



# MANE 6961:

## Adjoint for Scientists and Engineers

Lecture 11

Prof. Hicken  
JEC 2036

# Lecture Overview

The purpose of this lecture is to review how adjoints can be used to inexpensively compute gradients of functionals.

- We covered this subject in the first lecture, but this will be slightly “deeper dive.”
- After this lecture, you will have all the knowledge necessary to complete Assignment 2.

Time permitting, I will also review the forward and reverse mode of algorithmic differentiation, which are useful to know for next lecture.

# Direct Sensitivity Method

# Direct Sensitivities: Discrete

Throughout this lecture, we will use  $\alpha \in \mathbb{R}^n$  to denote parameters that we want to differentiate with respect to.

- $\alpha$  may be a set of shape parameters, a distributed control, or problem parameters like  $\text{Re}$  or  $\text{M}$ .

Let  $R_h : \mathbb{R}^s \times \mathbb{R}^n \rightarrow \mathbb{R}^s$  be a nonlinear state equation, with unique solution  $u_h \in \mathbb{R}^s$ :

$$R_h(u_h, \alpha) = 0.$$

Furthermore, let  $J_h : \mathbb{R}^s \times \mathbb{R}^n \rightarrow \mathbb{R}$  be a nonlinear, continuously differentiable function.

## Direct Sensitivities: Discrete (cont.)

The total derivative of  $J_h$  with respect to  $\alpha_i$  is

$$\frac{DJ_h}{D\alpha_i} =$$

# Direct Sensitivities: Discrete (cont.)

# Direct Sensitivities: Discrete (cont.)

## Definition: Direct Sensitivity Method

The total derivative of  $J_h$  with respect to  $\alpha_i$  is

$$\frac{DJ_h}{D\alpha_i} = \frac{\partial J_h}{\partial \alpha_i} + \frac{\partial J_h}{\partial u_h} v_h$$

where  $v_h \in \mathbb{R}^s$  is the direct sensitivity and satisfies

$$\frac{\partial R_h}{\partial u_h} v_h = -\frac{\partial R_h}{\partial \alpha_i}.$$

# Direct Sensitivities: Continuous

One can also derive and use a direct sensitivity BVP:

$$\frac{DJ}{D\alpha_i} = J'[\alpha_i] + J'[u]v,$$

where  $v \in \mathcal{V}$  satisfies the linear BVP

$$\begin{aligned} N'[u]v &= -N'[\alpha_i], & \forall x \in \Omega, \\ B'[u]v &= -B'[\alpha_i], & \forall x \in \Gamma. \end{aligned}$$



# Adjoint Sensitivity Method

# Adjoint-Based Sensitivities

There are two ways to derive the adjoint sensitivity method. The first, which we saw in the first lecture, is to substitute the direct sensitivity into the derivative, and identify the adjoint with the appropriate product:

$$\begin{aligned}\frac{DJ_h}{D\alpha_i} &= \frac{\partial J_h}{\partial \alpha_i} + \frac{\partial J_h}{\partial u_h} v_h \\ &= \frac{\partial J_h}{\partial \alpha_i} - \frac{\partial J_h}{\partial u_h} \left( \frac{\partial R_h}{\partial u_h} \right)^{-1} \frac{\partial R_h}{\partial \alpha_i}\end{aligned}$$

# Lagrangian Approach

The second approach to deriving the adjoint sensitivity method, is to introduce the Lagrangian:

$$L_h(u_h, \psi_h, \alpha) = J_h(u_h, \alpha) - \psi_h^T R_h(u_h, \alpha)$$

- The minus sign is inconsequential
- Since  $R_h(u_h, \alpha) = 0$ , we have  $L_h = J_h$ .
- However, this does not imply that  $\partial J_h / \partial \alpha = \partial L_h / \partial \alpha$ .

The key idea is that  $L_h$  should be stationary with respect to changes in  $\psi_h$  and  $u_h$ .

# Lagrangian Approach (cont.)

# Lagrangian Approach (cont.)

## Definition: Adjoint Sensitivity Method

The total derivative of  $J_h$  with respect to  $\alpha_i$  is

$$\frac{DJ_h}{D\alpha_i} = \frac{\partial J_h}{\partial \alpha_i} - \psi_h^T \frac{\partial R_h}{\partial \alpha_i}$$

where  $\psi_h \in \mathbb{R}^s$  satisfies the discrete adjoint equation

$$\left( \frac{\partial R_h}{\partial u_h} \right)^T \psi_h = \frac{\partial J_h}{\partial u_h}.$$

# Continuous or Discrete Adjoint?

Suppose you use the continuous, rather than discrete, adjoint.

- Recall, the continuous-adjoint approach involves finding the adjoint BVP, discretizing it, and solving the resulting discretized problem.

Let  $\tilde{\psi}_h$  denote the solution to the discretized adjoint BVP, and define

$$\frac{\widetilde{DJ}_h}{D\alpha_i} = \frac{\partial J_h}{\partial \alpha_i} - \tilde{\psi}_h^T \frac{\partial R_h}{\partial \alpha_i}$$

## Continuous or Discrete Adjoint? (cont.)

Therefore, the continuous adjoint approach does not, in general, produce a gradient that is consistent with the exact derivative,  $DJ_h/D\alpha$ .

- This can cause problems when the continuous adjoint is used with conventional optimization algorithms, which expect accurate derivatives.

On the other hand, Collis and Heinkenschloss [CH02] have shown that **the derivative based on the continuous adjoint converges at a faster rate to the true derivative.**

## Continuous or Discrete Adjoint? (cont.)

This brings us to one of the advantages of adjoint-consistent discretizations.

If you use an adjoint-consistent method, then you get the best of both worlds: exactness with respect to the finite-dimensional gradient, and optimal convergence to the infinite-dimensional gradient.



# Computing Derivatives: Forward-mode AD

# Algorithmic Differentiation

Algorithmic, or automatic, differentiation (AD) is an approach that systematically differentiates the source code.

- It is **not symbolic differentiation**: we do not get an explicit expression for the function's derivative.
- Instead, for a given set of input (design) variables,  $x$ , we get the derivative of the outputs with respect to those inputs at the given values: **we get numbers not an expression**.

See [GW09] for a comprehensive treatment of the subject.

# Algorithmic Differentiation (cont.)

There are two “modes” of AD.

**forward mode:** Also called the tangent mode. Computes directional derivatives of the form  $(\nabla f)^T p$ .

**reverse mode:** Also called the adjoint mode. Computes (weighted) gradients  $(\nabla f)$ .

The forward mode is easier to understand, so we will start there.

# Algorithmic Differentiation: forward mode

Consider a simple Julia function that computes

$$f(x_1, x_2) = x_1^2 + x_2 \sin(x_1).$$

```
1  function func(x1, x2)
2  # compute a simple function value
3  v1 = x1.^2
4  v2 = x2.*sin(v1)
5  f = v1 + v2
6  return f
7  end
```

- This is a toy example, but it models how we often use intermediate values ( $v1$  and  $v2$ ) to arrive at the value we want ( $f$ ).

# Algorithmic Differentiation: forward mode (cont.)

Suppose we want to compute  $Df/Dx_1$ , *at a particular value of  $x_1$  and  $x_2$* , using the above code.

- In the code,  $f$  does not depend explicitly on  $x_1$ .
- To find the desired derivative, we must account for how the intermediate values depend on  $x_1$ .
- How? Use the chain rule.

# Algorithmic Differentiation: forward mode (cont.)

$$\begin{aligned}
 \frac{Df}{Dx_1} &= \frac{\partial f}{\partial x_1} + \frac{\partial f}{\partial v_1} \frac{\partial v_1}{\partial x_1} + \frac{\partial f}{\partial v_2} \frac{Dv_2}{Dx_1} \\
 &= \frac{\partial f}{\partial x_1} + \frac{\partial f}{\partial v_1} \frac{\partial v_1}{\partial x_1} + \frac{\partial f}{\partial v_2} \frac{\partial v_2}{\partial v_1} \frac{\partial v_1}{\partial x_1} \\
 &= 0 + (1)(2x_1) + (1)(x_2 \cos(v_1))(2x_1)
 \end{aligned}$$

# Algorithmic Differentiation: forward mode (cont.)

$$\frac{Df}{Dx_1} = 2x_1 + x_2 \cos(x_1^2)2x_1$$

- As usual we use  $Df/Dx_1$  to denote the **total derivative** of  $f$  with respect to  $x_1$ .
- Similarly,  $Dv_2/Dx_1$  is the total derivative of  $v_2$  with respect to  $x_1$ .

# Algorithmic Differentiation: forward mode (cont.)

Now the question becomes, how do we implement this in the code?

- A key observation is that the total derivative of the intermediate values with respect to  $x_1$  can be found sequentially as we progress through the function
- That is, we compute the values of  $Dv_j/Dx_1$  together with the value  $v_j$



# Algorithmic Differentiation: forward mode (cont.)

Applying the above reasoning, we obtain a new function that computes  $f$  and  $df/dx_1$ , the total derivative  $Df/Dx_1$ .

```
1  function dfunc(x1, x2)
2  # compute a simple function value and its derivative w.r.t. x1
3  v1 = x1.^2
4  dv1dx1 = 2.0.*x1
5  v2 = x2.*sin(v1)
6  dv2dx1 = x2.*cos(v1).*dv1dx1
7  f = v1 + v2
8  dfdx1 = dv1dx1 + dv2dx1
9  return f, dfdx1
10 end
```

# Algorithmic Differentiation: forward mode (cont.)

- This example illustrates the **forward mode** of algorithmic differentiation.
- We have used an implementation of forward mode called source-code transformation.
- In object-oriented languages, the same result can be achieved by defining a new data type that tracks both the function values and their derivatives.

# Pros & Cons of Forward-mode AD

- ✓ no truncation error and no  $h$  to worry about!
- ✓ the cost of evaluating the differentiated function is only a small factor more than the original code (approximately twice the cost)
- ✗ requires access to the source code
- ✗ computational cost still scales with the  $\#$  of design variables

Cost is still proportional to the number of design variables, so the advantages over complex-step are minimal.

Kenobi: *That [method] was our last hope.*

Yoda: *No. There is another.*

# Exercise

Apply forward-mode to compute the partial derivative  $\partial f / \partial x_2$  of the function below.

```
1  function func(x1, x2)
2  # compute a simple function value
3  v1 = x1.^2
4  v2 = x2.*sin(v1)
5  f = v1 + v2
6  return f
7  end
```

# Computing Derivatives: Reverse-mode AD

# Reverse Mode

To understand the reverse mode of AD, we again consider the function

$$\begin{aligned} f(x_1, x_2) &= x_1^2 + x_2 \sin(x_1^2) \\ &= v_1 + v_2 \end{aligned}$$

where  $v_1 = x_1^2$  and  $v_2 = x_2 \sin(v_1)$ .

The total derivative of  $f$  with respect to  $x_i$  ( $i = 1, 2$ ) is

$$\frac{Df}{Dx_i} = \frac{\partial f}{\partial x_i} + \frac{\partial f}{\partial v_1} \frac{\partial v_1}{\partial x_i} + \frac{\partial f}{\partial v_2} \left( \frac{\partial v_2}{\partial v_1} \frac{\partial v_1}{\partial x_i} + \frac{\partial v_2}{\partial x_i} \right)$$

## Reverse Mode (cont.)

The forward mode works by evaluating the partial derivatives from right-to-left in the products:

$$\text{Step 1: } \frac{Df}{Dx_i} = \frac{\partial f}{\partial x_i} + \frac{\partial f}{\partial v_1} \frac{\partial v_1}{\partial x_i} + \frac{\partial f}{\partial v_2} \left( \frac{\partial v_2}{\partial v_1} \frac{\partial v_1}{\partial x_i} + \frac{\partial v_2}{\partial x_i} \right)$$

$$\text{Step 2: } \frac{Df}{Dx_i} = \frac{\partial f}{\partial x_i} + \frac{\partial f}{\partial v_1} \frac{\partial v_1}{\partial x_i} + \frac{\partial f}{\partial v_2} \left( \frac{\partial v_2}{\partial v_1} \frac{\partial v_1}{\partial x_i} + \frac{\partial v_2}{\partial x_i} \right)$$

$$\text{Step 3: } \frac{Df}{Dx_i} = \frac{\partial f}{\partial x_i} + \frac{\partial f}{\partial v_1} \frac{\partial v_1}{\partial x_i} + \frac{\partial f}{\partial v_2} \left( \frac{\partial v_2}{\partial v_1} \frac{\partial v_1}{\partial x_i} + \frac{\partial v_2}{\partial x_i} \right)$$

## Reverse Mode (cont.)

The reverse mode works by evaluating the partial derivatives from left-to-right in the products:

$$\text{Step 1: } \frac{Df}{Dx_i} = \frac{\partial f}{\partial x_i} + \frac{\partial f}{\partial v_1} \frac{\partial v_1}{\partial x_i} + \frac{\partial f}{\partial v_2} \left( \frac{\partial v_2}{\partial v_1} \frac{\partial v_1}{\partial x_i} + \frac{\partial v_2}{\partial x_i} \right)$$

$$\text{Step 2: } \frac{Df}{Dx_i} = \frac{\partial f}{\partial x_i} + \frac{\partial f}{\partial v_1} \frac{\partial v_1}{\partial x_i} + \frac{\partial f}{\partial v_2} \left( \frac{\partial v_2}{\partial v_1} \frac{\partial v_1}{\partial x_i} + \frac{\partial v_2}{\partial x_i} \right)$$

$$\text{Step 3: } \frac{Df}{Dx_i} = \frac{\partial f}{\partial x_i} + \frac{\partial f}{\partial v_1} \frac{\partial v_1}{\partial x_i} + \frac{\partial f}{\partial v_2} \left( \frac{\partial v_2}{\partial v_1} \frac{\partial v_1}{\partial x_i} + \frac{\partial v_2}{\partial x_i} \right)$$



## Reverse Mode (cont.)

To make this work in practice, new variables are introduced in the code that keep track of the total derivatives.

- We will use an overbar on variables to denote the total derivative of  $f$  with respect to that variable

$$\bar{f} = \frac{Df}{Df} = 1$$

$$\bar{v}_1 = \frac{Df}{Dv_1}$$

$$\bar{x}_1 = \frac{Df}{Dx_1}$$

$$\bar{v}_2 = \frac{Df}{Dv_2}$$

$$\bar{x}_2 = \frac{Df}{Dx_2}$$

- $\bar{x}_1$  and  $\bar{x}_2$  define the gradient we want

## Reverse Mode (cont.)

Recall our Julia implementation:

```
1  function func(x1, x2)
2  # compute a simple function value
3  v1 = x1.^2
4  v2 = x2.*sin(v1)
5  f = v1 + v2
6  return f
7  end
```

- For the reverse mode, we first evaluate lines 3-5
- Then we **step backwards through the code**, differentiating line by line

Initialize  $\bar{f} = 1$ ,  $\bar{v}_1 = 0$ ,  $\bar{v}_2 = 0$ ,  $\bar{x}_1 = 0$ ,  $\bar{x}_2 = 0$ .

# Reverse Mode (cont.)

```
1  function func(x1, x2)
2  # compute a simple function value
3  v1 = x1.^2
4  v2 = x2.*sin(v1)
5  f = v1 + v2
6  return f
7  end
```

## Reverse Mode (cont.)

Differentiating line 5 with respect to the **variables that appear on the right** (i.e.  $v_1$  and  $v_2$ ) gives us

$$\bar{v}_2 = \bar{v}_2 + \frac{\partial f}{\partial v_2} \bar{f} = 1$$

$$\bar{v}_1 = \bar{v}_1 + \frac{\partial f}{\partial v_1} \bar{f} = 1$$

# Reverse Mode (cont.)

```
1  function func(x1, x2)
2  # compute a simple function value
3  v1 = x1.^2
4  v2 = x2.*sin(v1)
5  f = v1 + v2
6  return f
7  end
```

## Reverse Mode (cont.)

Differentiating line 4 with respect to the variables on its right ( $x_2$  and  $v_1$ ) gives us

$$\bar{v}_1 = \bar{v}_1 + \frac{\partial v_2}{\partial v_1} \bar{v}_2 = 1 + x_2 \cos(v_1)$$

$$\bar{x}_2 = \bar{x}_2 + \frac{\partial v_2}{\partial x_2} \bar{v}_2 = \sin(v_1)$$

## Reverse Mode (cont.)

```
1  function func(x1, x2)
2  # compute a simple function value
3  v1 = x1.^2
4  v2 = x2.*sin(v1)
5  f = v1 + v2
6  return f
7  end
```

Finally, differentiating line 3 with respect to the variables on its right ( $x_1$ ) gives us

$$\bar{x}_1 = \bar{x}_1 + \frac{\partial v_1}{\partial x_1} \bar{v}_1 = 2x_1(1 + x_2 \cos(v_1))$$

## Reverse Mode (cont.)

Thus, in the end we have

$$\frac{Df}{Dx_1} = \bar{x}_1 = 2x_1(1 + x_2 \cos(v_1))$$
$$\frac{Df}{Dx_2} = \bar{x}_2 = \sin(v_1)$$

- You can easily verify that these are the correct values for the gradient of  $f$  evaluated at  $(x_1, x_2)$ .
- Putting this into the form of Julia, we get the following...



# Reverse Mode (cont.)

```
1  function dfunc_reverse(x1, x2)
2  # compute a simple function value and its gradient
3  v1 = x1.^2
4  v2 = x2.*sin(v1)
5  f = v1 + v2
6  # initialize bar variables
7  f_bar = 1.0; v2_bar = 0.0; v1_bar = 0.0
8  x2_bar = 0.0; x1_bar = 0.0;
9  v2_bar = v2_bar + f_bar # line 5
10 v1_bar = v1_bar + f_bar # line 5
11 v1_bar = v1_bar + x2.*cos(v1).*v2_bar # line 4
12 x2_bar = x2_bar + sin(v1).*v2_bar # line 4
13 x1_bar = x1_bar + 2.0*x1*v1_bar # line 3
```

## Reverse Mode (cont.)

```
14   dfdx1 = x1_bar; dfdx2 = x2_bar
15   return f, dfdx1, dfdx2
16   end
```

# Pros & Cons of Reverse-mode AD

- ✓ no truncation error and no  $h$  to worry about!
- ✓ the cost of evaluating the differentiated function is only a small factor more than the original code (approximately twice the cost)
- ✓ produces the entire gradient at once! (cost is virtually independent of the number of design variables)
- ✗ requires access to the source code

**Note:** you can only get the gradient of one output at a time, therefore...

# Important Guideline

Suppose you have a function that takes in  $n$  inputs and produces  $m$  outputs, and you need the gradient, with respect to all inputs, of all outputs. Generally speaking,

- if  $n > m$ , use the reverse mode
- if  $m > n$ , use the forward mode (or complex step)

# References

- [CH02] Scott S. Collis and Matthias Heinkenschloss, *Analysis of the streamline upwind/Petrov Galerkin method applied to the solution of optimal control problems*, Tech. Report TR02-01, Houston, Texas, 2002.
- [GW09] Andreas Griewank and Andrea Walther, *Evaluating derivatives: principles and techniques of algorithmic differentiation*, Society for Industrial & Applied Mathematics ; Cambridge University Press [distributor], 2009.