NTNU | Norwegian University of Science and Technology

The Research Council of Norway

Automatic Differentiation for Optimization problems with Complementarity Constraints

Saket Adhau

Department of Chemical Engineering,
Norwegian University of Science and Technology (NTNU).

Friday, 26 January 2024

Aim of Talk

1. What are Complementarity Constraints?
2. Motivation for Studying Complementarity Constraints in Optimization
3. Automatic Differentiation (AD)
4. How to use them ?

Classic Example : The Door Dilemma

Imagine two people, Alice and Bob, trying to go through a door, but only one person can pass at a time due to limited space. The decision variables are binary:

- $A = 1$ if Alice goes through the door.
- $B = 1$ if Bob goes through the door.

The complementarity constraint in this scenario is straightforward:
$A + B = 1$ or $A \cdot B = 0$.

They are used to model situations where two or more variables must satisfy a certain relationship.

Karush-Kuhn–Tucker (KKT) Conditions

The early motivation for studying the Linear Complementarity Problem (LCP) was because the KKT optimality conditions for Linear and Quadratic Programs (QP) constitute of Complementarity Problems.

The complementarity condition in KKT often involves the product of a Lagrange multiplier and the corresponding constraint.

William Karush

Harold W. Kuhn

Albert W. Tucker

Karush Kuhn Tucker (KKT) Conditions

Inequality-constrained optimization problem

$$\min_x f(x)$$
$$\text{s.t } g_i(x) \leq 0, \ \forall_i \in \{1, \ldots, m\}$$

the Lagrangian is defined as,

$$\mathcal{L}(x, \lambda) = f(x) + \sum_{i=1}^{m} \lambda_i g_i(x^\star)$$

and the KKT conditions are given by,

$$\nabla_x \mathcal{L}(x^\star, \lambda^\star) = 0$$
$$g_i(x^\star) \leq 0, \ \forall_i \in \mathcal{I}$$
$$\boxed{\lambda_i^\star \geq 0}, \ \forall_i \in \mathcal{I}$$
$$\boxed{\lambda_i^\star g_i(x^\star) = 0}, \ \forall_i \in \mathcal{I}$$

- $\lambda_i^\star g_i(x^\star) = 0$ is a complementary slackness condition.
- strict complementarity holds if $\lambda_i^\star \geq 0$ for all $i \in \mathcal{A}(x^\star)$

1 What are Complementarity Constraints?

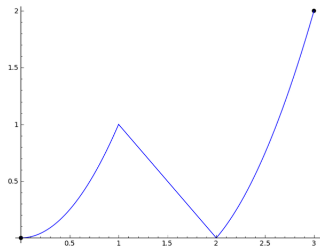
2 Motivation for Studying Complementarity Constraints in Optimization

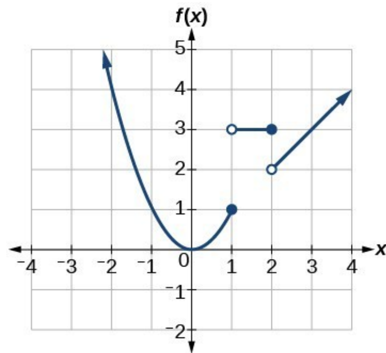
3 Automatic Differentiation (AD)

4 How to use them ?

Motivation for Studying Complementarity Constraints in Optimization

Complementarity conditions often introduce non-smoothness to an optimization problem due to the presence of non-differentiable functions, discontinuities, or piecewise-defined relationships within the constraints or the complementarity terms themselves.

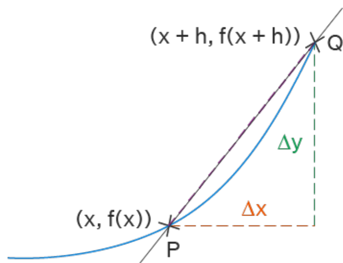




The lack of smoothness in this scenario presents difficulties for conventional optimization algorithms that heavily depend on gradients.

What is a derivative ?

Slope of the tangent line.



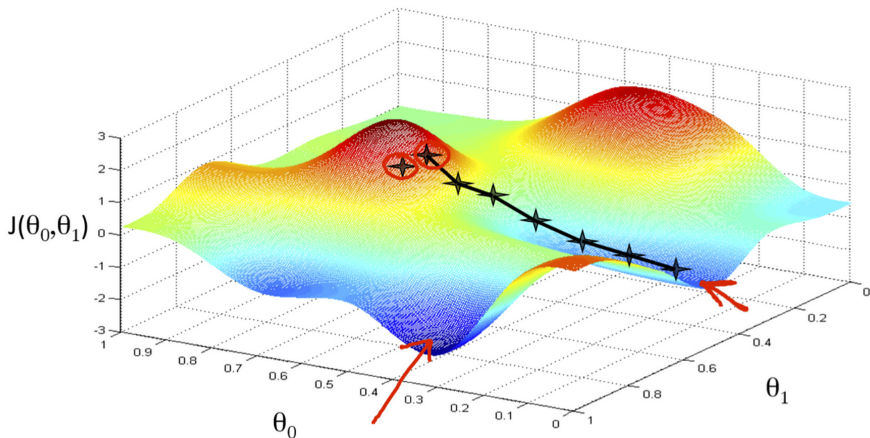
$$\text{Slope} = \frac{\Delta y}{\Delta x}$$

$$= \frac{f(x + h) - f(x)}{(x + h) - x}$$

$$= \frac{f(x + h) - f(x)}{h}$$

What is a Gradient ?

Direction along which the function increases at the fastest rate



Manual Differentiation

It will help if we first contrast AutoDiff with some other methods.

- Manual Differentiation

$$f(x) = e^{2x} - x^3 \quad \longrightarrow \quad f'(x) = 2e^{2x} - 3x^2$$

- Differentiate using basic derivatives rules.
- We could simply code up the result and call it a day
- But this can be a fairly tedious process for more complicated functions.

Numerical Differentiation

Approximating derivatives through finite differences involves utilizing the values of the original function computed at specific sample points.

For a scalar valued function $f : \mathbb{R}^n \to \mathbb{R}$

Approximate Gradient ∇f is given by:

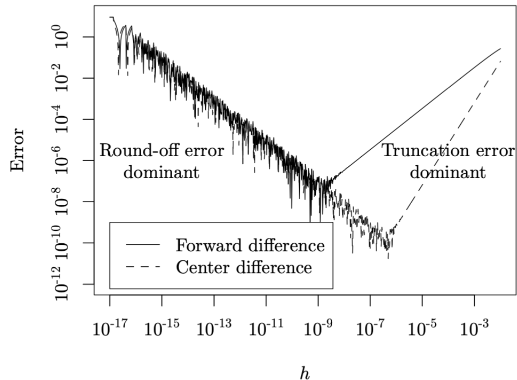
$$\frac{\partial f(\mathbf{x})}{\partial x_i} \approx \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h},$$

where $\mathbf{e}_i$ is the ith unit vector and $h > 0$ is a small step size, e.g. 10^{-5}.

Numerical differentiation inherently suffers from ill-conditioning and instability, giving rise to truncation and rounding errors.

Carefull Consideration of Step size h

Truncation error due to non-zero h
Rounding error due to limited precision of floating point arithmetic



Needs $O(n)$ evaluations of f for a gradient in n dimensions .

Symbolic Differentiation: Automated Version of Manual Differentiation

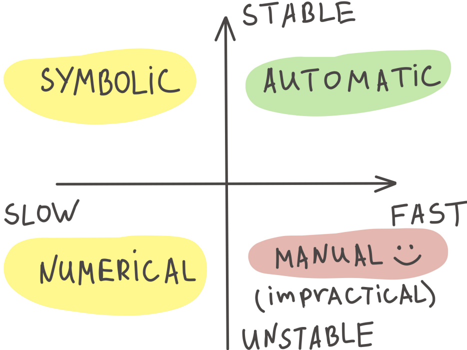
Automatic manipulation of expressions for obtaining derivative expressions, carried out by applying transformations representing rules of differentiation.

Iterations of the logistic map $l_{n+1} = 4l_n(1 - l_n), l_1 = x$ and the corresponding derivatives of l_n with respect to x

n	l_n	$\frac{d}{dx} l_n$
1	x	1
2	$4x(1-x)$	$4(1-x) - 4x$
3	$16x(1-x)(1-2x)^2$	$16(1-x)(1-2x)^2 - 16x(1-2x)^2 - 64x(1-x)(1-2x)$
4	$64x(1-x)(1-2x)^2 (1-8x+8x^2)^2$	$128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$

When functions become increasingly complex, a phenomenon called “expression swell” occurs.

Automatic Differentiation (AD): Efficiently Calculating Derivatives of Mathematical Functions



Automatic Differentiation (AD): Enhancing Precision in Derivative Computation

AD provides accurate and exact derivatives, making it a powerful tool in optimization, machine learning, and scientific computing.

$$f(x_1, x_2) = \left[\sin\left(\frac{x_1}{x_2}\right) + \frac{x_1}{x_2} - e^{x_2}\right] \times \left[\frac{x_1}{x_2} - e^{x_2}\right]$$

```
def f(x1, x2):
    a = x1 / x2
    b = np.exp(x2)
    return (np.sin(a) + a - b) * (a - b)
```

Automatic Differentiation (AD): Enhancing Precision in Derivative Computation

- Bypasses symbolic inefficiency by leveraging intermediate variables present in the original functions implementations.

```
def f(x1, x2):
    a = x1 / x2
    b = np.exp(x2)
    return (np.sin(a) + a - b) * (a - b)
```

Automatic Differentiation (AD): Enhancing Precision in Derivative Computation

- Implemented differential functions are composed of underlying primitive operations whose **derivatives we know**.
- The chain rule allows us to compose this together.

```
def f(x1, x2):
    a = x1 / x2
    b = np.exp(x2)
    return (np.sin(a) + a - b) * (a - b)
```

Automatic Differentiation (AD)

Forward Mode

Reverse Mode

Forward Mode

- Augment each individual variable during evaluation of a function with its derivative.

$$\nu_i \longrightarrow (\nu_i, \dot{\nu}_i)$$

Forward Mode

$$f(x_1, x_2) = \left[\sin\left(\frac{x_1}{x_2}\right) + \frac{x_1}{x_2} - e^{x_2}\right] \times \left[\frac{x_1}{x_2} - e^{x_2}\right]$$

We have two scalar input x_1, x_2 and a single scalar output. We also see repeated use of some sub expressions.

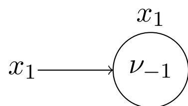
x_1

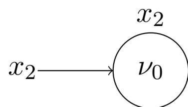
x_2

[1] notation from Griewank and Walther, 2008

Forward Mode

$$f(x_1, x_2) = \left[\sin\left(\frac{x_1}{x_2}\right) + \frac{x_1}{x_2} - e^{x_2}\right] \times \left[\frac{x_1}{x_2} - e^{x_2}\right]$$

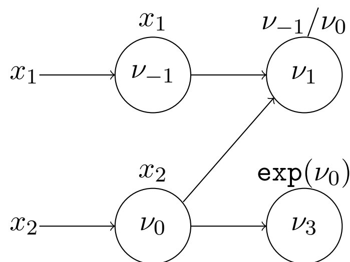




[1] notation from Griewank and Walther, 2008

Forward Mode

$$f(x_1, x_2) = \left[\sin\left(\frac{x_1}{x_2}\right) + \frac{x_1}{x_2} - e^{x_2}\right] \times \left[\frac{x_1}{x_2} - e^{x_2}\right]$$

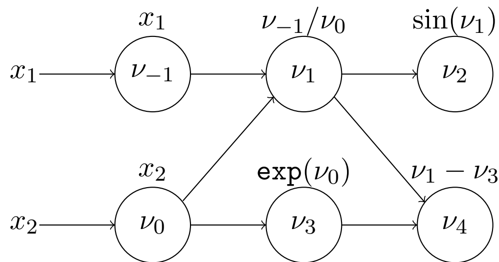


[1] notation from Griewank and Walther, 2008

Forward Mode

$$f(x_1, x_2) = \left[\sin\left(\frac{x_1}{x_2}\right) + \frac{x_1}{x_2} - e^{x_2}\right] \times \left[\frac{x_1}{x_2} - e^{x_2}\right]$$

As we proceed, we evaluate intermediate variables some of which will be used later in the computation.

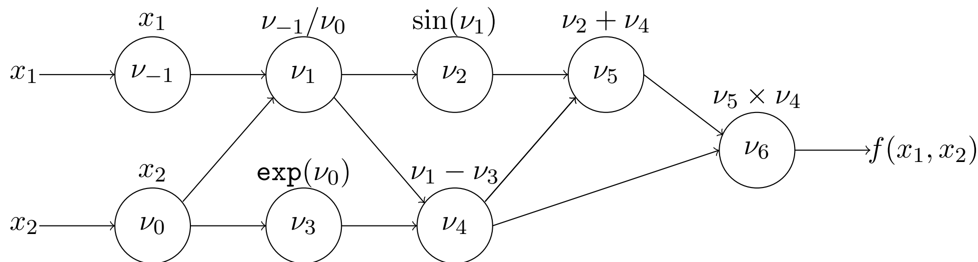


[1] notation from Griewank and Walther, 2008

Forward Mode

$$f(x_1, x_2) = \left[\sin\left(\frac{x_1}{x_2}\right) + \frac{x_1}{x_2} - e^{x_2}\right] \times \left[\frac{x_1}{x_2} - e^{x_2}\right]$$

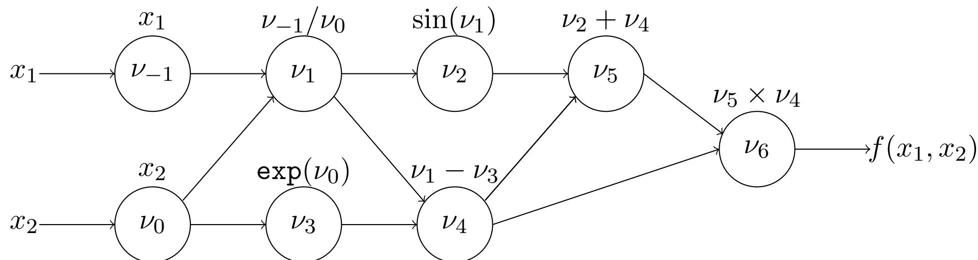
We eventually reach the final function output.



[1] notation from Griewank and Walther, 2008

Forward Mode

$$\frac{\partial f}{\partial x_1} \text{ at } (1.5, 0.5)?$$



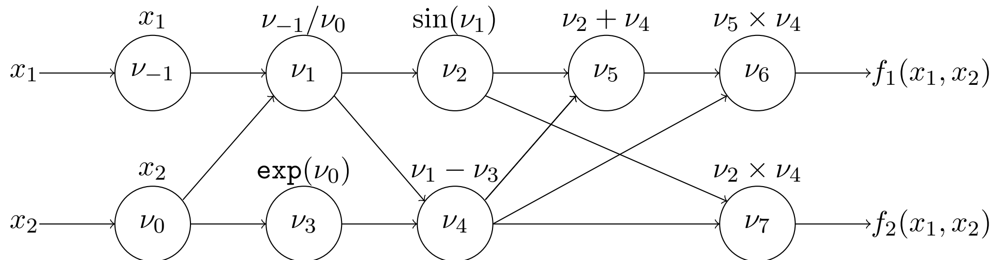
A single pass through the function now produces not only the original output, but partial derivative of the interest.

[1] notation from Griewank and Walther, 2008

Forward Mode

What about multiple outputs ?

$$\text{compute } \frac{\partial f}{\partial x_1} \text{ and } \frac{\partial f}{\partial x_2} \text{ in a single forward pass.}$$



A separate forward pass is required for each input variable of interest.

[1] notation from Griewank and Walther, 2008

Forward Mode

Consider general function

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^m$$

$$J_f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Each pass produces one column of the corresponding jacobian.
Forward mode is ideal/preferred when $n << m$ (few inputs and many outputs)
for e.g. , in "Sensitivity analysis"

Jacobian Vector Product

More generally we can compute the jacobian vector products without ever computing the jacobian matrix itself.

$$J_f r = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \begin{bmatrix} r_1 \\ \vdots \\ r_n \end{bmatrix}$$

We set $\dot{x} = r$ to the vector if interest, and proceed with forward mode AutoDiff.

Now, this seems a little bit weird, after all we just said that forward mode requires one pass through the function for each input.
We can basically think of the jacobian vector product as just the jacobian of a different function.

Jacobian Vector Product

The composition of our original function and one with a single scalar input whose jacobian is the column vector r.

$$J_h = J_f r = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \begin{bmatrix} r_1 \\ \vdots \\ r_n \end{bmatrix}$$

$$h = f \cdot g$$
$$J_g = r$$

Jacobian Vector Product

Because, the overall jacobian of this composed function, now only has one column.
Therefore a single pass is sufficient.

$$J_h = J_f r = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \begin{bmatrix} r_1 \\ \vdots \\ r_n \end{bmatrix} \in \mathbb{R}^{m \times 1}$$

$$h = f \cdot g$$
$$J_g = r$$

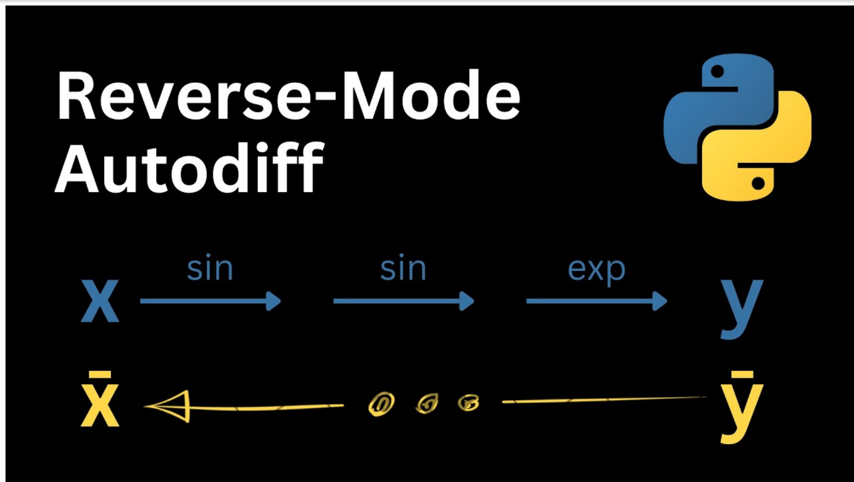
Automatic Differentiation (AD)

Forward Mode

Reverse Mode

Reverse Mode

Rather than propagating derivatives forward, they will be propagated backwards, from the output.



Reverse Mode

$$f(x_1, x_2) = \left[\sin\left(\frac{x_1}{x_2}\right) + \frac{x_1}{x_2} - e^{x_2}\right] \times \left[\frac{x_1}{x_2} - e^{x_2}\right]$$

Two part process, let's start with forward pass.

Forward Pass $\longrightarrow$

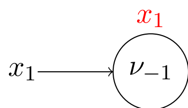
x_1

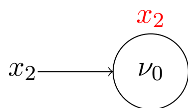
x_2

Reverse Mode

$$f(x_1, x_2) = \left[\sin\left(\frac{x_1}{x_2}\right) + \frac{x_1}{x_2} - e^{x_2}\right] \times \left[\frac{x_1}{x_2} - e^{x_2}\right]$$

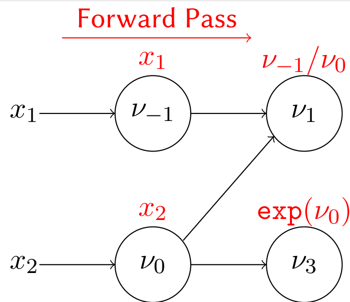
Forward Pass





Reverse Mode

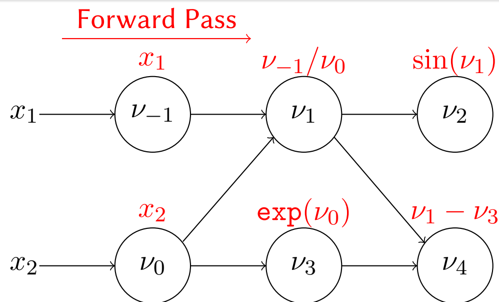
$$f(x_1, x_2) = \left[\sin\left(\frac{x_1}{x_2}\right) + \frac{x_1}{x_2} - e^{x_2}\right] \times \left[\frac{x_1}{x_2} - e^{x_2}\right]$$



Reverse Mode

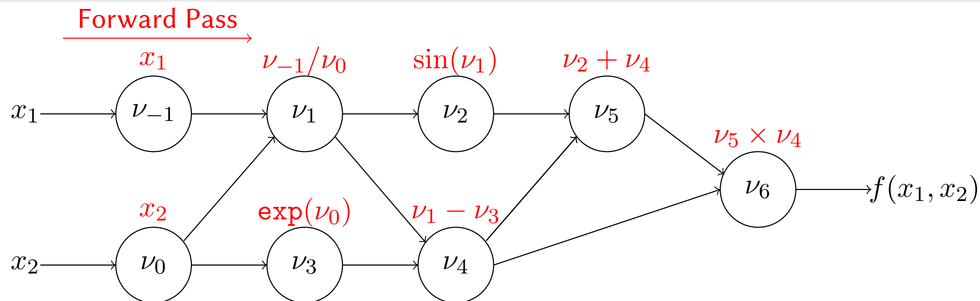
$$f(x_1, x_2) = \left[\sin\left(\frac{x_1}{x_2}\right) + \frac{x_1}{x_2} - e^{x_2}\right] \times \left[\frac{x_1}{x_2} - e^{x_2}\right]$$

evaluating intermediate variables as we did before.



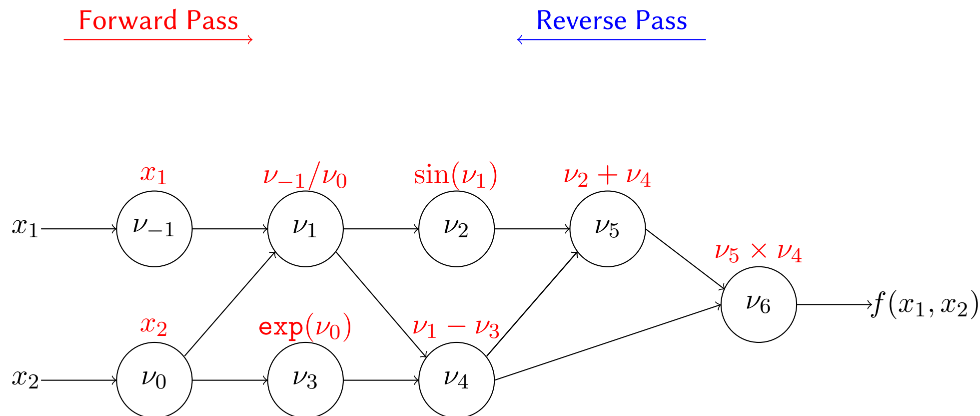
Reverse Mode

However, instead of simultaneously computing derivatives, we store the dependencies of the expression tree in memory.



Reverse Mode

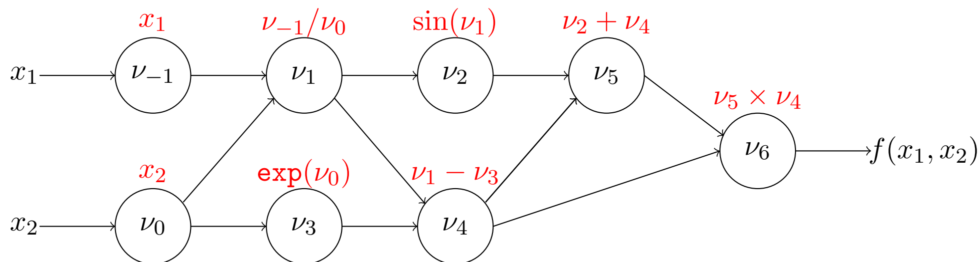
After completion of the forward pass, we compute partial derivatives of the output with respect to the intermediate variable quantities known as adjoints.



"Adjoint" $\overline{\nu}_i = \frac{\partial f}{\partial \nu_i}$

Reverse Mode

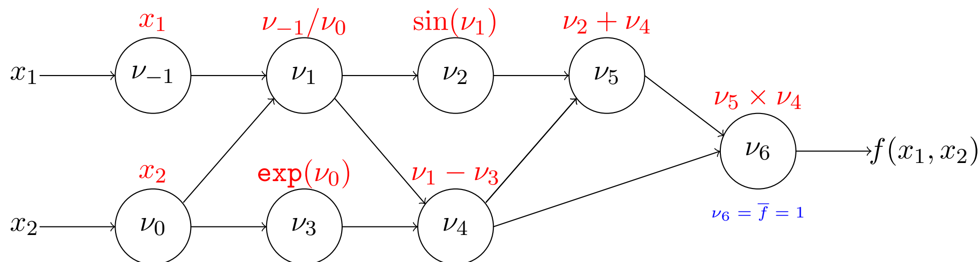
To obtain the adjoint $\bar{\nu}_i$ for a particular node, we look at each of the nodes children.
Further we multiply the adjoint with the partial derivative of the child w.r.t. ν_i



"Adjoint" $\overline{\nu}_i = \frac{\partial f}{\partial \nu_i} = \sum_{j:\texttt{child of } i} \overline{\nu}_j \frac{\partial \nu_j}{\partial \nu_i}$

Reverse Mode

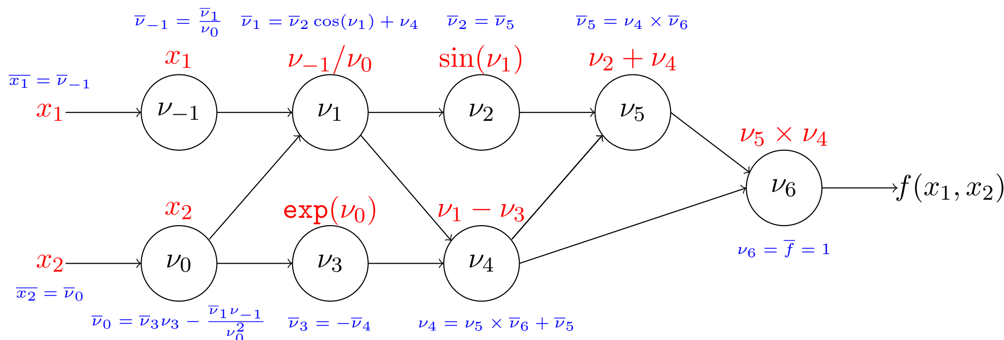
So ν_i's contribution to the final output is determined both by how it's children affect the output and how it affects each of it's children.



'Adjoint" $\overline{\nu}_i = \frac{\partial f}{\partial \nu_i} = \sum_{j:\texttt{child of } i} \overline{\nu}_j = \frac{\partial \nu_j}{\partial \nu_i}$

Reverse Mode

At the end we obtain partial derivatives w.r.t each input. So the gradient is computed with just a single execution of Reverse mode AutoDiff



"Adjoint" $\overline{\nu}_i = \frac{\partial f}{\partial \nu_i} = \sum_{j:\texttt{child of } i} \overline{\nu}_j = \frac{\partial \nu_j}{\partial \nu_i}$

Reverse Mode

For general vector valued function,
$f : \mathbb{R}^n \to \mathbb{R}^m$
Reverse mode produces one row of the jacobian at a time, when we have few outputs compared to inputs.
Computational cost of one sweep forward or reverse is roughly equivalent, but reverse mode requires access to intermediate variables, requiring more memory.

$$J_f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Automatic Differentiation (AD)

Forward Mode

Reverse Mode

- Forward and Reverse mode are the two extremes of automatic differentiation, however, in some settings a hybrid approach is preferred.
- For example, in second order optimization, where information on objective's curvature is taken into account, a hessian vector product is sometimes required.

What about Hessian ?

Reverse-on-Forward Version of AutoDiff allows efficient computation of this product.

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^m$$
Goal : Compute $H_f \nu$

First we use forward mode to compute directional $\nabla f \cdot \nu$ (set $\dot{x} = \nu$) as we did before,

then reverse mode is used to differentiate resulting in $\nabla^2 f \cdot \nu = H_f \nu$ without the explicit computation of hessian matrix itself.

[1] see Pearlmutter, 1994

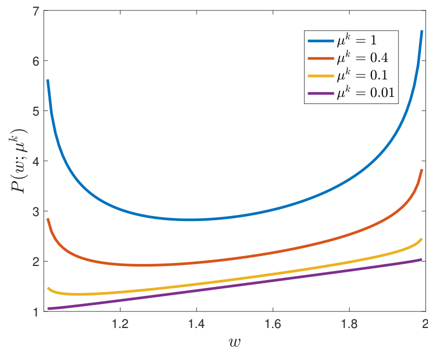
The Logarithmic Barrier Method

Inequality-constrained optimization problem

$$\min_{w} f(w)$$
$$\text{s.t } g_i(w) \leq 0, \ \forall_i \in \{1, \ldots, m\}$$

When converted into an unconstrained optimization problem,

$$\min_{w,\mu^k} \ f(w) - \mu^k \sum_{i=1}^{m} \log(-g_i(w)),$$



The Fischer-Burmeister Method

Consider complementarity constraints of the form.

$$x_i \geq 0, \;\; y_i \geq 0, \;\; x_i \cdot y_i = 0$$

where x_i and y_i are variables.

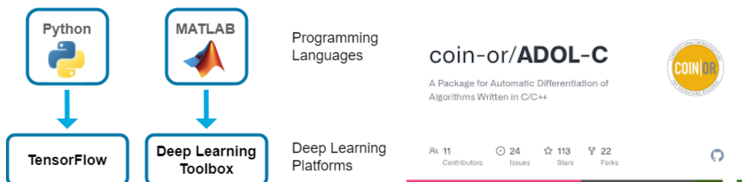
Fischer-Burmeister function:

$$FB(x, y) = x + y - \sqrt{x^2 + y^2}$$

This function is designed to capture the essence of the complementarity condition $x \cdot y = 0$

Implementation Example

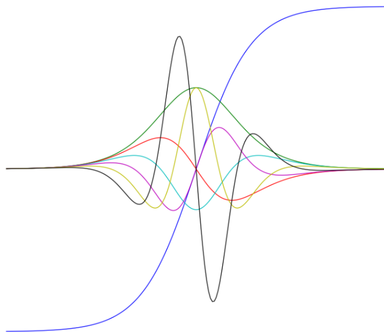
```
>>> import autograd.numpy as np  # Thinly-wrapped numpy
>>> from autograd import grad    # The only autograd function you may ever need
>>>
>>> def tanh(x):                 # Define a function
...     y = np.exp(-2.0 * x)
...     return (1.0 - y) / (1.0 + y)
...
>>> grad_tanh = grad(tanh)       # Obtain its gradient function
>>> grad_tanh(1.0)               # Evaluate the gradient at x = 1.0
0.41997434161402603
>>> (tanh(1.0001) - tanh(0.9999)) / 0.0002  # Compare to finite differences
0.41997434264973155
```



[1] Example from Autograd Github Library

Implementation Example

```
>>> from autograd import elementwise_grad as egrad  # for functions that vectorize over inpu
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-7, 7, 200)
>>> plt.plot(x, tanh(x),
...          x, egrad(tanh)(x),                                     # first  derivative
...          x, egrad(egrad(tanh))(x),                              # second derivative
...          x, egrad(egrad(egrad(tanh)))(x),                       # third  derivative
...          x, egrad(egrad(egrad(egrad(tanh))))(x),                # fourth derivative
...          x, egrad(egrad(egrad(egrad(egrad(tanh)))))(x),         # fifth  derivative
...          x, egrad(egrad(egrad(egrad(egrad(egrad(tanh))))))(x))  # sixth  derivative
>>> plt.show()
```

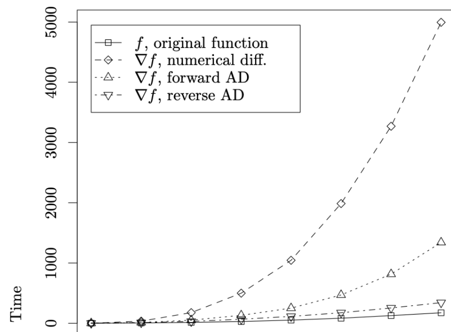


[1] Example from Autograd Github Library

Computational Efficiency

Helmholtz free energy function of a mixed fluid

$$f(x) = RT \sum_{i=0}^{n} \log \frac{x_i}{1 - b^\top x} - \frac{x^\top A x}{\sqrt{8} b^\top x} \log \frac{1 + (1 + \sqrt{2} b^\top x)}{1 + (1 - \sqrt{2} b^\top x)}$$



[1] Baydin et. all, 2018, Peng and Robinson, 1976

Overview

- Automatic differentiation (AutoDiff) refers to a general way of taking a program which computes a value, and automatically constructing a procedure for computing derivatives of that value.
- Back-propagation is the special case of AutoDiff applied to neural nets
- Autodiff is both efficient (linear in the cost of computing the value) and numerically stable

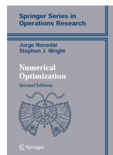
What is Autodiff not ?

- Autodiff is not symbolic differentiation (e.g. Mathematica)
- The goal of AutoDiff is not a formula, but a procedure for computing derivatives.

Applications of AutoDiff

- Newton's method for solving nonlinear equations
- Optimization (utilizing gradients/Hessian)
- Inverse problems/data assimilation
- Neural networks
- Solving stiff ODEs

References



- *Baydin, A.G., Pearlmutter, B.A., Radul, A.A. and Siskind, J.M., 2017. Automatic differentiation in machine learning: a survey. Journal of Machine Learning Research (JMLR), 18(153), pp.1-153.*
- *Baydin, Atılım Gunes, Barak A. Pearlmutter, and Jeffrey Mark Siskind. 2016. "Tricks from Deep Learning." In 7th International Conference on Algorithmic Differentiation, Christ Church Oxford, UK, September 12-15, 2016.*
- *Baydin, Atılım Gunes, Barak A. Pearlmutter, and Jeffrey Mark Siskind. 2016. "DiffSharp: An AD Library for .NET Languages." In 7th International Conference on Algorithmic Differentiation, Christ Church Oxford, UK, September 12-15, 2016.*
- *Griewank, A. and Walther, A., 2008. Evaluating derivatives: principles and techniques of algorithmic differentiation (Vol. 105). SIAM.*

Thank you.