CS107/AC207 Final Project: Documentation

Matthew Hawes, Junyi Guo, Jack Scudder, Paul Tembo, Arthur Young

Due: 12:00 PM, December 12, 2020

1 INTRODUCTION

CMAutoDiff is a potential flow solver that is built around a library of auto-differentiation. The library allows users to create graphs to visualize a series of pre-defined potential flow problems. At it's core, CMAutoDiff computes derivatives in the forward mode. In computational engineering and data science, optimization problems are at the core of every challenge that individuals and teams in the field face. Finding derivatives is the common approach in dealing with optimization problems and sometimes it is hard and time-consuming to calculate the symbolic derivative of real-life equations. Automatic differentiation gives us the ease of solving the derivative of complex functions with accuracy to machine precision. In contrast with the difficulties in implementing and dealing with complex derivatives using symbolic differentiation and numerical differentiation, automatic differentiation provide us a more precise, efficient, and scalable way to compute derivatives when equations increase in complexity and number of inputs variables. Potential flow is a useful theory for solving simple fluid flow problems, and is a particularly well posed problem for an automatic gradient solver such as ours. This solver was made with students in mind, and further details about the application and visualization of potential flow are found in the sections below.

2 BACKGROUND

The following section summarized the two major components of mathematical theory motivating the development of our package. In the first subsection, we briefly describe the foundation of automatic differentiation as an algorithmic implementation of basic single variable calculus, and in the second we provide the basic problem statement behind solving potential flows.

2.1 Automatic differentiation

The basic principle of automatic differentiation can be understood as a systematic evaluation of the chain rule of differentiation. Let $x \in \mathbb{R}$, $y \in \mathbb{R}$ be mapped via the composite function $f : x \to y$ where f is defined by a sequence of evaluations of sub-functions as follows:

$$f := f_n(...(f_3(f_2(f_1(x))))) = y \tag{1}$$

the chain rule of differentiation gives us that:

$$\frac{dy}{dx} = \frac{d}{dx}(f) = \prod_{i=1}^{n} \frac{d}{d\omega_i}(f_i(\omega_i)) \tag{2}$$

where we take turns evaluating the derivative of each function $f_i(\omega_i)$. Thus, we can consider each function f_i to be a mapping from $\omega_i \to \omega_{i+1}$

Importantly, automatic differentiation does not yield an analytical expression for the derivative of a given function. Rather, if one considers the process of automatic differentiation in a black box format, then there are two inputs to automatic differentiation, the function form and the evaluation point. Consider a composite function $f : f_n(...(f_3(f_2(f_1(\omega_1)))))$ defined over the differential field Ω. We seek the value of $\frac{df}{dx}|_{x_0}$, where $x_0 \in \Omega$. We can view automatic differentiation as an operator $\mathcal{D} : f, x_0 \to \frac{df}{dx}|_{x_0}$.

There are two approaches one can take to apply the chain rule given by equation 2, and they are referred to by the terms *forward* and *reverse* accumulation. In forward accumulation, the evaluation of the derivative of composite function f is considered recursively from "inside out," meaning:

$$\begin{aligned}
\mathcal{D}_{x_0}\{f\} &:= \mathcal{D}_{x_0}\{f_{n-1}\} \frac{\partial f_n}{\partial \omega_n}\Big|_{f_{n-1}(\omega_{n-1})} \\
&= \mathcal{D}_{x_0}\{f_{n-2}\} \left(\frac{\partial f_{n-1}}{\partial \omega_{n-1}}\Big|_{f_{n-2}(\omega_{n-2})}\right) \left(\frac{\partial f_n}{\partial \omega_n}\Big|_{f_{n-1}(\omega_{n-1})}\right) \\
&\vdots \\
&= \frac{\partial f_1}{\partial \omega_1}\Big|_{x_0} \left(\prod_{i=2}^{n} \frac{\partial f_i}{\partial \omega_i}\Big|_{f_{i-1}(\omega_{i-1})}\right)
\end{aligned}$$

This implies a recursion where one starts at the lowest embedded sub function f_1 and systematically evaluates the expression outwards. The process would therefore be:

1. $\omega_{it} \leftarrow x_0$
2. $\dot{x} \leftarrow 1$
3. for i in range(n)
   $\omega_{it} \leftarrow f_i(\omega_{it})$
   $\dot{x} \leftarrow \dot{x} \cdot \frac{\partial f_i}{\partial \omega_i}|_{\omega_{it}}$

Contrast the above workflow with that of *reverse* accumulation: given the standard chain rule expression,

$$\begin{aligned}
\frac{\partial f}{\partial x}\Big|_{x_0} &= \left(\frac{\partial f_n}{\partial \omega_1}\Big|_{f_1(\omega_1)}\right)\left(\frac{\partial \omega_1}{\partial x}\Big|_{x_0}\right) = \left(\frac{\partial f_n}{\partial \omega_2}\frac{\partial \omega_2}{\partial \omega_1}\right)\Big|_{f_1(\omega_1)}\left(\frac{\partial \omega_1}{\partial x}\right)\Big|_{x_0} = ... \\
&= \left(\prod_{i=n}^{2} \frac{\partial \omega_i}{\partial \omega_{i-1}}\Big|_{f_{n-1}(\omega_{n-1})}\right)\left(\frac{\partial \omega_1}{\partial x}\Big|_{x_0}\right)
\end{aligned}$$

We note that the above implies a different order of operations. Instead of beginning with the innermost component of the composite function, we are instead differentiating the outermost expression with respect to the next outermost function, and then carrying until we reach the independent variable x.

2.2 Potential flow

In short, potential flow is an idealization of fluid mechanics made possible by treating the question of "what is the velocity of a moving fluid?" with continuum mechanics. Suppose one wished to know the exact trajectory of an infinitesimally small fluid "piece" in a 2D plane. Let the position of this "piece" of fluid be given by the position vector $\vec{p} \in \mathbb{R}^2$, and its velocity be given by $\vec{u} \in \mathbb{R}^2$. This piece of fluid, which we call a fluid "parcel" or equivalently "control volume" is surrounded by other similar fluid parcels, each with their own associated velocity and position vectors. Since the fluid parcels are continuously adjacent (i.e., we assume that there is no space between these parcels), we are describing a velocity *field*, where for every point $\vec{p}$, there is an associated velocity $\vec{u}$. This statement implies the existence of a function $f : \mathbb{R}^2 \to \mathbb{R}^2$, where $f(\vec{p}) = \vec{u}$. Potential flow theory endeavors to determine such a function.

Given such a problem statement, let us define a scalar valued function $\phi : \mathbb{R}^2 \to \mathbb{R}$. This function is called the *velocity potential*, and is directly analogous to electromagnetic potential or gravitational potential in other continuum mechanical models. One seeks to describe this function ϕ so that the evaluation of its *gradient* will yield the above described velocity field. If one assumes that the sought after velocity field is *irrotational*. The irrotational condition is mathematically expressed as:

$$\nabla \times \vec{u} = 0$$

Where $\nabla \times \vec{u}$ is called the *curl* of the velocity field $\vec{u}$. If this condition is satisfied, then $\exists \phi : \nabla \phi = \vec{u}$. Physically, the fluid must be *inviscid* for the curl to be zero, meaning that the aforementioned fluid parcels must not exert a frictional force on one another as they move past each other. If there was such a frictional force, one could imagine that if the parcels would "drag" their adjacent fluid parcels along as they move past each other. As it happens, the inviscid assumption is applicable for a variety of flow scenarios chiefly involving the (relatively) slow, steady movement of fluid past a solid surface, such as flow past an aerofoil, or a plane wing. But there are limitations to potential flow theory, and thus it is typically reserved for a finite set of flow scenarios in which the assumptions are accurate.

In our algorithmic implementation, we exclusively consider "free stream" and "external" flows in two dimensions. There are a number of derived potential flow equations ϕ that describe flows belonging to these two categories, and we have implemented some of them. A list of our implemented potential flow equations can be found in section 6, as well as further details on our product.

3 HOW TO USE

Installing Cache-Money404 is best done in a virtual environment this is because you should have **numpy version 1.16.3** and **matplotlib version 3.3.1** for it to work. You can create a virtual environment by following these steps:

1. conda create -n **env_name** python=3.7 anaconda
2. source activate **env_name**
3. install using pip install or by cloning the github repository
4. Git clone , using the following code:
   (a) Git clone , using the following code:
   (b) git clone git@github.com:Cache-Money404/cs107-FinalProject.git
   (c) cd Cache-Money404
   (d) pip install -r requirements.txt
   (e) pip install -e . The "-e ." flags the source code and makes it editable without having to reinstall it.
5. Using pip install:

pip install -i https://test.pypi.org/simple/CMFluid==0.01

3.1 Package Interaction

The following are Guidelines for download, installation, and use of package via git clone. The package is available for download on **GitHub** through the following **URL**:

`https://github.com/Cache-Money404/cs107-FinalProject.git`

It can be cloned from the command line as follows:

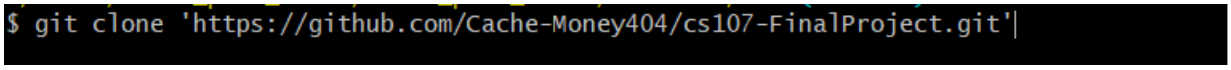


Figure 1: Clone the package using git clone

The package can also be downloaded (see image below), to do so :

1. Click on the "Code" Tab
2. Scroll down and download the ZIP file.
3. UnZip the files into your work directory and install using the package installation in the next section.

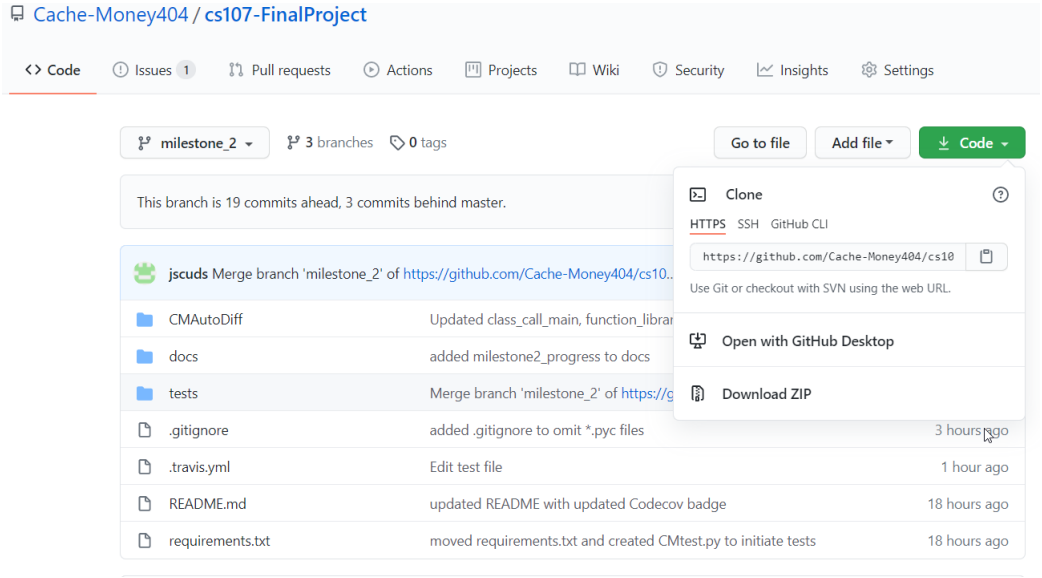


Figure 2: Download Package

3.2 Package Framework

The package doesn't use any formal framework at this point (in the future it will use PyPi). It can be installed using the **git clone and download** instructions listed above.

1. To install dependencies for the package in Python 3, run the following which is in the main directory of the library:

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

2. The library can be imported and used from the path on your local system from which you installed it.

3.3 Imports

To ensure that CMAutoDiff works correctly, ensure you import the following in the environment in which you wish to use CMAutoDiff:

```
import matplotlib.pyplot as plt
import numpy as np

from CMGradobject import CMGobject, CMvector
from CMfunc import CMfunc
from CMflow import CMflow
```

3.4 User Interface

After importing the necessary modules, a user creates a class instance of an `CMGobject` with the value to be used as input to a function.

Steps for Instantiating Variables and Functions

1. Initialize an input variable (i.e. 'x') with the value at which the function will be evaluated.

```
x = CMGobject(3)
```

2. Declare a function (i.e. 'f') with the variable and the FuncObj class.

   You can decide between 'sin', 'cos', 'tan', 'log', and 'exp'

   Here is a simple example:

$$f(x) = tan(x)$$

```
f = CMfunc.tan(x1)
```

   Here is a more complex example:

$$f(x) = sin(tan(x)) + 2^{cos(x)} + sin(x)^{tan(x)} - cos^2(x)$$

```
f = CMfunc.sin(CMfunc.tan(x))
    + 2**CMfunc.cos(x)
    + CMfunc.sin(x)**CMfunc.tan(x)
    - CMfunc.cos(x)**2
```

3. `f.val` will return the value of the function evaluated at the specific value
4. `f.grad` will return the derivative at the specific value

```
[31]
x = CMGobject(3)
f = CMfunc.sin(CMfunc.tan(x)) + 2**CMfunc.cos(x) + CMfunc.sin(x)**CMfunc.tan(x) - CMfu
print(f.val)
print(f.grad)

0.7033058537794523
[-0.63793616]
```

The above description of generating function instances valid for single and multivariate functions. In the circumstance that one wishes to describe a vector valued function, one begins, as before, declaring the variables that will feature in such a function. Consider the following function $f : \mathbb{R}^3 \to \mathbb{R}^2$:

```
import numpy as np
import CMfunc as cm

x = CMGobject(1, np.array([1, 0, 0]))
y = CMGobject(2, np.array([0, 1, 0]))
z = CMGobject(3, np.array([0, 0, 1]))
```

Next, one defines the functions comprising each of elements of f, concatenating them together as a list:

```
f1 = cm.tan(x) - z/y
f2 = x*y*cm.log(z)
f_list = [f1, f2]
```

Finally, one can instantiate the function f using this list:

```
f = CMvector(f_list)
```

The resultant CMvector object `f` has attributes `val` and `jac`, the former of which will be, in this case, a numpy array of length 2, the latter of which will be a 2 by 3 numpy array, representing the Jacobian.

[14]
```
x = CMGobject(1, np.array([1, 0, 0]) )
y = CMGobject(2, np.array([0, 1, 0]) )
z = CMGobject(3, np.array([0, 0, 1]) )
```

[23]
```
f  = (CMfunc.sin(x) + CMfunc.cos(y) + CMfunc.tan(z))/CMfunc.exp(x)
print(f.val)
print(f.grad)
```

```
0.10402806737179966
[ 0.09473804 -0.33451183  0.37535457]
```

Steps for Computing Flow Gradients

1. Run `CMflow.py` from the command line.

```
(base) Matthews-MBP-5:cmautodiff matthewhawes$ python CMflow.py
```

2. Enter the bounds to which the graph will be restricted.

```
enter lower bound of x domain (example: -1)
-1
enter upper bound of x domain (example: 1)
1
enter lower bound of y domain (example: -1)
-1
enter upper bound of y domain (example: 1)
1
```

3. Enter your preferred parameter values: strength, X value, Y value.

```
Enter Strength:3
Enter X Value:0
Enter Y Value:0
```

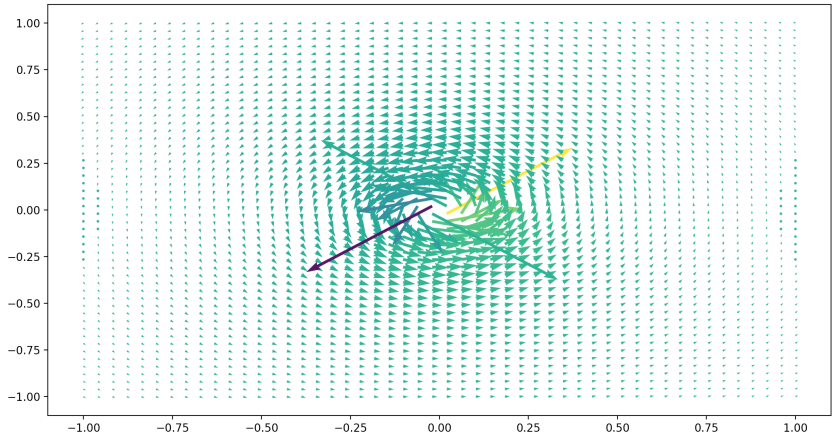
4. Select your flow visualization type (1-7, or 8 to exit).

```
Enter Potential Flow Vizualization:
 1) uniform
 2) doublet
 3) sink
 4) source
 5) vortex
 6) tornado
 7) whirlpool
 8) exit
8
```

5. Your graph will be plotted with an output confirming your previously defined parameters.

```
{'vortex1': [3.0, 0.0, 0.0]}
computing flow gradients for the following potential flow solutions:
'vortex1: a vortex of strength 3.0 at (x, y) = [0. 0.]'
Done. Generating plots:


Plots generated. Close window to continue
```



6. Once you close out of the graph, you will be prompted to calculate the velocity gradient at a specific point. If you chose to, you will type in the x- and y-coordinates, and it will output the equivalent polar coordinate, flow potential, and the polar and Cartesian gradient values

```
Would you like to calculate another velocity gradient at a specific point?
1) Yes, I love this flow scenario
2) No, I've seen enough of this one
```

```
enter x coordinate:
1
enter y coordinate:
1
computing flow gradient for the following potential flow solutions:
Done. At (x, y) = [1. 1.], the following has been calculated:
equivalent polar coordinate: (r, theta) =  [1.41421356 0.78539816]
flow potential:  11.780972450961723
polar gradient value: (dr, dtheta) =  [0. 3.]
cartesian gradient value: (dx, dy) =  [-1.4999999999999998, 1.5]
```

7. Finally, you will be prompted to rerun the program from the beginning

```
Would you like to calculate another velocity gradient at a specific point?
1) Yes, I love this flow scenario
2) No, I've seen enough of this one
2
start over?
1) Yes, potential flow is wonderful and I want more
2) No, I think I've had enough potential flow
2
exiting potential flow visualization
```

4 SOFTWARE ORGANIZATION

4.1 Directory Structure

```
main
├── tests
│   └── test_CM.py
├── docs
│   ├── README.md
│   ├── documentation.pdf
│   ├── milestone1.pdf
│   ├── milestone2.pdf
│   └── milestone2_progress.pdf
├── CMAutoDiff
│   ├── CMgradobject.py
│   │   └── contains objects for calculating derivatives for scalar and vector cases
│   ├── CMfunc.py
│   │   └── main elementary functions implementations
│   └── CMflow.py
│       └── contains several potential flow cases with ability to generate graphs
├── requirements.txt
├── setup.py
└── .travis.yml
```

The directory tree structure above shows the general design for our package. The main directory contains 3 sub-directories. The detailed functionality for each directory is listed below:

- `docs/` : contains all documentation for using, designing, and developing the CMAutoDiff package
- `CMAutoDiff/` : contains all the code for visualizing potential flow problems, the common math functions, and the automatic differentiation object
- `tests/` : directory that contains test files for `CMAutoDiff`

4.2 Modules

The following modules are included in the `CMAutoDiff` package:

- `CMGradobject.py` : contains the CMGobject that a user incorporates into a function to perform automatic differentiation; overloads arithmetic operations to calculate values and derivatives of a function. Also contains the CMvector object that is used for creating vector functions.
- `CMfunc.py` : contains elementary functions and their derivatives. All math functions make use of the `numpy` library. At this time the user can use the following elementary functions:
  1. Trigonometric functions: sine, cosine, tangent
  2. Inverse Trigonometric functions: arcsine, arccosine, arctangent
  3. Exponential (Euler's number) and logarithms of any base
  4. Hyperbolic functions: sinh(x), cosh(x), tanh(x)
  5. Logistic function
  6. Square root
- `CMflow.py` : contains pre-defined potential flow cases, allowing the user to choose a visualization and certain parameters; then outputting a 2D graph.

4.3 Test Suite

Currently, the test suite consists of:

- `TravisCI` and `CodeCov`, utilized with their badges on the `README.md` in the main directory.
- A developer series of tests at `tests/test_CM.py` that test every function and object method.
- Examples of program tests and handled cases currently included in the test suite:
  - Simple differentiation of 1 input and 1 function
  - Complex differentiation of multiple input variables and multiple output functions of several nested trigonometric, logarithmic, and power calculations (tests every basic function coded in the library)
  - Newton's Method as a root-finding algorithm
  - Every overloaded class method for the `CMGobject` class
  - Complex differentiation of 4 inputs and 1 vector valued functions
  - Whether or not the function is differentiable using common derivative rules.

4.4 Package Distribution

Currently, a user will follow the instructions in 3.1 Package Interaction to install and use `CMAutoDiff`. A user will install the latest version of `CMAutoDiff`, use `pip install` as follows (using ssh to avoid repetitively inputting credentials):

```
pip install git+https://github.com/Cache-Money404/cs107-FinalProject.git
```

4.5 Package Framework

The package doesn't use any formal framework and can be installed using the `pip` instructions listed above.

5 IMPLEMENTATION

As it stands, the architecture of classes entailed split into the classes `CMGobject` and `CMvector`, the former of which characterises forward mode differentiation properties of a **scalar** valued function of multiple variables $f : \mathbb{R}^n \to \mathbb{R}$, the latter of which evaluates the case of a vector valued function $g : \mathbb{R}^m \to \mathbb{R}^n$ and accordingly will calculate a Jacobian. This latter implementation is irrelevant for our extension, but is implemented if one wishes to use the mechanisms entailed in our library for a task that would involve evaluating such a function.

Referencing the above section 3.4, the basic automatic differentiation routine may be considered as follows: variables are declared using CMGobject, which defaults to single variable input. On such variables, the user may import and call function entailed in CMfunc to generate new CMGobjects with values and derivatives respecting the properties of the applied function and the chain rule of differentiation.

Importantly, the CMGobject is capable of multivariate input. To specify a multivariate function, one begins by constructing the variables that will constitute the function, and passes a second parameter to the CMGobject declaration which is a "one hot" numpy array. The position of the one in such a vector identifies the variable. For example, if one wished to define a function $f(x, y, z)$ where $x = 1, y = 2, z = 3$, then one would declare the following CMGobjects:

```
x = CMGobject(1, np.array([1, 0, 0]) )
y = CMGobject(2, np.array([0, 1, 0]) )
z = CMGobject(3, np.array([0, 0, 1]) )
```

One can go on to describe the function $f(x, y, z)$ using the functions in CMfunc as usual. The congruence of multivariate function and single variate function definitions is made possible by the vectorization routines applied within the methods of the CMGobject, but more aptly, by the linearity property of the chain rule of partial differentiation. In the context of automatic differentiation, this linearity property allows our generalized case because at each layer of the implied evaluation trace, the partial derivatives are separable into their respective gradient vector positions, as implied by the "one hot vector".

As a proof of concept for our algorithm, we present an extension detailed in the CMflow.py file of our module. Here, we evaluate gradients of multivariate potential functions with the objective of simulating fluid flow.

6 EXTENSION: POTENTIAL FLOW SOLVER

Potential flow modelling is typically instructed to students as follows: a selection of velocity potentials ϕ are provided, each describing a particular flow feature such as a vortex or uniform onset flow. For incompressible flows, $\nabla \cdot \vec{u} = \nabla \cdot (\nabla \phi) = 0$, and the following differential equation is implied:

$$\nabla \cdot (\nabla \phi) = \nabla^2 \phi = 0$$

The above linear, homogenous differential equation is known as Laplace's equation, and because it is linear, then by the principle of linear superposition of solutions, any linear combination of functions $\phi \in \mathbb{R}$ individually satisfying Laplace's equation will also solve Laplace's equation [Batchelor (2010)].

In practical terms, this means that if the above differential equation describes our flow field, then we may sum up scalar multiples of arbitrarily many velocity potential functions ϕ and guarantee that the resultant function $\Phi = \sum_i \phi_i$ will also be a valid flow solution. In the context of our algorithm, this means that we may take sums of many different flow types, each described by a velocity potential ϕ, and the resultant gradient of their sum will describe a new flow state that can be understood as a fluid continuum featuring *all* of the flow features described by each individual ϕ.

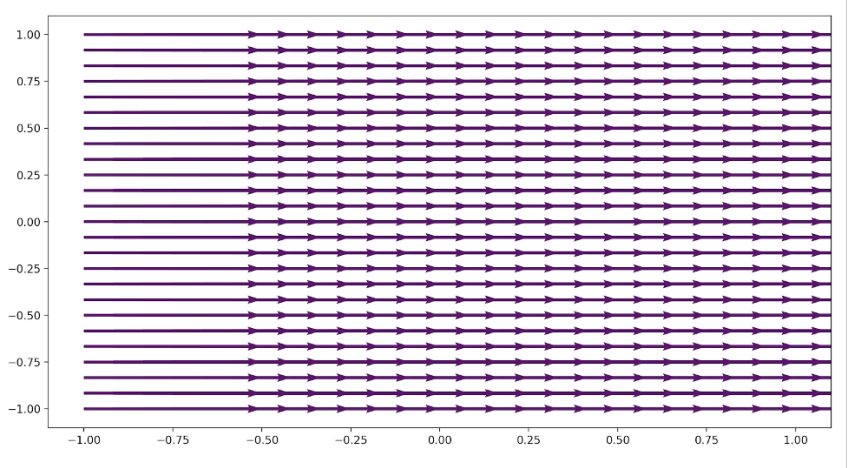
Thus, with the above guarantee as given, we may also exploit the linear property of the gradient operator $\nabla(\phi_1 + \phi_2) = \nabla\phi_1 + \nabla\phi_2$ to cast this problem as a very well posed linear program. One simply has to evaluate the gradient of a flow solution ϕ_1 and add it to the gradient of another flow solution ϕ_2 to generate a vector field $\vec{u}$ encoded with the properties of both potential fields.

Our supported functions are as follows:

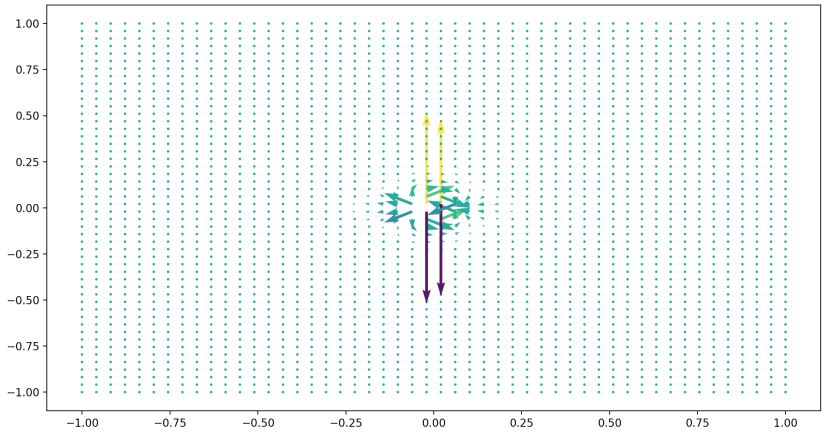
- uniform flow: $\phi = Ar\cos(\theta)$
- doublet flow: $\phi = \frac{A}{r}\cos(\theta)$
- source flow: $\phi = \frac{\lambda}{2\pi b}\log(r)$
- sink flow: $\phi = -\frac{\lambda}{2\pi b}\log(r)$
- vortex flow: $\phi = \Gamma\theta$
- tornado flow: $\phi = K\log(r) + \Gamma\theta$
- whirlpool flow: $\phi = -K\log(r) + \Gamma\theta$

Example flow graphs:

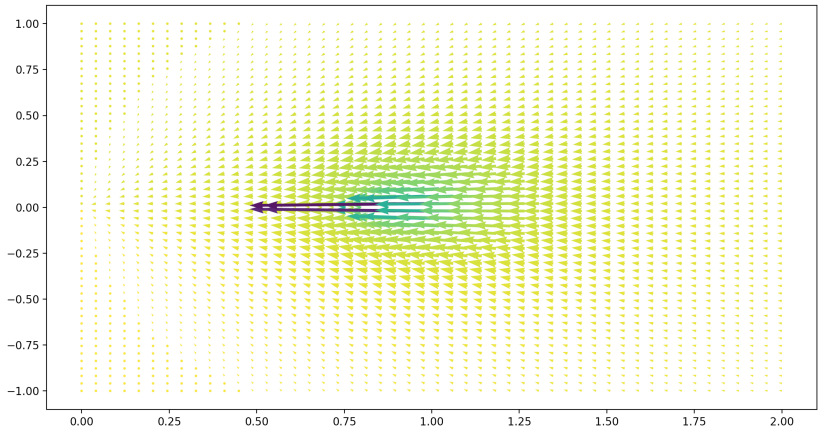
Uniform Flow



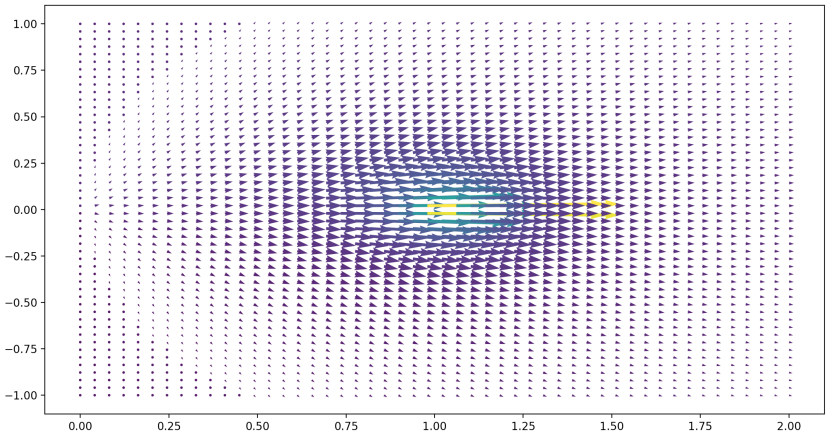
Doublet



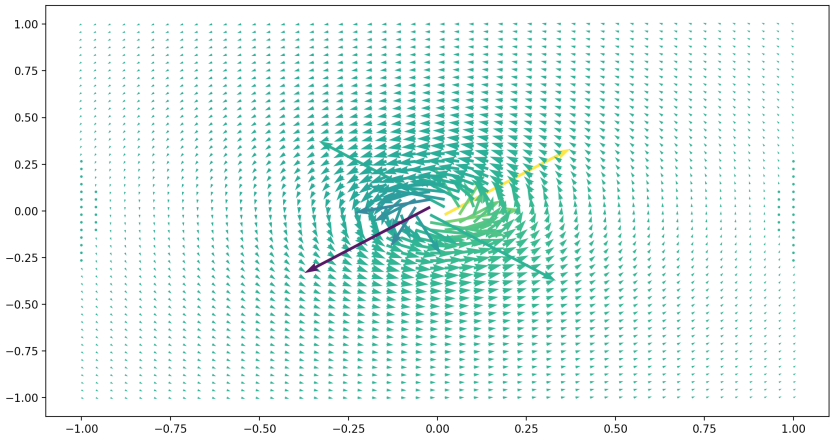
Sink



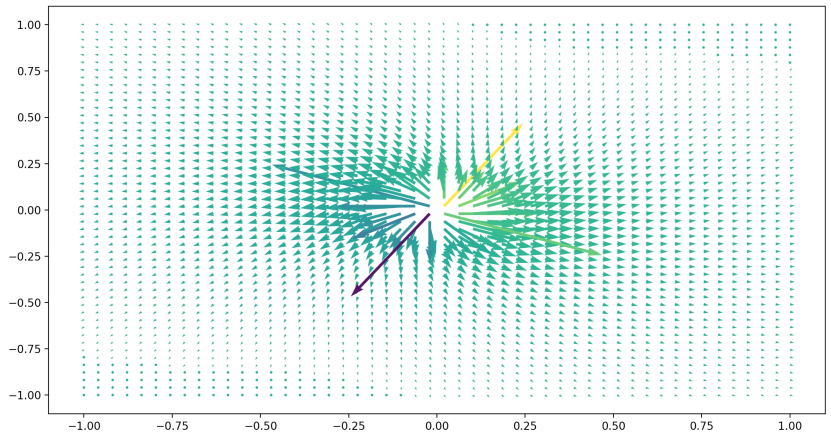
Source



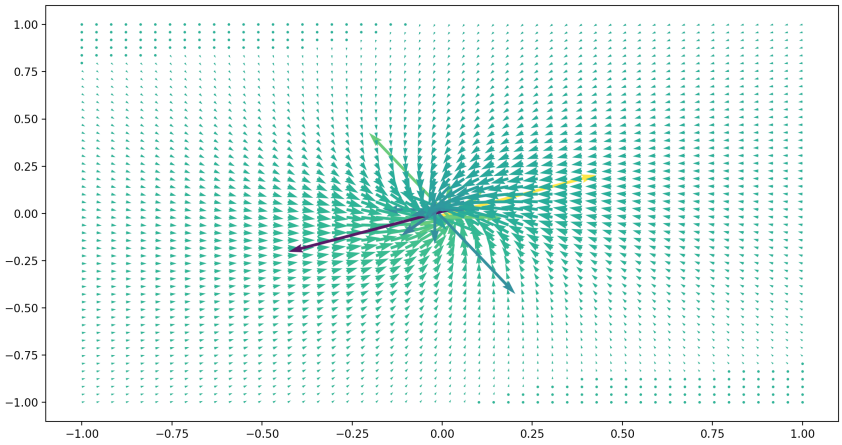
Vortex



Tornado



Whirlpool



6.1 Description

Given our flexible automatic differentiation technique described in section 5, we have implemented the following solver: from a list of supported potential flow solutions, the user may specify the features they desire in their flow field. From this specification, the algorithm *analytically* evaluates the gradient of each of these flow features in turn over a finite set of points indicated by user input. The algorithm then combines these evaluated gradients invoking the above property of linear superposition and relays them back to the user both graphically and numerically. This process allows the user to visualize and evaluate complex flows, which are often intractable to analyse from solely their closed form potential functions. Thus, using a powerful method of automatic differentiation, our package can compute analytically accurate velocities even faster than comparable numerical schemes involving finite difference, the traditional route to computing potential flow solutions [White (2008)].

7 BROADER IMPACT AND INCLUSIVITY STATEMENT

Historically, scientists, engineers, and technological developers have not always considered the future impacts of their discoveries and inventions. While the utility of the cellphone or the fiber optic cable is not disputed, other inventions such as the automobile have caused harmful external impacts that may have been reduced with careful thought during development. Additionally, unfair or discriminatory practices are often exacerbated by new technologies, increasing the divide between historically privileged and underprivileged groups. We kept these ideas in mind when designing our package.

7.1 Broader Impact

While creating CMAutoDiff, our group debated the consequences of various people using our software. We present one positive and one negative effect of our package here and both relate to education. Since the primary purpose of potential flow visualization is for educational purposes, our code is accessible and modifiable. This allows instructors and students alike to explore different sets of prescribed potential flow problems; both visually and mathematically. It also allows for forking and further development. Unfortunately, a malicious actor could take advantage of our code accessibility and modify it to calculate and produce incorrect potential flow results. In the digital age, it is paramount that developers do everything within their power to guard against disinformation, whether scientific or otherwise.

7.2 Inclusivity

Since CMAutoDiff is distributed as an educational tool, we designed it with a broad scope of users in mind. By making it publicly available on GitHub and PyPI, individual users can fork and access our code at will. We have restricted contributors (due to the scope of the class in which we are enrolled), but any individual wishing to contributed to our library can reach out to the group members via GitHub. One of the team members will approve collaborators on an individual basis and subsequently approve any pull request with a meaningful contribution to the code base. We have a passion for sharing an interest in fundamentals of fluid mechanics and look forward to other users' perspectives and input!

8 FUTURE FEATURES

In this package we implemented the forward mode of automatic differentiation with one primary application: visualizing potential flow. There are many other potential flow solutions that can be implemented, and so inclusion of some additional cases such as Rankine half bodies and polygonal bodies would be welcome.

Aside from additional potential flow solutions, our package can be further improved by adding additional features that can be used to solve problems in other subjects. Here we present several possible ideas for future work.

8.1 Future Feature 1: Reverse Mode

The reverse mode of automatic differentiation can sometimes be more useful than the forward mode we implemented in our package. The major differences between the forward mode and the reverse mode of automatic differentiation is that the previous solution starts from the inputs and calculate the derivative at every step along the way to the output, whereas the latter solution starts from the outputs and propagates back along the pathway to the inputs. Both methods require implementation of chain rules to complete the calculation. In the future, we could add another module that utilizes a tree structure to allow us to retrieve the previous calculation recursively on top of the differentiation rules and chain rules our package currently implements.

The reverse mode of automatic differentiation are widely used in assisting the calculation of the back propagation process in the training of deep learning architectures. If our package is equipped with the reverse mode, we can extend our current application to machine learning and deep leaning services.

8.2 Future Feature 2: Higher Orders of Automatic Differentiation

Currently our package is only equipped with the ability to calculate the first order derivative directly. However, higher degrees of derivatives can also be very useful in various areas including physics and mathematics. One simple application is finding the acceleration of a moving object, a valuable physics calculation. Another possible application is optimizing curves and algorithms used in physics, mathematics or biology. First order derivatives can help us find the local maximum and local minimum, but without the second order derivatives we can never decide whether a local minimum is actually a global minimum. With the broad usage of optimization, algorithms that help with higher order derivative finding can be very useful and can be applied to different fields.

8.3 Future feature 3: Root-finding algorithm

Another application we can potentially add to our package is a series of root finding algorithms. In lecture we learned about Newton's Method of approximating the roots of a function. We can further extend this method to higher order derivatives or more variables being involved in equations. One possible application of the root finding algorithm can be calculating the solutions for Partial Differential Equations (PDEs) in Applied Mathematics, which requires efficient algorithms to deal with higher order derivatives and root finding.

9 FIGURE FOR FINAL IMPLEMENTATION

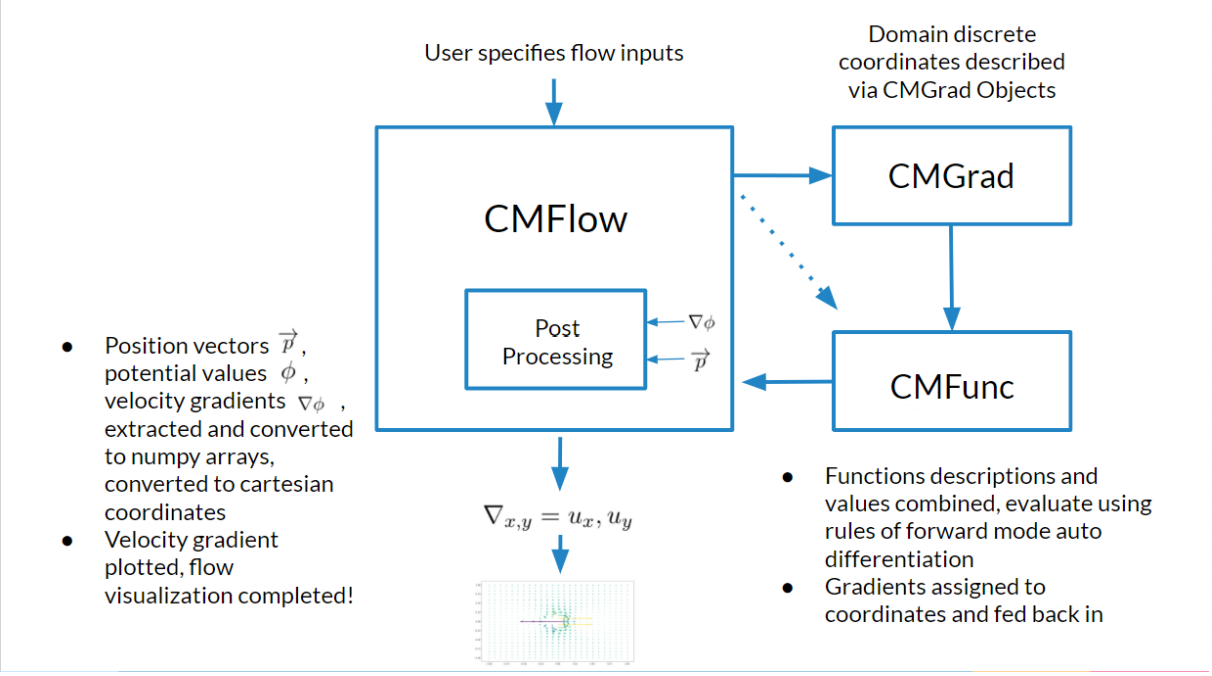


Figure 3

REFERENCES

G. Batchelor. *An Introduction to Fluid Mechanics*. Cambridge University Press, 2010.

F. White. *Fluid Mechanics*. McGraw Hill, 2008.