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 ϵ to approximate the limit to zero; however, these systems can suffer from accuracy problems, and choosing a suitable value ϵ 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 PackageName:

The package will be be distributed using setuptools, and users will install *autodiff* using pip-install. The package will live temporarily on test.pypi.org while in development. Users should import autodiff and instantiate an object as described below:

```
# install autodiff packages with dependencies in current python environment
python -m pip install autodiff

# example for instantiating a node using ad
>>> from autoDiff import ad
>>> node = ad.Node(f,x)
```

Software Organization:

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

```
autoDiff
├── README.md
├── LICENSE
├── setup.py
├── tests
│   ├── test_autoDiff.py
├── autoDiff
│   ├── __init__.py
│   ├── node.py
│   ├── dualNum.py
│   ├── elemFunctions.py
│   ├── differentiation.py
```

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.
- `elemFunctions.py`:
    - With our base class constructed, we need to write overload operations so we can perform elementary functions (sin, cos, etc.) on different input types. If the input is a DualNumber type, we calculate the value and its derivative. 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, reverse mode, and gradient calculation for higher level functions.

Test Suite

- Our test suite lives in the root directory.

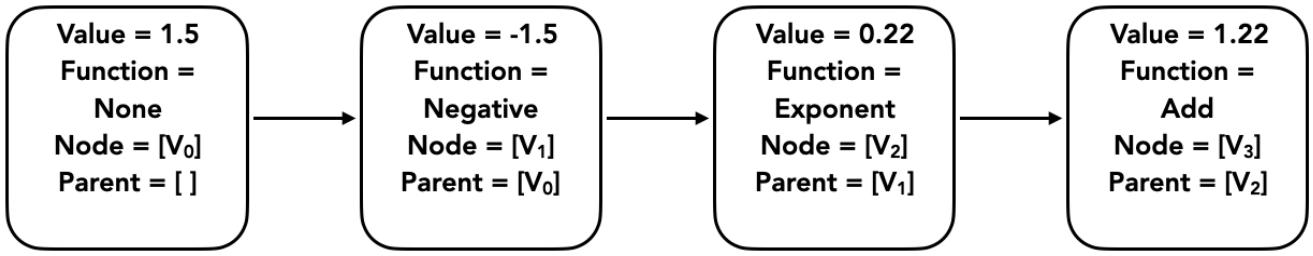
Distribution of Package

- Our package will be distributed using PyPI

Implementation

First, we need to create an abstract class to initalize a computational graph. For forward mode, we do not need to construct the entire computational graph. Instead, we can build a `Node` class which represents one node of the graph and build the graph by tracing the computation in each forward pass. Each node in the graph has three attributes: value (the actual value computed on a particular set of inputs, can be a `DualNumber` type), func (the primitive operation performed on the value), parents (a list which records parent nodes). However, we will also add an additional function which constructs the entire computational graph when we call reverse mode.

Construction of the node will be the most challenging, but we anticipate that we will use concepts pertaining to binary trees to perform the forward trace for linear and nonlinear functions. In order to implicitly build a computational graph, the node module will convert each elementary operation to a node instance. The inputs from the parent node are first decomposed and evaluated at the given value, and the result is converted into a new node instance. A simple schematic of this is shown below for the example function, $f(x) = 1 + e^{-x}$:



Next, we will have to make a class which implements a `DualNumber` object. A `DualNumber` type has two attributes: value and derivative. We will initialize the first value to a point, given by the user and first derivative will also be initalized to 1. This class will need to overload the basic numerical type of python and implement numerical operators (+, -, , *etc). These overloaded operators will return a `DualNumber` and calculate the value and derivative.* *The DualNumber module will store the value and derivative for a given function and input using a dictionary**. For example:

```
def __add__(self, other):
      return DualNumber(
          value = self.value + other.value,
          derivative = self.derivative + other.derivative
      )

def __mul__(self, other):
      return DualNumber(
```

```
            value = self.value*other.value,
            derivative = other.value*self.derivative + self.value*other.derivative
        )
```

This module will also import two other files which implement elementary operations on an input, which could be a scalar, array, or `DualNumber`. Similarly, we will create another module, `elemFunctions`, which contains the derivatives of elementary operations and will be called to calulate elementary derivatives for a intermediate value and return a value of the input type. This module will import DualNumbers. Specifically, the elementary operations module will overload all operations to perform derivative computations. Thus, unary functions and operators such as sine, cosine, square root etc., will be overloaded in a fashion demonstrated below:

```
def sin(x):
  if isinstance(x, DualNumber):
      return DualNumber(
          value = np.sin(x.value),
          derivative = np.cos(x.value) * x.derivative
      )
  else:
      return np.sin(x)

def cos(x):
  if isinstance(x, DualNumber):
      return DualNumber(
          value = np.cos(x.value),
          derivative = -np.sin(x.value) * x.derivative
      )
  else:
      return np.cos(x)
```

This will be useful when we try to implement reverse mode later on. For now, when we are calculating the forward tangent trace and reach a non-elementary operation, we use the chain rule to propagate the information from parent nodes and solve the derivative.

To handle cases for $f : \mathbb{R}^m \rightarrow \mathbb{R}$, we will have to add several independent partial derivative components to our dual numbers. In our `differentiation.py` module, we will have to add a `grad()` function which calculates the directional derivative. To calculate the directional derivative, $\nabla f(x) * p$, we will need the user to provide, p, a seed vector defining the directional derivative of f in the direction p. We should use our DualNumbers class, but initialize the seed vector as the first derivative component.

To handle cases for $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$, we will have to apply the `grad()` method from the `differentiation` module. For each component function $f_i : \mathbb{R}^m \rightarrow \mathbb{R}$, where $f(x) = (f_1(x), f_2(x), \ldots f_m(x))$, the directional derivative needs to be calculated using `grad()` in the n different directions of the seed vector. We will consider using the `__getitem__` operator that executes indexing operations so that function can access the each element of the `node`. **In order to accomodate multiple inputs (i.e. $f(x_1, x_2, .., x_n)$), all inputs wll be converted to Numpy arrays (see next section for anticipated external dependencies).**

We anticipate that our main external dependencies will include numpy, matplotlib and pandas. Utilizing numpy will help deal with vectorized inputs, since this library supports vectorized arguments. As a possible extension, we will explore generating computational graphs, which may require using pandas to handle data structures for the decomposed function and storing its nodes and edges. Finally, we will offer the user the option to visualize their function and its derivative using matplotlib.

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.

Feedback

Milestone 1

Below are our responses (bullet points) to the feedback for Milestone 1. We implemented the feedback into our document - all of our changes are bolded. In addition to the feedback, we also corrected typos and unnecessary/inaccurate wording.

1. Your introduction part can contain some introduction about the derivative and comparison between different numeric methods calculating the derivative.

- We agree that the introduction needs more information regarding how AD performs relative to other numeric methods. We modified this section to emphasize pitfalls of other numerical methods and how AD can help perform computationally precise first derivative approximations. This section now reads: *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 ϵ to approximate the limit to zero; however, these systems can suffer from accuracy problems, and choosing a suitable value ϵ 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.*

2. Good ! Please remember to add details about unary operators such as sin, cos, pow, exp, etc. And please add some information about how you will handle the forward mode. Can you add any concrete example about tracing computation in forward pass and how you are going to design the forward mode.

- We modified the implementation section to explicitly explain how we will treat unary operators in the `elemFunctions` module, and we offer an example for how overloading will be implemented (see section for example)
- We describe how the forward trace will be constructed by using a conceptual schematic. Similar to our previous homework (hw4), the forward trace will be constructed by decomposing a function into individual nodes, and calculating the value and derivative, with respect to a seed vector, for each node and associated elementary operation. To more robustly demonstrate this, we provided a schematic for an example function. Starting at the parent node for a given fucntion ($f(x)$), the value of the x will be evaluated for a given value and this output will be fed into the next node, where a new elementary operation will be performed and so forth. In order to store nodes for a given function in reverse mode, we anticipate that we will implement binary search trees however we can not offer more information at this point since we have yet to cover this material in class.

Colab paid products - Cancel contracts here