Automatic Differentiation for Inverse Modeling Notes

Toby Harvey

December 14, 2025

1 Introduction

Some parameters in ice sheet models are very hard to get observations of (basal friction coefficients, rheological coefficients, etc...). One strategy for estimating these parameters is to systematically adjust them so that outputs of the model best fit data that is observable. A cost function $\mathcal{J}(\alpha, u^{obs}, \hat{u}(\alpha))$, where α is the model parameter, is used to quantify the error between the model outputs $\hat{u}$ and the observable data u^{obs}. Most methods of minimizing $\mathcal{J}$ (gradient decent, etc...) require taking the derivative of the cost function with respect to the parameter, $\frac{d\mathcal{J}}{d\alpha}$. By the chain rule this is computed as:

$$\frac{d\mathcal{J}}{d\alpha} = \frac{\partial \mathcal{J}}{\partial \alpha} + \frac{\partial \mathcal{J}}{\partial \hat{u}} \frac{\partial \hat{u}}{\partial \alpha}.$$

The first two derivatives are easy to compute analytically, but the 3rd one is much harder. One possible solution is to try to compute it via perturbing the parameter of interest slightly and taking a finite difference:

$$\frac{\partial \hat{u}}{\partial \alpha} \approx \frac{\hat{u}^{pert} - \hat{u}}{\alpha^{pert} - \alpha}.$$

This requires running your model at least twice to compute a gradient for a single parameter, and in the case of a spatially variable parameter at least $O(n)$ times where n is the number of spatial degrees of freedom. The adjoint method is a method that requires more mathematical machinery, but requires solving only one additional model/system. The problem here is that it can be very difficult, if not impossible to derive this system. A third technique, automatic differentiation (AD), is summarized here. It is automatic in the sense that nothing coming from the specific model system has to be derived, although it is more expensive than the adjoint method.

2 Automatic Differentiation of a Simple Code/Function

Lets say we write a code that computes a very simple function:

$$y = \begin{bmatrix} f_1(x_1, x_2, x_3) \\ f_2(x_1, x_2, x_3) \end{bmatrix} = \begin{bmatrix} x_1 x_2 + \sin(x_3) - \log(x_1) \\ \sin(x_3) \end{bmatrix}.$$

At its essence, regardless of language or computer architecture, we need to follow the dependency graph in Figure 1. Where circular nodes are program inputs, and square nodes are intermediate computations. If we take this as our primary (think forward model) code, we can have a secondary code that, at runtime builds, the dependency graph in the background. This is the AD code.

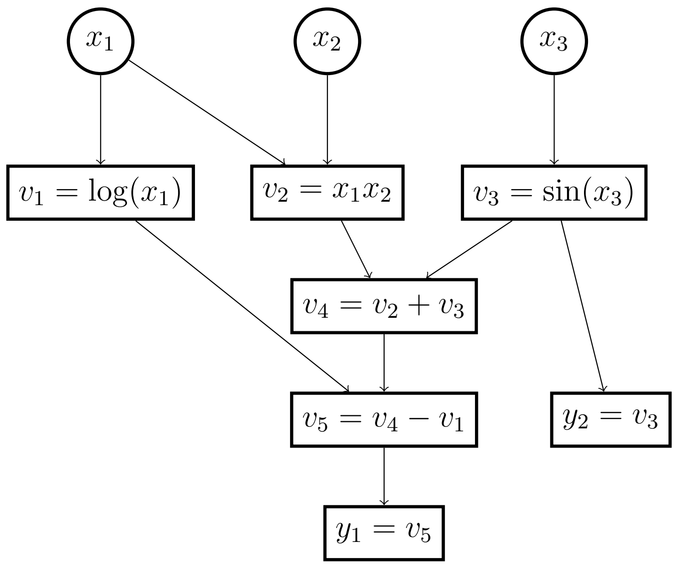


Figure 1: computational dependency graph.

2.1 Forward Mode AD

When we are building this graph, we of course aren't interested in the graph itself because the program is already doing those computations. Instead we can use the dependencies and operations to apply the chain rule as we build the graph. This is forward mode AD. We are interested in computing $\frac{\partial y}{\partial x}$, where x is one of the input variables x_1, x_2 or x_3, we just haven't decided which one yet, so we write it down generically. We then apply the chain rule:

$$\frac{\partial v}{\partial x} = \sum_{i \in \text{parents}} \frac{\partial v}{\partial u_i} \frac{\partial u_i}{\partial x}$$

where v is node/operation we are currently on and u are all its previous dependencies. This gives the following graph:

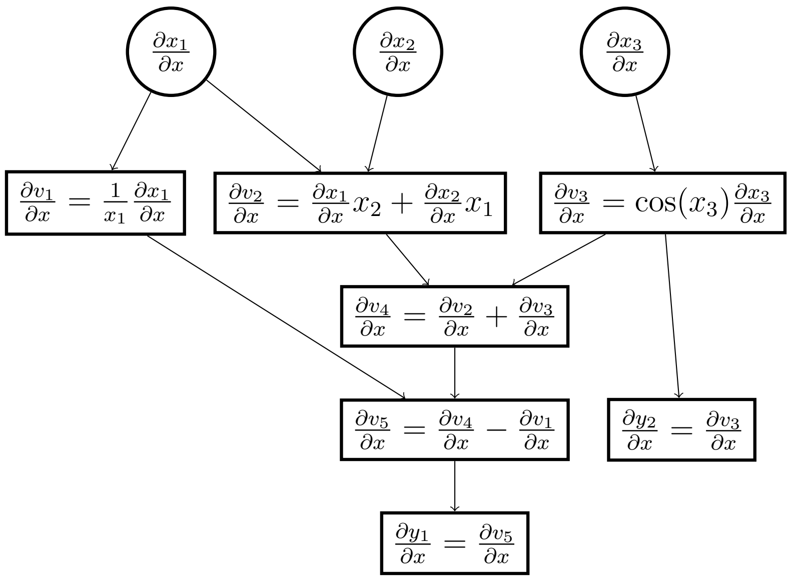

$\frac{\partial x_1}{\partial x}$
$\frac{\partial x_2}{\partial x}$
$\frac{\partial x_3}{\partial x}$
$\frac{\partial v_1}{\partial x} = \frac{1}{x_1} \frac{\partial x_1}{\partial x}$
$\frac{\partial v_2}{\partial x} = \frac{\partial x_1}{\partial x} x_2 + \frac{\partial x_2}{\partial x} x_1$
$\frac{\partial v_3}{\partial x} = \cos(x_3) \frac{\partial x_3}{\partial x}$
$\frac{\partial v_4}{\partial x} = \frac{\partial v_2}{\partial x} + \frac{\partial v_3}{\partial x}$
$\frac{\partial v_5}{\partial x} = \frac{\partial v_4}{\partial x} - \frac{\partial v_1}{\partial x}$
$\frac{\partial y_2}{\partial x} = \frac{\partial v_3}{\partial x}$
$\frac{\partial y_1}{\partial x} = \frac{\partial v_5}{\partial x}$

Figure 2: chain rule applied forward through the dependency graph.

We can construct this graph as the forward model program runs, as long as we have a table of derivatives of primitive operations like, multiplication, trigonometric, and transcendental functions.

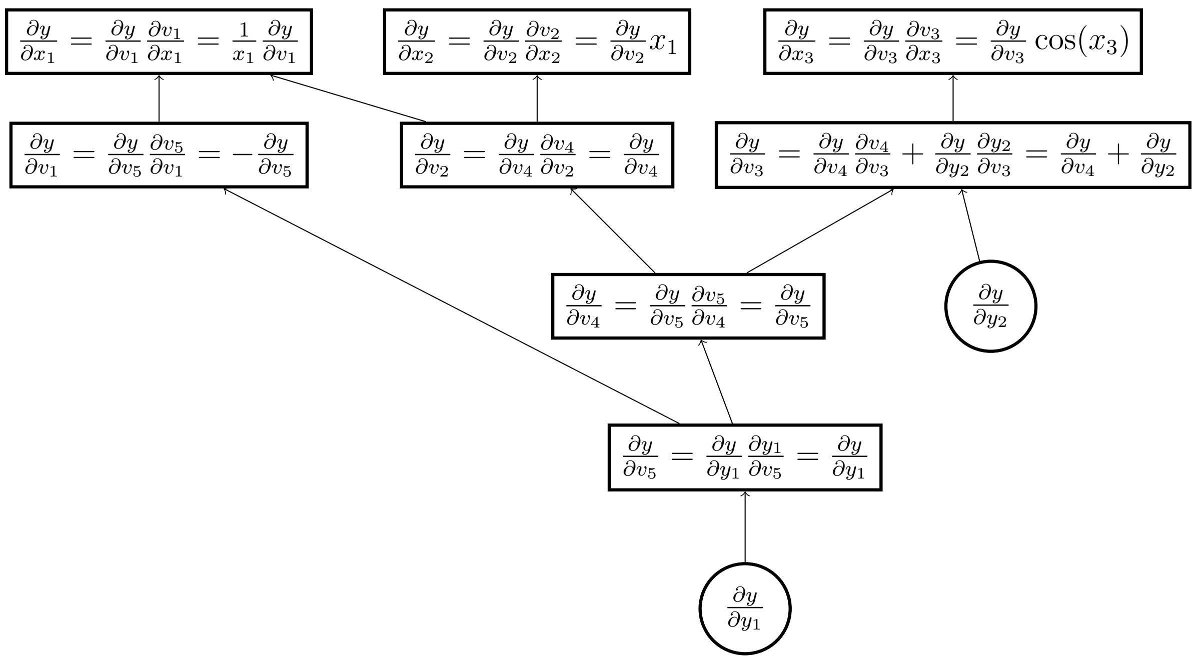
If we want to then compute a specific derivative (for instance $\frac{\partial y}{\partial x_1}$) we just need to seed the values for $\frac{\partial x_i}{\partial x}$. So for $\frac{\partial y}{\partial x_1}$ we seed with, $\frac{\partial x_1}{\partial x_1} = 1, \quad \frac{\partial x_2}{\partial x_1} = 0, \quad \frac{\partial x_3}{\partial x_1} = 0$. Then as long as the graph is in a topological ordering, we will have access to all of the previous derivatives we need to use to compute the next derivative on the next level. One observation to make is that we only need to traverse the graph 1 time to get the derivative of all outputs w.r.t. to one input, but if we want the derivative of all inputs w.r.t to y_1 or y_2, we need to traverse the graph 3 times (or n times where n is the number of input variables). In the case of a generic vector valued function $y = f(x) : \mathbb{R}^n \to \mathbb{R}^m$ then this method is faster at getting the jacobian if $n \ll m$.

2.2 Reverse Mode AD

Reverse mode AD is more confusing, but solves the problem about calculating the jacobian quickly if $m \ll n$. The intuition behind reverse mode is to completely ignore the fact that x are inputs and that y are outputs in the forward model, while retaining the idea that we want derivatives of $\frac{\partial y}{\partial x}$ and not $\frac{\partial x}{\partial y}$. Now, instead of seeding the inputs, we instead seed the outputs and use the chain rule:

$$\frac{\partial y}{\partial u} = \sum_{i \in \text{parents}} \frac{\partial y}{\partial v_i} \frac{\partial v_i}{\partial u}$$

in the reverse direction up the graph we to get:



$\frac{\partial y}{\partial x_1} = \frac{\partial y}{\partial v_1}\frac{\partial v_1}{\partial x_1} = \frac{1}{x_1}\frac{\partial y}{\partial v_1}$
$\frac{\partial y}{\partial x_2} = \frac{\partial y}{\partial v_2}\frac{\partial v_2}{\partial x_2} = \frac{\partial y}{\partial v_2}x_1$
$\frac{\partial y}{\partial x_3} = \frac{\partial y}{\partial v_3}\frac{\partial v_3}{\partial x_3} = \frac{\partial y}{\partial v_3}\cos(x_3)$
$\frac{\partial y}{\partial v_1} = \frac{\partial y}{\partial v_5}\frac{\partial v_5}{\partial v_1} = -\frac{\partial y}{\partial v_5}$
$\frac{\partial y}{\partial v_2} = \frac{\partial y}{\partial v_4}\frac{\partial v_4}{\partial v_2} = \frac{\partial y}{\partial v_4}$
$\frac{\partial y}{\partial v_3} = \frac{\partial y}{\partial v_4}\frac{\partial v_4}{\partial v_3} + \frac{\partial y}{\partial y_2}\frac{\partial y_2}{\partial v_3} = \frac{\partial y}{\partial v_4} + \frac{\partial y}{\partial y_2}$
$\frac{\partial y}{\partial v_4} = \frac{\partial y}{\partial v_5}\frac{\partial v_5}{\partial v_4} = \frac{\partial y}{\partial v_5}$
$\frac{\partial y}{\partial y_2}$
$\frac{\partial y}{\partial v_5} = \frac{\partial y}{\partial y_1}\frac{\partial y_1}{\partial v_5} = \frac{\partial y}{\partial y_1}$
$\frac{\partial y}{\partial y_1}$

Figure 3: chain rule applied backwards through the dependency graph.

Where all of the chain rules have been written out explicitly. Now if we instead seed with $\frac{\partial y_1}{\partial y_1} = 1, \quad \frac{\partial y_1}{\partial y_2} = 0$, we will compute the complete gradient ∇y_1 in one traversal. Similar to forward mode if we traverse the graph in a topological ordering we will have access to all derivatives that we need to compute the next node on the next level. In the reverse mode case we now compute the jacobian in fewer traversals when $m \ll n$. One disadvantage of reverse mode is that the graph must be constructed in advance of the actual reverse pass. In the forward case the construction and differentiation could be done in tandem.

3 Automatic Differentiation of Large Scale Model

A forward numerical PDE solver that generates $\hat{u}$ (the notation from the introduction) is a function $\hat{u}(x, \alpha, ...) : \mathbb{R}^n \to \mathbb{R}^m$ and so AD can be applied to it in the same way as the simple function above. In total reverse mode AD can be run to get $\frac{d\mathcal{J}}{d\alpha}$ in one pass since $\mathcal{J}$ is a scalar function. Even though $\mathcal{J}$ is a scalar function, we should remember that the dependency graph gets wide because $\frac{\partial \hat{u}}{\partial \alpha}$ is a $m \times n_\alpha$ matrix. This is unavoidable and both the width and the height of the graph also contribute to the efficiency, but not even close to having to traverse the graph n_α times. In other words the cost of reverse mode scales with solution/state size and not with control variable size.