VELOCITY ANALYSIS OF THREE LINK PLANAR ROBOT USING ADMAT 2.0

Table of Contents

Automatic Differentiation:

Automatic Differentiation (A.D.) is a technique to evaluate the derivatives of a function defined by a computer program.

In mathematics and computer algebra, automatic differentiation (AD), also called algorithmic differentiation or computational differentiation is a set of techniques to numerically evaluate the derivative of a function specified by a computer program. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.). By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, accurately to working precision, and using at most a small constant factor more arithmetic operations than the original program.

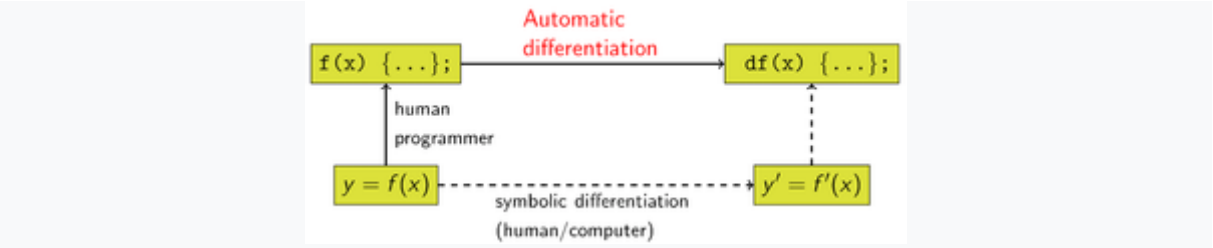


Figure 1: How automatic differentiation relates to symbolic differentiation

Automatic differentiation is not:

- Symbolic differentiation, nor
- Numerical differentiation (the method of finite differences).

These classical methods run into problems: symbolic differentiation leads to inefficient code (unless carefully done) and faces the difficulty of converting a computer program into a single expression, while numerical differentiation can introduce round-off errors in the discretization process and cancellation. Both classical methods have problems with calculating higher derivatives, where the complexity and errors increase. Finally, both classical methods are slow at computing the partial derivatives of a function with respect to *many* inputs, as is needed for gradient-based optimization algorithms. Automatic differentiation solves all of these problems, at the expense of introducing more software dependencies.

The chain rule, forward and reverse accumulation:

Fundamental to AD is the decomposition of differentials provided by the chain rule. For the simple composition $y = g(h(x)) = g(w)$ the chain rule gives

$$\frac{dy}{dx} = \frac{dy}{dw}\frac{dw}{dx}$$

Usually, two distinct modes of AD are presented, forward accumulation (or forward mode) and reverse accumulation (or reverse mode). Forward accumulation specifies that one traverses the chain rule from inside to outside (that is, first one computes dw/dx and then dy/dw, while reverse accumulation has the traversal from outside to inside. Generally,

both forward and reverse accumulation are specific manifestations of applying the operator of program composition, with the appropriate one of the two mappings being fixed.

Forward accumulation

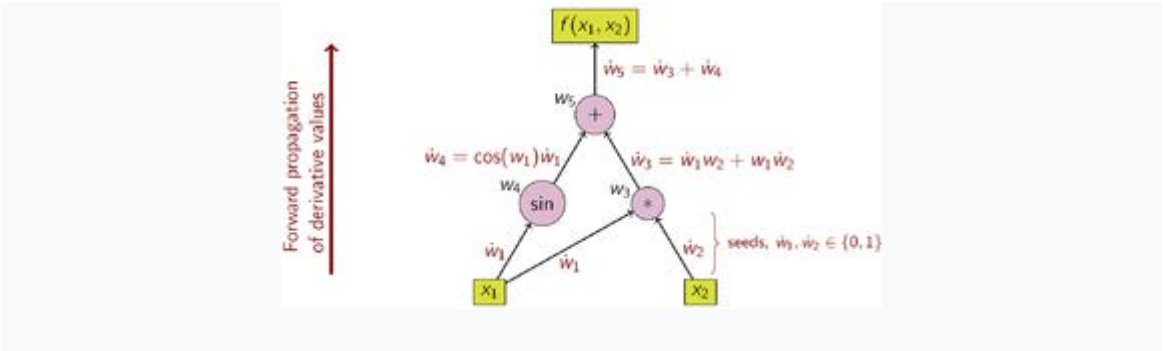


Figure 2: Example of forward accumulation with computational graph

In forward accumulation AD, one first fixes the *independent variable* to which differentiation is performed and computes the derivative of each sub-expression recursively. In a pen-and-paper calculation, one can do so by repeatedly substituting the derivative of the *inner* functions in the chain rule:

$$\frac{\partial y}{\partial x} = \frac{dy}{dw_1}\frac{dw_1}{dx} = \frac{\partial y}{\partial w_1}\left(\frac{\partial w_1}{\partial w_2}\frac{\partial w_2}{\partial x}\right) = \frac{\partial y}{\partial w_1}\left(\frac{\partial w_1}{\partial w_2}\left(\frac{\partial w_2}{\partial w_3}\frac{\partial w_3}{\partial x}\right)\right) = \cdots$$

This can be generalized to multiple variables as a matrix product of Jacobians.

Compared to reverse accumulation, forward accumulation is very natural and easy to implement as the flow of derivative information coincides with the order of evaluation. One simply augments each variable w with its derivative $\dot{w}$ (stored as a numerical value, not a symbolic expression),

$$\dot{w} = \frac{\partial w}{\partial x}$$

as denoted by the dot. The derivatives are then computed in sync with the evaluation steps and combined with other derivatives via the chain rule.

The computational complexity of one sweep of forward accumulation is proportional to the complexity of the original code.

Forward accumulation is more efficient than reverse accumulation for functions $f : \mathbb{R}^n \to \mathbb{R}^m$ with $m \gg n$ as only n sweeps are necessary, compared to m sweeps for reverse accumulation.

Reverse accumulation

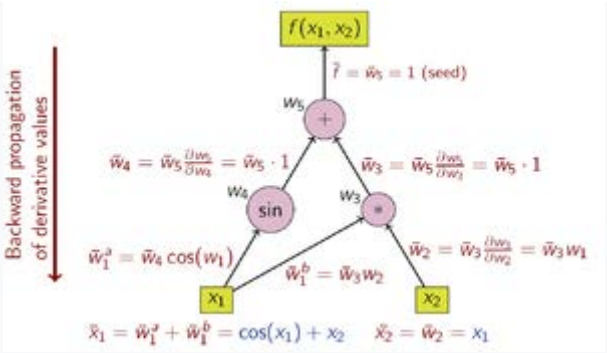


Figure 3: Example of reverse accumulation with computational graph

In reverse accumulation AD, one first fixes the *dependent variable* to be differentiated and computes the derivative *with respect to* each sub-expression recursively. In a pen-and-paper calculation, one can perform the equivalent by repeatedly substituting the derivative of the *outer* functions in the chain rule:

$$\frac{\partial y}{\partial x} = \frac{dy}{dw_1}\frac{dw_1}{dx} = \left(\frac{\partial y}{\partial w_2}\frac{\partial w_2}{\partial w_1}\right)\frac{\partial w_1}{\partial x} = \left(\left(\frac{\partial y}{\partial w_3}\frac{\partial w_3}{\partial w_2}\right)\frac{\partial w_2}{\partial w_1}\right)\frac{\partial w_1}{\partial x} = \cdots$$

In reverse accumulation, the quantity of interest is the *adjoint*, denoted with a bar ($\bar{w}$); it is a derivative of a chosen dependent variable with respect to a subexpression w:

$$\bar{w} = \frac{\partial y}{\partial w}$$

Reverse accumulation traverses the chain rule from outside to inside, or in the case of the computational graph in Figure 3, from top to bottom. There is only one seed for the derivative computation, and only one sweep of the computational graph is needed in order to calculate the (two-component) gradient. This is only half the work when compared to forward accumulation, but reverse accumulation requires the storage of the intermediate variables w_i as well as the instructions that produced them in a data structure known as a Wengert list (or "tape"),which may represent a significant memory issue if the computational graph is large. This can be mitigated to some extent by storing only a subset of the intermediate variables and then reconstructing the necessary work variables by repeating the evaluations, a technique known as checkpointing.

The data flow graph of a computation can be manipulated to calculate the gradient of its original calculation. This is done by adding an adjoint node for each primal node, connected by adjoint edges which parallel the primal edges but flow in the opposite direction. The nodes in the adjoint graph represent multiplication by the derivatives of the functions calculated by the nodes in the primal. For instance, addition in the primal causes fanout in the adjoint; fanout in the primal causes addition in the adjoint; a unary function $y = f(x)$ in the primal causes $\bar{x} = \bar{y}\, f'(x)$ in the adjoint; etc.

Reverse accumulation is more efficient than forward accumulation for functions $f : \mathbb{R}^n \to \mathbb{R}^m$ with $m \ll n$ as only m sweeps are necessary, compared to n sweeps for forward accumulation.

Reverse mode AD was first published in 1970 by Seppo Linnainmaa in his master thesis. Backpropagation of errors in multilayer perceptrons, a technique used in machine learning, is a special case of reverse mode AD.

Beyond forward and reverse accumulation

Forward and reverse accumulation are just two (extreme) ways of traversing the chain rule. The problem of computing a full Jacobian of $f : \mathbb{R}^n \to \mathbb{R}^m$ with a minimum number of arithmetic operations is known as the *optimal Jacobian accumulation* (OJA) problem, which is NP-complete. Central to this proof is the idea that there may exist algebraic dependencies between the local partials that label the edges of the graph. In particular, two or more edge labels may be recognized as equal. The complexity of the problem is still open if it is assumed that all edge labels are unique and algebraically independent.

ADMAT V2.0: Automatic Differentiation Toolbox:

Many scientific computing tasks require the repeated computation of derivatives. Hand-coding of derivative functions can be tedious, complex, and error-prone. Moreover, the computation of first and second derivatives, and sometimes the Newton step, is often a dominant expense in a scientific computing code. Derivative approximations such as finite-differences involve additional errors. This toolbox is designed to help a MATLAB user compute first and second derivatives and related structures efficiently, accurately, and automatically. ADMAT employs many sophisticated techniques, exploiting sparsity and structure, to gain efficiency in the calculation of derivative structures (e.g., gradients, Jacobians, and Hessians).
Moreover, ADMAT can directly and effectively calculate Newton steps for nonlinear systems.
To use ADMAT to evaluate a smooth nonlinear 'objective function' at a given argument, a MATLAB user need only supply an M-file. On request and when appropriate,
ADMAT will ensure that in addition to the objective function evaluation, the Jacobian matrix, the Hessian matrix, and possibly the Newton step will also be evaluated at the supplied argument. The user need not supply derivative codes or approximation schemes.

ADMAT 2.0 Features:

- Efficient gradient computation (by 'reverse mode').
- Efficient evaluation of sparse Jacobian and Hessian matrices.
- A template design for the efficient calculation of 'structured' Jacobian and Hessian matrices.
- Efficient direct computation of Newton steps (In some cases avoiding the full computation of the Jacobian and/or Hessian matrix).
- Mechanisms and procedures for combining automatic differentiation of M-files with the finite differencing approximation for MEX files (for C and Fortran subfunctions).
- " Forward" mode of automatic differentiation: A new MATLAB class "deriv" which overloads more than 100 MATLAB built-in functions.
- "Reverse" mode of automatic differentiation: A new MATLAB class "derivtape" which uses a virtual tape to record all functions and overloads more than 100 MATLAB built-in functions.
- MATLAB interpolation function INTERP1 is available.

Kinematic Model:

In order to specify the geometry of the planar 3R robot, we require three parameters, L_1, L_2 and L_3. These are the three link lengths. In the figure, the three joint angles are labelled θ_1, θ_2 and θ_3. These are obviously variable. The precise definitions for the link lengths and joint angles are as follows. For each pair of adjacent axes, we can define a common normal or the perpendicular between the axes.

•The i^{th} common normal is the perpendicular between the axes for joint i and joint i+1.

•The i^{th} link length is the length of the ith common normal, or the distance between the axes for joint i and joint i+1.

•The i^{th} joint angle is the angle between the $(i-1)^{th}$ common normal and ith common normal measured counter clockwise going from the $(i-1)^{th}$ common normal to the ith common normal.

Direct Kinematics:

From basic trigonometry, the position and orientation of the end effector can be written in terms of the joint coordinates in the following way:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} L_1(\cos(\theta_1) + L_2(\cos(\theta_1 + \theta_2) + L_3(\cos(\theta_1 + \theta_2 + \theta_3) \\ L_1(\sin(\theta_1) + L_2(\sin(\theta_1 + \theta_2) + L_3(\sin(\theta_1 + \theta_2 + \theta_3) \end{bmatrix}$$

Note that all the angles have been measured counter clockwise and the link lengths are assumed to be positive going from one joint axis to the immediately distal joint axis.

The equation is a set of three nonlinear equations that describe the relationship between end effector coordinates and joint coordinates. Notice that we have explicit equations for the end effector coordinates in terms of joint coordinates. However, to find the joint coordinates for a given set of end effector coordinates (yn), one needs to solve the nonlinear equations for θ_1, θ_2, and θ_3.

Velocity Analysis

When controlling a robot to go from one position to another, it is not just enough to determine the joint and end effector coordinates of the target position. It may be necessary to continuously control the trajectory or the path taken by the robot as it moves toward the target position. This is essential to avoid obstacles in the workspace. More importantly, there are tasks where the trajectory of the end effector is critical. For example, when welding, it is necessary to maintain the tool at a desired orientation and a fixed distance away from the workpiece while moving uniformly5 along a desired path. Thus one needs to control the velocity of the end effector or the tool during the motion. Since the control action occurs at the joints, it is only possible to control the joint velocities. Therefore, there is a need to be able to take the desired end effector velocities and calculate from them the joint velocities.

The rate of change of end effector coordinates with respect to joint coordinates is given by the Jacobian Matrix, A as

$$A = \begin{bmatrix} \frac{\partial x}{\partial \theta_1} & \frac{\partial x}{\partial \theta_2} & \frac{\partial x}{\partial \theta_3} \\ \frac{\partial y}{\partial \theta_1} & \frac{\partial y}{\partial \theta_2} & \frac{\partial y}{\partial \theta_3} \end{bmatrix}$$

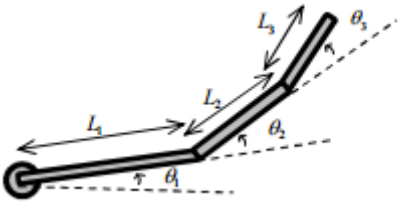
For a given tip velocity $\dot{x}$, one way to choose a corresponding set of joint velocities is to choose the joint velocities that minimizes $||\dot{\theta}||_2$ while satisfying $\dot{x} = A\dot{\theta}$

In many cases, it is more costly to move some joints than others. Rather than minimize the standard 2-norm as in part (d), we could find a solution to $\dot{x} = A\dot{\theta}$ that minimizes a different norm $||\dot{\theta}||_D = \dot{\theta^T} D\dot{\theta} = (d_1\dot{\theta}_1^2 + d_2\dot{\theta}_2^2 + d_3\dot{\theta}_3^2)^{\frac{1}{2}}$, where D is a diagonal matrix with strictly positive weights d_i on the diagonal. Each weight penalizes motion of the corresponding joint. The solution to

$\dot{x} = A\dot{\theta}$ that minimizes $||\dot{\theta}||_D$ is given by $\dot{\theta} = D^{-1}A^T(AD^{-1}A^T)^{-1}\dot{x}$.

Problem Statement:

Consider the 3 link planar robot. The position x of the tip relative to robot's base in terms of the joint angles $[\theta_1, \theta_2, \theta_3] = [0, pi/4, pi/4]$ and $L_1 = 4$, $L_2 = 2$, $L_3 = 1$.



a) We find A, using ADMAT 2.0, using the functions
di = ADfun('direct_kinematics',n);
[f, A] = feval(di,theta);

$$A = \begin{bmatrix} -2.4142 & -2.4142 & -1.0000 \\ 5.4142 & 1.4142 & 0.0000 \end{bmatrix}$$

b) Given tip velocity $\dot{x} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, we find the solution to $\dot{x} = A\dot{\theta}$ that minimises the 2-norm.

min_norm = A'*inv(A*A')*x;

$$= \begin{bmatrix} 0.2306 \\ -0.1758 \\ -0.1323 \end{bmatrix}$$

c) In many cases, it is more costly to move some joints than others. Rather than minimizing the 2-norm, we find solution to $\dot{x} = A\dot{\theta}$ that minimises the different norm. For the x from previous step, and for d1 = 6, d2 = 3, d3 = 1.
norm_d = inv(D)*A'*inv(A*inv(D)*A')*x;

$$= \begin{bmatrix} 0.2047 \\ -0.0768 \\ -0.3090 \end{bmatrix}$$

Codes:

Velocity Analysis:

velocity_analysis.m

```
n = 3;
%angle
theta = [0, pi/4, pi/4];
%rate of change of the end effector coordinates
%with respect to the joint coordinates
di = ADfun('direct_kinematics',n);
[f, A] = feval(di,theta);

%for given tip velocity x, choose a corresponding set of joint velocities
%is to choose the joint velocities that minimizes 2-norm
x = [0;1];
min_norm = A'*inv(A*A')*x;

%In many cases, it is more costly to move some joins than other.
%so, instead of minimizing 2-norm, we find a solution which minimizes a
%different norm
D = diag([6,3,1]);
norm_d = inv(D)*A'*inv(A*inv(D)*A')*x;
```

Direct Kinematics:

direct_kinematics .m

```
function x = direct_kinematics(theta, Extra)
L1 = 4;
L2 = 2;
L3 = 1;
c1 = cos(theta(1));
c2 = cos(theta(1)+theta(2));
c3 = cos(theta(1)+theta(2)+theta(3));
s1 = sin(theta(1));
s2 = sin(theta(1)+theta(2));
s3 = sin(theta(1)+theta(2)+theta(3));
x(1) = (L1*c1)+(L2*c2)+(L3*c3);
x(2) = (L1*s1)+(L2*s2)+(L3*s3);
```