Introduction to Automatic Differentiation for MATLAB (ADiMat)

Johannes Willkomm
johannes@johannes-willkomm.de

Advances in Computational Economics and Finance
2018-10-31

Outline

Begin with a formula

We start with a mathematical example

A polynomial

$$f(x) = \sum_{i=0}^{n} c_n x^n$$

Begin with a formula

We start with a mathematical example

A polynomial

$$f(x) = \sum_{i=0}^{n} c_n x^n$$

The derivative

$$\mathrm{d}f(x) = \sum_{i=1}^{n} c_n n x^{n-1} \mathrm{d}x$$

- This is symbolic differentiation.

A mechanical process

- Differentiation is a fairly mechanical process
- It lends itself to automatization
- Differentiation can be done by computer programs
- Differentiation can also be done *on* computer programs

Begin with a program

Implement f in MATLAB, in file `f.m`.

Example (Example code)

```
function r = f(x)
  global c
  xi = eye(size(x));
  r = 0;
  for i=1:length(c)
    r = r + c(i) * xi;
    xi = xi * x;
  end
```

Obtaining results

Run the program

```
global c
c = ones(1,5);
x = 2;
r = f(x)

r = 31
```

Compute derivatives

Finite differences

```
h = sqrt(eps);
df_fx = (f(x + h) - f(x - h)) ./ (2.*h)
```

```
df_fx =  49
```

Compute derivatives

Finite differences

```
h = sqrt(eps);
df_fx = (f(x + h) - f(x - h)) ./ (2.*h)
```

```
df_fx =  49
```

Analytical derivation

```
df_fx = polyval(polyder(c), x)
```

```
df_fx =  49
```

ADiMat

Differentiate f in forward mode of automatic differentiation

```
admDiffFor(@f, 1, x)

Differentiated function g_f does not exist.
Differentiating function f in forward mode (FM) to
 produce g_f...
Differentiation took 0.0892689 s.
ans =  49
```

- The message shows that ADiMat performs *source transformation*
- A *transpiler* differentiates the program code → file `g_f.m`
- Let's not look at the generated code, but try it ourselves

Code differentiation

Example (Differentiated function signature)

1. Change function name, add derivative inputs and outputs

```
function [d_r, r] = d_f(d_x, x)
  global c
  xi = eye(size(x));


  r = 0;
  for i=1:length(c)

    r = r + c(i) * xi;

    xi = xi * x;
  end
```

Code differentiation

Example (Differentiated constants)

2. Constants have zero derivatives

```
function [d_r, r] = d_f(d_x, x)
  global c                    % no derivative
  xi = eye(size(x));
  d_xi = zeros(size(xi));     % deriv. of constant
  d_r = 0;                    % deriv. of constant
  r = 0;
  for i=1:length(c)

    r = r + c(i) * xi;

    xi = xi * x;
  end
```

Code differentiation

Example (Differentiated control flow statements)

3. Control flow statements are not differentiated

```
function [d_r, r] = d_f(d_x, x)
  global c                     % no derivative
  xi = eye(size(x));
  d_xi = zeros(size(xi)); % deriv. of constant
  d_r = 0;                    % deriv. of constant
  r = 0;
  for i=1:length(c)       % control flow unchanged

    r = r + c(i) * xi;

    xi = xi * x;
  end
```

Code differentiation

Example (Differentiated assignments)

4. Differentiate both sides of assignments

```
function [d_r, r] = d_f(d_x, x)
  global c                    % no derivative
  xi = eye(size(x));
  d_xi = zeros(size(xi)); % deriv. of constant
  d_r = 0;                    % deriv. of constant
  r = 0;
  for i=1:length(c)       % control flow unchanged
    d_r = d_r + c(i) * d_xi;      % deriv. of r
    r = r + c(i) * xi;
    d_xi = d_xi * x + xi * d_x; % deriv. of xi
    xi = xi * x;
  end
```

Run the differentiated code

Compute derivative with respect to x

```
d_x = 1;
[d_r, r] = d_f(d_x, x)

d_r =  49
r =  31
```

Directional derivatives

Definition (Directional derivative)

- The result of the AD process is a *directional derivative* along a direction vector $\vec{v}$

$$\frac{\mathrm{d}f}{\mathrm{d}\vec{x}}\Big|_{\vec{v}} := \frac{\mathrm{d}f(\vec{x} + t \cdot \vec{v})}{\mathrm{d}t}, \quad t \in \mathbb{R}$$

- Consider the values in `x` as a vector, so $\vec{x}$ corresponds to `x(:)`

Directional derivatives

Definition (Directional derivative)

- The result of the AD process is a *directional derivative* along a direction vector $\vec{v}$

$$\frac{\mathrm{d}f}{\mathrm{d}\vec{x}}|_{\vec{v}} := \frac{\mathrm{d}f(\vec{x} + t \cdot \vec{v})}{\mathrm{d}t}, \quad t \in \mathbb{R}$$

- Consider the values in `x` as a vector, so $\vec{x}$ corresponds to `x(:)`

- This is the AD interface
  - Given $\vec{v}$, return $\frac{\mathrm{d}f}{\mathrm{d}\vec{x}}|_{\vec{v}}$

Matrix arithmetic

Compute directional derivative w.r.t. x

```
x = magic(2)
```

Matrix arithmetic

Compute directional derivative w.r.t. x

```
x = magic(2)
```

```
d_x = [1 0; 0 1]; % deriv. direction v=(1,0,0,1)
```

Matrix arithmetic

Compute directional derivative w.r.t. x

```
x = magic(2)
```

```
d_x = [1 0; 0 1]; % deriv. direction v=(1,0,0,1)
```

```
[d_r, r] = d_f(d_x, x)

d_r =
  442  432
  144  154
r =
  587  582
  194  199
```

Verify the matrix AD result

Check matrix AD result with finite differences

```
v = [1  0;  0  1];
df_fx = (f(x + h.*v) - f(x - h.*v)) ./ (2.*h)

df_fx =
  442  432
  144  154
```

Verify the matrix AD result

Check matrix AD result analytically

```
df_dx = polyvalm(polyder(c), x)
```

Verify the matrix AD result

Check matrix AD result analytically

```
df_dx = polyvalm(polyder(c), x)
```

```
df_dx =
   442   432
   144   154
```

Verify the matrix AD result

Check matrix AD result analytically

```
df_dx = polyvalm(polyder(c), x)
```

```
df_dx =
   442   432
   144   154
```

- Where is the derivative direction here?
  - Implicit in the initial product `xi = eye(2)`

Differentiation background

- Differentiation is linear
- Jacobian matrix
- Computational expense of AD

Differentiation is linear

Differentiation = Linearization

$$\frac{\mathrm{d}f}{\mathrm{d}\vec{x}}|_{\vec{v}+\vec{u}} = \frac{\mathrm{d}f}{\mathrm{d}\vec{x}}|_{\vec{v}} + \frac{\mathrm{d}f}{\mathrm{d}\vec{x}}|_{\vec{u}}$$

Differentiation is linear

Differentiation = Linearization

$$\frac{\mathrm{d}f}{\mathrm{d}\vec{x}}|_{\vec{v}+\vec{u}} = \frac{\mathrm{d}f}{\mathrm{d}\vec{x}}|_{\vec{v}} + \frac{\mathrm{d}f}{\mathrm{d}\vec{x}}|_{\vec{u}}$$

Differentiation = Matrix product

$$\frac{\mathrm{d}f}{\mathrm{d}\vec{x}} \cdot (\vec{v} + \vec{u}) = \frac{\mathrm{d}f}{\mathrm{d}\vec{x}} \cdot \vec{v} + \frac{\mathrm{d}f}{\mathrm{d}\vec{x}} \cdot \vec{u}$$

Jacobian matrix

Definition (Jacobian matrix)

A function $f : \mathbb{R}^n \to \mathbb{R}^m$ has the derivative, or *Jacobian* matrix

$$J = \frac{\mathrm{d}f}{\mathrm{d}x} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix} \in \mathbb{R}^{m \times n}$$

Flatten MATLAB arrays to vectors, i.e. `x`$\to$`x(:)`$\in \mathbb{R}^n$ with $n =$ `numel(x)`

Jacobian matrix

Definition (Jacobian matrix)

A function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ has the derivative, or *Jacobian* matrix

$$J = \frac{\mathrm{d}f}{\mathrm{d}x} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix} \in \mathbb{R}^{m \times n}$$

Flatten MATLAB arrays to vectors, i.e. $\texttt{x} \rightarrow \texttt{x(:)} \in \mathbb{R}^n$ with $n =$ `numel(x)`

- For the i-th column of J, set $\vec{v} = \vec{e}_i$, the i-th canonical basis vector

Jacobian in forward mode

Example (Compute the full Jacobian of f)

```
J = zeros(numel(r), numel(x)); % J is m x n matrix
for i=1:numel(x)
  d_x = zeros(size(x));       % setup derivative input
  d_x(i) = 1;                 % set i-th component to 1
  [d_r, r] = d_f(d_x, x);     % run d_f
  J(:,i) = d_r(:);            % i-th col. of J
end
```

Jacobian in forward mode

Example (Compute the full Jacobian of f)

```
J = zeros(numel(r), numel(x)); % J is m x n matrix
for i=1:numel(x)
  d_x = zeros(size(x));       % setup derivative input
  d_x(i) = 1;                 % set i-th component to 1
  [d_r, r] = d_f(d_x, x);     % run d_f
  J(:,i) = d_r(:);            % i-th col. of J
end
```

```
J =
  403   255    85    39
   85   233    13    59
  255   117   233   177
   39   177    59   115
```

Computational expense of AD

- One run of `d_f` for each directional derivative
- n runs of `d_f` for the full Jacobian

Computational expense of AD

- One run of `d_f` for each directional derivative
- n runs of `d_f` for the full Jacobian

- Runtime of `d_f` within a constant factor c of runtime of `f`
  - $\frac{T_{\texttt{d\_f}}}{T_{\texttt{f}}} < c$
  - c is about 3, due to the multiplication rule
  - $\frac{T_{\texttt{d\_f}}}{T_{\texttt{f}}} = O(1)$

Computational expense of AD

- One run of `d_f` for each directional derivative
- n runs of `d_f` for the full Jacobian

- Runtime of `d_f` within a constant factor c of runtime of `f`
  - $\frac{T_{\mathtt{d\_f}}}{T_{\mathtt{f}}} < c$
  - c is about 3, due to the multiplication rule
  - $\frac{T_{\mathtt{d\_f}}}{T_{\mathtt{f}}} = O(1)$

- Runtime for the full Jacobian
  - $\frac{T_J}{T_{\mathtt{f}}} < cn$
  - $\frac{T_J}{T_{\mathtt{f}}} = O(n)$

Reverse mode

- Reverse accumulation of dervatives
- Adjoint code example

Reverse accumulation of derivatives

Consider a single assignment

$$r \leftarrow f(x, y)$$

with the derivative

$$\mathrm{d}r \leftarrow \frac{\partial f}{\partial x} \cdot \mathrm{d}x + \frac{\partial f}{\partial y} \cdot \mathrm{d}y$$

- Partial derivatives must be known
- When $\mathrm{d}r$ is known we can compute $\mathrm{d}x$ and $\mathrm{d}y$
- Reverse propagation of *adjoints* $\bar{x} := \mathrm{d}x^T$

$$\bar{x} \leftarrow \bar{r} \cdot \frac{\partial f}{\partial x}, \quad \bar{y} \leftarrow \bar{r} \cdot \frac{\partial f}{\partial y}$$

- Do this in reverse order of assignments → adjoint code

Adjoint code I

Example (Forward sweep)

```
function [a_x, nr_r] = adj_f(a_r, x)
  global c
  xi = eye(size(x));
  r = 0;
  for i=1:length(c)
    push(r)
    r = r + c(i) * xi;
    push(xi)
    xi = xi * x;
  end
  nr_r = r;
```

Adjoint code II

Example (Reverse sweep)

```
  a_xi = zeros(size(xi));
  a_x  = zeros(size(x));
  for i=fliplr(1:length(c))
    % xi = xi * x;
    xi    = pop();
    a_x   = a_x  + xi.' * a_xi;
    a_xi  =          a_xi * x.';
    % r   = r + c(i) * xi;
    r     = pop();
    a_xi  = a_xi + c(i).' * a_r;
    a_r   =          a_r;
  end
end
```

Running adjoint code

Example (Run the adjoint code)

```
a_r = [1  0;  0  0];
[a_x,  r] = adj_f(a_r,  x)
```

Running adjoint code

Example (Run the adjoint code)

```
a_r = [1  0;  0  0];
[a_x,  r] = adj_f(a_r,  x)
```

```
a_x =
   403   85
   255   39
r =
   587   582
   194   199
```

Running adjoint code

Example (Run the adjoint code)

```
a_r = [1 0; 0 0];
[a_x, r] = adj_f(a_r, x)
```

```
a_x =
  403   85
  255   39
r =
  587   582
  194   199
```

This is the derivative of the 1st component of `r` with respect to `x`

- First *row* of the Jacobian

Differentiation background II

- Jacobian of `f`
- Computational expense of AD

Jacobian in reverse mode

Example (Compute the full Jacobian of f in reverse mode)

```
J = zeros(numel(r), numel(x)); % J is m x n matrix
for i=1:numel(r)
  a_r = zeros(size(r));          % setup adjoint input
  a_r(i) = 1;                    % i-th component = 1
  [a_x, r] = adj_f(a_r, x);      % run adj_f
  J(i,:) = a_x(:).';             % i-th row of J
end
```

Jacobian in reverse mode

Example (Compute the full Jacobian of f in reverse mode)

```
J = zeros(numel(r), numel(x)); % J is m x n matrix
for i=1:numel(r)
  a_r = zeros(size(r));          % setup adjoint input
  a_r(i) = 1;                    % i-th component = 1
  [a_x, r] = adj_f(a_r, x);     % run adj_f
  J(i,:) = a_x(:).';             % i-th row of J
end
```

Directional adjoint

- The reverse mode produces a "directional adjoint"
- Linear combination of the *rows* of the Jacobian

$$\vec{v}^T \cdot J$$

Computational expense of reverse mode AD

- One run of `adj_f` for each directional adjoint
- m runs of `adj_f` for the full Jacobian

Computational expense of reverse mode AD

- One run of `adj_f` for each directional adjoint
- m runs of `adj_f` for the full Jacobian

- Runtime of `adj_f` within a constant factor of runtime of `f`
  - $\frac{T_{\texttt{adj\_f}}}{T_{\texttt{f}}} = O(1)$
  - Hidden constant c is about 10, due to the additional measures

Computational expense of reverse mode AD

- One run of `adj_f` for each directional adjoint
- m runs of `adj_f` for the full Jacobian

- Runtime of `adj_f` within a constant factor of runtime of `f`
  - $\frac{T_{\texttt{adj\_f}}}{T_{\texttt{f}}} = O(1)$
  - Hidden constant c is about 10, due to the additional measures

- Runtime for the full Jacobian
  - $\frac{T_J}{T_{\texttt{f}}} = O(m)$
- Runtime for the gradient of a scalar function
  - $\frac{T_{grad}}{T_{\texttt{f}}} = O(1)$

Computational expense of reverse mode AD

- One run of `adj_f` for each directional adjoint
- m runs of `adj_f` for the full Jacobian

- Runtime of `adj_f` within a constant factor of runtime of `f`
  - $\frac{T_{\texttt{adj\_f}}}{T_{\texttt{f}}} = O(1)$
  - Hidden constant c is about 10, due to the additional measures

- Runtime for the full Jacobian
  - $\frac{T_J}{T_{\texttt{f}}} = O(m)$
- Runtime for the gradient of a scalar function
  - $\frac{T_{grad}}{T_{\texttt{f}}} = O(1)$

- Stack memory for a single run of `adj_f`
  - $M_{\texttt{adj\_f}} = O(T_{\texttt{f}})$

ADiMat

ADiMat is one of the leading purveyors of fine derivatives

ADiMat

ADiMat is one of the leading purveyors of fine derivatives

- `http://www.adimat.de`
- Differentiation of MATLAB in forward and reverse mode

ADiMat

ADiMat is one of the leading purveyors of fine derivatives

- http://www.adimat.de
- Differentiation of MATLAB in forward and reverse mode

Agenda

- Using ADiMat
- Sparsity exploitation

Using ADiMat

ADiMat provides very easy-to-use driver functions for the various differentiation methods, all with the same calling sequence

AD driver functions

`admDiffFor(@f, S, x, ..., opts)` forward mode, return $J \cdot S$

`admDiffRev(@f, S, x, ..., opts)` reverse mode, return $S \cdot J$

Numerical differentiation functions

`admDiffFD(@f, S, x, ..., opts)` finite differences

`admDiffComplex(@f, S, x, ..., opts)` complex step method

both return $J \cdot S$

Seed matrices

- The forward mode computes $J \cdot S$
  - S: Bundle of derivative directions
  - $J \cdot S$: Bundle of directional derivatives

```
S = eye(4);
admDiffFor(@f, S, x)

ans =
   403   255    85    39
    85   233    13    59
   255   117   233   177
    39   177    59   115
```

- 4 columns in S
- Costs: $4cT_{\mathtt{f}}$

```
S = [1 0 0 1].';
admDiffFor(@f, S, x)

ans =
   442
   144
   432
   154
```

- 1 column in S
- Costs: $1cT_{\mathtt{f}}$

Seed matrices

- The reverse mode computes $S \cdot J$
  - S: Bundle of adjoint directions
  - $S \cdot J$: Bundle of directional adjoints

```
S = eye(4);
admDiffRev(@f, S, x)

ans =
   403   255    85    39
    85   233    13    59
   255   117   233   177
    39   177    59   115
```

- 4 rows in S
- Costs: $4cT_{\mathtt{f}}$

```
S = [1 0 0 1];
admDiffRev(@f, S, x)

ans =
   442   432   144   154
```

- 1 row in S
- Costs: $1cT_{\mathtt{f}}$

Sparsity exploitation

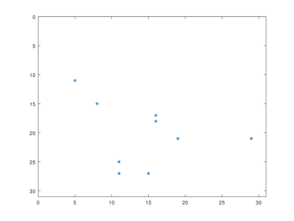
When the Jacobian is *sparse* we can optimize

- When any two columns (FM) or rows (RM) can be added non-destructively, add the derivative directions, and save one
  - *Compression* of the Jacobian
- Required: *non-zero pattern* of the Jacobian ($\rightarrow$ **spy**)
- General: *graph coloring* discrete optimization problem
  - Heuristic: Curtis-Powell-Reid coloring [CURTIS et al.(1974)CURTIS, POWELL, and REID]

Sparsity exploitation with ADiMat

Example (30x30 sparse matrix)

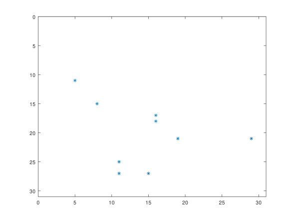
```
N = 30;
x = sprand(N, N, 1e-2);
spy(x);
```



Sparsity exploitation with ADiMat

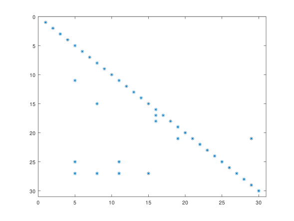
Example (30x30 sparse matrix)

```
N = 30;
x = sprand(N, N, 1e-2);
spy(x);
```



Example (Polynomial of 30x30 sparse matrix)

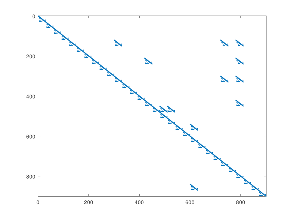
```
r = f(x);
spy(r);
```



Sparsity exploitation with ADiMat

Compute the sparse Jacobian fully

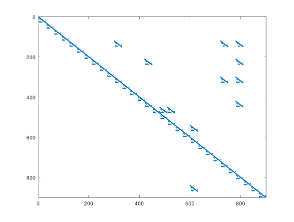
```
[J, r] = admDiffFor(@f, 1, x);
spy(J);
```



Sparsity exploitation with ADiMat

Compute the sparse Jacobian fully

```
[J, r] = admDiffFor(@f, 1, x);
spy(J);
```



Exploit sparsity

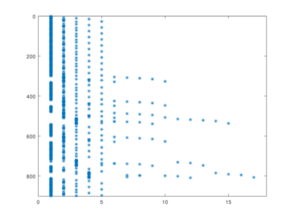
```
adopts = admOptions('JPattern', J ~= 0);
[J, r] = admDiffFor(@f, 1, x, adopts);

Colored the pattern with 17 colors
Compressed seed matrix: 900x17
```

Sparsity exploitation with ADiMat

Internal: the 900x17 compressed seed matrix

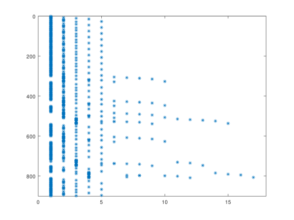
```
cS = admColorSeed(J ~= 0);
spy(cS)
```



Sparsity exploitation with ADiMat

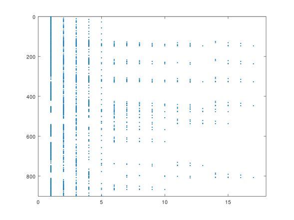
Internal: the 900x17 compressed seed matrix

```
cS = admColorSeed(J ~= 0);
spy(cS)
```



Internal: the 900x17 compressed Jacobian

```
cJ = admDiffFor(@f, cS, x);
spy(cJ)
```



Alternatives to automatic differentiation

- Numerical methods
  - Finite differences
  - Complex step method
- Symbolic differentiation
- Analytical derivation

Numerical method: Finite differences

- Evaluate derivative definition with *appropriate* $h > 0$
- Central finite differences are more accurate

$$\frac{\mathrm{d}f}{\mathrm{d}\vec{x}}|_{\vec{v}} \approx \frac{f(\vec{x}+h\vec{v}) - f(\vec{x}-h\vec{v})}{2h}$$

- Computational expense: $O(1)\,T_{\mathtt{f}}$ per directional derivative

Numerical method: Finite differences

- Evaluate derivative definition with *appropriate* $h > 0$
- Central finite differences are more accurate

$$\frac{\mathrm{d}f}{\mathrm{d}\vec{x}}|_{\vec{v}} \approx \frac{f(\vec{x} + h\vec{v}) - f(\vec{x} - h\vec{v})}{2h}$$

- Computational expense: $O(1)\, T_{\mathtt{f}}$ per directional derivative

Pros

- Only f is needed (black box)
- Corresponds to definition of derivative

Numerical method: Finite differences

- Evaluate derivative definition with *appropriate* $h > 0$
- Central finite differences are more accurate

$$\frac{\mathrm{d}f}{\mathrm{d}\vec{x}}\Big|_{\vec{v}} \approx \frac{f(\vec{x} + h\vec{v}) - f(\vec{x} - h\vec{v})}{2h}$$

- Computational expense: $O(1)\,T_{\mathtt{f}}$ per directional derivative

Pros

- Only f is needed (black box)
- Corresponds to definition of derivative

Cons

- Accuracy *at best* half of the machine precision
- Find appropriate value for h
- Spurious results when too close to a jump

Numerical method: Complex step

- Evaluate function with complex argument with a very small imaginary part

$$\frac{\mathrm{d}f}{\mathrm{d}\vec{x}}|_{\vec{v}} = \frac{\mathrm{imag}\, f(\vec{x} + \mathrm{i}h\vec{v})}{h}, \quad h = 10^{-60}$$

  [Lyness and Moler(1967)]
- Computational expense: $O(1)\, T_{\mathtt{f}}$ per directional derivative

Numerical method: Complex step

- Evaluate function with complex argument with a very small imaginary part

$$\frac{\mathrm{d}f}{\mathrm{d}\vec{x}}|_{\vec{v}} = \frac{\mathrm{imag}\, f(\vec{x} + \mathrm{i}h\vec{v})}{h}, \quad h = 10^{-60}$$

  [Lyness and Moler(1967)]
- Computational expense: $O(1)\, T_{\mathtt{f}}$ per directional derivative

Pros

- Full accuracy

Numerical method: Complex step

- Evaluate function with complex argument with a very small imaginary part

$$\frac{\mathrm{d}f}{\mathrm{d}\vec{x}}|_{\vec{v}} = \frac{\mathrm{imag}\, f(\vec{x} + \mathrm{i}h\vec{v})}{h}, \quad h = 10^{-60}$$

[Lyness and Moler(1967)]
- Computational expense: $O(1)\,T_{\mathtt{f}}$ per directional derivative

Pros

- Full accuracy

Cons

- Only when function is *real analytic*
- Change of variable types
  - In MATLAB usually no problem at all

Mathematical method: Symbolic differentiation

- Differentiate formula for f symbolically
  - With a computer program

Mathematical method: Symbolic differentiation

- Differentiate formula for f symbolically
  - With a computer program

Pros

- Full accuracy

Mathematical method: Symbolic differentiation

- Differentiate formula for f symbolically
  - With a computer program

Pros

- Full accuracy

Cons

- Need to convert f to formula
- Need to convert $\mathrm{d}f$ back to program
- Combinatorial explosion
- Computational expense: at least $O(1)T_{\mathtt{f}}$ per directional derivative

Combinatorial explosion in symbolic differentiation

Example (Differentiation squares number of factors)

Maxima code

```
f(x) := u(x) * v(x) * w(x);
diff(f(x), x);
```

- Product of $p = 3$ factors

Combinatorial explosion in symbolic differentiation

Example (Differentiation squares number of factors)

Maxima code

```
f(x) := u(x) * v(x) * w(x);
diff( f(x), x);
```

- Product of $p = 3$ factors

```
             d                        d
u(x) v(x) (-- (w(x))) + u(x) w(x) (-- (v(x)))
             dx                       dx
                     d
      + v(x) w(x) (-- (u(x)))
                     dx
```

- p^2 factors in total

Avoiding combinatorial explosion in AD

In AD you also do symbolic differentiation on the RHS of assignments? Yes, but . . .

Example (Outlining limits depth of RHS expressions)

```
tmp   = u(x) * v(x);
r     = tmp * w(x);
```

- $p - 1$ assignments

Avoiding combinatorial explosion in AD

In AD you also do symbolic differentiation on the RHS of assignments? Yes, but ...

Example (Outlining limits depth of RHS expressions)

```
tmp   = u(x) * v(x);
r     = tmp * w(x);
```

- $p - 1$ assignments

```
d_tmp   = d_u(x) * v(x) + u(x) * d_v(x);
tmp     = u(x) * v(x);
d_r     = d_tmp * w(x) + tmp * d_w(x);
r       = tmp * w(x);
```

- $(p - 1) \cdot 4$ factors in total

Analytical derivation

- Derive derivative on pen and paper
  - Special case: the adjoint PDE of a PDE being considered
- Implement as computer program
  - E.g. solve adjoint PDE with adequate method

Analytical derivation

- Derive derivative on pen and paper
  - Special case: the adjoint PDE of a PDE being considered
- Implement as computer program
  - E.g. solve adjoint PDE with adequate method

Pros

- Computational expense: $O(1)\,T_{\mathtt{f}}$ per direction
- Full accuracy
- Solving an adjoint PDE is equivalent to the reverse mode
  - Computational expense: $O(1)\,T_{\mathtt{f}}$ per directional adjoint

Analytical derivation

- Derive derivative on pen and paper
  - Special case: the adjoint PDE of a PDE being considered
- Implement as computer program
  - E.g. solve adjoint PDE with adequate method

Pros

- Computational expense: $O(1)T_{\mathtt{f}}$ per direction
- Full accuracy
- Solving an adjoint PDE is equivalent to the reverse mode
  - Computational expense: $O(1)T_{\mathtt{f}}$ per directional adjoint

Cons

- Very tedious and error-prone
  - Danger of $\mathrm{d}f$ and f diverging when f is changed
- Adjoint PDE may be mathematically, technically more complex

Second order derivatives

- Second order derivatives
- Computational expense of Hessians
- Hessians with ADiMat

Second order derivatives

- Second order derivatives might be obtained by differentiating the forward mode or adjoint code again
  - Difficult to do with (our) source transformation
- In ADiMat we use *operator overloading* to achieve forward-over-reverse mode
  - A class propagates derivatives in forward mode
  - Provides all relevant operators and methods
  - Also, higher order univariate Taylor series
- Forward-over-reverse mode: run adjoint code with Taylor objects

Computational expense of Hessians

Costs for full Hessian

- Gradient of $f : \mathbb{R}^n \to \mathbb{R}$ in reverse mode: $O(1)T_{\mathtt{f}}$
  - Gradient is a function $\mathrm{d}f : \mathbb{R}^n \to \mathbb{R}^n$
- Differentiate gradient in forward mode: $O(n)T_{\mathtt{f}}$
- Overall cost for the full Hessian: $O(n)T_{\mathtt{f}}$
  - Compare to second order forward mode or FD: $O(n^2)T_{\mathtt{f}}$
- Stack memory: $O(T_{\mathtt{f}})$

Computational expense of Hessians

Costs for full Hessian

- Gradient of $f : \mathbb{R}^n \to \mathbb{R}$ in reverse mode: $O(1)T_{\mathtt{f}}$
  - Gradient is a function $\mathrm{d}f : \mathbb{R}^n \to \mathbb{R}^n$
- Differentiate gradient in forward mode: $O(n)T_{\mathtt{f}}$
- Overall cost for the full Hessian: $O(n)T_{\mathtt{f}}$
  - Compare to second order forward mode or FD: $O(n^2)T_{\mathtt{f}}$
- Stack memory: $O(T_{\mathtt{f}})$

Costs for Hessian-vector product

- Single directional derivative of $\mathrm{d}f$ in forward mode: $O(1)T_{\mathtt{f}}$
- Overall cost for the Hessian-vector product: $O(1)T_{\mathtt{f}}$
  - Compare to second order forward mode: $O(n)T_{\mathtt{f}}$
- Stack memory: $O(T_{\mathtt{f}})$

Hessians with ADiMat

Hessian driver

`admHessian(@f, {Y, V, W}, x, ..., opts)`
: forward-over-reverse mode, return linear combinations, per the rows of Y, of $V \cdot H_k \cdot W$ of the Hessians H_k of the $1 \leq k \leq m$ function results

Costs

- Time: $O(pq)T_{\mathtt{f}}$, where
  - p is the number of rows in Y
  - q is the number of columns in W
- Memory: $O(T_{\mathtt{f}})$

Generalization: Taylor-over-reverse mode

admTaylorRev
: Run adjoint code with Taylor objects with truncation order > 1

Constraint optimization with ADiMat

- Most optimization routines or solvers provide a mechanism for the user to supply derivatives
- Example: **fmincon** [MathWorks(2018)]
- Relevant options, depending on the algorithm

SpecifyObjectiveGradient set to `true` $\rightarrow$ objective function returns gradient

SpecifyConstraintGradient set to `true` $\rightarrow$ constraints function returns gradients

HessianFcn set to `'objective'` $\rightarrow$ objective function returns Hessian

set to function handle $\rightarrow$ returns Hessian of Lagrangian

HessianMultiplyFcn set to function handle $\rightarrow$ returns Hessian-vector product

fmincon with ADiMat

Example (Function to evaluate objective and derivatives)

```
function [f, g, H] = myobj_wrap(x)
  if nargout == 1            % Objective only
    f = myobj(x);
  elseif nargout == 2        % Gradient required
    [g, f] = admDiffRev(@myobj, 1, x);
  elseif nargout == 3        % Hessian and gradient
    [H, g, f] = admHessian(@myobj, 1, x);
  end
```

fmincon with ADiMat

Example (Function to evaluate objective and derivatives)

```
function [f, g, H] = myobj_wrap(x)
  if nargout == 1              % Objective only
    f = myobj(x);
  elseif nargout == 2          % Gradient required
    [g, f] = admDiffRev(@myobj, 1, x);
  elseif nargout == 3          % Hessian and gradient
    [H, g, f] = admHessian(@myobj, 1, x);
  end
```

```
options = optimoptions('fmincon',...
  'Algorithm','trust-region', ...
  'SpecifyObjectiveGradient',true,...
  'HessianFcn','objective');
[x fval] = fmincon(@myobj_wrap, x0, options)
```

Non-linear constraints

With non-linear constraints the so-called Lagrangian arises

$$L(x, \lambda) = f(x) + \sum_i \lambda_{g,i} g_i(x) + \sum_i \lambda_{h,i} h_i(x)$$

- The solver requires the Hessian of the Lagrangian

$$\nabla^2_{xx} L(x, \lambda) = \nabla^2 f(x) + \sum_i \lambda_{g,i} \nabla^2 g_i(x) + \sum_i \lambda_{h,i} \nabla^2 h_i(x)$$

- With ADiMat, set the adjoint seed matrix $Y = \lambda^T$

Non-linear constraints

With non-linear constraints the so-called Lagrangian arises

$$L(x,\lambda) = f(x) + \sum_i \lambda_{g,i} g_i(x) + \sum_i \lambda_{h,i} h_i(x)$$

- The solver requires the Hessian of the Lagrangian

$$\nabla^2_{xx} L(x,\lambda) = \nabla^2 f(x) + \sum_i \lambda_{g,i} \nabla^2 g_i(x) + \sum_i \lambda_{h,i} \nabla^2 h_i(x)$$

- With ADiMat, set the adjoint seed matrix $Y = \lambda^T$

Example (Evaluate the Hessian of the Lagrangian)

```
function H = myhess(x, lambda)
 H = admHessian(@myobj, 1, x) ...
  + admHessian(@myc, {lambda.ineqnonlin.',1,1}, x)...
  + admHessian(@myeqc, {lambda.eqnonlin.',1,1}, x);
```

fmincon with non-linear constraints

Example (Evaluate the Hessian of the Lagrangian II)

Implement Lagrangian

```
function r = mylag(x, lambda)
   r = myobj(x) ...
       + dot(lambda.ineqnonlin, myc(x)) ...
       + dot(lambda.eqnonlin, myeqc(x));
```

fmincon with non-linear constraints

Example (Evaluate the Hessian of the Lagrangian II)

Implement Lagrangian

```
function r = mylag(x, lambda)
  r = myobj(x) ...
      + dot(lambda.ineqnonlin, myc(x)) ...
      + dot(lambda.eqnonlin, myeqc(x));
```

Apply AD

```
function H = myhess2(x, lambda)
  H = admHessian(@mylag, 1, x, lambda, ...
                 admOptions('i', 1));
```

fmincon with non-linear constraints

Example (Evaluate constraint gradients)

```
function [c,ceq,gc,gceq] = myconstr_wrap(x)
  if nargout <= 2
    c = myc(x);
    ceq = myeqc(x);
  else
    % experiment with FM and RM, sparsity, etc.
    [gc, c]      = admDiffFor(@myc, 1, x);
    [gceq, ceq] = admDiffRev(@myeqc, 1, x);
    % appearently, fmincon wants J transposed
    gc   = gc.';
    gceq = gceq.';
  end
```

fmincon with non-linear constraints

Example (Run fmincon with constraints)

```
options = optimoptions('fmincon',...
    'Algorithm','interior-point',...
    'SpecifyObjectiveGradient',true,...
    'SpecifyConstraintGradient',true,...
    'HessianFcn',@myhess);
[x,fval,exitflag,output] = ...
   fmincon(@myobj_wrap,x0,[],[],[],[],[],[],...
           @myconstr_wrap,options);
```

Literature I

Community Portal for Automatic Differentiation.
URL http://www.autodiff.org.

Christian H. Bischof, Bruno Lang, and André Vehreschild.
Automatic differentiation for MATLAB programs.
Proceedings in Applied Mathematics and Mechanics, 2(1):50–53, 2003.
doi: 10.1002/pamm.200310013.

A. R. CURTIS, M. J. D. POWELL, and J. K. REID.
On the Estimation of Sparse Jacobian Matrices.
IMA Journal of Applied Mathematics, 13(1):117–119, 1974.
URL http://dx.doi.org/10.1093/imamat/13.1.117.

Literature II

Mike Giles and Paul Glasserman.
Smoking Adjoints: fast evaluation of Greeks in Monte Carlo calculations.
Technical report, Unspecified, 2005.
URL http://eprints.maths.ox.ac.uk/1138/.

J.N. Lyness and C.B. Moler.
Numerical differentiation of analytic functions.
SIAM Journal on Numerical Analysis, 4(2):202–210, 1967.

MathWorks.
fmincon, 2018.
URL https://www.mathworks.com/help/optim/ug/fmincon.html.

Literature III

André Vehreschild.
Automatisches Differenzieren für MATLAB.
PhD thesis, RWTH Aachen University, 2009.
URL http://darwin.bth.rwth-aachen.de/opus3/volltexte/2009/2680/.

Johannes Willkomm, Christian H. Bischof, and H. Martin Bücker.
A New User Interface for ADiMat: Toward Accurate and Efficient Derivatives of MATLAB Programmes with Ease of Use.
Int. J. Comput. Sci. Eng., 9(5/6):408–415, September 2014.
URL http://dx.doi.org/10.1504/IJCSE.2014.064526.