Introduction to Gradient-Based Optimisation

Part 5: Computation of derivatives

Dr. J.-D. Müller
School of Engineering and Materials Science,
Queen Mary, University of London
j.mueller@qmul.ac.uk

UK Fluids Network SIG on Numerical Optimisation with Fluids
Cambridge, 8-10 August 2018

Notes

Organisation of the lectures

1. Univariate optimisation
   - Bisection, Steepest Descent, Newton's method
2. Multivariate optimisation
   - Steepest descent, Newton's method
   - and line-search methods: Wolfe and Armijo conditions,
   - Quasi-Newton methods,
3. Constrained Optimisation:
   - Projected gradient methods,
   - Penalty methods, exterior and interior point methods,
   - SQP
4. Adjoint methods
   - Reversing time, Automatic Differentiation
   - Adjoint CFD codes
5. Gradient computation
   - Manual derivation, Finite Differences
   - Algorithmic and automatic differentiation, fwd and bkwd.

Notes

Outline

Notes

Outline

Notes

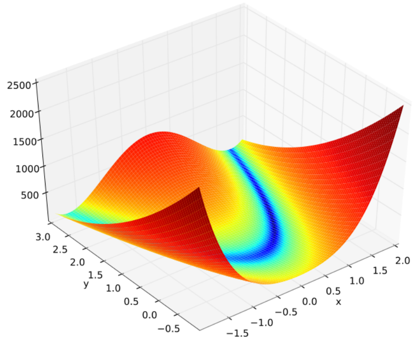
Rosenbrock function

$f(x, y) = (1-x)^2 + 100(y-x^2)^2$

There is a global minimum at $[x, y] = [1, 1]$ with f=0.

$$\begin{aligned} f(\mathbf{x}) &= f(x_1, x_2, \dots, x_N) \\ &= \sum_{i=1}^{N/2} [100(x_{2i-1}^2 - x_{2i})^2 \\ &\qquad + (x_{2i-1} - 1)^2]. \end{aligned}$$

For $N = 3$ there is a single minimum at $[1, 1, 1]$, for $4 \leq N \leq 7$ there are two, for $N > 7$ there is no analytic solution.



(Source: (Image) Wikipedia)

Notes

Computing the derivative of n-variate Rosenbrock

$$f(\mathbf{x}) = \sum_{i=1}^{N/2} [100(x_{2i-1}^2 - x_{2i})^2 + (x_{2i-1} - 1)^2].$$

- Option 1: derive the derivatives by hand and program,
- Option 2: Finite Differences
- Option 3: Algorithmic Differentiation

Notes

Analytic derivative of n-variate Rosenbrock

$$f(\mathbf{x}) = \sum_{i=1}^{N/2}[100(x_{2i-1}^2 - x_{2i})^2 + (x_{2i-1} - 1)^2]$$

$$= \sum_{i=1}^{N/2}[100(x_{2i-1}^4 - 2x_{2i-1}^2 x_{2i} + x_{2i}^2) + x_{2i-1}^2 - 2x_{2i-1} + 1]$$

$$\frac{\partial f(x)}{\partial x_{2i-1}} = 100(4x_{2i-1}^3 - 4x_{2i-1}x_{2i}) + 2x_{2i-1} - 2$$

$$\frac{\partial f(x)}{\partial x_{2i}} = 100(-2x_{2i-1})^2 + 2x_{2i})$$

- needs knowledge of the exact equations of the model
- can be very complex to compute
- needs manual programming
- difficult to verify

Notes

Finite difference derivative

Approximate the derivative as a forward difference

$$\frac{\partial f(x)}{\partial x_k} = \frac{f(x + \varepsilon\delta_k) - f(x)}{\varepsilon} + O(\varepsilon)$$

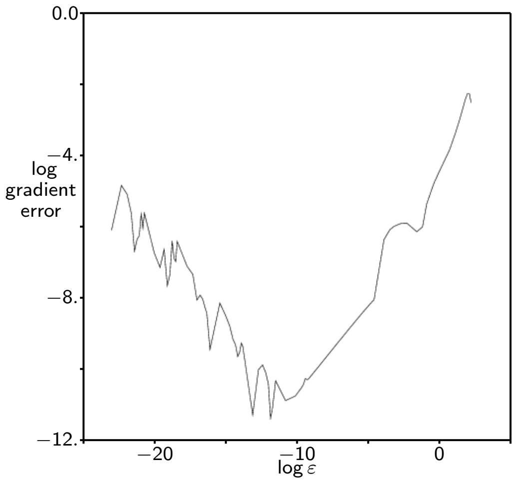
with ε a small perturbation size and δ_k a vector of the same length as x with zeros every where, but one in position k.
Similarly with a central difference

$$\frac{\partial f(x)}{\partial x_k} = \frac{f(x + \varepsilon\delta_k) - f(x - \varepsilon\delta_k)}{2\varepsilon} + O(\varepsilon^2)$$

Can we let $\varepsilon \to 0$ to make the truncation error vanish?

Notes

Errors of finite differences



Forward difference error dependence on ε (CFD case)

Notes

Finite differences for gradient computation

- Needs no knowledge of the equations or implementation, can call $f(x)$ as *black-box*.
- Needs careful setting of the stepsize ε:
- If ε is too large, there is a large truncation error.
- T.E. is $\propto O(\varepsilon)$ for forward or backward differences, one additional evaluation per design variable.
- T.E. is $\propto O(\varepsilon^2)$ for the central difference, but costs two additional evaluations per design variable.
- If ε is too small, there is a large round-off error.

Notes

Algorithmic Differentiation (AD)

- Also known as *Automatic Differentiation*.
- A computer program that computes a function $f(x)$ can be viewed as a sequence of simple operations such as addition, multiplication, etc:

$$f(x) = f_n(f_{n-1}(\cdots f_2(f_1(x))))$$

- We can straightforwardly compute the derivative of each of these operations and concatenate the derivatives using the chain rule.

$$\frac{\partial f(x)}{\partial x_i} = \frac{\partial f_n}{\partial f_{n-1}} \cdot \frac{\partial f_{n-1}}{\partial f_{n-2}} \cdot \ \cdots \ \cdot \frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_1(x)}{\partial x_i}$$

- While f_1 can only be a function of the input variables x, f_n will typically also depend on intermediate results $f_{n-1}, f_{n-2}, \ldots$.
- We can proceed to compute the derivative (automatically) instruction by instruction.

Notes

Outline

Notes

Simple example of AD

Using the chain rule, compute $\frac{\partial f}{\partial x_1}$ for

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \pi \cdot \cos(3x_1 + 2x_2 + x_3) \cdot \pi \cdot \sin(3x_1 + 2x_2 + x_3) \\ \pi \cdot \sin(3x_1 + 2x_2 + x_3) \cdot x_1 \end{bmatrix}$$

```
u = 3*x(1)+2*x(2)+x(3)          gx(1) = 1
pi = 3.14                       gx(2) = gx(3) = 0
v = pi*cos(u)                   gu = 3*gx(1)+2*gx(2)+gx(3)
w = pi*sin(u)                   gv = -pi*sin(u)*gu
sum = v + u                     gw = pi*cos(u)*gu
y(1) = v * w                    gy(1) = gv*w + v*gw
y(2) = w*x(1)                   gy(2) = gw*x(1) + gx(1)*w
```

The initial values in the chain rule need to be *seeded*, either set at the beginning of the computation, or computed in a preceding function call.

Notes

Outline

Examples

Introduction to Algorithmic Differentiation

Graph view of AD

Matrix-view of forward-mode AD

Reverse-mode AD

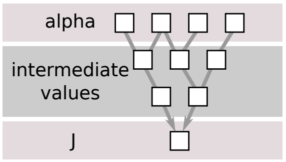
Application of AD

Automatic Differentiation tools

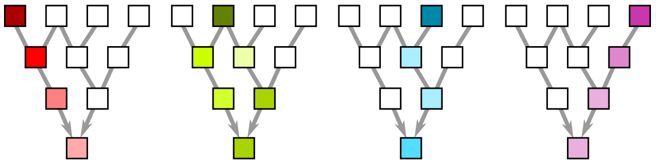
Notes

Algorithms as graphs

Original program



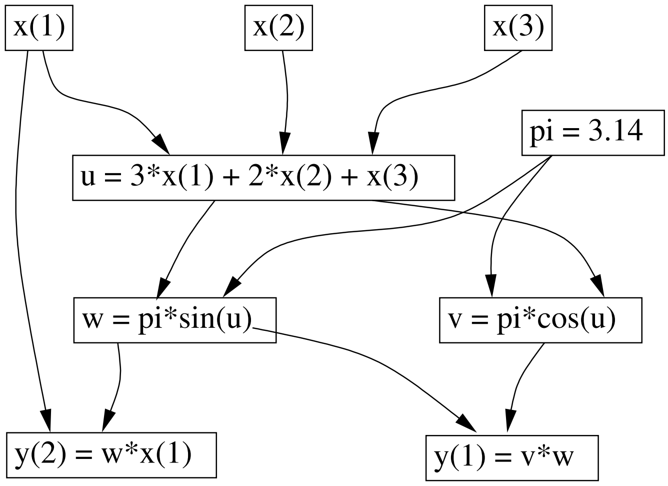
Forward differentiation



- Forward: propagate influence of each alpha through program

Notes

Graph view of the example algorithm



x(1)

x(2)

x(3)

pi = 3.14

u = 3*x(1) + 2*x(2) + x(3)

w = pi*sin(u)

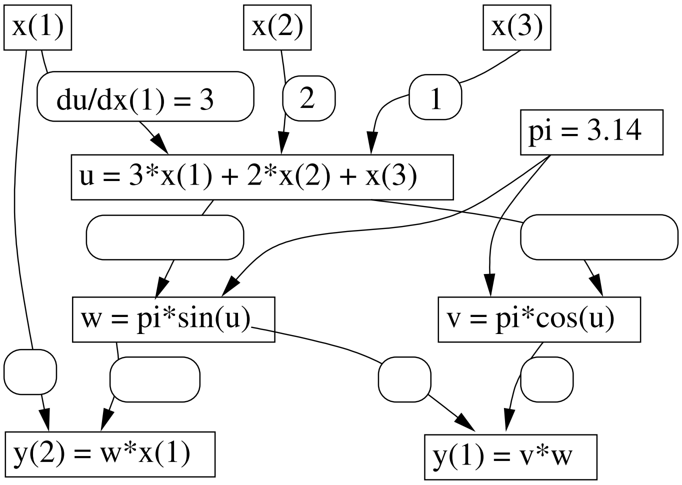
v = pi*cos(u)

y(2) = w*x(1)

y(1) = v*w

Notes

Add partial derivatives along each path



x(1)

x(2)

x(3)

du/dx(1) = 3

2

1

pi = 3.14

u = 3*x(1) + 2*x(2) + x(3)

w = pi*sin(u)

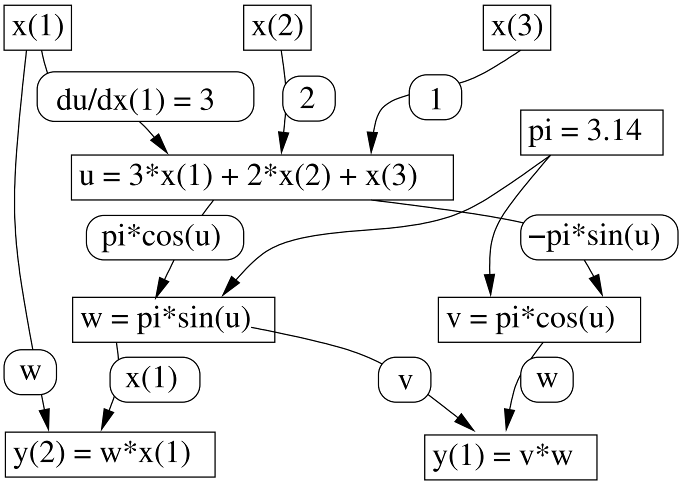
v = pi*cos(u)

y(2) = w*x(1)

y(1) = v*w

Notes

Add partial derivatives along each path



x(1)

x(2)

x(3)

du/dx(1) = 3

2

1

pi = 3.14

u = 3*x(1) + 2*x(2) + x(3)

pi*cos(u)

−pi*sin(u)

w = pi*sin(u)

v = pi*cos(u)

w

x(1)

v

w

y(2) = w*x(1)

y(1) = v*w

Notes

Forward-mode AD using the graph

- Looking at the graph leading to the computation of `y(1)` we have two incoming paths for `v` and `w`.
- The partial derivatives along the paths are $\frac{\partial y(1)}{\partial w} = v$, $\frac{\partial y(1)}{\partial v} = w$.
- The linearised change in $y(1)$ is then

$$\Delta y(1) = \frac{\partial y(1)}{\partial v}\Delta v + \frac{\partial y(1)}{\partial w}\Delta w = w\Delta v + v\Delta w$$

- The corresponding code statement is
```
gy(1) = w*gv + v*gw
```

Notes

Outline

Examples

Introduction to Algorithmic Differentiation

Graph view of AD

Matrix-view of forward-mode AD

Reverse-mode AD

Application of AD

Automatic Differentiation tools

Notes

Matrix view of the simple example I

- The example function is 3-variate, but there are 3 further intermediate and 2 *dependent* i.e. output variables, hence each program statement can be seen as multiplying a 8x8 matrix with an 8x1 column vector.

Step 1: `gx(1) = ! ext. assignment of gx(1)`

$$\begin{bmatrix} gx_1 \\ gx_2 \\ gx_3 \\ gu \\ gv \\ gw \\ gy_1 \\ gy_2 \end{bmatrix}_1 = \begin{bmatrix} 1 & & & & & & & \\ 0 & 0 & & & & & & \\ 0 & 0 & 0 & & & & & \\ 0 & 0 & 0 & 0 & & & & \\ 0 & 0 & 0 & 0 & 0 & & & \\ 0 & 0 & 0 & 0 & 0 & 0 & & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} gx_1 \\ gx_2 \\ gx_3 \\ gu \\ gv \\ gw \\ gy_1 \\ gy_2 \end{bmatrix}_0$$

Notes

Matrix view of the simple example I

- The example function is 3-variate, but there are 3 further intermediate and 2 *dependent* i.e. output variables, hence each program statement can be seen as multiplying a 8x8 matrix with an 8x1 column vector.

Step 2: `gx(2) = !  ext.  assignment of gx(2)`

$$\begin{bmatrix} gx_1 \\ gx_2 \\ gx_3 \\ gu \\ gv \\ gw \\ gy_1 \\ gy_2 \end{bmatrix}_2 = \begin{bmatrix} 1 & & & & & & & \\ 0 & 1 & & & & & & \\ 0 & 0 & 0 & & & & & \\ 0 & 0 & 0 & 0 & & & & \\ 0 & 0 & 0 & 0 & 0 & & & \\ 0 & 0 & 0 & 0 & 0 & 0 & & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} gx_1 \\ gx_2 \\ gx_3 \\ gu \\ gv \\ gw \\ gy_1 \\ gy_2 \end{bmatrix}_1$$

Notes

Matrix view of the simple example I

- The example function is 3-variate, but there are 3 further intermediate and 2 *dependent* i.e. output variables, hence each program statement can be seen as multiplying a 8x8 matrix with an 8x1 column vector.

Step 3: `gx(3) = !  ext.  assignment of gx(3)`

$$\begin{bmatrix} gx_1 \\ gx_2 \\ gx_3 \\ gu \\ gv \\ gw \\ gy_1 \\ gy_2 \end{bmatrix}_3 = \begin{bmatrix} 1 & & & & & & & \\ 0 & 1 & & & & & & \\ 0 & 0 & 1 & & & & & \\ 0 & 0 & 0 & 0 & & & & \\ 0 & 0 & 0 & 0 & 0 & & & \\ 0 & 0 & 0 & 0 & 0 & 0 & & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} gx_1 \\ gx_2 \\ gx_3 \\ gu \\ gv \\ gw \\ gy_1 \\ gy_2 \end{bmatrix}_2$$

Notes

Matrix view of the simple example I

- The example function is 3-variate, but there are 3 further intermediate and 2 *dependent* i.e. output variables, hence each program statement can be seen as multiplying a 8x8 matrix with an 8x1 column vector.

Step 4: `gu = 3*gx(1)+2*gx(2)+gx(3)`

$$\begin{bmatrix} gx_1 \\ gx_2 \\ gx_3 \\ gu \\ gv \\ gw \\ gy_1 \\ gy_2 \end{bmatrix}_4 = \begin{bmatrix} 1 & & & & & & & \\ 0 & 1 & & & & & & \\ 0 & 0 & 1 & & & & & \\ 3 & 2 & 1 & 0 & & & & \\ 0 & 0 & 0 & 0 & 0 & & & \\ 0 & 0 & 0 & 0 & 0 & 0 & & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} gx_1 \\ gx_2 \\ gx_3 \\ gu \\ gv \\ gw \\ gy_1 \\ gy_2 \end{bmatrix}_3$$

Notes

Matrix view of the simple example I

- The example function is 3-variate, but there are 3 further intermediate and 2 *dependent* i.e. output variables, hence each program statement can be seen as multiplying a 8x8 matrix with an 8x1 column vector.

Step 5: `gv = -gu*pi*sin(u)`

$$\begin{bmatrix} gx_1 \\ gx_2 \\ gx_3 \\ gu \\ gv \\ gw \\ gy_1 \\ gy_2 \end{bmatrix}_5 = \begin{bmatrix} 1 &&&&&&& \\ 0 & 1 &&&&&& \\ 0 & 0 & 1 &&&&& \\ 0 & 0 & 0 & 1 &&&& \\ 0 & 0 & 0 & -\pi\sin(u) &&&& \\ 0 & 0 & 0 & 0 & 0 & 0 && \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} gx_1 \\ gx_2 \\ gx_3 \\ gu \\ gv \\ gw \\ gy_1 \\ gy_2 \end{bmatrix}_4$$

Notes

Matrix view of the simple example I

- The example function is 3-variate, but there are 3 further intermediate and 2 *dependent* i.e. output variables, hence each program statement can be seen as multiplying a 8x8 matrix with an 8x1 column vector.

Step 6: `gw = gu*pi*cos(u)`

$$\begin{bmatrix} gx_1 \\ gx_2 \\ gx_3 \\ gu \\ gv \\ gw \\ gy_1 \\ gy_2 \end{bmatrix}_6 = \begin{bmatrix} 1 &&&&&&& \\ 0 & 1 &&&&&& \\ 0 & 0 & 1 &&&&& \\ 0 & 0 & 0 & 1 &&&& \\ 0 & 0 & 0 & 0 & 1 &&& \\ 0 & 0 & 0 & \pi\cos(u) & 0 & 0 && \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} gx_1 \\ gx_2 \\ gx_3 \\ gu \\ gv \\ gw \\ gy_1 \\ gy_2 \end{bmatrix}_5$$

Notes

Matrix view of the simple example I

- The example function is 3-variate, but there are 3 further intermediate and 2 *dependent* i.e. output variables, hence each program statement can be seen as multiplying a 8x8 matrix with an 8x1 column vector.

Step 7: `gy(1) = gv*w + v*gw`

$$\begin{bmatrix} gx_1 \\ gx_2 \\ gx_3 \\ gu \\ gv \\ gw \\ gy_1 \\ gy_2 \end{bmatrix}_7 = \begin{bmatrix} 1 &&&&&&& \\ 0 & 1 &&&&&& \\ 0 & 0 & 1 &&&&& \\ 0 & 0 & 0 & 1 &&&& \\ 0 & 0 & 0 & 0 & 1 &&& \\ 0 & 0 & 0 & 0 & 0 & 1 && \\ 0 & 0 & 0 & 0 & w & v & 0 & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} gx_1 \\ gx_2 \\ gx_3 \\ gu \\ gv \\ gw \\ gy_1 \\ gy_2 \end{bmatrix}_6$$

Notes

Matrix view of the simple example I

- The example function is 3-variate, but there are 3 further intermediate and 2 *dependent* i.e. output variables, hence each program statement can be seen as multiplying a 8x8 matrix with an 8x1 column vector.

Step 8: `gy(2) = gw*x(1) + w*gx(1)`

$$\begin{bmatrix} gx_1 \\ gx_2 \\ gx_3 \\ gu \\ gv \\ gw \\ gy_1 \\ gy_2 \end{bmatrix}_7 = \begin{bmatrix} 1 & & & & & & & \\ 0 & 1 & & & & & & \\ 0 & 0 & 1 & & & & & \\ 0 & 0 & 0 & 1 & & & & \\ 0 & 0 & 0 & 0 & 1 & & & \\ 0 & 0 & 0 & 0 & 0 & 1 & & \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & \\ w & 0 & 0 & 0 & 0 & x(1) & 0 & 0 \end{bmatrix} \begin{bmatrix} gx_1 \\ gx_2 \\ gx_3 \\ gu \\ gv \\ gw \\ gy_1 \\ gy_2 \end{bmatrix}_6$$

Notes

What is forward-mode AD computing?

- Forward-mode AD computes the Jacobian-vector product $z_n \quad = \quad E_n E_{n-1} \cdots E_2 E_1 z_1 \quad = \quad E\, z_1 \quad = \quad J\, z_1$
- Hiding the internal intermediate variables, we are left with

$$\nabla f \cdot \dot{x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \cdots & & & \\ \frac{\partial f_M}{\partial x_1} & \frac{\partial f_M}{\partial x_2} & \cdots & \frac{\partial f_M}{\partial x_n} \end{bmatrix} \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \vdots \\ \dot{x}_n \end{bmatrix} = \dot{y}$$

- AD computes a *directional derivative*.
- For n inputs to f (at program start), we need to invoke the differentiated chain n times, once for each column of the Jacobian with a different seed vector $\dot{x}$.
- We compute the derivatives of all output variables in one Jacobian column at each invocation of `f_d`.

Notes

Forward AD in our example:

$$\nabla f \cdot \dot{x} = \dot{y}$$

$$\begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \frac{\partial y_1}{\partial x_3} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_2}{\partial x_3} \end{bmatrix} \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{bmatrix} = \begin{bmatrix} \dot{y}_1 \\ \dot{y}_2 \end{bmatrix}$$

Using $\dot{x} = [1, 0, 0]^T$, we find

$$\begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \frac{\partial y_1}{\partial x_3} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_2}{\partial x_3} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} \\ \frac{\partial y_2}{\partial x_1} \end{bmatrix}$$

Seeding the inputs $\dot{x}_i$ one at a time, we extract one column of the Jacobian at a time.

Notes

Summary of forward-mode AD

- The forward mode computes directional derivatives by multiplying the Jacobian $\frac{\partial y}{\partial x}$ with a direction (or weighting) vector $\dot{x}$:
$$\dot{y} = \frac{\partial y}{\partial x}\dot{x}.$$
- Forward mode follows the statements in the same order as in the original *primal* function.
- For n *independent* (input) var., `f_d` needs to be invoked n times to compute one row for each input in the Jacobian.
- All rows of one columns of the Jacobian (different output variables) are obtained with one invocation of `f_d`.
- Typically in engineering applications we have many more input variables (design variables) than output variables (cost functions).
  Hence the forward mode is expensive, as it scales linearly with the number of design variables and is constant in the number of cost functions.

Notes

Forward mode with vector output function

- Viewed from the outside we compute the Jacobian-vector product $z_n = E_n E_{n-1} \cdots E_2 E_1 z_1 = E z_1 = Jp$

$$\nabla f \cdot p = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_1} & \dots & \frac{\partial f_2}{\partial x_n} \\ \vdots & & & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{bmatrix}$$

- If there are m components of the output function f, we obtain all rows in one column at the same time, but still need to invoke the differentiated routine n times with n different seed vectors p.

Notes

Outline

Examples

Introduction to Algorithmic Differentiation

Graph view of AD

Matrix-view of forward-mode AD

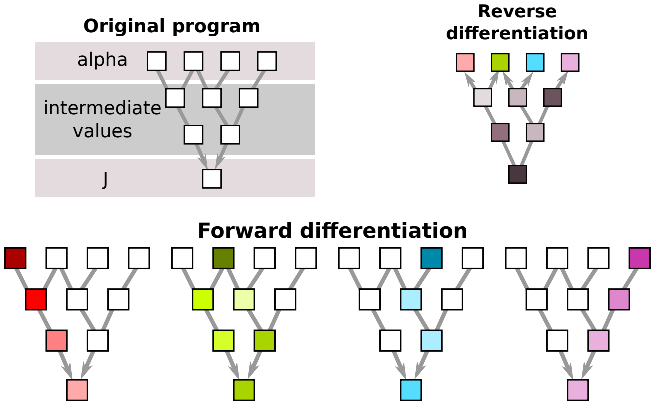
Reverse-mode AD

Application of AD

Automatic Differentiation tools

Notes

Algorithms as graphs



Original program

alpha

intermediate values

J

Reverse differentiation

Forward differentiation

- Forward: propagate influence of each alpha through program
- Reverse: trace back every influence on result. One pass is enough to get all derivatives.

Notes

Reverse-mode algorithmic differentiation

Forward-mode computes

$$\dot{y} = \frac{\partial y}{\partial x}\dot{x}.$$

What if we computed

$$\bar{y}\frac{\partial y}{\partial x} = \bar{x}.$$

Note that $\bar{y}$ has to be a row vector with dimension 2 to be multiplied with the 2×3 matrix of our example.
This is the *reverse-mode* of AD.
Again, a directional derivative is computed, but this time a *vector-matrix* product, or a transpose matrix-vector product.

Notes

Matrix-view of reverse mode AD

$$\bar{y}\frac{\partial y}{\partial x} = \bar{x}.$$

$$\bar{y}\nabla f = [\bar{y}_1, \bar{y}_2, \cdots, \bar{y}_n] \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_1} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \cdots & & & \\ \frac{\partial f_M}{\partial x_1} & \frac{\partial f_M}{\partial x_1} & \cdots & \frac{\partial f_M}{\partial x_n} \end{bmatrix} = [\bar{x}_1, \bar{x}_2 \cdots \bar{x}_n]$$

Notes

Reverse-mode AD in our example:

$$[\bar{y}_1, \bar{y}_2] \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_n} \end{bmatrix} = [\bar{x}_1, \bar{x}_2, \bar{x}_3]$$

Using $\bar{y} = [1, 0]$, we find

$$[1, 0] \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_3} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_3} \end{bmatrix} = \left[\frac{\partial y_1}{\partial x_1}, \frac{\partial y_1}{\partial x_1}, \frac{\partial y_1}{\partial x_3}\right]$$

Seeding the outputs $\bar{y}_i$ one at a time, we extract one row of the Jacobian at a time.

Notes

Properties of reverse-mode AD

- As in the forward-mode, reverse-mode AD computes a directional derivative.
- All complexity arguments transpose:
- Each invocation of $\bar{f}$ provides one row the Jacobian: sensitivity of one output variable w.r.t. all n input variables.
- For m outputs $y_1 \cdots y_m$, $\bar{f}$ needs to be invoked m times.
- Typically in engineering applications we have many more input variables (design variables) than output variables (cost functions).
  Hence the reverse mode is cheap, as its cost is linear in the number of cost functions, but is independent of the number of design variables

Notes

How to apply reverse-mode AD?

Forward-mode computes

$$\dot{y} = \frac{\partial y}{\partial x}\dot{x} = E_n E_{n-1} \cdots E_2 E_1 \dot{x} = E\dot{x}.$$

Applying simple rules of transpose matrix multiplication:

$$\begin{aligned} \left(\bar{y}\,\frac{\partial y}{\partial x}\right)^T &= \frac{\partial y}{\partial x}^T \bar{y}^T = E^T \bar{y}^T = (E_n E_{n-1} \cdots E_2 E_1)^T \bar{y}^T \\ &= E_1^T E_2^T \cdots E_{n-1}^T E_n^T \bar{y}^T \end{aligned}$$

- We apply the same differentiation operations E_i as in the forward mode
- But we accumulate the chain rule in reverse, starting with the final operation E_n.
- We follow the logic of the primal in *reverse* hence the name *reverse-differentiation*.

Notes

“Transposing” a statement in reverse-mode

Primal statement: `y(1) = v*w`

forward-mode

$$gy(1) = gv*w + v*gw$$

$$\begin{bmatrix} gv \\ gw \\ gy_1 \end{bmatrix}_7 = \begin{bmatrix} 1 & & \\ 0 & 1 & \\ w & v & 0 \end{bmatrix} \begin{bmatrix} gv \\ gw \\ gy_1 \end{bmatrix}_6$$

$$\dot{z}_{n+1} = E_n \dot{z}_n$$

reverse-mode

$$vb = vb + w*yb(1)$$
$$wb = wb + v*yb(1)$$

$$\begin{bmatrix} vb \\ wb \\ yb_1 \end{bmatrix}_6 = \begin{bmatrix} 1 & 0 & w \\ 0 & 1 & v \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} vb \\ wb \\ yb_1 \end{bmatrix}_7$$

$$\begin{aligned} \bar{z}_n E_n &= \bar{z}_{n-1} \\ (\bar{z}_n E_n)^T &= E_n^T \bar{z}_n^T = \bar{z}_{n-1}^T \end{aligned}$$

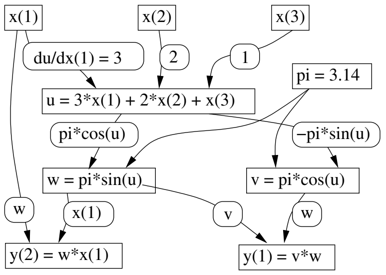
Notes

Reverse-mode AD using the graph

- We accumulate the change going through the graph in backward order
- The path between `v` and `y(1)` carries the partial derivative `w`.
- Hence along that path we accumulate
  `vb = vb + w*y(1)`
- Similarly for `w`:
  `wb = wb + v*y(1)`
- But `w` also contributes to `y(2)` with derivative `x(1)`, hence
  `wb = wb + x(1)*y(2)`
- In our primal code `y(2)` is computed last, hence the increment of `x(1)*y(2)` is the first, so we could omit initialising `wb=0` and write `wb = x(1)*y(2)`

Notes

Reverse mode AD, graph and code



```
u = 3*x(1) + 2*x(2) + x(3)
pi = 3.14
v = pi*cos(u)
w = pi*sin(u)
xb(:) = 0.
yb(:) = 0., yb(1) = 1
wb = x(1)*yb(2)
xb(1) = xb(1) + w*yb(2)
vb = w*yb(1)
wb = wb + v*yb(1)
ub = pi*cos(u)*wb-
     pi*sin(u)*vb
xb(1) = xb(1) + 3*ub
xb(2) = xb(2) + 2*ub
xb(3) = xb(3) + ub
```

Notes

Example of reverse mode AD

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \pi \cdot \cos(3x_1 + 2x_2 + x_3) \cdot \pi \cdot \sin(3x_1 + 2x_2 + x_3) \\ \pi \cdot \sin(3x_1 + 2x_2 + x_3) \cdot x_1 \end{bmatrix}$$

```
gx(1) = 1
gx(2) = gx(3) = 0
gu = 3*gx(1)+2*gx(2)+gx(3)
u = 3*x(1)+2*x(2)+x(3)
pi = 3.14
gv = -gu*pi*sin(u)
v = pi*cos(u)
gw = gu*pi*cos(u)
w = pi*sin(u)
gy(1) = gv*w + v*gw
y(1) = v * w
gy(2) = gw*x(1) + gx(1)*w
y(2) = w*x(1)
```

```
yb(1) = 1., yb(2) = 0.
u = 3*x(1) + 2*x(2) + x(3)
pi = 3.14
v = pi*cos(u)
w = pi*sin(u)
xb(:)  = 0.
wb = x(1)*yb(2)
xb(1) = xb(1) + w*yb(2)
vb = w*yb(1)
wb = wb + v*yb(1)
ub = pi*cos(u)*wb]-
     pi*sin(u)*vb
xb(1) = xb(1) + 3*ub
xb(2) = xb(2) + 2*ub
xb(3) = xb(3) + ub
```

Notes

Implementation the reverse-mode AD

- For each cost-function we need to seed with $\bar{y}_i = 1$.
- We obtain all the derivatives of y_i w.r.t. all x in one invocation.
- The logic is followed in reverse, hence we need to store or recompute all the intermediate values needed to compute the derivatives.

Notes

Outline

Notes

Forward-mode AD at function level

So far we have seen AD applied at the level of the code statements, we can also 'zoom out' and consider AD at the level of functions.

```
!  a,e variable, c constant
!  inputs.  Scalar J.
[a,f] =
 pre_proc ( a, c, e )
[a,h] =
 solve ( a, f )
[J] =
 obj ( a, c, h, e )
```

```
!  seed
ga(:)=0,ge(:)=0,ga(1)=1
[a,ga,f,gf] =
 gpre_proc(a,ga,c,e,ge)
[a,ga,h,gh] =
 gsolve(a,ga,f,gf)
[J,gJ] =
 gobj(a,ga,c,h,gh,e,ge)
```

Notes

Reverse mode AD at function level

In reverse mode the program traverses the graph from end to start: inputs and outputs reverse roles for the perturbations 'b'.

```
!  a,e variable, c constant
!  inputs.  Scalar J.
[a,f] =
 pre_proc ( a, c, e )
[a,h] =
 solve ( a, f )
[J] =
 obj ( a, c, h, e )
```

```
!seed
ab=0,hb=0,eb=0,Jb=1
!  recompute e,f
[a,f] =
 pre_proc ( a, c, e )
[a,h] =
 solve ( a, f )

[J,ab,hb,eb] =
 objb(a,ab,c,h,hb,e,eb,Jb)
[a,ab,h,fb] =
 solveb(a,ab,f,hb)
[a,ab,f,eb] =
 pre_procb(a,ab,c,e,eb,fb)
```

Notes

Linear operators

```
!  a,e,f variable inputs,
function [r] =
 linFun ( a,e,f )
  r = 3*a + 2*e + f
end function
```

```
 ga,ge,gf = ....  !  seed ,
function [r,gr] =
 glinFun ( a,ga,e,ge,f,gf )
  r = 3*a + 2*e + f
  gr = 3*ga + 2*ge + gf
end function
```

Alternatively, as the function is linear and `gr` does not depend on the values of `a,e,f`, call the original, *primal*, `linFun` twice:

```
ga,ge,gf = ....  !  seed ,
[r] =  linFun(a,e,f)
[gr] =  linFun(ga,ge,gf)
```

Notes

Symmetric operators

Recall that AD can be viewed as a matrix multiplication, the reverse mode uses the transpose. If the matrix is symmetric, *self-adjoint*, this produces the same operation:

```
!  in array a(:), out r(:)
function [r] =
  symFun ( a )
   for i=1:size(a)
     r(i) =
a(i-1)+a(i)+a(i+1)
  end for
end function
```

```
function [r,ab] =
  symFunb ( a,rb )
  for i=1:size(a)
    r(i) = a(i-1)+a(i)+a(i+1)
  end for
  ab(:)  = 0 !  accum. from 0
  for i=size(a),1,-1
    ab(i-1) += rb(i)
    ab(i) +=rb(i)
    ab(i+1) += rb(i)
  end for
end function
```

Notes

Symmetric operators II

Alternatively, as the function is self-adjoint, call the original (*primal*) `symFun` twice:

```
!  in array a(:), out r(:)
function [r] =
  symFun ( a )
   for i=1:size(a)
     r = a(i-1)+a(i)+a(i+1)
  end for
end function
```

```
[r] =   symFun(a)
[ab] =   symFun(rb)
```

In this example `symFun` is linear and self-adjoint, hence we can reuse the *primal* code. If non-linear and self-adjoint, we can reuse the simpler forward-AD code:

`[r,gr] = gsymfun (a,ga)`

but call it as

`[r,ab] = gsymfun (a,rb)`

which reverses the gradient arguments and computes in reverse mode.

Notes

Outline

Notes

From Algorithmic to Automatic Differentiation

- Forward-mode steps through the statements in the same order, add a derivative computation statement before each primal statement.
- This is a straightforward (i.e. rigorous and stupid) process, why not have this done by software.
- The reverse-mode records all partial derivatives in each statement, then accumulates the derivatives in reverse.
- This is a straightforward (i.e. rigorous and stupid), potentially memory consuming process, why not have this done by software.

There are two main options to apply automatic differentiation:

- Source-transformation
- Operator-overloading

Notes

AD via source transformation

Procedure:

- *Parse* (i.e. interpret) the statements in the primal source code
- then add the necessary statements to produce modified source code
- then compile the modified source code.

Source-transformation AD tools:

- Tapenade (INRIA): Fortran, C. Forward and reverse, most popular tool.
- TAF, TAC (FastOpt, commercial): Fortran, C. Forward and reverse, produces highly performing code.
- TAMC (FastOpt, free to use): Fortran
- AdiFor (Argonne, free to use): Fortran, forward-mode only

Notes

Properties of source transformation AD

Advantages/Disadvantages:

- Modified source code can be analysed, to inform a rewrite of the primal to improve performance
- Modified source code can be optimised by the compiler,
- differentiated source code modules can easily be assembled with non- or hand-differentiated code to optimise memory and runtime.
- Compile-time parsing can only take account of information available at compile-time (i.e. information embedded in the code structure), it is oblivious of run-time effect such as values of pointers.
- The entire code needs differentiating, regardless whether or not parts of the code will be used at run-time.

Notes

AD via Operator-Overloading

Principle:

- Most modern languages allow *operator-overloading*, i.e. to define special data-types and then define extensions of standard operations such as `*` or `+` for these data-types.
- E.g. we could define a forward derivative-enhanced double in C:

```
struct {
  double val ;
  double val_d ;
} double_d
```

- An overloaded multiplication in C++ then would be:

```
double_d operator *( double_d a, double_d b ) {
  double_d prod ;
  prod.val_d = a.val*b.val_d + a.val_d*b.val ;
  prod.val = a.val * b.val ;
return ( prod ) ; }
```

Notes

AD via Operator-Overloading

- Operator-overloading very naturally gives rise to a forward-mode differentiation.
- All operators need overloading, all simple data-types such as `double` promoted to enhanced ones `double_d`.
- For reverse mode we need create a *tape* of operations and operands which is then run backwards at the end.

Properties of operator-overloading AD

- High memory requirements due to large tapes.
- The tape is difficult to analyse or inspect, limited possibilites to assemble differentiated parts in other code.
- The tape contains run-time analysis, only required code branches are differentiated.
- All `val` are calculated, whether or not needed to form `val.d`. Static compile-time optimisation is not possible.
- S-T AD usually outperforms O-O AD.

Notes

Operator-overloading AD tools

The majority of AD tools for languages other than Fortran use operator-overloading (O-O):

- ADOL-C (Univ. Paderborn): C,C++. Open-source. The most widely used and most mature tool for C,C++.
- codipack for C++. Claims to have a more efficient tape implementation.
- fadBad, cppAD for C++
- tools also available for matlab, R

Main source of information on AD: http://www.autodiff.org

Notes

Organisation of the lectures

1. Univariate optimisation
    - Bisection, Steepest Descent, Newton's method
2. Multivariate optimisation
    - Steepest descent, Newton's method
    - and line-search methods: Wolfe and Armijo conditions,
    - Quasi-Newton methods,
3. Constrained Optimisation:
    - Projected gradient methods,
    - Penalty methods, exterior and interior point methods,
    - SQP
4. Adjoint methods
    - Reversing time, Automatic Differentiation
    - Adjoint CFD codes
5. Gradient computation
    - Manual derivation, Finite Differences
    - Algorithmic and automatic differentiation, fwd and bkwd.

Notes

Exercises for AD

1. Perform forward-mode AD to obtain the first derivative of the bi-variate Rosenbrock function coded in `multivar_opt.m`. Verify the gradients against finite-differences and analytic derivatives.
2. Draw the graph for Rosenbock, perform reverse-mode AD. Verify the gradients.
3. Use Tapenade's online interface to produce derivative code for Rosenbrock in fwd and rev modes. Verify the gradients.

Notes

Notes