

SENSITIVITY ANALYSIS AND AUTOMATIC DIFFERENTIATION

ADAM ATTARIAN

In the study of mathematical models, it is important to understand how changes in the parameters affect the model output. This is referred to sensitivity analysis, where the derivatives of the model with respect to the parameters are called the sensitivities. We will formulate the problem as follows.

Sensitivity Analysis. Suppose our mathematical model is a system of ordinary differential equations,

$$(1) \quad \frac{d\mathbf{x}(t)}{dt} = f(t, \mathbf{x}(\mathbf{q}); \mathbf{q}),$$

where $\mathbf{x} \in \mathbb{R}^n$ denotes the state variables to be solved for and $\mathbf{q} \in \mathbb{R}^m$ denotes the parameters. After differentiating both sides with respect to \mathbf{q} and switching the order of differentiation we obtain the $n + mn$ dimensional system of differential equations for both the model and sensitivities

$$(2) \quad \begin{aligned} \frac{d\mathbf{x}(t)}{dt} &= f(t, \mathbf{x}(\mathbf{q}); \mathbf{q}) \\ \frac{d}{dt} \frac{\partial \mathbf{x}}{\partial \mathbf{q}} &= \frac{\partial f}{\partial \mathbf{x}} \frac{\partial \mathbf{x}}{\partial \mathbf{q}} + \frac{\partial f}{\partial \mathbf{q}}. \end{aligned}$$

The state variables $\frac{\partial \mathbf{x}}{\partial \mathbf{q}}$ are solved for using an ODE integrator such as `ode15s` in MATLAB, whereas the derivatives $\frac{\partial f}{\partial \mathbf{x}}, \frac{\partial f}{\partial \mathbf{q}}$, are obtained through automatic differentiation (AD). We assume that the initial conditions on the sensitivities are zero, as the initial conditions of the model would not be considered to be dependent on the model parameters.

Once we have the sensitivity functions computed, we relate to these functions as we would any other derivative function: when the sensitivity of one parameter is negative, that models state is decreasing and so forth. After computing the sensitivity functions, we then compute the *relative ranking* of these functions using a modified ℓ_2 norm,

$$(3) \quad \left\| \frac{\partial \mathbf{x}_i}{\partial \mathbf{q}_j} \right\|_2 = \left[\frac{1}{t_f - t_0} \int_{t_0}^{t_f} \left(\frac{\partial \mathbf{x}_i}{\partial \mathbf{q}_j} \right)^2 dt \right]^{1/2} \frac{\mathbf{q}_j}{\max \mathbf{x}_i},$$

where t_f, t_0 are the final and initial time points, respectively. In practice, we approximate the integral with a trapezoidal method. After computing these sensitivity ranks, we garner an idea of which parameters the model is most sensitive to, and by extension which parameters the inverse problem will reveal as being identifiable.

Example. Consider the spring mass model that has been normalized with respect to mass,

$$(4) \quad \ddot{x} + C\dot{x} + Kx = 0,$$

where C is the damping divided by mass and K is the spring constant divided by mass. To compute the sensitivities we first recast the model as a system of first order equations:

$$(5) \quad \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x \\ \dot{x} \end{bmatrix} \implies \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -Cx_2 - Kx_1 \end{bmatrix}$$

And then (manually, by hand) taking the partial derivatives we have

$$(6) \quad \frac{d}{dt} \frac{\partial x_1}{\partial C} = \frac{\partial x_2}{\partial C}$$

$$(7) \quad \frac{d}{dt} \frac{\partial x_1}{\partial K} = \frac{\partial x_2}{\partial K}$$

$$(8) \quad \frac{d}{dt} \frac{\partial x_2}{\partial C} = -x_2 - C \frac{\partial x_2}{\partial C} - K \frac{\partial x_1}{\partial C}$$

$$(9) \quad \frac{d}{dt} \frac{\partial x_2}{\partial K} = -C \frac{\partial x_2}{\partial K} - x_1 - K \frac{\partial x_1}{\partial K}$$

You can see how for large compartment models with many states and even more parameters this is a daunting, error prone task. After the partials are computed, we would then solve the system of ODEs with MATLAB or another appropriate integrator. This analysis a local concept, as the model is being analyzed in a neighborhood about a nominal \mathbf{q} value. We could also solve for the model sensitivities by a central finite difference scheme,

$$(10) \quad \frac{\partial x_i}{\partial q_j} \approx \frac{f(x_i(q_j + e_j h_j)) - f(x_i(q_j - e_j h_j))}{2h_j},$$

where e_j is the elementary basis vector and h_j the step size. This is easy to implement, but the solutions are only approximate. The final method to compute sensitivities with high accuracy and is through automatic differentiation, and we will discuss this next.

Automatic Differentiation (AD). AD is essentially the application of the chain rule on an expression. When most computer algebra systems like Maple symbolically differentiate a function, it does not use the things we learned in calculus class like the power rule or that the derivative of sine is cosine. When AD is used, it explicitly breaks down a function into its basic constituent operations and then applies the chain rule to get an answer. It is not a numerical approximation like finite difference is, it is machine-precision exact.

The code we will be using is called myAD, written by our collaborator Martin Fink from Oxford. The code is available from the Matlab file exchange. Once downloaded, place the code in your matlab working path. I'll show you how to use the code in this example.

Example 1. Suppose we wish to compute the derivative of a vectored value function, $f : \mathbb{R}^2 \rightarrow \mathbb{R}^2$, at the point $(1, 1)$ where

$$f(x_1, x_2) = \begin{bmatrix} 2x_1 + x_2^2 \\ x_1^3 + x_1 x_2 \end{bmatrix}.$$

The derivative is given by the Jacobian matrix,

$$f'(x_1, x_2) = \left[\begin{array}{cc} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{array} \right] \Big|_{(1,1)} \implies \left[\begin{array}{cc} 2 & 2x_2 \\ 3x_1^2 + x^2 & x_1 \end{array} \right] \Big|_{(1,1)} = \left[\begin{array}{cc} 2 & 2 \\ 4 & 1 \end{array} \right]$$

The following Matlab function uses the myAD code to compute the Jacobian:

```

1 function adex
2
3 x=[1; % the point where we are taking the derivative
4     1];
5
6 bigX=myAD(x); % convert to an AD variable
7 adOut=fun(bigX); % evaluate the function at the AD variable
8
9 dfdx=getderivs(adOut) % get the jacobian
10
11 function f=fun(x)
12
13 f=[2*x(1)+x(2)^2; % this is the function we are taking the derivative of
14     x(1)^3+x(1)*x(2)];

```

Running the code returns the result we expect.

Example 2. Now that we know how to use the code, we'll write a Matlab function to compute the sensitivities of the spring model from before. We'll first need to augment the spring model with the AD code to compute the sensitivities. This code will do the job.

```

1 function dx=smodsens(t,x,q)
2
3 % the ode file that will be integrated to compute the sens
4
5 n=2; % number of states
6 m=2; % number of params
7
8 dx=zeros(n+m*n,1); % init the dx
9
10 % implement the spring model, where C=q(1), K=q(2). AD likes the dot mult.
11 dx(1:n)=[x(2);
12         -q(1).*x(2)-q(2).*x(1)];
13
14 x0=x(1:n); % these are the states
15 q0=q; % save the params for consistency
16
17 % create the state and param AD variables
18 bigx=myAD(x0);
19 bigq=myAD(q0);

```

```

20
21 % its easiest to recreate the model equations as a subfunction. otherwise, it
22 % becomes a recursive problem that i'm not quite sure about.
23
24 % eval the model with the AD variables
25 resultx=smod(t,bigx,q0);
26 resultp=smod(t,x0,bigq);
27
28 % pull the derivatives out
29 dfdx=getderivs(resultx);
30 dfdq=getderivs(resultp);
31
32 % reshape the current sensitivites into a matrix so we can update
33 sens=reshape(x(n+1:end),n,m);
34
35 % calculus says...
36 dsens=dfdx*sens+dfdq;
37
38 % need to reshape to a vector for export to the integrator. sorts by
39 % columns — the column sort determines the output order.
40 dx(n+1:end)=dsens(:);
41
42 % note the order of dx=[x1 x2 dx1dq1 dx2dq1 dx1dq2 dx2dq2]
43
44 function dx=smod(t,x,q)
45
46 dx=[x(2);
47      -q(1)*x(2)-q(2)*x(1)];

```

Our integration call and the computation of the relative sensitivities can be done for a small system with the code below. For larger systems, we have a more robust code that orders all of the sensitivities into 3D arrays.

```

1 clear all;
2
3 K=2; C=50; q=[K C]';
4 x0=[1 0]';
5 m=2; % params
6 n=2; % states
7
8 % call the integrator on the augmented system
9 [tsens,regsens]=ode15s(@smodsens,[0 5],[x0; zeros(m*n,1)],[],q);
10
11 % we'll now pick out the states and the sensitivities.
12
13 x=regsens(:,1); % this is the positions
14 vel=regsens(:,2); % call this v

```

```
15 dxdK=regsens(:,3); % reshape stacks by columns, so...
16 dvdK=regsens(:,4);
17 dxdC=regsens(:,5);
18 dvdC=regsens(:,6);
19
20 % compute relative sensitivities, dxdq*(q/max(x))
21
22 dxdKrel=dxdK.*(K./max(dxdK));
23 dvdKrel=dvdK.*(K./max(dvdK));
24 dxdCrel=dxdC.*(C./max(dxdC));
25 dvdCrel=dvdC.*(C./max(dvdC));
26
27 % and then we'd compute the ranks, etc...
```