# Algorithmic differentiation

TEMPO Course on Numerical Optimal Control, 4-13 August 2014, Freiburg im Breisgau, Germany

Joel Andersson

5 August 2014

1. Calculating derivatives
2. Algorithmic differentiation
3. Jacobians and Hessians
4. Software
5. Summary

# Outline


1 Calculating derivatives

2 Algorithmic differentiation

3 Jacobians and Hessians

4 Software

5 Summary

Methods for calculating derivatives

Methods for calculating derivatives

- By hand

Methods for calculating derivatives

- By hand $\leftarrow$ **Time consuming & error prone!**

Methods for calculating derivatives

- By hand $\leftarrow$ **Time consuming & error prone!**
- Symbolic differentiation

Methods for calculating derivatives

- By hand $\leftarrow$ **Time consuming & error prone!**
- Symbolic differentiation
- Finite difference approximation

Methods for calculating derivatives

- By hand $\leftarrow$ **Time consuming & error prone!**
- Symbolic differentiation
- Finite difference approximation
- Complex step differentiation ("Imaginary trick")

Methods for calculating derivatives

- By hand $\leftarrow$ **Time consuming & error prone!**
- Symbolic differentiation
- Finite difference approximation
- Complex step differentiation ("Imaginary trick")
- Automatic differentiation (AD)

## Symbolic differentiation

We can obtain an expression of the derivatives we need with:

- Mathematica
- Maple
- Symbolic Toolbox for MATLAB
- SymPy
- ...

## Symbolic differentiation

We can obtain an expression of the derivatives we need with:

- Mathematica
- Maple
- Symbolic Toolbox for MATLAB
- SymPy
- ...

**Often this results in a very long code which is expensive to evaluate.**

## Finite differences

Consider a function $f : \mathbb{R}^{n_x} \rightarrow \mathbb{R}^{n_y}$ with Jacobian $J(x) = \frac{\partial f}{\partial x}$

$$J(x)\,\hat{x} \approx \frac{f(x + t\,\hat{x}) - f(x)}{t}$$

Pros and cons:

## Finite differences

Consider a function $f : \mathbb{R}^{n_x} \rightarrow \mathbb{R}^{n_y}$ with Jacobian $J(x) = \dfrac{\partial f}{\partial x}$

$$J(x)\,\hat{x} \approx \frac{f(x + t\,\hat{x}) - f(x)}{t}$$

Pros and cons:

+ Really easy to implement

## Finite differences

Consider a function $f : \mathbb{R}^{n_x} \to \mathbb{R}^{n_y}$ with Jacobian $J(x) = \dfrac{\partial f}{\partial x}$

$$J(x)\,\hat{x} \approx \frac{f(x + t\,\hat{x}) - f(x)}{t}$$

Pros and cons:

+ Really easy to implement
+ Relatively fast

## Finite differences

Consider a function $f : \mathbb{R}^{n_x} \to \mathbb{R}^{n_y}$ with Jacobian $J(x) = \frac{\partial f}{\partial x}$

$$J(x)\,\hat{x} \approx \frac{f(x + t\,\hat{x}) - f(x)}{t}$$

Pros and cons:

+ Really easy to implement
+ Relatively fast
− Poor accuracy

## Finite differences

Consider a function $f : \mathbb{R}^{n_x} \to \mathbb{R}^{n_y}$ with Jacobian $J(x) = \frac{\partial f}{\partial x}$

$$J(x)\,\hat{x} \approx \frac{f(x + t\,\hat{x}) - f(x)}{t}$$

Pros and cons:

+ Really easy to implement
+ Relatively fast
− Poor accuracy
  - Small $t \Rightarrow$ *cancellation errors*

## Finite differences

Consider a function $f : \mathbb{R}^{n_x} \rightarrow \mathbb{R}^{n_y}$ with Jacobian $J(x) = \dfrac{\partial f}{\partial x}$

$$J(x)\,\hat{x} \approx \frac{f(x + t\,\hat{x}) - f(x)}{t}$$

Pros and cons:

+ Really easy to implement
+ Relatively fast
− Poor accuracy
  - Small $t \Rightarrow$ *cancellation errors*
  - Large $t \Rightarrow$ *approximation errors*

## Finite differences

Consider a function $f : \mathbb{R}^{n_x} \rightarrow \mathbb{R}^{n_y}$ with Jacobian $J(x) = \frac{\partial f}{\partial x}$

$$J(x)\,\hat{x} \approx \frac{f(x + t\,\hat{x}) - f(x)}{t}$$

Pros and cons:

+ Really easy to implement
+ Relatively fast
− Poor accuracy
  - Small $t \Rightarrow$ *cancellation errors*
  - Large $t \Rightarrow$ *approximation errors*
  - Rule of thumb: $t \approx \sqrt{\epsilon}$, where $\epsilon$ is $f$ accuracy, typically $\approx 10^{-16}$

## Finite differences

Consider a function $f : \mathbb{R}^{n_x} \to \mathbb{R}^{n_y}$ with Jacobian $J(x) = \dfrac{\partial f}{\partial x}$

$$J(x)\,\hat{x} \approx \frac{f(x + t\,\hat{x}) - f(x)}{t}$$

Pros and cons:

+ Really easy to implement
+ Relatively fast
− Poor accuracy
  - Small $t \Rightarrow$ *cancellation errors*
  - Large $t \Rightarrow$ *approximation errors*
  - Rule of thumb: $t \approx \sqrt{\epsilon}$, where $\epsilon$ is $f$ accuracy, typically $\approx 10^{-16}$
− No efficient way to calculate $\hat{y}^\mathrm{T}\,J(x)$

## Complex step differentiation ("Imaginary trick")

Finite differences with imaginary perturbation:

$$J(x)\,\hat{x} \approx \Re\left(\frac{f(x + i\,t\,\hat{x}) - f(x)}{i\,t}\right)$$

## Complex step differentiation ("Imaginary trick")

Finite differences with imaginary perturbation:

$$J(x)\,\hat{x} \approx \Re\left(\frac{f(x + i\,t\,\hat{x}) - f(x)}{i\,t}\right) = -\frac{\Im(f(x + i\,t\,\hat{x}))}{t}$$

Pros and cons:

## Complex step differentiation ("Imaginary trick")

Finite differences with imaginary perturbation:

$$J(x)\,\hat{x} \approx \Re\left(\frac{f(x + i\,t\,\hat{x}) - f(x)}{i\,t}\right) = -\frac{\Im(f(x + i\,t\,\hat{x}))}{t}$$

Pros and cons:

+ Really easy to implement in MATLAB, Python

## Complex step differentiation ("Imaginary trick")

Finite differences with imaginary perturbation:

$$J(x)\,\hat{x} \approx \Re\left(\frac{f(x + i\,t\,\hat{x}) - f(x)}{i\,t}\right) = -\frac{\Im(f(x + i\,t\,\hat{x}))}{t}$$

Pros and cons:

+ Really easy to implement in MATLAB, Python
+ Relatively fast

## Complex step differentiation ("Imaginary trick")

Finite differences with imaginary perturbation:

$$J(x)\,\hat{x} \approx \Re\left(\frac{f(x + i\,t\,\hat{x}) - f(x)}{i\,t}\right) = -\frac{\Im(f(x + i\,t\,\hat{x}))}{t}$$

Pros and cons:

+ Really easy to implement in MATLAB, Python
+ Relatively fast
+ Good accuracy (no cancellation errors $\Rightarrow$ $t$ small)

## Complex step differentiation (“Imaginary trick”)

Finite differences with imaginary perturbation:

$$J(x)\,\hat{x} \approx \Re\left(\frac{f(x + i\,t\,\hat{x}) - f(x)}{i\,t}\right) = -\frac{\Im(f(x + i\,t\,\hat{x}))}{t}$$

Pros and cons:

+ Really easy to implement in MATLAB, Python
+ Relatively fast
+ Good accuracy (no cancellation errors ⇒ $t$ small)
− Error prone, e.g. `x^y`

## Complex step differentiation ("Imaginary trick")

Finite differences with imaginary perturbation:

$$J(x)\,\hat{x} \approx \Re\left(\frac{f(x + i\,t\,\hat{x}) - f(x)}{i\,t}\right) = -\frac{\Im(f(x + i\,t\,\hat{x}))}{t}$$

Pros and cons:

+ Really easy to implement in MATLAB, Python
+ Relatively fast
+ Good accuracy (no cancellation errors $\Rightarrow$ $t$ small)
− Error prone, e.g. `x^y`
− Restricted

## Complex step differentiation ("Imaginary trick")

Finite differences with imaginary perturbation:

$$J(x)\,\hat{x} \approx \Re\left(\frac{f(x + i\,t\,\hat{x}) - f(x)}{i\,t}\right) = -\frac{\Im(f(x + i\,t\,\hat{x}))}{t}$$

Pros and cons:

+ Really easy to implement in MATLAB, Python
+ Relatively fast
+ Good accuracy (no cancellation errors $\Rightarrow$ $t$ small)
- Error prone, e.g. `x^y`
- Restricted
- No efficient way to calculate $\hat{y}^\mathsf{T}\,J(x)$

# Outline


1 Calculating derivatives

2 Algorithmic differentiation

3 Jacobians and Hessians

4 Software

5 Summary

## Decomposable function: $y = F(x)$

## Decomposable function: $y = F(x)$

- $F : \mathbb{R}^{n_0} \to \mathbb{R}^{n_K}$ sufficiently smooth

## Decomposable function: $y = F(x)$

- $F : \mathbb{R}^{n_0} \to \mathbb{R}^{n_K}$ sufficiently smooth
- Decompose into "atomic operations" which we know how to differentiate:

$z_0 \leftarrow x$
**for** $k = 1, \ldots, K$ **do**
  $z_k \leftarrow f_k\left(\{z_i\}_{i \in \mathcal{I}_k}\right)$
**end for**
$y \leftarrow z_K$
**return** $y$

## Decomposable function: $y = F(x)$

- $F : \mathbb{R}^{n_0} \to \mathbb{R}^{n_K}$ sufficiently smooth
- Decompose into "atomic operations" which we know how to differentiate:

$z_0 \leftarrow x$
**for** $k = 1, \ldots, K$ **do**
    $z_k \leftarrow f_k\left(\{z_i\}_{i \in \mathcal{I}_k}\right)$
**end for**
$y \leftarrow z_K$
**return** $y$

*Such a decomposition is always available if $F$ written as a computer program!*

## Decomposable function: $y = F(x)$

- $F : \mathbb{R}^{n_0} \to \mathbb{R}^{n_K}$ sufficiently smooth
- Decompose into "atomic operations" which we know how to differentiate:

$z_0 \leftarrow x$
**for** $k = 1, \ldots, K$ **do**
  $z_k \leftarrow f_k\left(\{z_i\}_{i \in \mathcal{I}_k}\right)$
**end for**
$y \leftarrow z_K$
**return** $y$

*Such a decomposition is always available if $F$ written as a computer program!*

## Example

$$y = \sin(\sqrt{x})$$

$z_0 \leftarrow x$
$z_1 = \sqrt{z_0}$
$z_2 = \sin z_1$
$y \leftarrow z_2$
**return** $y$

- Decomposition can be with simple scalar operations . . .

- Decomposition can be with simple scalar operations ...
  - $x + y$, $x * y$, $\sin(x)$, $x^y$

- Decomposition can be with simple scalar operations . . .
  - $x + y$, $x * y$, $\sin(x)$, $x^y$
  - Usual case in software

- Decomposition can be with simple scalar operations ...
  - $x + y$, $x * y$, $\sin(x)$, $x^y$
  - Usual case in software
- ... or with more general operations

- Decomposition can be with simple scalar operations ...
  - $x + y$, $x * y$, $\sin(x)$, $x^y$
  - Usual case in software
- ...or with more general operations
  - $x^{\mathsf{T}}$, $x[i] = y$, $XY$, $e^X$

- Decomposition can be with simple scalar operations ...
  - $x + y$, $x * y$, $\sin(x)$, $x^y$
  - Usual case in software
- ...or with more general operations
  - $x^\mathsf{T}$, $x[i] = y$, $XY$, $e^X$
  - E.g. gradient of $\det(X)$:

- Decomposition can be with simple scalar operations ...
  - $x + y$, $x * y$, $\sin(x)$, $x^y$
  - Usual case in software
- ...or with more general operations
  - $x^\mathsf{T}$, $x[i] = y$, $XY$, $e^X$
  - E.g. gradient of $\det(X)$: $\det(X)\, X^{-\mathsf{T}}$

- Decomposition can be with simple scalar operations ...
  - $x + y$, $x * y$, $\sin(x)$, $x^y$
  - Usual case in software
- ...or with more general operations
  - $x^\intercal$, $x[i] = y$, $XY$, $e^X$
  - E.g. gradient of $\det(X)$: $\det(X)\, X^{-\intercal}$
  - In e.g. CasADi

- Decomposition can be with simple scalar operations . . .
  - $x + y$, $x * y$, $\sin(x)$, $x^y$
  - Usual case in software
- . . . or with more general operations
  - $x^\mathsf{T}$, $x[i] = y$, $XY$, $e^X$
  - E.g. gradient of $\det(X)$: $\det(X)\, X^{-\mathsf{T}}$
  - In e.g. CasADi
- Derivative propagation rules exist for

- Decomposition can be with simple scalar operations ...
  - $x + y$, $x * y$, $\sin(x)$, $x^y$
  - Usual case in software
- ...or with more general operations
  - $x^\mathsf{T}$, $x[i] = y$, $XY$, $e^X$
  - E.g. gradient of $\det(X)$: $\det(X)\, X^{-\mathsf{T}}$
  - In e.g. CasADi
- Derivative propagation rules exist for
  - ODE/DAE integrators, "sensitivity analysis"

- Decomposition can be with simple scalar operations ...
  - $x + y$, $x * y$, $\sin(x)$, $x^y$
  - Usual case in software
- ...or with more general operations
  - $x^\mathsf{T}$, $x[i] = y$, $XY$, $e^X$
  - E.g. gradient of $\det(X)$: $\det(X)\, X^{-\mathsf{T}}$
  - In e.g. CasADi
- Derivative propagation rules exist for
  - ODE/DAE integrators, "sensitivity analysis"
  - Linear and nonlinear systems of equations

# Differentiate the algorithm!

# Differentiate the algorithm!

$$
\begin{array}{l}
z_0 \leftarrow x \\
\textbf{for } k = 1, \ldots, K \textbf{ do} \\
\quad z_k \leftarrow f_k\left(\{z_i\}_{i \in \mathcal{I}_k}\right) \\
\textbf{end for} \\
y \leftarrow z_K \\
\textbf{return } y
\end{array}
$$

$$\Rightarrow$$

$$
\begin{array}{l}
z_0 \leftarrow x \\
\dfrac{dz_0}{dx} \leftarrow I \\
\textbf{for } k = 1, \ldots, K \textbf{ do} \\
\quad z_k \leftarrow f_k\left(\{z_i\}_{i \in \mathcal{I}_k}\right) \\
\quad \dfrac{dz_k}{dx} \leftarrow \sum_{i \in \mathcal{I}_k} \dfrac{\partial f_k}{\partial z_i}\left(\{z_i\}_{i \in \mathcal{I}_k}\right) \dfrac{dz_i}{dx} \\
\textbf{end for} \\
y \leftarrow z_K \\
J \leftarrow \dfrac{dz_K}{dx} \\
\textbf{return } y, J
\end{array}
$$

# Differentiate the algorithm!

$z_0 \leftarrow x$
**for** $k = 1, \ldots, K$ **do**
  $z_k \leftarrow f_k\left(\{z_i\}_{i \in \mathcal{I}_k}\right)$
**end for**
$y \leftarrow z_K$
**return** $y$

$\Longrightarrow$

$z_0 \leftarrow x$
$\dfrac{dz_0}{dx} \leftarrow I$
**for** $k = 1, \ldots, K$ **do**
  $z_k \leftarrow f_k\left(\{z_i\}_{i \in \mathcal{I}_k}\right)$
  $\dfrac{dz_k}{dx} \leftarrow \sum_{i \in \mathcal{I}_k} \dfrac{\partial f_k}{\partial z_i}\left(\{z_i\}_{i \in \mathcal{I}_k}\right) \dfrac{dz_i}{dx}$
**end for**
$y \leftarrow z_K$
$J \leftarrow \dfrac{dz_K}{dx}$
**return** $y, J$

Write as a system of linear equations:

$$\frac{dz}{dx} = B + L\,\frac{dz}{dx}, \qquad J = A^{\mathsf{T}}\,\frac{dz}{dx},$$

Write as a system of linear equations:

$$\frac{dz}{dx} = B + L\,\frac{dz}{dx}, \qquad J = A^{\mathsf{T}}\,\frac{dz}{dx},$$

Write as a system of linear equations:

$$\frac{dz}{dx} = B + L\,\frac{dz}{dx}, \qquad J = A^{\mathsf{T}}\,\frac{dz}{dx},$$

with

$$z = \begin{pmatrix} z_0 \\ z_1 \\ \vdots \\ z_K \end{pmatrix}, \quad A = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ I \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} I \\ 0 \\ \vdots \\ 0 \end{pmatrix},$$

with $I$ and $0$ of appropriate dimensions, as well as the *extended Jacobian*,

$$L = \begin{pmatrix} 0 & \dots & \dots & 0 \\ \frac{\partial f_1}{\partial z_0} & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ \frac{\partial f_K}{\partial z_0} & \dots & \frac{\partial f_K}{\partial z_{K-1}} & 0 \end{pmatrix},$$

Write as a system of linear equations:

$$\frac{dz}{dx} = B + L\,\frac{dz}{dx}, \qquad J = A^{\mathsf{T}}\,\frac{dz}{dx},$$

with

$$z = \begin{pmatrix} z_0 \\ z_1 \\ \vdots \\ z_K \end{pmatrix}, \quad A = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ I \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} I \\ 0 \\ \vdots \\ 0 \end{pmatrix},$$

with $I$ and 0 of appropriate dimensions, as well as the *extended Jacobian*,

$$L = \begin{pmatrix} 0 & \dots & \dots & 0 \\ \frac{\partial f_1}{\partial z_0} & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ \frac{\partial f_K}{\partial z_0} & \dots & \frac{\partial f_K}{\partial z_{K-1}} & 0 \end{pmatrix},$$

Since $I - L$ is invertible, we can solve for $J$:

$$J = A^{\mathsf{T}}\,(I - L)^{-1}\,B$$

- Have $J = A^{\mathsf{T}} \, (I - L)^{-1} \, B$

- Have $J = A^{\mathsf{T}} (I - L)^{-1} B$
- Multiply $J$ from the right:

- Have $J = A^{\mathsf{T}} (I - L)^{-1} B$
- Multiply $J$ from the right:
  - $\hat{y} := J \hat{x} = A^{\mathsf{T}} (I - L)^{-1} B \hat{x}$

- Have $J = A^{\mathsf{T}} \, (I - L)^{-1} \, B$
- Multiply $J$ from the right:
  - $\hat{y} := J \, \hat{x} = A^{\mathsf{T}} \, (I - L)^{-1} \, B \, \hat{x}$
  - Cheap with *forward substitution* of lower triangular $(I - L)$

- Have $J = A^{\mathsf{T}} \, (I - L)^{-1} \, B$
- Multiply $J$ from the right:
  - $\hat{y} := J \, \hat{x} = A^{\mathsf{T}} \, (I - L)^{-1} \, B \, \hat{x}$
  - Cheap with *forward substitution* of lower triangular $(I - L)$
  - No storage of $L$ needed

- Have $J = A^{\mathsf{T}} (I - L)^{-1} B$
- Multiply $J$ from the right:
  - $\hat{y} := J \hat{x} = A^{\mathsf{T}} (I - L)^{-1} B \hat{x}$
  - Cheap with *forward substitution* of lower triangular $(I - L)$
  - No storage of $L$ needed
  - Forward mode of AD

- Have $J = A^{\mathsf{T}} (I - L)^{-1} B$
- Multiply $J$ from the right:
  - $\hat{y} := J \hat{x} = A^{\mathsf{T}} (I - L)^{-1} B \hat{x}$
  - Cheap with *forward substitution* of lower triangular $(I - L)$
  - No storage of $L$ needed
  - Forward mode of AD
- Multiply $J$ from the left:

- Have $J = A^{\mathsf{T}} (I - L)^{-1} B$
- Multiply $J$ from the right:
  - $\hat{y} := J \hat{x} = A^{\mathsf{T}} (I - L)^{-1} B \hat{x}$
  - Cheap with *forward substitution* of lower triangular $(I - L)$
  - No storage of $L$ needed
  - Forward mode of AD
- Multiply $J$ from the left:
  - $\bar{x} := J^{\mathsf{T}} \bar{y} = B^{\mathsf{T}} (I - L)^{-\mathsf{T}} A \bar{y}$

- Have $J = A^\mathsf{T}\,(I - L)^{-1}\,B$
- Multiply $J$ from the right:
    - $\hat{y} := J\,\hat{x} = A^\mathsf{T}\,(I - L)^{-1}\,B\,\hat{x}$
    - Cheap with *forward substitution* of lower triangular $(I - L)$
    - No storage of $L$ needed
    - Forward mode of AD
- Multiply $J$ from the left:
    - $\bar{x} := J^\mathsf{T}\,\bar{y} = B^\mathsf{T}\,(I - L)^{-\mathsf{T}}\,A\,\bar{y}$
    - Cheap with *backward substitution* of upper triangular $(I - L)^\mathsf{T}$

- Have $J = A^{\mathsf{T}}\,(I - L)^{-1}\,B$
- Multiply $J$ from the right:
  - $\hat{y} := J\,\hat{x} = A^{\mathsf{T}}\,(I - L)^{-1}\,B\,\hat{x}$
  - Cheap with *forward substitution* of lower triangular $(I - L)$
  - No storage of $L$ needed
  - Forward mode of AD
- Multiply $J$ from the left:
  - $\bar{x} := J^{\mathsf{T}}\,\bar{y} = B^{\mathsf{T}}\,(I - L)^{-\mathsf{T}}\,A\,\bar{y}$
  - Cheap with *backward substitution* of upper triangular $(I - L)^{\mathsf{T}}$
  - Storage of $L$ needed

- Have $J = A^{\mathsf{T}} (I - L)^{-1} B$
- Multiply $J$ from the right:
  - $\hat{y} := J \hat{x} = A^{\mathsf{T}} (I - L)^{-1} B \hat{x}$
  - Cheap with *forward substitution* of lower triangular $(I - L)$
  - No storage of $L$ needed
  - Forward mode of AD
- Multiply $J$ from the left:
  - $\bar{x} := J^{\mathsf{T}} \bar{y} = B^{\mathsf{T}} (I - L)^{-\mathsf{T}} A \bar{y}$
  - Cheap with *backward substitution* of upper triangular $(I - L)^{\mathsf{T}}$
  - Storage of $L$ needed
  - Reverse mode of AD

## Forward mode of AD

## Forward mode of AD

- Calculate Jacobian-times-vector product cheaply

## Forward mode of AD

- Calculate Jacobian-times-vector product cheaply
- Computational cost: $\approx$ cost of evaluating $F$

## Forward mode of AD

- Calculate Jacobian-times-vector product cheaply
- Computational cost: $\approx$ cost of evaluating $F$
- Small memory requirements

## Forward mode of AD

- Calculate Jacobian-times-vector product cheaply
- Computational cost: $\approx$ cost of evaluating $F$
- Small memory requirements

## Reverse mode of AD

## Forward mode of AD

- Calculate Jacobian-times-vector product cheaply
- Computational cost: $\approx$ cost of evaluating $F$
- Small memory requirements

## Reverse mode of AD

- Calculate vector-times-Jacobian product cheaply

## Forward mode of AD

- Calculate Jacobian-times-vector product cheaply
- Computational cost: $\approx$ cost of evaluating $F$
- Small memory requirements

## Reverse mode of AD

- Calculate vector-times-Jacobian product cheaply
- In particular: Gradient of scalar-valued $f$ cheap!

## Forward mode of AD

- Calculate Jacobian-times-vector product cheaply
- Computational cost: $\approx$ cost of evaluating $F$
- Small memory requirements

## Reverse mode of AD

- Calculate vector-times-Jacobian product cheaply
- In particular: Gradient of scalar-valued $f$ cheap!
- Computational cost: $\approx$ cost of evaluating $F$

## Forward mode of AD

- Calculate Jacobian-times-vector product cheaply
- Computational cost: $\approx$ cost of evaluating $F$
- Small memory requirements

## Reverse mode of AD

- Calculate vector-times-Jacobian product cheaply
- In particular: Gradient of scalar-valued $f$ cheap!
- Computational cost: $\approx$ cost of evaluating $F$
- Intermediate operations (or their linearization) must be stored

## Forward mode of AD

- Calculate Jacobian-times-vector product cheaply
- Computational cost: $\approx$ cost of evaluating $F$
- Small memory requirements

## Reverse mode of AD

- Calculate vector-times-Jacobian product cheaply
- In particular: Gradient of scalar-valued $f$ cheap!
- Computational cost: $\approx$ cost of evaluating $F$
- Intermediate operations (or their linearization) must be stored
- Can trade storage for extra computation ("checkpointing")

# Outline


1 Calculating derivatives

2 Algorithmic differentiation

3 Jacobians and Hessians

4 Software

5 Summary

## Calculating complete Jacobians and Hessians

## Calculating complete Jacobians and Hessians

- Jacobians can be calculated by multiplying with $n_{\text{col}}$ vectors from the right or $n_{\text{row}}$ vectors from the left

## Calculating complete Jacobians and Hessians

- Jacobians can be calculated by multiplying with $n_{\text{col}}$ vectors from the right or $n_{\text{row}}$ vectors from the left
- Worst-case: $\approx \min(n_{\text{row}}, n_{\text{col}})$ times cost of evaluating $F$

## Calculating complete Jacobians and Hessians

- Jacobians can be calculated by multiplying with $n_{\text{col}}$ vectors from the right or $n_{\text{row}}$ vectors from the left
- Worst-case: $\approx \min(n_{\text{row}}, n_{\text{col}})$ times cost of evaluating $F$
- *Much cheaper* if $J$ is sparse, e.g. banded

## Calculating complete Jacobians and Hessians

- Jacobians can be calculated by multiplying with $n_{\text{col}}$ vectors from the right or $n_{\text{row}}$ vectors from the left
- Worst-case: $\approx \min(n_{\text{row}}, n_{\text{col}})$ times cost of evaluating $F$
- *Much cheaper* if $J$ is sparse, e.g. banded
- Requires prior knowledge of sparsity pattern (automation possible)

## Calculating complete Jacobians and Hessians

- Jacobians can be calculated by multiplying with $n_{\text{col}}$ vectors from the right or $n_{\text{row}}$ vectors from the left
- Worst-case: $\approx \min(n_{\text{row}}, n_{\text{col}})$ times cost of evaluating $F$
- *Much cheaper* if $J$ is sparse, e.g. banded
- Requires prior knowledge of sparsity pattern (automation possible)
- Hessians can be calculated as Jacobian-of-gradient

## Calculating complete Jacobians and Hessians

- Jacobians can be calculated by multiplying with $n_{\text{col}}$ vectors from the right or $n_{\text{row}}$ vectors from the left
- Worst-case: $\approx \min(n_{\text{row}}, n_{\text{col}})$ times cost of evaluating $F$
- *Much cheaper* if $J$ is sparse, e.g. banded
- Requires prior knowledge of sparsity pattern (automation possible)
- Hessians can be calculated as Jacobian-of-gradient
- Symmetry can be exploited

## Calculating complete Jacobians and Hessians

- Jacobians can be calculated by multiplying with $n_{\text{col}}$ vectors from the right or $n_{\text{row}}$ vectors from the left
- Worst-case: $\approx \min(n_{\text{row}}, n_{\text{col}})$ times cost of evaluating $F$
- *Much cheaper* if $J$ is sparse, e.g. banded
- Requires prior knowledge of sparsity pattern (automation possible)
- Hessians can be calculated as Jacobian-of-gradient
- Symmetry can be exploited
- *Much cheaper* if $H$ is sparse

# Outline


1 Calculating derivatives

2 Algorithmic differentiation

3 Jacobians and Hessians

4 Software

5 Summary

## Generic tools to differentiate "black-box" code

## Generic tools to differentiate "black-box" code

- Language-specific: `www.autodiff.org`

## Generic tools to differentiate “black-box” code

- Language-specific: `www.autodiff.org`
- ADOL-C, ADIC, CppAD for C/C++

## Generic tools to differentiate "black-box" code

- Language-specific: `www.autodiff.org`
- ADOL-C, ADIC, CppAD for C/C++
- ADIFOR, TAPENADE for FORTRAN

## Generic tools to differentiate "black-box" code

- Language-specific: `www.autodiff.org`
- ADOL-C, ADIC, CppAD for C/C++
- ADIFOR, TAPENADE for FORTRAN

## AD implemented inside other tools

## Generic tools to differentiate "black-box" code

- Language-specific: `www.autodiff.org`
- ADOL-C, ADIC, CppAD for C/C++
- ADIFOR, TAPENADE for FORTRAN

## AD implemented inside other tools

- CasADi

## Generic tools to differentiate "black-box" code

- Language-specific: `www.autodiff.org`
- ADOL-C, ADIC, CppAD for C/C++
- ADIFOR, TAPENADE for FORTRAN

## AD implemented inside other tools

- CasADi
- AMPL, GAMS: Algebraic modelling languages

# Outline


1 Calculating derivatives

2 Algorithmic differentiation

3 Jacobians and Hessians

4 Software

5 Summary

## Key points

## Key points

- Jacobian-times-vector products can be calculated cheaply

## Key points

- Jacobian-times-vector products can be calculated cheaply
  - Important special case: gradient of a scalar-valued function

## Key points

- Jacobian-times-vector products can be calculated cheaply
  - Important special case: gradient of a scalar-valued function
- Complete Jacobians and Hessians: depends on sparsity pattern.

## Key points

- Jacobian-times-vector products can be calculated cheaply
  - Important special case: gradient of a scalar-valued function
- Complete Jacobians and Hessians: depends on sparsity pattern.
  - Worse case: $\approx \min(n_{\text{row}}, n_{\text{col}})$ times cost of evaluating $F$

## Key points

- Jacobian-times-vector products can be calculated cheaply
  - Important special case: gradient of a scalar-valued function
- Complete Jacobians and Hessians: depends on sparsity pattern.
  - Worse case: $\approx \min(n_{\text{row}}, n_{\text{col}})$ times cost of evaluating $F$
- Good software exists

## Key points

- Jacobian-times-vector products can be calculated cheaply
  - Important special case: gradient of a scalar-valued function
- Complete Jacobians and Hessians: depends on sparsity pattern.
  - Worse case: $\approx \min(n_{\text{row}}, n_{\text{col}})$ times cost of evaluating $F$
- Good software exists

## Literature

Griewank & Walther, *Evaluating Derivatives* (2008)