Documentation

Git Fighters

1 Introduction

This project aims to enable automatic differentiation with Python. Automatic differentiation (AD) is a technique to find the derivative of a function computationally. Unlike some other techniques, automatic differentiation evaluates derivatives to machine precision, and it is less costly than symbolic differentiation. Differentiation is an important part of mathematical analysis, and differentiation is frequently used in machine learning, numerical analysis, economic optimization problems and more. This package is hosted on Github. Using this package requires no python knowledge thanks to the command-line interface.

2 Background

Enabling automatic differentiation involves several mathematical concepts and techniques. Brief overview can be found below:

2.1 Differentiation

The derivative of a function $y = f(x)$ at the point $P = (x_1, f(x_1))$ is the slope of the tangent line at that point and can be written as:

$f'(x_1) = lim_{\Delta x \to 0} \frac{f(x_1 + \Delta x) - f(x_1)}{\Delta x}$

Only if this limit exists is the function $y = f(x)$ is differentiable at point x_1 .

2.1.1 Rules of Differentiation

Multiple rules apply to differentiation:

Rule 1: Derivative of a constant function

If $f(x) = c$, with c constant, then $f'(x) = 0$.

Rule 2: Derivative of a linear function

If $f(x) = mx + b$, with m and b constants, then $f'(x) = m$.

Rule 3: Derivative of a power function

If $f(x) = x^n$, then $f'(x) = nx^{n-1}$.

Rule 4: Derivative of the constant multiple of a function

If $g(x) = cf(x)$, with c constant, then $g'(x) = cf'(x)$.

Rule 5: Derivative of the sum or difference of a pair of functions

If $h(x) = g(x) + f(x)$, then $h'(x) = g'(x) + f'(x)$, while if $h(x) = g(x) - f(x)$, then $h'(x) = g'(x) - f'(x)$.

Rule 6: Derivative of the sum of an arbitrary but finite number of functions

If $h(x) = \sum_{i=1}^{n} g_i(x)$, then $h'(x) = \sum_{i=1}^{n} g_i'(x)$.

Rule 7: Derivative of the product of two functions

If $h(x) = f(x)g(x)$, then $h'(x) = f'(x)g(x) + f(x)g'(x)$

Rule 8: Derivative of the quotient of two functions

If $h(x) = \frac{f(x)}{g(x)}$, g(x) ≠ 0, then $h'(x) = \frac{f'(x)g(x) - f(x)g'(x)}{[g(x)]^2}$.

Rule 9: Derivative of a function of a function (chain rule)

If $y = f(u)$ and $u = g(x)$ so that $y = f(g(x)) = h(x)$, then $h'(x) = f'(u)g'(x)$ or

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

Rule 10: Derivative of the inverse of a function

If $y = f(x)$ has the inverse function $x = g(x)$, that is, if $g(y) = f^{-1}(y)$ and f'(x) ≠ 0, then

$\frac{dx}{dy} = \frac{1}{dy/dx}$ or $g'(x) = \frac{1}{f'(x)}$ where y = f(x).

Rule 11: Derivative of an exponential function

If $y = e^x$, then $dy/dx = e^x$.

Rule 12: Derivative of a logarithmic function

If $y = ln(x)$, then $dy/dx = 1/x$.

2.1.2 Partial Derivatives

Functions with more than one variable have more than one derivative. A function of n variables, $y = f(x_1, x_2, \ldots, n_n)$, has n partial derivatives $f_1, f_2, \ldots f_n$. The partial derivatives can be arranged in a column or row vector called the gradient (∇_x denotes the gradient with respect to x).

For scalar-valued functions the gradient and the Jacobian matrix are identical. For vector-valued functions, the Jacobian matrix is defined as:

$$\boldsymbol{J}_{ij} = \frac{\partial f_i}{\partial x_j} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \cdots & \cdots & \cdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}.$$

The entries of the Jacobian matrix are all points in which a vector-valued function is differentiable.

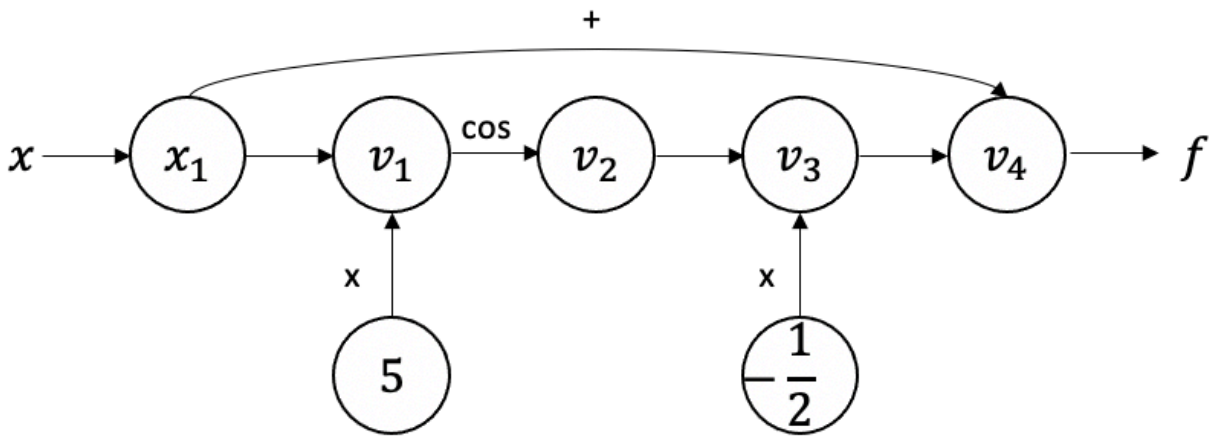
2.1.3 Gitfighters

The gitfighters package adheres to the above mentioned mathematical rules and implements them all in code.

2.2 Graph Structure

The graph structure visualizes the elementary operations performed to evaluate a function at a point (x). The point x is the input value, and f is the output value. The nodes (v_i) illustrate the sequence of operations necessary to compute the output value. Each node is labeled accordingly (like times 5 or cos). The example below illustrates a graph:

$$f(x) = x - \frac{1}{2}\cos(5x)$$



The example below illustrates the corresponding evaluation table:

$$f(x) = x - \frac{1}{2}\cos(5x)$$

Trace	Elementary Operation	Derivative	Numerical Values
x_1	1	1	(1, 1)
v_1	$5x_1$	$5\dot{x}_1$	(5, 5)
v_2	$\cos(v_1)$	$-\sin(v_1)\dot{v}_1$	(0.2836, 4.7946)
v_3	$-\frac{1}{2}v_2$	$-\frac{1}{2}\dot{v}_2$	(-0.1418, -2.3973)
v_4	$x_1 + v_3$	$\dot{x}_1 + \dot{v}_3$	(0.8582, -1.3973)

2.3 Elementary Functions

This package supports several elementary functions. The rules of differentiation list most elementary functions, such as quadratic, exponential, or logarithmic functions, but (inverse) trigonometric, (inverse) hyperbolic, or logistic functions are also supported. The comparison operators equal and not equal can be applied, too.

3 Software Organization

3.1 Directory Structure

The outline below presents the directory structure and the main modules that our package contains:

```
cs107-FinalProject/
    docs/
        milestone1.md
        milestone2.ipynb
        documentation.ipynb
        ...
    gitfighters/
        __init__.py
        git_fighters.py
        vector.py
        ...
    tests/
        __init__.py
        test_AD.py
        test_fightingAD.py
        test_elementary_operations.py
        ...
    .gitignore
    .travis.yml
    requirements.txt
    LICENSE
    README.md
    ...
```

3.2 Basic Modules Implementation and Their Functionality

- **git_fighters.py:** This file contains the `fightingAD()` class, which is the core of our custom automatic differentiation library. The `fightingAD()` class represents a scalar and its derivative. Users can perform various operations on it since most of its dunder methods were overwritten, to enable summation, multiplication etc... Additional elementary operations, e.g. trigonometrical or logarithmic operations, have been defined out of the scope of the class and can be used on any class instance. The User is not supposed to use this class directly. The AD class provides more functionality and is easier to use.
- **vector.py:** This file contains the `AD()` class. The `AD()` class enables functions with multiple input. Users can pass n variables and n x n derivatives (optional) to this class. The functions `differentiate` and `evaluate` allow users to retrieve the derivative vector or Jacobian and the values of the object. Users can also iterate over items in this class, perform operations on them, or compare objects.
- **cli.py:** This file is an executable file, containing the user friendly interactive interface.
- **parsing.py:** This file contains all the functions and the pipeline for string cleaning, parsing and finding the equations.
- **visualize.py:** This file contains functions creating graphs of the functions incl. the value of the derivative

at the point defined by the users.
- **latex.py:** This file contains a function to create a LaTeX file displaying the derivative or Jacobian as well as a graph of the function.

3.3 Tests and Coverage

Gitfighters uses Pytest to ensure our code works correctly:

- **test_fightingAD.py:** Tests for the `fightingAD()` class and its functions run by pytest.
- **test_AD.py:** Tests for the `AD()` class and its functions run by pytest.
- **test_parsing.py:** Tests for the string parsing functions run by pytest.
- **test_visualize.py:** Tests for the plotting functions run by pytest.
- **test_latex.py:** Tests for the LaTeX file creation function run by pytest.

Gitfighters also uses codecov and TravisCI to ensure our project functionality is maintained even through development:

(https://travis-ci.com/Git-fighters/cs107-FinalProject)

(undefined)

3.4 Package Distribution

We will use Python Package Index (PyPI) to build, upload and distribute our package. Python Package Index is an official third-party ecosystem for distribution of Python packages in the **sdist** format (Pure-Python packages) or in the **wheels** format (packages with Compiled Artifacts).

As of Milestone 2, however, our package is not yet available on Pypi. Instead a user can copy the existing repository and continue from there:

```
git clone https://github.com/Git-fighters/cs107-FinalProject.git
```

3.5 Software Packaging

We will use **wheels**, a package format designed to ship libraries with compiled artifacts. The **wheels** package format is the most common tool for distributing reusable code, libraries between developers. Our implementation will not use a framework to distribute our software, since **frameworks** are primarily used for development and packaging of common python applications (web site backends, dynamic web frontends and mobile clients) targeting a non-developer audience.

4 Implementation

The gitfighters implementation is based on two classes: the base class `fightingAD()` and its wrapper class, `AD()`. Every variable in a function is represented as a `fightingAD()` instance internally, however, user access them from the parent wrapper class (`AD()`), like so:

```
variables = AD([1,2])  f = variables[0]**2 - sin(variables[1])
```

4.1. Classes and functions

1. **Class `fightingAD()`** - This class acts as a placeholder for a variable. The main aim of this class is to compute the values and the derivatives, and to carry them over through each elementary operation. The argument passed to the class represents the point at which the function shall be evaluated. Instances of this class store following values:

   - value (val): value of the function evaluated at at the given point.
   - derivative (der): evaluated derivative at the given point.

   Furthermore, the `fightingAD()` class overrides following dunder methods:

- __str__
- __repr__
- __eq__
- __ne__
- __neg__
- __pos__
- __abs__
- __mul__
- __rmul__
- __add__
- __radd__
- __sub__
- __rsub__
- __pow__
- __rpow__
- __trudiv__
- __rtruediv__

   In our overridden functions we use self.val and self.der for evaluating the value and the derivative of the returned object. An example implementation:

```
def __add__(self, other):
  try:
      return fightingAD(self.val + other.val, self.der + other.der)
  except AttributeError:
      try:
          return fightingAD(self.val + other, self.der)
      except:
          raise TypeError(
              "unsupported operand type(s) for +: {} and {}".format
(
                  type(self).__name__, type(other).__name__
              )
          )
  else:
      raise Exception("unsupported operation for +")
```

2. **Elementary Functions** - Additionally, we define trigonometric, logarithmic and other types of functions separately, which can all be imported on their own. They all accept either `fightingAD()` instances, or vectors/scalars.

    - sin
    - cos
    - tan
    - sinh
    - cosh
    - tanh
    - arcsin
    - arccos
    - arctan
    - arcsinh
    - arccosh
    - arctanh
    - log
    - sqrt
    - exp
    - sigmoid

3. **Class `AD()`** - This class acts as a wrapper for the `fightingAD()` class . The main motivation behind the '''AD''' class is to enable easy multi is to compute the values and the derivatives of multivariable functions while utilizing our base class fightingAD elementary operations. AD is also an iterable enabling user to index the instantiated variables.

1. **Functions `evaluate(), differentiate()`** - These functions are the intended usage method to access the derivatives and values. They take as input either a `fightingAD` or a `AD` instance, and return corresponding values or derivatives. In case the user wants to evaluate a multi-output function, they can feed a list of functions into either `evaluate()` or `differentiate()` and then get the corresponding value vector and jacobian.

4.2. Dependencies

1. Gitfighters makes use of following external libraries and services:
   A. Travis Ci for tests
   B. Codecove for code coverage
   C. NumPy for mathematical operations and array data structures

5 Usage

5.1. Installation

Gitfighters can be installed through PyPI (https://pypi.org/project/gitfighters/0.0.1/) or by cloning our GitHub (https://github.com/Git-fighters/cs107-FinalProject) repository.

5.1.1. PyPI Installation

- Step 1: PyPI installation

```
pip install gitfighters
```

5.1.2. GitHub Installation

- Step 1: Create a virtual environment

```
conda create --name myenv
```

- Step 2: Activate the virtual environment

```
conda activate myenv
```

- Step 3: Clone the GitHub repository

```
git clone https://github.com/Git-fighters/cs107-FinalProject.git
```

- Step 4: Install dependencies using pip

```
pip install -r cs107-FinalProject/requirements.txt
```

- Step 5: Install gitfighters

```
cd cs107-FinalProject

python3 setup.py install
```

Once the modules are loaded, the user will be able to use the code as follows:

Evaluating a function like $f(x_0, x_1) = 200 + x_0^3 + sin(x_1)$ at points $(x_0, x_1) = [-2, 1]$:

In [22]:
```
from gitfighters.vector import *

# instantiate the trace variables
x = AD([-2,1])

# perform elementary operations on them
f = 200 + (x[0]**3) + sin(x[1])
# ...
#  all elementary operations done
```

User can then access the function value and derivative value by:

In [23]:
```
# fightingAD.val or fightingAD.der
print(evaluate(f), differentiate(f))
```

```
192.8414709848079 [12.          0.54030231]
```

Alternatively, user can skip doing all the elementary operations manually and just define a function and let our library do all the work, as follows:

In [24]:
```
def func(x):  # example mathematical function
    return 200 + (x[0]**3) + sin(x[1])

x = AD([-2,1])  # instantiate ad variable at the point at which where
we want func() evaluated

f = func(x)

# Now the user can access the function values and derivates automatica
lly:
print(evaluate(f), differentiate(f))
```

```
192.8414709848079 [12.          0.54030231]
```

This will also be doable with vector inputs/outputs.

The above example is somewhat trivial. Our library also enables more complex functionality. One appropriate example is Newton's method, where the roots of a function are found by procedurally following a functions derivative until 0 is reached.

Here is a great visualization of the procedure:

https://en.wikipedia.org/wiki/File:NewtonIteration_Ani.gif
(https://en.wikipedia.org/wiki/File:NewtonIteration_Ani.gif)

We can define this functionality in python as follows:

In [25]:
```
def newton(f, x=1):
    # finds the root of a function f with Newton's method
    # using our gitfighters library
    # inputs: function f, optional starting point x

    x = fightingAD(x)
    st_condition = 1

    # Newton's method main loop
    while st_condition > 1e-16:
        y = f(x)
        xval = x - y.val/y.der
        x_old = np.copy(x.val)
        x = fightingAD(xval.val)
        st_condition = np.abs(x-x_old).val

    return x.val
```

Lets use this to find the root of a simple linear function, like f(x) = x + 1

In [26]:
```
def f(x):
    return x + 1

root = newton(f)
print(f'root of f is at x = {root}')
```

```
root of f is at x = -1.0
```

Cool! Just remember to ignore x.der in this case. However, this function was relatively simple, and this doesn't seem particularly useful. How about we apply it to a real world use case?

5.1. A real world use case

Professor Sondak has a particular taste: he likes to drink starbucks coffee, but at specific temperature of 45°C (or 113°F). He also likes to have it ready for just when he starts class. However, often times it is still too hot and he gets his tongue burned, and other times he has let it sit too long and it has become too cold.

If we assume a constant room temperature of 21°C, how many minutes before class should Professor Sondak order his coffee for that perfect temperature?

Answer: This is a great problem to use automatic differentiation on! Let's do some calculations.

We can model the temperature in the cup by following equation (modeling conduction loss and ignoring, for simplicity, the small amount of radiative and convection loss):

$T_{\text{coffee}}(t) = T_{room} + (T_{\text{coffee at time 0}} - T_{room}) * e^{-t/\tau}$ where $\tau = \frac{d*m*c_h}{k*A}$

[Reference (http://web.mit.edu/21w.732-esg/www/handouts/729_simplified_model_of_heat_loss_in_a_coffee_cup.pdf)](http://web.mit.edu/21w.732-esg/www/handouts/729_simplified_model_of_heat_loss_in_a_coffee_cup.pdf)

A definition of variables:

- c_h is the heat capacity of coffee (4.184 Joules)
- k is the thermal conductivity of the cup material (we assume cardboard, so ~0.05)
- d is the thickness of the cup (assume 0.5cm)
- A is the surface area of the cup (assuming r=3 and h=15, this gives an area of $2 * pi * r^2 + 2pi * r * 15 = 0.0339m^2$)
- m is the mass of the coffee (assuming the previous dimensions and with density of water 1000kg/m3, then we get $pi * r^2 * 0.15 * 1000 = 0.42kg$)

With all that specified, let's start using the gitfighters library to solve this problem!

In [27]:
```
from git_fighters import *
import numpy as np

# specify the temperature function given values derived above:
def temperature_coffee(t, t_initial=100, t_room=21, c_h=4.184, m=0.42,
k=0.05, d=0.005, A=0.0339,):
    tau = (d*m*c_h) / (k*A)
    t_coffee = t_room + (t_initial - t_room)*np.e**((-t)/(tau))
    return t_coffee

def celsius_to_fahrenheit(t):
    return (t*(9/5)) + 32

def fahrenheit_to_celsius(t):
    return (t - 32)*(5/9)
```

In [28]:
```
# Lets find out the temperature gradient:

# 1. We create an automatic differentiation instance at t=0
t = 1
ad =  fightingAD(t)

# 2. We apply our previously defined function to our AD instance:
temp = temperature_coffee(ad)

# 3. Now we can check the derivative:
print(f'At t={t} mins, the temperature has dropped to {"{:.2f}".format
(temp.val)}°C and \
the temperature gradient is {"{:.2f}".format(temp.der)}°C/min')
```

```
At t=1 mins, the temperature has dropped to 86.14°C and the temperatu
re gradient is -12.57°C/min
```

With this functionality in place, we can now make use of newtons method (which requires derivatives provided by our library) to efficiently find the root of this problem:

In [29]:
```
def newton(f, x=1):
    # finds the root of a function f with Newton's method
    # using our fightingAD library
    # inputs: function f, optional starting point x

    x = fightingAD(x)
    st_condition = 1

    # Newton's method main loop
    while st_condition > 1e-16:
        y = f(x)
        xval = x - y.val/y.der
        x_old = np.copy(x.val)
        x = fightingAD(xval.val)
        st_condition = np.abs(x-x_old).val

    return x.val

# professor sondak likes his coffee at 45 degrees C
def perfect_temp(t):
    return temperature_coffee(t) - 45
```

In [30]:
```
# and now we find the root:
f = perfect_temp
root = newton(f)
print(f'root is at {root}')
```

```
root is at 6.1758492247475205
```

And there we have it! Professor Sondak should drink his coffee after exactly 6.1758492247475205 minutes. We have determined the point at which his coffee is at the ideal temperature to machine precision! (nevermind that our physical assumptions are much less precise ^^)

Here is a small visualization:

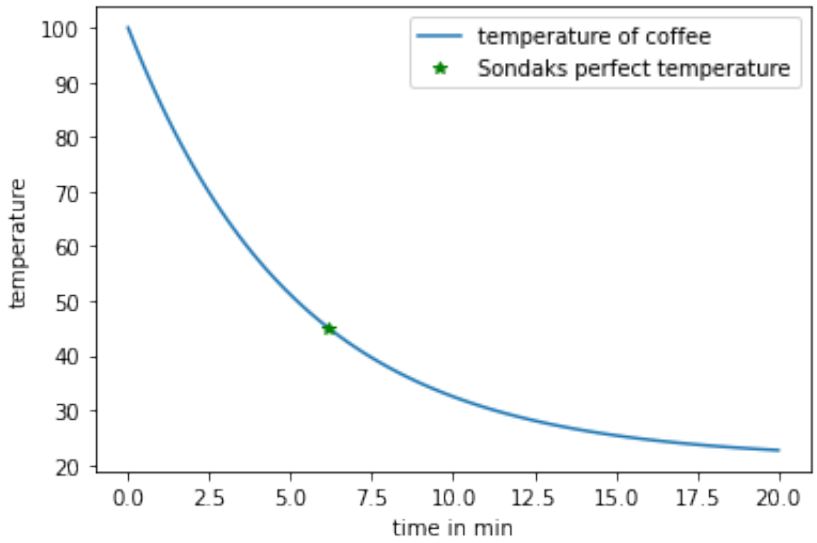
In [31]:
```
import matplotlib.pyplot as plt

# plot temperature curve
xs = np.linspace(0,20,100)
ys = temperature_coffee(xs)
plt.plot(xs, ys)

# plot horizontal line
# xs = np.linspace(0,20,100)
# ys = np.full(100, 45)
# plt.plot(xs, ys, 'r')

# plot the perfect point using newtons method
plt.plot(newton(f), temperature_coffee(newton(f)), 'g*')

plt.xlabel('time in min')
plt.ylabel('temperature')
plt.legend(['temperature of coffee', 'Sondaks perfect temperature'])
plt.show()
```



5.2. Optimization use case, Rosenbrock's Function

- Using gitfighters' automatic differentiation library.
- Using symbolic differentiation.

A well-known benchmark problem for optimization algorithms is minimization of Rosenbrock's function.

$$f(x, y) = 100(y - x^2)^2 + (1 - x)^2$$

This function has a global minimum of 0 at $(x, y) = (1, 1)$.

In this part we will compare the performance of the optimization algorithm using two distinct differentiation approaches (gitfighters' automatic differentation library and symbolic differentiation) to perform gradient descent on Rosenbrock's function, starting at $x, y = [-1, 1]^T$. Having performed gradient descent, we plot the contours of Rosenbrock's function, as well as the optimization path followed by the gradient descent algorithm.

Using gitfighters' automatic differentiation library

In [32]:
```
def rosenbrockAD(x):
    return (100*(x[1]-x[0]**2)**2+(1-x[0])**2).val
```

In [33]:
```
gradientAD = lambda x: (100*(x[1]-x[0]**2)**2+(1-x[0])**2).der
```

In [34]:
```
def r_lineAD(q,x1,y1,gx,gy):
    return rosenbrockAD(AD([x1-q*gx,y1-q*gy]))
```

In [38]:
```
def gradient_descentAD(x, max_it=2000, precision=1e-8):

    cur_xy=[x[0].val,x[1].val]
    position=[]
    position.append(cur_xy)

    previous_step_size = np.inf

    iters = 0

    values=[]
    values.append(rosenbrockAD(AD([cur_xy[0],cur_xy[1]])))

    norm_grad=[]
    func_eval = []

    while previous_step_size > precision and iters < max_it:
        prev_xy = cur_xy.copy()

        g=gradientAD(AD([prev_xy[0],prev_xy[1]]))
        ans=scipy.optimize.minimize(r_lineAD,0,args=(cur_xy[0],cur_xy[
1],g[0],g[1]),method='Nelder-Mead').x[0]
        cur_xy = np.asarray(cur_xy) - ans*gradientAD(AD([prev_xy[0],pr
ev_xy[1]]))
        position.append(cur_xy)
        values.append(rosenbrockAD(AD([cur_xy[0],cur_xy[1]])))
        change_square_1=(cur_xy[0] - prev_xy[0])**2
        change_square_2=(cur_xy[1] - prev_xy[1])**2
        previous_step_size = np.sqrt(change_square_1+change_square_2)

        iters+=1

    if iters<max_it:
        print(f"Number of iterations: {iters}\nThe local minimum occur
s at x,y: {cur_xy}")
    if iters==max_it:
        print(f"Reached the maximum number of iterations: {iters}.\nTh
e current minimum occurs at x,y: {cur_xy}")


    return cur_xy, values, previous_step_size, position
```

In [42]:
```
import scipy
cur_xyAD, valAD, prev_step_sizeAD, posAD= gradient_descentAD(AD([-1, 1
]))
```

```
Reached the maximum number of iterations: 2000.
The current minimum occurs at x,y: [0.9624918  0.92631697]
```

Using symbolic differentiation

In [43]:
```
def rosenbrock(x,y):
    return 100*(y-x**2)**2+(1-x)**2
```

In [44]:
```
partial_x = lambda x,y: 2*(100*(y-x**2)*(-2*x)+x-1)
partial_y = lambda x,y: 200*(y-x**2)
gradient = lambda x,y: np.asarray([partial_x(x,y), partial_y(x, y)])
```

In [45]:
```
def r_line(q,x,y,gx,gy):
    return rosenbrock(x-q*gx,y-q*gy)
```

In [46]:
```
def gradient_descent(x, y, max_it=2000, precision=1e-8):

    cur_xy=[x,y]
    position=[]
    position.append(cur_xy)

    previous_step_size = np.inf

    iters = 0

    values=[]
    values.append(rosenbrock(cur_xy[0],cur_xy[1]))

    norm_grad=[]

    while previous_step_size > precision and iters < max_it:
        prev_xy = cur_xy.copy()

        g=gradient(prev_xy[0],prev_xy[1])
        ans=scipy.optimize.minimize(r_line,0,args=(cur_xy[0],cur_xy[1]
,g[0],g[1]),method='Nelder-Mead').x[0]
        cur_xy = np.asarray(cur_xy) - ans*gradient(prev_xy[0],prev_xy[
1])
        position.append(cur_xy)
        change_square_1=(cur_xy[0] - prev_xy[0])**2
        change_square_2=(cur_xy[1] - prev_xy[1])**2
        previous_step_size = np.sqrt(change_square_1+change_square_2)

        iters+=1

    if iters<max_it:
        print(f"Number of iterations: {iters}\nThe local minimum occur
s at x,y: {cur_xy}")
    if iters==max_it:
        print(f"Reached the maximum number of iterations: {iters}.\nTh
e current minimum occurs at x,y: {cur_xy}")


    return cur_xy, values, previous_step_size, position
```

In [47]:
```
cur_xy, val, prev_step_size, pos= gradient_descent(x=-1, y=1)
```

```
Reached the maximum number of iterations: 2000.
The current minimum occurs at x,y: [0.9624918  0.92631697]
```

Comparing the optimization path's followed using the two distinct differentiation techniques

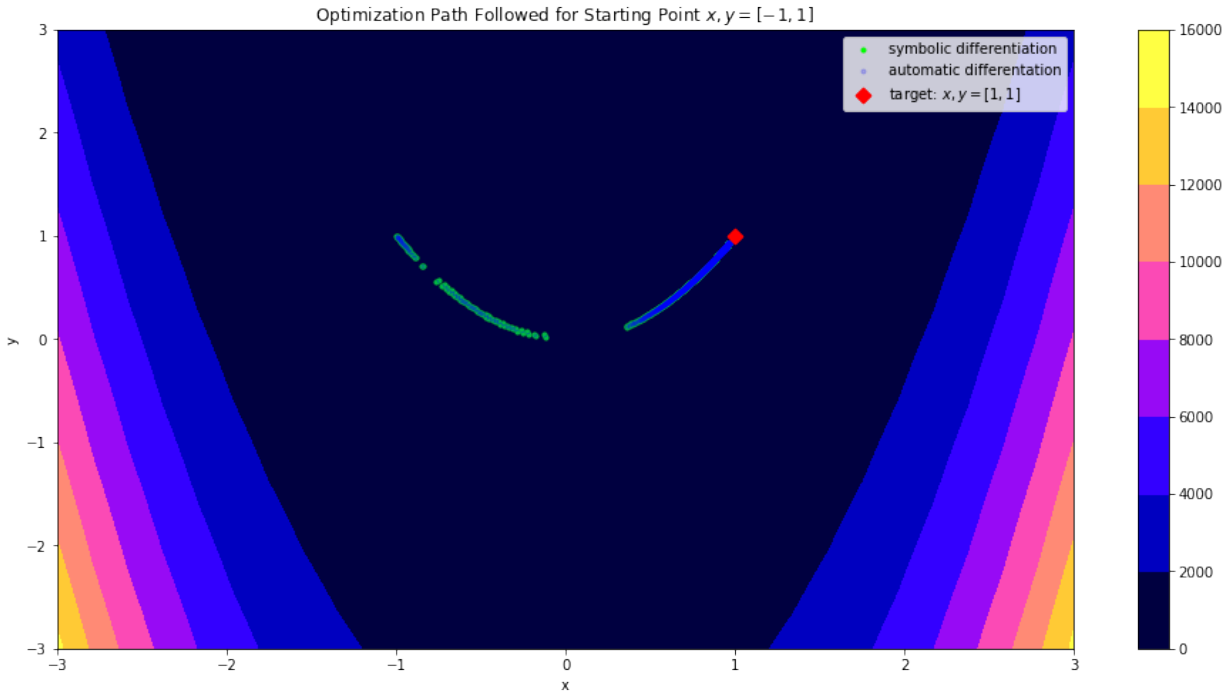
In [48]:
```
scat_xAD=[]
scat_yAD=[]
for i in range(len(posAD)):
    scat_xAD.append(posAD[i][0])
    scat_yAD.append(posAD[i][1])
```

In [49]:
```
scat_x=[]
scat_y=[]
for i in range(len(pos)):
    scat_x.append(pos[i][0])
    scat_y.append(pos[i][1])
```

In [50]:
```
x_ax=np.linspace(-3,3,30)
y_ax=np.linspace(-3,3,30)

Z_p=np.zeros((30,30))
X_p=np.zeros((30,30))
Y_p=np.zeros((30,30))
for i in range(30):
    for j in range(30):
        Z_p[i][j]=100*(y_ax[j]-x_ax[i]**2)**2+(1-x_ax[i])**2
        X_p[i][j]=x_ax[i]
        Y_p[i][j]=y_ax[j]

fig,ax=plt.subplots(figsize = (16,8))
cp = ax.contourf(X_p,Y_p,Z_p, cmap = 'gnuplot2')
fig.colorbar(cp)
ax.set_title('Optimization Path Followed for Starting Point $x, y = [-1,1]$')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.scatter(scat_x,scat_y, marker='.', color='lime', label = 'symbolic differentiation')
ax.scatter(scat_xAD,scat_yAD, marker='.', color='blue', alpha = 0.2, label = 'automatic differentation')
ax.scatter(1,1, marker = 'x', color = 'red', linewidths=8, label = 'target: $x, y = [1,1]$')
ax.legend(loc='best')
plt.show()
```



From the figure above, we observe that the optimization path using gitfighters' automatic differentiation library and using symbolic differentiation is identical.

6 Extension

To extend and make computation tools applicable to general, non-coder population our team decided to enable an alternative usage for our gitfighters package which tries to understand the human-friendly mathematics language. The main idea is to allow the users to define the functions with their variables as a string statement. The package extension utilizes string parsing techniques to find the equation, variables and their values to instantiate the AD objects under the hood and execute the user-desired equations. Additionally, to make it more useful for reproducible research, our extension allows the user to graph the equations and output a LaTeX file (.tex) with all the desired equations/graphs.

6.1. The Goal

While python has very set way of defining the mathematical functions we know that humans vary with their individual approaches to the equation. You might encounter casually *x^2* which python would have required to be *x**2* instead. Similarly, we like to write *3 + 6xy* when the pythonic way of defining it is *3 + 6*\x*y *. Furthermore, instantiating the variables with AD object and self-defining the equations might not be the most intuitive way of using computational tools for non-coder population when they think of the equation in their head as: " x + sin(xy) - 3*y where x is 2 and y is 4". Moreover, people who are largely involved in the research favor LaTeX outputted papers but if they are not as skilled in creating .tex files, generating equations and debugging them takes a lot of time when that time could be spent on making discoveries. Hence, with our functionalities we try to make our library accessible to the general population and save the precious time of the researchers.

6.2. The Assumptions

Our string parsing algorithm internally assumes that the user inputted strings will consist of reasonable equations and have variables and values clearly stated. For example:

reasonable input: 'x^2 + 3y when x is 4 and y is 5'

unreasonable input: '3 4 x^2 + 3y'

To keep our extension as simple and accessible as possible, we also assume that the users will use our extension couple of times if they need to evaluate/differentiate several functions.

6.3. The Technicalities

6.3.1. String Parsing

The String Parsing algorithm is stored in the *parsing.py* and the corresponding tests reside in *test_parsing.py*. The string parsing algorithm uses the already existing corpuses in **nltk** to tokenize and get rid of stop words in the user input. We also utilize the regex packages to clean and select the variables/values. The general flow of the parsing includes tokenizing the input, cleaning it, getting the variables, getting the values and separating the equation. As a last note we "pythonize" the string equation (make sure all the operations and variables are in python executable format), put the varaiables and values in a dictionary and return the equation with variable-value dictionary.

On the other end, the user interacts with this functionality in an interactive shell running the *cli.py* in the terminal after intalling our package. The user is prompted to input a reasonable equation with clear variables and values. Once the user inputs their request, we parse the string, instantiate an AD object with the requested values and create/execute the function. The users will also be prompted to answer if they want to generate graphs and tec files. Please read more about these functionalities in the section below.

6.3.2. Visualization

The results of the automatic differentiation process can be visualized. Users can generate graphs displaying the function, the point at which the users want to evaluate it, and the value of the derivative. Graphs are based on matplotlib.pyplot. Graphs will be saved in a subdirectory called 'graphs' under the name of the variable plus timestamp.

Users can also generated a tex files displaying the derivative or Jacobian (depending on the function). Text files also include graphs for the function. The tex files are based LaTeX. Users will be able to find the tex files in the same directory as 'jacobian_' plus timestamp.

6.4. Examples

Some string parsing examples:

```
In [58]: from gitfighters import parsing
         parsing.pipeline('x + y when x is 2 and y is 3')
```

```
Out[58]: ('x+y', {'x': 2.0, 'y': 3.0})
```

```
In [59]: parsing.pipeline('sin(x) + xy where x = 2 and y is 3')
```

```
Out[59]: ('sin(x)+x*y', {'x': 2.0, 'y': 3.0})
```

```
In [61]: parsing.pipeline('arctan(x) + 5y - xyz , where x = 2 , y is 3 and z is
         101')

Out[61]: ('arctan(x)+5*y-x*y*z', {'x': 2.0, 'y': 3.0, 'z': 101.0})
```

A screenshot example of the cli.py interecative session:

```
(cs107) talelokvenec@Tales-MacBook-Pro cs107_project_demos % cli.py
Please provide a function and corresponding point(s) at which to evaluate it.
EXAMPLE:   'x^2 - e^(y-1) when x=2 and y=5'
(x**2)+y*sin(x) where x is 2 and y is 1

We have evaluated your function! These are the derivative values:
x: 3.5838531634528574
y: 0.9092974268256817
    -->Would you like visualize your function and its derivative? Y/n
Y
graph saved as graphs/x20201212_161727_graph.png
graph saved as graphs/y20201212_161730_graph.png
    -->Would you like to output a nicely formatted latex file? Y/n
Y
latex file created under jacobian_20201212_161735.tex
(cs107) talelokvenec@Tales-MacBook-Pro cs107_project_demos % ls
graphs                          jacobian_20201212_161735.tex
(cs107) talelokvenec@Tales-MacBook-Pro cs107_project_demos % vi jacobian_20201212_161735.tex
```

A screenshot example of the graph and LaTeX file generation:

$\mathbf{sin(x) + 5y}$

$$\frac{\partial f}{\partial x_i} = \begin{bmatrix} 1 & 2 \end{bmatrix}$$