CS207 final project

Release 0.0

Nov 19, 2019

CONTENTS

This is the base documentation for the `autodiff` package.

CHAPTER

ONE

WHAT IS AUTOMATIC DIFFERENTIATION?

1.1 Introduction

This software aims to numerically evaluate the derivative of any function with high precision utilizing automatic differentiation (AD). Specifically, the Jacobian matrix of dimension $n \times m$ of any function $func : \mathbb{R}^m \mapsto \mathbb{R}^n$ will be computed. Automatic differentiation is different from numerical differentiation and symbolic differentiation, which are introduced in the following:

1. Numerical differentiation, i.e., differentiation with the method of finite difference, can become unstable depending on step size and the particular function we're trying to differentiate.
2. Symbolic differentiation: A difficult example:

$$f(x, y, z) = \frac{\cos(\exp(\frac{-5x^2}{y}))}{\frac{\sin(x)}{x^3} - \mathrm{erf}(z)}$$

Symbolic differentiation (such as `sympy`) performs well for simple math forms, but becomes complex with arbitrary functions, and requires that every function have an analytical representation. This is very computationally expensive and almost never implemented in application.

1.1.1 Why is AD important?

First of all, AD dissects each function and its derivatives to a sequence of elementary functions. The chain rule is applied repeatedly on these elementary terms. Accuracy is maintained because differentiating elementary operations is simple and minimal error is propagated over the process. Efficiency is also maintained because increasing order does not increase computation difficulty. Also, AD computes partial derivatives, or the Jacobian matrices, which are one of the most common steps in science and engineering. One important application is optimization, which is extremely useful and implemented in every field such as machine learning. One advantage of AD is high accuracy, which is an essential requirement to computation because small errors could accumulate in higher dimensions and over iterations and result in a catastrophe. Another advantage of AD is efficiently. Efficiency is very important because the time and energy are usually limited for a particular project.

1.2 Background

1.2.1 The Chain Rule

The chain rule is used to differentiate composite functions. It is applied when the derivatives of nested functions are computed. A simple case is the following function: $g(f(x))$. The derivative of g with respect to x is $\frac{\delta g}{\delta x} = \frac{\delta g}{\delta f} \cdot \frac{\delta f}{\delta x}$. This derivative can also be written in prime notation as $g'(f(x)) \cdot f'(x)$.

We can also consider a slightly more complex case, in which the function takes more than one variable: $g(f(x), h(x))$. The derivative of g with respect to x will then become: $\frac{\delta g}{\delta x} = \frac{\delta g}{\delta f} \cdot \frac{\delta f}{\delta x} + \frac{\delta g}{\delta h} \cdot \frac{\delta h}{\delta x}$. This derivative can also be written in prime notation as $g'(f(x)) \cdot f'(x) + g'(h(x)) \cdot h'(x)$.

1.2.2 Elementary Functions and Computation Accuracy

An elementary function is a single-variable function such as constant, sum, product, sin, cos, exp and etc. The derivative of each elementary function is known or very simple. High accuracy is maintained when AD is applied because differentiating elementary operations is simple and minimal error is propagated over the process when chain rule is applied.

1.2.3 The Forward Mode

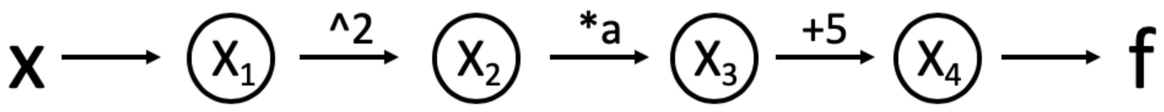
In forward mode, the Jacobian vector matrix of the composite function is calculated through sequential evaluting each sub-function's value and the derivative value, and chain rule takes such values to the next function, whose value and derivative value are evaluated. The process goes on until all sub-functions in the composite function are evaluated. The value of the first function, usually a vector of constants, is set by a seed vector, which in turn decides the value of the first set of derivatives. This process is highly efficient even at high order because the complexity of AD does not scale with the dimension of the functions.

1.2.4 The Graph Structure

We can visualize each evaluation step in an AD process with a computation graph. For example, we have a simple function

$$f(x) = ax^2 + 5$$

The computation graph is the following:



1.2.5 The Evaluation Table

We can also demonstrate each evaluation using an evaluation table. Using the same example at $x = 2$ (seed vector):

Step	Elementary Operations	Numerical Value	$\frac{\mathrm{d}f}{\mathrm{d}x}$	$\frac{\mathrm{d}f}{\mathrm{d}x}$ Value
x_1	x	2	$\dot{x}_1$	1
x_2	x_1^2	4	$2x_1\dot{x}_1$	4
x_3	ax_2	$4a$	$a\dot{x}_2$	$4a$
x_4	$x_3 + 5$	$4a + 5$	$\dot{x}_3$	$4a$

CHAPTER

TWO

HOW TO INSTALL

To use our software, do the following:

1. Clone our project repository from Github.

```
git clone https://github.com/rocketscience0/cs207-FinalProject.git
```

1. Navigate to the local repository.

```
cd /path/to/cs207-FinalProject
```

1. Create a new `conda python` virtual environment. Note: This command only needs to be done once.

```
conda env create -n autodiff -f autodiff-env.yml
```

1. Activate the virtual environment.

```
conda activate autodiff
```

CHAPTER

THREE

HOW TO USE AUTODIFF

The core data structure is a Number, which stores both a value and a dictionary of derivatives. After instantiation, a number's derivative with respect to itself will be automatically set to 1 unless otherwise specified:

```
>>> import autodiff
>>> x = autodiff.Number(3)
>>> x.value
3
>>> x.deriv[x]
1
>>> a = autodiff.Number(3,2)
>>> a.value
3
>>> a.deriv[a]
2
```

Also, import autodiff will import numpy automatically. Therefore, user will also be able to make use of constants such as np.pi and vectors using numpy arrays.

Using elementary operations will update derivatives according to the chain rule:

```
>>> import autodiff
>>> x = autodiff.Number(3)
>>> y = x**2
>>> y.value
9
>>> y.deriv[x]
6
>>> y.deriv[y]
1
```

Right now, 'Number' overloads basic operations, including addition, subtraction, multiplication, division and power.

To call other basic operations such as sin, cos, tan, exponential, log, and negation, simply call the following on Number:

```
>>> x = autodiff.Number(np.pi/6)
>>> y = x.sin()
>>> y.value
1/2
```

Note that the deriv attribute is a dict storing partial derivatives with respect to each Number object involved in preceding elementary operations.

When any elementary operation takes in two `Number()` objects, that elementary operation will return a `Number()` with a partial derivative with respect to every key of both `Number()` objects:

```
>>> x = autodiff.Number(2)
>>> y = autodiff.Number(3)
>>> def f(x, y, a=3):
>>>     return a * x * y
>>> q = f(x, y, a=3)
>>> q.deriv[x]
9
>>> q.deriv[y]
6
>>> q.deriv
{Number(value=2): 9, Number(value=3): 6}
```

Similarly, `autodiff` can work with vector functions of scalars. In these cases, each value in `deriv` is an array with the same shape as the output vector:

```
x = autodiff.Number(np.pi / 2)
y = autodiff.Number(3 * np.pi / 2)

def f(x, y):
    return autodiff.array((
        y * autodiff.sin(x),
        x * autodiff.sin(y)
    ))
q = f(x,y)
```

```
>>> q.deriv[x]
autodiff.array([0, 1])
>>> q.deriv[y]
autodiff.array([1, 0])
```

The `autodiff` package also works for scalar functions of vectors and vector functions of vectors, which behave the same.

Of course, most users will like to work with Jacobians and gradients rather than a `dict` of partial derivatives. Doing so is simple through the `jacobian` method. When an expression returns a scalar, `jacobian` will return that expression's gradient. When an expression returns a vector, `jacobian` will return that expression's Jacobian as a two-dimensional array.

```
>>> x = autodiff.array((1, 2))
>>> y = autodiff.array((3, 4))
>>> q.jacobian((*x, *y))
autodiff.array([3, 4, 1, 2])
>>> q.jacobian((*x, *y)).shape
(4,)
```

Or with a vector function:

```
>>> x = autodiff.Number(np.pi / 2)
>>> y = autodiff.Number(3 * np.pi / 2)

>>> def f(x, y):
        return autodiff.array((
            y * autodiff.sin(x),
            x * autodiff.sin(y)
```

(continues on next page)

(continued from previous page)

```
        ))

>>> q = f(x,y)
>>> q.jacobian((x, y))
autodiff.array([[0, 1],
                [1, 0]])
>>> q.jacobian((x, y)).shape
(2, 2)
```

Note that `autodiff.Number.jacobian()` does require the user to specify an order of input `Number` objects to ensure consistency within the user's own code. Otherwise, `autodiff` would have to infer which element belongs to which function input. As the user strings together multiple elementary operations, it is likely that `autodiff`'s understanding would differ from the user's. An example of the suggested usage is:

```
x = autodiff.Number(2)
y = autodiff.Number(3)
z = autodiff.Number(4)

order = (x, y, z)

# The gradient of f1 and f2 do not have an inherent order.
# If we displayed Numbers in the order they were used, the implied order would be
# (grad_x, grad_z, grad_y)---likely not what the user desires.
f1 = x**z
f2 = f1 * x * y
```

```
>>> f2.deriv[x]
240
>>> f2.deriv[y]
32
>>> f2.deriv[z]
66.542
>>> f2.jacobian((order))
autodiff.array([240, 32, 66.542])
```

CHAPTER

FOUR

SOFTWARE ORGANIZATION

4.1 Directory structure

```
.
├── README.md
├── .travis.yml
├── autodiff-env.yml
├── .gitignore
└── docs
    ├── pandoc-minted.py
    └── source
            ├── sphinx-requirements.txt
            ├── index.rst
            └── api-doc
                    ├── autodiff.rst
                    └── modules.rst
└── autodiff
    ├── __init__.py
    ├── operations.py
    └── structures.py
└── tests
    ├── newtons_method.py
    └── test_operations.py
```

4.1.1 Modules

There are two modules. The `autodiff` module implements the forward mode of automatic differentiation. It contains `structures.py`, the definition of the `Number` class, and `operations.py`, the implementations of the various methods of `Number` (elementary operations, the derivatives of elementary operations). The `tests` module runs tests for `autodiff`. See below for details about testing.

4.1.2 Testing

All tests live in `tests/test_autodiff.py`. We will use both `TravisCI` and `CodeCov` to distribute reports.

CHAPTER

FIVE

ROOT-FINDING ALGORITHM

Here, we demonstrate a use case of a root-finding algorithm that requires calculation of the Jacobian.

The test function is $y = 5x^2 + 10x - 8$.

```
>>> import autodiff.operations as operations
>>> from autodiff.structures import Number
```

```
def func(x):
    return 5 * x ** 2 + 10 * x - 8
```

Newton's method is implemented to find the root of the test function. The user should be able to find the root and access the Jacobian of each iteration.

How does this implementation work?

1. The function and its derivative are evaluated at x_0, then x_1 is calculated as $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$. The Jacobian is stored in the `jacobians` list.
2. The evaluation of the function at x_1 is compared with the threshhold, in this case 10^{-7}. The absolute value of the function's value at x_1 is larger than the threshold, so x_0's value is updated, i.e., $x_0 = x_1$.
3. The function and its derivative is again evaluated at the new x_0. Derivative is stored in the `jacobians` list. This process is repeated until the threshold is met.
4. The `Newton()` method returns the root and the `jacobians` list. User may access each step's derivative from this list.

```
def newtons_method(func, initial_guess):

    # Store a list of jacobians from each iteration
    jacobians = []
    x0 = initial_guess
        fxn = func(initial_guess)
    fpxn = fxn.jacobian(initial_guess)
    x1 = x0 - fxn/fpxn
        jacobians.append(fpxn)

    while abs(fxn.val) > 1e-7:
        x0 = x1
                fxn = func(x0)
                fpxn = fxn.jacobian(x0)
                jacobians.append(fpxn)
        x1 = x0- fxn / fpxn

    return x1, jacobians
```

We can instantiate a `Number(5)` as the initial guess (x_0) of the root.

```
>>> x0 = Number(5)
```

The `Newton()` method takes the test function and the initial guess.

```
>>> xstar, jacobians = newtons_method(func, x0)
>>> print(xstar, jacobians[-1])
Number(val=0.6124515496597099) 16.124515496597116
```

CHAPTER

SIX

IMPLEMENTATION

Note: A full API documentation is avalable here: *autodiff*

6.1 External dependencies

The autodiff package proper requires only numpy. Running tests requires pytest and codecov, while generating this documentation requires sphinx (version 1.7.9).

6.2 Core data structures and classes

Currently, the autodiff package has one core data structure, the Number. A Number is a scalar that stores a value and a derivative. Future versions will include the array, which subclasses the numpy.ndarray to support functions with vector inputs.

6.2.1 Important attributes of the Number class

The Number class has only two attributes, a value (val) and a dict of partial derivatives (deriv). The user can can define a new type of number easily:

```python
class NewInt(Number):
    def __init__(self, a, b):
        super(self).__init__(a, b)
        self.val = int(a)
        self.deriv = b
```

6.3 Methods and name attributes

The Number class overloads the following common elementary operations:

- +
- -
- *
- /

- `**`
- `@`

We have also included the following elementary operations, all of which use their *numpy* counterparts and live in the *autodiff.operations* module.

- `autodiff.operations.sin()`
- `autodiff.operations.cos()`
- `autodiff.operations.tan()`
- `autodiff.operations.asin()`
- `autodiff.operations.acos()`
- `autodiff.operations.atan()`
- `autodiff.operations.log()`
- `autodiff.operations.exp()`
- `autodiff.operations.sqrt()`

Defining custom elementary functions is straightforward, using the `elementary` decorator (this is the same method we use internally). The decorator takes one input, a function with the same arguments as the elementary operation, but calculates the derivative of the operation rather than the value. We call this derivative function internally.

```
def my_pow_deriv(a, b):
    """ Returns the derivative of my_pow at a and b
    """
    return b * a ** (b - 1)

@elementary(my_pow_deriv)
def my_pow(a, b):
    return pow(a, b)
```

```
def sin_deriv(a):
    """ Returns the derivative of the sin() elemental operation"""
    try:
        return a.deriv * np.cos(a.value)
    except AttributeError
        return np.cos(a)

@elementary(sin_deriv)
def sin(a):
    try ...
    return np.sin(a)
```

The `Number()` class overloads __add__ and __radd__, along with other elementary operations as follows. The `autodiff.array` class overloads vector operations similarly.

```
# From autodiff.operations
def add_deriv(x,y):
    """Derivative of additions, one of x and y has to be a Number object

    Args:
        x: a Number
        y: a Number object or an int/float to be added

```

(continues on next page)

(continued from previous page)

```
    Returns:
        The derivative of the sum of x and y
    """
    try:
        d={}
        for key in x.deriv.keys():
            if key in y.deriv.keys():
                d[key] = x.deriv[key] + y.deriv[key]
            else:
                d[key] = x.deriv[key]
        for key in y.deriv.keys():
            if not key in x.deriv.keys():
                d[key] = y.deriv[key]
    except AttributeError:
        d = x.deriv
    return d

@elementary(add_deriv)
def add(x,y):
    """add two numbers together, one of x and y has to be a Number object

    Args:
        x: a Number object or an int/float
        y: a Number object or an int/float to be added

    Returns:
        value of the sum
    """
    try:
        s = x.val + y.val
    except:
        s = x.val + y
    return s


# In autodiff.structures
class Number():
    ...

    def __add__(self, other):
        '''
        Overloads the add method to add a number object to another Number object or an integer/float.

        Args:
            other: a Number object or an integer or float to be added.

        Returns:
            another Number object, which is the sum.
        '''
        return operations.add(self, other)

    def __radd__(self, other):
        '''
        Overloads the right add method to add a number object to another Number object or an integer/
↪float.
```

(continues on next page)

(continued from previous page)

```
        Args:
            other: a Number object or an integer or float to be added.

        Returns:
            another Number object, which is the sum.
        '''
        return operations.add(self, other)
```

6.4 To include in future versions

At this time, `autodiff` only supports scalar functions with scalar outputs. Soon, we will also support vector functions with vector outputs. An `autodiff.array` will subclass `numpy.array`, but will hold `Number` objects. Therefore, matrix operations will be available as they are in `numpy`, including:

- Matrix multiplication (`@`, `dot`)
- Element-wise operations (`+`, `-`, `*`, `/`, `**`)

There will be a few differences when defining function with vector outputs. Rather than each value of the `deriv` dict being a scalar, a vector `deriv` value will instead be an `array`—interpreted as a column of the Jacobian. Once again, it will be necessary for the user to specify in which order he or she will like their Jacobian. Internally, we will treat the user's specified order as a set of seed vectors to calculate each column of the Jacobian.

CHAPTER

SEVEN

EXAMPLES

We've included a few examples of how to use `autodiff`. For interactive jupyter notebooks, please see the `docs` folder `autodiff`'s root directory.

7.1 A root finding example

The test function is $y = 5x^2 + 10x - 8$. Newton's method is implemented to find the root of the test function. The user should be able to find the root and access the Jacobian of each iteration.

First, we instantiate a `Number(5)` as the initial guess (x_0) of the root. The `Newton()` method takes the test function and the initial guess.

Second, the function and its derivative are evaluated at x_0, then x_1 is calculated as $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$. The jacobian is stored in the `jacobians` list

Third, the evaluation of the function at x_1 is compared with the threshhold, in this case 10^{-7}. The absolute value of the function's value at x_1 is larger than the threshold, so x_0's value is updated, i.e., $x_0 = x_1$.

Fourth, the function and its derivative is again evaluated at the new x_0. Derivative is stored in the `jacobians` list. This process is repeated until the threshold is met.

Fifth, the `Newton()` method returns the root and the `jacobians` list. User may access each step's derivative from this list.

```python
import sys
sys.path.append('..')

import autodiff.operations as operations
from autodiff.structures import Number
import numpy as np
from copy import deepcopy

def func(x):
    return 5 * x ** 2 + 10 * x - 8

def newtons_method(func, initial_guess):

    #stores a list of jacobians from each iteration
    jacobians = []

    x0 = initial_guess

    fxn = func(initial_guess)
```

(continues on next page)

(continued from previous page)

```
    fpxn = fxn.jacobian(initial_guess)

    x1 = x0 - fxn/fpxn

    jacobians.append(fpxn)


    while abs(fxn.val) > 1e-7:
        x0 = x1

        fxn = func(x0)

        fpxn = fxn.jacobian(x0)

        jacobians.append(fpxn)

        x1 = x0- fxn / fpxn

    return x1, jacobians


x0 = Number(5)
xstar, jacobians = newtons_method(func, x0)

print(xstar, jacobians[-1])
```

```
Number(val=0.6124515496597099) 16.124515496597116
```

7.2 An optimization example

The test function is $y = 5x^2 + 10x - 8$. BFGS's method is implemented to find the minimum of the test function. The user should be able to find the x and y of the minimum as well as access the Jacobian of each optimization step.

First, we instantiate a `Number(5)` as the initial guess (x_0) of the root to the minimum. The `bfgs()` method takes the test function and the initial guess.

Second, the function and its derivative are evaluated at x_0. BFGS requires a speculated Hessian, and the initial guess is usually an identity matrix, or in the scalar case, `1`. The initial guess of hessian is stored in b_0 Then an intermediate s_0 is determined through solving $b_0 s_0 = -\nabla func(x_0)$.

Third, x_1's value is set to be $x_0 + s_0$

Fourth, another intermediate y_0's value is set to be $\nabla(x_1) - \nabla(x_0)$

Fifth, b_1 is updated and its value is equal to $b_1 = b_0 + \Delta b_0$, where Δb_0 is equivalent to $\frac{y_0}{s_0} - b0$

Sixth, The values of b_0 and x_0 are updated with b_1 and x_1, respectively. Such process repeats until the Jacobian turns `0`

Note, in our example,

```
import sys
sys.path.append('..')
```

(continues on next page)

(continued from previous page)

```python
import autodiff.operations as operations
from autodiff.structures import Number
import numpy as np


def func(x):
    return 5 * x ** 2 + 10 * x - 8

def bfgs(func, initial_guess):

    #bfgs for scalar functions

    x0 = initial_guess

    #initial guess of hessian
    b0 = 1

    fxn0 = func(x0)

    fpxn0 = fxn0.jacobian(x0)

    jacobians = []

    jacobians.append(fpxn0)

    while(np.abs(fpxn0)>1*10**-7):
        fxn0 = func(x0)

        fpxn0 = fxn0.jacobian(x0)

        s0 = -fpxn0/b0

        x1=x0+s0

        fxn1 = func(x1)
        fpxn1 = fxn1.jacobian(x1)


        y0 = fpxn1-fpxn0

        if y0 == 0:
            break

        #delta_b = y0**2/(y0*s0)-b0*s0**2*b0/(s0*b0*s0)
        delta_b = y0/s0-b0
        b1 = b0 + delta_b

        x0 = x1

        b0 = b1

        jacobians.append(fpxn1)
```

(continues on next page)

(continued from previous page)

```
    return x0,func(x0),jacobians

x0 = Number(5)
xstar,minimum,jacobians = bfgs(func,x0)

print("The jacobians at 1st, 2nd and final steps are:",jacobians,'. The jacobian value is 0 in the last
↪step, indicating completion of the optimization process.')
print()
print("The x* is", xstar )
```

```
The jacobians at 1st, 2nd and final steps are: [60, -540.0, 0.0] . The jacobian value is 0 in the last
↪step, indicating completion of the optimization process.

The x* is Number(val=-1.0)
```

CHAPTER

EIGHT

HOW TO EDIT DOCUMENTATION

We're using Sphinx to document `autodiff`. This page has instructions on how to add additional pages of documentation.

The file `index.rst` is the source for the main landing page. While it's in restructured text format, most of the documentation can be written in markdown (this page was written in markdown). To add a new page, all we need to do is create a new markdown file and tell `index.rst` to include it in the main table of contents:

1. Create a new file in the `source` directory.
2. Add that file's relative path to the `toctree` in `index.rst`

```
.. toctree::
    :maxdepth: 2
    :caption: Contents:

    introduction.md
    test-page/how-to-edit-documentation.md
    test-page/new-test-page.md
    api-doc/modules.rst
```

3. Once that's done, run `make html` from the `docs` directory. Doing so will create `/docs/source/html/`. Opening `index.html` in a browser will show a local copy of our documentation page. Our documentation site does the same thing every time we push code to github.

The markdown syntax Sphinx uses is slightly different than Jupyter's. This test page has some examples of markdown that works.

CHAPTER

NINE

AUTODIFF

9.1 autodiff package

9.1.1 Submodules

9.1.2 autodiff.operations module

Collection of tools to make new elementary operations

`autodiff.operations.`**`add`**`(`*x*, *y*`)`

add two numbers together, one of x and y has to be a Number object

Parameters

- **x** – a Number object or an int/float
- **y** – a Number object or an int/float to be added

Returns value of the sum

`autodiff.operations.`**`add_deriv`**`(`*x*, *y*`)`

Derivative of additions, one of x and y has to be a Number object

Parameters

- **x** – a Number
- **y** – a Number object or an int/float to be added

Returns The derivative of the sum of x and y

`autodiff.operations.`**`cos`**`(`*x*`)`

Take the cos(x)

Parameters **x** (`Number`) – Number to take the cos of

Returns cos(x.val)

Return type float

`autodiff.operations.`**`cos_deriv`**`(`*x*`)`

Derivative of cos(x)

Parameters **x** (`structures.Number()`) – Number to take the cos of. Must have a `deriv` attribute

Returns dictionary of partial derivatives

Return type dict

`autodiff.operations.`**`div`**`(`*x*, *y*`)`

Subtract one number from another, one of x and y has to be a Number object

Parameters

- **x** – a Number object
- **y** – a Number object or an int/float to be subtracted

Returns value of the difference

`autodiff.operations.`**`div_deriv`**`(`*x*, *y*`)`

Derivative of division, one of x and y has to be a Number object

Parameters

- **x** – a Number
- **y** – a Number object or an int/float to be divided

Returns The derivative of the quotient of x and y

`autodiff.operations.`**`elementary`**`(`*deriv_func*`)`

Decorator to create an elementary operation

This takes as an argument a function that calculates the derivative of the function the user is calculating. When the decorated function is called, *@elementary* also calls *deriv_func* and stores both the value and derivative in a new *Number()* object

Example

```
>>> import numpy as np
>>> def sin_deriv(x):
...     return {x: np.cos(x.val) * x.deriv[x]}
>>> @elementary(sin_deriv)
... def sin(x):
...     return np.sin(x.val)
>>> a = Number(np.pi / 2)
>>> sina = sin(a)
>>> sina.val
1.0
>>> sina.deriv[a]
0.0
```

Parameters **`deriv_func`** (*`function`*) – Function specifying the derivative of this function. Must return a dictionary where each key-value pair is the partial derivative of the decorated function

Returns Decorated function

Return type function

`autodiff.operations.`**`exp`**`(`*x*`)`

Take the exp(x)

Parameters **x** (`Number`) – Number to take the exp of

Returns exp(x.val)

Return type float

`autodiff.operations.`**`exp_deriv`**`(`*x*`)`

Derivative of exp(x)

Parameters **x** (`structures.Number()`) – Number to take the exp of. Must have a `deriv` attribute

Returns dictionary of partial derivatives

Return type dict

`autodiff.operations.`**`log`**(*x*, *y=2.718281828459045*)

Take the log(x) at base y

Parameters

- **x** (`Number`) – Number to take the log of
- **y** (*a Number object or an int/float*) – Base of the logarithm.

Returns log(x.val)

Return type float

`autodiff.operations.`**`log_deriv`**(*x*, *y=2.718281828459045*)

Derivative of log(x) at base y

Parameters

- **x** (`structures.Number()`) – Number to take the log of. Must have a `deriv` attribute
- **y** (*a Number object or an int/float*) – Base of the logarithm.

Returns dictionary of partial derivatives

Return type dict

`autodiff.operations.`**`mul`**(*x*, *y*)

Subtract one number from another, one of x and y has to be a Number object

Parameters

- **x** – a Number object
- **y** – a Number object or an int/float to be subtracted

Returns value of the difference

`autodiff.operations.`**`mul_deriv`**(*x*, *y*)

Derivative of multiplication, one of x and y has to be a Number object

Parameters

- **x** – a Number
- **y** – a Number object or an int/float to be multiplied

Returns The derivative of the product of x and y

`autodiff.operations.`**`negate`**(*x*)

`autodiff.operations.`**`negate_deriv`**(*x*)

`autodiff.operations.`**`pow_deriv`**(*x*, *a*)

Derivative of power of a Number

Parameters

- **x** – a Number
- **a** – a Number

Returns The derivative of the power

`autodiff.operations.`**`power`**`(x, y)`

Subtract one number from another, one of x and y has to be a Number object

Parameters

- **x** – a Number object
- **y** – a Number object or an int/float to be subtracted

Returns value of the difference

`autodiff.operations.`**`sin`**`(x)`

Take the sin(x)

Parameters **x** (`Number`) – Number to take the sin of

Returns sin(x.val)

Return type float

`autodiff.operations.`**`sin_deriv`**`(x)`

Derivative of sin(x)

Parameters **x** (`structures.Number()`) – Number to take the sin of. Must have a `deriv` attribute

Returns dictionary of partial derivatives

Return type dict

`autodiff.operations.`**`subtract`**`(x, y)`

Subtract one number from another, one of x and y has to be a Number object

Parameters

- **x** – a Number object
- **y** – a Number object or an int/float to be subtracted

Returns value of the difference

`autodiff.operations.`**`subtract_deriv`**`(x, y)`

Derivative of subtractions, one of x and y has to be a Number object

Parameters

- **x** – a Number
- **y** – a Number object or an int/float to be subtracted

Returns The derivative of the difference of x and y

`autodiff.operations.`**`tan`**`(x)`

Take the tan(x)

Parameters **x** (`Number`) – Number to take the tan of

Returns tan(x.val)

Return type float

`autodiff.operations.`**`tan_deriv`**`(x)`

Derivative of tan(x)

Parameters **x** (*`structures.Numbers()`*) – Number to take the tan of. Must have a `deriv` attribute

Returns dictionary of partial derivatives

Return type dict

9.1.3 autodiff.structures module

Data structures for autodiff

`class autodiff.structures.`**`Number`**`(`*val*, *deriv=None*`)`

Bases: `object`

Number class is the core data structure for 'autodiff'. It instantiates a Number object by specifying a value and a derivative. Derivative with respect to itself is automatically instantiated to 1.

Parameters

- **`val`** – value of the Number
- **`deriv`** – a dictionary of partial derivatives. It is automatically instantiated to {self: 1} unless otherwise specified.

Returns Number, an object to perform automatic differentiation on.

Example

```
>>> import autodiff
>>> x = autodiff.Number(3)
>>> x.value
3
>>> x.deriv[x]
1
>>> a = autodiff.Number(3,2)
>>> a.value
3
>>> a.deriv[a]
2
```

`cos``()`

Calculates the cosine of the Number object.

Returns another Number object, which is cosine of the original one.

`exp``()`

Calculates the exponential of Number object.

Returns another Number object, which is the exponential of the original one.

`jacobian``(`*order*`)`

Returns the jacobian matrix by the order specified.

Parameters **`order`** – the order to return the jacobian matrix in. Has to be not null

Returns a list of partial derivatives specified by the order.

`sin``()`

Calculates the sin of the Number object.

Returns another Number object, which is sin of the original one.

`tan``()`

Calculates the tangent of the Number object.

Returns another Number object, which is tangent of the original one.

9.1.4 Module contents

CHAPTER

TEN

PROPOSED EXTENSION

As an extension of the `autodiff` package, we would like to propose making two additional submodules that make use of automatic differentiation: an `optimization` package that contains an easy-to-use API for finding the optima of scalar- and vector-valued functions, and a `rootfinding` package that can find roots of scalar- and vector-valued functions.

These will be two new new submodules under the `autodiff` package, so the new directory structure will be:

```
.
├── autodiff
│   ├── __init__.py
│   ├── operations.py
│   ├── optimization.py
│   ├── rootfinding.py
│   └── structures.py
...
```

For `autodiff.optimization`, we will allow the user to use two different optimization algorithms: `L-BFGS` and gradient descent. These will be an extension of the optimization example in our current `docs` folder.

In `autodiff.rootfinding`, we will have convenient API that allows the user to find the roots of complex functions using Newton's method, an extension of the `root_finding` example in our `docs` folder.

10.1 Proposed use cases

10.1.1 Optimization

```
>>> from autodiff import optimize
>>> def rosenbrock(x, y, a=1, b=100):
...     """A test function to optimize"""
...     return (a - x) ** 2 + b * (y - x ** 2) ** 2

>>> # Find correct answer for optima and root finding
>>> test_func = lambda p: rosenbrock(p[0], p[1])
>>> x_0 = autodiff.array([20, 20])
>>> opt = optimization.Optimizer(func=test_func, x_0=x_0, algorithm='L-BFGS')

>>> # Run the optimization
>>> opt.run()
autodiff.array([1., 1.])

>>> # Analyze performance
```

(continues on next page)

(continued from previous page)

```
>>> opt.time
Wall clock time: 1 s
CPU time (user): 0.5 s
CPU time (sys): 0.4 s
```

10.1.2 Root finding

```
>>> from autodiff import rootfinding
>>> def test_func(x,):
...     return 0.5 * x ** 2 + 2 * x + 1

>>> rf = rootfinding.RootFinder(test_func, x_0=0)
>>> rf.run()
>>> rf.root()
... -3.41421356237309
```

10.2 Comparison to finite differencing

We will also include finite-differencing versions of the optimization and root finding algorithms in `optimization` and `rootfinding` as instructional tools to demonstrate the differences between automatic differentiation and finite differencing.

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX