DeriveAlive Documentation

Chen Shi, Stephen Slater, Yue Sun

Feb 11, 2019

Contents

1 Introduction 1

2 How to Use `DeriveAlive` 3

3 Background 7

4 Software Organization 11

5 Implementation 15

6 Additional Features 19

7 Future 45

8 References 47

CHAPTER 1

Introduction

Differentiation, i.e. finding derivatives, has long been one of the key operations in computation related to modern science and engineering. In optimization and numerical differential equations, finding the extrema will require differentiation. There are many important applications of automatic differentiation in optimization, machine learning, and numerical methods (e.g., time integration, root-finding).

This documentation introduces `DeriveAlive`, a software library that uses the concept of automatic differentiation to solve differentiation problems in scientific computing. Additional features of a root finding suite, an optimization suite and a quadratic spline suite are also listed in this documentation.

CHAPTER 2

How to Use `DeriveAlive`

2.1 How to install

The url to the project is https://pypi.org/project/DeriveAlive/.

- Create a virtual environment and activate it

```
# If you don't have virtualenv, install it
sudo easy_install virtualenv
# Create virtual environment
virtualenv env
# Activate your virtual environment
source env/bin/activate
```

- Install `DeriveAlive` using pip. In the terminal, type:

```
pip install DeriveAlive
```

- Run module tests before beginning.

```
# Navigate to https://pypi.org/project/DeriveAlive/#files
# Download tar.gz folder, unzip, and enter the folder
pytest tests
```

2.2 Basic demo

```
python
>>> from DeriveAlive import DeriveAlive as da
>>> from DeriveAlive import rootfinding as rf
>>> from DeriveAlive import optimize as opt
```

(continues on next page)

(continued from previous page)

```
>>> from DeriveAlive import spline as sp
>>> import numpy as np
>>> import matplotlib.pyplot as plt
```

2.2.1 Declare Variables

- Denote constants

```
# None has to be typed, otherwise will be denoted as an R^1 variable
>>> a = da.Var([1], None)
>>> a
Var([1], None)
```

Note: Constant (scalar or vector): User must initialize derivative to 'None'. Otherwise, the variable will be denoted as an $\mathbb{R}^1$ variable with derivative [1].

- Denote scalar variables and functions

```
# The first way to denote a scalar varibale
>>> x = da.Var([1])
>>> x
Var([1], [1])

# The second way to denote a scalar variable
>>> x = da.Var([1], [1])
>>> x
Var([1], [1])

# Denote a scalar function
>>> f = 2 * x + np.sin(x)
>>> f
Var([2.84147098], [2.54030231])

# Define a callable scalar function:
>>> def f(x):
        return 2 * x + np.sin(x)
<function f at 0x116080950>
```

- Denote vector variables and functions

```
# Suppose we want to denote variables in R^3
>>> x = da.Var([1], [1, 0, 0])
>>> y = da.Var([2], [0, 1, 0])
>>> z = da.Var([3], [0, 0, 1])

# Alternatively, users can use the following notation to declare the same variables
# 'x,y': x denotes the length of the derivative, y denotes the position of the 1
>>> x = da.Var([1], '3,0')
>>> y = da.Var([2], '3,1')
>>> z = da.Var([3], '3,2')

# Suppose we want to denote an R^3 to R^1 function
f = x + y + z
```

(continues on next page)

(continued from previous page)

```
>>> f
Var([6], [1 1 1])

# Alternatively, the user can define the R^3 to R^1 function
# by explicitly defining a da.Var vector with one entry:

>>> g = da.Var([2 * x + x * y])
>>> g
Var([4], [4 1 0])

# Suppose we want to denote an R^3 to R^3 function
>>> f = da.Var([x, y ** 2, z ** 4])
>>> f
    Values:
    [ 1  4 81],
    Jacobian:
    [[  1   0   0]
     [  0   4   0]
     [  0   0 108]]

# Suppose we want to denote an R^1 to R^3 function
>>> x = da.Var([1])
>>> f = da.Var([x, np.sin(x), np.exp(x-1)])
>>> f
    Values:
    [1.         0.84147098 1.        ],
    Jacobian:
    [[1.        ]
     [0.54030231]
     [1.        ]]
```

2.2.2 Demo 1: $\mathbb{R}^1 \to \mathbb{R}^1$

Consider the case $f(x) = \sin(x) + 5\tan(x/2)$. We want to calculate the value and the first derivative of $f(x)$ at $x = \frac{\pi}{2}$.

```
# Expect value of 6.0, derivative of 5.0
>>> x = da.Var([np.pi/2])
>>> f = np.sin(x) + 5 * np.tan(x/2)
>>> print(f.val)
[6.]
>>> print(f.der)
[5.]
```

2.2.3 Demo 2: $\mathbb{R}^m \to \mathbb{R}^1$

Consider the case $f(x, y) = \sin(x) + \exp(y)$. We want to calculate the value and the jacobian of $f(x, y)$ at $x = \frac{\pi}{2}, y = 1$.

```
# Expect value of 3.71828183, jacobian of [0, 2.71828183]
>>> x = da.Var([np.pi/2], [1, 0])
>>> y = da.Var([1], [0, 1])
>>> f = np.sin(x) + np.exp(y)
```

(continues on next page)

(continued from previous page)

```
>>> print(f.val)
[3.71828183]
>>> print(f.der)
[0.         2.71828183]
```

2.2.4 Demo 3: $\mathbb{R}^1 \to \mathbb{R}^n$

Consider the case $f(x) = (\sin(x), x^2)$. We want to calculate the value and the Jacobian of $f(x)$ at $x = \frac{\pi}{2}$.

```
# Expect value of [1. 2.4674011], jacobian of [[0], [3.14159265]]
>>> x = da.Var([np.pi/2], [1])
>>> f = da.Var([np.sin(x), x ** 2])
>>> f
    Values:
    [1.        2.4674011],
    Jacobian:
    [[0.        ]
     [3.14159265]]
```

2.2.5 Demo 4: $\mathbb{R}^m \to \mathbb{R}^n$

Consider the case $f(x, y, z) = (\sin(x), 4y + z^3)$. We want to calculate the value and the jacobian of $f(x, y, z)$ at $x = \frac{\pi}{2}, y = 3, z = -2$.

```
# Expect value of [1, 4], jacobian of [[0 0 0], [0 4 12]]
>>> x = da.Var([np.pi/2], [1, 0, 0])
>>> y = da.Var([3], [0, 1, 0])
>>> z = da.Var([-2], [0, 0, 1])
>>> f = da.Var([np.sin(x), 4 * y + z ** 3])
>>> f
Values:
[1. 4.],
Jacobian:
[[ 0.  0.  0.]
 [ 0.  4. 12.]]
```

..Note:: Demos for additional features are listed in the corresponding additional features tab.

CHAPTER 3

Background

The chain rule, gradient (Jacobian), computational graph, elementary functions and several numerical methods serve as the mathematical cornerstone for this software. The mathematical concepts here come from CS 207 Lectures 9 and 10 on Autodifferentiation.

3.1 The Chain Rule

The chain rule is critical to AD, since the derivative of the function with respect to the input is dependent upon the derivative of each trace in the evaluation with respect to the input.
If we have $h(u(x))$ then the derivative of h with respect to x is:

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial u} \cdot \frac{\partial u}{\partial x}$$

If we have another argument $h(u, v)$ where u and v are both functions of x, then the derivative of $h(x)$ with respect to x is:

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial u} \cdot \frac{\partial u}{\partial x} + \frac{\partial h}{\partial v} \cdot \frac{\partial v}{\partial x}$$

3.2 Gradient and Jacobian

If we have $x \in \mathbb{R}^m$ and function $h\left(u\left(x\right), v\left(x\right)\right)$, we want to calculate the gradient of h with respect to x:

$$\nabla_x h = \frac{\partial h}{\partial u} \nabla_x u + \frac{\partial h}{\partial v} \nabla_x v$$

In the case where we have a function $h(x) : \mathbb{R}^m \to \mathbb{R}^n$, we write the Jacobian matrix as follows, allowing us to store the gradient of each output with respect to each input.

$$J = \begin{bmatrix} \frac{\partial h_1}{\partial x_1} & \frac{\partial h_1}{\partial x_2} & \cdots & \frac{\partial h_1}{\partial x_m} \\ \frac{\partial h_2}{\partial x_1} & \frac{\partial h_2}{\partial x_2} & \cdots & \frac{\partial h_2}{\partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial h_n}{\partial x_1} & \frac{\partial h_n}{\partial x_2} & \cdots & \frac{\partial h_n}{\partial x_m} \end{bmatrix}$$

In general, if we have a function $g\left(y\left(x\right)\right)$ where $y \in \mathbb{R}^n$ and $x \in \mathbb{R}^m$. Then g is a function of possibly n other functions, each of which can be a function of m variables. The gradient of g is now given by

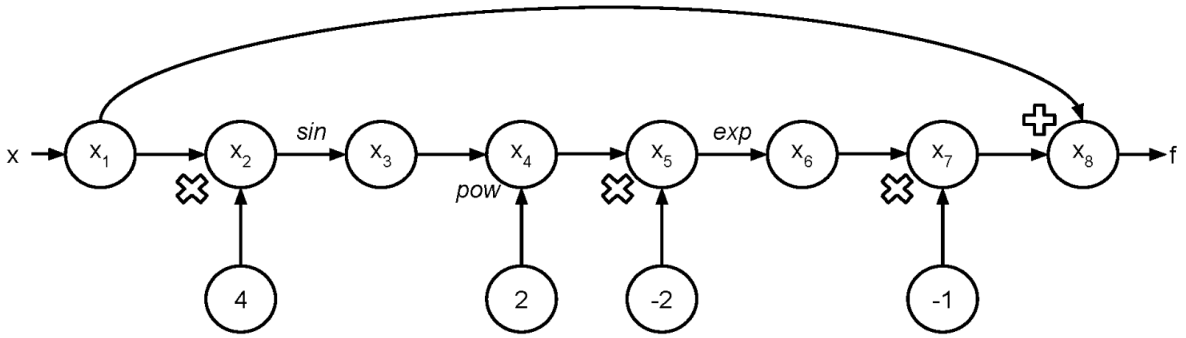
$$\nabla_x g = \sum_{i=1}^{n} \frac{\partial g}{\partial y_i} \nabla_x y_i \left(x\right).$$

3.3 The Computational Graph

Let us visualize what happens during the evaluation trace. The following example is based on Lectures 9 and 10. Consider the function:

$$f\left(x\right) = x - \exp\left(-2\sin^2\left(4x\right)\right)$$

If we want to evaluate f at the point x, we construct a graph where the input value is x and the output is y. Each input variable is a node, and each subsequent operation of the execution trace applies an operation to one or more previous nodes (and creates a node for constants when applicable).



As we execute $f(x)$ in the "forward mode", we can propagate not only the sequential evaluations of operations in the graph given previous nodes, but also the derivatives using the chain rule.

3.4 Elementary functions

An elementary function is built up of a finite combination of constant functions, field operations $(+, -, \times, \div)$, algebraic, exponential, trigonometric, hyperbolic and logarithmic functions and their inverses under repeated compositions. Below is a table of some elementary functions and examples that we will include in our implementation.

Elementary Functions	Example
powers	x^2
roots	$\sqrt{x}$
exponentials	e^x
logarithms	$\log(x)$
trigonometrics	$\sin(x)$
inverse trigonometrics	$\arcsin(x)$
hyperbolics	$\sinh(x)$

Note: Background for additional features, Newton's root finding, Gradient Descent, BFGS and quadratic splines can be found in Additional Features.

CHAPTER 4

Software Organization

4.1 Current directory structure

```
cs207-FinalProject/
|-- DeriveAlive/
|   |-- DeriveAlive.py
|   |-- __init__.py
|   |-- optimize.py
|   |-- rootfinding.py
|   `-- spline.py
|-- demos/
|   |-- Presentation.ipynb
|   `-- surprise.py
|-- documentation/
|   |-- docs/
|   |-- documentation.pdf
|   |-- milestone1.pdf
|   `-- milestone2.pdf
|-- tests/
|   |-- __init__.py
|   |-- test_DeriveAlive.py
|   |-- test_optimize.py
|   |-- test_rootfinding.py
|   `-- test_spline.py
|-- LICENSE
|-- __init__.py
|-- README.md
|-- requirements.txt
|-- setup.cfg
`-- setup.py
```

4.2 Basic modules and their functionality

- `DeriveAlive`: This module contains our custom library for autodifferentiation. It includes functionality for a `Var` class that contains values and derivatives, as well as class-specific methods for the operations that our model implements (e.g., tangent, sine, power, exponentiation, addition, multiplication, and so on).
- `optimize`: This module utilizes our custom library for autodifferentiation to perform optimization. It includes `DeriveAlive.Var` class-specific methods. Users can define a custom function to optimize, where this function is $\mathbb{R}^1 \to \mathbb{R}^1$ or $\mathbb{R}^m \to \mathbb{R}^1$. If the function is $\mathbb{R}^m \to \mathbb{R}^1$, it must take as input a list of m variables. Our suggestion is to extract the variables from this list on the first line of the user-defined function, and then use them individually. Furthermore, `optimize` allows for dataset compatability with regression optimization. A user can input a numpy matrix with m rows and n columns, where $n >= 2$ and $m >= 1$. The first $n - 1$ columns denote the features of the data, and the final column represents the labels. The user must specify the function to optimize as "mse". Then, the function will find a local minimum of the mean squared error objective function. Finally, the module allows for static and animated plots in 2D to 4D using `plot_results`.
- `rootfinding`: This module utilizes our custom library for autodifferentiation to find roots of a given $\mathbb{R}^1 \to \mathbb{R}^1$ or $\mathbb{R}^m \to \mathbb{R}^1$ function. It includes `DeriveAlive.Var` class-specific methods for Newton's method. It also allows the user to visualize static or animated results in 2D to 4D using `plot_results`.
- `spline`: This module utilizes our custom library for autodifferentiation to draw quadratic splines and return corresponding coefficients for quadratic functions of a given scalar function. It includes `DeriveAlive.Var` class-specific methods for quadratic spline generation.

4.3 Test Suite

All test files live in `tests/` folder.

- `test_DeriveAlive`: This is a test suite for `DeriveAlive`. It includes tests for scalar functions and vector functions to ensure that the `DeriveAlive` module properly calculates values of scalar functions and gradients with respect to scalar inputs, and vector functions and gradients with respect to vector inputs.
- `test_rootfinding`: This is a test suite for `rootfinding`.
- `test_optimize`: This is a test suite for `optimization`.
- `test_spline`: This is a test suite for `spline`.

We use Travis CI mfor automatic testing for each push, and Coveralls for line coverage metrics. We have already set up these integrations, with badges included in the `README.md`. Users may run the test suite by navigating to the `tests/` folder and running the command `pytest test_<module>.py` from the command line (or `pytest tests` if the user is outside the `tests/` folder).

4.4 Installation using PyPI and GitHub

We provide two ways for our package installation: PyPI and GitHub.

- Installation using PyPI

  We also utilized the Python Package Index (PyPI) for distributing our package. PyPI is the official third-party software repository for Python and primarily hosts Python packages in the form of archives called sdists (source distributions) or precompiled wheels. The url to the project is https://pypi.org/project/DeriveAlive/.

  - Create a virtual environment and activate it

```
# If you don't have virtualenv, install it
sudo easy_install virtualenv
# Create virtual environment
virtualenv env
# Activate your virtual environment
source env/bin/activate
```

  - Install DeriveAlive using pip. In the terminal, type:

```
pip install DeriveAlive
```

  - Run module tests before beginning.

```
# Navigate to https://pypi.org/project/DeriveAlive/#files
# Download tar.gz folder, unzip, and enter the folder
pytest tests
```

  - Use DeriveAlive Python package # (see demo in Section 2.2)

```
python
>>> from DeriveAlive import DeriveAlive as da
>>> import numpy as np
>>> x = da.Var([np.pi/2])
>>> x
Var([1.57079633], [1.])
...
>>> quit()

# deactivate virtual environment
deactivate
```

- Installation from GitHub
  - **Download the package from GitHub to your folder via these commands** in the terminal:

```
mkdir test_cs207
cd test_cs207/
git clone https://github.com/cs207-group19/cs207-FinalProject.git
cd cs207-FinalProject/
```

  - Create a virtual environment and activate it

```
# If you don't have virtualenv, install it
sudo easy_install virtualenv
# Create virtual environment
virtualenv env
# Activate your virtual environment
source env/bin/activate
```

  - Install required packages and run module tests in `tests/`

```
pip install -r requirements.txt
pytest tests
```

  - Use DeriveAlive Python package (see demo in Section 2.2)

```
python
>>> import DeriveAlive.DeriveAlive as da
>>> import numpy as np
>>> x = da.Var([np.pi/2])
>>> x
Var([1.57079633], [1.])
...
>>> quit()

# deactivate virtual environment
deactivate
```

CHAPTER 5

Implementation

5.1 Forward Mode Implementation

- Variable domain: The variables are defined as real numbers, hence any calculations or results involving complex numbers will be excluded from the package.
- Type of user input: Regardless of the input type (e.g., an int, a float or a list or a numpy array), the `Var` class will automatically convert the input into a numpy array.
- Core data structures: The core data structures will be classes, lists and numpy arrays.
  - Classes will help us provide an API for differentiation and custom functions, including custom methods for our elementary functions.
  - Numpy arrays are the main data structure during the calculation. We store the list of derivatives as a numpy array so that we can apply entire functions to the array, rather than to each entry separately. Each trace `Var` has a numpy array of derivatives where the length of the array is the number of input variables in the function. In the vector-vector case, if we have a function $f : \mathbb{R}^m \to \mathbb{R}^n$ or $f : \mathbb{R}^1 \to \mathbb{R}^n$, we can process this as $f = [f_1, f_2, \ldots, f_n]$, where each f_i is a function $f_i : \mathbb{R}^m \to \mathbb{R}$. Our implementation can act as a wrapper over these functions, and we can evaluate each f_i independently, so long as we define f_i in terms of the m inputs. Currently, the module supports both scalar to scalar, scalar to vector, vector to scalar and vector to vector functions.
- Our implementation plan includes 1 class which accounts for trace variables and derivatives with respect to each input variable.
  - `Var` class. The class instance itself has two main attributes: the value and the evaluated derivatives (Jacobian) with respect to each input. Within the class we redefine the elementary functions and basic algebraic functions, including both evaluation and derivation. Since our computation graph includes "trace" variables, this class will account for each variable.

5.2 API

5.2.1 DeriveAlive.DeriveAlive

- Class attributes and methods:
  - Attributes in Var: self.var, self.der. To cover vector-to-vector cases, we implement our self.var and self.der as numpy arrays, in order to account for derivatives with respect to each input variable. Also the constructor checks whether the values and derivatives are integers, floats, or lists, and transforms them into numpy arrays automatically.
  - We have overloaded elementary mathematical operations such as addition, subtraction, multiplication, division, sine, pow, log, etc. that take in 1 Var type, or 2 types, or 1 Var type and 1 constant, and return a new Var (i.e. the next "trace" variable). All other operations on constants will use the standard Python library. In each Var, we will store as attributes the value of the variable (which is calculated based on the current operation and previous trace variables) and the evaluated gradient of the variable with respect to each input variable.
  - Methods in Var:
    - __init__: initialize a Var class object, regardless of the user input, with values and derivatives stored as numpy arrays.
    - __repr__: overload the print format, prints self in the form of Var([val], [der]) when self is a scalar or constant; prints self in the form of Values([val]) Jacobian([der]) when self is a vector.
    - __add__: overload add function to handle addition of Var class objects and addition of Var and non-Var objects.
    - __radd__: preserve addition commutative property.
    - __sub__: overload subtraction function to handle subtraction of Var class objects and subtraction between Var and non-Var objects.
    - __rsub__: allow subtraction for $a -$ Var case where a is a float or an integer.
    - __mul__: overload multiplication function to handle multiplication of Var class objects and multiplication between Var and non-Var objects.
    - __rmul__: preserve multiplication commutative property.
    - __truediv__: overload division function to handle division of Var class objects over floats or integers.
    - __rtruediv__: allow division for $a \div$ Var case where a is a float or an integer.
    - __neg__: return negated Var.
    - __abs__: return the absolute value of Var.
    - __eq__: return True if two Var objects have the same value and derivative, False otherwise.
    - __ne__: return False if two Var objects have the same value and derivative, True otherwise.
    - __lt__: return True if the value of Var object is less than an integer / a float / the value of Var object, False otherwise.
    - __le__: return True if the value of Var object is less than or equal to an integer / a float / the value of Var object, False otherwise.
    - __gt__: return True if the value of Var object is greater than an integer / a float / the value of Var object, False otherwise.

    * __ge__: return True if the value of Var object is greater than or equal to an integer / a float / the value of Var object, False otherwise.
    * __pow__, __rpow__, pow: extend power functions to Var class objects.
    * sin, cos, tan: extend trigonometric functions to Var class objects.
    * arcsin, arccos, arctan: extend inverse trigonometric functions to Var class objects.
    * sinh, cosh, tanh: extend hyperbolic functions to Var class objects.
    * sqrt: return the square root of Var class objects.
    * log: extend logarithmic functions with custom base input to Var class objects.
    * exp: extend exponential functions to Var class objects.
    * logistic: return the logistic function value with input of Var objects.

- External dependencies:
  - NumPy - This provides an API for a large collection of high-level mathematical operations. In addition, it provides support for large, multi-dimensional arrays and matrices.
  - doctest - This module searches for pieces of text that look like interactive Python sessions (typically within the documentation of a function), and then executes those sessions to verify that they work exactly as shown.
  - pytest - This is an alternative, more Pythonic way of writing tests, making it easy to write small tests, yet scales to support complex functional testing. We plan to use this for a comprehensive test suite.
  - setuptools - This package allows us to create a package out of our project for easy distribution. See more information on packaging instructions here: https://packaging.python.org/tutorials/packaging-projects/.
  - Test suites: Travis CI, Coveralls
- Elementary functions
  - Our explanation of our elementary functions is included in the "Class attributes and methods" section above. For the elementary functions, we defined our own custom methods within the Var class so that we can calculate, for example, the $\sin(x)$ of a variable x using a package such as numpy, and also store the proper gradient $(\cos(x)dx)$ to propagate the gradients forward. For example, consider a scalar function where self.val contains the current evaluation trace and self.der is a numpy array of the derivative of the current trace with respect to the input. When we apply sin, we propagate as follows:

```
def sin(self):
            val = np.sin(self.val)
            der = np.cos(self.val) * self.der
            return Var(val, der)
```

    The structure of each elementary function is that it calculates the new value (based on the operation) and the new derivative, and then returns a new Var with the updated arguments.

5.2.2 DeriveAlive.rootfinding

Detailed methods with inputs and return information are listed in Additional Features - Root Finding.

- Methods:
  - NewtonRoot: return a root of a function $f : \mathbb{R}^m \Rightarrow \mathbb{R}^1$
  - plot_results: See docstring.

- External dependencies:
  - `DeriveAlive`
  - `NumPy`
  - `matplotlib.pyplot`
  - Test suites: Travis CI, Coveralls

5.2.3 `DeriveAlive.optimize`

Detailed methods with inputs and return information are listed in Additional Features - Optimization.

- Methods:
  - `GradientDescent`: solve for a local minimum of a function $f : \mathbb{R}^m \Rightarrow \mathbb{R}^1$. If f is a convex function, then the local minimum is a global minimum.

Note: Supports data set compatibility and mean squared error optimization.

- `BFGS`: sovle for a local stationary point, i.e. $\nabla f = 0$, of a function $f : \mathbb{R}^m \Rightarrow \mathbb{R}^1$.
- `plot_results`: See docstring.

- External dependencies:
  - `DeriveAlive`
  - `NumPy`
  - `matplotlib.pyplot`
  - Test suites: Travis CI, Coveralls

5.2.4 `DeriveAlive.spline`

Detailed methods with inputs and return information are listed in Additional Features - Quadratic Splines.

- Methods:
  - `quad_spline_coeff`: calculate the coefficients of quadratic splines.
  - `spline_points`: get the coordinates of points on the corresponding splines.
  - `quad_spline_plot`: plot the original function and the corresponding splines.
  - `spline_error`: calculate the average absolute error of the spline and the original function at one point.
- External dependencies:
  - `DeriveAlive`
  - `NumPy`
  - `matplotlib.pyplot`
  - Test suites: Travis CI, Coveralls

CHAPTER 6

Additional Features

6.1 Root finding

6.1.1 Background

Newton root finding starts from an initial guess for x_1 and converges to x such that $f(x) = 0$. The algorithm is iterative. At each step t, the algorithm finds a line (or plane, in higher dimensions) that is tangent to f at x_t. The new guess for x_{t+1} is where the tangent line crosses the x-axis. This generalizes to m dimensions.

- Algorithm (univariate case)

for t iterations or until step size < `tol`: $x_{t+1} \leftarrow x_t - \frac{f(x_t)}{f'(x_t)}$

- Algorithm (multivariate case)

for t iterations or until step size < `tol`: $\mathbf{x}_{t+1} \leftarrow \mathbf{x}_t - (J(f)(\mathbf{x}_t))^{-1} f(\mathbf{x}_t)$

In the multivariate case, $J(f)$ is the Jacobian of f. If $J(f)$ is non-square, we use the pseudoinverse.

Here is an example in the univariate case:

A common application of root finding is in Lagrangian optimization. For example, consider the Lagrangian $\mathcal{L}(\mathbf{b}, \lambda)$. One can solve for the weights $\mathbf{b}, \lambda$ such that $\frac{\partial \mathcal{L}}{\partial b_j} = \frac{\partial \mathcal{L}}{\partial \lambda} = 0$.

6.1.2 Implementation

- Methods
  - `NewtonRoot`: return a root of a function $f : \mathbb{R}^m \Rightarrow \mathbb{R}^1$
    - input:
      - `f`: function of interest, callable. If f is a scalar to scalar function, then define f as follows:

```
def f(x):
    # Use x in function
    return x ** 2 + np.exp(x)
```

If f is a function of multiple scalars (i.e. $\mathbb{R}^m \Rightarrow \mathbb{R}^1$), the arguments to f must be passed in as a list. In this case, define f as follows:

```
def f(variables):
    x, y, z = variables
    return x ** 2 + y ** 2 + z ** 2 + np.sin(x)
```

- `x`: int, float, or da.Var (univariate), or list of int, float, or da.Var objects (multivariate). Inital guess for a root of f. If f is a scalar to scalar function (i.e. $\mathbb{R}^1 \Rightarrow \mathbb{R}^1$), and the initial guess for the root is 1, then x = [da.Var(1)]. If f is a function of multiple scalars, with initial guess for the root as (1, 2, 3), then the user can define `x` as:

```
x = [1, 2, 3]
```

- `iters`: int, optional, default=2000. The maximum number of iterations to run the Newton root finding algorithm. The algorithm will run for min $(t, iters)$ iterations, where t is the number of steps until `tol` is satisfied.
- `tol`: int or float, optional, default=1e-10. If the size of the update step (L2 norm in the case of $\mathbb{R}^m \Rightarrow \mathbb{R}^1$) is smaller than `tol`, then the algorithm will add that step and then terminate, even if the number of iterations has not reached `iters`.

* return:

- `root`: da.Var $\in \mathbb{R}^m$. The *val* attribute contains a numpy array of the root that the algorithm found in $min(iters, t)$ iterations ($iters, t$ defined above). The *der* attribute contains the Jacobian value at the specified root.
- `var_path`: a numpy array ($\mathbb{R}^{n \times m}$), where $n = min(iters, t)$ is the number of steps of the algorithm and m if the dimension of the root, where rows of the array are steps taken in consecutive order.
- `g_path`: a numpy array ($\mathbb{R}^{n \times 1}$), containing the consecutive steps of the output of f at each guess in `var_path`.

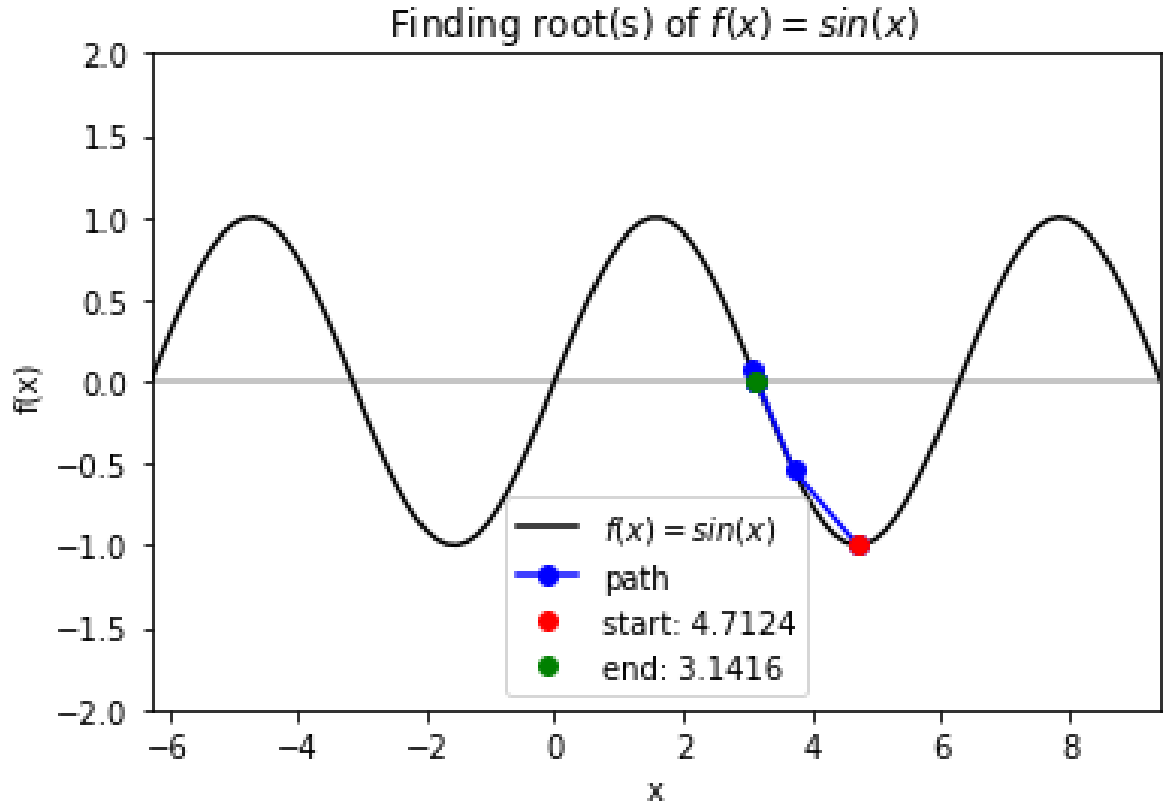
- External dependencies
  - `DeriveAlive`
  - `NumPy`
  - `matplotlib.pyplot`

6.1.3 Demo

```
>>> from DeriveAlive import rootfinding as rf
```

Case 1: $f = sin(x)$ with starting point $x_0 = \frac{3\pi}{2}$. Note: Newton method is not guaranteed to converge when $f\prime(x_0) = 0$. In our case, if the current guess has derivative of 0, we randomly set the derivative to be ± 1 and move in that directino to avoid getting stuck and avoid calculating an update step that has an extreme magnitude (which would occur if the derivative is very close to 0).

```
# define f function
>>> f_string = 'f(x) = sin(x)'

>>> def f(x):
        return np.sin(x)

>>> # Start at 3*pi/2
>>> x0 = 3 * np.pi / 2

    # finding the root
>>> for val in [np.pi - 0.25, np.pi, 1.5 * np.pi, 2 * np.pi - 0.25, 2 * np.pi + 0.25]:
        solution, x_path, y_path = rf.NewtonRoot(f, x0)

    # visualize the trace
>>> x_lims = -2 * np.pi, 3 * np.pi
>>> y_lims = -2, 2
>>> rf.plot_results(f, x_path, y_path, f_string, x_lims, y_lims)
```



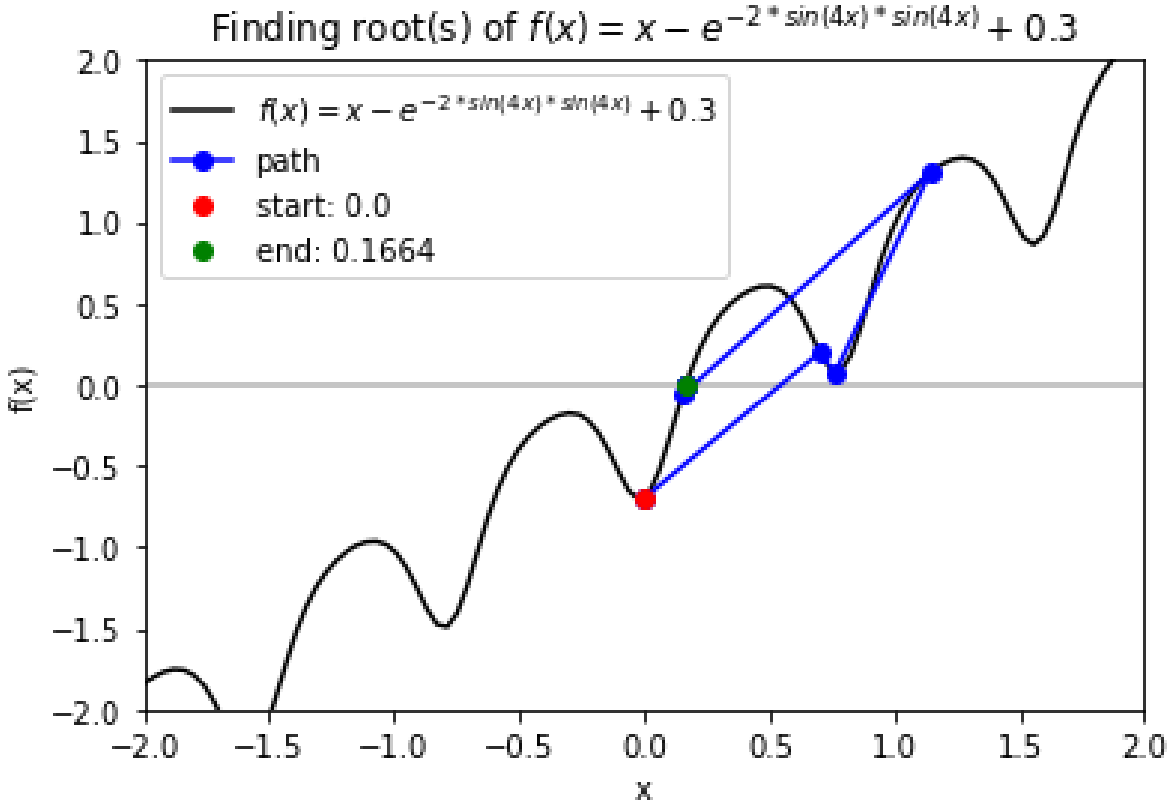
Case 2: $f = x - \exp(-2\sin(4x)sin(4x) + 0.3$ with starting point $x_0 = 0$.

```
# define f function
f_string = 'f(x) = x - e^{-2 * sin(4x) * sin(4x)} + 0.3'

>>> def f(x):
        return x - np.exp(-2.0 * np.sin(4.0 * x) * np.sin(4.0 * x)) + 0.3

# start at 0
>>> x0 = 0
```

(continues on next page)

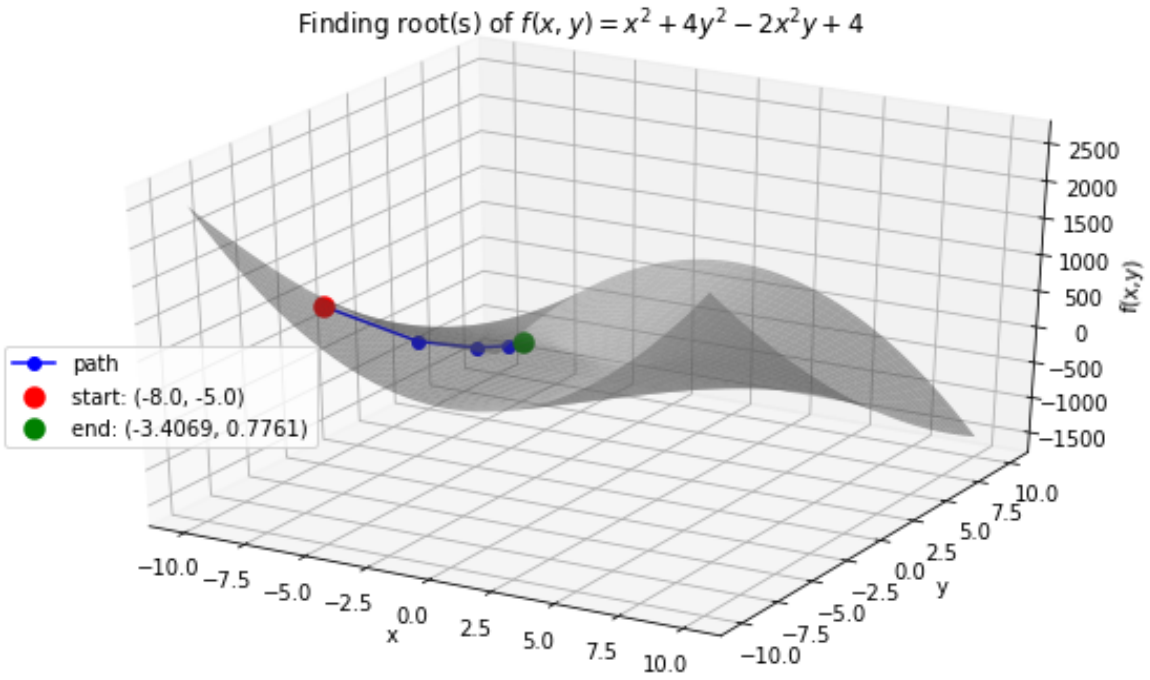
(continued from previous page)

```
# finding the root
>>> for val in np.arange(-0.75, 0.8, 0.25):
        solution, x_path, y_path = rf.NewtonRoot(f, x0)

# visualize the trace
>>> x_lims = -2, 2
>>> y_lims = -2, 2
>>> rf.plot_results(f, x_path, y_path, f_string, x_lims, y_lims)
```



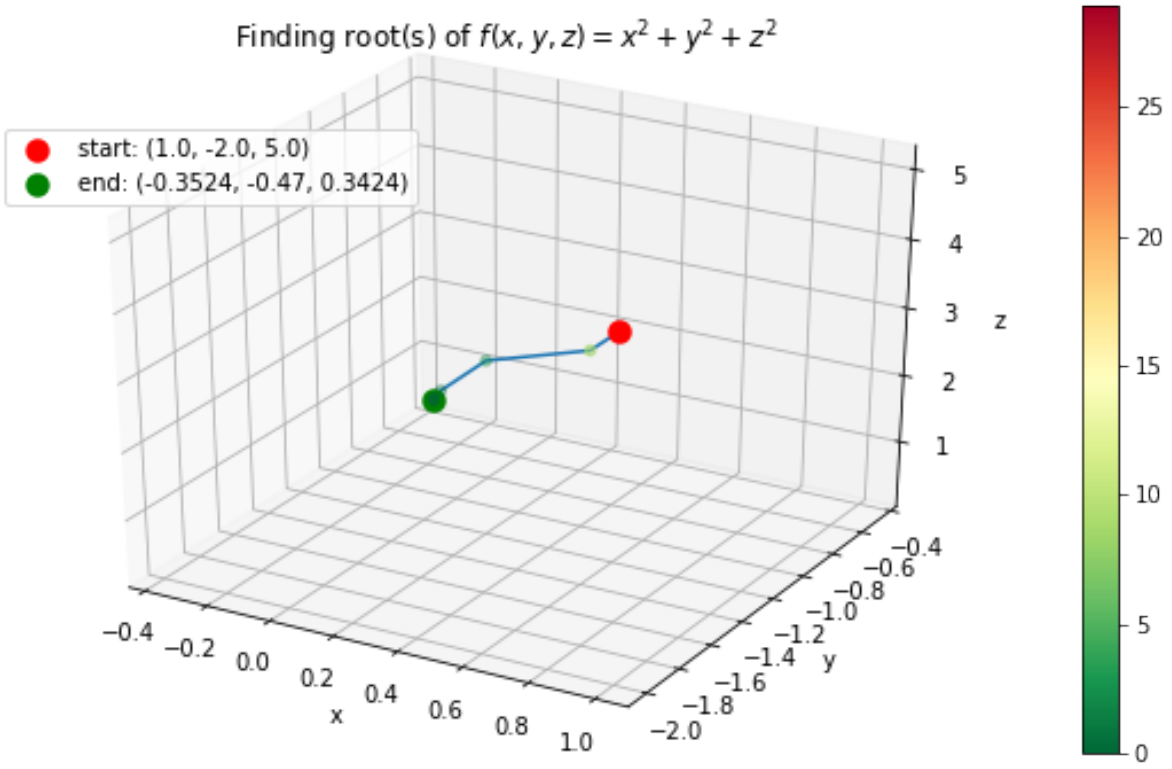
Case 3: $f(x, y) = x^2 + 4y^2 - 2x^2y + 4$ with starting points $x_0 = -8.0, y_0 = -5.0$.

```
# define f function
>>> f_string = 'f(x, y) = x^2 + 4y^2 -2x^2y + 4'

>>> def f(variables):
        x, y = variables
        return x ** 2 + 4 * y ** 2 - 2 * (x ** 2) * y + 4

# start at x0=8.0,y0= 5
>>> x0 = -8.0
>>> y0 = -5.0
>>> init_vars = [x0, y0]

# finding the root and visualize the trace
>>> solution, xy_path, f_path = rf.NewtonRoot(f, init_vars)
>>> rf.plot_results(f, xy_path, f_path, f_string, threedim=True)
```



Case 4: $f(x, y, z) = x^2 + y^2 + z^2$ with starting points $x_0 = 1, y_0 = -2, z_0 = 5$.

```
# define f function
>>> f_string = 'f(x, y, z) = x^2 + y^2 + z^2'

>>> def f(variables):
        x, y, z = variables
        return x ** 2 + y ** 2 + z ** 2 + np.sin(x) + np.sin(y) + np.sin(z)

# start at
>>> x0= 1
>>> y0= -2
>>> z0= 5
>>> init_vars = [x0, y0, z0]

# finding the root and visualize the trace
>>> solution, xyz_path, f_path = rf.NewtonRoot(f, init_vars)
>>> m = len(solution.val)
>>> rf.plot_results(f, xyz_path, f_path, f_string, fourdim=True)
```

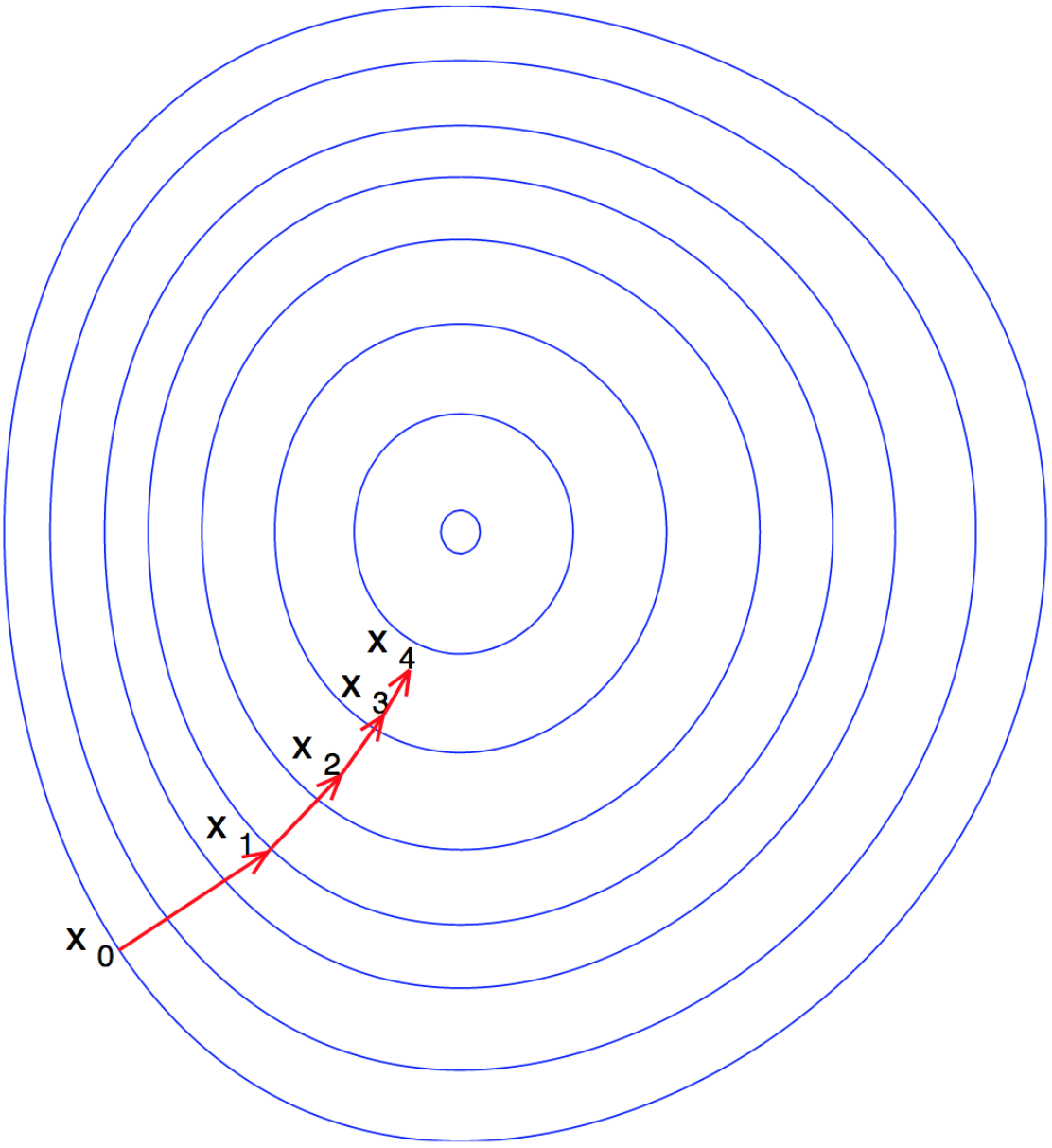


6.2 Optimization

6.2.1 Background

Gradient Descent is used to find the local minimum of a function f by taking locally optimum steps in the direction of steepest descent. A common application is in machine learning when a user desires to find optimal weights to minimize a loss function.

Here is a visualization of Gradient Descent on a convex function of 2 variables:



BFGS, short for "Broyden–Fletcher–Goldfarb–Shanno algorithm", seeks a stationary point of a function, i.e. where the gradient is zero. In quasi-Newton methods, the Hessian matrix of second derivatives is not computed. Instead, the Hessian matrix is approximated using updates specified by gradient evaluations (or approximate gradient evaluations).

Here is a pseudocode of the implementation of BFGS.

Actual implementation of BFGS: store and update inverse Hessian to avoid solving linear system:

1: choose initial guess x_0
2: choose H_0, initial inverse Hessian guess, *e.g.* $H_0 = \mathrm{I}$
3: **for** $k = 0, 1, 2, \ldots$ **do**
4: calculate $s_k = -H_k \nabla f(x_k)$
5: $x_{k+1} = x_k + s_k$
6: $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$
7: $H_{k+1} = \Delta H_k$
8: **end for**

where

$$\Delta H_k \equiv (I - s_k \rho_k y_k^T) H_k (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T, \qquad \rho_k = \frac{1}{y_k^T s_k}$$

BFGS pseudocode from AM205 Lecture 19

6.2.2 Implementation

- Methods
  - `GradientDescent`: solve for a local minimum of a function $f : \mathbb{R}^m \Rightarrow \mathbb{R}^1$. If f is a convex function, then the local minimum is a global minimum.
    - input:
      - `f`: function of interest, callable. In machine learning applications, this should be the cost function. For example, if solving for optimal weights to minimize a cost function f, then f can be defined as $\frac{1}{2m}$ times the sum of m squared residuals.

        If f is a scalar to scalar function, then define f as follows:

```
def f(x):
    # Use x in function
    return x ** 2 + np.exp(x)
```

        If f is a function of multiple scalars (i.e. $\mathbb{R}^m \Rightarrow \mathbb{R}^1$), the arguments to f must be passed in as a list. In this case, define f as follows:

```
def def f(variables):
    x, y, z = variables
    return x ** 2 + y ** 2 + z ** 2 + np.sin(x)
```

      - `x`: int, float, or da.Var (univariate), or list of int, float, or da.Var objects (multivariate). Initial guess for a root of f. If f is a scalar to scalar function (i.e. $\mathbb{R}^1 \Rightarrow \mathbb{R}^1$), and the initial guess for

the root is 1, then a valid x is x = 1. If f is a function of multiple scalars, with initial guess for the root as (1, 2, 3), then a valid definition of `x` is as follows:

- `iters`: int, optional, default=2000. The maximum number of iterations to run the Newton root finding algorithm. The algorithm will run for min $(t, iters)$ iterations, where t is the number of steps until `tol` is satisfied.
- `tol`: int or float, optional, default=1e-10. If the size of the update step (L2 norm in the case of $\mathbb{R}^m \Rightarrow \mathbb{R}^1$) is smaller than `tol`, then the algorithm will add that step and then terminate, even if the number of iterations has not reached `iters`.

* return:

- `minimum`: da.Var $\in \mathbb{R}^m$. The *val* attribute contains a numpy array of the minimum that the algorithm found in $min(iters, t)$ iterations ($iters, t$ defined above). The *der* attribute contains the Jacobian value at the specified root.
- `var_path`: a numpy array ($\mathbb{R}^{n \times m}$), where $n = min(iters, t)$ is the number of steps of the algorithm and m if the dimension of the minimum, where rows of the array are steps taken in consecutive order.
- `g_path`: a numpy array ($\mathbb{R}^{n \times 1}$), containing the consecutive steps of the output of f at each guess in `var_path`.

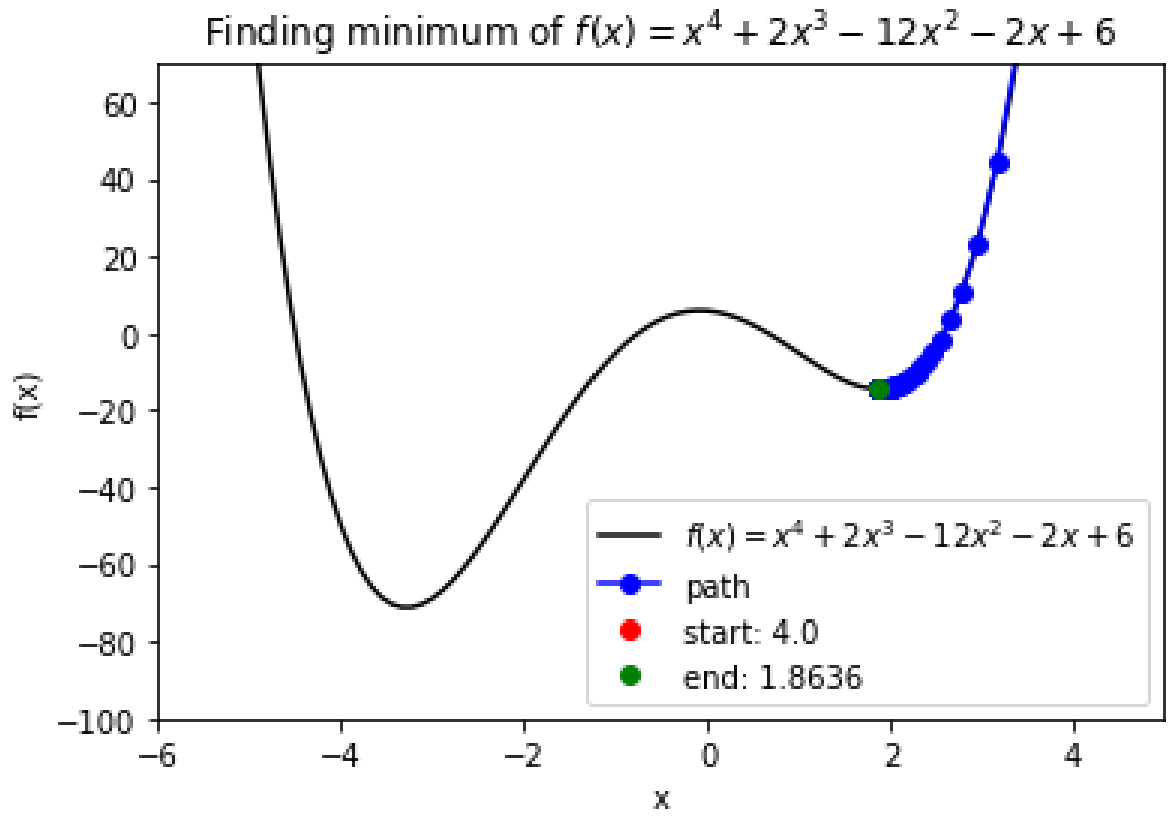
- External dependencies
  - `DeriveAlive`
  - `NumPy`
  - `matplotlib.pyplot`

6.2.3 Demo

```
>>> import DeriveAlive.optimize as opt
>>> import numpy as np
>>> import matplotlib.pyplot as plt
```

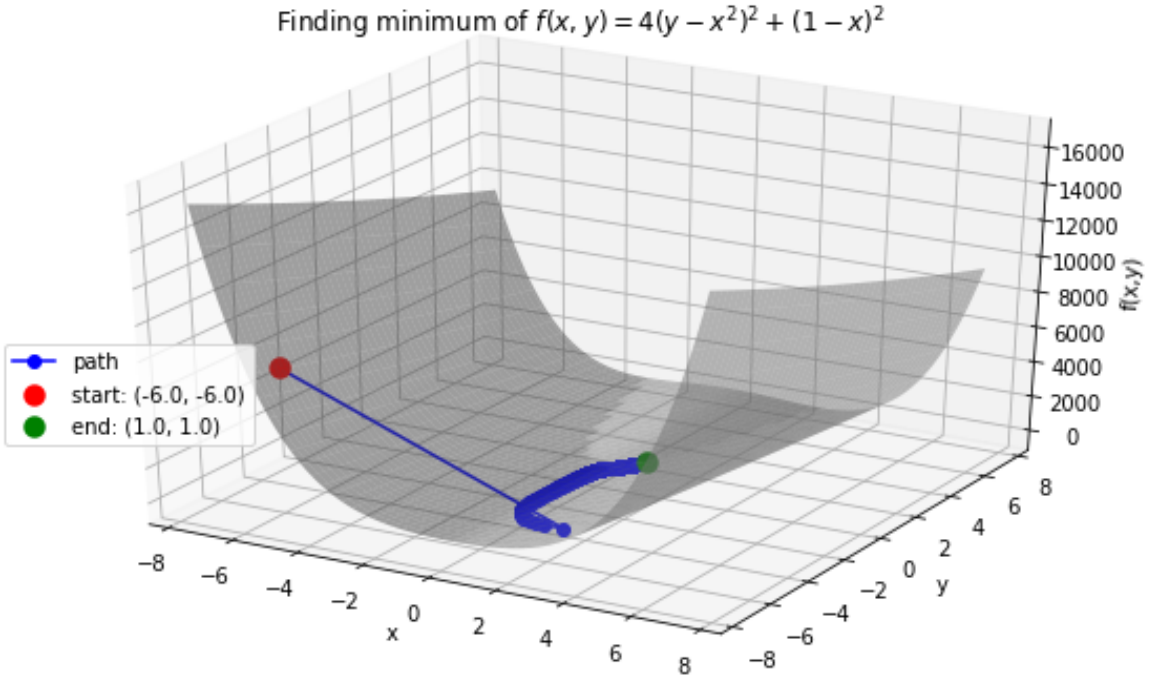
Case 1: Minimize quartic function $f(x) = x^4$. Get stuck in local minimum.

```
>>> def f(x):
        return x ** 4 + 2 * (x ** 3) - 12 * (x ** 2) - 2 * x + 6

    # Function string to include in plot
>>> f_string = 'f(x) = x^4 + 2x^3 -12x^2 -2x + 6'

>>> x0 = 4
>>> solution, xy_path, f_path = opt.GradientDescent(f, x0, iters=1000, eta=0.002)
>>> opt.plot_results(f, xy_path, f_path, f_string, x_lims=(-6, 5), y_lims=(-100, 70))
```

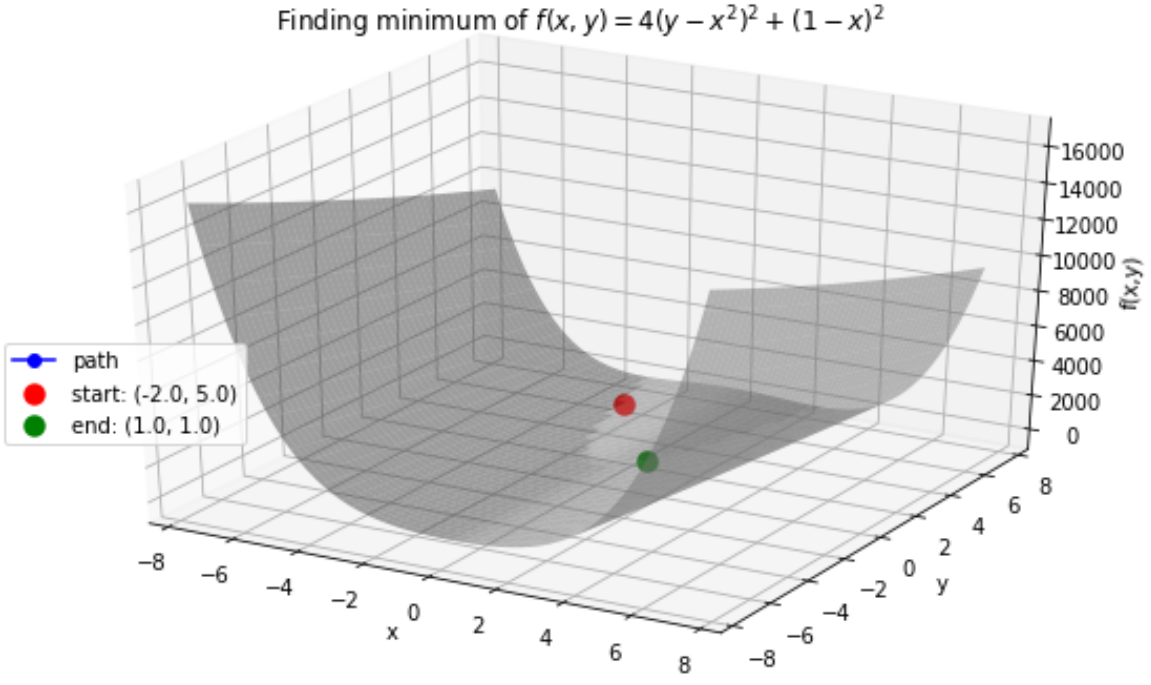


Case 2: Minimize Rosenbrock's function $f(x, y) = 4(y - x^2)^2 + (1 - x)^2$. Global minimum: 0 at $(x, y) = (1, 1)$.

```
# Rosenbrock function with leading coefficient of 4
>>> def f(variables):
        x, y = variables
        return 4 * (y - (x ** 2)) ** 2 + (1 - x) ** 2

# Function string to include in plot
>>> f_string = 'f(x, y) = 4(y - x^2)^2 + (1 - x)^2'

>>> x_val, y_val = -6, -6
>>> init_vars = [x_val, y_val]
>>> solution, xy_path, f_path = opt.GradientDescent(f, init_vars, iters=25000, eta=0.
↪002)
>>> opt.plot_results(f, xy_path, f_path, f_string, x_lims=(-7.5, 7.5), threedim=True)
```

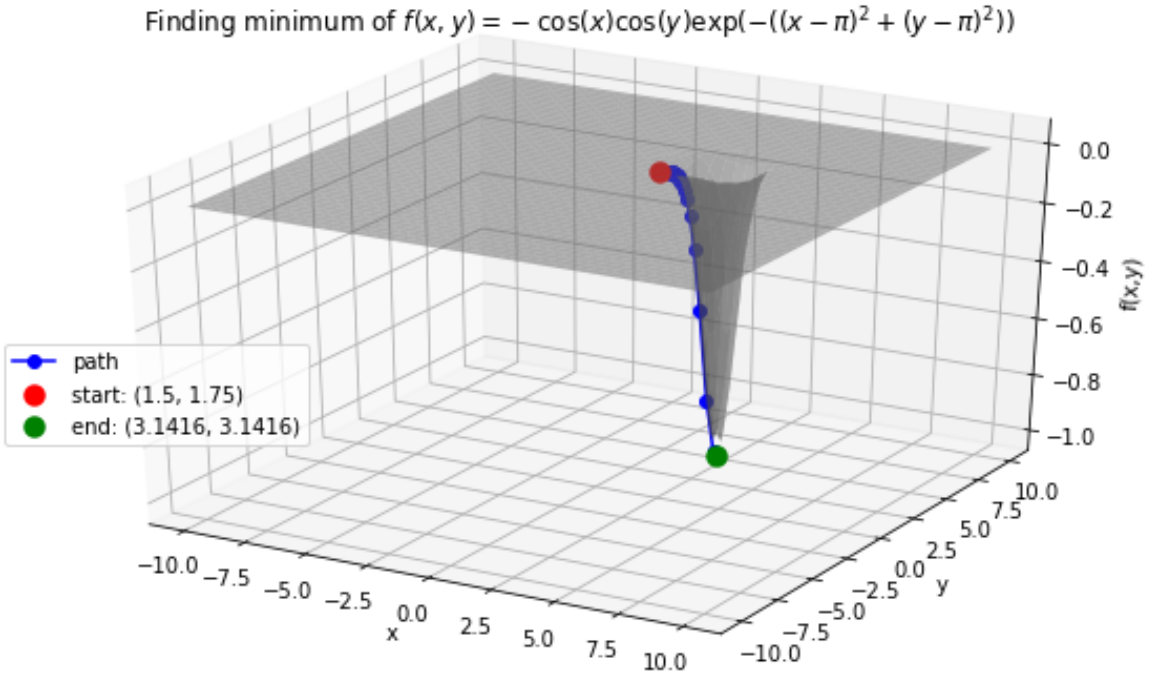



```
>>> x_val, y_val = -2, 5
>>> init_vars = [x_val, y_val]
>>> solution, xy_path, f_path = opt.GradientDescent(f, init_vars, iters=25000, eta=0.
↪002)
>>> opt.plot_results(f, xy_path, f_path, f_string, x_lims=(-7.5, 7.5), threedim=True)
```



Case 3: Minimize Easom's function: $f(x, y) = -\cos(x)\cos(y)\exp(-((x-\pi)^2 + (y-\pi)^2))$. Global minimum: -1 at $(x, y) = (\pi, \pi)$.

```
# Easom's function
>>> def f(variables):
        x, y = variables
        return -np.cos(x) * np.cos(y) * np.exp(-((x - np.pi) ** 2 + (y - np.pi) ** 2))

# Function string to include in plot
>>> f_string = 'f(x, y) = -\cos(x)\cos(y)\exp(-((x-\pi)^2 + (y-\pi)^2))'

# Initial guess
>>> x0 = 1.5
>>> y0 = 1.75
>>> init_vars = [x0, y0]

# Visulaize gradient descent
solution, xy_path, f_path = opt.GradientDescent(f, init_vars, iters=10000, eta=0.3)
opt.plot_results(f, xy_path, f_path, f_string, threedim=True)
```



Case 4: Machine Learning application: minimize mean squared error in regression

$$\hat{y}_i = \mathbf{w}^\top \mathbf{x}_i \tag{6.1}$$

$$MSE(X, y) = \frac{1}{m} \sum_{i=1}^{m} (\mathbf{w}^\top \mathbf{x}_i - y_i)^2 \tag{6.2}$$

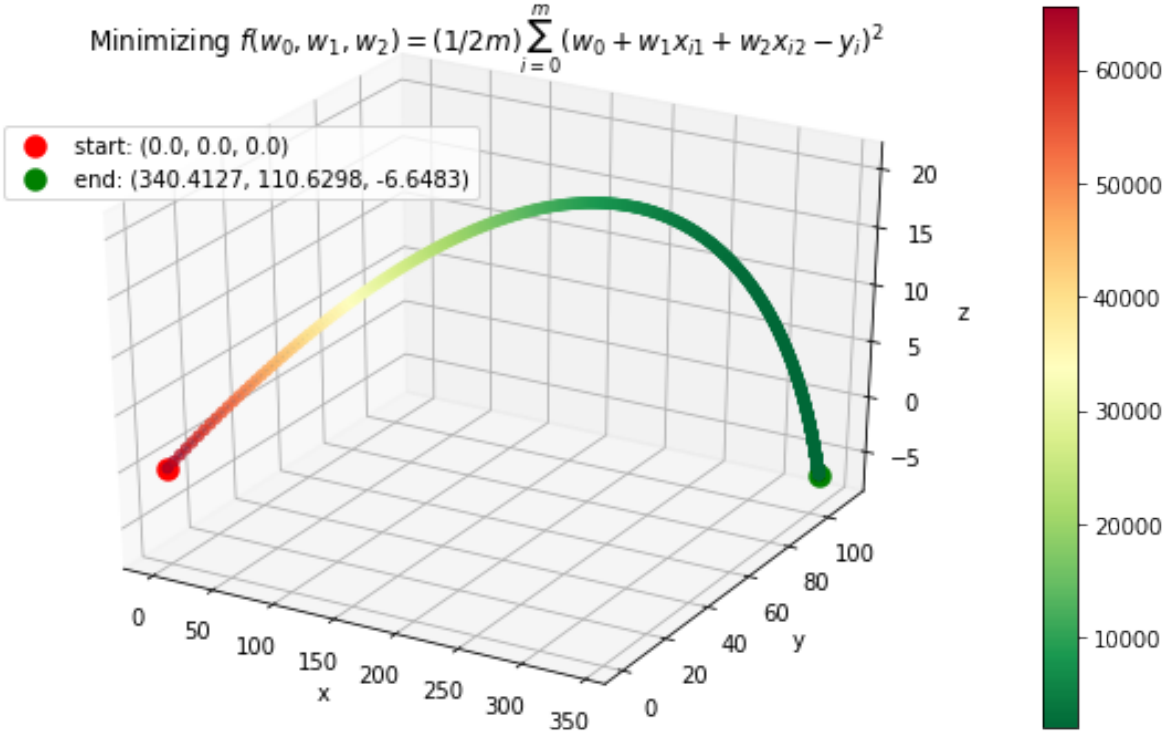
where $\mathbf{w}$ contains an extra dimension to fit the intercept of the features. - Example dataset (standardized): 47 homes from Portland, Oregon. Features: area (square feet), number of bedrooms. Output: price (in thousands of dollars).

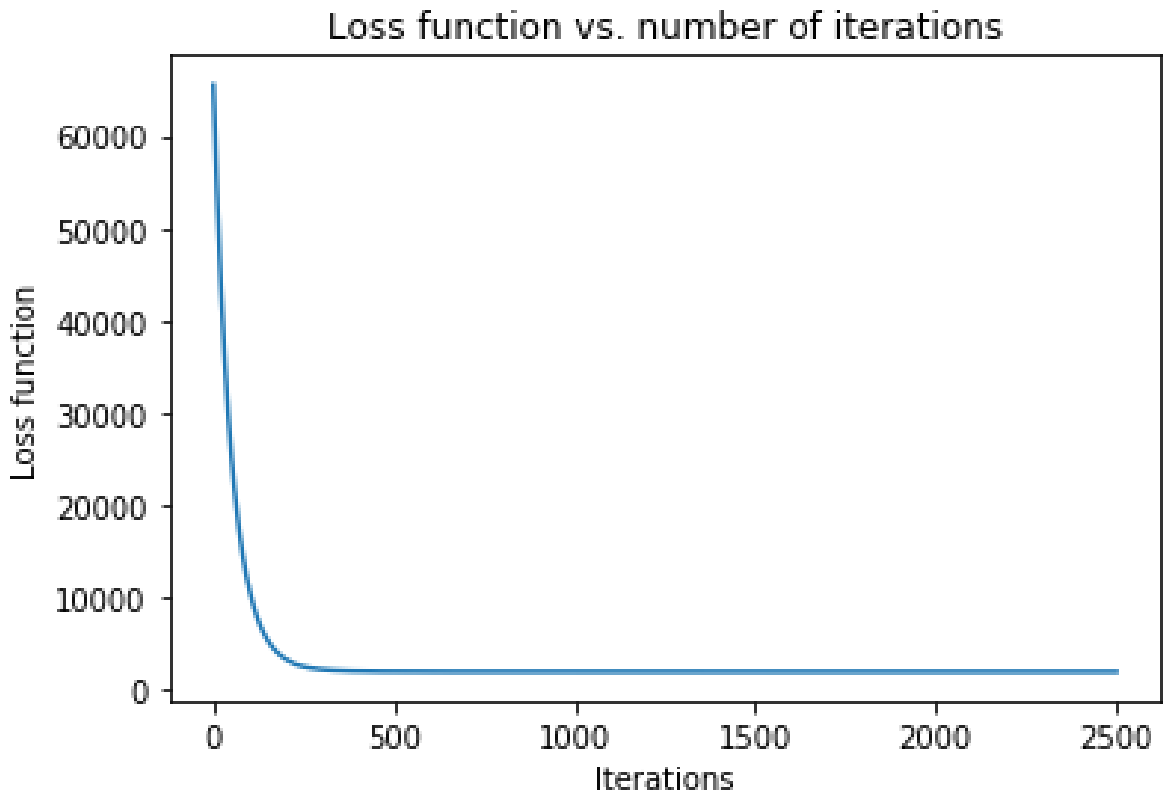
```
>>> f = "mse"
>>> init_vars = [0, 0, 0]

# Function string to include in plot
>>> f_string = 'f(w_0, w_1, w_2) = (1/2m)\sum_{i=0}^m (w_0 + w_1x_{i1} + w_2x_{i2} -␣
↪y_i)^2'
```

(continues on next page)

(continued from previous page)

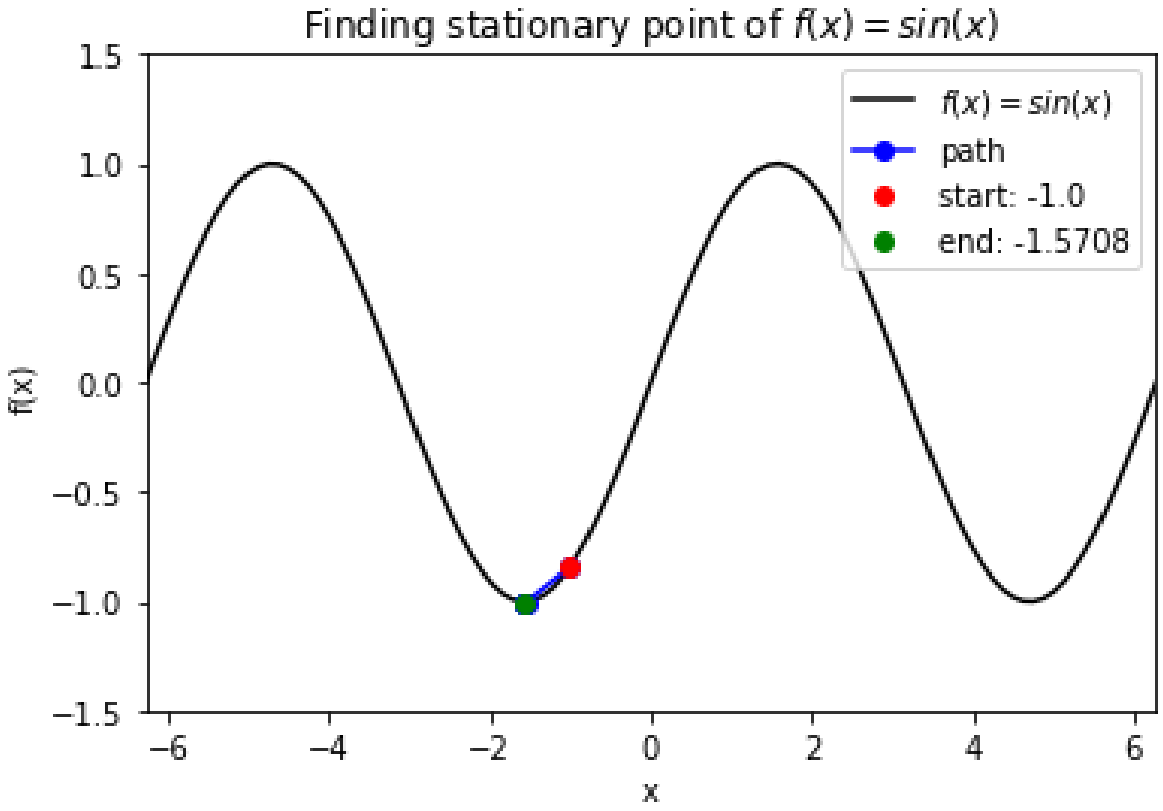
```
# Visulaize gradient descent
>>> solution, w_path, f_path, f = opt.GradientDescent(f, init_vars, iters=2500,␣
↪data=data)
>>> print ("Gradient descent optimized weights:\n{}".format(solution.val))
>>> opt.plot_results(f, w_path, f_path, f_string, x_lims=(-7.5, 7.5), fourdim=True)
Gradient descent optimized weights:
[340.41265957 110.62984204  -6.64826603]
```





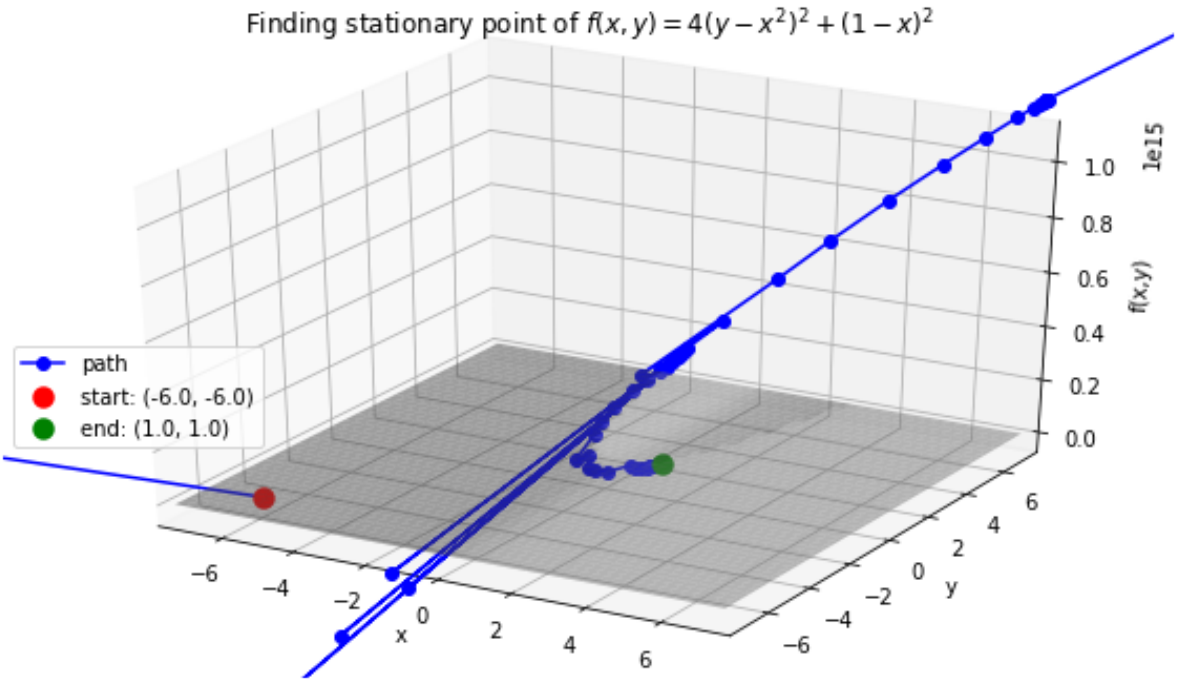
Case 5: Find stationary point of $f(x) = \sin(x)$. Note: BFGS finds stationary point, which can be maximum, not minimum.

```
>>> def f(x):
        return np.sin(x)

>>> f_string = 'f(x) = sin(x)'

>>> x0 = -1
>>> solution, x_path, f_path = opt.BFGS(f, x0)
>>> anim = opt.plot_results(f, x_path, f_path, f_string, x_lims=(-2 * np.pi, 2 * np.
↪pi), y_lims=(-1.5, 1.5), bfgs=True)
```



Case 6: Find stationary point of Rosenbrock function: $f(x, y) = 4(y - x^2)^2 + (1 - x)^2$. Stationary point: 0 at $(x, y) = (1, 1)$.

```
>>> def f(variables):
        x, y = variables
        return 4 * (y - (x ** 2)) ** 2 + (1 - x) ** 2

>>> f_string = 'f(x, y) = 4(y - x^2)^2 + (1 - x)^2'

>>> x0, y0 = -6, -6
>>> init_vars = [x0, y0]
>>> solution, xy_path, f_path = opt.BFGS(f, init_vars, iters=25000)
>>> xn, yn = solution.val
>>> anim = opt.plot_results(f, xy_path, f_path, f_string, x_lims=(-7.5, 7.5), y_
↪lims=(-7.5, 7.5), threedim=True, bfgs=True)
```



6.3 Quadratic Splines

6.3.1 Background

The `DeriveAlive` package can be used to calculate quadratic splines since it automatically returns the first derivative of a function at a given point.

We aim to construct a piecewise quadratic spline $s(x)$ using N equally-sized intervals over an interval for $f(x)$. Define $h = 1/N$, and let $s_k(x)$ be the spline over the range $[kh, (k+1)h]$ for $k = 0, 1, \ldots, N-1$. Each $s_k(x) = a_k x^2 + b_k x + c_k$ is a quadratic, and hence the spline has $3N$ degrees of freedom in total.

Example: $f(x) = 10^x, x \in [0, 1]$, with $N = 10$ intervals, the spline coefficients satisfy the following constraints:

- Each $s_k(x)$ should match the function values at both of its endpoints, so that $s_k(kh) = f(kh)$ and $s_k((k+1)h) = f((k+1)h)$. (Provides $2N$ constraints.)
- At each interior boundary, the spline should be differentiable, so that $s_{k-1}(kh) = s_k(kh)$ for $k = 1, \ldots, N-1$. (Provides $N - 1$ constraints.)
- Since $f'(x+1) = 10f'(x)$, let $s'_{N-1}(1) = 10s'_0(0)$. (Provides 1 constraint.)

Since there are $3N$ constraints for $3N$ degrees of freedom, there is a unique solution.

6.3.2 Implementation

- Methods

- `quad_spline_coeff`: calculate the coefficients of quadratic splines
  - input:
    - `f`: function of interest
    - `xMin`: left endpoint of the x interval
    - `xMax`: right endpoint of the x interval
    - `nIntervals`: number of intervals that you want to slice the original function
  - return:
    - `y`: the right hand side of $Ax = y$
    - `A`: the sqaure matrix in the left hand side of $Ax = y$
    - `coeffs`: coefficients of a_i, b_i, c_i
    - `ks`: points of interest in the x interval as `DeriveAlive` objects
- `spline_points`: get the coordinates of points on the corresponding splines
  - input:
    - `f`: function of interest
    - `coeffs`: coefficients of a_i, b_i, c_i
    - `ks`: points of interest in the x interval as `DeriveAlive` objects
    - `nSplinePoints`: number of points to draw each spline
  - return:
    - `spline_points`: a list of spline points (x, y) on each s_i
- `quad_spline_plot`: plot the original function and the corresponding splines
  - input:
    - `f`: function of interest
    - `coeffs`: coefficients of a_i, b_i, c_i
    - `ks`: points of interest in the x interval as `DeriveAlive` objects
    - `nSplinePoints`: number of points to draw each spline
  - return:
    - `fig`: the plot of $f(x)$ and splines
- `spline_error`: calculate the average absolute error of the spline and the original function at one point
  - input:
    - `f`: function of interest
    - `spline_points`: a list of spline points (x, y) on each s_i
  - return:
    - `error`: average absolute error of the spline and the original function on one given interval

- External dependencies
  - `DeriveAlive`
  - `NumPy`

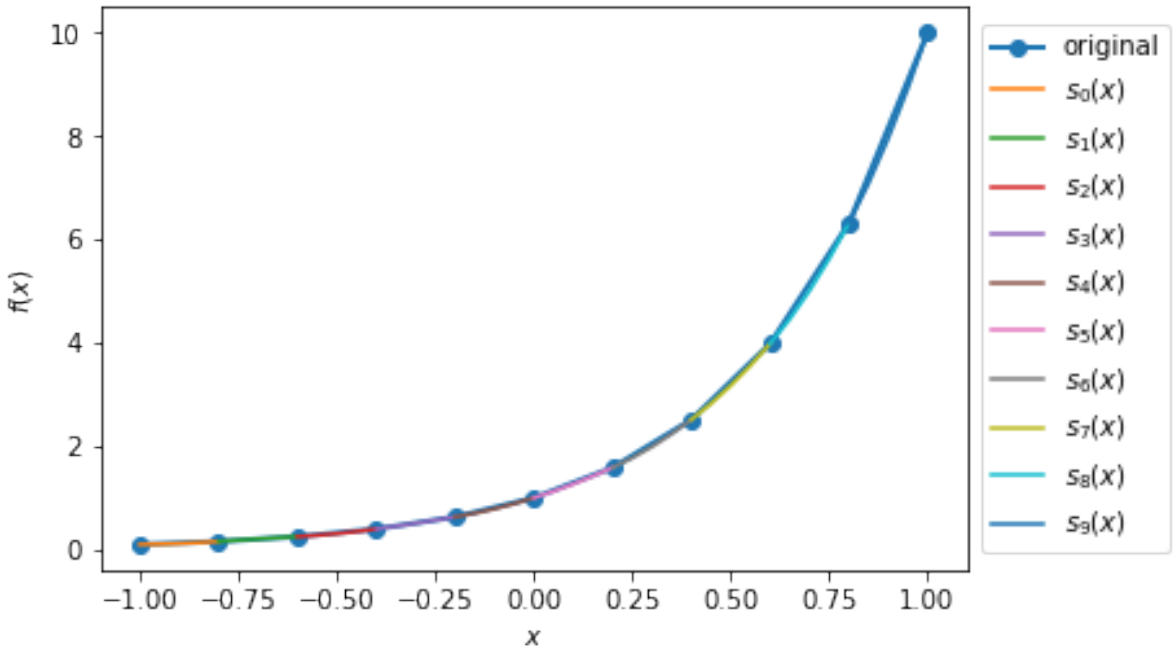
- matplotlib.pyplot

6.3.3 Demo

```
>>> import DeriveAlive.spline as sp
>>> import numpy as np
>>> import matplotlib.pyplot as plt
```

Case 1: Plot the quadratic spline of $f_1(x) = 10^x, x \in [-1, 1]$ with 10 intervals.

```
>>> def f1(var):
        return 10**var

>>> xMin1 = -1
>>> xMax1 = 1
>>> nIntervals1 = 10
>>> nSplinePoints1 = 5

>>> y1, A1, coeffs1, ks1 = sp.quad_spline_coeff(f1, xMin1, xMax1, nIntervals1)
>>> fig1 = sp.quad_spline_plot(f1, coeffs1, ks1, nSplinePoints1)
>>> spline_points1 = sp.spline_points(f1, coeffs1, ks1, nSplinePoints1)
>>> sp.spline_error(f1, spline_points1)
0.0038642295476342416

>>> fig1
```



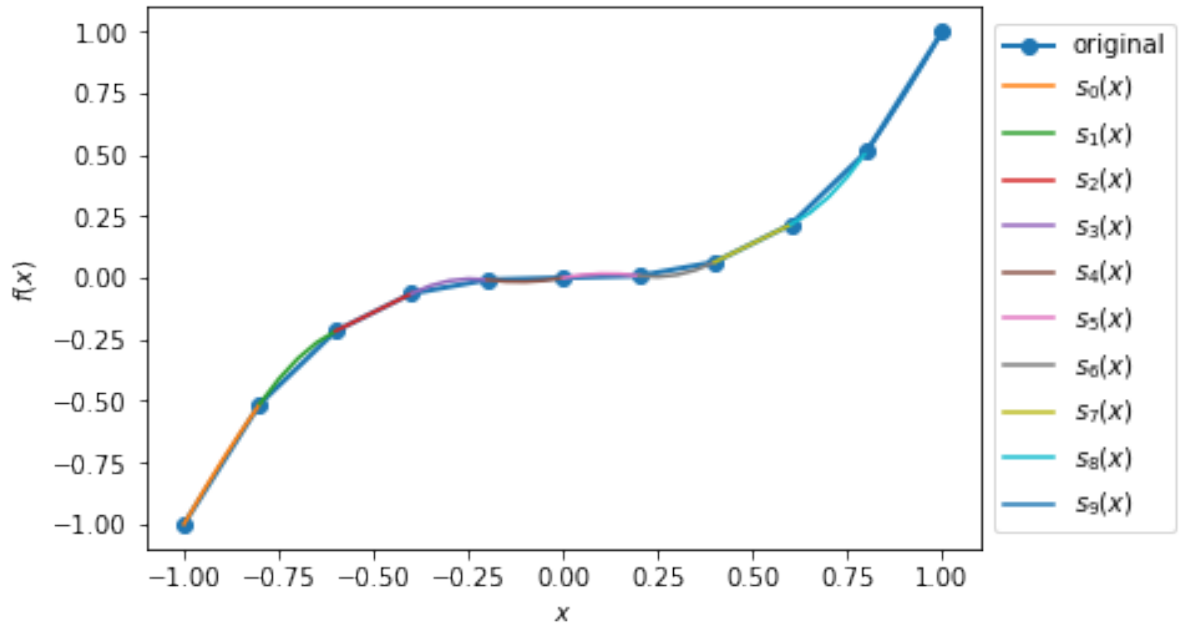
Case 2: Plot the quadratic spline of $f_2(x) = x^3, x \in [-1, 1]$ with 10 intervals.

```
>>> def f2(var):
        return var**3

>>> xMin2 = -1
```

(continues on next page)

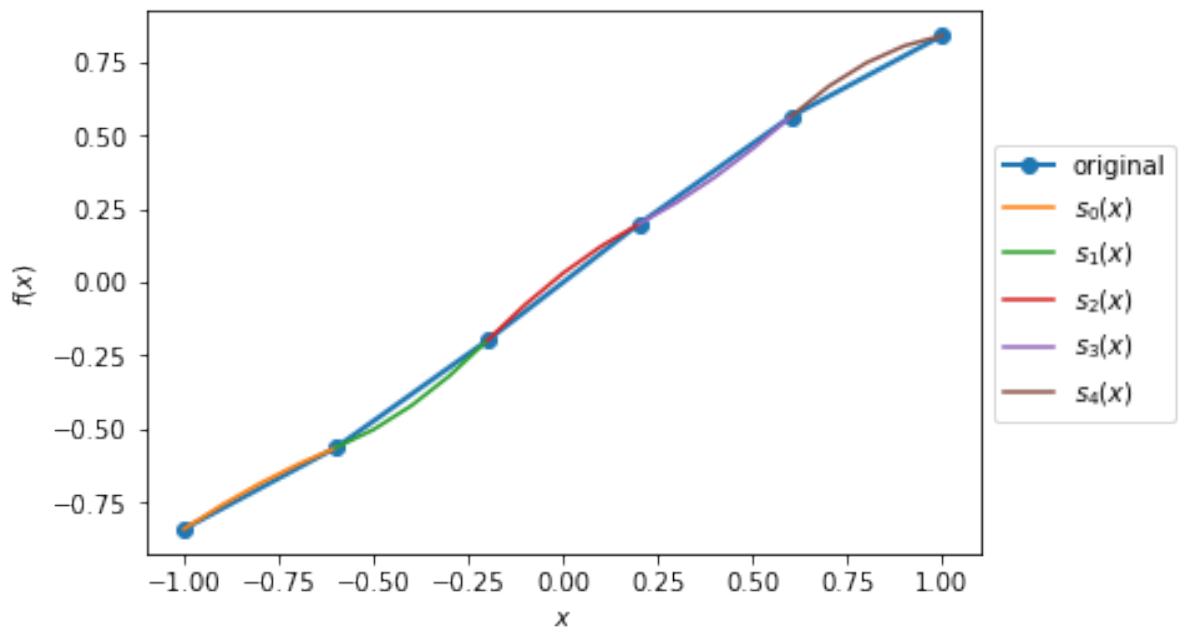
(continued from previous page)

```
>>> xMax2 = 1
>>> nIntervals2 = 10
>>> nSplinePoints2 = 5

>>> y2, A2, coeffs2, ks2 = sp.quad_spline_coeff(f2, xMin2, xMax2, nIntervals2)
>>> fig2 = sp.quad_spline_plot(f2, coeffs2, ks2, nSplinePoints2)
>>> spline_points2 = sp.spline_points(f2, coeffs2, ks2, nSplinePoints2)
>>> sp.spline_error(f2, spline_points2)
0.0074670329670330216

>>> fig2
```

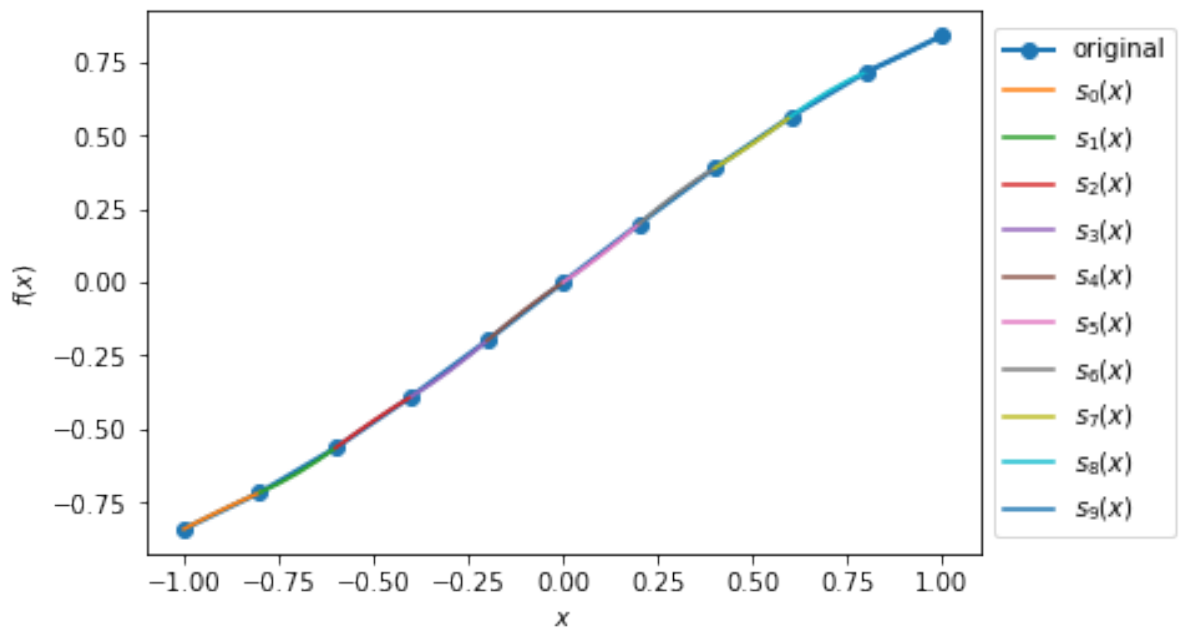


Case 3: Plot the quadratic spline of $f_3(x) = \sin(x), x \in [-1, 1]$ and $x \in [-\pi, \pi]$ with 5 intervals and 10 intervals.

```
>>> def f3(var):
        return np.sin(var)

>>> xMin3 = -1
>>> xMax3 = 1
>>> nIntervals3 = 5
>>> nSplinePoints3 = 5

>>> y3, A3, coeffs3, ks3 = sp.quad_spline_coeff(f3, xMin3, xMax3, nIntervals3)
>>> fig3 = sp.quad_spline_plot(f3, coeffs3, ks3, nSplinePoints3)
>>> spline_points3 = sp.spline_points(f3, coeffs3, ks3, nSplinePoints3)
>>> sp.spline_error(f3, spline_points3)
0.015578205778177232

>>> fig3
```




```
>>> xMin4 = -1
>>> xMax4 = 1
>>> nIntervals4 = 10
>>> nSplinePoints4 = 5

>>> y4, A4, coeffs4, ks4 = sp.quad_spline_coeff(f3, xMin4, xMax4, nIntervals4)
>>> fig4 = sp.quad_spline_plot(f3, coeffs4, ks4, nSplinePoints4)
>>> spline_points4 = sp.spline_points(f3, coeffs4, ks4, nSplinePoints4)
>>> sp.spline_error(f3, spline_points4)
0.0034954287455489196

>>> fig4
```



Note: We can see that the quadratic splines do not work that well with linear-ish functions. While adding more intervals may help to make the approximated splines better.

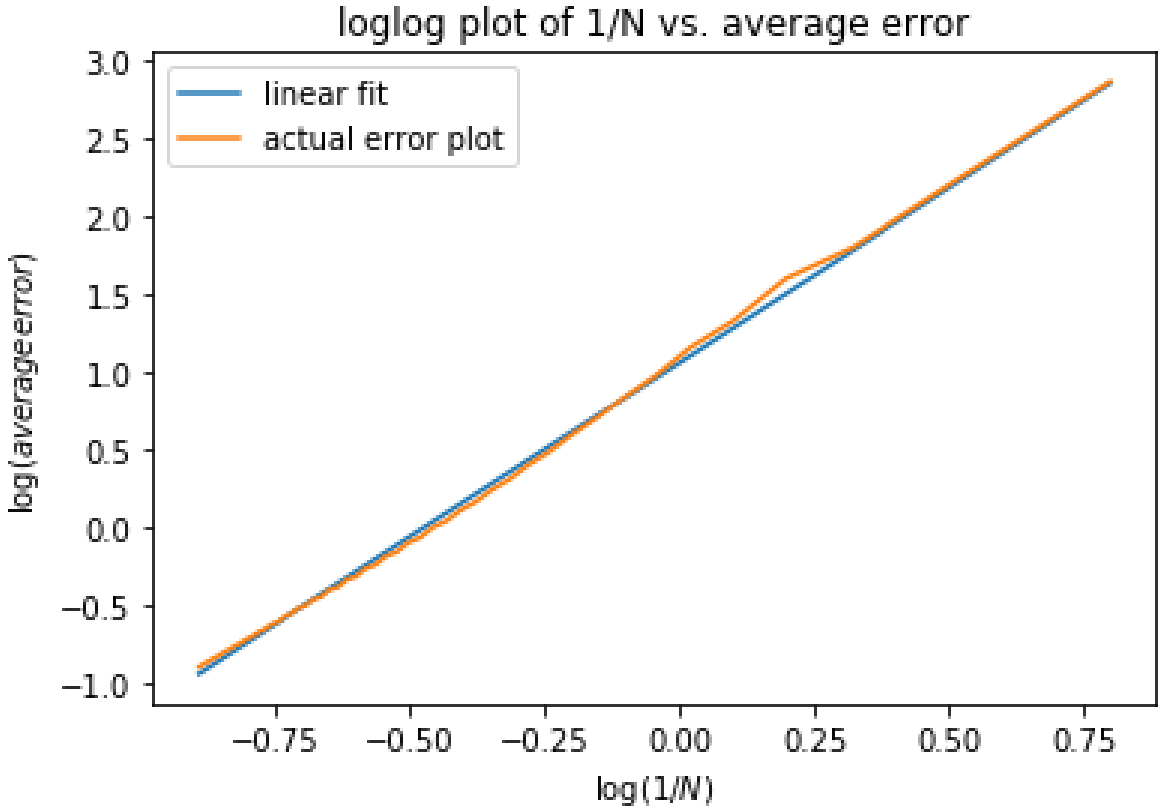
Casee 4: Here we demonstrate that the more intervals will make the splines approximations better using a $log - log$ plot of the absolute average error with respect to :math: frac{1}{N}‘ with $f(x) = 10^x, x \in [-\pi, \pi]$ at intervals from 5 to 100.

```
>>> def f(var):
        return 10 ** var

>>> xMin = -sp.np.pi
>>> xMax = sp.np.pi
>>> nIntervalsList = sp.np.arange(1, 50, 1)
>>> nSplinePoints = 10
>>> squaredErrorList = []

>>> for nIntervals in nIntervalsList:
        y, A, coeffs, ks = sp.quad_spline_coeff(f, xMin, xMax, nIntervals)
        spline_points = sp.spline_points(f, coeffs, ks, nSplinePoints)
        error = sp.spline_error(f, spline_points)
        squaredErrorList.append(error)

>>> plt.figure()

>>> coefficients = np.polyfit(np.log10(2*np.pi/nIntervalsList), np.
↪log10(squaredErrorList), 1)
>>> polynomial = np.poly1d(coefficients)
>>> ys = polynomial(np.log10(2*np.pi/nIntervalsList))
>>> plt.plot(np.log10(2*np.pi/nIntervalsList), ys, label='linear fit')
>>> plt.plot(np.log10(2*np.pi/nIntervalsList), np.log10(squaredErrorList), label=
↪'actual error plot')
>>> plt.xlabel(r'$\log(1/N)$')
>>> plt.ylabel(r'$\log(average error)$')
```

(continues on next page)

(continued from previous page)

```
>>> plt.legend()
>>> plt.title('loglog plot of 1/N vs. average error')
>>> plt.show()
```




```
>>> beta, alpha = coefficients[0], 10**coefficients[1]
>>> beta, alpha
(2.2462166565957835, 11.414027075895813)
```

Note: We can see in the $log - log$ plot that the log of absolute average error is proportional to the log of $\frac{1}{N}$, i.e. $E_{1/N} \approx 11.4(\frac{1}{N})^{2.25}$.

6.3.4 Drawing with Splines

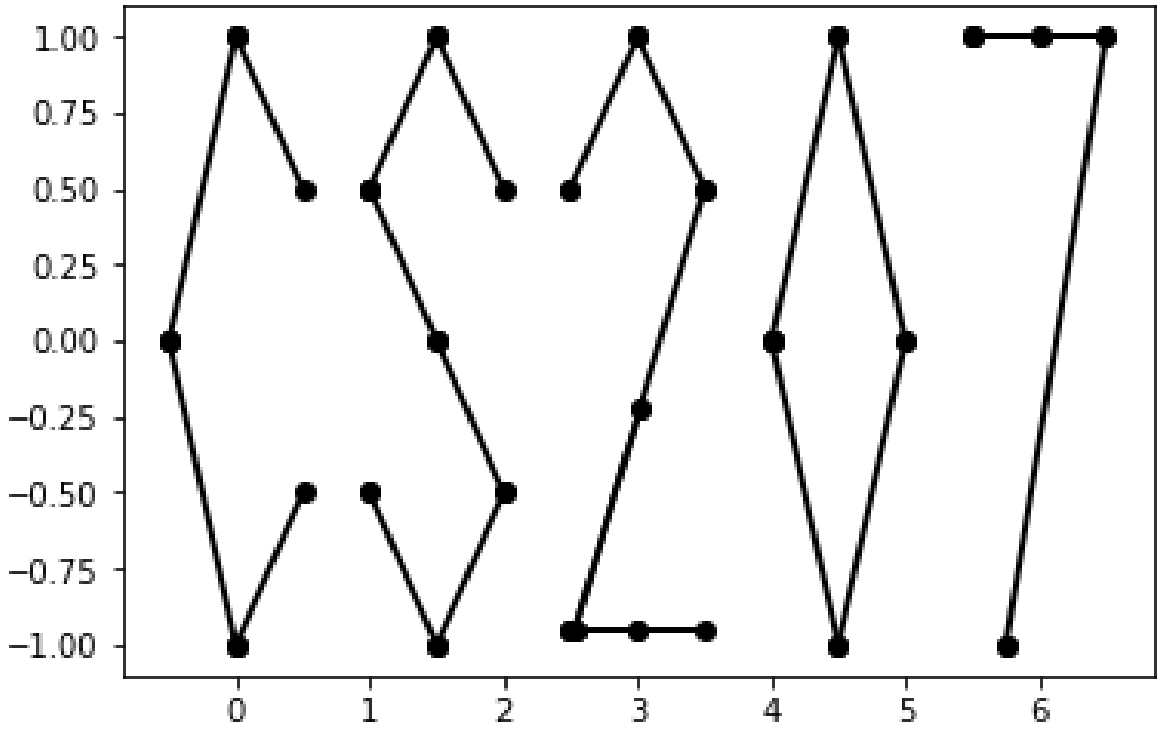
This graph is shipped within `DeriveAlive` package as a surprise.

We want to draw a graph based on the follow 20 functions.

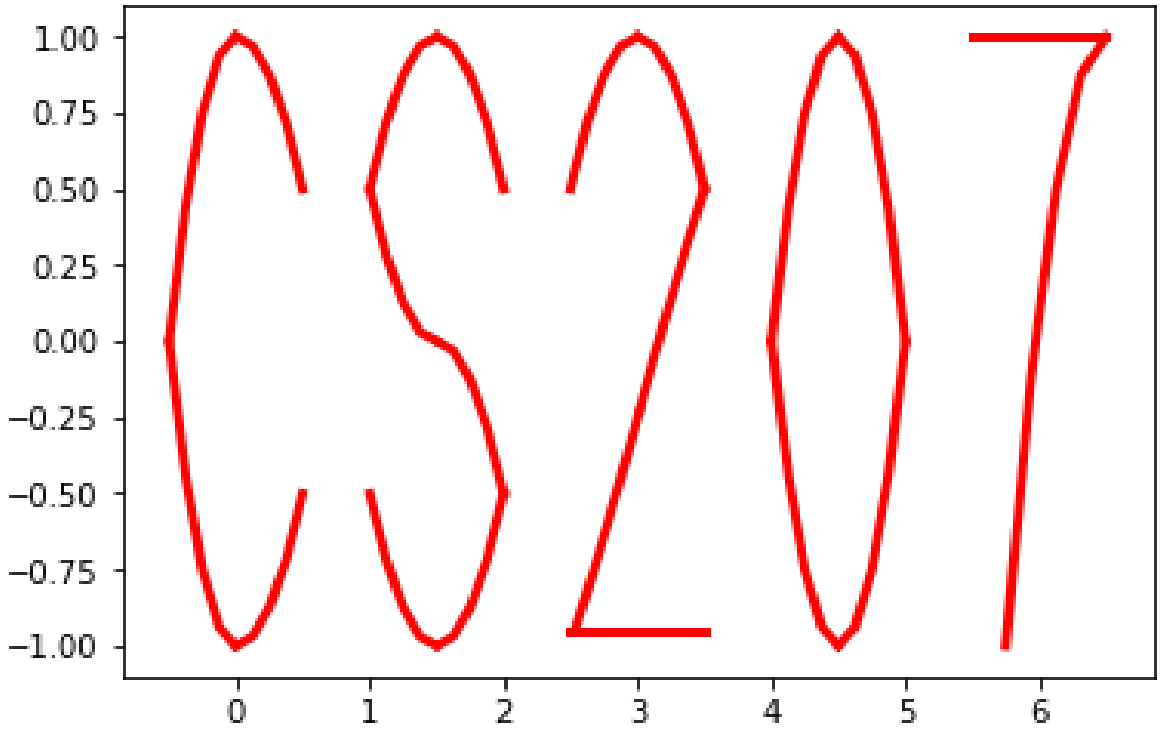
- $f_1(x) = \frac{-1}{0.5^2}x^2 + 1, x \in [-0.5, 0]$
- $f_2(x) = \frac{1}{0.5^2}x^2 - 1, x \in [-0.5, 0]$
- $f_3(x) = \frac{-1}{0.5}x^2 + 1, x \in [0, 0.5]$

- $f_4(x) = \frac{1}{0.5}x^2 - 1, x \in [0, 0.5]$
- $f_6(x) = \frac{-1}{0.5}(x - 1.5)^2 + 1, x \in [1, 1.5]$
- $f_7(x) = \frac{1}{0.5}(x - 1.5)^2 - 1, x \in [1, 1.5]$
- $f_8(x) = \frac{-1}{0.5}(x - 1.5)^2, x \in [1.5, 2]$
- $f_9(x) = \frac{-1}{0.5}(x - 1.5)^2 + 1, x \in [1.5, 2]$
- $f_{10}(x) = \frac{1}{0.5}(x - 1.5)^2 - 1, x \in [1.5, 2]$
- $f_{11}(x) = \frac{-1}{0.5}(x - 3)^2 + 1, x \in [2.5, 3]$
- $f_{12}(x) = \frac{-1}{0.5}(x - 3)^2 + 1, x \in [3, 3.5]$
- $f_{13}(x) = 1.5x - 4.75, x \in [2.5, 3.5]$
- $f_{14}(x) = -1, x \in [2.5, 3.5]$
- $f_{15}(x) = \frac{-1}{0.5^2}(x - 4.5)^2 + 1, x \in [4, 4.5]$
- $f_{16}(x) = \frac{1}{0.5^2}(x - 4.5)^2 - 1, x \in [4, 4.5]$
- $f_{17}(x) = \frac{-1}{0.5^2}(x - 4.5)^2 + 1, x \in [4, 4.5]$
- $f_{18}(x) = \frac{1}{0.5^2}(x - 4.5)^2 - 1, x \in [4.5, 5]$
- $f_{19}(x) = 1, x \in [5.5, 6.5]$
- $f_{20}(x) = \frac{-1}{(-0.75)^2}(x - 6.5)^2 + 1, x \in [5.75, 6.5]$

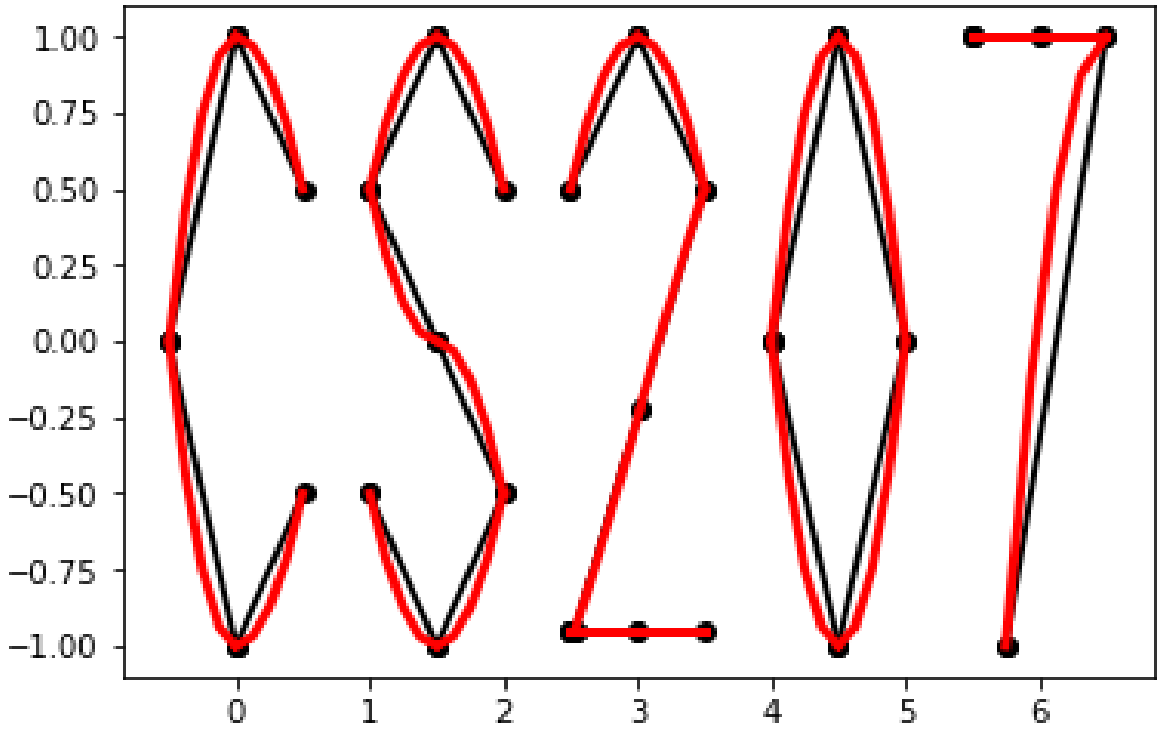
```
>>> import surprise
# We first draw out the start and end points of each function
>>> surprise.drawPoints()
```



```
# Then we use the spline suite to draw quadratic splines based on the two points
>>> surprise.drawSpline()
```




```
>>> surprise.drawTogether()
```


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
1
2
3
4
5
6

CHAPTER 7

Future

Currently, our `DeriveAlive` can handle scalar to scalar, scalar to vector, vector to scalar and vector to vector functions. The further improvement for the software can be expected as follows:

7.1 Module Extension

- Reverse mode. Now that our `DeriveAlive` can work perfectly with the forward mode, we are expecting to implement the reverse mode as well. This improvement will allow our users to play with custom Neural Network models using backpropagation.
- Hessian. By calculating and storing the second derivatives in a Hessian matrix, we can make use of more applications of automatic differentiation that use second derivatives, such as Newton optimization and cubic splines.
- Higher-order splines (cubic). We also want to extend the quadratic spline suite to a cubic spline suite or even higher order splines, which would utilize higher order derivatives to be implemented using autodifferentiation. We would also like to allow users to draw any custom plots with this module.

CHAPTER 8

References

- CS 207 Lectures 9 and 10 (Autodifferentiation)
- AM 205 Lectures 2 (Splines)
- Elementary functions: https://en.wikipedia.org/wiki/Elementary_function
- Package distribution: https://packaging.python.org/tutorials/packaging-projects/
- Newton root finding (univariate): https://en.wikipedia.org/wiki/Newton%27s_method
- Newton root finding for $\mathbb{R}^m \Rightarrow \mathbb{R}^1$: https://calculus.subwiki.org/wiki/Newton%27s_method_for_root-finding_for_a_vector-valued_function_of_a_vector_variable
- Gradient descent: https://en.wikipedia.org/wiki/Gradient_descent
- Dataset for predicting housing prices: http://cs229.stanford.edu