDomini package to solve for the gradients within the methods. We include demos of each method in the directory, demos.

A more detailed description of each method is provided below:

6.1 Newton's Root Finding

6.1.1 Background

Newton's root finding algorithm is a useful approach to finding the root(s) of functions that are difficult to solve for analytically, i.e. $log(2x) + arctan(3x + 5)$. Mathematically, the algorithm is given by

$x_{n+1} = x_n - \frac{f(x_n)}{\prime f(x_n)}$

For the multivariate case:

$x_{n+1} = x_n - (J(f)(x_n))^{-1} f(x_n)$

where we begin with an initial guess x_0 (scalar), and iterate until a maximum number of iters has been reached (default: 50 iters), or when the estimated root converges ($x_{n+1} - x_n < T$, for some tolerance T). A useful resource is found here.

6.1.2 API

`Newton Class`:

- `f`: function of interest. If f is a scaler function, we can define f as follows:

```
# option one
def f(x):
   return np.sin(x) + x * np.cos(x)

# option two
f = lambda x: np.sin(x) + x * np.cos(x)
```

  if f is a multivariate function:

```
def f(x,y):
        return x ** 2 + y ** 2 - 3 * x * y - 4 # (x-y)^2 = 9

# or
f = lambda x, y: x ** 2 + y ** 2 - 3 * x * y - 4
```

- `x0`: initial point to start. Support both scaler and vector expressions.

```
x0 = [1,2,3] # supported
x0 = 1 # supported
```

- `maxiter`: max iteration number if convergence not method
- `tol`: stop condition parameter. :math: *||x_n - x_{n-1}|| < tol*
- `alpha`: constant in the newtons_method. The smaller the smaller the step.

`find_root`: return a root of a function.

6.1.3 Demo

```
>>> from AnnoDomini.newtons_method import Newton
>>> import numpy as np
>>> from scipy import linalg as la
>>> f = lambda x: np.sin(x) + x * np.cos(x)
>>> x0 = -3
>>> demo = Newton(f,x0)
>>> root = demo.find_root()
>>> print(root)
-2.028757838110434
>>> quit()
$ deactivate
```

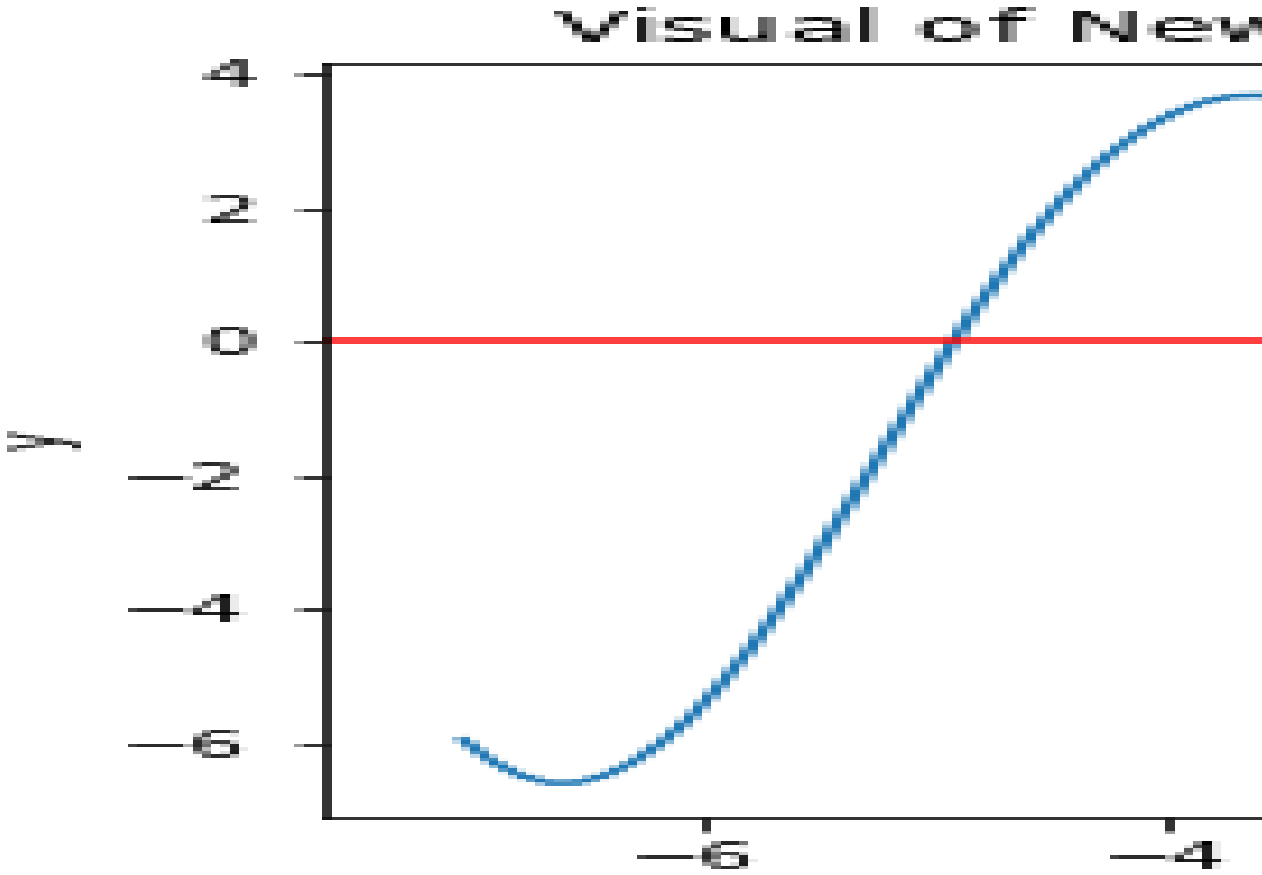
And then we can do the visulization:

```
from matplotlib import pyplot as plt
xs = np.linspace(-7,5,100)
plt.plot(xs, f(xs), label="f")
plt.scatter(root, f(root),label="Root", color = 'black')
plt.scatter(x0, f(x0),label="initial", color = 'red')
plt.xlabel("x")
plt.ylabel("y")
```

(continues on next page)

(continued from previous page)

```
plt.title("Visual of Newton's Method on $sin(x) + x * cos(x)$")
plt.axhline(y = 0, color = 'red')
plt.legend()
plt.show()
```




```
>>> def f(x,y):
        return x ** 2 + y ** 2 - 3 * x * y - 4 # (x-y)^2 = 9
>>> x0 = 1.0
>>> y0 = -2.0
>>> init_vars = [x0, y0]
>>> demo = Newton(f,init_vars)
>>> ans = demo.find_root()
>>> print(ans)
[0.45925007 -1.37598283]
```

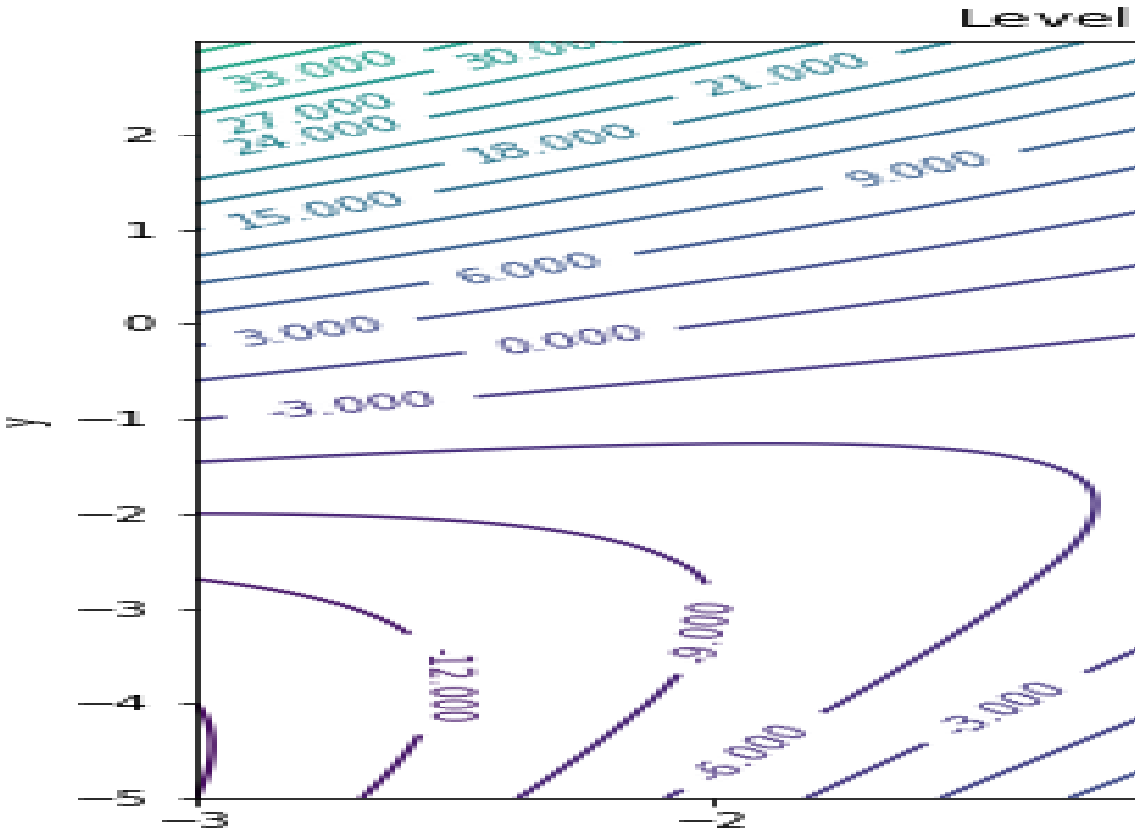
And we can do the visulization by the following ways:

```
delta = 0.025
lam1 = np.arange(-3, 3, delta)
lam2 = np.arange(-5, 3, delta)
Lam1, Lam2 = np.meshgrid(lam1, lam2)
```

(continues on next page)

(continued from previous page)

```
value = Lam1 ** 2 + Lam2 ** 2 - 3 * Lam1 * Lam2 -4
CS = plt.contour(Lam1, Lam2, value,levels = 30)
plt.scatter(x0,y0,color = "red",label = "Initialization")
plt.scatter(ans[0],ans[1],color = "green",label = "root found")
plt.clabel(CS, inline=1, fontsize=10)
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.title('Level Curve of $x^2 + y^2 - 3*x*y - 4$ wrt x and y')
plt.savefig('newton_multivar.png')
```



A full demo of this method is available in the demos subdirectory.

6.2 Steepest Descent

6.2.1 Background

Steepest Descent is an unconstrained optimization algorithm used to find the minima/maxima for a specified function. It achieves this by iteratively following the direction of the negative gradient at every step. We determine the optimal

step size by following the line-search approach: evaluate the function over a range of possible stepsizes and choosing the minimum value.

Mathematically, the algorithm is give by

1. Initialize initial guess, x_0
2. Compute $s_k = -\nabla f(x_k)$
3. Iteratively update the estimated value, $x_{k+1} = x_k + \eta_k s_k$, where η_k is the optimal step size
4. Terminate if maximum number of iterations is reached, or $x_{n+1} - x_n < T$, for some tolerance T

Our method works for both single and multivariable inputs, and single output functions. The user must input a function that accounts for specified number of variables desired. This method can be implemented as follows:

6.2.2 API

`SteepestDescent Class`:

- `f`: function of interest. If f is a scaler function, we can define f as follows:

```
# option one
def f(x):
   return np.sin(x) + x * np.cos(x)

# option two
f = lambda x: np.sin(x) + x * np.cos(x)
```

  if f is a multivariate function:

```
def f(args):
        [x,y] = args
        ans = 100*(y-x**2)**2 + (1-x)**2
        return ans
```

- `x0`: initial point to start. Support both scaler and vector expressions.

```
x0 = [1,2,3] # supported
x0 = 1 # supported
```

- `maxiter`: max iteration number if convergence not method
- `tol`: stop condition parameter. :math: *||x_n - x_{n-1}|| < tol*
- `step`: constant in the steepest descent. The smaller the smaller the step.

`find_root`: return a root of a function.

6.2.3 Demos

```
>>> from AnnoDomini.steepest_descent import SteepestDescent
>>> import numpy as np
>>> from scipy import linalg as la
>>> def f(args):
>>>     [x,y] = args
>>>     ans = 100*(y-x**2)**2 + (1-x)**2
>>>     return ans
```

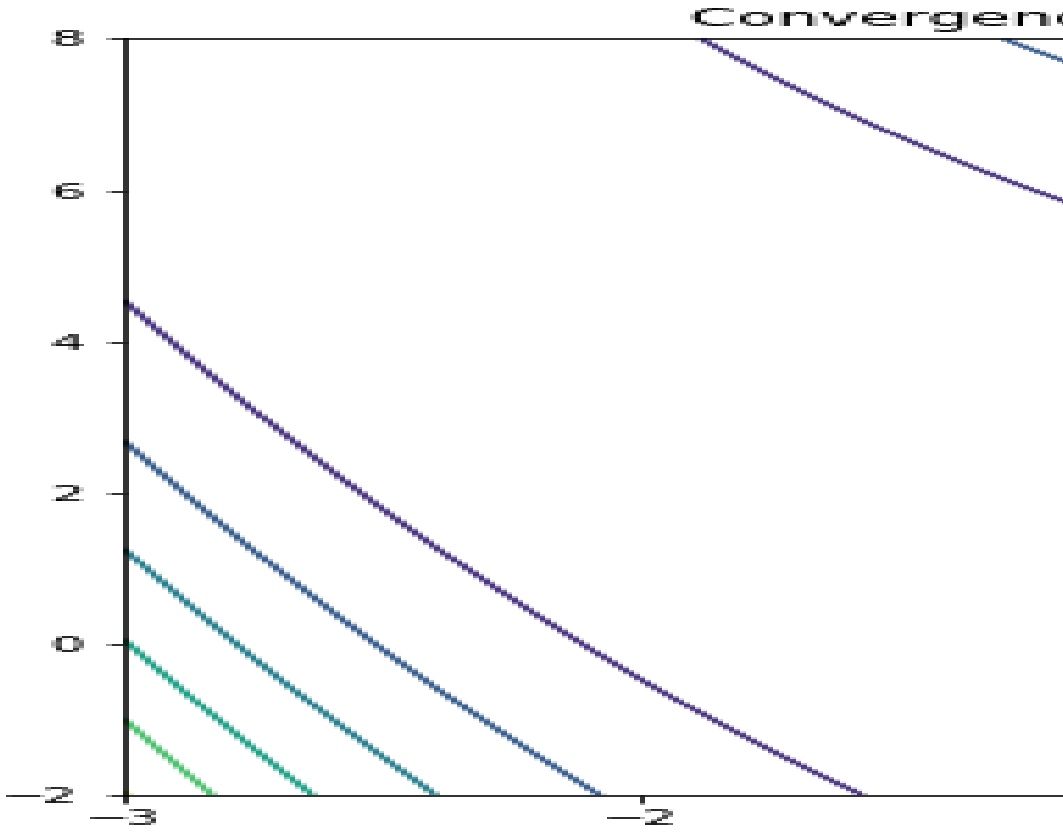
(continues on next page)

(continued from previous page)

```
>>> x0 = [2,1]
>>> sd = SteepestDescent(f, x0)
>>> root = sd.find_root()
>>> print(root)
[1. 1.]
```

And also we can do the visualization:

```
X, Y = np.meshgrid(np.linspace(-3, 3, 100), np.linspace(-2, 8, 100))
Z = f(np.array([X,Y]))
fig = plt.subplots(1,1, figsize = (10,7))
plt.contour(X, Y, Z)
plt.plot(ans[:,0], ans[:,1], "-.", label="Trajectory")
plt.scatter(root[0],root[1], label="Root", c="red")
plt.scatter(-1,1, label="Initial Guess", c ="orange")
plt.title("Convergence of Steepest Descent on Rosenbrock Function")
plt.xlim(-3, 3)
plt.ylim(-2, 8)
plt.legend()
plt.show()
```



A full demo of this method is available in the demos subdirectory.

6.3 Broyden–Fletcher–Goldfarb–Shanno (BFGS)

6.3.1 Background

BFGS, or the Broyden–Fletcher–Goldfarb–Shanno algorithm, is a first-order quasi-Newton optimization method, which approximates the Hessian matrix with the gradient and direction of a function. The algorithm is as follows, in terms of the Approximate Hessian, B_k, the step s_k, and y_k

1. Solve for s_k by solving the linear system $B_k s_k = -y_k$
2. $x_{k+1} = s_k + x_k$
3. $y_k = \nabla x_{k+1} - \nabla x_k$
4. $B_{k+1} = B_k + \frac{y_k y_k^T}{y_k^T s_k} + \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k}$
5. Terminate when $s_k <$ Tolerance

6.3.2 API

`BFGS Class`:

- `f`: function of interest. If f is a scaler function, we can define f as follows:

```
# option one
def f(x):
   return np.sin(x) + x * np.cos(x)

# option two
f = lambda x: np.sin(x) + x * np.cos(x)
```

  if f is a multivariate function:

```
def f(args):
        [x,y] = args
        ans = 100*(y-x**2)**2 + (1-x)**2
        return ans
```

- `x0`: initial point to start. Support both scaler and vector expressions.

```
x0 = [1,2,3] # supported
x0 = 1 # supported
```

- `maxiter`: max iteration number if convergence not method
- `tol`: stop condition parameter. :math: *||x_n - x_{n-1}|| < tol*

`find_root`: return a root of a function.

6.3.3 Demos

This method can be implemented as follows:

```
>>> from AnnoDomini.BFGS import BFGS
>>> import numpy as np
>>> from scipy import linalg as la
```

(continues on next page)

(continued from previous page)

```
>>> def f(args):
>>>     [x,y] = args
>>>     ans = 100*(y-x**2)**2 + (1-x)**2
>>>     return ans
>>> x0 = [2,1]
>>> sd = BFGS(f, x0)
>>> root = sd.find_root()
>>> print(root)
[1. 1.]
```

And the visualization could be done:

```
X, Y = np.meshgrid(np.linspace(-3, 3, 100), np.linspace(-2, 8, 100))
Z = f(np.array([X, Y]))
xmesh, ymesh = np.mgrid[-4:4:80j, -4:4:80j]
fmesh = f(np.array([xmesh, ymesh]))
fig = plt.subplots(1,1, figsize = (10,7))
plt.title('BFGS Path for Rosenbrock's Function, Starting at [2,1]')
plt.xlim(-4, 4)
plt.ylim(-4, 4)
plt.contour(xmesh, ymesh, fmesh, 50)
it_array = np.array(ans)
plt.plot(it_array.T[0], it_array.T[1], "x-", label="Path")
plt.plot(it_array.T[0][0], it_array.T[1][0], 'xr', label='Initial Guess',␣
↪markersize=12)
plt.plot(it_array.T[0][-1], it_array.T[1][-1], 'xg', label='Solution', markersize=12)
plt.legend()
```

A full demo of this method is available in the demos subdirectory.

6.4 Davidon–Fletcher–Powell formula (DFP)

6.4.1 Background

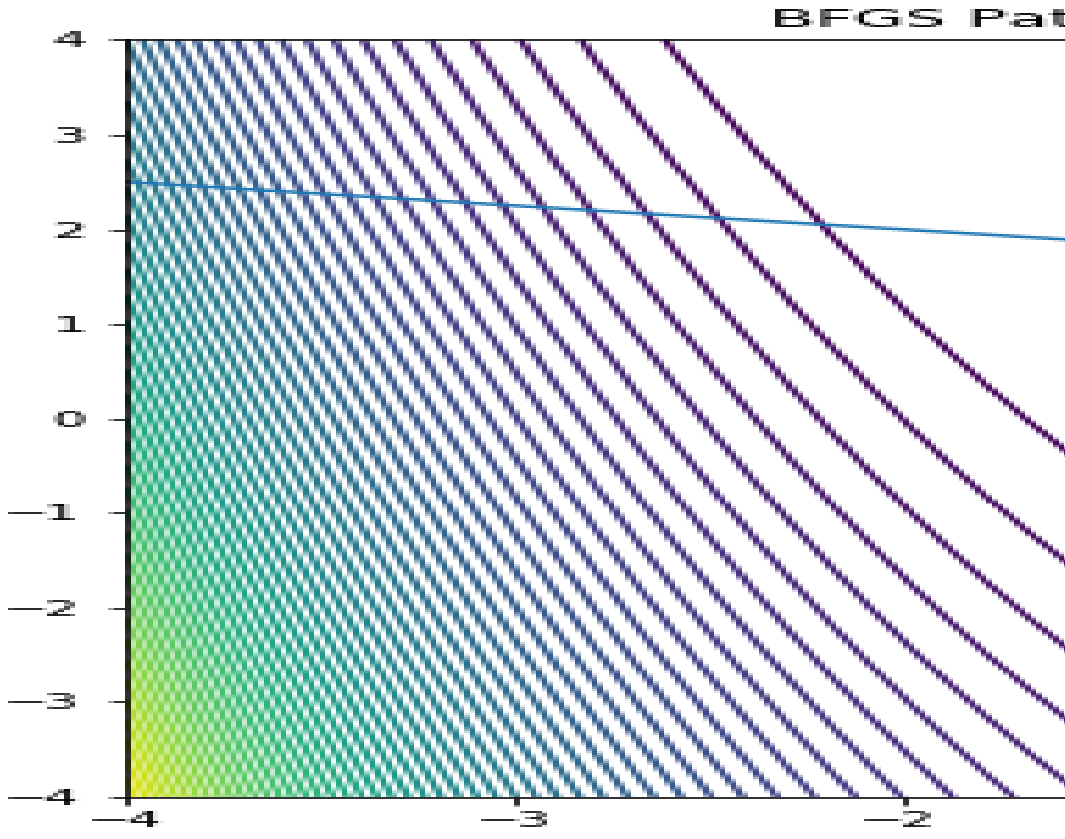
DFP, or the Davidon–Fletcher–Powell formula, is another first-order quasi-Newton optimization method, which also approximates the Hessian matrix with the gradient and direction of a function. The algorithm is as follows, in terms of the Approximate Hessian, B_k, the step s_k, $\gamma_k = \frac{1}{y_k^T s_k}$

1. Solve for s_k by solving the linear system $B_k s_k = -y_k$
2. $x_{k+1} = s_k + x_k$
3. $y_k = \nabla x_{k+1} - \nabla x_k$
4. $B_{k+1} = (I - \gamma_k y_k s_k^T) B_k (I - \gamma_k s_k y_k^T) + \gamma_k y_k y_k^T$
5. Terminate when $s_k <$ Tolerance

6.4.2 API

`DFP Class`:

- `f`: function of interest. If f is a scaler function, we can define f as follows:


BFGS Pat
4
3
2
1
0
−1
−2
−3
−4
−4
−3
−2

```
# option one
def f(x):
   return np.sin(x) + x * np.cos(x)

# option two
f = lambda x: np.sin(x) + x * np.cos(x)
```

if f is a multivariate function:

```
def f(args):
        [x, y] = args
        return np.e**(x+1) + np.e**(-y+1) + (x-y)**2
```

- `x0`: initial point to start. Support both scaler and vector expressions.

```
x0 = [1,2,3] # supported
x0 = 1 # supported
```

- `maxiter`: max iteration number if convergence not method
- `tol`: stop condition parameter. :math: *||x_n - x_{n-1}|| < tol*

`find_root`: return a root of a function.

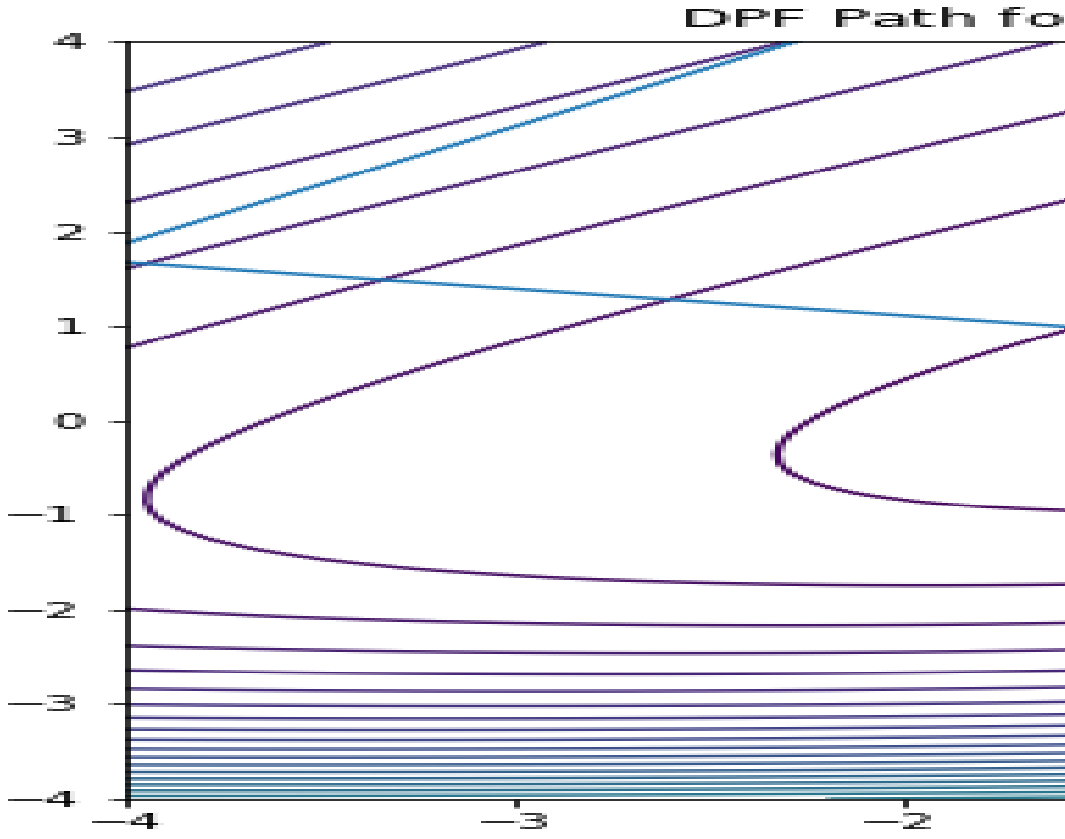
6.4.3 Demos

This method can be implemented as follows:

```
>>> from AnnoDomini.DFP import DFP
>>> import numpy as np
>>> from scipy import linalg as la
>>> def f(args):
>>>     [x,y] = args
>>>     return np.e**(x+1) + np.e**(-y+1) + (x-y)**2
>>> x0 = [2,0]
>>> sd = DFP(f, x0)
>>> root = sd.find_root()
>>> print(root)
[-0.43837842  0.43837842]
```

```
X, Y = np.meshgrid(np.linspace(-3, 3, 100), np.linspace(-2, 8, 100))
Z = f(np.array([X, Y]))
xmesh, ymesh = np.mgrid[-4:4:80j, -4:4:80j]
fmesh = f(np.array([xmesh, ymesh]))
fig = plt.subplots(1,1, figsize = (10,7))
plt.title('DPF Path for $f(x,y) = e^{x+1} + e^{1-y} + {(x-y)}^2$ Starting at [2,0]')
plt.xlim(-4, 4)
plt.ylim(-4, 4)
plt.contour(xmesh, ymesh, fmesh, 50)
it_array = np.array(ans)
plt.plot(it_array.T[0], it_array.T[1], "x-", label="Path")
plt.plot(it_array.T[0][0], it_array.T[1][0], 'xr', label='Initial Guess',␣
↪markersize=12)
plt.plot(it_array.T[0][-1], it_array.T[1][-1], 'xg', label='Solution', markersize=12)
plt.legend()
```

A full demo of this method is available in the demos subdirectory.


DPF Path fo
4
3
2
1
0
−1
−2
−3
−4
−4
−3
−2

Note: DFP is empirically significantly less performant than BFPS. For instance, it may take up to 1 million iterations to converge on the Rosenbrock function.

6.5 Hamiltonian Monte Carlo

6.5.1 Background

In computational physics and statistics, the Hamiltonian Monte Carlo algorithm is a Markov chain Monte Carlo method for obtaining a sequence of random samples which converge to being distributed according to a target probability distribution for which direct sampling is difficult. This sequence can be used to estimate integrals with respect to the target distribution (expected values).

Hamiltonian Monte Carlo corresponds to an instance of the Metropolis–Hastings algorithm, with a Hamiltonian dynamics evolution simulated using a time-reversible and volume-preserving numerical integrator (typically the leapfrog integrator) to propose a move to a new point in the state space. Compared to using a Gaussian random walk proposal distribution in the Metropolis–Hastings algorithm, Hamiltonian Monte Carlo reduces the correlation between successive sampled states by proposing moves to distant states which maintain a high probability of acceptance due to the approximate energy conserving properties of the simulated.

An easy way to implement Hamiltonian Monte Carlo is to use leap frog Integrator. The implementation is as follows:

Let $\pi(q)$ be our target distribution with $q \in \mathbb{R}^D$. We turn π into an energy function $U(q)$ by $U(q) = -\log(\pi(q))$. We choose a kinetic energy function $K(p)$.

0. start with a random $q^{(0)} \in \mathbb{R}^D$

1. repeat:

- A. (kick-off) sample a random momentum from the Gibbs distribution of $K(p)$, i.e.

$p^{(current)} \sim \frac{1}{Z}\exp(-K(p))$

- B. (simulate movement) simulate Hamiltonian motion for L steps each with time interval :math‘epsilon‘, using the leap-frog integrator.
  - a. Repeat for T - 1 times, for $p^{(\text{step } 0)} = p^{(current)}$, $q^{(\text{step } 0)} = q^{(current)}$
    - i.(half-step update for momentum) $p^{(\text{step } t+1/2)} \leftarrow p^{(\text{step } t)} - \epsilon/2\frac{\partial U}{\partial q}(q^{(\text{step } t)})$
    - ii.(full-step update for position) $q^{(\text{step } t+1)} \leftarrow q^{(\text{step } t)} + \epsilon\frac{\partial K}{\partial p}(p^{(\text{step } t)})$
    - iii. (half-step update for momentum) $p^{(\text{step } t+1)} \leftarrow p^{(\text{step } t+1/2)} - \epsilon/2\frac{\partial U}{\partial q}(q^{(\text{step } t+1)})$
  - b.(reverse momentum) $p^{(\text{step}T)} \leftarrow -p^{(\text{step}T)}$
- C. (correction for simulation error) implement Metropolis-Hasting accept mechanism:
  - a. compute $\alpha = \min\left(1, \exp\left\{H(q^{(current)}, p^{(current)}) - H(q^{(\text{step } T)}, p^{(\text{step } T)})\right\}\right)$
  - b. sample $U \sim U(0,1)$, if :math:‘U leq alpha‘then accept, else keep old sample

In our implementation, we use the simplist Euclidean-Gaussian Kinetic Energy function, i.e.

$K(p) = \frac{1}{2}p^\top M^{-1}p + \frac{1}{2}\log|M| + \frac{D}{2}\log(2\pi)$

6.5.2 API

HMC:

- q_init: initial point to start with. For the scaler case, we support both scaler and list input like [0].
- target_pdf and Ù. Target density function to sample from and target negative log (density function). If target_pdf is provided, U would be calculated via:

```
U = lambda q: -np.log(target_pdf(q))
```

However, since there would be cases where negative log density function is easier to obtain, users can specify U directly. In this case, target_pdf would be ignored.

- D: dimension of the input. Could be infered from the q_init and also could be specified directly.
- chain_len: length of hamiltonian monte carlo chain. default 1000
- ‘‘ T‘‘: length of leapfrog in HMC, default 5
- burn_in, thinning: burn in and thinning to the chain. default 0 and 1
- epsilon: step length in the HMC for the leap-frog.

describe: a straight forward way to estimate the mean, variance and quantiles for a certain distribution based on HMC.

The describe shares same parameters with HMC. It returns a dict where the mean, variance and quantiles are stored in the “mean”, “var” and “quantiles”.

6.5.3 Demos

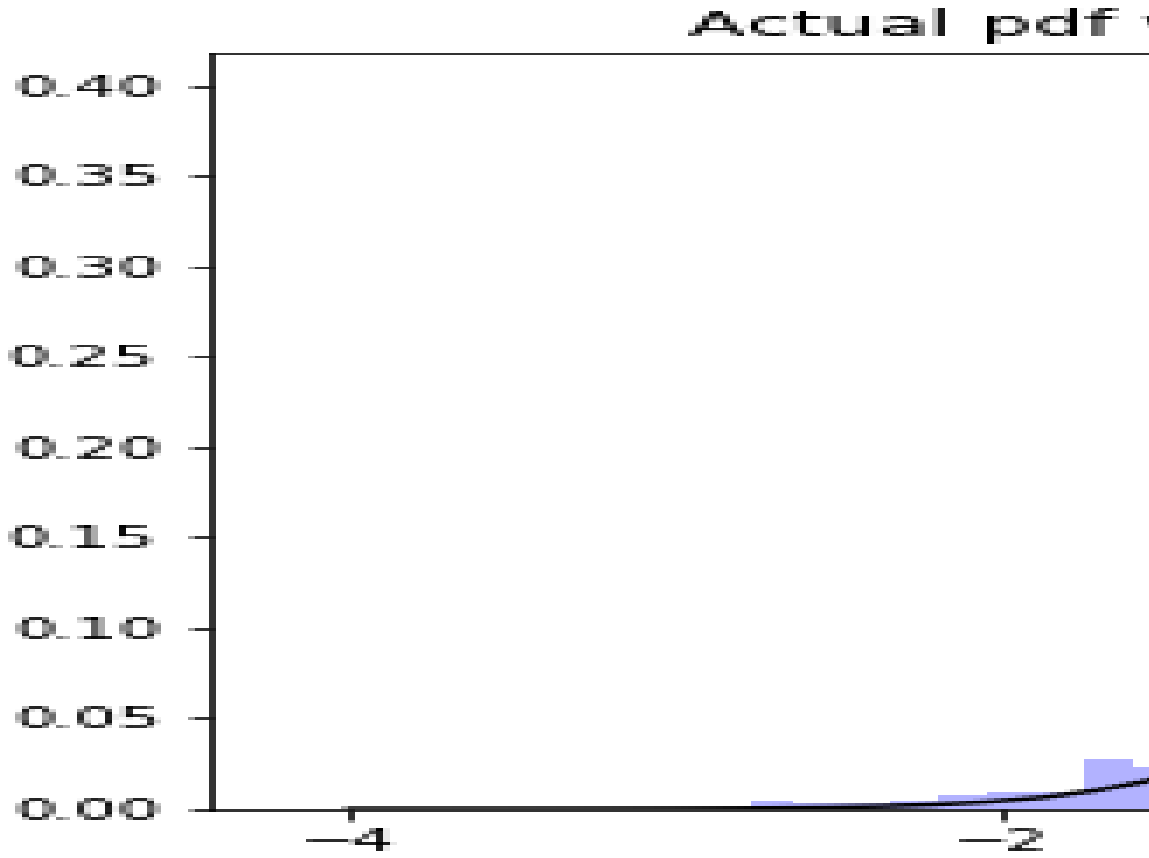
Here is a demo for singular Gaussian family.

```
>>> import numpy as np
>>> from AnnoDomini.hamilton_mc import HMC, describe
>>> def norm_function(mu = 0, var = 1):
        def norm(x):
                denom = (2*np.pi*var)**.5
                num = np.exp(-(x-mu)**2/(2*var))
                return num/denom
        return norm

>>> start_point = -10.0 # start from far apart
>>> func = norm_function(1,1)

>>> chain,accepts_ratio = HMC(target_pdf = func, burn_in=200, thinning=2,chain_
↪len=10000, q_init=[start_point],epsilon = 0.05)
100%|| 10000/10000 [00:05<00:00, 1706.38it/s]
>>> print("Accepts ratio = {}".format(accepts_ratio))
Accepts ratio = 0.9919
>>> print(chain.shape)
(4900, 1)
```

Note: accepts_ratio is a good indicator for the quanlity of the chain. it should be greater than 90%. If not, try to adjust epsilon and T in the leap-frog stage.

We can visually check the correctness of HMC by:

```
q = chain[:,0]
fig,ax = plt.subplots(1,1,figsize = (8,5))
x = np.linspace(-4,4)
ax.plot(x,func(x),color = "black",label = "actual pdf")
ax.hist(q,bins = 50, density = True, color = "blue",alpha = 0.3, label = "histogram␣
↪of samples")
ax.set_title("Actual pdf vs sampling by hamiltonian monte carlo")
ax.legend()
```



And describe function could be used to estimate mean and variance by:

```
>>> import numpy as np
>>> from AnnoDomini.hamilton_mc import HMC, describe
>>> def norm_function(mu = 0, var = 1):
        def norm(x):
                denom = (2*np.pi*var)**.5
                num = np.exp(-(x-mu)**2/(2*var))
                return num/denom
        return norm

>>> start_point = -10.0 # start from far apart
>>> func = norm_function(1,0.1)
```

(continues on next page)

(continued from previous page)

```
>>> chain,accepts_ratio = describe(target_pdf = func, burn_in=200, thinning=2,chain_
↪len=10000, q_init=[start_point],epsilon = 0.05)
100%|| 10000/10000 [00:05<00:00, 1706.38it/s]

>>> print("Accepts ratio = \{\}".format(accepts_ratio))
Accepts ratio = 0.9267
```

```
>>> print("Mean = \{\}".format(d['mean'])) # 1
Mean = 0.9976236280753902
>>> print("Var = \{\}".format(d['var'])) # 0.1
Var = 0.1267435837161925
>>> print("quantiles(25%, 75%) = \{\}".format(d['quantiles']))
quantiles(25%, 75%) = [0.75131308 1.23747691]
```

A full demo of this method is available in the demos subdirectory. (Includes the weibull distribution)

CHAPTER 7

Future Features

We would like to extend our package in a few different ways:

7.1 Module Extension

7.1.1 Reverse Mode

This improvement will allow our users to play with custom Neural Network models using backpropagation. Since we have performed forward mode successfully, it should not be a big deal.

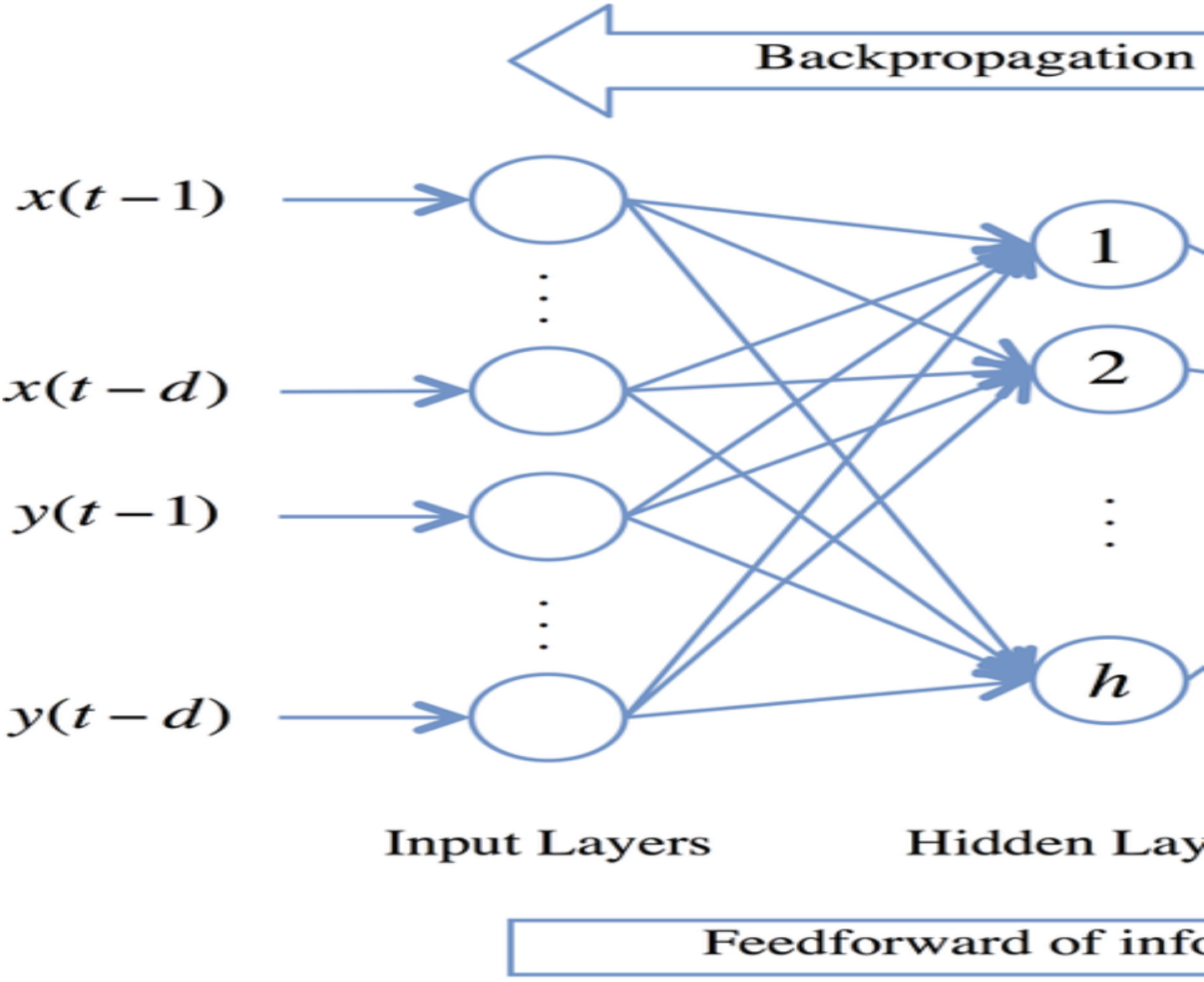
7.1.2 Second order derivative approximates

It would be beneficial to extend our package to account for calculating second derivatives, because many algorithms and methods rely on computing the second derivitve (single variable case) or the Hessian matrix (multivariable case). By handing this case, our package will be more user-friendly and can be applied to more algorithms that require this calculation.

7.2 Possible Applications

7.2.1 1. Hospital Scheduling

A good potential application would be in the health science. In many countries, at the hospital patients have to queue and wait for a long time for many procedures, from register to meeting doctor to taking medicine. This could be modeled as a schedule optimization problem. With the automatic differentiation many algorithms could be applied to solving this scheduling problem, which could greatly improve the efficiency in the hospital.


Backpropagation
x(t − 1)
x(t − d)
y(t − 1)
y(t − d)
1
2
h
Input Layers
Hidden Lay
Feedforward of info

7.2.2 2. Supply Chain Management

As for the optimization, a good example would be supply chain management. Supply chain management is the management of the flow of goods and services, involves the movement and storage of raw materials, of work-in-process inventory, and of finished goods from point of origin to point of consumption. Often it could be modeled as a complex optimization problem. With our package we can try to solve those hard problems and thus make contribute to the lower cost during this management.

7.2.3 3. Obesity Prevention:

A relatively recent Harvard health article found that while Asian body types are categorized as "skinny", there is a higher risk for type 2 diabetes among this race. This is because type 2 diabetes is correlated with body fat percentage. Because Asians typically have a smaller body size, they are mistaken for being "healthy" even with poor eating habits. Consequently, those with poor eating habits could have a larger percentage of body fat, but it is disguised as "skinny fat". It would be interesting to model the body statistics (i.e. BMI, body fat percentage, height, weight, etc.) of this particular subgroup and determine the optimal diet plan to help reduce their risk of type 2 diabetes. This could be done by using an optimization algorithm to find the amount of nutrients that could help their body quickly recover. This problem could also involve modeling how their body reacts to certain nutrients, and finding the optimal amount of specified nutrients that could help reduce their body fat percentage. All of which would need to derive gradients within the algorithms used.

7.2.4 4. Bayesian Statistics

In Bayesian statistics, with Hamiltonian Monte Carlo, we are able to sample from difficult posterior distributions, thus make it possible for us to do the inference from the model.


Supply chain managem
Manufacturing
Supplier
Raw materials