
DFYS-Autodiff Documentation

Release 1.0.3

fei Chen; YuetingFuoyu; Yan Zhao

Dec 12, 2018

Contents

1	Background	1
1.1	Introduction	1
1.2	Mathematical Background	1
2	Installation	3
2.1	Install Through PyPI	3
2.2	Install Manually	3
3	Getting Started	5
3.1	Univariate Functions	5
3.2	Multivariate Functions	6
3.3	Vector Functions	9
4	Libraries Demo	11
4.1	<code>autodiff.forward</code>	11
4.2	<code>autodiff.rootfinding</code>	15
4.3	<code>autodiff.optimize</code>	17
4.4	<code>autodiff.plot</code>	22
4.5	<code>autodiff.backprop</code>	24
5	Implementation Details	27
5.1	High-level Design	27
5.2	Core Classes	28
5.3	External Dependencies	28
6	Functions Details	29
6.1	<code>Autodiff.forward</code>	29
6.2	<code>Autodiff.backprop</code>	29
6.3	<code>Autodiff.rootfinding</code>	29
6.4	<code>Autodiff.optimize</code>	29
6.5	<code>Autodiff.plot</code>	29
7	Software Organization	31
7.1	Directory Structure	31
7.2	Modules	32
7.3	Test Automation	32
7.4	Distribution	32

8	Future Development	33
8.1	1. Optimization	33
8.2	2. Extensions	33
8.3	3. Improvement	33

1.1 Introduction

Automatic differentiation (AD) is a family of techniques for efficiently and accurately evaluating derivatives of numeric functions expressed as computer programs. Application of AD includes Newton's method for solving nonlinear equations, real-parameter optimization, probabilistic inference, and backpropagation in neural networks. AD has been extremely popular because of the booming development in machine learning and deep learning techniques. Our AD software package enable user to calculate derivatives using the forward and reverse mode.

Our package has feature including support for second order derivatives (including Hessian matrix), rooting finding, optimization(Newton, Gradient Descent, BFGS), and backpropagation.

1.2 Mathematical Background

Automatic Differentiation decomposes a complex function into a sequence of operations on elementary functions, evaluates the derivatives at each intermediate stage, repeatedly applies the chain rule to obtain the derivative of the outermost function. We provides explanations for related math concepts below.

Elimentary functions

The class of functions consisting of the polynomials, the exponential functions, the logarithmic functions, the trigonometric functions, the inverse trigonometric functions, and the functions obtained from those listed by the four arithmetic operations and by superposition(i.e. composition), applied by finitely many times.

Chain Rule - Used to compute the derivative of a composite function - Core of automatic differentiation

For the first derivative:

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$$

For the second derivative:

$$\frac{\partial^2 t}{\partial x_i \partial x_j} = \sum_k \left(\frac{\partial y}{\partial u_k} \frac{u_k^2}{\partial x_i \partial x_j} \right) + \sum_{k,l} \left(\frac{\partial^2 y}{\partial u_k \partial u_l} \frac{\partial u_k}{\partial x_i} \frac{\partial u_l}{\partial x_j} \right)$$

Topological Graph - Each node represent a variable - Arrows indicate topological orders(order of operations) and operations themselves.

Forward Mode Autodifferentiation

Follow the topological order and store the values of each variable in the nodes. visit each node in topological order. Let x denote our innermost function. For variable $u_i = g_i(v)$ we already know $\frac{dv}{dx}$, calculate $\frac{du_i}{dx} = \frac{du_i}{dv} \frac{dv}{dx}$

Reverse Mode Autodifferentiation

Has forward computation and backward computation

Step 1: Forward Computation

Follow the topological order and store the values of each variable in each nodes.

Step 2: Backward Computation

let y denote our final output variable and u_j, v_j denote the intermediate variables

1. Initialize all partial derivative $\frac{dy}{du_j}$ to 0 and $dy/dy = 1$
2. visit each node in reverse topological order. For variable $u_i = g_i(v_1, \dots, v_n)$ we already know $\frac{dy}{du_i}$, increment $\frac{dy}{dv_j}$ by $\frac{dy}{du_i} \frac{du_i}{dv_j}$

2.1 Install Through PyPI

The easiest way to install `autodiff` is by `pip`. Just type in `pip install DYFS-autodiff` in the command line.

```
pip install DYFS-autodiff
```

2.2 Install Manually

The user can choose to install `autodiff` directly from the source in this repository. We suppose that the user has already installed `pip` and `virtualenv`:

1. clone the project repo by `git clone git@github.com:D-F-Y-S/cs207-FinalProject.git`
2. `cd` into the local repo and create a virtual environment by `virtualenv env`
3. activate the virtual environment by `source env/bin/activate` (use `deactivate` to deactivate the virtual environment later.)
4. install the dependencies by `pip install -r requirements.txt`
5. install `autodiff` by `pip install -e .`

3.1 Univariate Functions

The standard workflow for autodiff is to first initiate a `Variable`, or several `Variables`. We then use these `Variable` to construct `Expressions`, which can then be queried for values and derivatives.

```
In [24]: import numpy          as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

from autodiff.forward import *
```

Suppose we want to calculate the derivatives of $f(x) = \cos(\pi x) \exp(-x^2)$. We can start with creating a `Variable` called `x`.

```
In [3]: x = Variable()
```

We then create the `Expression` for $f(x)$. Note that here `cos` and `exp` are library functions from `autodiff`.

```
In [4]: f = cos(np.pi*x)*exp(-x**2)
```

We can then evaluate $f(x)$'s value and derivative by calling the `evaluation_at` method and the `derivative_at` method. For `derivative_at` method, the first argument specifies which variable to take derivative with respect to, the second argument specifies which point in the domain are the derivative to be calculated.

```
In [5]: f.evaluation_at({x: 1})
```

```
Out[5]: -0.36787944117144233
```

```
In [6]: f.derivative_at(x, {x: 1})
```

```
Out[6]: 0.7357588823428846
```

The `derivative_at` method supports second order derivative. If we want to calculate $\frac{d^2 f}{dx^2}$, we can add another argument `order=2`.

```
In [7]: f.derivative_at(x, {x: 1}, order=2)
```

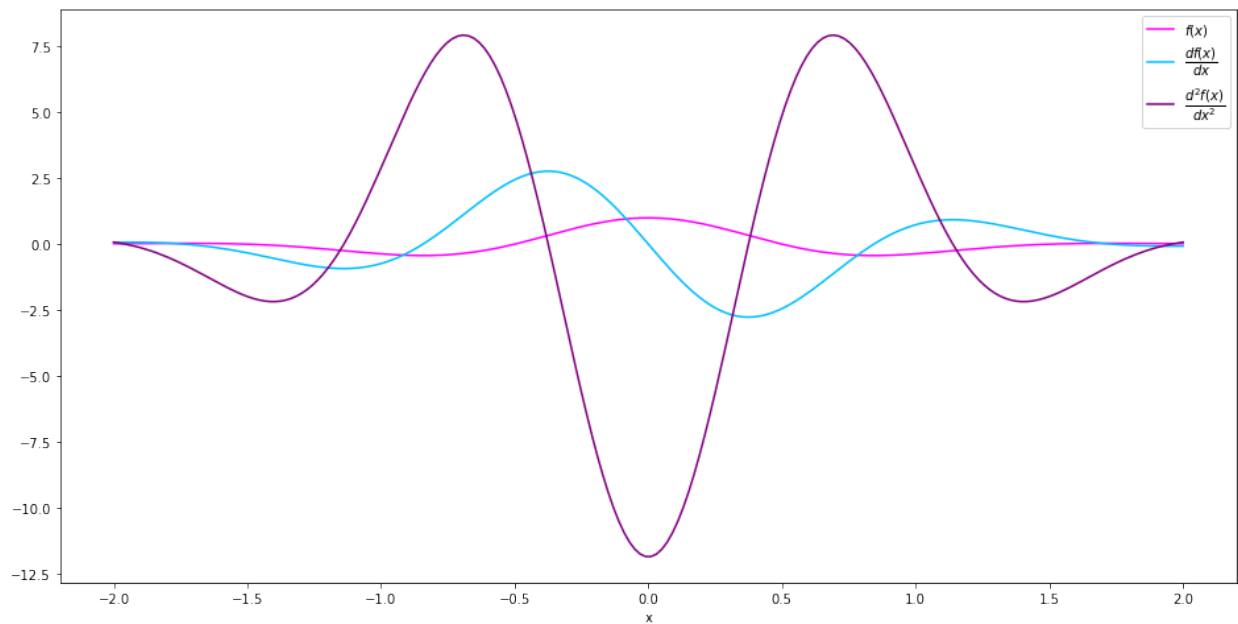
```
Out[7]: 2.895065669313077
```

Both the methods `evaluation_at` and `derivative_at` are vectorized, and instead of pass in a scalar value, we can pass in a `numpy.array`, and the output will be `f`'s value / derivative at all entried of the input. For example, we can calculate the value, first order derivative and second order derivative of $f(x)$ on the interval $[-2, 2]$ simply by

```
In [8]: interval = np.linspace(-2, 2, 200)
        values = f.evaluation_at({x: interval})
        der1st = f.derivative_at(x, {x: interval})
        der2nd = f.derivative_at(x, {x: interval}, order=2)
```

Let's see what they look like.

```
In [9]: fig = plt.figure(figsize=(16, 8))
        plt.plot(interval, values, c='magenta', label='$f(x)$')
        plt.plot(interval, der1st, c='deepskyblue', label='$\frac{df(x)}{dx}$')
        plt.plot(interval, der2nd, c='purple', label='$\frac{d^2f(x)}{dx^2}$')
        plt.xlabel('x')
        plt.legend()
        plt.show()
```



3.2 Multivariate Functions

The workflow with multivariate functions are essentially the same.

Suppose we want to calculate the derivatives of $g(x, y) = \cos(\pi x) \cos(\pi y) \exp(-x^2 - y^2)$. We can start with adding another Variable called `y`.

```
In [10]: y = Variable()
```

We then create the Expression for $g(x, y)$.

```
In [11]: g = cos(np.pi*x) * cos(np.pi*y) * exp(-x**2-y**2)
```

We can then evaluate $f(x)$'s value and derivative by calling the `evaluation_at` method and the `derivative_at` method, as usual.

```
In [12]: g.evaluation_at({x: 1.0, y: 1.0})
```

```
Out[12]: 0.1353352832366127
```

```
In [13]: g.derivative_at(x, {x: 1.0, y: 1.0})
```

```
Out[13]: -0.27067056647322535
```

```
In [14]: g.derivative_at(x, {x: 1.0, y: 1.0})
```

```
Out[14]: -1.0650351405815222
```

Now we have two variables, we may want to calculate $\frac{\partial^2 g}{\partial x \partial y}$. We can just replace the first argument of `derivative_at` to a tuple `(x, y)`. In this case the third argument `order=2` can be omitted, because the Expression can infer from the first argument that we are looking for a second order derivative.

```
In [15]: g.derivative_at((x, y), {x: 1.0, y: 1.0})
```

```
Out[15]: 0.5413411329464506
```

We can also ask `g` for its Hessian matrix. A `numpy.array` will be returned.

```
In [29]: g.hessian_at({x: 1.0, y: 1.0})
```

```
Out[29]: array([[ -1.06503514,  0.54134113],
               [ 0.54134113, -1.06503514]])
```

Since the `evaluation_at` method and `derivative_at` method are vectorized, we can as well pass in a mesh grid, and the output will be a grid of the same shape. For example, we can calculate the value, first order derivative and second order derivative of $f(x)f(y)$ on the interval $x \in [-2, 2], y \in [-2, 2]$ simply by

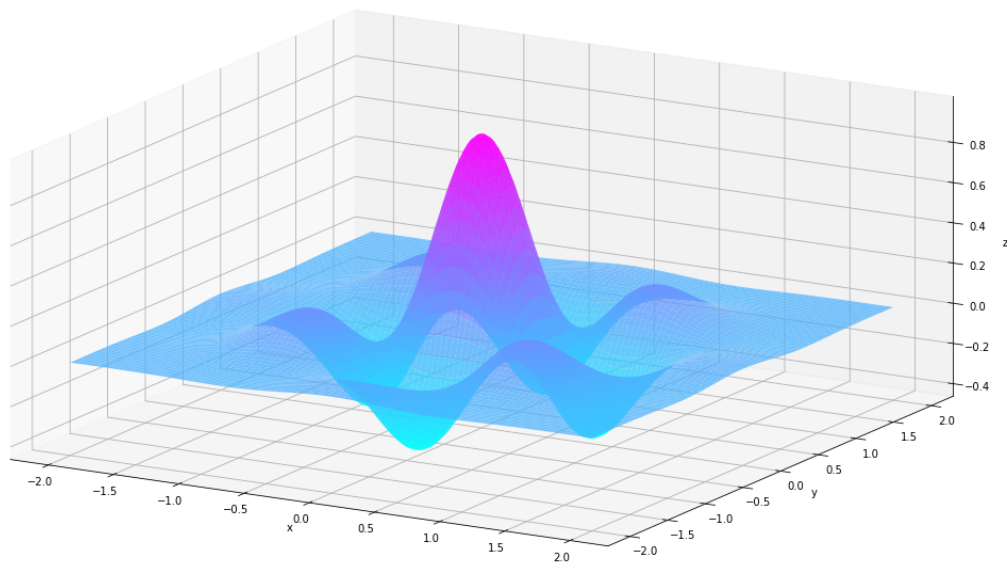
```
In [20]: us, vs = np.linspace(-2, 2, 200), np.linspace(-2, 2, 200)
         uu, vv = np.meshgrid(us, vs)
```

```
In [21]: values = g.evaluation_at(      {x: uu, y: vv})
         der1st = g.derivative_at(x,    {x: uu, y: vv})
         der2nd = g.derivative_at((x, y), {x: uu, y: vv})
```

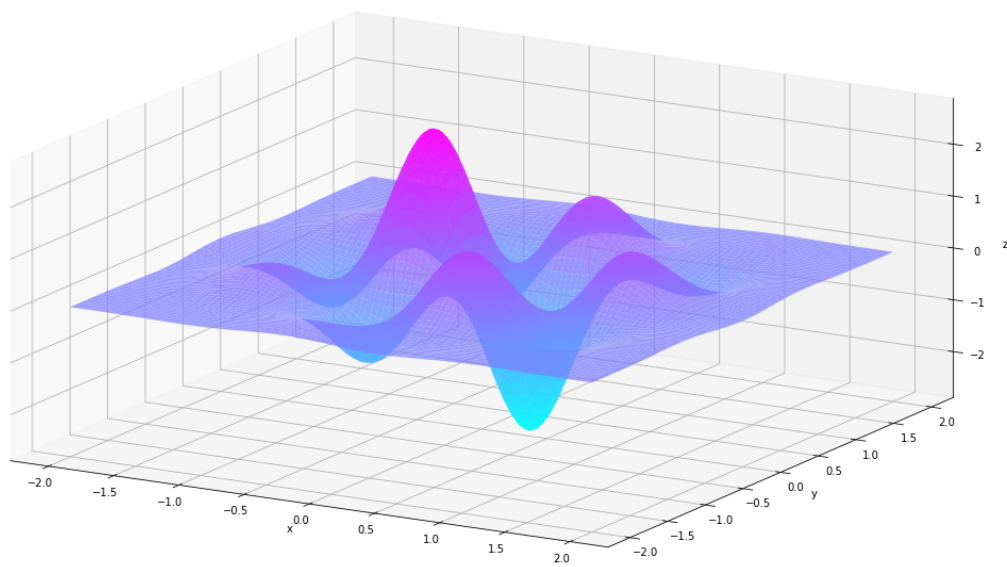
Let's see what they look like.

```
In [22]: def plt_surf(uu, vv, zz):
         fig = plt.figure(figsize=(16, 8))
         ax = Axes3D(fig)
         surf = ax.plot_surface(uu, vv, zz, rstride=2, cstride=2, alpha=0.8, cmap='cool')
         ax.set_xlabel('x')
         ax.set_ylabel('y')
         ax.set_zlabel('z')
         ax.set_proj_type('ortho')
         plt.show()
```

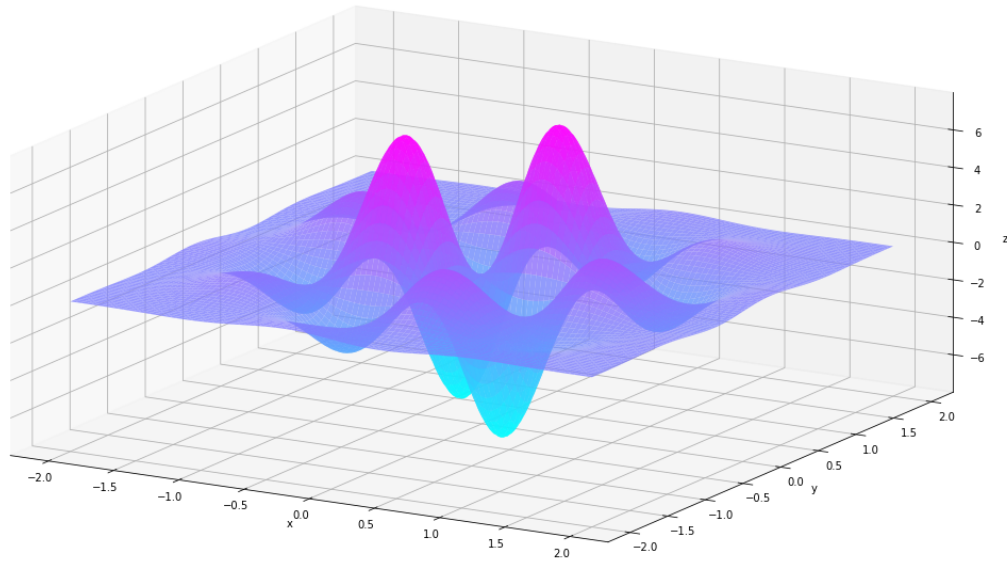
```
In [25]: plt_surf(uu, vv, values)
```



```
In [26]: plt_surf(uu, vv, der1st)
```



```
In [27]: plt_surf(uu, vv, der2nd)
```



3.3 Vector Functions

Functions defined on $\mathbb{R}^n \mapsto \mathbb{R}^m$ are also supported. Here we create an `VectorFunction` that represents $h\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \begin{bmatrix} f(x) \\ g(x, y) \end{bmatrix}$.

```
In [30]: h = VectorFunction(exprlist=[f, g])
```

We can then evaluate $h\left(\begin{bmatrix} x \\ y \end{bmatrix}\right)$'s value and gradient $\left(\begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial g}{\partial x} \end{bmatrix}\right)$ and $\left(\begin{bmatrix} \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial y} \end{bmatrix}\right)$ by calling its `evaluation_at` method and

`gradient_at` method. The `jacobian_at` function returns the Jacobian matrix $\begin{pmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{pmatrix}$.

```
In [31]: h.evaluation_at({x: 1.0, y: -1.0})
Out[31]: array([-0.36787944,  0.13533528])
In [35]: h.gradient_at(0, {x: 1.0, y: -1.0})
Out[35]: array([0., 0.])
In [33]: h.jacobian_at({x: 1.0, y: -1.0})
Out[33]: array([[ 0.73575888,  0.          ],
                [-0.27067057,  0.27067057]])
```


4.1 autodiff.forward

4.1.1 Univariate Functions

The standard workflow for autodiff is to first initiate a `Variable`, or several `Variables`. We then use these `Variable` to construct `Expressions`, which can then be queried for values and derivatives.

```
In [65]: import numpy          as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from autodiff.forward import *
```

Suppose we want to calculate the derivatives of $f(x) = \cos(\pi x) \exp(-x^2)$. We can start with creating a `Variable` called `x`.

```
In [66]: x = Variable()
```

We then create the `Expression` for $f(x)$. Note that here `cos` and `exp` are library functions from `autodiff`.

```
In [67]: f = cos(np.pi*x)*exp(-x**2)
```

We can then evaluate $f(x)$'s value and derivative by calling the `evaluation_at` method and the `derivative_at` method. For `derivative_at` method, the first argument specifies which variable to take derivative with respect to, the second argument specifies which point in the domain are the derivative to be calculated.

```
In [68]: f.evaluation_at({x: 1})
```

```
Out[68]: -0.36787944117144233
```

```
In [69]: f.derivative_at(x, {x: 1})
```

```
Out[69]: 0.73575888234288456
```

The `derivative_at` method supports second order derivative. If we want to calculate $\frac{d^2 f}{dx^2}$, we can add another argument `order=2`.

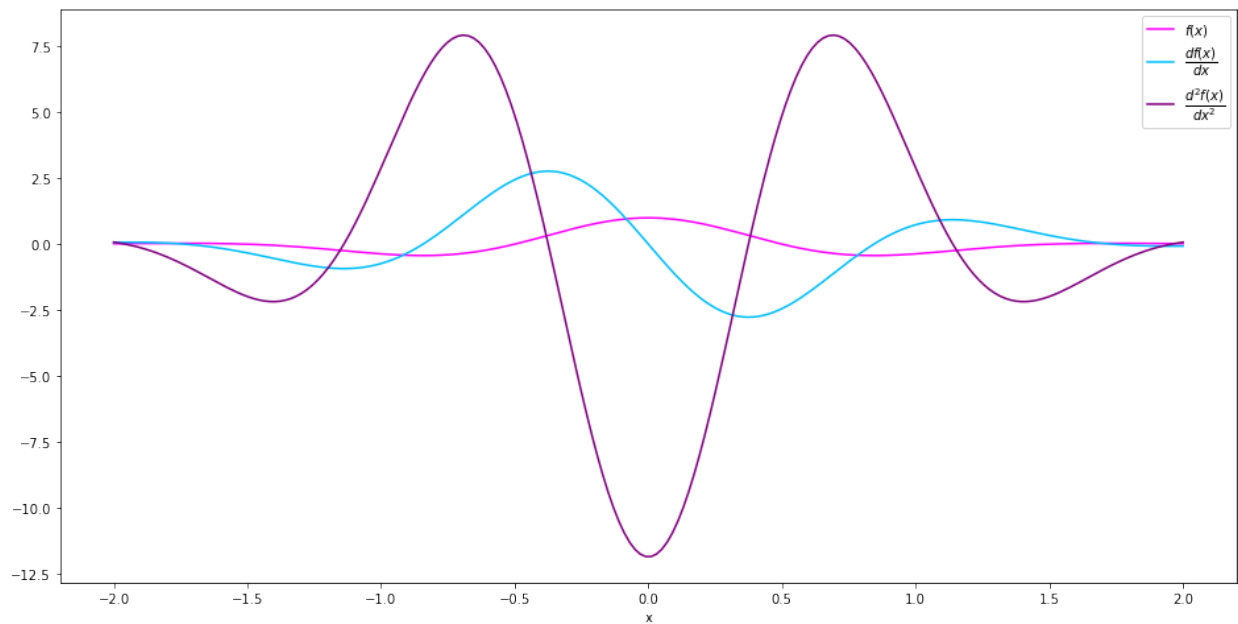
```
In [70]: f.derivative_at(x, {x: 1}, order=2)
```

```
Out [70]: 2.8950656693130772
```

Both the methods `evaluation_at` and `derivative_at` are vectorized, and instead of pass in a scalar value, we can pass in a `numpy.array`, and the output will be `f`'s value / derivative at all entried of the input. For example, we can calculate the value, first order derivative and second order derivative of $f(x)$ on the interval $[-2, 2]$ simply by

```
In [71]: interval = np.linspace(-2, 2, 200)
        values = f.evaluation_at( {x: interval})
        der1st = f.derivative_at(x, {x: interval})
        der2nd = f.derivative_at(x, {x: interval}, order=2)

In [72]: fig = plt.figure(figsize=(16, 8))
        plt.plot(interval, values, c='magenta',      label='$f(x)$')
        plt.plot(interval, der1st, c='deepskyblue',  label='$\frac{df(x)}{dx}$')
        plt.plot(interval, der2nd, c='purple',       label='$\frac{d^2f(x)}{dx^2}$')
        plt.xlabel('x')
        plt.legend()
        plt.show()
```



4.1.2 Multivariate Functions

The workflow with multivariate functions are essentially the same.

Suppose we want to calculate the derivatives of $g(x, y) = \cos(\pi x) \cos(\pi y) \exp(-x^2 - y^2)$. We can start with adding another `Variable` called `y`.

```
In [73]: y = Variable()
```

We then create the `Expression` for $g(x, y)$.

```
In [74]: g = cos(np.pi*x) * cos(np.pi*y) * exp(-x**2-y**2)
```

We can then evaluate $f(x)$'s value and derivative by calling the `evaluation_at` method and the `derivative_at` method, as usual.

```
In [75]: g.evaluation_at({x: 1.0, y: 1.0})
```

```
Out [75]: 0.1353352832366127
```



```
In [76]: g.derivative_at(x, {x: 1.0, y: 1.0})
```

```
Out[76]: -0.27067056647322535
```

```
In [77]: g.derivative_at(x, {x: 1.0, y: 1.0})
```

```
Out[77]: -0.27067056647322535
```

Now we have two variables, we may want to calculate $\frac{\partial^2 g}{\partial x \partial y}$. We can just replace the first argument of `derivative_at` to a tuple `(x, y)`. In this case the third argument `order=2` can be omitted, because the Expression can infer from the first argument that we are looking for a second order derivative.

```
In [78]: g.derivative_at((x, y), {x: 1.0, y: 1.0})
```

```
Out[78]: 0.54134113294645059
```

We can also ask `g` for its Hessian matrix. A `numpy.array` will be returned.

```
In [79]: g.hessian_at({x: 1.0, y: 1.0})
```

```
Out[79]: array([[ -1.06503514,  0.54134113],
               [ 0.54134113, -1.06503514]])
```

Since the `evaluation_at` method and `derivative_at` method are vectorized, we can as well pass in a mesh grid, and the output will be a grid of the same shape. For example, we can calculate the value, first order derivative and second order derivative of $f(x)f(y)$ on the interval $x \in [-2, 2], y \in [-2, 2]$ simply by

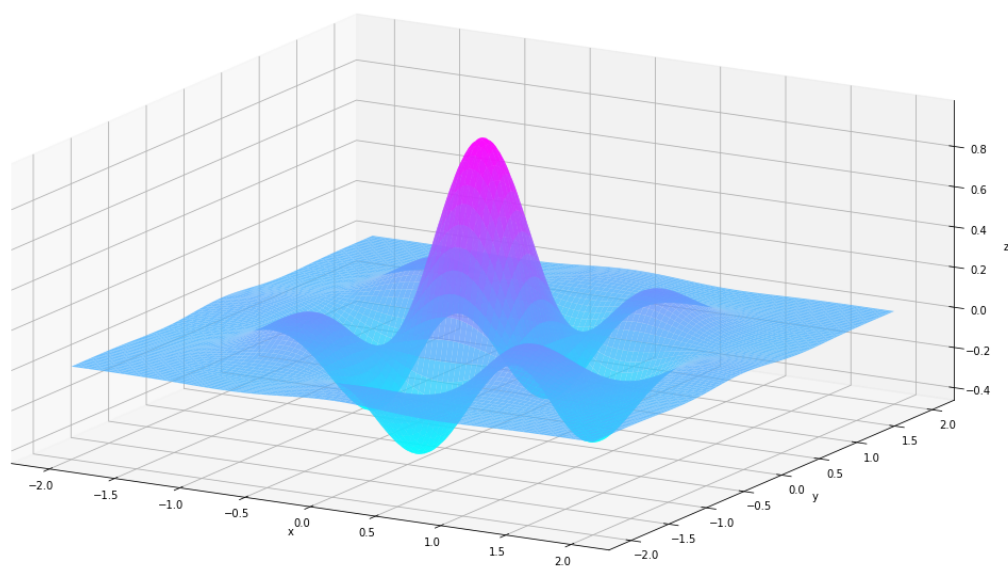
```
In [80]: us, vs = np.linspace(-2, 2, 200), np.linspace(-2, 2, 200)
         uu, vv = np.meshgrid(us, vs)
```

```
In [81]: values = g.evaluation_at(      {x: uu, y: vv})
         der1st = g.derivative_at(x,    {x: uu, y: vv})
         der2nd = g.derivative_at((x, y), {x: uu, y: vv})
```

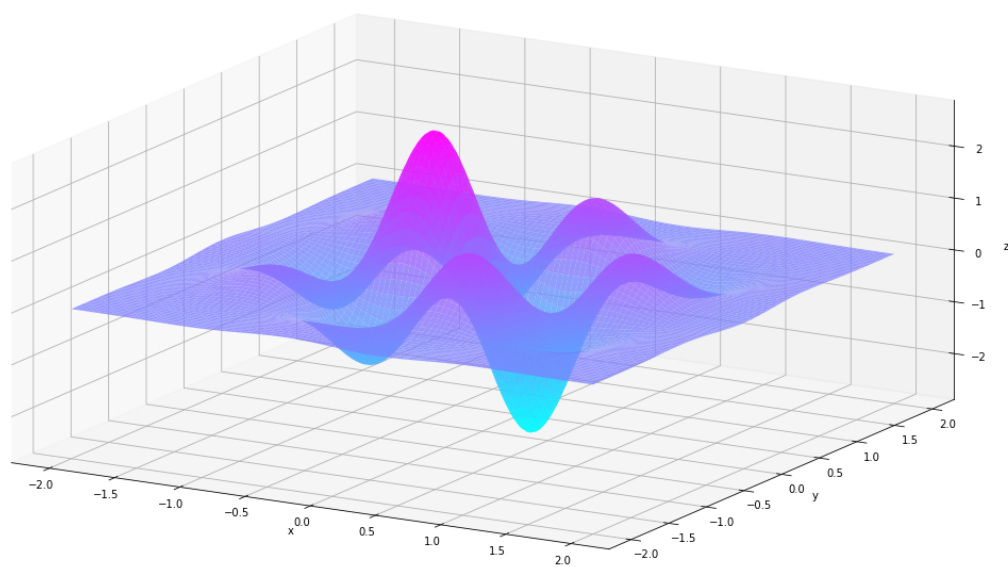
Let's see what they look like.

```
In [82]: def plt_surf(uu, vv, zz):
         fig = plt.figure(figsize=(16, 8))
         ax = Axes3D(fig)
         surf = ax.plot_surface(uu, vv, zz, rstride=2, cstride=2, alpha=0.8, cmap='cool')
         ax.set_xlabel('x')
         ax.set_ylabel('y')
         ax.set_zlabel('z')
         ax.set_proj_type('ortho')
         plt.show()
```

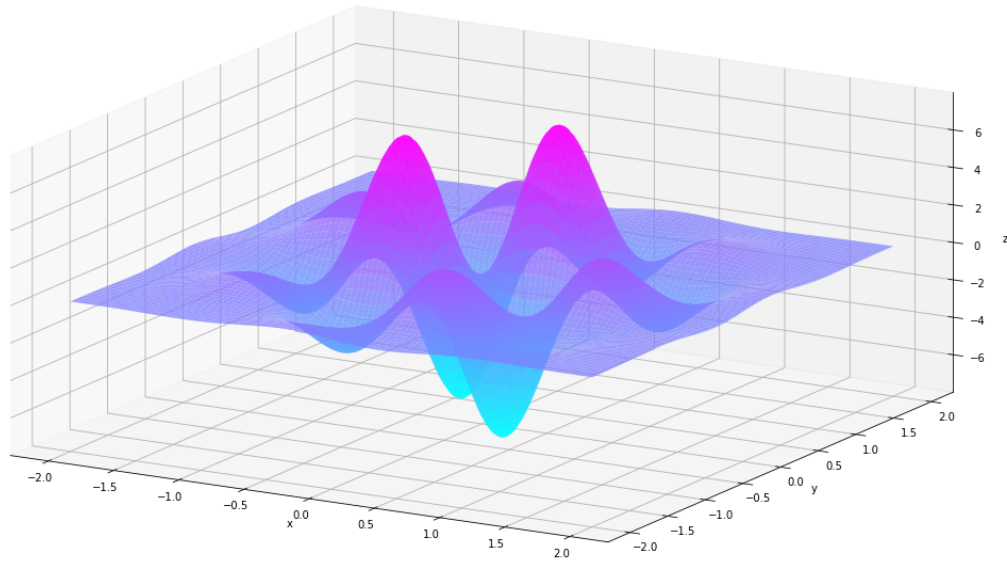
```
In [83]: plt_surf(uu, vv, values)
```



```
In [84]: plt_surf(uu, vv, der1st)
```



```
In [85]: plt_surf(uu, vv, der2nd)
```



4.1.3 Vector Functions

Functions defined on $\mathbb{R}^n \mapsto \mathbb{R}^m$ are also supported. Here we create an `VectorFunction` that represents $h\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \begin{bmatrix} f(x) \\ g(x, y) \end{bmatrix}$.

```
In [86]: h = VectorFunction(exprlist=[f, g])
```

We can then evaluate $h\left(\begin{bmatrix} x \\ y \end{bmatrix}\right)$'s value and gradient $\left(\begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial g}{\partial x} \end{bmatrix}\right)$ and $\left(\begin{bmatrix} \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial y} \end{bmatrix}\right)$ by calling its `evaluation_at` method and

`gradient_at` method. The `jacobian_at` function returns the Jacobian matrix $\begin{pmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{pmatrix}$.

```
In [87]: h.evaluation_at({x: 1.0, y: -1.0})
Out[87]: array([-0.36787944,  0.13533528])
In [88]: h.gradient_at(0, {x: 1.0, y: -1.0})
Out[88]: array([ 0.,  0.])
In [89]: h.jacobian_at({x: 1.0, y: -1.0})
Out[89]: array([[ 0.73575888,  0.          ],
                [-0.27067057,  0.27067057]])
```

4.2 autodiff.rootfinding

Rootfinding module provides function `newton_scalar` to find the root of a given function with arbitrarily many variables. It also works with back propagation mode. Here for visualization purpose we only show up to 2 variables.

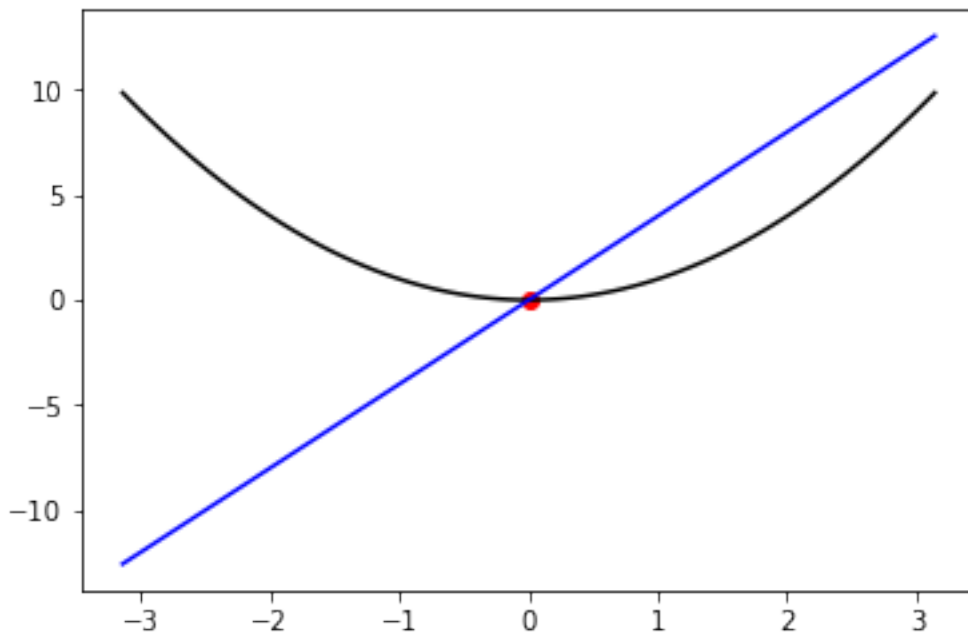
Example1: try to approximate: $f = \sin(x) - 0.4x = 0$ from $x = -2.5, y = -1.5$

```
In [90]: import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
from autodiff.forward import *
from autodiff.rootfinding import *
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

In [91]: x = Variable()
f = x**2-4*x
result_d = newton_scalar(f,{x:1},max_itr=100)

In [92]: xx= np.linspace(-np.pi,np.pi,100)
plt.plot(xx,xx**2,color = 'black')
plt.plot(xx,4*xx,color = 'blue')
plt.scatter([result_d[x]], [f.evaluation_at({x:result_d[x]})],color = 'red')

Out[92]: <matplotlib.collections.PathCollection at 0x107e954e0>
```

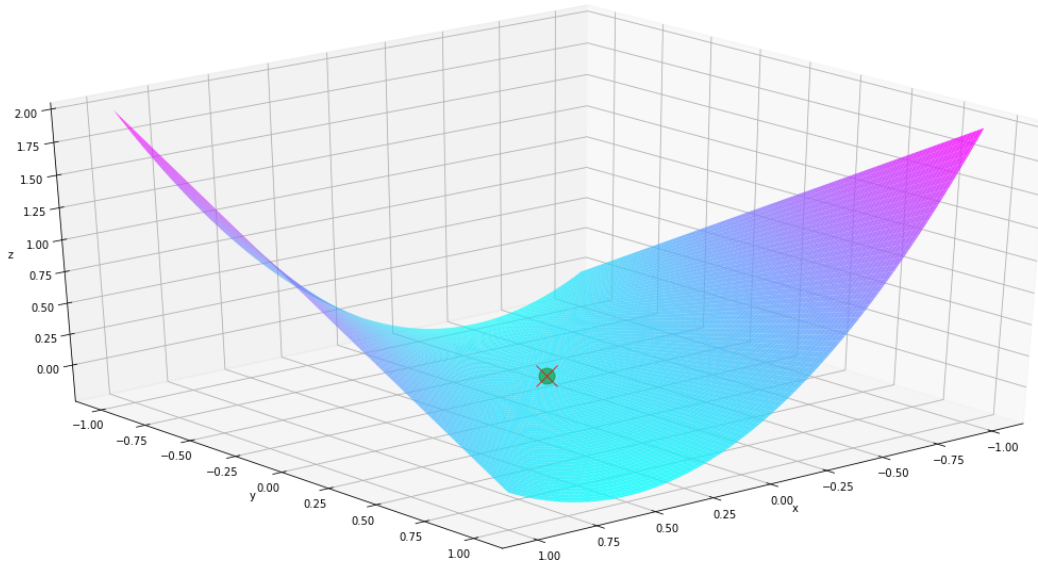


Example 2: $f(x, y) = x^2 - xy = 0$ from $x = 1$, and $y = 10$

```
In [93]: x, y = Variable(), Variable()
f = x**2-x*y
result_d = newton_scalar(f,{x:1,y:10},max_itr = 100)

In [94]: fig = plt.figure(figsize=(16, 8))
ax = Axes3D(fig)
us, vs = np.linspace(-1, 1, 200), np.linspace(-1, 1, 200)
uu, vv = np.meshgrid(us, vs)
zz = f.evaluation_at({x: uu, y:vv})
ax.plot([0], [0], [0], marker='o', markersize=15, c='green',alpha = .5)
surf = ax.plot_surface(uu, vv, zz, rstride=2, cstride=2, alpha=0.8, cmap='cool')
ax.plot([result_d[x]], [result_d[y]],
        [f.evaluation_at({x:result_d[x],y:result_d[y]})],
        marker='x', markersize=20, c='red',alpha = .8)
ax.set_xlabel('x')
```

```
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.view_init(30, 50)
plt.show()
```



4.3 autodiff.optimize

```
In [95]: import numpy          as np
         from autodiff.forward import *
         import autodiff.optimize as opt
         from mpl_toolkits.mplot3d import Axes3D
         import matplotlib.pyplot as plt
         %matplotlib inline
```

We included several basic optimization routines built on `autodiff.forward`. Here we'll use the Rosenbrock function to demonstrate the use of these optimization routines. The Rosenbrock function is defined as $f(x, y) = (a - x)^2 + b(y - x^2)^2$. Here we use $a = 1, b = 100$.

```
In [96]: x, y = Variable(), Variable()
         f = (1-x)**2 + 100*(y-x**2)**2

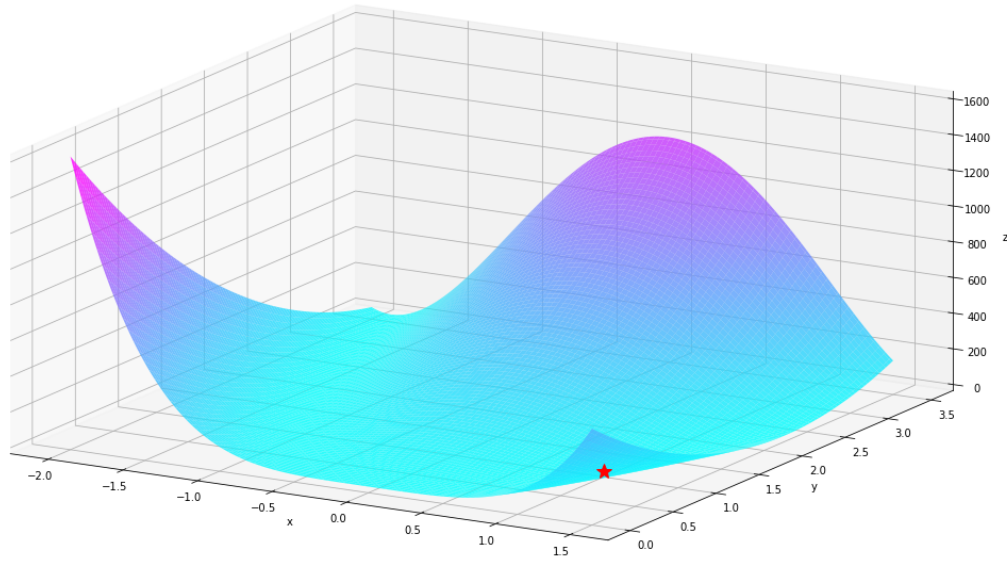
In [97]: us, vs = np.linspace(-2, 1.5, 200), np.linspace(0.0, 3.5, 200)
         uu, vv = np.meshgrid(us, vs)
         values = f.evaluation_at({x: uu, y: vv})
```

The landscape of the function looks like below. The global minimum is at $[-1, 1]$, it is marked by the red star.

```
In [98]: def plt_surf(uu, vv, zz, traj=None, show_dest=False, show_traj=False):
         fig = plt.figure(figsize=(16, 8))
         ax = Axes3D(fig)
         if show_traj: ax.plot(traj[0], traj[1], traj[2], marker='>', markersize=7, c='orange')
         if show_dest: ax.plot([1.0], [1.0], [0.0], marker='*', markersize=15, c='red')
         surf = ax.plot_surface(uu, vv, zz, rstride=2, cstride=2, alpha=0.8, cmap='cool')
         ax.set_xlabel('x')
         ax.set_ylabel('y')
         ax.set_zlabel('z')
```

```
ax.set_proj_type('ortho')  
plt.show()
```

```
In [99]: plt_surf(uu, vv, values, show_dest=True)
```



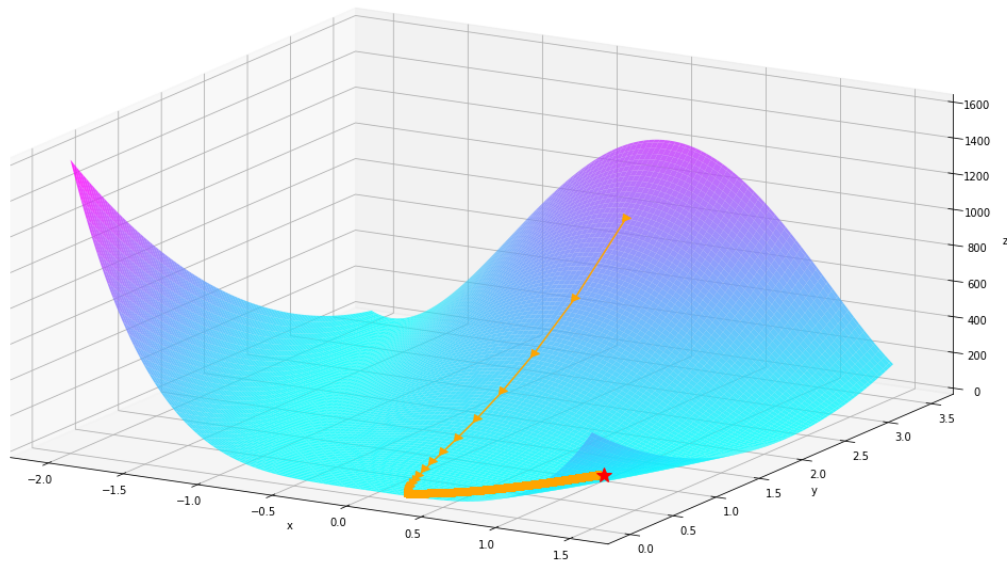
4.3.1 autodiff.optimize.gradient_descent

Let's say we start from (0.0, 3.0). We'll first use gradient descent to find the minimum. The gradient descent is implemented in `autodiff.optimize.gradient_descent`. Here we set the argument `return_history=True` to return a whole history of optimization.

```
In [100]: hist = opt.gradient_descent(f, init_val_dict={x: 0.0, y: 3.0}, max_iter=10000,  
                                     return_history=True)
```

We can plot our optimization path as below. We can see that gradient descent approaches the minimum slowly because the gradient around the minimum is small.

```
In [101]: hist = np.array(hist)  
us, vs = hist[:, 0].flatten(), hist[:, 1].flatten()  
zs      = f.evaluation_at({x: us, y: vs})  
plt_surf(uu, vv, values, (us, vs, zs), show_dest=True, show_traj=True)
```



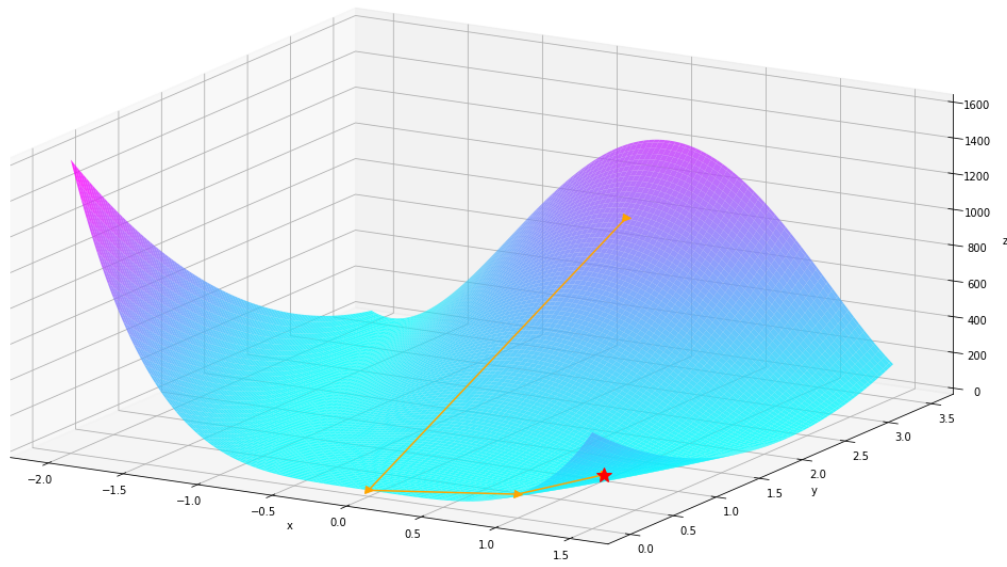
4.3.2 autodiff.optimize.newton

We'll then use Newton's method to find the minimum. The Newton's method is implemented in `autodiff.optimize.newton`. Here we set the argument `return_history=True` to return a whole history of optimization.

```
In [102]: hist = opt.newton(f, init_val_dict={x: 0.0, y: 3.0}, max_iter=10000,
                        return_history=True)
```

We can plot our optimization path as below. The Newton's method makes use of second-derivative information. We can see that the Newton's method takes much fewer steps to reach the minimum.

```
In [103]: hist = np.array(hist)
          us, vs = hist[:, 0].flatten(), hist[:, 1].flatten()
          zs = f.evaluation_at({x: us, y: vs})
          plt_surf(uu, vv, values, (us, vs, zs), show_dest=True, show_traj=True)
```



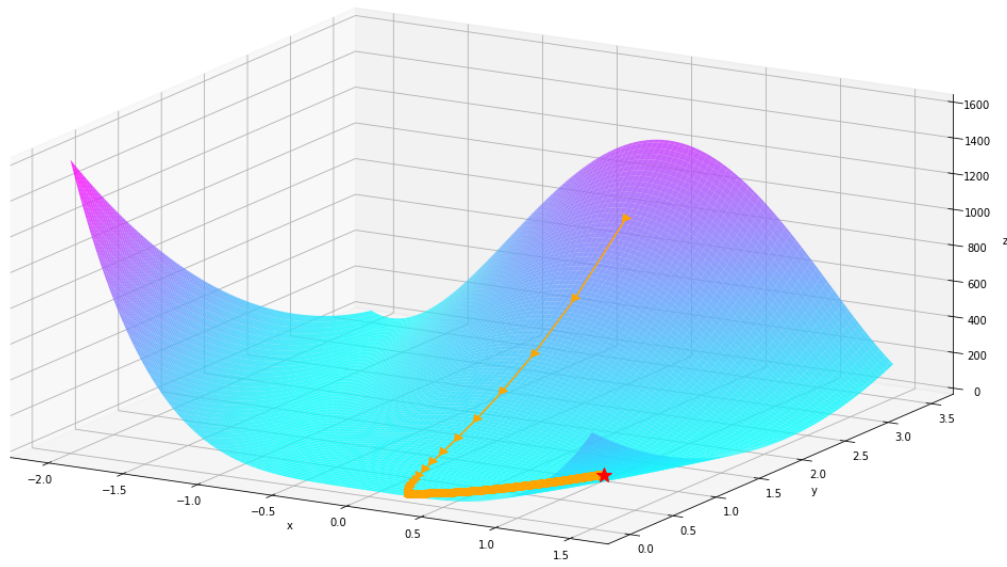
4.3.3 autodiff.optimize.gradient_descent

Now let's look at the `gradient_descent` method, unlike Newton's method, one does not need the Hessian matrix to find the minimum, while the trade off is that the algorithm might stuck in local minimum and takes more iteration.

```
In [104]: hist = opt.gradient_descent(f, init_val_dict={x: 0.0, y: 3.0}, max_iter=10000,  
                                     return_history=True)
```

We see gradient descent took a lot more steps than newton's method.

```
In [105]: hist = np.array(hist)  
us, vs = hist[:, 0].flatten(), hist[:, 1].flatten()  
zs      = f.evaluation_at({x: us, y: vs})  
plt_surf(uu, vv, values, (us, vs, zs), show_dest=True, show_traj=True)
```

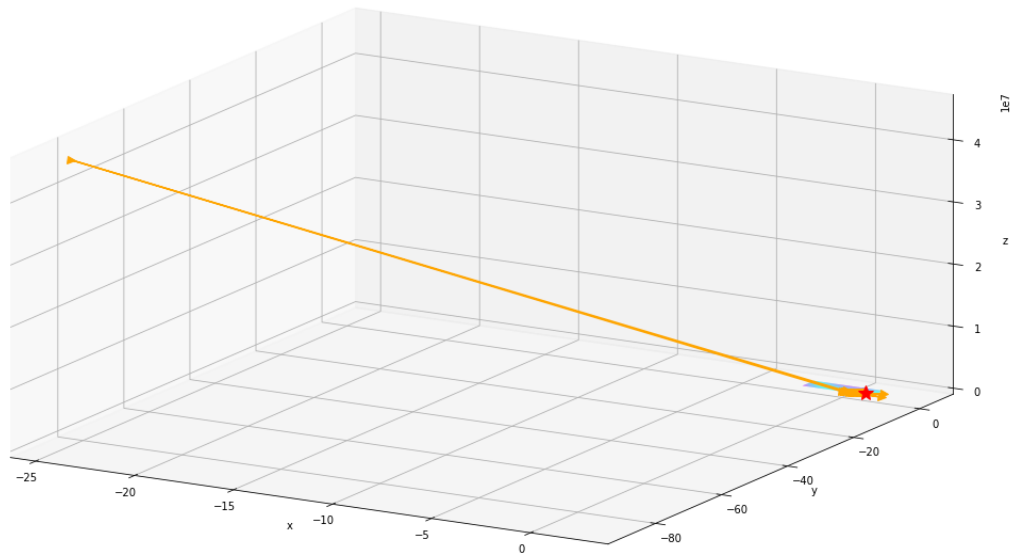
4.3.4 autodiff.optimize.bfgs

Lastly, we'll use BFGS to find the minimum. BFGS is a quasi-Newton method that approximates the Hessian matrix while doing the optimization. The optimization path of BFGS can be quite hysterical, so we'll just show the optimization result. It is $[1.0, 1.0]$ as we expected.

```
In [106]: res = opt.bfgs(f, init_val_dict={x: 0.0, y: 3.0})
In [107]: print(res[x], res[y])
1.000000000001 1.000000000001
```

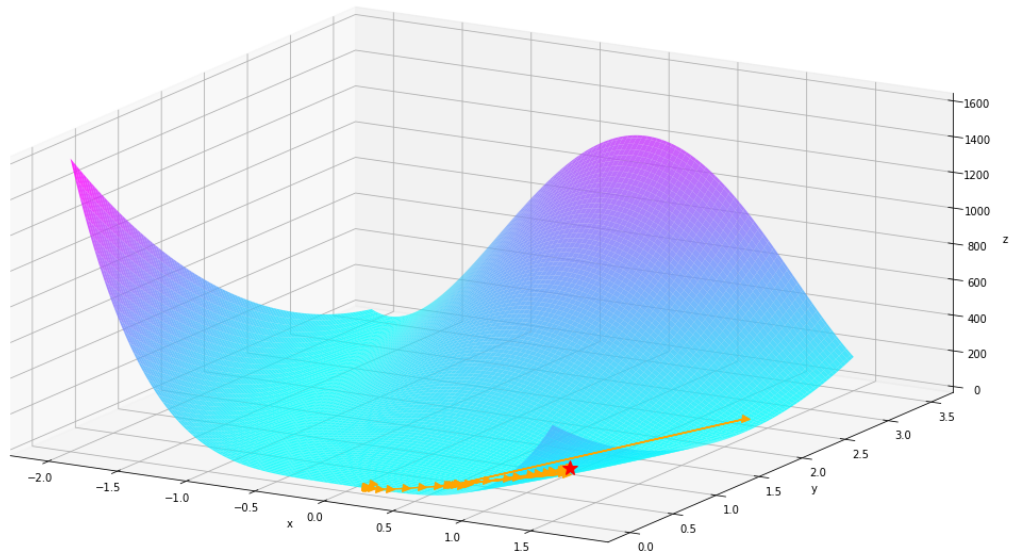
Let's look at the plot for bfgs, we see it blows up before it get to the minimum

```
In [108]: hist = opt.bfgs(f, init_val_dict={x: 0.0, y: 0.0}, max_iter=10000,
                        return_history=True)
hist = np.array(hist)
us, vs = hist[:, 0].flatten(), hist[:, 1].flatten()
zs = f.evaluation_at({x: us, y: vs})
plt_surf(uu, vv, values, (us, vs, zs), show_dest=True, show_traj=True)
```



Let take a closer look by excluding the very large value in the first few iterations

```
In [109]: hist_trim = hist[5:,:]
In [110]: us, vs = hist_trim[:, 0].flatten(), hist_trim[:, 1].flatten()
          zs = f.evaluation_at({x: us, y: vs})
          plt_surf(uu, vv, values, (us, vs, zs), show_dest=True, show_traj=True)
```



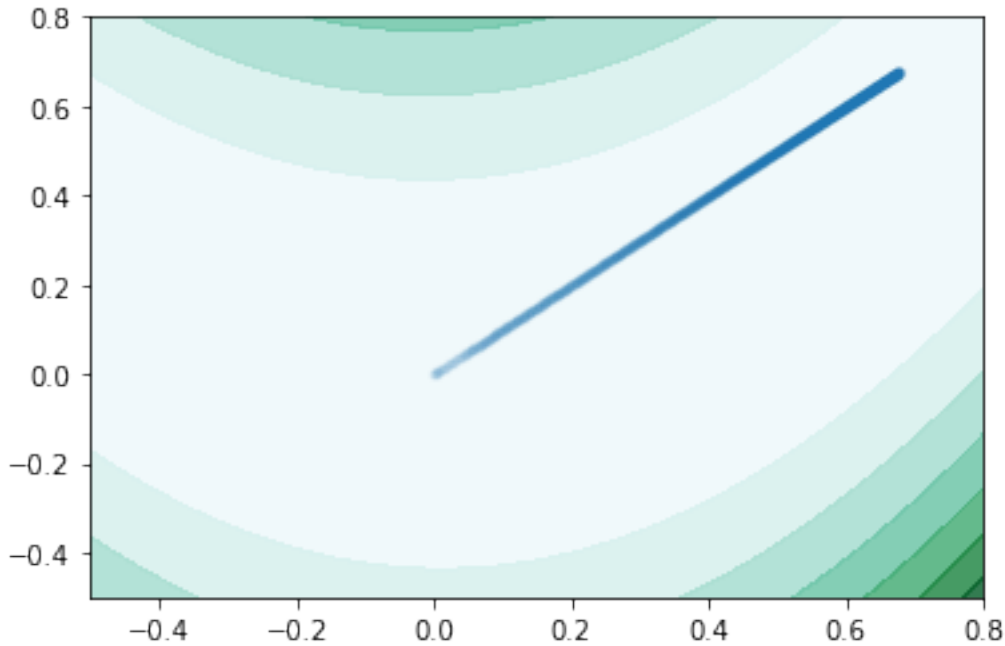
4.4 autodiff.plot

Plot function takes in a single expression, which only has two subcomponent. It then use either Newton's Method or Gradient Descent to calculate the minimum of the given function. It plots the values of the function at different points

in a contour map, with ranges specified by the user, and highlights the trajectory of the optimization algorithm reaching the minimum.

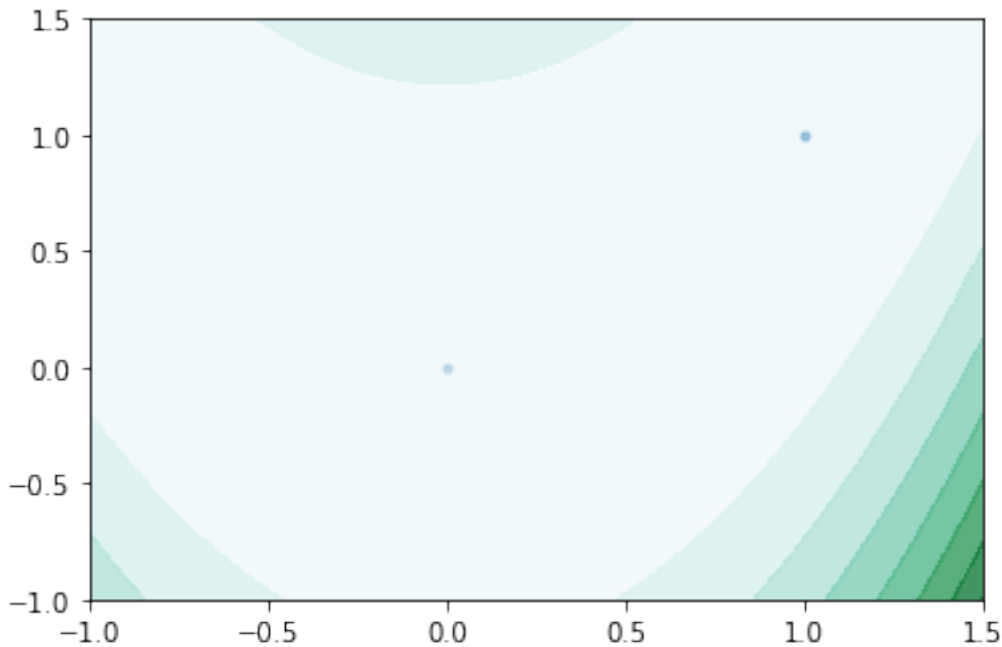
```
In [111]: import autodiff.forward as fwd
import autodiff.optimize as opt
from autodiff.plot import plot_contour

In [112]: x, y = fwd.Variable(), fwd.Variable()
f = 100.0*(y - x**2)**2 + (1 - x)**2.0
init_val_dict = {x: 0.0, y: 1.0}
plot_contour(f, init_val_dict, x, y, plot_range=[-0.5, 0.8], method = "gradient_descent")
```



We see that newton method merely used 2 iteration

```
In [113]: plot_contour(f, init_val_dict, x, y, plot_range=[-1, 1.5], method = "newton")
```



4.5 autodiff.backprop

```
In [134]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from autodiff.backprop import *
from autodiff.forward import *
from autodiff.rootfinding import *
import time
```

Backpropagation module is built upon the interfaces developed in central code file “Autodiff.forward”. It calculate the derivative of each nodes in the computational graph with respect to the root nodes. Therefore with different root nodes, we should expect to see different values of derivative. suppose we have the following structure:

$$x = 1, y = 2$$

$$c = \sin(x)$$

$$d = c \cdot y$$

Note: after one round of back propagation, the .bder attributes stores the answer from the last round until it is cleared when a new round is called upon.

```
In [135]: x = Variable()
y = Variable()
c = sin(x)
d = c*y
back_propagation(c, {x:1,y:2})
print('derivative of x with respect to c is ', x.bder)
print('derivative of y with respect to c is ', y.bder)
back_propagation(d, {x:1,y:2})
print('derivative of x with respect to c is ', x.bder)
print('derivative of y with respect to c is ', y.bder)
```

```

derivative of x with respect to c is 0.540302305868
derivative of y with respect to c is 0
derivative of x with respect to c is 1.08060461174
derivative of y with respect to c is 0.841470984808

```

If we calculate by hand:

\$

$$\frac{dc}{dx} = \cos(1) = 0.54 \quad (4.1)$$

$$\frac{dc}{dy} = 0 \quad (4.2)$$

$$\frac{dd}{dx} = y * \frac{dc}{dx} = 2 * \cos(1) = 1.08 \quad (4.3)$$

$$\frac{dd}{dy} = c = \sin(1) = 0.84 \quad (4.4)$$

(4.5)

\$

Our Backward Mode is faster than Forward Mode when getting the derivatives of all nodes in a certain computational graph because of caching the results in the process.

User can use our backward mode to make their own neural network

```

In [136]: start1 = time.time()
          x = Variable()
          y = Variable()
          c = sin(x)
          d = cos(y)
          e = sin(x)*cos(y)
          f = tan(e)
          for i in range(10000):
              back_propagation(f, {x:1,y:2})
          endl = time.time()
          interval = endl-start1
          print('derivative of x with respect to f is ', x.bder)
          print('derivative of y with respect to f is ', y.bder)
          print('derivative of c with respect to f is ', c.bder)
          print('derivative of d with respect to f is ', d.bder)
          print('derivative of e with respect to f is ', e.bder)
          print('derivative of f with respect to f is ', f.bder)
          print('derivative of g with respect to f is ', g.bder)
          print('time taken is {} second'.format(interval))

```

```

derivative of x with respect to f is -0.254837416116
derivative of y with respect to f is -0.867211207612
derivative of c with respect to f is 0
derivative of d with respect to f is 0
derivative of e with respect to f is 1.1333910384
derivative of f with respect to f is 1
derivative of g with respect to f is 0
time taken is 0.43090200424194336 second

```

```

In [137]: start2 = time.time()
          for i in range(10000):
              forward_x = f.derivative_at(x, {x:1,y:2})
              forward_y = f.derivative_at(y, {x:1,y:2})
              forward_c = f.derivative_at(c, {x:1,y:2})
              forward_d = f.derivative_at(d, {x:1,y:2})

```

```
        forward_e = f.derivative_at(e, {x:1, y:2})
        forward_f = f.derivative_at(f, {x:1, y:2})
        forward_g = f.derivative_at(g, {x:1, y:2})
    end2 = time.time()
    interval = end2-start2
    print('derivative of x with respect to f is ', forward_x)
    print('derivative of y with respect to f is ', forward_y)
    print('derivative of c with respect to f is ', forward_c)
    print('derivative of d with respect to f is ', forward_d)
    print('derivative of e with respect to f is ', forward_e)
    print('derivative of f with respect to f is ', forward_f)
    print('derivative of g with respect to f is ', forward_g)
    print(interval)

derivative of x with respect to f is  -0.254837416116
derivative of y with respect to f is  -0.867211207612
derivative of c with respect to f is  -0.0
derivative of d with respect to f is  -0.0
derivative of e with respect to f is  1.1333910384
derivative of f with respect to f is  1.0
derivative of g with respect to f is  -0.0
0.8440079689025879
```

Back propagation is also integrated with the function Newton's

Note that sine function have multiple roots, and newton's method will only give you the first one it finds

```
In [142]: result_d=newton_scalar(d, {x:1, y:-1}, max_itr = 25, method = 'backward')

In [143]: print('x:', result_d[x])
          print('y:', result_d[y])
          print('function value:', abs(d.evaluation_at({x:result_d[x], y:result_d[y]})))

x: 2.84112466652
y: -1.5707963268
function value: 5.91243550575e-13
```

5.1 High-level Design

5.1.1 Core Functions: Static Structure

The central data structure in `autodiff` are `Expression` and `ElementaryFunction` (which is the common interface shared by `Add`, `Mul`, `Pow`, `Exp`, `Sin`...). `Expression` represents a mathematical expression. It is composed of one `ElementaryFunction` plus two sub-`Expression`'s. `Expression` has two child class: `Variable`, which represents a 'base' variable and `Constant`, which represents a constant.

5.1.2 Core Functions: Dynamic Behavior

When a `Expression`'s `derivative_at` method is called, it will pass its sub-`Expression`(s) to the `ElementaryFunction`'s `derivative_at` method. `ElementaryFunction`'s `derivative_at` method will then compute the derivative based on chain rule. In this process, the `ElementaryFunction` will need the values and derivatives of the sub-`Expression`(s), so it will call the `evaluation_at` method and `derivative_at` method of the sub-`Expression`(s), and use the returned value to calculate the derivative. In other words, `Expression` and `ElementaryFunctions` will be calling each other recursively, until the base of this recursive process is reached.

The base of this recursive process lies in the `Constant` class and the `Variable` class. When a `Constant` is called to give its derivative, it returns 0. When a `Variable` is called to give its derivative, it checks whether itself is the variable to be taken derivative with respect of, if yes, then it returns 1.0, otherwise it returns 0.0.

5.1.3 On Second Order derivatives

The implementation of second order derivative is conceptually very similar to the implementation of first order derivative, except that it implements a different chain rule. The knowledge of the chain rule is encompassed within the `derivative_at` method of `ElementaryFunction`. Because all the `ElementaryFunctions` involves either one or two sub-`Expression`, the Faà di Bruno's formula is actually much less frightening to implement than it seems in the following figure.

5.2 Core Classes

The core class of autodiff is `Expression` and its child classes (`Variable` and `Constant`). They share the same interface: all implements their own `evaluation_at` and `derivative_at` methods. The dunder methods of `Expression` is overridden so that any operation on `Expression` will also return an `Expression`. `Variable` and `Constant` inherits these dunder methods so that they have the same behavior as `Expression`.

`Expression` is composed of one `ElementaryFunction` and two sub-Expressions. `ElementaryFunctions` like `Sin`, `Exp` and `Add` implements the chain rule associated with the corresponding elementary function. Note that `sin` and `exp` are different from `Sin` and `Exp`. The former two are actually factory functions that returns a `Expression` which has `Sin` and `Exp` as its `ElementaryFunction`.

5.3 External Dependencies

`autodiff` depends on `numpy`. All of `autodiff`'s calculation is done in `numpy` for the efficiency and the advantage of vectorization. The `optimize` module depends on `scipy` for solving linear systems. The `plot` module depends on `matplotlib` for plotting.

6.1 Autodiff.forward

6.2 Autodiff.backprop

6.3 Autodiff.rootfinding

6.4 Autodiff.optimize

6.5 Autodiff.plot

7.1 Directory Structure

The structure of `autodiff`'s project directory is as follows.

```
autodiff/  
  
    __init__.py  
    README.md  
    forward.py  
    backward.py  
    optimize.py  
    rootfinding.py  
    plot.py  
  
tests/  
  
    README.md  
    test_forward.py  
    test_backward.py  
    test_optimize.py  
    test_rootfinding.py  
    test_plot.py  
  
docs/  
  
    README.md  
    milestone1.ipynb  
    milestone2.ipynb  
    source/  
        Background.ipynb  
        Getting Started.ipynb  
        Implementation.ipynb  
        index.rst
```

(continues on next page)

(continued from previous page)

```
Installation.ipynb
Libraries_demo.ipynb
Future_Development.ipynb
License.rst

.gitignore
.travis.yml
LICENSE.txt
README.md
requirements.txt
setup.cfg
setup.py
```

The source codes lies in the directory `autodiff`, in which the `__init__.py` is there to make `autodiff` a package. The file `forward.py` contains the source code of forward mode autodifferentiation. The file `backward.py` contains the source code of backward mode autodifferentiation. The file `optimize.py` contains optimization routines built on top of autodifferentiation. The file `rootfinding.py` contains rootfinding routines. The file `plot.py` contains utility functions for plotting.

The test suites lies in the directory `tests`. The test files are named to represent the module that they test.

The documents lies in the directory `docs`. `milestone1.ipynb` is the history version of document when submitting milestone 1. `milestone2.ipynb` is the history version of document when submitting milestone 2. `document.ipynb`, which is this file itself, is the final document.

Other files in the root directory includes: `.gitignore`, which specifies the files that should not be tracked by git, `.travis.yml`, which is the configuration file for TravisCI, `LICENSE.txt`, which is the license for this package, `README.md`, which is the README file for this package, `requirements.txt`, which specifies the dependencies of this package, `setup.cfg`, which is the configuration file for installing this package, `setup.py`, which is the script for installing this package.

7.2 Modules

There are now five modules: `autodiff.forward` for forward mode autodifferentiation, `backward` for backward mode autodifferentiation, `optimize` for optimization, `rootfinding` for rootfinding, and `plot` for plotting.

7.3 Test Automation

The continuous integration is done by TravisCI and Coveralls.

7.4 Distribution

`autodiff` is distributed with [PyPI](#).

8.1 1. Optimization

One of the shortcomings we notice of our current design is that, during the calculation, the derivative/value of an expression at a single point may be evaluated multiple times. When the Expression tree is shallow, this doesn't have much effect on the computation time. However, when the Expression tree is deep, the time spent on redundant work will grow exponentially, which can be a serious problem. We may want to add a cache to our library, so that when the same derivative/value is queried, it is fetched from the cache instead of being computed again. This can largely accelerate our library in the case of complex Expressions.

8.2 2. Extensions

Since most machine learning problems can be formulated as optimization problems, and optimization routines can make use of automatic differentiations, we can actually develop a machine learning library on the top of our library. Other possible extensions include: more visualization tools, more optimization methods, even higher-order derivative than second-order, a neural network framework based on backward mode automatic differentiation.

8.3 3. Improvement

If user wish to add additional feature for the DFYS-autodiff package, please go to our [GitHub](#) repository, fork the repository, make the improvement, and submit pull request to us.

This project is hosted on [GitHub](#) and [PyPI](#).