Introduction

Describe the problem the software solves and why it's important to solve that problem.

- Our software package, ad-AHJZ, computes gradients by leveraging the technique of automatic differentiation. Before we can understand automatic differentiation, we must first describe and motivate the importance of differentiation itself. Derivatives are vital to quantifying the change that's occurring over a relationship between multiple factors. Finding the derivative of a function measures the sensitivity to change of a function value with respect to a change in its input argument. Derivatives generalize across multiple scenarios and are well defined for both scalar inputs and outputs, as well as vector inputs and outputs. Derivatives are not only essential in science applications, but numerically solving differential equations and optimizing and solving linear systems, they are useful in many real world, scientific settings. For example, in finance they help analyze the change regarding the portfolios for a business or finding the minimum amount of material to construct a building. In physics, they help calculate the speed and distance of a moving object. Derivatives are crucial to understanding how such relationships change and move.
- To perform differentiation, two different approaches are solving the task symbolically or numerically computing the derivatives. Symbolic differentiation yields accurate answers, however depending on the complexity of the function, it could be expensive to evaluate and result in inefficient code. On the other hand, numerically computing derivatives is less expensive, however it suffers from potential issues with numerical stability and a loss of accuracy.
- Our software package, ad-AHJZ, overcomes the shortcomings of both the symbolic and numerical approach. Our package uses automatic differentiation which is less costly than symbolic differentiation, but evaluates derivatives at machine precision. The technique leverages both forward mode and backward mode and evaluates each step with the results of previous computations or values. As a result of this, automatic differentiation is less costly, to evaluate the entire analytical expression to compute the derivative and is hence iteratively evaluating a gradient based on input values. Thus, based on these key advantages, our library implements and performs forward mode automatic differentiation to efficiently and accurately compute derivatives.

Background

Describe (briefly) the mathematical background and concepts as you see fit.

Part 1: Chain Rule

The underlying motivation of automatic differentiation is the Chain Rule that enables us to decompose a complex derivative into a set of derivatives involving elementary functions of which we know explicit forms.

We will first introduce the case of 1-D input and generalize it to multidimensional inputs.

One-dimensional (scalar) Input: Suppose we have a function $f(y(t))$ and we want to compute the derivative of f with respect to t. This derivative is given by:

$$\frac{df}{dt} = \frac{df}{dy}\frac{dy}{dt}$$

Before introducing vector inputs, let's first take a look at the gradient operator ∇

That is, for $y: \mathbb{R}^n \to \mathbb{R}$, its gradient $\nabla y: \mathbb{R}^n \to \mathbb{R}^n$ is defined at the point $x = (x_1, \ldots, x_n)$ in n-dimensional space as the vector:

$$\nabla y(x) = \begin{bmatrix} \frac{\partial y}{\partial x_1}(x) \\ \vdots \\ \frac{\partial y}{\partial x_n}(x) \end{bmatrix}$$

Multi-dimensional (vector) Inputs: Suppose that we have a function $f(y_1(x), \ldots, y_n(x))$ and we want to compute the derivative of f with respect to x. This derivative is given by:

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

We will introduce direction vector p later to retrieve the derivative with respect to each y_i.

Part 2: Jacobian-vector Product

The Jacobian-vector product is equivalent to the tangent trace in direction p if we input the same direction vector p.

$D_p v = J p$

Part 3: Seed Vector

Seed vectors provide an efficient way to retrieve every element in a Jacobian matrix and also recover the full Jacobian in high dimensions.

Scenario: Seed vectors often come into play when we want to find $\frac{\partial f_i}{\partial x_j}$, which corresponds to the i, j element of the Jacobian matrix.

Procedure: In high dimension automatic differentiation, we will apply seed vectors before we start the forward trace such that we have recursively calculated the explicit forms of tangent trace of f_i's and then multiply each of them by the indicator vector p_j where the j-th element of the p vector is 1.

Part 4: Evaluation (Forward) Trace

Definition: Suppose $x = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$, we defined $v_{k-m} = x_k$ for $k = 1, 2, \ldots, m$ in the evaluation trace.

Motivation: The evaluation trace introduces intermediate results v_{k-m} of elementary operations to track the differentiation.

Consider the function $f(x): \mathbb{R}^2 \to \mathbb{R}$:

$f(x) = \log(x_1) + \sin(x_1 + x_2)$

We want to evaluate the gradient ∇f at the point $x = \begin{bmatrix} \frac{\pi}{2} \\ \frac{\pi}{3} \end{bmatrix}$. Computing the gradient manually:

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} \frac{1}{x_1} + \cos(x_1 + x_2) \\ \cos(x_1 + x_2) \end{bmatrix} = \begin{bmatrix} \frac{2}{\pi} + \cos(\frac{5\pi}{6}) \\ \cos(\frac{5\pi}{6}) \end{bmatrix}$$

Forward primal trace	Forward tangent trace	Pass with $p = [1, 0]^T$	Pass with $p = [0, 1]^T$
$v_{-1} = x_1$	$D_p v_{-1} = p_1$	1	0
$v_0 = x_2$	$D_p v_0 = p_2$	0	1
$v_1 = v_{-1} + v_0$	$D_p v_{-1} + D_p v_0$	1	1
$v_2 = \sin(v_1)$	$\cos(v_1) D_p v_1$	$\cos(v_1)$	$\cos(v_1)$
$v_3 = \log(v_{-1})$	$\frac{1}{v_{-1}} D_p v_{-1}$	$\frac{1}{v_{-1}}$	0
$v_4 = v_2 + v_3$	$D_p v_2 + D_p v_3$	$\frac{1}{v_{-1}} + \cos(v_1)$	$\cos(v_1)$

$D_p v_{-1} = \nabla v_{-1}^T p = p_1$

$D_p v_0 = \nabla v_0^T p = p_2$

$D_p v_1 = \nabla v_1^T p = (\nabla v_{-1} + \nabla v_0)^T p = D_p v_{-1} + D_p v_0$

$D_p v_2 = \nabla v_2^T p = (\cos(v_1) \nabla v_1)^T p = \cos(v_1) D_p v_1$

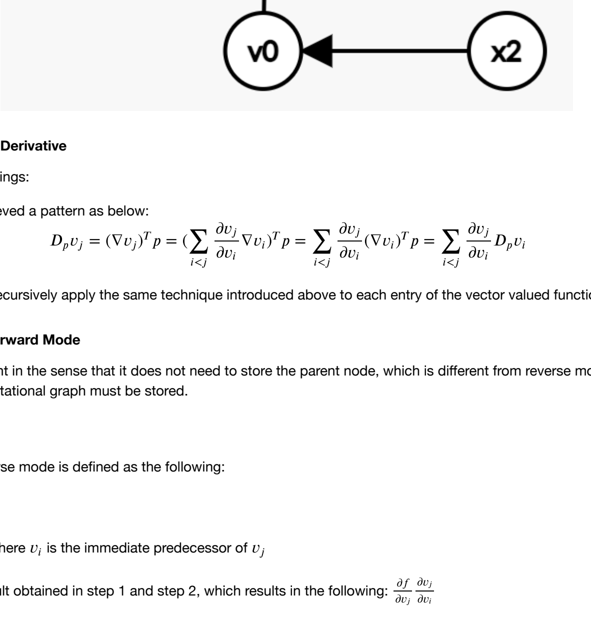
$D_p v_3 = \nabla v_3^T p = (\frac{1}{v_{-1}} \nabla v_{-1})^T p = \frac{1}{v_{-1}} D_p v_{-1}$

$D_p v_4 = \nabla v_4^T p = (\nabla v_2 + \nabla v_3)^T p = D_p v_2 + D_p v_3$

Part 5: Computation (Forward) Graph

We have connected each v_{k-m} to a node in a graph for a visualization of the ordering of operations.

From the above example, its computational graph is given by:



Part 6: Computing the Derivative

Let's generalize our findings:

From the table, we retrieved a pattern as below:

$$D_p v_j = (\nabla v_j)^T p = \left(\sum_i \frac{\partial v_j}{\partial v_i} \nabla v_i\right)^T p = \sum_i \frac{\partial v_j}{\partial v_i} (\nabla v_i)^T p = \sum_i \frac{\partial v_j}{\partial v_i} D_p v_i$$

Higher dimension: We recursively apply the same technique introduced above to each entry of the vector valued function f.

Part 7: Efficiency of Forward Mode

Forward mode is efficient in the sense that it does not need to store the parent node, which is different from reverse mode (see below) where the whole computational graph must be stored.

Part 8: Reverse Mode

The mechanism of reverse mode is defined as the following:

Step 1: Calculate $\frac{\partial f}{\partial v_j}$

Step 2: Calculate $\frac{\partial v_j}{\partial v_i}$ where v_i is the immediate predecessor of v_j

Step 3: Multiply the result obtained in step 1 and step 2, which results in the following: $\frac{\partial f}{\partial v_j}\frac{\partial v_j}{\partial v_i}$

Part 9: Dual Number

Naively: We define a dual number $d_i = v_i + \delta_i$ where $\delta_i = D_p v_i \epsilon$ that satisfies $\epsilon^2 = 0$

A k-th differentiable function f can be written as:

$f(d_i) = f(v_i + \delta_i) = f(v_i) + f'(v_i)\delta_i + \frac{f''(v_i)}{2!}\delta_i^2 + \ldots + \frac{f^k(v_i)}{k!}(\zeta - v_i)^k$ for some $\zeta \in (v_i, v_i + \delta_i)$ by Taylor expansion.

Now we substitute the definition of δ_i back into the above expansion and use the fact that all higher terms go to 0 assuming $\epsilon^2 = 0$. We will have the following:

$f(d_i) = f(v_i) + f'(v_i) D_p v_i \epsilon$

Advantage: Operations on Dual Number pertain to the form of Taylor expansion, which makes the implementation easier to retrive the value and derivative.

Consider the following example:

$$d_i = v_i + D_p v_i \epsilon$$
$$f(d_i) = d_i^2 = v_i^2 + 2v_i D_p v_i \epsilon + D_p v_i^2 \epsilon^2 = v_i^2 + 2v_i D_p v_i \epsilon$$

where v_i^2 refers to the value and $2v_i D_p v_i$ refers to the derivative.

More specifically, v_i^2 corresponds to $f(v_i)$, $2v_i$ corresponds to $f'(v_i)$, and $D_p v_i$ is just $D_p v_i$.

How to Use ad_AHJZ

How do you envision that a user will interact with your package? What should they import? How can they instantiate AD objects?

1. Installing the package:

- 1a. User can install the package and its dependencies using the virtual environment venv:

```
# Create a directory to store your virtual environment(s).
mkdir ~/.virtualenvs
python3 -m venv ~/.virtualenvs/env_name
# Activate your env_name virtual environment.
source ~/.virtualenvs/env_name/bin/activate
python3 -m pip install ad-AHJZ
python3 -m pip install -r requirements.txt
echo >'file_name'.py
```

- 1b. User can install the package and its dependencies using the virtual environment conda:

```
# Create a directory to store your virtual environment(s).
mkdir 'directory_name_for_virtual_environment'
cd 'directory_name_for_virtual_environment'
conda create -n 'env_name' python=3.7 anaconda
# Activate your env_name virtual environment.
source activate env_name
python3 -m pip install ad-AHJZ
python3 -m pip install -r requirements.txt
echo >'file_name'.py
```

2. Importing the package:

- 2a. User imports package into the desired python file with the following line:

```
from ad_AHJZ import foward_mode, #reverse_mode (once implemented)
```

- 2b. User imports numpy into the desired python file with the following line:

```
import numpy as np
```

3. Calling/Using package modules:

- 3a.Using the class forward_mode() create an automatic differentiation object that can use either a scalar or vector input to obtain both the function value and derivative. Below are examples using a scalar input and a vector input:
- 3b. Example of foward_mode() using a scalar input:

```
# define desired evaluation value (scalar)
x = 0.5
# define a simple function:
f_x = lambda x: np.sin(x) + 2 * x
# create a foward_mode() object using the defined values x, f_x from above
fm = forward_mode(x, f_x)
# option 1: retrieves both the function value and the derivative using get_function_value_and
_jacobian()
x, x_der = fm.get_function_value_and_jacobian()
print(x, x_der)
1.479425538604203
[2.87758256]
# option 2: retrieves only the function value using get_function_value()
x_value = fm.get_function_value()
print(x_value)
1.479425538604203
# option 3: retrieves only the function derivative using get_jacobian()
x_derivative = fm.get_jacobian()
print(x_derivative)
[2.87758256]
```

- 3c. Example of foward_mode() using a vector input:

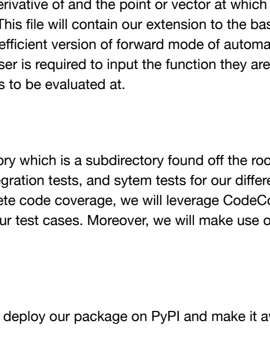
```
# define desired evaluation value (scalar)
multi_input = [0.5, 1]
# define a simple function:
f_xy = lambda x, y: np.sin(x) + 2 * y
# create a foward_mode() object using the defined values x, f_x from above
fm = forward_mode(multi_input, f_xy)
# option 1: retrieves both the function value and the jacobian using get_function_value_and_j
acobian()
multi_xy, multi_xy_der = fm.get_function_value_and_jacobian()
print(multi_xy, multi_xy_der)
2.479425538604203
[0.87758256 2.        ]
# option 2: retrieves only the function value using get_function_value()
multi_xy_value = fm.get_function_value()
print(multi_xy_value)
2.479425538604203
# option 3: retrieves only the function jacobian using get_jacobian()
multi_xy_derivative = fm.get_jacobian()
print(multi_xy_derivative)
[0.87758256 2.        ]
```

Software Organization

Discuss how you plan on organizing your software package.

1. Directory Structure:

- 1a. We include our project directory structure in the image below. Our package is called *ad-AHJZ*, where our code for automatic differentiation lies within "AHJZ_autodiff", our milestone documentation lies within "docs", all unit testing files are located in "testing", and the root of the directory holds our readme, license, and requirements.txt file.
- 1b. Directory structure layout:



2. Modules:

- 2a. *functions.py*: This file contains sixteen methods to compute the function and derivative values of the elementary operations outlined below. These functions form the partial computational pieces required to perform automatic differentiation. The fourteen elementary functions are the following: '+', '-', '*' '/', 'sqrt(x)', 'power(x,n)', 'exp(x)', 'log(x, b)', 'ln(x)', 'sin(x)', 'cos(x)', 'tan(x)', 'cot(x)', 'csc(x)', 'sec(x)'.
- 2b. *forward_mode.py*: This file computes the gradient using automatic differentiation forward mode. A user is required to input the function they are interested in computing the derivative of and the point or vector at which the derivative is to be evaluated at.
- 2c. *fast_forward_mode.py* (extension module): This file will contain our extension to the basic automatic differentiation functionality. In this module we compute the gradient using an efficient version of forward mode of automatic differentation by using an efficient graph data structure and optimized tree traversal. A user is required to input the function they are interested in computing the derivative of and the point or vector at which the derivative is to be evaluated at.

3. Test Suite Location:

- 3a. The test suite will live in the "testing" directory which is a subdirectory found off the root directory (see 1. Directory Structure). The "testing" directory will contain all unit tests, integration tests, and sytem tests for our different modules.
- 3b. To ensure our testing procedure has complete code coverage, we will leverage CodeCov. CodeCov will enable us to quickly understand which lines are being executed in our test cases. Moreover, we will make use of TravisCI in order to see which of our unit tests are passing/failing.

4. Package Distribution:

- 4a. The package will be distributed via PyPI. To deploy our package on PyPI and make it available for others to use, we would also need to add setup.py and setup.cfg files.

5. Packaging the Software:

- 5a. The software will be packaged using PyScaffold, which is a key builder for Python packages pertaining to data science. We plan on using this framework for packaging as it enables us to maintain and upgrade our package using templates and ready-to-use configurations to improve and further develop our program with newer versions of Python and external library dependencies. Moreover, we will use readthedocs to document, build, and host our documentation automatically. With readthedocs, if we were to upgrade the package to new versions, it would seemlessly update our documentation accordingly.

6. Other Considerations: Package Dependencies

- 6a. The only library dependency our package will rely on is numpy. We design our software this way to ensure that we are not creating multiple external dependencies and thereby increase our software's reliability .

Implementation

Discuss how you plan on implementing the forward mode of automatic differentiation.

1. Core Data Structure:

- 1a. Our primary core data structure is going to be a dictionary to store each node in the computational graph that uses the tangent trace and primal trace to compute the partial differentiation and function value for a specific variable. More specifically, the keys would be the node of the computational graph (state name) and the values are going to be a tuple that holds the associated operation at the specific state (function value and derivative).

2. Classes:

- 2a. *Dual Numbers*: Class which represents the way in which operations are computed for a dual number
- 2b. *Forward Mode*: Class which represents the computational graph as a dictionary and performs forward mode differentiation
- 2c. *Fast Forward Mode* (extension module): Class which represents an optimized data structure for how to store the compututional graph and performs a faster version of forward mode differentiation.

3. Method and Name Attributes:

- 3a. *Dual Numbers*:
  - Method to initialize the real value and dual number value for the input of the function
  - Methods to overload the elementary operations for a variable that is dual number. Note that we will overload the following fourteen operators: '+', '-', '*' '/', 'sqrt(x)', 'power(x,n)', 'exp(x)', 'log(x, b)', 'ln(x)', 'sin(x)', 'cos(x)', 'tan(x)', 'cot(x)', 'csc(x)', 'sec(x)'.
    - For example, we would use the below method to overload the "add" operation:

```
def __add__(self, other):
        if isinstance(other, DualNumber):
            return DualNumber(self.real + other.real, self.dual + other.dual)
```

- 3b. *Forward Mode*:
  - Method to initialize the computational graph dictionary, its initial *x* value, and an empty dictionary of the primal trace and tangent trace to perform forward mode automatic differentiation
  - Method to iterate through the input function and append to the primal and tangent trace dictionary
  - Method to get the node's value and partial derivative at the current step of forward mode
  - Method to obtain the primal trace to arrive at the next node's value and partial derivative
  - Method to obtain the tangent trace to arrive at the next node's value and partial derivative
  - Method to run the entire forward mode process and obtain the final function value and derivative from the computational graph.
- 3c. *Fast Forward Mode*:
  - Method to initialize the optimized computational graph data structure, its initial *x* value, and an empty optimized structure of the primal and tangent trace to perform forward mode automatic differentiation
  - Method to iterate through the input function and append to the primal and tangent trace data structure
  - Method to get the node's value and partial derivative at the current step of forward mode
  - Method to obtain the primal trace to arrive at the next node's value and partial derivative
  - Method to obtain the tangent trace to arrive at the next node's value and partial derivative
  - Method to efficiently run the entire forward mode process and obtain the final function value and derivative from the optimized computational graph.

4. External Dependencies:

- 4a. The only external library we will rely on is numpy, which we will use to perform computations and evaluate small expressions with. With this being our only external dependency, our software increases its reliability and can be viewed as a near stand alone software package.

5. Dealing With Elementary Functions

- 5a. For all elementary functions like *sin*, *sqrt*, *log*, and *exp* (and all the others mentioned in *Modules*) we will define separate methods for them in *functions.py*. This module will generalize each of the functions in order to handle both scalar and vector input. Each method will take in as an input a vector or scalar value stored at the previous node in the computational graph and output the derivative value and function value for that elementary function. We can then store the methods' outputs as a tuple in the computational graph dictionary alongside its primal and tangent traces.
- 5b. For example, we would use the below functions to implement *sin* and *sqrt*, both of which work with scalar or vector input *x* values:

```
# input x can be either a scalar or vector value
def sin(x):
   x_val = np.sin(x)
   x_der = np.cos(x)
   return (x_val, x_der)
```

```
# input x can be either a scalar or vector value
def sqrt(x):
   x_val = np.sqrt(x)
   x_der = 0.5 * np.power(x, -0.5)
   return (x_val, x_der)
```

Licensing

Briefly motivate your license choice

Our *ad-AHJZ* package is licensed under the GNU General Public License v3.0. This free software license allows users to do just about anything they want with our project, except distribute closed source versions. This means that any improved versions of our package that individuals seek to release must also be free software. We find it essential to allow users to help each other share their bug fixes and improvements with other users. Our hope is that users of this package continually find ways to improve it and share these improvements within the broader scientific community that uses automatic differentation.

Feedback

2/2 Introduction: Would have been nice to see more about why do we care about derivatives anyways and why is undefined a solution compared to other approaches?

Response: In the Introduction, we addressed these comments by explaining the purpose of derivatives, expanding upon the real-world applications of derivatives, and their generalizability across multiple dimensions. We addressed this comment in the updated first bullet point of the introduction section.

2/2 Background: Good start to the background. Going forward, I would like to see more discussion on automatic differentiation. How do forward mode and reverse mode work? I would also like to see more discussion on what forward mode actually computes (Jacobian-vector product), the "seed" vector, and the efficiency of forward mode.

Response: In the Background, we addressed these comments by adding four new subsections which discuss the topics of reverse mode, jacobian-vector product, seed vectors, and the efficiency of forward mode. This new information is contained in Part 2, Part 3, Part 7, and Part 8, in the background section.

3/3 How to use: Good Job!

Response: N/A

2/2 Software Organization: Nicely Done!

Response: N/A

4/4 Implementation: Classes and methods are very well thought-through. It would be great if you could list all the elementary operations that you will overload.

Response: In the Implementation, we addressed this comment by listing all fourteen elementary operations which we plan on overloading. We addressed this comment in the subsection "3a. Dual Numbers" under the second bullet point.

2/2 License: Good Job!

Response: N/A