# Anno Domini's Documentation

# 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.

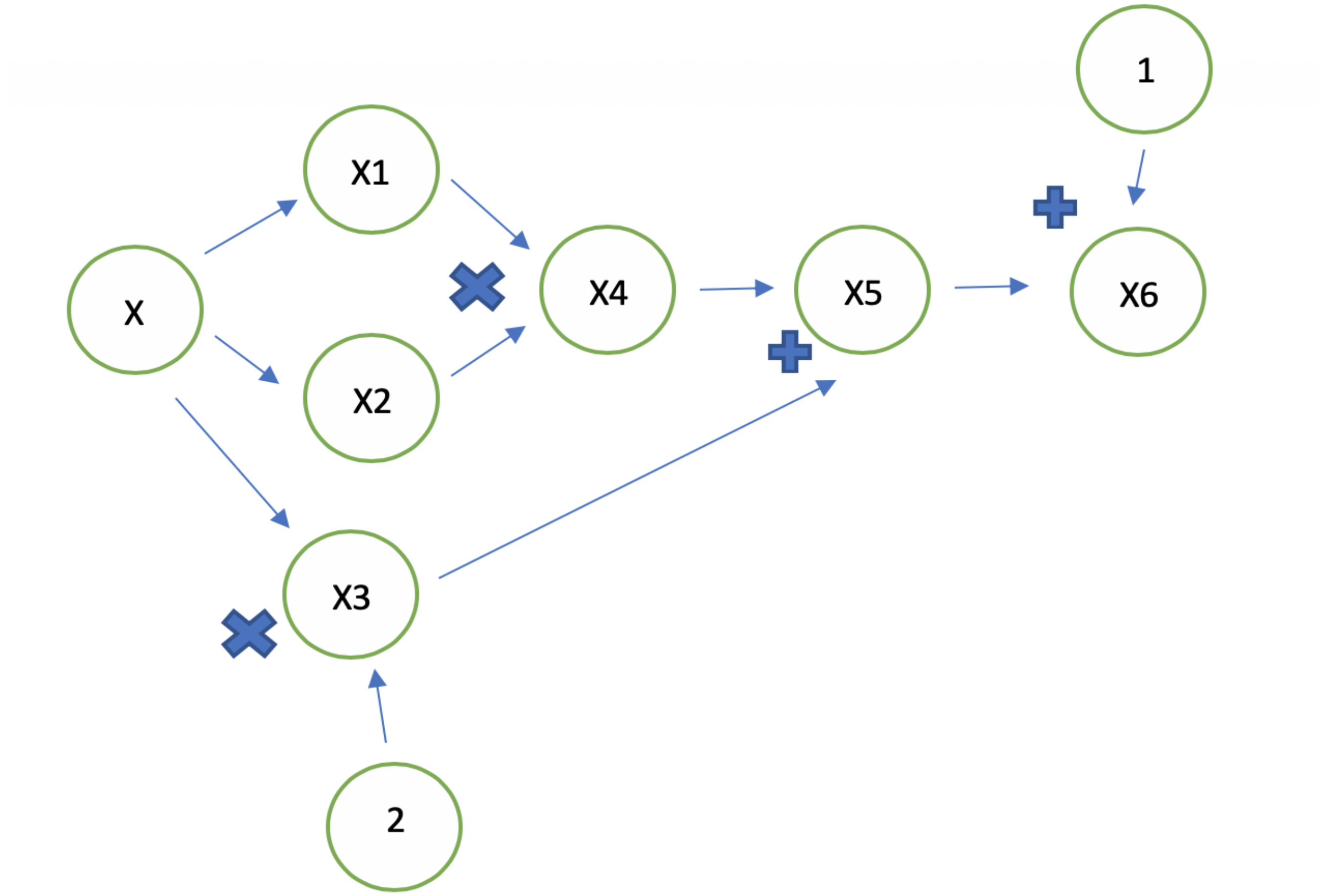
Automatic differentiation is an algorithm that can solve complex derivatives in a way that reduces these compounding floating point errors. The algorithm achieves this by breaking down functions into their elementary components and then calculates and evaluates derivatives at these elementary components. 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.

# 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 Derivati |
|---|---|---|---|
| $x_1$ | $x_1$ | x | 1 |
| $x_2$ | $x_2$ | x | 1 |
| $x_3$ | $x_3$ | x | 1 |
| $x_4$ | $x_1 x_2$ | $x^2$ | $\dot{x_1} x_2 + x_1 \dot{x_2}$ |
| $x_5$ | $x_4 + 2x_3$ | $x^2 + 2x$ | $\dot{x_4} + 2\dot{x_3}$ |
| $x_6$ | $x_5 + 1$ | $x^2 + 2x + 1$ | $\dot{x_5}$ |

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.

# How to use Anno Domini

## 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
>> AD.AutoDiff(3)
Function Value: 3 | Derivative Value: 1.0
>> quit()
$ deactivate
```

*Note*: Numpy and Pytest are also required. If they are missing an error message will indicate as much.

## How to Use

Consider the following example in scalar input case:s Suppose we want to to find the derivative of $x^2 + 2x + 1$. We can the utilize the AnnoDomini package as follows:

```
import AnnoDomini.AutoDiff as AD
f = lambda x: x**2 + 2*x + 1
temp = AD.AutoDiff(1.5)
print(temp)
>> Function Value: 1.5 | Derivative Value: 1.0
df = f(temp)
>> Function Value: 6.25 | Derivative Value: 5.0
```

Say we only want to access only the value or derivative component. We can do this as follows:

```
val, der = df.val, df.der
print(der)
>> 5.0
print(val)
>> 6.25
```

# Software Organization

## Directory Structure

```
AnnoDomini/
  AutoDiff.py
docs/
  source/
    .index.rst.swp
    conf.py
    index.rst (documentation file)
  Makefile
  make.bat
  milestone1.md
tests/
  initial_test.py
  test_AutoDiff.py
.gitignore
.travis.yml
LICENSE
README.md
```

## 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).

## Testing

- Where do the tests live?
- How are they run?
- How are they integrated?

Our tests are contained in tests directory. test_AutoDiff.py is used to test the functions in the AutoDiff Class.

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.

```

tests/initial_test.py .                                           [  3%]
tests/test_AutoDiff.py .........................                  [100%]

----------- coverage: platform linux, python 3.6.7-final-0 -----------
Name                     Stmts   Miss  Cover
--------------------------------------------
AnnoDomini/AutoDiff.py     143      1    99%


============================== 26 passed in 0.48s ==============================
The command "pytest --cov AnnoDomini" exited with 0.

$ codecov                                                  after_success   1.04s

Done. Your build exited with 0.
                                                                           Top
```

## 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; after we notice that the package is mature, 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.

# Implementation Details

The *AutoDiff* class 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 $x = 5.0$).

Central to this entire process is the *AutoDiff* class which is initiated by:

```
class AutoDiff:
    def __init__(self, val=0.0, der=1.0):
        self.val = val
        self.der = der
```

As mentioned above, the *AutoDiff* class has its own methods that define its behavior for common elementary functions such as addition and multiplication. Specifically, we currently have the following methods implemented:

```python
def __add__
def __radd__
def __sub__
def __rsub__
def __mul__
def __rmul__
def __truediv__
def __rtruediv__
def __pow__
def __rpow__
def __neg__
def sqrt
def sin
def cos
def tan
def arcsin
def arccos
def arctan
def sinh
def cosh
def log
def exp
def logistic
```

As a simple illustration, here is the way the *__add__* method is implemented:

```python
def __add__(self, other):
    try:
        val = self.val + other.val
        der = self.der + other.der
    except AttributeError:
        val = self.val + other
        der = self.der
    return AutoDiff(val, der)
```

We can see that the method returns a new *AutoDiff* object with new updated *val* and *der*.

Note that many methods in the *AutoDiff* class, such as *cos* and *exp*, rely on their counterparts in NumPy (e.g., *numpy.cos* and *numpy.exp*). NumPy will play even more important role in our future development to support multiple functions of multiple inputs as NumPy arrays support fast and effective vectorized operations.

The following code shows a deeper example of how our *AutoDiff* class is implemented and useful. Consider again the function, $x^2 + 2x + 1$. Suppose we want to use Newton's Method to find the root, using our package. Then we have:

# Future Features

For our future feature, our idea is to provide an extensive guide of Newton's Method implementation that includes scalar and vector valued functions with single and multiple inputs. As a part of this guide, we will provide visuals and in depth explanation of the algorithm. Provided we have enough time, we could also generate a nice, cleaned up trace table as a visual for the demo functions we use. Additionally, we could also provide an extensive guide of the Hamiltonian Monte Carlo.