# Introduction:

This software implements automatic differentiation, a computationally ergonomic method to compute first derivatives of functions. As first derivatives are the heart of Jacobian matrices, which in turn are the heart of many methods in dynamical systems and statistics including linearization techniques like Newton's Method, determination of stability of dynamic systems through observation of eigenvalues, bifurcation analysis, and nonlinear least squares regression.Systems can be modelled through explicit coding of the derivation of a model, but this method is tedious and prone to errors. Alternatively, symbolic derivation through a graph based analysis can be used to find derivatives; however, these methods aren't conducive to interpretative code environments such as Python. This leaves numerical methods such as finite-differences and automatic differentiation to bridge the gap. Finite differences approximates derivatives from first principles, using some small value of $\epsilon$ to approximate the limit to zero; however, these systems can suffer from accuracy problems, and choosing a suitable value $\epsilon$ becomes non-trivial. The last alternative, automatic differentiation, is what this package implements. Automatic differentiation is exact, efficient, and amenable to OOP systems. This library provides a resource for automatic differentiation.

# Background:

In it's simplest form, automatic differentiation (AD) approximates the gradient for a differentiable function. Its forward mode implementation is based on the notion that a differentiable function consists of a finite set of differentiable elementary functions, which the functional form of the derivative is known. In other words, a given function can be decomposed into multiple elementary steps, which forms a primal trace. The derivative of each elementary step, or node, is computed recursively and ultimately forms the tangent trace. By using the chain rule, the derivatives of the node are combined to approximate the gradient (or jacobian for multivariate functions) for a given value.

The chain rule is integral to AD methods. Utilizing the chain rule allows for the accumulation of the tangent trace and computation of the derivative. In general, the chain rule will be used in AD to compute the gradient of a function $f = f(y(x))$ where $y \in \mathbb{R}^n$ and $x \in \mathbb{R}^m$:

$\nabla_x f = \sum_{i=1}^{n} \frac{\partial f}{\partial y_i} \nabla y_i(x)$

Importantly, forward mode AD computes $\nabla f * p$, where p is a seed vector. If $f$ is scalar, forward AD computes the gradient, whereas it will compute the Jacobian matrix ($\frac{\partial f_i}{\partial x_j}$) if $f$ is a vector. In this package, forward AD is implemented using a dual numbers approach. This approach computes the value of the function and its gradient (or Jacobian) for a given input in parallel. More specifically, an input variable can be decomposed into a real and dual part:

$x = a + b\varepsilon$

where $a, b \in \mathbb{R}^m$, $\varepsilon^2 = 0$, and $\varepsilon \neq 0$. Here, a value $x$ is converted to a real part, $a$, and dual part, $b$. Using these principles, the dual numbers method can be demonstrated using a Taylor series expansion:

$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \dots$

Because $\varepsilon^2 = 0$, this expression simplifies to:

$f(x) = f(a + b\varepsilon) = f(a) + f'(a)b\varepsilon$

Therefore, dual numbers computes the primal trace of the real part and the tangent trace of the dual part simultaneously. This expression also generalizes to high-order functions.

In brief, reverse AD utilizes forward pass and reverse pass in sequence. The forward determines the primal trace of the function and computes the partial derivative with respect to the parent node, and the gradient of the function is computed by accumulating the values in the reverse direction during the reverse path. Similar to forward pass, the reverse mode hinges on principles of the chain rule, although it is not explicitly applied for this mode. More specifically, during the forward pass, intermediate variables are determined and the adjoint values for all the variables are computed in the reverse pass. Thus, the partial deriviatives are computed with respect to intermediate variables, which is distinct from foward mode.

# How to Use:

`autoDiff` is distributed using setuptools, and users can install it using pip-install. The package will live temporarily on test.pypi.org while in development. **For this milestone, you should clone the repository into a folder, begin a virtual environment, pip install build, and then complete the following steps in the root of the cloned repository:**

```
python3 –m build
python3 –m pip install .
```

**You will then be able to import all modules described below by typing the following into any test script:**

```
from autoDiff import *
```

For the next deadline the installation `autoDiff` will be conducted via pip from PyPi, such that the user can simply use:

```
pip –m install autoDiff
```

Using this package will require dependencies: Numpy and pytest (if the user is testing tasks). The setup.py and pyproject.toml files contains instructions to install these packages when the user installs our package.

After installing `autoDiff` , the user can utilize three core functions: `derivative` , `gradient` , and `jacobian` which performs forward mode AD for univariate scalar inputs as well mutlivariate scalar/vector inputs. In its current form, the core functions require a defined function and value (i.e. value to evaluate the derivative/gradient) as the passing arguments. To execute all functions, the user must import the all the modules which are integrated into the `differentation.py` module

```
#import functions from package modules
>>> from autoDiff import *
```

The user must then define a function to evaluate. For univariate functions with a scalar input, the derivative can be computed. For multivariate functions with scalar or vector inputs, the gradient or jacobian can be computed, respectively. The following example demonstrates how to implement the package for the core functions:

```
#define functions to evaluate for derivative
>>> def fn(x):
        return sin(x)

#compute the derivative
>>> deriv = derivative(fn,0)
```

This function will return a single value, the derivative for the function $sin(x)$ for $x = 0$. Similarly, the directional derivative or gradient can be computed for multivariate functions with a scalar input:

```
#define functions to evaluate for gradient

>>> def fn(x,y):
        return sin(x) + cos(y)

#compute the gradient
>>> grad = gradient(fn,[0,1])
```

Here, the gradient is computed for the function, $sin(x) + cos(y)$ for the values $x = 0$ and $y = 1$. The gradient function will return an array with $\frac{\partial f}{\partial x}$ as the first value and $\frac{\partial f}{\partial y}$ as the second value. At this point, the user also has the option to compute the jacobian. The `jacobian` function will return the vector valued matrix of the first order partial derivatives. The user can compute the jacobian using the following example:

```
#define functions to evaluate for jacobian
>>> def fn(x,y):
        return sin(x)

>>> def fn2(x,y):
        return sin(x) + cos(y)
```

```
#compute the jacobian
>>> jacob = jacobian([fn,fn2],[0,1])
```

The jacobian is computed for matrix:

$$\begin{matrix} sin(x) & 0 \\ sin(x) & cos(y) \end{matrix}$$

The resulting matrix is:

$$\begin{matrix} cos(x) & 0 \\ cos(x) & -sin(y) \end{matrix}$$

and will be evaluated at the $x = 0$ and $y = 1$. The `jacobian` function will return a matrix, like above, evaluated at a given x and y value.

The current implementation requires that the functions are specifically implemented like the examples above in order to retrieve the derivatve, gradient and jacobian. A more detailed example can be found the example directory.

## Software Organization:

**Directory Structure**: The package is organized with a series subpackages which implement basic functionalities to perform autodifferentiation.

```
autoDiff
├── README.md
├── pyproject.toml
├── LICENSE
├── setup.py
├── examples
   ├── rootfinding.py
├── src
    ├── autoDiff
        ├── __init__.py
        ├── dualNum.py
        ├── elemFunctions.py
        ├── differentiation.py
        ├── node.py
├── tests
   ├── autodiff_tests
        ├── test_elem.py
        ├── test_dual.py
        ├── test_differentiation.py
   ├── run_tests.sh
```

**Modules**:

- `node.py`
  - Given a function, returns a node in the computational graph. Each node has three attributes: a value, the derivative, and a list of parent nodes. (for future extension)
- `elemFunctions.py` :
  - Overloads elementary operation functions and stores their derivatives. For an a DualNumber input, this modules calculates the value and its derviative. However, if the input is another type, we resort to numpy implementations of mathematical operators.
- `dualNum.py` :
  - Implements a DualNumber class and overloads basic numerical and mathematical operations of python. The class implements operators to calculate the the numerical value of a given function and at its derivatives. There is also a derivative function which implements the chain rule for non-elementary operations.
- `differentiation.py` :

- Contains methods which implements forward mode AD, reverse mode (for future extension), and computes the derivative a gradient and jacobiann calculation for higher level functions.

**Test Suite**

- The test suite lives in the root directory. Each module in the package is tested using pytest. Code coverage of >90% is ensured using pytest-cov package.The tests are integrated into the CI workflow, using yml extension filed linked to our Github project. The `run_tests.sh` and `code_coverage.sh` bash scripts are called in the test and coverage yml extension files.

# Implementation

**Core attributes and methods**

- In its current form, foward mode AD is achieved by integrating the `dualNum.py` and `elemFunctions.py` modules in the driver module, `differentiaion.py` . As noted above, the core attributes of this module are the `derivative` (scalar input), `gradient` (for multivariate scalar input), `jacobian` (multivariate vector input). Specifically, we use a Dual Numbers method to compute the derivative/gradient/jacobian for a given function and value. The `dualNum.py` module implements a `DualNumber` object. A `DualNumber` type has two attributes: value and derivative. This class overloads basic numerical type of python and implement numerical operators (+, -, *, etc). These overloaded operators return a `DualNumber` and calculate the value and derivative. In its current form, the following methods have been overloaded:

  - `__mul__` , `__rmul__` , `__add__` , `__radd__` , `__sub__` ,
  - `__truediv__` , `__rtruediv__` , `__pow__` , `__rpow__` , `__gt__`
  - `__le__` , `__eq__` , `__ne__` , `__neg__` , `__rsub__`

  -Similarly, the `elemFunctions.py` contains the known derivatives of elementary operations using `numpy` . This modules utilizes `DualNumbers.py` , and overloads unary functions, which include the following functions:

  - `sin` , `cos` , `tan` , `arcsin` , `arccos`
  - `arctan` , `sinh` , `cosh` , `tanh` , `exp`
  - `log` , `sigmoid` , `power/sqrt`

**Core data structures**

- The core data structures used in this package are matrices, vectors, and lists.

**Multivariate functions**

- To handle cases for $f : \mathbb{R}^m \rightarrow \mathbb{R}$, we added several independent partial derivative components to our dual numbers. In our `differentiation.py` module, we added a `gradient` function which calculates the directional derivative. To calculate the directional derivative, $\nabla f(x) * p$, the user needs to provide, p, a seed vector defining the directional derivative of f in the direction p. We use our DualNumbers class, and initialize the seed vector as the first derivative component. A similar approach is used for the `jacobian` fucntion

**External dependencies**

- Our main external dependencies at this point are only numpy which we use to test our package modules as well as for implementation of the elementary functions module.

# Future Extension

For our future extension, we plan to implement reverse mode AD. Reverse mode AD is preferred for functions with many inputs and few outputs (whereas the inverse is preferred for forward mode AD). Reverse mode is achieved in two steps: 1) a forward pass, which evaluatels the elementary functions ans stores the partial derviative and 2) a reverse pass which derivatives relative to each parent node. Ultimately, reverse mode AD will compute the Jacobian transpose-product. Implementation of reverse mode will require re-designing our current package module because reverse mode requires

construction of a computational graph. This construction will be implemented in the `node.py` (currently empty file in our repo). We envision our node module to perform the forward pass, where each elementary operation is represented as a node and the children of the node are stored. Subsequently, it will also implement reverse pass where the nodes are traversed and the gradient will be computed each node until reaching the input variables. Thus our implementation will traverse through the graph and compute the derivative with respect to every node. The `node.py` module will have two main attributes: `value` and the `local_grad`, or the derivative with respect to an individual node. Using this approach, we will sum the derivatives of all the children per parent node and propogate computation of the derviatives recursively.

Implementing reverse mode will likely change the structure our module. For example, we will modify our `differentiation.py` module to have a class, such that we can AD objects for reverse or forward autoDiff. An argument made upon instantation of the AD object can be used to implement either forward or reverse mode.

## Licensing

For our project's license, we are deciding to use the MIT License. We want a permissive license allowing the distribution of our package without frills or profits, and in a simple, familiar, and straightforward manner because we want it to be as accessible for use and modification as possible; thus, the MIT License makes the most sense, as it is permissive, brief, and the most popular open source license today.

As our only dependencies are numpy, matplotlib and pandas, which are all BSD licenses, we do not have to consider other copyright. Others may freely advertise and monetize software that makes use of our package, but users of our software need to include the MIT License in their use of our code. The copyright holder is Harvard University, as this software was produced in a Harvard University course setting.

It can be found in the LICENSE file in our root directory.

In [ ]: