Algorithmic Differentiation

C++ & Extremum Estimation

Matt P. Dziubinski

CppCon 2015

`matt@math.aau.dk` // `@matt_dz`
Department of Mathematical Sciences, Aalborg University
CREATES (Center for Research in Econometric Analysis of Time Series)

Outline

Slides

- Latest version: https://speakerdeck.com/mattpd

Why?

Motivation: Data & Models

- Have: data, y (numbers)

Motivation: Data & Models

- Have: data, y (numbers)
- Want: understand, explain, forecast

Motivation: Data & Models

- Have: data, y (numbers)
- Want: understand, explain, forecast
- Tool: model, $f(y; \theta)$

Motivation: Data & Models

- Have: data, y (numbers)
- Want: understand, explain, forecast
- Tool: model, $f(y; \theta)$
- Need: connection, θ

Motivation: Data & Models

- Have: data, y (numbers)
- Want: understand, explain, forecast
- Tool: model, $f(y; \theta)$
- Need: connection, θ
- Idea: fitting model to data

Motivation: Data & Models

- Have: data, y (numbers)
- Want: understand, explain, forecast
- Tool: model, $f(y; \theta)$
- Need: connection, θ
- Idea: fitting model to data
- How: cost function — minimize / maximize

- Have: data, y (numbers)
- Want: understand, explain, forecast
- Tool: model, $f(y;\theta)$
- Need: connection, θ
- Idea: fitting model to data
- How: cost function — minimize / maximize
- Nowadays one of the most widely applied approaches — estimation (econometrics, statistics), calibration (finance), training ((supervised) machine learning)

Motivation: Data & Models

- Have: data, y (numbers)
- Want: understand, explain, forecast
- Tool: model, $f(y;\theta)$
- Need: connection, θ
- Idea: fitting model to data
- How: cost function — minimize / maximize
- Nowadays one of the most widely applied approaches — estimation (econometrics, statistics), calibration (finance), training ((supervised) machine learning)
- In practice: Generally no are closed-form expressions — estimation relies on numerical optimization

Motivation: Parametric Models

- Parametric model: parameter vector $\theta \in \Theta \subseteq \mathbb{R}^p$,
  Θ — parameter space

Motivation: Parametric Models

- Parametric model: parameter vector $\theta \in \Theta \subseteq \mathbb{R}^p$,
  Θ — parameter space
- An estimator $\hat{\theta}$ is an *extremum estimator* if $\exists Q : \Theta \to \mathbb{R}$ s.t.

$$\hat{\theta} = \operatorname*{argmax}_{\theta \in \Theta} Q(\theta)$$

- Parametric model: parameter vector $\theta \in \Theta \subseteq \mathbb{R}^p$,
  Θ — parameter space
- An estimator $\hat{\theta}$ is an *extremum estimator* if $\exists Q : \Theta \to \mathbb{R}$ s.t.

$$\hat{\theta} = \operatorname*{argmax}_{\theta \in \Theta} Q(\theta)$$

  - can be obtained as an *extremum* (*maximum* or *minimum*) of a scalar objective function

Motivation: Parametric Models

- Parametric model: parameter vector $\theta \in \Theta \subseteq \mathbb{R}^p$,
  Θ — parameter space
- An estimator $\hat{\theta}$ is an *extremum estimator* if $\exists Q : \Theta \to \mathbb{R}$ s.t.

$$\hat{\theta} = \operatorname*{argmax}_{\theta \in \Theta} Q(\theta)$$

  - can be obtained as an *extremum* (*maximum* or *minimum*) of a scalar objective function
  - class includes OLS (Ordinary Least Squares), NLS (Nonlinear Least Squares), GMM (Generalized Method of Moments), and QMLE (Quasi Maximum Likelihood Estimation)

Numerical Optimization

- Numerical optimization algorithms — can be broadly categorized into two categories:

Numerical Optimization: Algorithms

- Numerical optimization algorithms — can be broadly categorized into two categories:
  - derivative-free — do not rely on the knowledge of the objective function's gradient in the optimization process

Numerical Optimization: Algorithms

- Numerical optimization algorithms — can be broadly categorized into two categories:
  - derivative-free — do not rely on the knowledge of the objective function's gradient in the optimization process
  - gradient-based — need the gradient of the objective function

Numerical Optimization: Algorithms

- Numerical optimization algorithms — can be broadly categorized into two categories:
  - derivative-free — do not rely on the knowledge of the objective function's gradient in the optimization process
  - gradient-based — need the gradient of the objective function
    - iteration form:
      initial iterate: θ_0
      new iterate: $\theta_k = \theta_{k-1} + s_k$, iteration $k \geq 1$
      step: $s_k = \alpha_k p_k$
      length: scalar $\alpha_k > 0$
      direction: vector p_k s.t. $H_k p_k = \nabla Q(\theta_k)$

Numerical Optimization: Algorithms

- Numerical optimization algorithms — can be broadly categorized into two categories:
  - derivative-free — do not rely on the knowledge of the objective function's gradient in the optimization process
  - gradient-based — need the gradient of the objective function
    - iteration form:
      initial iterate: θ_0
      new iterate: $\theta_k = \theta_{k-1} + s_k$, iteration $k \geq 1$
      step: $s_k = \alpha_k p_k$
      length: scalar $\alpha_k > 0$
      direction: vector p_k s.t. $H_k p_k = \nabla Q(\theta_k)$
    - H_k:
      $H_k = I$ (Steepest Ascent) — many variants, including SGD (Stochastic Gradient Descent)
      $H_k = \nabla^2 Q(\theta_k)$ (Newton)
      H_k = Hessian approximation satisfying the secant equation:
      $H_{k+1} s_k = \nabla Q_{k+1} - \nabla Q_k$ (Quasi-Newton)

Numerical Optimization: Gradient

- Gradient-based algorithms often exhibit superior convergence rates (superlinear or even quadratic)

- Gradient-based algorithms often exhibit superior convergence rates (superlinear or even quadratic)
- However: this property depends on the gradient either being exact — or at minimum sufficiently accurate in the limit sense, Nocedal and Wright (2006 Chapter 3)

Numerical Optimization: Gradient

- Gradient-based algorithms often exhibit superior convergence rates (superlinear or even quadratic)
- However: this property depends on the gradient either being exact — or at minimum sufficiently accurate in the limit sense, Nocedal and Wright (2006 Chapter 3)
- For many modern models analytical gradient formulas not available in closed-form — or non-trivial to derive

- Gradient-based algorithms often exhibit superior convergence rates (superlinear or even quadratic)
- However: this property depends on the gradient either being exact — or at minimum sufficiently accurate in the limit sense, Nocedal and Wright (2006 Chapter 3)
- For many modern models analytical gradient formulas not available in closed-form — or non-trivial to derive
- This often leads to the use of numerical approximations — in particular, finite difference methods (FDMs), Nocedal and Wright (2006 Chapter 8)

Numerical Optimization: Gradient

- Gradient-based algorithms often exhibit superior convergence rates (superlinear or even quadratic)
- However: this property depends on the gradient either being exact — or at minimum sufficiently accurate in the limit sense, Nocedal and Wright (2006 Chapter 3)
- For many modern models analytical gradient formulas not available in closed-form — or non-trivial to derive
- This often leads to the use of numerical approximations — in particular, finite difference methods (FDMs), Nocedal and Wright (2006 Chapter 8)
  - Inference: Covariance matrix estimators also involve the use of derivative information.

- Need a way obtain the gradient — preferably: fast **and** accurate

- Need a way obtain the gradient — preferably: fast **and** accurate
- This is where Algorithmic Differentiation (AD) comes in:

- Need a way obtain the gradient — preferably: fast **and** accurate
- This is where Algorithmic Differentiation (AD) comes in:
    - fast

- Need a way obtain the gradient — preferably: fast **and** accurate
- This is where Algorithmic Differentiation (AD) comes in:
  - fast
  - **and** accurate

- Need a way obtain the gradient — preferably: fast **and** accurate
- This is where Algorithmic Differentiation (AD) comes in:
  - fast
  - **and** accurate
  - **and** simple

- Need a way obtain the gradient — preferably: fast **and** accurate
- This is where Algorithmic Differentiation (AD) comes in:
  - fast
  - **and** accurate
  - **and** simple
- Essentially:

- Need a way obtain the gradient — preferably: fast **and** accurate
- This is where Algorithmic Differentiation (AD) comes in:
    - fast
    - **and** accurate
    - **and** simple
- Essentially:
    - an automatic computational differentiation method

- Need a way obtain the gradient — preferably: fast **and** accurate
- This is where Algorithmic Differentiation (AD) comes in:
  - fast
  - **and** accurate
  - **and** simple
- Essentially:
  - an automatic computational differentiation method
  - based on the systematic application of the chain rule

- Need a way obtain the gradient — preferably: fast **and** accurate
- This is where Algorithmic Differentiation (AD) comes in:
  - fast
  - **and** accurate
  - **and** simple
- Essentially:
  - an automatic computational differentiation method
  - based on the systematic application of the chain rule
  - exact to the maximum extent allowed by the machine precision

- Need a way obtain the gradient — preferably: fast **and** accurate
- This is where Algorithmic Differentiation (AD) comes in:
  - fast
  - **and** accurate
  - **and** simple
- Essentially:
  - an automatic computational differentiation method
  - based on the systematic application of the chain rule
  - exact to the maximum extent allowed by the machine precision
- How does it compare to other computational differentiation methods?

Calculating Derivatives

- Main approaches:

Calculating Derivatives

- Main approaches:
  - Finite Differencing

- Main approaches:
  - Finite Differencing
  - Algorithmic Differentiation (AD), a.k.a. Automatic Differentiation

- Main approaches:
  - Finite Differencing
  - Algorithmic Differentiation (AD), a.k.a. Automatic Differentiation
  - Symbolic Differentiation

Finite Differencing

- Recall $f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$

Finite Differencing

- Recall $f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$
  - forward-difference approximation:

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x + he_i) - f(x)}{h},$$

- Recall $f'(x) = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h}$
  - forward-difference approximation:

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x + he_i) - f(x)}{h},$$

  - central-difference approximation:

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x + he_i) - f(x - he_i)}{2h},$$

- Recall $f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$
  - forward-difference approximation:

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x + he_i) - f(x)}{h},$$

  - central-difference approximation:

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x + he_i) - f(x - he_i)}{2h},$$

- Computational cost:

Finite Differencing

- Recall $f'(x) = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h}$
  - forward-difference approximation:

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x+he_i)-f(x)}{h},$$

  - central-difference approximation:

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x+he_i)-f(x-he_i)}{2h},$$

- Computational cost:
  - forward-difference formula: $p+1$ function evaluations

- Recall $f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$
    - forward-difference approximation:

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x + he_i) - f(x)}{h},$$

    - central-difference approximation:

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x + he_i) - f(x - he_i)}{2h},$$

- Computational cost:
    - forward-difference formula: $p + 1$ function evaluations
    - central-difference formula: $2 * p$ function evaluations

Finite Differencing — Truncation Error

- Truncation Error Analysis — Taylor's theorem

- Truncation Error Analysis — Taylor's theorem
- Consider univariate function f and the forward-difference formula:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

Finite Differencing — Truncation Error

- Truncation Error Analysis — Taylor's theorem
- Consider univariate function f and the forward-difference formula:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

- Recall Taylor's formula

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \frac{1}{6}h^3 f'''(x) + O(h^4)$$

Finite Differencing — Truncation Error

- Truncation Error Analysis — Taylor's theorem
- Consider univariate function f and the forward-difference formula:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

- Recall Taylor's formula

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \frac{1}{6}h^3 f'''(x) + O(h^4)$$

- Hence

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{1}{2}hf''(x) - \frac{1}{6}h^2 f'''(x) - O(h^3)$$

Finite Differencing — Truncation Error

- Truncation Error Analysis — Taylor's theorem
- Consider univariate function f and the forward-difference formula:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

- Recall Taylor's formula

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \frac{1}{6}h^3 f'''(x) + O(h^4)$$

- Hence

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{1}{2}hf''(x) - \frac{1}{6}h^2 f'''(x) - O(h^3)$$

- Truncation error $c_1 h + c_2 h^2 + \ldots$

Finite Differencing — Truncation Error

- Truncation Error Analysis — Taylor's theorem
- Consider univariate function f and the forward-difference formula:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

- Recall Taylor's formula

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \frac{1}{6}h^3 f'''(x) + O(h^4)$$

- Hence

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{1}{2}hf''(x) - \frac{1}{6}h^2 f'''(x) - O(h^3)$$

- Truncation error $c_1 h + c_2 h^2 + \ldots$
- Decreases with smaller h

Floating Point Arithmetic

- Floating Point Arithmetic: NO associativity (or distributivity) of addition (and multiplication)

- Floating Point Arithmetic: NO associativity (or distributivity) of addition (and multiplication)
- Example:

Floating Point Arithmetic

- Floating Point Arithmetic: NO associativity (or distributivity) of addition (and multiplication)
- Example:
  - Input:

```
a = 1234.567; b = 45.678; c = 0.0004
a + b + c
c + b + a
(a + b + c) - (c + b + a)
```

Floating Point Arithmetic

- Floating Point Arithmetic: NO associativity (or distributivity) of addition (and multiplication)
- Example:
  - Input:
    ```
    a = 1234.567; b = 45.678; c = 0.0004
    a + b + c
    c + b + a
    (a + b + c) - (c + b + a)
    ```
  - Output:

Floating Point Arithmetic

- Floating Point Arithmetic: NO associativity (or distributivity) of addition (and multiplication)
- Example:
  - Input:

    ```
    a = 1234.567; b = 45.678; c = 0.0004
    a + b + c
    c + b + a
    (a + b + c) - (c + b + a)
    ```
  - Output:
    - `1280.2454`

Floating Point Arithmetic

- Floating Point Arithmetic: NO associativity (or distributivity) of addition (and multiplication)
- Example:
  - Input:

    ```
    a = 1234.567; b = 45.678; c = 0.0004
    a + b + c
    c + b + a
    (a + b + c) - (c + b + a)
    ```
  - Output:
    - `1280.2454`
    - `1280.2454`

Floating Point Arithmetic

- Floating Point Arithmetic: NO associativity (or distributivity) of addition (and multiplication)
- Example:
  - Input:
    ```
    a = 1234.567; b = 45.678; c = 0.0004
    a + b + c
    c + b + a
    (a + b + c) - (c + b + a)
    ```
  - Output:
    - `1280.2454`
    - `1280.2454`
    - `-2.273737e-13`

Floating Point Arithmetic

- Floating Point Arithmetic: NO associativity (or distributivity) of addition (and multiplication)
- Example:
  - Input:
    ```
    a = 1234.567; b = 45.678; c = 0.0004
    a + b + c
    c + b + a
    (a + b + c) - (c + b + a)
    ```
  - Output:
    - `1280.2454`
    - `1280.2454`
    - `-2.273737e-13`
- "Many a serious mathematician has attempted to give rigorous analyses of a sequence of floating point operations, but has found the task to be so formidable that he has tried to content himself with plausibility arguments instead." — Donald E. Knuth, TAOCP2

Floating Point Representation

- Floating point representation: inspired by scientific notation

Floating Point Representation

- Floating point representation: inspired by scientific notation
- The IEEE Standard for Floating-Point Arithmetic (IEEE 754)

Floating Point Representation

- Floating point representation: inspired by scientific notation
- The IEEE Standard for Floating-Point Arithmetic (IEEE 754)
- Form: significant digits $\times$ base$^{\text{exponent}}$

- Floating point representation: inspired by scientific notation
- The IEEE Standard for Floating-Point Arithmetic (IEEE 754)
- Form: significant digits $\times$ base$^{\text{exponent}}$
- Machine epsilon, $\epsilon_M := \inf\{\epsilon > 0 : 1.0 + \epsilon \neq 1.0\}$

- Floating point representation: inspired by scientific notation
- The IEEE Standard for Floating-Point Arithmetic (IEEE 754)
- Form: significant digits $\times$ base$^{\text{exponent}}$
- Machine epsilon, $\epsilon_M := \inf\{\epsilon > 0 : 1.0 + \epsilon \neq 1.0\}$
  - the difference between 1.0 and the next value representable by the floating-point number $1.0 + \epsilon$

- Floating point representation: inspired by scientific notation
- The IEEE Standard for Floating-Point Arithmetic (IEEE 754)
- Form: significant digits $\times$ base$^{\text{exponent}}$
- Machine epsilon, $\epsilon_M := \inf\{\epsilon > 0 : 1.0 + \epsilon \neq 1.0\}$
  - the difference between 1.0 and the next value representable by the floating-point number $1.0 + \epsilon$
  - IEEE 754 binary32 (single precision): $2^{-23} \approx$ `1.19209e-07` (1 sign bit, 8 exponent bits, leaving 23 bits for significand)

Floating Point Representation

- Floating point representation: inspired by scientific notation
- The IEEE Standard for Floating-Point Arithmetic (IEEE 754)
- Form: significant digits $\times$ base$^{\text{exponent}}$
- Machine epsilon, $\epsilon_M := \inf\{\epsilon > 0 : 1.0 + \epsilon \neq 1.0\}$
  - the difference between 1.0 and the next value representable by the floating-point number $1.0 + \epsilon$
  - IEEE 754 binary32 (single precision): $2^{-23} \approx$ `1.19209e-07` (1 sign bit, 8 exponent bits, leaving 23 bits for significand)
  - IEEE 754 binary64 (double precision): $2^{-52} \approx$ `2.22045e-16` (1 sign bit, 11 exponent bits, leaving 52 bits for significand)

Floating Point Representation

- Floating point representation: inspired by scientific notation
- The IEEE Standard for Floating-Point Arithmetic (IEEE 754)
- Form: significant digits $\times$ base$^{\text{exponent}}$
- Machine epsilon, $\epsilon_M := \inf\{\epsilon > 0 : 1.0 + \epsilon \neq 1.0\}$
  - the difference between 1.0 and the next value representable by the floating-point number $1.0 + \epsilon$
  - IEEE 754 binary32 (single precision): $2^{-23} \approx$ `1.19209e-07` (1 sign bit, 8 exponent bits, leaving 23 bits for significand)
  - IEEE 754 binary64 (double precision): $2^{-52} \approx$ `2.22045e-16` (1 sign bit, 11 exponent bits, leaving 52 bits for significand)
- floating point number representation: $fl(x) = x(1 + \delta)$, where $|\delta| < \epsilon_M$

Floating Point Representation Properties

- floating point number representation: $fl(x) = x(1+\delta)$, where $|\delta| < \epsilon_M$

Floating Point Representation Properties

- floating point number representation: $fl(x) = x(1+\delta)$, where $|\delta| < \epsilon_M$
- representation (round-off) error: $x - fl(x)$

Floating Point Representation Properties

- floating point number representation: $fl(x) = x(1 + \delta)$, where $|\delta| < \epsilon_M$
- representation (round-off) error: $x - fl(x)$
- absolute round-off error: $|x - fl(x)| < \epsilon_M |x|$

Floating Point Representation Properties

- floating point number representation: $fl(x) = x(1+\delta)$, where $|\delta| < \epsilon_M$
- representation (round-off) error: $x - fl(x)$
- absolute round-off error: $|x - fl(x)| < \epsilon_M |x|$
- relative round-off error: $|x - fl(x)|/|x| < \epsilon_M$

Floating Point Representation Properties

- floating point number representation: $fl(x) = x(1+\delta)$, where $|\delta| < \epsilon_M$
- representation (round-off) error: $x - fl(x)$
- absolute round-off error: $|x - fl(x)| < \epsilon_M |x|$
- relative round-off error: $|x - fl(x)|/|x| < \epsilon_M$
  - floating point number representation: $fl(x) = x(1+\delta)$, where $|\delta| < \epsilon_M$

Floating Point Representation Properties

- floating point number representation: $fl(x) = x(1+\delta)$, where $|\delta| < \epsilon_M$
- representation (round-off) error: $x - fl(x)$
- absolute round-off error: $|x - fl(x)| < \epsilon_M |x|$
- relative round-off error: $|x - fl(x)|/|x| < \epsilon_M$
  - floating point number representation: $fl(x) = x(1+\delta)$, where $|\delta| < \epsilon_M$
  - representation (round-off) error: $x - fl(x)$

Floating Point Representation Properties

- floating point number representation: $fl(x) = x(1+\delta)$, where $|\delta| < \epsilon_M$
- representation (round-off) error: $x - fl(x)$
- absolute round-off error: $|x - fl(x)| < \epsilon_M |x|$
- relative round-off error: $|x - fl(x)|/|x| < \epsilon_M$
    - floating point number representation: $fl(x) = x(1+\delta)$, where $|\delta| < \epsilon_M$
    - representation (round-off) error: $x - fl(x)$
    - absolute round-off error: $|x - fl(x)| < \epsilon_M |x|$

Floating Point Representation Properties

- floating point number representation: $fl(x) = x(1+\delta)$, where $|\delta| < \epsilon_M$
- representation (round-off) error: $x - fl(x)$
- absolute round-off error: $|x - fl(x)| < \epsilon_M |x|$
- relative round-off error: $|x - fl(x)|/|x| < \epsilon_M$
    - floating point number representation: $fl(x) = x(1+\delta)$, where $|\delta| < \epsilon_M$
    - representation (round-off) error: $x - fl(x)$
    - absolute round-off error: $|x - fl(x)| < \epsilon_M |x|$
    - relative round-off error: $|x - fl(x)|/|x| < \epsilon_M$

Floating Point Representation Properties

- floating point number representation: $fl(x) = x(1+\delta)$, where $|\delta| < \epsilon_M$
- representation (round-off) error: $x - fl(x)$
- absolute round-off error: $|x - fl(x)| < \epsilon_M |x|$
- relative round-off error: $|x - fl(x)|/|x| < \epsilon_M$

    - floating point number representation: $fl(x) = x(1+\delta)$, where $|\delta| < \epsilon_M$
    - representation (round-off) error: $x - fl(x)$
    - absolute round-off error: $|x - fl(x)| < \epsilon_M |x|$
    - relative round-off error: $|x - fl(x)|/|x| < \epsilon_M$

- ϵ_M a.k.a. unit round-off, $\mathbf{u} = ULP(1.0)$

Floating Point Representation Properties

- floating point number representation: $fl(x) = x(1 + \delta)$, where $|\delta| < \epsilon_M$
- representation (round-off) error: $x - fl(x)$
- absolute round-off error: $|x - fl(x)| < \epsilon_M |x|$
- relative round-off error: $|x - fl(x)|/|x| < \epsilon_M$
    - floating point number representation: $fl(x) = x(1 + \delta)$, where $|\delta| < \epsilon_M$
    - representation (round-off) error: $x - fl(x)$
    - absolute round-off error: $|x - fl(x)| < \epsilon_M |x|$
    - relative round-off error: $|x - fl(x)|/|x| < \epsilon_M$
- ϵ_M a.k.a. unit round-off, $\mathbf{u} = ULP(1.0)$
- ULP — Unit in the last place

Floating Point Representation Properties

- floating point number representation: $fl(x) = x(1+\delta)$, where $|\delta| < \epsilon_M$
- representation (round-off) error: $x - fl(x)$
- absolute round-off error: $|x - fl(x)| < \epsilon_M |x|$
- relative round-off error: $|x - fl(x)|/|x| < \epsilon_M$
    - floating point number representation: $fl(x) = x(1+\delta)$, where $|\delta| < \epsilon_M$
    - representation (round-off) error: $x - fl(x)$
    - absolute round-off error: $|x - fl(x)| < \epsilon_M |x|$
    - relative round-off error: $|x - fl(x)|/|x| < \epsilon_M$
- ϵ_M a.k.a. unit round-off, $\mathbf{u} = ULP(1.0)$
- ULP — Unit in the last place
    - the distance between the two closest *straddling* floating-point numbers a and b (i.e., those with $a \leq x \leq b$ and $a \neq b$),

Floating Point Representation Properties

- floating point number representation: $fl(x) = x(1+\delta)$, where $|\delta| < \epsilon_M$
- representation (round-off) error: $x - fl(x)$
- absolute round-off error: $|x - fl(x)| < \epsilon_M |x|$
- relative round-off error: $|x - fl(x)|/|x| < \epsilon_M$

    - floating point number representation: $fl(x) = x(1+\delta)$, where $|\delta| < \epsilon_M$
    - representation (round-off) error: $x - fl(x)$
    - absolute round-off error: $|x - fl(x)| < \epsilon_M |x|$
    - relative round-off error: $|x - fl(x)|/|x| < \epsilon_M$

- ϵ_M a.k.a. unit round-off, $\mathbf{u} = ULP(1.0)$
- ULP — Unit in the last place
    - the distance between the two closest *straddling* floating-point numbers a and b (i.e., those with $a \le x \le b$ and $a \ne b$),
    - a measure of precision in numeric calculations. In base b, if x has exponent E, then ULP(x) = $\epsilon_M \cdot b^E$.

Finite Differencing — Round-off Error I

- Consider

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

Finite Differencing — Round-off Error I

- Consider
$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$
- Suppose that x and $x+h$ exact

Finite Differencing — Round-off Error I

- Consider
$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$
- Suppose that x and $x+h$ exact
- Round-off errors when evaluating f:
$$\frac{f(x+h)(1+\delta_1) - f(x)(1+\delta_2)}{h} = \frac{f(x+h) - f(x)}{h} + \frac{\delta_1 f(x+h) - \delta_2 f(x)}{h}$$

Finite Differencing — Round-off Error I

- Consider
$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$
- Suppose that x and $x + h$ exact
- Round-off errors when evaluating f:
$$\frac{f(x+h)(1+\delta_1) - f(x)(1+\delta_2)}{h} = \frac{f(x+h) - f(x)}{h} + \frac{\delta_1 f(x+h) - \delta_2 f(x)}{h}$$
- We have $|\delta_i| \leq \epsilon_M$ for $i \in 1, 2$.

Finite Differencing — Round-off Error I

- Consider
$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$
- Suppose that x and $x+h$ exact
- Round-off errors when evaluating f:
$$\frac{f(x+h)(1+\delta_1) - f(x)(1+\delta_2)}{h} = \frac{f(x+h) - f(x)}{h} + \frac{\delta_1 f(x+h) - \delta_2 f(x)}{h}$$
- We have $|\delta_i| \leq \epsilon_M$ for $i \in 1, 2$.
- $\implies$ round-off error bound:
$$\epsilon_M \frac{|f(x+h)| + |f(x)|}{h}$$

Finite Differencing — Round-off Error I

- Consider
$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$
- Suppose that x and $x+h$ exact
- Round-off errors when evaluating f:
$$\frac{f(x+h)(1+\delta_1) - f(x)(1+\delta_2)}{h} = \frac{f(x+h) - f(x)}{h} + \frac{\delta_1 f(x+h) - \delta_2 f(x)}{h}$$
- We have $|\delta_i| \leq \epsilon_M$ for $i \in 1, 2$.
- $\implies$ round-off error bound:
$$\epsilon_M \frac{|f(x+h)| + |f(x)|}{h}$$
- $\implies$ round-off error bound order: $O(\frac{\epsilon_M}{h})$

Finite Differencing — Round-off Error I

- Consider
$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$
- Suppose that x and $x+h$ exact
- Round-off errors when evaluating f:
$$\frac{f(x+h)(1+\delta_1) - f(x)(1+\delta_2)}{h} = \frac{f(x+h) - f(x)}{h} + \frac{\delta_1 f(x+h) - \delta_2 f(x)}{h}$$
- We have $|\delta_i| \leq \epsilon_M$ for $i \in 1, 2$.
- $\implies$ round-off error bound:
$$\epsilon_M \frac{|f(x+h)| + |f(x)|}{h}$$
- $\implies$ round-off error bound order: $O(\frac{\epsilon_M}{h})$
- round-off error — **increases** with smaller h

Finite Differencing — Round-off Error I

- Consider
$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$
- Suppose that x and $x+h$ exact
- Round-off errors when evaluating f:
$$\frac{f(x+h)(1+\delta_1) - f(x)(1+\delta_2)}{h} = \frac{f(x+h) - f(x)}{h} + \frac{\delta_1 f(x+h) - \delta_2 f(x)}{h}$$
- We have $|\delta_i| \leq \epsilon_M$ for $i \in 1, 2$.
- $\implies$ round-off error bound:
$$\epsilon_M \frac{|f(x+h)| + |f(x)|}{h}$$
- $\implies$ round-off error bound order: $O(\frac{\epsilon_M}{h})$
- round-off error — **increases** with smaller h
- error contribution **inversely** proportial to h, proportionality constant $O(\epsilon_M)$

Finite Differencing — Round-off Error II

- $\epsilon_M \neq 0 \implies \exists \hat{h} : x + \hat{h} = x$

Finite Differencing — Round-off Error II

- $\epsilon_M \neq 0 \implies \exists \hat{h} : x + \hat{h} = x$
  - i.e., x and $x + h$ not guaranteed to be exactly representable, either

Finite Differencing — Round-off Error II

- $\epsilon_M \neq 0 \implies \exists \hat{h} : x + \hat{h} = x$
  - i.e., x and $x + h$ not guaranteed to be exactly representable, either
  - $\implies f(x) = f(x + \hat{h})$

Finite Differencing — Round-off Error II

- $\epsilon_M \neq 0 \implies \exists \hat{h} : x + \hat{h} = x$
  - i.e., x and $x + h$ not guaranteed to be exactly representable, either
  - $\implies f(x) = f(x + \hat{h})$
  - $\implies \frac{f(x+h) - f(x)}{\hat{h}} = 0$

Finite Differencing — Round-off Error II

- $\epsilon_M \neq 0 \implies \exists \hat{h} : x + \hat{h} = x$
  - i.e., x and $x + h$ not guaranteed to be exactly representable, either
  - $\implies f(x) = f(x + \hat{h})$
  - $\implies \frac{f(x+h) - f(x)}{\hat{h}} = 0$
  - $\implies$ no correct digits in the result at (or below) step-size $\hat{h}$

- Total error: $O(h) + O(\frac{\epsilon_M}{h})$ (forward-difference formula)

Finite Differencing — Trucation—Round-off Trade-off

- Total error: $O(h) + O(\frac{\epsilon_M}{h})$ (forward-difference formula)
- Best accuracy when contributions approximately equal

Finite Differencing — Trucation—Round-off Trade-off

- Total error: $O(h) + O(\frac{\epsilon_M}{h})$ (forward-difference formula)
- Best accuracy when contributions approximately equal
  - $h \approx \frac{\epsilon_M}{h}$

Finite Differencing — Trucation—Round-off Trade-off

- Total error: $O(h) + O(\frac{\epsilon_M}{h})$ (forward-difference formula)
- Best accuracy when contributions approximately equal
  - $h \approx \frac{\epsilon_M}{h}$
  - $h^2 \approx \epsilon_M$

Finite Differencing — Trucation—Round-off Trade-off

- Total error: $O(h) + O(\frac{\epsilon_M}{h})$ (forward-difference formula)
- Best accuracy when contributions approximately equal
  - $h \approx \frac{\epsilon_M}{h}$
  - $h^2 \approx \epsilon_M$
  - $h \approx \sqrt{\epsilon_M}$

Finite Differencing — Trucation—Round-off Trade-off

- Total error: $O(h) + O(\frac{\epsilon_M}{h})$ (forward-difference formula)
- Best accuracy when contributions approximately equal
  - $h \approx \frac{\epsilon_M}{h}$
  - $h^2 \approx \epsilon_M$
  - $h \approx \sqrt{\epsilon_M}$
    - binary64: `1.490116e-08`

Finite Differencing — Trucation—Round-off Trade-off

- Total error: $O(h) + O(\frac{\epsilon_M}{h})$ (forward-difference formula)
- Best accuracy when contributions approximately equal
    - $h \approx \frac{\epsilon_M}{h}$
    - $h^2 \approx \epsilon_M$
    - $h \approx \sqrt{\epsilon_M}$
        - binary64: `1.490116e-08`
    - total error's minimum value $\in O(\sqrt{\epsilon_M})$

- Total error: $O(h) + O(\frac{\epsilon_M}{h})$ (forward-difference formula)
- Best accuracy when contributions approximately equal
  - $h \approx \frac{\epsilon_M}{h}$
  - $h^2 \approx \epsilon_M$
  - $h \approx \sqrt{\epsilon_M}$
    - binary64: `1.490116e-08`
  - total error's minimum value $\in O(\sqrt{\epsilon_M})$
    - binary64: `1.490116e-08`

Finite Differencing — Trucation—Round-off Trade-off

- Total error: $O(h) + O(\frac{\epsilon_M}{h})$ (forward-difference formula)
- Best accuracy when contributions approximately equal
  - $h \approx \frac{\epsilon_M}{h}$
  - $h^2 \approx \epsilon_M$
  - $h \approx \sqrt{\epsilon_M}$
    - binary64: `1.490116e-08`
  - total error's minimum value $\in O(\sqrt{\epsilon_M})$
    - binary64: `1.490116e-08`
- Total error: $O(h^2) + O(\frac{\epsilon_M}{h})$ (central-difference formula)

Finite Differencing — Trucation—Round-off Trade-off

- Total error: $O(h) + O(\frac{\epsilon_M}{h})$ (forward-difference formula)
- Best accuracy when contributions approximately equal
  - $h \approx \frac{\epsilon_M}{h}$
  - $h^2 \approx \epsilon_M$
  - $h \approx \sqrt{\epsilon_M}$
    - binary64: `1.490116e-08`
  - total error's minimum value $\in O(\sqrt{\epsilon_M})$
    - binary64: `1.490116e-08`
- Total error: $O(h^2) + O(\frac{\epsilon_M}{h})$ (central-difference formula)
  - $h \approx \epsilon_M^{1/3}$

Finite Differencing — Trucation—Round-off Trade-off

- Total error: $O(h) + O(\frac{\epsilon_M}{h})$ (forward-difference formula)
- Best accuracy when contributions approximately equal
  - $h \approx \frac{\epsilon_M}{h}$
  - $h^2 \approx \epsilon_M$
  - $h \approx \sqrt{\epsilon_M}$
    - binary64: `1.490116e-08`
  - total error's minimum value $\in O(\sqrt{\epsilon_M})$
    - binary64: `1.490116e-08`
- Total error: $O(h^2) + O(\frac{\epsilon_M}{h})$ (central-difference formula)
  - $h \approx \epsilon_M^{1/3}$
    - binary64: `6.055454e-06`

Finite Differencing — Trucation—Round-off Trade-off

- Total error: $O(h) + O(\frac{\epsilon_M}{h})$ (forward-difference formula)
- Best accuracy when contributions approximately equal
  - $h \approx \frac{\epsilon_M}{h}$
  - $h^2 \approx \epsilon_M$
  - $h \approx \sqrt{\epsilon_M}$
    - binary64: `1.490116e-08`
  - total error's minimum value $\in O(\sqrt{\epsilon_M})$
    - binary64: `1.490116e-08`
- Total error: $O(h^2) + O(\frac{\epsilon_M}{h})$ (central-difference formula)
  - $h \approx \epsilon_M^{1/3}$
    - binary64: `6.055454e-06`
  - total error's minimum value $\in O(\epsilon_M^{2/3})$

- Total error: $O(h) + O(\frac{\epsilon_M}{h})$ (forward-difference formula)
- Best accuracy when contributions approximately equal
  - $h \approx \frac{\epsilon_M}{h}$
  - $h^2 \approx \epsilon_M$
  - $h \approx \sqrt{\epsilon_M}$
    - binary64: `1.490116e-08`
  - total error's minimum value $\in O(\sqrt{\epsilon_M})$
    - binary64: `1.490116e-08`
- Total error: $O(h^2) + O(\frac{\epsilon_M}{h})$ (central-difference formula)
  - $h \approx \epsilon_M^{1/3}$
    - binary64: `6.055454e-06`
  - total error's minimum value $\in O(\epsilon_M^{2/3})$
    - binary64: `3.666853e-11`

- Subtracting the backward-difference approximation from the forward-difference approximation:

Finite Differencing — Central-Difference Formula

- Subtracting the backward-difference approximation from the forward-difference approximation:
    - (central-difference)

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - O(h^2)$$

Finite Differencing — Central-Difference Formula

- Subtracting the backward-difference approximation from the forward-difference approximation:
  - (central-difference)

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - O(h^2)$$

- Truncation error $\in O(h^2)$ (second-order approximation)

Finite Differencing — Central-Difference Formula

- Subtracting the backward-difference approximation from the forward-difference approximation:
  - (central-difference)

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - O(h^2)$$

- Truncation error $\in O(h^2)$ (second-order approximation)
- Round-off error $\in O(\epsilon_M/h)$

Finite Differencing — Central-Difference Formula

- Subtracting the backward-difference approximation from the forward-difference approximation:
  - (central-difference)

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - O(h^2)$$

- Truncation error $\in O(h^2)$ (second-order approximation)
- Round-off error $\in O(\epsilon_M / h)$
- Balanced trade-off:

Finite Differencing — Central-Difference Formula

- Subtracting the backward-difference approximation from the forward-difference approximation:
  - (central-difference)

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - O(h^2)$$

- Truncation error $\in O(h^2)$ (second-order approximation)
- Round-off error $\in O(\epsilon_M/h)$
- Balanced trade-off:
  - $h \approx {\epsilon_M}^{1/3}$

Finite Differencing — Central-Difference Formula

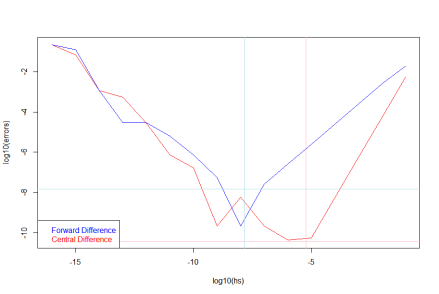
- Subtracting the backward-difference approximation from the forward-difference approximation:
  - (central-difference)

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - O(h^2)$$

- Truncation error $\in O(h^2)$ (second-order approximation)
- Round-off error $\in O(\epsilon_M/h)$
- Balanced trade-off:
  - $h \approx {\epsilon_M}^{1/3}$
  - total error's minimum value $\in O({\epsilon_M}^{2/3})$

Finite Difference Approximation Error

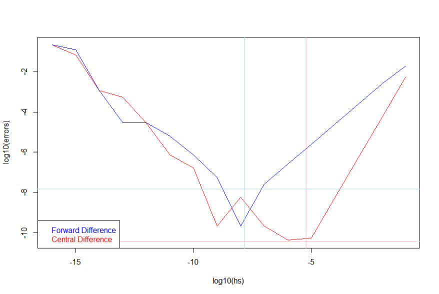
Example: $f(x) = cos(sin(x) * cos(x)) \quad f'(x) = -cos(2 * x) * sin(sin(x) * cos(x))$



- Total error: $O(h) + O(\frac{\epsilon_M}{h})$ (forward-difference formula)

Finite Difference Approximation Error

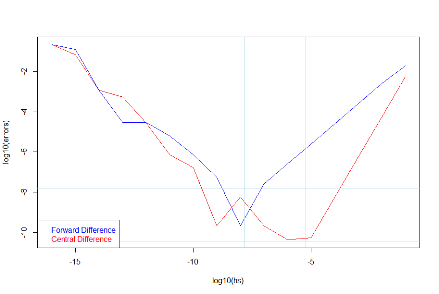
Example: $f(x) = cos(sin(x) * cos(x))$ $\quad f'(x) = -cos(2 * x) * sin(sin(x) * cos(x))$



- Total error: $O(h) + O(\frac{\epsilon_M}{h})$ (forward-difference formula)
  - $h \approx \sqrt{\epsilon_M}$ — binary64: `1.490116e-08`

Finite Difference Approximation Error

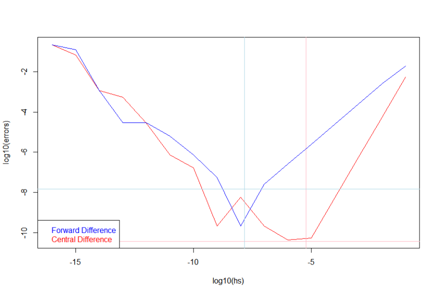
Example: $f(x) = cos(sin(x) * cos(x))$ $f'(x) = -cos(2 * x) * sin(sin(x) * cos(x))$



- Total error: $O(h) + O(\frac{\epsilon_M}{h})$ (forward-difference formula)
  - $h \approx \sqrt{\epsilon_M}$ — binary64: `1.490116e-08`
  - total error's minimum value $\in O(\sqrt{\epsilon_M})$ — binary64: `1.490116e-08`

Finite Difference Approximation Error

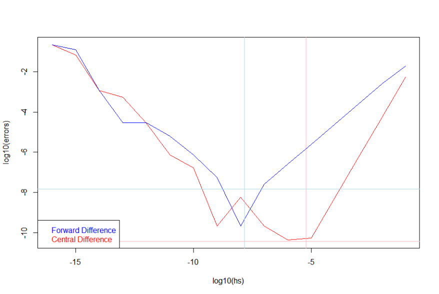
Example: $f(x) = cos(sin(x) * cos(x))$ $f'(x) = -cos(2 * x) * sin(sin(x) * cos(x))$



- Total error: $O(h) + O(\frac{\epsilon_M}{h})$ (forward-difference formula)
  - $h \approx \sqrt{\epsilon_M}$ — binary64: `1.490116e-08`
  - total error's minimum value $\in O(\sqrt{\epsilon_M})$ — binary64: `1.490116e-08`
- Total error: $O(h^2) + O(\frac{\epsilon_M}{h})$ (central-difference formula)

Finite Difference Approximation Error

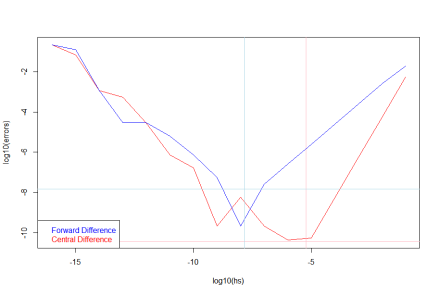
Example: $f(x) = cos(sin(x) * cos(x))$ $\quad f'(x) = -cos(2 * x) * sin(sin(x) * cos(x))$



- Total error: $O(h) + O(\frac{\epsilon_M}{h})$ (forward-difference formula)
  - $h \approx \sqrt{\epsilon_M}$ — binary64: `1.490116e-08`
  - total error's minimum value $\in O(\sqrt{\epsilon_M})$ — binary64: `1.490116e-08`
- Total error: $O(h^2) + O(\frac{\epsilon_M}{h})$ (central-difference formula)
  - $h \approx \epsilon_M^{1/3}$ — binary64: `6.055454e-06`

Finite Difference Approximation Error

Example: $f(x) = cos(sin(x) * cos(x))$ $f'(x) = -cos(2 * x) * sin(sin(x) * cos(x))$



- Total error: $O(h) + O(\frac{\epsilon_M}{h})$ (forward-difference formula)
  - $h \approx \sqrt{\epsilon_M}$ — binary64: `1.490116e-08`
  - total error's minimum value $\in O(\sqrt{\epsilon_M})$ — binary64: `1.490116e-08`
- Total error: $O(h^2) + O(\frac{\epsilon_M}{h})$ (central-difference formula)
  - $h \approx \epsilon_M^{1/3}$ — binary64: `6.055454e-06`
  - total error's minimum value $\in O(\epsilon_M^{2/3})$ — binary64: `3.666853e-11`

- Algebraic specification for f manipulated by symbolic manipulation tools to produce new algebraic expressions for each component of the gradient

- Algebraic specification for f manipulated by symbolic manipulation tools to produce new algebraic expressions for each component of the gradient
- Commonly used symbolic manipulation tools can be found in the packages Mathematica (also Wolfram Alpha), Macsyma, Maple, or Mathics.

Symbolic Differentiation

- Algebraic specification for f manipulated by symbolic manipulation tools to produce new algebraic expressions for each component of the gradient
- Commonly used symbolic manipulation tools can be found in the packages Mathematica (also Wolfram Alpha), Macsyma, Maple, or Mathics.
- Disadvantages:

Symbolic Differentiation

- Algebraic specification for f manipulated by symbolic manipulation tools to produce new algebraic expressions for each component of the gradient
- Commonly used symbolic manipulation tools can be found in the packages Mathematica (also Wolfram Alpha), Macsyma, Maple, or Mathics.
- Disadvantages:
  - narrow applicability

Symbolic Differentiation

- Algebraic specification for f manipulated by symbolic manipulation tools to produce new algebraic expressions for each component of the gradient
- Commonly used symbolic manipulation tools can be found in the packages Mathematica (also Wolfram Alpha), Macsyma, Maple, or Mathics.
- Disadvantages:
  - narrow applicability
  - expression swell

- Symbolic differentiation (SD) applicable to ”paper-and-pencil” algebraic expressions

- Symbolic differentiation (SD) applicable to ”paper-and-pencil” algebraic expressions
  - Input: function specified by an *algebraic formula* or *closed-form expression*

Numerical Objective Function & Algorithmic Differentiation I

- Symbolic differentiation (SD) applicable to ”paper-and-pencil” algebraic expressions
  - Input: function specified by an *algebraic formula* or *closed-form expression*
- Objective function in numerical optimization — a *computer program*

- Symbolic differentiation (SD) applicable to ”paper-and-pencil” algebraic expressions
  - Input: function specified by an *algebraic formula* or *closed-form expression*
- Objective function in numerical optimization — a *computer program*
- How are computer programs different?

Numerical Objective Function & Algorithmic Differentiation I

- Symbolic differentiation (SD) applicable to ”paper-and-pencil” algebraic expressions
  - Input: function specified by an *algebraic formula* or *closed-form expression*
- Objective function in numerical optimization — a *computer program*
- How are computer programs different?
  - named intermediates

Numerical Objective Function & Algorithmic Differentiation I

- Symbolic differentiation (SD) applicable to ”paper-and-pencil” algebraic expressions
  - Input: function specified by an *algebraic formula* or *closed-form expression*
- Objective function in numerical optimization — a *computer program*
- How are computer programs different?
  - named intermediates
  - program branches

Numerical Objective Function & Algorithmic Differentiation I

- Symbolic differentiation (SD) applicable to ”paper-and-pencil” algebraic expressions
  - Input: function specified by an *algebraic formula* or *closed-form expression*
- Objective function in numerical optimization — a *computer program*
- How are computer programs different?
  - named intermediates
  - program branches
  - joint allocations / mutation / overwriting

Numerical Objective Function & Algorithmic Differentiation I

- Symbolic differentiation (SD) applicable to ”paper-and-pencil” algebraic expressions
  - Input: function specified by an *algebraic formula* or *closed-form expression*
- Objective function in numerical optimization — a *computer program*
- How are computer programs different?
  - named intermediates
  - program branches
  - joint allocations / mutation / overwriting
  - iterative loops

Numerical Objective Function & Algorithmic Differentiation I

- Symbolic differentiation (SD) applicable to ”paper-and-pencil” algebraic expressions
    - Input: function specified by an *algebraic formula* or *closed-form expression*
- Objective function in numerical optimization — a *computer program*
- How are computer programs different?
    - named intermediates
    - program branches
    - joint allocations / mutation / overwriting
    - iterative loops
- Do we have to resort to finite differences?

Numerical Objective Function & Algorithmic Differentiation II

- Algorithmic Differentiation (AD) offers another solution.

- Algorithmic Differentiation (AD) offers another solution.
- AD — a computational differentiation method with the roots in the late 1960s, Griewank (2012)

- Algorithmic Differentiation (AD) offers another solution.
- AD — a computational differentiation method with the roots in the late 1960s, Griewank (2012)
  - Machine Learning — Neural Networks — Backpropagation: Bryson, Denham, Dreyfus (1963), Werbos (1974)

Numerical Objective Function & Algorithmic Differentiation II

- Algorithmic Differentiation (AD) offers another solution.
- AD — a computational differentiation method with the roots in the late 1960s, Griewank (2012)
  - Machine Learning — Neural Networks — Backpropagation: Bryson, Denham, Dreyfus (1963), Werbos (1974)
  - AD widely known and applied in natural sciences — Naumann (2012), and computational finance — Giles and Glasserman (2006), Homescu (2011)

- Idea:

Algorithmic Differentiation — Idea I

- Idea:
  - computer code for evaluating the function can be broken down

Algorithmic Differentiation — Idea I

- Idea:
  - computer code for evaluating the function can be broken down
  - into a composition of elementary arithmetic operations

Algorithmic Differentiation — Idea I

- Idea:
  - computer code for evaluating the function can be broken down
  - into a composition of elementary arithmetic operations
  - to which the chain rule (one of the basic rules of calculus) can be applied

Algorithmic Differentiation — Idea I

- Idea:
  - computer code for evaluating the function can be broken down
  - into a composition of elementary arithmetic operations
  - to which the chain rule (one of the basic rules of calculus) can be applied
  - e.g., given an objective function $Q(\theta) = f(g(\theta))$, we have

$$Q' = (f \circ g)' = (f' \circ g) \cdot g'$$

- Idea:
  - computer code for evaluating the function can be broken down
  - into a composition of elementary arithmetic operations
  - to which the chain rule (one of the basic rules of calculus) can be applied
  - e.g., given an objective function $Q(\theta) = f(g(\theta))$, we have

$$Q' = (f \circ g)' = (f' \circ g) \cdot g'$$

- Relies on the systematic application of the chain rule by a computer program

Algorithmic Differentiation — Idea I

- Idea:
  - computer code for evaluating the function can be broken down
  - into a composition of elementary arithmetic operations
  - to which the chain rule (one of the basic rules of calculus) can be applied
  - e.g., given an objective function $Q(\theta) = f(g(\theta))$, we have

$$Q' = (f \circ g)' = (f' \circ g) \cdot g'$$

- Relies on the systematic application of the chain rule by a computer program
  - operating at the source code level

Algorithmic Differentiation — Idea I

- Idea:
  - computer code for evaluating the function can be broken down
  - into a composition of elementary arithmetic operations
  - to which the chain rule (one of the basic rules of calculus) can be applied
  - e.g., given an objective function $Q(\theta) = f(g(\theta))$, we have

$$Q' = (f \circ g)' = (f' \circ g) \cdot g'$$

- Relies on the systematic application of the chain rule by a computer program
  - operating at the source code level
  - performed automatically

Algorithmic Differentiation — Idea I

- Idea:
  - computer code for evaluating the function can be broken down
  - into a composition of elementary arithmetic operations
  - to which the chain rule (one of the basic rules of calculus) can be applied
  - e.g., given an objective function $Q(\theta) = f(g(\theta))$, we have

$$Q' = (f \circ g)' = (f' \circ g) \cdot g'$$

- Relies on the systematic application of the chain rule by a computer program
  - operating at the source code level
  - performed automatically
  - with minimal or no user intervention

Algorithmic Differentiation — Idea II

- Mathematical operations performed *exactly*

- Mathematical operations performed *exactly*
    - to the maximum extent allowed by the machine precision — i.e., accurate to within round-off error

- Mathematical operations performed *exactly*
  - to the maximum extent allowed by the machine precision — i.e., accurate to within round-off error
- truncation error: none

Algorithmic Differentiation — Idea II

- Mathematical operations performed *exactly*
  - to the maximum extent allowed by the machine precision — i.e., accurate to within round-off error
- truncation error: none
- round-off error: equivalent to using a source code analogously implementing the analytical derivatives.

Algorithmic Differentiation — Idea III

- AD similar to symbolic differentiation

- AD similar to symbolic differentiation
  - major difference:

- AD similar to symbolic differentiation
  - major difference:
    - SD restricted to symbolic manipulation of algebraic expressions

Algorithmic Differentiation — Idea III

- AD similar to symbolic differentiation
  - major difference:
    - SD restricted to symbolic manipulation of algebraic expressions
    - AD operates on (and integrates with) the existing source code of a computer program, Nocedal and Wright (2006 Section 8.2).

- AD similar to symbolic differentiation
  - major difference:
    - SD restricted to symbolic manipulation of algebraic expressions
    - AD operates on (and integrates with) the existing source code of a computer program, Nocedal and Wright (2006 Section 8.2).
  - Note: less than required by SD, more than required by finite differences (which can operate on black box implementations)

Algorithmic Differentiation — Idea III

- AD similar to symbolic differentiation
  - major difference:
    - SD restricted to symbolic manipulation of algebraic expressions
    - AD operates on (and integrates with) the existing source code of a computer program, Nocedal and Wright (2006 Section 8.2).
  - Note: less than required by SD, more than required by finite differences (which can operate on black box implementations)
    - AD Input: (source code of) a computer program for evaluating a vector function

Algorithmic Differentiation — Idea III

- AD similar to symbolic differentiation
  - major difference:
    - SD restricted to symbolic manipulation of algebraic expressions
    - AD operates on (and integrates with) the existing source code of a computer program, Nocedal and Wright (2006 Section 8.2).
  - Note: less than required by SD, more than required by finite differences (which can operate on black box implementations)
    - AD Input: (source code of) a computer program for evaluating a vector function
    - AD Output: (source code of) a computer program implementing derivative(s) thereof

- Two implementation styles, Naumann (2012):

- Two implementation styles, Naumann (2012):
    - source code transformation (SCT)

- Two implementation styles, Naumann (2012):
  - source code transformation (SCT)
    - e.g., ADIFOR — produce new code that calculates both function and derivative values

- Two implementation styles, Naumann (2012):
  - source code transformation (SCT)
    - e.g., ADIFOR — produce new code that calculates both function and derivative values
  - operator overloading

Algorithmic Differentiation — Implementations

- Two implementation styles, Naumann (2012):
  - source code transformation (SCT)
    - e.g., ADIFOR — produce new code that calculates both function and derivative values
  - operator overloading
    - e.g., ADOL-C — keep track of the elementary computations during the function evaluation, produce the derivatives at given inputs

Algorithmic Differentiation — Implementations

- Two implementation styles, Naumann (2012):
  - source code transformation (SCT)
    - e.g., ADIFOR — produce new code that calculates both function and derivative values
  - operator overloading
    - e.g., ADOL-C — keep track of the elementary computations during the function evaluation, produce the derivatives at given inputs
- Two modes: *forward mode* and *reverse mode*

Algorithmic Differentiation — Implementations

- Two implementation styles, Naumann (2012):
  - source code transformation (SCT)
    - e.g., ADIFOR — produce new code that calculates both function and derivative values
  - operator overloading
    - e.g., ADOL-C — keep track of the elementary computations during the function evaluation, produce the derivatives at given inputs
- Two modes: *forward mode* and *reverse mode*
- Example: Eigen's Auto Diff module

MLE

- a more concrete illustration: example in the context of statistical estimation

MLE

- a more concrete illustration: example in the context of statistical estimation
- evaluation and differentiation of the loglikelihood function of a normally distributed sample

MLE

- a more concrete illustration: example in the context of statistical estimation
- evaluation and differentiation of the loglikelihood function of a normally distributed sample
- we have $T > 1$ observations $x_1, x_2, \ldots, x_T \overset{\text{i.i.d.}}{\sim} \mathcal{N}(\mu, \sigma^2)$, with $\mu \in \mathbb{R}$, and $\sigma \in \mathbb{R}_+$

MLE

- a more concrete illustration: example in the context of statistical estimation
- evaluation and differentiation of the loglikelihood function of a normally distributed sample
- we have $T > 1$ observations $x_1, x_2, ..., x_T \overset{\text{i.i.d.}}{\sim} \mathcal{N}(\mu, \sigma^2)$, with $\mu \in \mathbb{R}$, and $\sigma \in \mathbb{R}_+$
- our parameter vector is $\theta = (\mu, \sigma^2)$

MLE

- a more concrete illustration: example in the context of statistical estimation
- evaluation and differentiation of the loglikelihood function of a normally distributed sample
- we have $T > 1$ observations $x_1, x_2, \ldots, x_T \overset{\text{i.i.d.}}{\sim} \mathcal{N}(\mu,\, \sigma^2)$, with $\mu \in \mathbb{R}$, and $\sigma \in \mathbb{R}_+$
- our parameter vector is $\theta = (\mu,\, \sigma^2)$
- the likelihood function is $\mathcal{L}(\theta; x) = f(x; \theta) = (\sigma\sqrt{2\pi})^{-T} \exp[-\frac{1}{2} \sum_{t=1}^{T} (x_t - \mu)^2 / \sigma^2]$

- objective function: the loglikelihood function, $\ell(\theta) = log\mathcal{L}(\theta; x) = \sum_{t=1}^{T} \ell_t(\theta)$

- objective function: the loglikelihood function, $\ell(\theta) = log\mathcal{L}(\theta; x) = \sum_{t=1}^{T} \ell_t(\theta)$
  - a contribution given by $\ell_t(\theta) = -\frac{T}{2}log(2\pi) - \frac{T}{2}log(\sigma^2) - \frac{1}{2}(x_t - \mu)^2/\sigma^2.$

- objective function: the loglikelihood function, $\ell(\theta) = log\mathcal{L}(\theta; x) = \sum_{t=1}^{T} \ell_t(\theta)$
  - a contribution given by $\ell_t(\theta) = -\frac{T}{2}log(2\pi) - \frac{T}{2}log(\sigma^2) - \frac{1}{2}(x_t - \mu)^2/\sigma^2$.
- goal: compute the value of ℓ_t corresponding to the arguments $x_t = 6$, $\mu = 6$, and $\sigma^2 = 4$.

- objective function: the loglikelihood function, $\ell(\theta) = log\mathcal{L}(\theta; x) = \sum_{t=1}^{T} \ell_t(\theta)$
  - a contribution given by $\ell_t(\theta) = -\frac{T}{2} log(2\pi) - \frac{T}{2} log(\sigma^2) - \frac{1}{2}(x_t - \mu)^2/\sigma^2$.
- goal: compute the value of ℓ_t corresponding to the arguments $x_t = 6$, $\mu = 6$, and $\sigma^2 = 4$.
- A computer program may execute the sequence of operations represented by an *evaluation trace*, Griewank and Walther (2008)

Variable		Operation		Expression		Value
v_{-1}			$=$	μ	$=$	6
v_0			$=$	σ^2	$=$	4
v_1	$=$	$\phi_1(v_0)$	$=$	$log(v_0)$	$=$	1.386294
v_2	$=$	$\phi_2(v_1)$	$=$	$-\frac{1}{2}log(2\pi) - \frac{1}{2}v_1$	$=$	-1.612086
v_3	$=$	$\phi_3(v_{-1})$	$=$	$x_t - v_{-1}$	$=$	0
v_4	$=$	$\phi_4(v_3)$	$=$	v_3^2	$=$	0
v_5	$=$	$\phi_5(v_0, v_4)$	$=$	v_4/v_0	$=$	0
v_6	$=$	$\phi_6(v_5)$	$=$	$-\frac{1}{2}v_5$	$=$	0
v_7	$=$	$\phi_7(v_2, v_6)$	$=$	$v_2 + v_6$	$=$	-1.612086
ℓ_t	$=$	v_7	$=$	-1.612086		

- evaluation of ℓ_t broken down

- evaluation of ℓ_t broken down
- into a composition of intermediate subexpressions v_i, with $i \in \{1, \dots, 7\}$

Algorithmic Differentiation — Evaluation Trace

- evaluation of ℓ_t broken down
- into a composition of intermediate subexpressions v_i, with $i \in \{1, \dots, 7\}$
  - intermediate subexpressions: elementary operations and functions ϕ_i with known, predefined derivatives

Algorithmic Differentiation — Evaluation Trace

- evaluation of ℓ_t broken down
- into a composition of intermediate subexpressions v_i, with $i \in \{1, \ldots, 7\}$
  - intermediate subexpressions: elementary operations and functions ϕ_i with known, predefined derivatives
  - note: similar to the static single assignment form (SSA) used in compiler design and implementation

- evaluation of ℓ_t broken down
- into a composition of intermediate subexpressions v_i, with $i \in \{1, \dots, 7\}$
  - intermediate subexpressions: elementary operations and functions ϕ_i with known, predefined derivatives
  - note: similar to the static single assignment form (SSA) used in compiler design and implementation
- We can now apply the chain rule to the intermediate operations v_i and collect the intermediate derivative results in order to obtain the derivative of the final expression $\ell_t = v_7$

Algorithmic Differentiation — Evaluation Trace

- evaluation of ℓ_t broken down
- into a composition of intermediate subexpressions v_i, with $i \in \{1, \dots, 7\}$
  - intermediate subexpressions: elementary operations and functions ϕ_i with known, predefined derivatives
  - note: similar to the static single assignment form (SSA) used in compiler design and implementation
- We can now apply the chain rule to the intermediate operations v_i and collect the intermediate derivative results in order to obtain the derivative of the final expression $\ell_t = v_7$
- Recall: Two *modes* of propagating the intermediate information: the *forward mode* and the *reverse mode*

- evaluation of ℓ_t broken down
- into a composition of intermediate subexpressions v_i, with $i \in \{1, \ldots, 7\}$
  - intermediate subexpressions: elementary operations and functions ϕ_i with known, predefined derivatives
  - note: similar to the static single assignment form (SSA) used in compiler design and implementation
- We can now apply the chain rule to the intermediate operations v_i and collect the intermediate derivative results in order to obtain the derivative of the final expression $\ell_t = v_7$
- Recall: Two *modes* of propagating the intermediate information: the *forward mode* and the *reverse mode*
  - forward mode: Eigen's Auto Diff module used here

- goal: the derivative of the output variable ℓ_t (dependent variable) with respect to the input variable μ (independent variable)

- goal: the derivative of the output variable ℓ_t (dependent variable) with respect to the input variable μ (independent variable)
- idea: compute the numerical value of the derivative of each of the intermediate subexpressions, keeping the track of the resulting derivatives values along the way.

- goal: the derivative of the output variable ℓ_t (dependent variable) with respect to the input variable μ (independent variable)
- idea: compute the numerical value of the derivative of each of the intermediate subexpressions, keeping the track of the resulting derivatives values along the way.
    - for each variable v_i: introduce another variable, $\dot{v}_i = \partial v_i / \partial \mu$

- goal: the derivative of the output variable ℓ_t (dependent variable) with respect to the input variable μ (independent variable)
- idea: compute the numerical value of the derivative of each of the intermediate subexpressions, keeping the track of the resulting derivatives values along the way.
  - for each variable v_i: introduce another variable, $\dot{v}_i = \partial v_i / \partial \mu$
  - then: apply the chain rule mechanically to each intermediate subexpression

- goal: the derivative of the output variable ℓ_t (dependent variable) with respect to the input variable μ (independent variable)
- idea: compute the numerical value of the derivative of each of the intermediate subexpressions, keeping the track of the resulting derivatives values along the way.
  - for each variable v_i: introduce another variable, $\dot{v}_i = \partial v_i / \partial \mu$
  - then: apply the chain rule mechanically to each intermediate subexpression
  - assigning the resulting numerical value to each $\dot{v}_i$.

- in our case

- in our case
  - start with $\dot{v}_{-1} = 1.0$ (active differentation variable μ),

- in our case
    - start with $\dot{v}_{-1} = 1.0$ (active differentation variable μ),
    - $\dot{v}_0 = 0.0$ (passive: we are not differentiating with respect to σ^2 and thus treat it as a constant)

- in our case
  - start with $\dot{v}_{-1} = 1.0$ (active differentation variable μ),
  - $\dot{v}_0 = 0.0$ (passive: we are not differentiating with respect to σ^2 and thus treat it as a constant)
  - then, given that $v_1 = \phi_1(v_0) = log(v_0)$, we obtain $\dot{v}_1 = (\partial\phi_1(v_0)/\partial v_0)\dot{v}_0 = \dot{v}_0/v_0 = 0.0$

- in our case
  - start with $\dot{v}_{-1} = 1.0$ (active differentation variable μ),
  - $\dot{v}_0 = 0.0$ (passive: we are not differentiating with respect to σ^2 and thus treat it as a constant)
  - then, given that $v_1 = \phi_1(v_0) = log(v_0)$, we obtain $\dot{v}_1 = (\partial\phi_1(v_0)/\partial v_0)\dot{v}_0 = \dot{v}_0/v_0 = 0.0$
  - analogously, $v_2 = \phi_2(v_1) = -\frac{1}{2}log(2\pi) - \frac{1}{2}v_1$, hence $\dot{v}_2 = (\partial\phi_2(v_1)/\partial v_1)\dot{v}_1 = -\frac{1}{2}\dot{v}_1 = 0.0$

- in our case
  - start with $\dot{v}_{-1} = 1.0$ (active differentation variable μ),
  - $\dot{v}_0 = 0.0$ (passive: we are not differentiating with respect to σ^2 and thus treat it as a constant)
  - then, given that $v_1 = \phi_1(v_0) = log(v_0)$, we obtain $\dot{v}_1 = (\partial\phi_1(v_0)/\partial v_0)\dot{v}_0 = \dot{v}_0/v_0 = 0.0$
  - analogously, $v_2 = \phi_2(v_1) = -\frac{1}{2}log(2\pi) - \frac{1}{2}v_1$, hence $\dot{v}_2 = (\partial\phi_2(v_1)/\partial v_1)\dot{v}_1 = -\frac{1}{2}\dot{v}_1 = 0.0$
- by continuing this procedure we ultimately obtain an augmented, forward-derived, evaluation trace

Variable		Operation		Expression		Value
v_{-1}			$=$	μ	$=$	6
$\dot{v}_{-1}$			$=$	$\partial v_{-1}/\partial v_{-1}$	$=$	1
v_0			$=$	σ^2	$=$	4
$\dot{v}_0$			$=$	$\partial v_0/\partial v_{-1}$	$=$	0
v_1	$=$	$\phi_1(v_0)$	$=$	$log(v_0)$	$=$	1.386294
$\dot{v}_1$	$=$	$(\partial\phi_1/\partial v_0)\dot{v}_0$	$=$	$\dot{v}_0/v_0$	$=$	0
...		...		...		...
v_7	$=$	$\phi_7(v_2, v_6)$	$=$	$v_2 + v_6$	$=$	-1.612086
$\dot{v}_7$	$=$	$(\partial\phi_7/\partial v_2)\dot{v}_2 + (\partial\phi_7/\partial v_6)\dot{v}_6$	$=$	$\dot{v}_2 + \dot{v}_6$	$=$	0
ℓ_t	$=$	v_7	$=$	-1.612086		
$\dot{\ell}_t$	$=$	$\dot{v}_7$	$=$	0		

- The basic forward mode of AD — called “forward” because the derivatives values $\dot{v}_i$ carried along simultaneously with the values v_i themselves

- The basic forward mode of AD — called “forward” because the derivatives values $\dot{v}_i$ carried along simultaneously with the values v_i themselves
- The problem addressed by the implementations:

- The basic forward mode of AD — called “forward” because the derivatives values $\dot{v}_i$ carried along simultaneously with the values v_i themselves
- The problem addressed by the implementations:
  - ”transform a program with a particular evaluation trace into an augmented program whose evaluation trace also contains exactly the same extra variables and additional lines as in the derived evaluation trace,” Griewank and Walther (2008).

- An alternative to the forward approach — the *reverse* or *adjoint* mode.

AD — Reverse Mode

- An alternative to the forward approach — the *reverse* or *adjoint* mode.
- Griewank and Walther (2008): ”Rather than choosing an input variable and calculating the sensitivity of every intermediate variable with respect to that input, we choose instead an output variable and calculate the sensitivity of that output with respect to each of the intermediate variables. As it turns out adjoint sensitivities must naturally be computed backward, i.e., starting from the output variables.”

- An alternative to the forward approach — the *reverse* or *adjoint* mode.
- Griewank and Walther (2008): ”Rather than choosing an input variable and calculating the sensitivity of every intermediate variable with respect to that input, we choose instead an output variable and calculate the sensitivity of that output with respect to each of the intermediate variables. As it turns out adjoint sensitivities must naturally be computed backward, i.e., starting from the output variables.”
- This talk: we shall not treat it any further detail and refer anyone interested to Griewank and Walther (2008).

- An alternative to the forward approach — the *reverse* or *adjoint* mode.
- Griewank and Walther (2008): ”Rather than choosing an input variable and calculating the sensitivity of every intermediate variable with respect to that input, we choose instead an output variable and calculate the sensitivity of that output with respect to each of the intermediate variables. As it turns out adjoint sensitivities must naturally be computed backward, i.e., starting from the output variables.”
- This talk: we shall not treat it any further detail and refer anyone interested to Griewank and Walther (2008).
- Worth noting: the choice between the modes depends on the computational cost.

- In general, for $f : \mathbb{R}^n \to \mathbb{R}^m$

- In general, for $f : \mathbb{R}^n \to \mathbb{R}^m$
  - rough $\text{Cost}_{AD,Forward}(f) \in n\ O(\text{Cost}(f))$

- In general, for $f : \mathbb{R}^n \to \mathbb{R}^m$
  - rough $\mathrm{Cost}_{AD,Forward}(f) \in n\ \mathrm{O}(\mathrm{Cost}(f))$
  - rough $\mathrm{Cost}_{AD,Reverse}(f) \in m\ \mathrm{O}(\mathrm{Cost}(f))$

- In general, for $f : \mathbb{R}^n \to \mathbb{R}^m$
  - rough $\text{Cost}_{AD,Forward}(f) \in n\ \text{O}(\text{Cost}(f))$
  - rough $\text{Cost}_{AD,Reverse}(f) \in m\ \text{O}(\text{Cost}(f))$
- Which is better? Depends: e.g., gradient vs. Jacobian

- In general, for $f : \mathbb{R}^n \to \mathbb{R}^m$
  - rough $\text{Cost}_{AD,Forward}(f) \in n\ \text{O}(\text{Cost}(f))$
  - rough $\text{Cost}_{AD,Reverse}(f) \in m\ \text{O}(\text{Cost}(f))$
- Which is better? Depends: e.g., gradient vs. Jacobian
  - Foward mode performance: can establish a worst-case performance bound for AD when used for gradient computation

The Model: GARCH(1, 1)

The spot asset price S follows the following process under the physical probability measure P:

$$\log \frac{S_{t+1}}{S_t} \equiv r_{t+1} = \mu + \epsilon_{t+1} \tag{1}$$

$$\epsilon_{t+1} = \sqrt{v_{t+1}} w_{t+1} \tag{2}$$

$$v_{t+1} = \omega + a\epsilon_t^2 + b v_t \tag{3}$$

$$w \overset{P}{\sim} GWN(0, 1) \tag{4}$$

- $\theta = (\mu, \omega, a, b)^\top$

Illustration: Simulation Study — Setup — Notation

- $\theta = (\mu, \omega, a, b)^\top$
- r — logarithmic return of the spot price

Illustration: Simulation Study — Setup — Notation

- $\theta = (\mu, \omega, a, b)^\top$
- r — logarithmic return of the spot price
- μ — conditional mean of r

Illustration: Simulation Study — Setup — Notation

- $\theta = (\mu, \omega, a, b)^\top$
- r — logarithmic return of the spot price
- μ — conditional mean of r
- v — conditional variance of r

Illustration: Simulation Study — Setup — Notation

- $\theta = (\mu, \omega, a, b)^\top$
- r — logarithmic return of the spot price
- μ — conditional mean of r
- v — conditional variance of r
  - non-negativity: $\omega, a, b > 0$

Illustration: Simulation Study — Setup — Notation

- $\theta = (\mu, \omega, a, b)^\top$
- r — logarithmic return of the spot price
- μ — conditional mean of r
- v — conditional variance of r
    - non-negativity: $\omega, a, b > 0$
- w — randomness source (innovation): standard Gaussian white noise

Illustration: Simulation Study — Setup — Notation

- $\theta = (\mu, \omega, a, b)^\top$
- r — logarithmic return of the spot price
- μ — conditional mean of r
- v — conditional variance of r
  - non-negativity: $\omega, a, b > 0$
- w — randomness source (innovation): standard Gaussian white noise
- $m_v = \frac{\omega}{1-a-b}$ — the unconditional mean of v (unconditional variance of r)

Illustration: Simulation Study — Setup — Notation

- $\theta = (\mu, \omega, a, b)^\top$
- r — logarithmic return of the spot price
- μ — conditional mean of r
- v — conditional variance of r
  - non-negativity: $\omega, a, b > 0$
- w — randomness source (innovation): standard Gaussian white noise
- $m_v = \frac{\omega}{1-a-b}$ — the unconditional mean of v (unconditional variance of r)
- $p_v = (a + b)$ — the mean-reversion rate or the persistence rate

Illustration: Simulation Study — Setup — Notation

- $\theta = (\mu, \omega, a, b)^\top$
- r — logarithmic return of the spot price
- μ — conditional mean of r
- v — conditional variance of r
  - non-negativity: $\omega, a, b > 0$
- w — randomness source (innovation): standard Gaussian white noise
- $m_v = \frac{\omega}{1-a-b}$ — the unconditional mean of v (unconditional variance of r)
- $p_v = (a + b)$ — the mean-reversion rate or the persistence rate
  - when $p_v < 1$, weak stationarity, the conditional variance will revert to the unconditional variance at a geometric rate of p_v

Illustration: Simulation Study — Setup — Notation

- $\theta = (\mu, \omega, a, b)^\top$
- r — logarithmic return of the spot price
- μ — conditional mean of r
- v — conditional variance of r
  - non-negativity: $\omega, a, b > 0$
- w — randomness source (innovation): standard Gaussian white noise
- $m_v = \frac{\omega}{1-a-b}$ — the unconditional mean of v (unconditional variance of r)
- $p_v = (a + b)$ — the mean-reversion rate or the persistence rate
  - when $p_v < 1$, weak stationarity, the conditional variance will revert to the unconditional variance at a geometric rate of p_v
  - smaller p_v — less persistent sensitivity of the volatility expectation to past market shocks

Illustration: Simulation Study — Setup — Notation

- $\theta = (\mu, \omega, a, b)^\top$
- r — logarithmic return of the spot price
- μ — conditional mean of r
- v — conditional variance of r
  - non-negativity: $\omega, a, b > 0$
- w — randomness source (innovation): standard Gaussian white noise
- $m_v = \frac{\omega}{1-a-b}$ — the unconditional mean of v (unconditional variance of r)
- $p_v = (a + b)$ — the mean-reversion rate or the persistence rate
  - when $p_v < 1$, weak stationarity, the conditional variance will revert to the unconditional variance at a geometric rate of p_v
  - smaller p_v — less persistent sensitivity of the volatility expectation to past market shocks
  - $p_v = 1$ — integrated GARCH (IGARCH) process

- $\theta = (\mu, \omega, a, b)^\top$
- r — logarithmic return of the spot price
- μ — conditional mean of r
- v — conditional variance of r
  - non-negativity: $\omega, a, b > 0$
- w — randomness source (innovation): standard Gaussian white noise
- $m_v = \frac{\omega}{1-a-b}$ — the unconditional mean of v (unconditional variance of r)
- $p_v = (a + b)$ — the mean-reversion rate or the persistence rate
  - when $p_v < 1$, weak stationarity, the conditional variance will revert to the unconditional variance at a geometric rate of p_v
  - smaller p_v — less persistent sensitivity of the volatility expectation to past market shocks
  - $p_v = 1$ — integrated GARCH (IGARCH) process
  - common finding: p_v close to 1 in financial data, ”near-integrated” process, Bollerslev and Engle (1993).

- DGP (data-generating process): near-integrated scenario, θ_{DGP}: $\mu = 0.01$ (risk-free rate), $\omega = 2e - 6$, $a = 0.10$, $b = 0.85$

- DGP (data-generating process): near-integrated scenario, θ_{DGP}: $\mu = 0.01$ (risk-free rate), $\omega = 2e - 6$, $a = 0.10$, $b = 0.85$
- M = 10,000 (MC estimation experiments / replications count)

Illustration: Monte Carlo Simulation

- DGP (data-generating process): near-integrated scenario, θ_{DGP}: $\mu = 0.01$ (risk-free rate), $\omega = 2e-6$, $a = 0.10$, $b = 0.85$
- M = 10,000 (MC estimation experiments / replications count)
- N = 2,000 … 16,000 (sample size)

- DGP (data-generating process): near-integrated scenario, θ_{DGP}: $\mu = 0.01$ (risk-free rate), $\omega = 2e - 6$, $a = 0.10$, $b = 0.85$
- M = 10,000 (MC estimation experiments / replications count)
- N = 2,000 … 16,000 (sample size)
  - 2,000 observations / 252 business days — roughly 8 years of data

Illustration: Monte Carlo Simulation

- DGP (data-generating process): near-integrated scenario, θ_{DGP}: $\mu = 0.01$ (risk-free rate), $\omega = 2e-6$, $a = 0.10$, $b = 0.85$
- M = 10,000 (MC estimation experiments / replications count)
- N = 2,000 … 16,000 (sample size)
    - 2,000 observations / 252 business days — roughly 8 years of data
    - S&P 500 from January 3rd, 1950 — over 16,000 observations

Illustration: Reliability — AD, L-BFGS

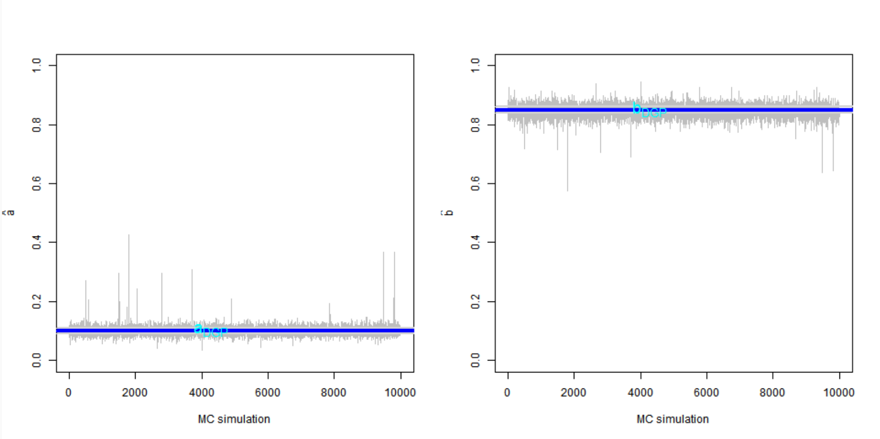
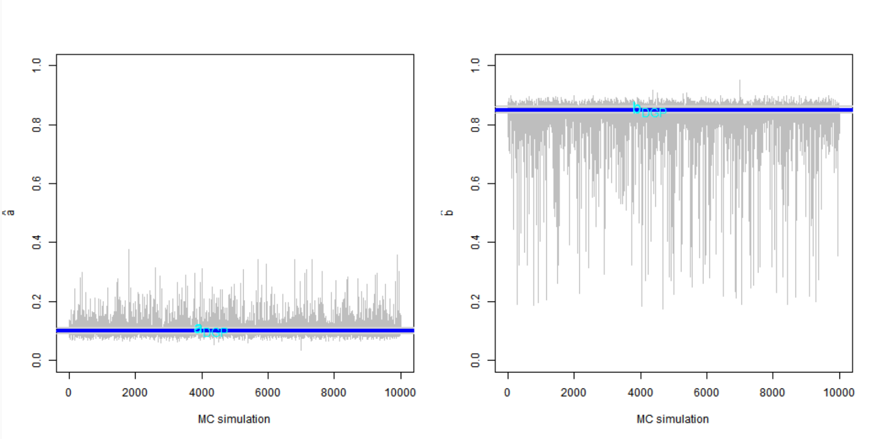


Illustration: Reliability — FD, TNR



- Reliability of estimation:

Illustration: Findings — Reliability

- Reliability of estimation:
  - using AD: successful convergence in the vast majority of Monte Carlo experiments

Illustration: Findings — Reliability

- Reliability of estimation:
  - using AD: successful convergence in the vast majority of Monte Carlo experiments
  - using FD: very high convergence failure ratio

- Reliability of estimation:
  - using AD: successful convergence in the vast majority of Monte Carlo experiments
  - using FD: very high convergence failure ratio
- Successful-convergence ratios (fraction of MC experiments with successful convergence within a 0.01 neighborhood true DGP values) for the best algorithm-gradient pairs are as follows:

- Reliability of estimation:
  - using AD: successful convergence in the vast majority of Monte Carlo experiments
  - using FD: very high convergence failure ratio
- Successful-convergence ratios (fraction of MC experiments with successful convergence within a 0.01 neighborhood true DGP values) for the best algorithm-gradient pairs are as follows:
  - L-BFGS(AD): average = 0.811, median: 0.99999903,

Illustration: Findings — Reliability

- Reliability of estimation:
  - using AD: successful convergence in the vast majority of Monte Carlo experiments
  - using FD: very high convergence failure ratio
- Successful-convergence ratios (fraction of MC experiments with successful convergence within a 0.01 neighborhood true DGP values) for the best algorithm-gradient pairs are as follows:
  - L-BFGS(AD): average = 0.811, median: 0.99999903,
  - L-BFGS(FD): average = 0.446, median: 0.02893254 (*sic*),

- Reliability of estimation:
  - using AD: successful convergence in the vast majority of Monte Carlo experiments
  - using FD: very high convergence failure ratio
- Successful-convergence ratios (fraction of MC experiments with successful convergence within a 0.01 neighborhood true DGP values) for the best algorithm-gradient pairs are as follows:
  - L-BFGS(AD): average = 0.811, median: 0.99999903,
  - L-BFGS(FD): average = 0.446, median: 0.02893254 (*sic*),
  - Truncated Newton with restarting (TNR) (FD): average = 0.708, median: 0.99856261.

- Overall: When using the fastest successful algorithm-gradient pair for both AD and FD:

- Overall: When using the fastest successful algorithm-gradient pair for both AD and FD:
  - AD achieves 12.5 times higher accuracy (in terms of the error norm) with a 3 slowdown compared to L-BFGS(FD)

Illustration: Findings

- Overall: When using the fastest successful algorithm-gradient pair for both AD and FD:
    - AD achieves 12.5 times higher accuracy (in terms of the error norm) with a 3 slowdown compared to L-BFGS(FD)
    - while achieving 4 times higher accuracy and 1.1 speedup compared to TNR(FD).

Example Source Code

```r
Rcpp::sourceCpp('ExampleGaussianRcpp.cpp')
```

- Note: Rcpp not necessary for any of this:

Example using Rcpp

```r
Rcpp::sourceCpp('ExampleGaussianRcpp.cpp')
```

- Note: Rcpp not necessary for any of this:
  - Feel free to skip over any line with `Rcpp::`

Example using Rcpp

```
Rcpp::sourceCpp('ExampleGaussianRcpp.cpp')
```

- Note: Rcpp not necessary for any of this:
  - Feel free to skip over any line with `Rcpp::`
  - Add `#include <Eigen/Dense>` and `#include <unsupported/Eigen/AutoDiff>` for plain vanilla Eigen

ExampleGaussianRcpp.cpp I

```
// [[Rcpp::plugins("cpp11")]]
#include <cstddef>

// [[Rcpp::depends(BH)]]
#include <boost/math/constants/constants.hpp>

// [[Rcpp::depends(RcppEigen)]]
#include <RcppEigen.h>

#include <cmath>
```

```cpp
namespace model
{
    // 2 parameters: mu, sigma^2
    enum parameter : std::size_t { mu, s2 };
    constexpr std::size_t parameters_count = 2;
}
```

```cpp
// [[Rcpp::export]]
double l_t_cpp(double xt,
               const Eigen::Map<Eigen::VectorXd> parameters)
{
    const auto mu = parameters[model::parameter::mu];
    const auto s2 = parameters[model::parameter::s2];
    constexpr auto two_pi =
      boost::math::constants::two_pi<double>();
    using std::log;
    using std::pow;
    return -.5 * log(two_pi) - .5 * log(s2) -
            .5 * pow(xt - mu, 2) / s2;
}
```

```r
Rcpp::sourceCpp('ExampleGaussianRcppEigen.cpp')
```

ExampleGaussianRcppEigen.cpp I

```
// [[Rcpp::plugins("cpp11")]]
#include <cstddef>

// [[Rcpp::depends(BH)]]
#include <boost/math/constants/constants.hpp>

// [[Rcpp::depends(RcppEigen)]]
#include <RcppEigen.h>
#include <unsupported/Eigen/AutoDiff>

#include <cmath>
```

ExampleGaussianRcppEigen.cpp II

```cpp
namespace model
{
  // 2 parameters: mu, sigma^2
  enum parameter : std::size_t { mu, s2 };
  constexpr std::size_t parameters_count = 2;

  template <typename ScalarType>
  using parameter_vector =
    Eigen::Matrix<ScalarType, parameters_count, 1>;
}
```

```
// note: data `xt` -- double-precision number(s), just as before
// only the parameters adjusted to `ScalarType`
template <typename VectorType>
typename VectorType::Scalar
l_t_cpp_AD(double xt,
           const VectorType & parameters)
{
  using Scalar = typename VectorType::Scalar;
  const Scalar & mu = parameters[model::parameter::mu];
  const Scalar & s2 = parameters[model::parameter::s2];
  // note: `two_pi` is, as always, a double-precision constant
  constexpr auto two_pi =
    boost::math::constants::two_pi<double>();
  using std::log; using std::pow;
  return -.5 * log(two_pi) - .5 * log(s2) -
          .5 * pow(xt - mu, 2) / s2;
}
```

```
// wrapper over `l_t_cpp_AD`
// (simply dispatching to the existing implementation)
//
// (Rcpp-exportable explicit template instantiations
//  would render this unnecessary)

// [[Rcpp::export]]
double l_t_value_cpp(double xt,
                     const Eigen::Map<Eigen::VectorXd> parameters)
{
  return l_t_cpp_AD(xt, parameters);
}
```

```
// objective function together with its gradient
// `xt`: input (data)
// `parameters_input`: input (model parameters)
// `gradient_output`: output (computed gradient)
// returns: computed objective function value
// [[Rcpp::export]]
double l_t_value_gradient_cpp
(
  double xt,
  const Eigen::Map<Eigen::VectorXd> parameters_input,
  Eigen::Map<Eigen::VectorXd> gradient_output
)
{
  using parameter_vector = model::parameter_vector<double>;
  using AD = Eigen::AutoDiffScalar<parameter_vector>;
  using VectorAD = model::parameter_vector<AD>;
```

```
  parameter_vector parameters = parameters_input;
  VectorAD parameters_AD = parameters.cast<AD>();

  constexpr auto P = model::parameters_count;
  for (std::size_t p = 0; p != P; ++p)
  {
    // forward mode:
    // active differentiation variable `p`
    // (one parameter at a time)
    parameters_AD(p).derivatives() = parameter_vector::Unit(p);
  }

  const AD & loglikelihood_result = l_t_cpp_AD(xt, parameters_AD);
  gradient_output = loglikelihood_result.derivatives();
  return loglikelihood_result.value();
}
```

Data Parallel Objective Function I

```
// First: make the data parallelism (DP) explicit
// [[Rcpp::export]]
Eigen::VectorXd l_t_value_cppDP
(
  const Eigen::Map<Eigen::VectorXd> xs,
  const Eigen::Map<Eigen::VectorXd> parameters
)
{
  const std::size_t sample_size = xs.size();
  Eigen::VectorXd result(sample_size);
  for (std::size_t t = 0; t != sample_size; ++t)
  {
    result[t] = l_t_cpp_AD(xs[t], parameters);
  }
  return result;
}
```

Data Parallel Objective Function II

```cpp
// Second: Parallelize using OpenMP
// [[Rcpp::export]]
Eigen::VectorXd l_t_value_cppDP_OMP
(
  const Eigen::Map<Eigen::VectorXd> xs,
  const Eigen::Map<Eigen::VectorXd> parameters
)
{
  const std::size_t sample_size = xs.size();
  Eigen::VectorXd result(sample_size);
  #pragma omp parallel for default(none) shared(result)
  for (std::size_t t = 0; t < sample_size; ++t)
  {
    result[t] = l_t_cpp_AD(xs[t], parameters);
  }
  return result;
}
```

Data Parallel Objective Function Performance I

```
> require("microbenchmark")

> microbenchmark(
+   l_t(xs, fixed_parameters),
+   l_t_value_cppDP(xs, fixed_parameters),
+   l_t_value_cppDP_OMP(xs, fixed_parameters)
+ )
Unit: microseconds
                                      expr  median neval
     l_t_value_cppDP(xs, fixed_parameters) 458.618   100
 l_t_value_cppDP_OMP(xs, fixed_parameters) 213.526   100
```

Note: changing `l_t_cpp_AD` to `l_t_cpp_AD_DP` (analogously) can also help, but we're not done, yet...

```
// [[Rcpp::export]]
Eigen::VectorXd l_t_value_gradient_cppDP
(
  const Eigen::Map<Eigen::VectorXd> xs,
  const Eigen::Map<Eigen::VectorXd> parameters_input,
  Eigen::Map<Eigen::MatrixXd> gradient_output
)
{
  const std::size_t sample_size = xs.size();
  Eigen::VectorXd result(sample_size);
  for (std::size_t t = 0; t != sample_size; ++t)
  {
    using parameter_vector = model::parameter_vector<double>;
    using AD = Eigen::AutoDiffScalar<parameter_vector>;
    using VectorAD = model::parameter_vector<AD>;
```

```
    parameter_vector parameters = parameters_input;
    VectorAD parameters_AD = parameters.cast<AD>();

    constexpr auto P = model::parameters_count;
    for (std::size_t p = 0; p != P; ++p)
    {
      parameters_AD(p).derivatives() = parameter_vector::Unit(p);
    }
    const AD & loglikelihood_result = l_t_cpp_AD(xs[t],
                                                 parameters_AD);
    gradient_output.row(t) = loglikelihood_result.derivatives();
    result[t] = loglikelihood_result.value();
  }
  return result;
}
```

Data Parallel Gradient III

```
// [[Rcpp::export]]
Eigen::VectorXd l_t_value_gradient_cppDP_OMP
(
  const Eigen::Map<Eigen::VectorXd> xs,
  const Eigen::Map<Eigen::VectorXd> parameters_input,
  Eigen::Map<Eigen::MatrixXd> gradient_output
)
{
  const std::size_t sample_size = xs.size();
  Eigen::VectorXd result(sample_size);
  #pragma omp parallel for default(none) \
          shared(result, gradient_output)
  for (std::size_t t = 0; t < sample_size; ++t)
  {
    using parameter_vector = model::parameter_vector<double>;
    using AD = Eigen::AutoDiffScalar<parameter_vector>;
    using VectorAD = model::parameter_vector<AD>;
```

```
    parameter_vector parameters = parameters_input;
    VectorAD parameters_AD = parameters.cast<AD>();

    constexpr auto P = model::parameters_count;
    for (std::size_t p = 0; p != P; ++p)
    {
      parameters_AD(p).derivatives() = parameter_vector::Unit(p);
    }
    const AD & loglikelihood_result = l_t_cpp_AD(xs[t],
                                                 parameters_AD);
    gradient_output.row(t) = loglikelihood_result.derivatives();
    result[t] = loglikelihood_result.value();
  }
  return result;
}
```

Data Parallel Gradient Performance I

```
microbenchmark(
  l_t_value_gradient_cppDP(xs, fixed_parameters, gradient),
  l_t_value_gradient_cppDP_OMP(xs, fixed_parameters, gradient)
)
Unit: microseconds
                        expr   median neval
    l_t_value_gradient_cppDP 631.3375   100
l_t_value_gradient_cppDP_OMP 258.6945   100
```

Worth noting:

- speed-up over naive serial version thanks to OpenMP

Worth noting:

- speed-up over naive serial version thanks to OpenMP
- we can do better: truly data-parallel (non-naive) objective function implementation `l_t_cpp_AD_DP`, reverse mode AD

Worth noting:

- speed-up over naive serial version thanks to OpenMP
- we can do better: truly data-parallel (non-naive) objective function implementation `l_t_cpp_AD_DP`, reverse mode AD
- left as an exercise for the audience! :-)

Caveats

- Source code

Caveats

- Source code
  - Recall: AD requires the access to the source code

Caveats

- Source code
  - Recall: AD requires the access to the source code
  - We don't always have it: Consider closed source software (proprietary, third party)

Caveats

- Source code
  - Recall: AD requires the access to the source code
  - We don't always have it: Consider closed source software (proprietary, third party)
  - Another challenge: Source code spanning multiple programming languages

- Source code
  - Recall: AD requires the access to the source code
  - We don't always have it: Consider closed source software (proprietary, third party)
  - Another challenge: Source code spanning multiple programming languages
- Reverse mode — memory requirements

Caveats

- Source code
  - Recall: AD requires the access to the source code
  - We don't always have it: Consider closed source software (proprietary, third party)
  - Another challenge: Source code spanning multiple programming languages
- Reverse mode — memory requirements
- Perverse (but valid) code:

```
if (x == 1.0) then y = 0.0 else y = x - 1.0
```

- Source code
  - Recall: AD requires the access to the source code
  - We don't always have it: Consider closed source software (proprietary, third party)
  - Another challenge: Source code spanning multiple programming languages
- Reverse mode — memory requirements
- Perverse (but valid) code:

```
if (x == 1.0) then y = 0.0 else y = x - 1.0
```

  - derivative at $x = 1$?

Resources

http://www.autodiff.org/

http://en.wikipedia.org/wiki/Automatic_differentiation

Resources I

A Gentle Introduction to Backpropagation
http://numericinsight.blogspot.com/2014/07/a-gentle-introduction-to-backpropagation.html
http://www.numericinsight.com/uploads/A_Gentle_Introduction_to_B

A Multi-Core Benchmark Used to Improve Algorithmic Differentiation
http://www.seanet.com/~bradbell/cppad_thread.htm

A Multi-Core Algorithmic Differentiation Benchmarking System — Brad Bell
https://vimeo.com/39008544

Resources II

A Step by Step Backpropagation Example
http://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

Adjoint Methods in Computational Finance
http://www.hpcfinance.eu/sites/www.hpcfinance.eu/files/Uwe%20Nau
http://www.nag.com/Market/seminars/Uwe_AD_Slides_July13.pdf

Adjoints and Automatic (Algorithmic) Differentiation in Computational Finance
http://arxiv.org/abs/1107.1831v1

Algorithmic Differentiation in More Depth
http://www.nag.com/pss/ad-in-more-depth

Algorithmic Differentiation of a GPU Accelerated Application
http://www.nag.co.uk/Market/events/jdt-hpc-new-thinking-in-finance-presentation.pdf

Automatic Differentiation and QuantLib
https://quantlib.wordpress.com/tag/automatic-differentiation/

Automatic differentiation in deep learning by Shawn Tan
https://cdn.rawgit.com/shawntan/presentations/master/Deep Learning-Copy1.slides.html

Calculus on Computational Graphs: Backpropagation
https://colah.github.io/posts/2015-08-Backprop/

Resources IV

Computing derivatives for nonlinear optimization: Forward mode automatic differentiation
http://nbviewer.ipython.org/github/joehuchette/OR-software-tools-2015/blob/master/6-nonlinear-opt/Nonlinear-DualNumbers.ipynb

Introduction to Automatic Differentiation
http://alexey.radul.name/ideas/2013/introduction-to-automatic-differentiation/

Jarrett Revels: Automatic differentiation
https://www.youtube.com/watch?v=PrXUl0sanro

Neural Networks (Part I) – Understanding the Mathematics behind backpropagation https://biasvariance.wordpress.com/2015/08/18/neural-networks-understanding-the-math-behind-backpropagation-part-i/

- http://www.johndcook.com/blog/2009/04/06/anatomy-of-a-floating-point-number/

Everything by Bruce Dawson:

- https://randomascii.wordpress.com/category/floating-point/

In particular:

- `https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/`
- `https://randomascii.wordpress.com/2012/04/05/floating-point-complexities/`
- `https://randomascii.wordpress.com/2013/01/03/top-eight-entertaining-blog-facts-for-2012/`

https://en.wikipedia.org/wiki/Automatic_differentiation#Software

ADNumber, Adept, ADOL-C, CppAD, Eigen (Auto Diff module)

CasADi
https://github.com/casadi/casadi/wiki
CasADi is a symbolic framework for algorithmic (a.k.a. automatic) differentiation and numeric optimization.

CppAD
https://github.com/coin-or/CppAD/

Dali
https://github.com/JonathanRaiman/Dali
An automatic differentiation library that uses reverse-mode differentation (backpropagation) to differentiate recurrent neural networks, or most mathematical expressions through control flow, while loops, recursion.

Open Porous Media Automatic Differentiation Library
https://github.com/OPM/opm-autodiff
Utilities for automatic differentiation and simulators based on AD.

The Stan Math Library (`stan::math`: includes a C++ reverse-mode automatic differentiation library)
https://github.com/stan-dev/math

References

References I

Bollerslev, Tim, and Robert Engle. 1993. "Common Persistence in Conditional Variances." *Econometrica* 61 (1). The Econometric Society: pp. 167–86. http://www.jstor.org/stable/2951782.

Giles, Michael, and Paul Glasserman. 2006. "Smoking Adjoints: Fast Monte Carlo Greeks." *Risk* 19: 88–92.

Griewank, Andreas. 2012. "Who Invented the Reverse Mode of Differentiation?" *Optimization Stories* Documenta Matematica, Extra Volume ISMP (2012): 389–400.

Griewank, Andreas, and Andrea Walther. 2008. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. 2nd ed. Other Titles in Applied Mathematics 105.

Philadelphia, PA: SIAM.
`http://www.ec-securehost.com/SIAM/OT105.html`.

Homescu, C. 2011. "Adjoints and Automatic (Algorithmic) Differentiation in Computational Finance." *ArXiv E-Prints*, July. `http://arxiv.org/abs/1107.1831`.

Naumann, Uwe. 2012. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*. Computer software, Environments, and Tools 24. Philadelphia, PA: SIAM. `http://www.ec-securehost.com/SIAM/SE24.html`.

Nocedal, Jorge, and Stephen J. Wright. 2006. *Numerical Optimization*. 2. ed. Springer: New York, NY.

Thank You!
Questions?