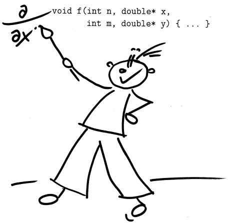
Algorithmic Differentiation Primer

Part IV: Mind the Gap (Overloading with dco/c++)



Uwe Naumann

Software and **T**ools for **C**omputational **E**ngineering
Computer Science, RWTH Aachen University
Aachen, Germany

naumann@stce.rwth-aachen.de
www.stce.rwth-aachen.de

and

The **N**umerical **A**lgorithms **G**roup Ltd.
Oxford, United Kingdom

Uwe.Naumann@nag.co.uk
www.nag.co.uk

Contents

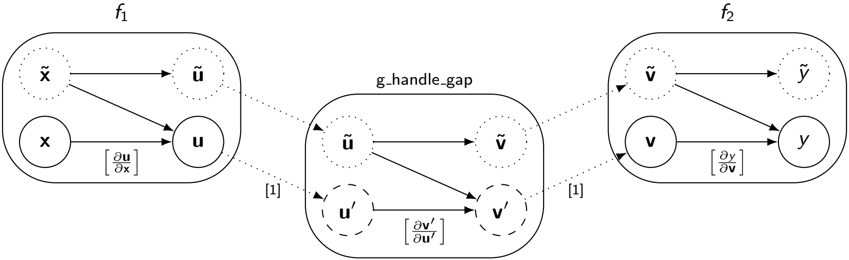
Motivation

Motivation

- checkpointing
  - joint call reversal
  - evolutions
  - ensembles
- symbolic differentiation of numerical methods
  - linear systems
  - nonlinear systems
  - nonlinear convex optimizers
- preaccumulation
  - statement-level adjoint gradients
  - local Jacobians
- embedded adjoint source code
- finite difference smoothing
- reverse accumulation of adjoint fix point iterations

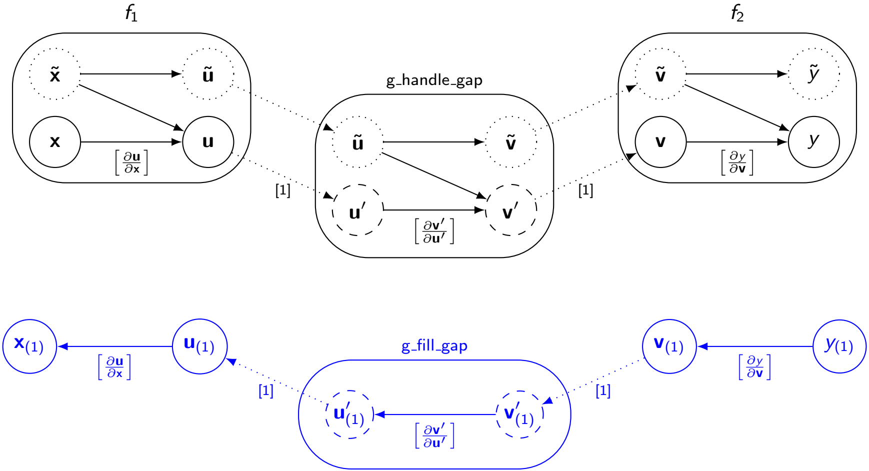

MIND THE GAP

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial f_2}{\partial \mathbf{v}} \cdot \frac{\partial g}{\partial \mathbf{u}} \cdot \frac{\partial f_1}{\partial \mathbf{x}}$$



Mind the Gap in the Chain Rule

... for Adjoint of $(y, \tilde{y}) = f_2(g(f_1(\mathbf{x}, \tilde{\mathbf{x}})))$



Checkpointing

Live: Derivative of 10^9 iterations of

$$x = \sin(x \cdot p)$$

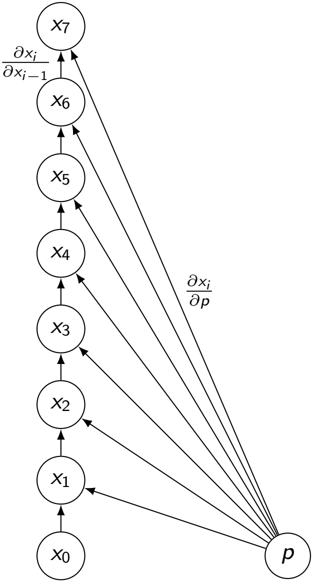
by

- ~/checkpointing/ga1s_split/

$\Rightarrow$ `std::bad_alloc`

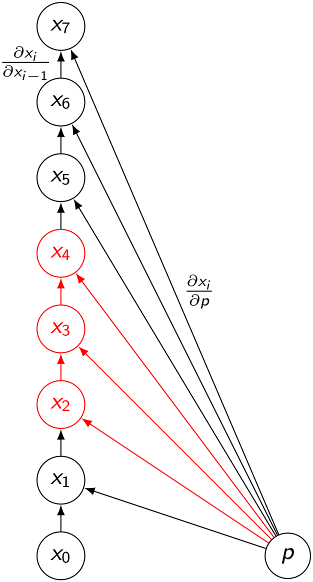
Call Reversal Modes

Split [2]



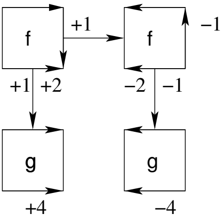
(MEM,OPS)

$(1,1) \leftarrow$ push(x_0); compute(x_1)
$(2,2) \leftarrow$ push(x_1); compute(x_2)
$(3,3) \leftarrow$ push(x_2); compute(x_3)
$(4,4) \leftarrow$ push(x_3); compute(x_4)
$(5,5) \leftarrow$ push(x_4); compute(x_5)
$(6,6) \leftarrow$ push(x_5); compute(x_6)
$(7,7) \leftarrow$ push(x_6); compute(x_7)
$(8,\ ) \leftarrow$ save(x_7)
$(7,\ ) \leftarrow$ pop(x_6); adjoint(x_6)
$(6,\ ) \leftarrow$ pop(x_5); adjoint(x_5)
$(5,\ ) \leftarrow$ pop(x_4); adjoint(x_4)
$(4,\ ) \leftarrow$ pop(x_3); adjoint(x_3)
$(3,\ ) \leftarrow$ pop(x_2); adjoint(x_2)
$(2,\ ) \leftarrow$ pop(x_1); adjoint(x_1)
$(1,\ ) \leftarrow$ pop(x_0); adjoint(x_0)
$(0,\ ) \leftarrow$ recover(x_7)



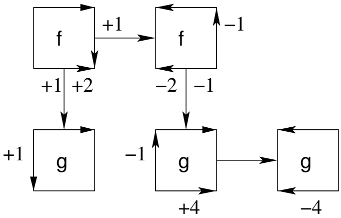
(MEM,OPS)

$(1,1) \leftarrow$ push(x_0); compute(x_1)
$(2,5) \leftarrow$ store(x_1); compute(x_5)
$(3,6) \leftarrow$ push(x_5); compute(x_6)
$(4,7) \leftarrow$ push(x_6); compute(x_7)
$(5,\ ) \leftarrow$ save(x_7)
$(4,\ ) \leftarrow$ pop(x_6); adjoint_to(x_6)
$(3,\ ) \leftarrow$ pop(x_5); adjoint_to(x_5)
$(2,\ ) \leftarrow$ restore(x_1);
$(3,8) \leftarrow$ push(x_1); compute(x_2)
$(4,9) \leftarrow$ push(x_2); compute(x_3)
$(5,10) \leftarrow$ push(x_3); compute(x_4)
$(6,11) \leftarrow$ push(x_4); compute(x_5);
$(5,\ ) \leftarrow$ pop(x_4); adjoint_to(x_4)
$(4,\ ) \leftarrow$ pop(x_3); adjoint_to(x_3)
$(3,\ ) \leftarrow$ pop(x_2); adjoint_to(x_2)
$(2,\ ) \leftarrow$ pop(x_1); adjoint_to(x_1)
$(1,\ ) \leftarrow$ pop(x_0); adjoint_to(x_0)
$(0,\ ) \leftarrow$ recover(x_7)



(f,g,0)

MEM=8, OPS=7



(f,g,1)

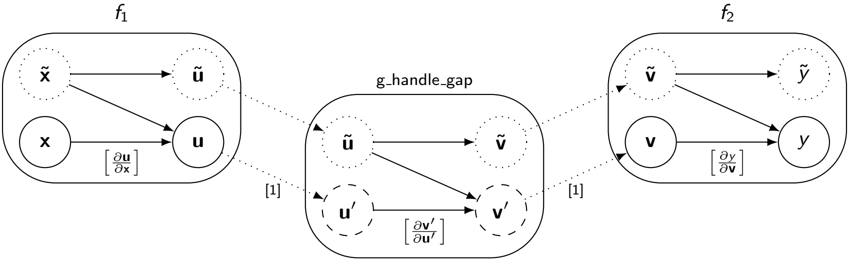
MEM=6, OPS=10

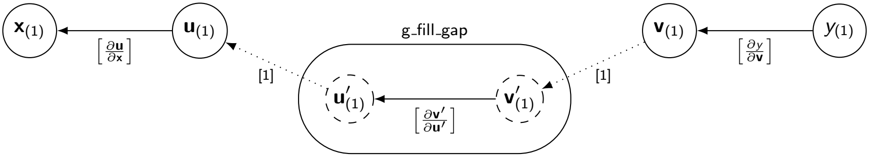
See [5] for notation.

[... read pre-order depth-first from left to right]

Mind the Gap in the Tape

… for Adjoint of $(y, \tilde{y}) = f_2(g(f_1(\mathbf{x}, \tilde{\mathbf{x}})))$





A gap $(\mathbf{v}', \tilde{\mathbf{v}}) = g(\mathbf{u}', \tilde{\mathbf{u}})$ in the tape is introduced by calling a user-defined function g_make_gap to record the following gap_data:

- Tape location of active gap inputs $\mathbf{u}$ in order to write $\mathbf{u}_{(1)} := \mathbf{u}_{(1)} + \mathbf{u}'_{(1)}$ correctly;
- adjoint gap input checkpoint $\subset (\mathbf{u}, \tilde{\mathbf{u}}, \mathbf{v}, \tilde{\mathbf{v}})$ in order to initialize interpretation of the gap correctly;
- tape location of active gap outputs $\mathbf{v}$ in order to initialize $\mathbf{v}'_{(1)} := \mathbf{v}_{(1)}$ and, hence, interpret gap correctly;

This data is stored in the tape (external function object factory) together with a reference to a user-defined function g_fill_gap to increment $\mathbf{u}_{(1)}$ with $\left(\frac{\partial \mathbf{v}}{\partial \mathbf{u}}\right)^T \cdot \mathbf{v}_{(1)}$.

```
template<typename T>
void g(const int n, T& x, const T& p) {
  for (int i=0;i<n;i++) x=sin(x*p);
}
template<typename T>
void f(const int n, T& x, const T& p) {
  for (int i=0;i<n/3;i++) x=sin(x*p);
  g(n/3,x,p);
  for (int i=0;i<n-2*n/3;i++) x=sin(x*p);
}
...
f(10,x,p);
```

See

- `~/checkpointing/ga1s_joint`
- use `~/checkpointing/gt2s_gt1s` as reference
- check memory requirement using AD_TAPE_POINTER->get_tape_memory_size()
- `~/checkpointing/gt2s_ga1s_joint`

```
template<typename T>
void g(const int n, T& x, const T& p) {
  for (int i=0;i<n;i++) x=cos(x)/p;
}

template<typename T>
void f(const int n, T& x, const T& p) {
  for (int i=0;i<n/3;i++) x=cos(x)/p;
  g(n/3,x,p);
  for (int i=0;i<n-2*n/3;i++) x=cos(x)/p;
}
```

Compute first derivative of output x wrt. input p for given inputs n and x by

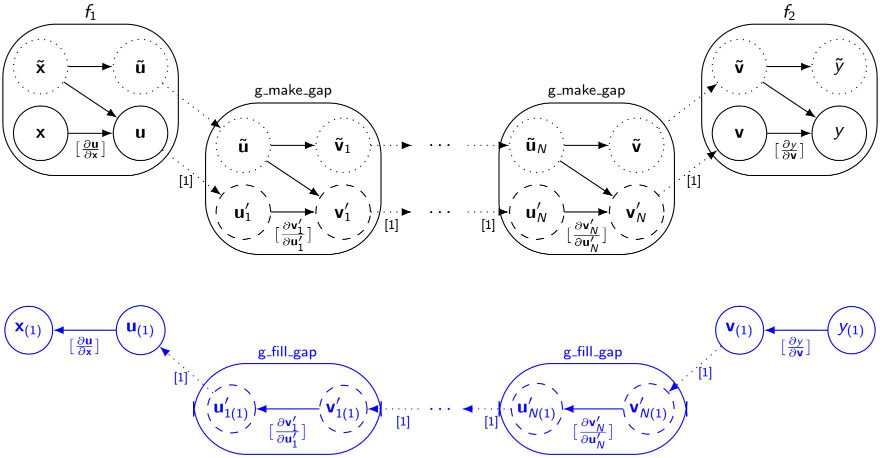
- central finite differences,
- tangent 1st-order scalar code,
- adjoint 1st-order scalar code in split mode,
- adjoint 1st-order scalar code in joint mode.

Compute second derivative of output x wrt. input p by

- tangent 2nd-order scalar code,
- adjoint 2nd-order scalar code as tangent over joint adjoint.

Checkpointing Evolutions

Checkpointing Evolutions



Let $F : \mathbb{R}^n \to \mathbb{R}^n$ defined by

$$F(x) = F_4(F_3(F_2(F_1(x))))$$

be implemented as

$$x = F_i(x) \quad \text{for } i = 1, \ldots, 4$$

with equal computational cost $OPS(F_i)$ for all i.

By the chain rule of differential calculus, differentiability of the F_i implies differentiability of F and

$$\nabla F(x) = \nabla F_4(F_3(F_2(F_1(x)))) \cdot \nabla F_3(F_2(F_1(x))) \cdot \nabla F_2(F_1(x)) \cdot \nabla F_1(x).$$

Checkpointing techniques aim to trade (the probably excessive) memory requirement of the (discrete) adjoint $F_{(1)}$ ($MEM(F_{(1)}) \approx 2 \cdot MEM(F) + OPS(F)$, $OPS(F_{(1)}) \approx \nu \cdot OPS(F)$) for additional operations.

For example,

$$\begin{aligned} x_0 &= x \\ x_2 &= F_2(F_1(x)) \end{aligned}$$

$$\nabla F(x)^T \cdot y_{(1)} = \nabla F_1(x_0)^T \cdot \left(\nabla F_2(\underbrace{F_1(x_0)}_{:=x_1})^T \cdot \left(\nabla F_3(x_2)^T \cdot \left(\nabla F_4(\underbrace{F_3(x_2)}_{:=x_3})^T \cdot y_{(1)} \right)\right)\right)$$

yields

$$\begin{aligned} OPS(F_{(1)}) &\approx \nu \cdot OPS(F) + 2 \cdot OPS(F_i) \\ MEM(F_{(1)}) &\approx 2 \cdot MEM(F) + OPS(F_i) \end{aligned}$$

```
template<typename AD_MODE>
void g_make_gap(int n, typename AD_MODE::type &x) {
  ...
}

template<typename AD_MODE>
void f(int n, int m, typename AD_MODE::type& x) {
  for (int i=0;i<n;i+=m)
    g_make_gap<AD_MODE>(min(m,n-i),x);
}
```

See

- ~/checkpointing/ga1s_checkpointing_loop_equidistantly
- ~/checkpointing/gt2s_ga1s_checkpointing_loop_equidistantly
- check memory requirement using AD_TAPE_POINTER->get_tape_memory_size()

```
template<typename T>
void g(int n, T& x, T p) {
  for (int i=0;i<n;i++) x=sin(x*p);
}

template<typename T>
void f(int n, int m, T& x, T p) {
  for (int i=0;i<n;i+=m) g(min(m,n-i),x,p);
}
```

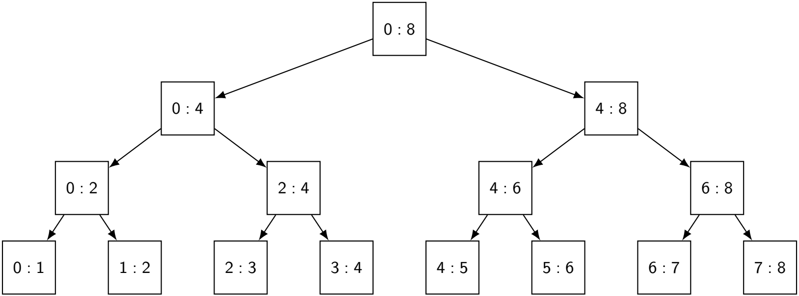
Compute first derivative of output `x` wrt. input `p` for given inputs `n`, `x`, and `p` by

- central finite differences,
- tangent 1st-order scalar code,
- adjoint 1st-order scalar code without checkpointing,
- adjoint 1st-order scalar code with equidistant checkpointing.

Compute second derivative of output `x` wrt. input `p` by

- tangent 2nd-order scalar code,
- adjoint 2nd-order scalar code as tangent over adjoint with equidistant checkpointing.

Consider the following call tree of a recursive bisection of loop of length 8:



See white board for fully joint reversal.

1. never push checkpoint for left-most subtree at recursion level ≥ 1
2. pop checkpoints only at leaf-level; access top checkpoint otherwise

```
template<typename AD_TYPE>
void g(int from, int to, int stride, AD_TYPE& x) {
  if (to-from>stride) {
    g(from,from+(to-from)/2,stride,x);
    g(from+(to-from)/2,to,stride,x);
  }
  else
    for (int i=from;i<to;i++) x=sin(x);
}
```

See

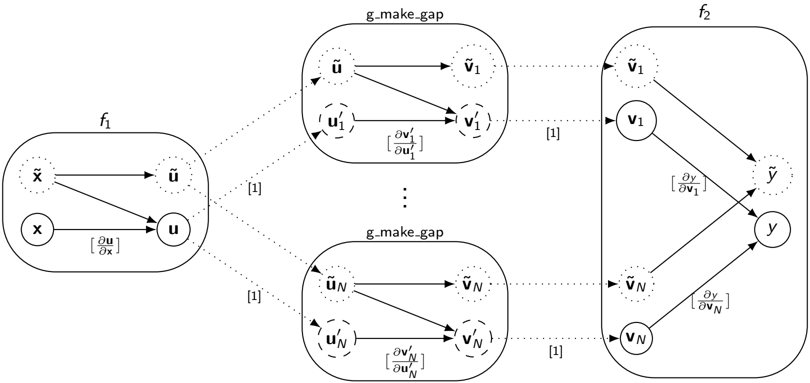
- `~/checkpointing/ga1s_checkpointing_loops_recursive_bisection`
- check memory requirement using AD_TAPE_POINTER->get_tape_memory_size()

```
template<typename AD_TYPE>
void g(int from, int to, int stride, AD_TYPE& x) {
  if (to-from>stride) {
    g(from,from+(to-from)/2,stride,x);
    g(from+(to-from)/2,to,stride,x);
  }
  else
    for (int i=from;i<to;i++) x=sin(x);
}
```

Compute first derivative of output `x` wrt. input `p` for given inputs `x`, `p`, `from`, `to`, and `stride` by

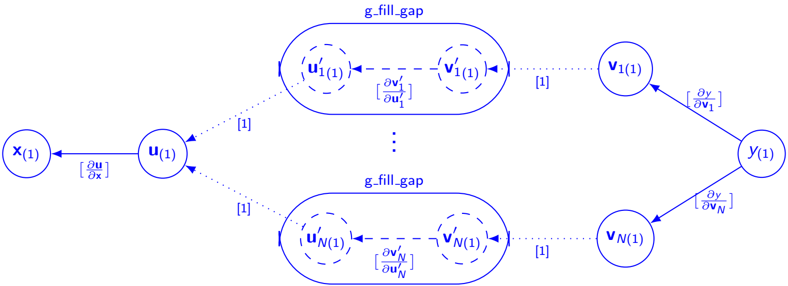
- central finite differences,
- tangent 1st-order scalar code,
- adjoint 1st-order scalar code without checkpointing,
- adjoint 1st-order scalar code with multilevel checkpointing (bisection).

Checkpointing Ensembles



f_1
$\tilde{\mathbf{x}}$
$\tilde{\mathbf{u}}$
$\mathbf{x}$
$\mathbf{u}$
$[\frac{\partial \mathbf{u}}{\partial \mathbf{x}}]$
[1]
[1]
g_make_gap
$\tilde{\mathbf{u}}$
$\tilde{\mathbf{v}}_1$
$\mathbf{u}'_1$
$\mathbf{v}'_1$
$[\frac{\partial \mathbf{v}'_1}{\partial \mathbf{u}'_1}]$
⋮
g_make_gap
$\tilde{\mathbf{u}}_N$
$\tilde{\mathbf{v}}_N$
$\mathbf{u}'_N$
$\mathbf{v}'_N$
$[\frac{\partial \mathbf{v}'_N}{\partial \mathbf{u}'_N}]$
[1]
[1]
f_2
$\tilde{\mathbf{v}}_1$
$\mathbf{v}_1$
$[\frac{\partial y}{\partial \mathbf{v}_1}]$
$\tilde{y}$
y
$\tilde{\mathbf{v}}_N$
$[\frac{\partial y}{\partial \mathbf{v}_N}]$
$\mathbf{v}_N$

Checkpointing Ensembles (cont.)



```
template<typename ATYPE>
void g(int m, const ATYPE& x, const double& r, ATYPE& y) {
  y=1; for (int i=0;i<m; i++) y*=sin(x+r);
}

template<typename ATYPE>
void f(int n, int m, const ATYPE& x, ATYPE& y) {
  double r; ATYPE sum;
  for (int i=0;i<n; i++) { r=rand(); g(m,x,r,y); sum+=y; }
  y=sum/n;
}
```

See

- `~/checkpointing/ga1s_ensemble`
- `~/checkpointing/gt2s_ga1s_ensemble`
- memory requirement using AD_TAPE_POINTER->get_tape_memory_size()

```
template<typename ATYPE>
void g(int m, const ATYPE& x, const double& r, ATYPE& y) {
  y=1; for (int i=0;i<m; i++) y*=cos(x)/r;
}

template<typename ATYPE>
void f(int n, int m, const ATYPE& x, ATYPE& y) {
  double r; ATYPE sum;
  for (int i=0;i<n; i++) { r=rand(); g(m,x,r,y); sum+=y; }
  y=sum/n;
}
```

Compute first derivative of output `y` wrt. input `x` for given inputs `n`, `m`, and `x` by

- central finite differences,
- tangent 1st-order scalar code,
- adjoint 1st-order scalar code without checkpointing ensemble,
- adjoint 1st-order scalar code with checkpointing ensemble.

Data Flow Reversal Problem

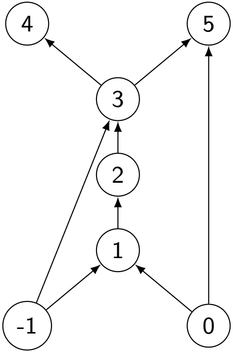
Data Flow Reversal Problem

For given amount of memory find distribution of checkpoints s.th. overall run time of adjoint code is minimzed.

Data Flow Reversal Problem: DAG Reversal

$t = x_0 \cdot \sin(x_0 \cdot x_1)$
$x_0 = \cos(t)$
$x_1 = t/x_1$

$v_{-1} = x_0$
$v_0 = x_1$
$v_1 = v_{-1} \cdot v_0$
$v_2 = \sin(v_1)$
$v_3 = v_{-1} \cdot v_2$
$v_4 = \cos(v_3)$
$v_5 = v_3/v_0$
$x_0 = v_4$
$x_1 = v_5$



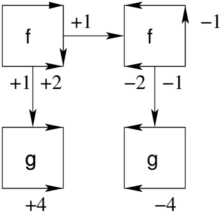
$G = (V, E)$

Wanted: $v_5, v_4, v_3, v_2, v_1, v_0, v_{-1}$

e.g. for debugging or adjoints

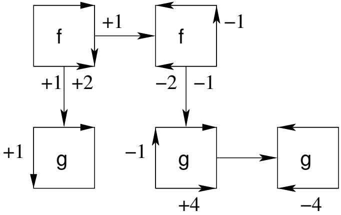
DAG REVERSAL is NP-complete. [4]

Data Flow Reversal Problem: Call Tree Reversal



(f,g,0)

MEM=8, OPS=7



(f,g,1)

MEM=6, OPS=10

The computational cost of a reversal scheme $R = R(T)$ for a call tree $T = (N, A)$ is defined by

1. the maximum amount of memory consumed in addition to the memory requirement of the original program, denoted by $\mathrm{MEM}(R)$
2. the number of arithmetic operations performed in addition to those required for recording, denoted by $\mathrm{OPS}(R)$

The choice between split and joint reversal is made for each edge individually. Consequently, the call tree $T = (N, A)$ given as

```
|_ f
   |_ g
      |_ h
```

yields a total of four *data flow reversal schemes* $R_j \subsetneq A \times \{0, 1\}$, $j = 1, \ldots, 4$.

The reversal of a call to $\mathtt{g}$ inside of $\mathtt{f}$ in split [joint] mode is denoted as $(\mathtt{f}, \mathtt{g}, 0)$ $[(\mathtt{f}, \mathtt{g}, 1)]$.

A subroutine $\mathtt{f}$ is separated into $\mathtt{f}_0, \ldots, \mathtt{f}_k$ if it contains k subroutine calls.

$\text{MEM}(\mathtt{f}_i)$ denotes the memory required to record $\mathtt{f}_i$ for $i = 0, \ldots, k$. The computational cost of running $\mathtt{f}_i$ is denoted by $\text{OPS}(\mathtt{f}_i)$. We set $\text{MEM}(\mathtt{f}) = \sum_{i=0}^{k} \text{MEM}(\mathtt{f}_i)$ and $\text{OPS}(\mathtt{f}) = \sum_{i=0}^{k} \text{OPS}(\mathtt{f}_i)$.

The memory occupied by an input checkpoint of $\mathtt{f}$ is denoted by $\text{MEM}(\mathbf{x}^{\mathtt{f}})$.

```
|_ a1_f(RECORD)
|      |_ a1_g(RECORD)
|            |_ a1_h(RECORD)
|_ a1_f(ADJOIN)
       |_ a1_g(ADJOIN)
             |_ a1_h(ADJOIN)
```

$$\mathrm{MEM}(R_1) = \mathrm{MEM}(\mathrm{f}) + \mathrm{MEM}(\mathrm{g}) + \mathrm{MEM}(\mathrm{h})$$
$$\mathrm{OPS}(R_1) = \mathrm{OPS}(\mathrm{f}) + \mathrm{OPS}(\mathrm{g}) + \mathrm{OPS}(\mathrm{h})$$

Call Tree Reversal

Joint over Split Reversal: $R_2 = ((\mathrm{f}, \mathrm{g}, 1), (\mathrm{g}, \mathrm{h}, 0))$


```
|_ a1_f(RECORD)
|        |_ a1_g(STORE_INPUTS)
|        |_ g
|             |_ h
|_ a1_f(ADJOIN)
         |_ a1_g(RESTORE_INPUTS)
         |_ a1_g(RECORD)
         |        |_ a1_h(RECORD)
         |_ a1_g(ADJOIN)
                  |_ a1_h(ADJOIN)
```

$$\mathrm{MEM}(R_2) = \max \begin{Bmatrix} \mathrm{MEM}(\mathrm{f}) + \mathrm{MEM}(\mathbf{x}^{\mathrm{g}}) \\ \mathrm{MEM}(\mathrm{f}_0) + \mathrm{MEM}(\mathrm{g}) + \mathrm{MEM}(\mathrm{h}) \end{Bmatrix}$$

$$\mathrm{OPS}(R_2) = \mathrm{OPS}(\mathrm{f}) + 2 \cdot (\mathrm{OPS}(\mathrm{g}) + \mathrm{OPS}(\mathrm{h}))$$

Call Tree Reversal

Split over Joint Reversal: $R_3 = ((\mathrm{f}, \mathrm{g}, 0), (\mathrm{g}, \mathrm{h}, 1))$


```
|_ a1_f(RECORD)
|      |_ a1_g(RECORD)
|             |_ a1_h(STORE_INPUTS)
|_ a1_f(ADJOIN)
       |_ a1_g(ADJOIN)
              |_ a1_h(RESTORE_INPUTS)
              |_ a1_h(RECORD)
              |_ a1_h(ADJOIN)
```

$$\mathrm{MEM}(R_3) = \max \left\{ \begin{matrix} \mathrm{MEM}(\mathrm{f}) + \mathrm{MEM}(\mathrm{g}) + \mathrm{MEM}(\mathbf{x}^{\mathrm{h}}) \\ \mathrm{MEM}(\mathrm{f}_0) + \mathrm{MEM}(\mathrm{g}_0) + \mathrm{MEM}(\mathrm{h}) \end{matrix} \right\}$$

$$\mathrm{OPS}(R_3) = \mathrm{OPS}(\mathrm{f}) + \mathrm{OPS}(\mathrm{g}) + 2 \cdot \mathrm{OPS}(\mathrm{h})$$

```
|_ a1_f(RECORD)
|        |_ a1_g(STORE_INPUTS)
|        |_ g
|             |_ h
|_ a1_f(ADJOIN)
         |_ a1_g(RESTORE_INPUTS)
         |_ a1_g(RECORD)
         |        |_ a1_h(STORE_INPUTS)
         |        |_ h
         |_ a1_g(ADJOIN)
                  |_ a1_h(RESTORE_INPUTS)
                  |_ a1_h(RECORD)
                  |_ a1_h(ADJOIN)
```

$$\mathrm{MEM}(R_4) = \max \left\{ \begin{array}{l} \mathrm{MEM}(\mathrm{f}) + \mathrm{MEM}(\mathbf{x}^{\mathrm{g}}) \\ \mathrm{MEM}(\mathrm{f}_0) + \mathrm{MEM}(\mathrm{g}) + \mathrm{MEM}(\mathbf{x}^{\mathrm{h}}) \\ \mathrm{MEM}(\mathrm{f}_0) + \mathrm{MEM}(\mathrm{g}_0) + \mathrm{MEM}(\mathrm{h}) \end{array} \right\}$$

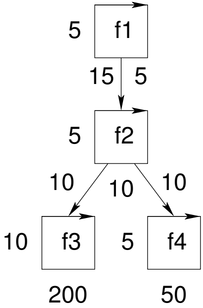
$$\mathrm{OPS}(R_4) = \mathrm{OPS}(\mathrm{f}) + 2 \cdot \mathrm{OPS}(\mathrm{g}) + 3 \cdot \mathrm{OPS}(\mathrm{h})$$

The Call Tree Reversal problem aims to determine for a given call tree $T = (N, A)$ and an integer $K > 0$ a reversal scheme $R \subsetneq A \times \{0, 1\}$ such that $\text{OPS}(R) \rightarrow \min$ subject to $\text{MEM}(R) \leq K$.

Call Tree Reversal is NP-complete. [3]

Ongoing research investigates heuristics for determining a near-optimal reversal scheme in preferably linear time.

Consider the following annotated call tree



and let the available memory be of size 250. Fully split reversal becomes infeasible. Fully joint reversal is an option, but can we do better?

Symbolic Differentiation of Numerical Methods

$$
\begin{tikzcd}
\boxed{x^2 - p = 0} \arrow[r, "\frac{d}{dp} \cdot p^{(1)}"] \arrow[d, "x^* = S(x^0, p)"'] \arrow[dr, "\bar{x}^{(1)} = S^{(1)}(x^0, p, p^{(1)})"] & \boxed{\left(2 \cdot x \cdot \frac{dx}{dp} - 1\right) \cdot p^{(1)} = 2 \cdot x \cdot \frac{dx}{dp} \cdot p^{(1)} - p^{(1)} = 2 \cdot x \cdot x^{(1)} - p^{(1)} = 0} \arrow[d, "\tilde{x}^{(1)} = \tilde{S}(x^*, p^{(1)})"] \\
\boxed{x^* \approx \sqrt{p}} \arrow[r, dotted] & \boxed{x^{(1)} = \frac{dx}{dp} \cdot p^{(1)} = \frac{p^{(1)}}{2 \cdot \sqrt{p}} = \frac{p^{(1)}}{2 \cdot x} \approx \bar{x}^{(1)} \approx \tilde{x}^{(1)}}
\end{tikzcd}
$$

Let $x = e^{\frac{3p}{2}}$ implemented as

$$\begin{aligned} p &:= e^p \\ x &:= S(x,p) \quad // \text{ Newton applied to } x^2 - p = 0 \\ x &:= x \cdot p \end{aligned}$$

Differentiation of $x^2 - p = 0$ with respect to p yields

$$\frac{d(x^2-p)}{dp} \cdot p^{(1)} = \left(2 \cdot x \cdot \frac{dx}{dp} - 1\right) \cdot p^{(1)} = 2 \cdot x \cdot x^{(1)} - p^{(1)} = 0$$

for given $x, p^{(1)} \in \mathbb{R}$ and wanted $x^{(1)}$.

Evaluation of the entire tangent code for $p^{(1)} = 1$ and $p = 5$ yields $x = 1808.04\ldots$ and $x^{(1)} = 2712.06\ldots$.

```
template<class T>
void g(const T& p, T& x) {
  const T eps=1e-3;
  T x_old=x+1;
  while (fabs(x-x_old)>eps) {
    x_old=x;
    x=x_old-(x_old*x_old-p)/(2*x_old);
  }
}

template<class T>
void f(T p, T& x) {
  p=exp(p);
  g(p,x);
  x=x*p;
}
```

See

- ~/numerics_symbolically/gt1s_user_defined_tangent

$$
\begin{tikzcd}
\boxed{x^2 - p = 0} \arrow[r, "x_{(1)} \cdot \frac{d}{dp}"] \arrow[d, "x^* = S(x^0, p)"] \arrow[dr, "\bar{p}_{(1)} = S_{(1)}(x^0, x_{(1)}, p)"] & \boxed{x_{(1)} \cdot \left(2 \cdot x \cdot \frac{dx}{dp} - 1\right) = 2 \cdot x \cdot x_{(1)} \cdot \frac{dx}{dp} - x_{(1)} = 2 \cdot x \cdot p_{(1)} - x_{(1)} = 0} \arrow[d, "\tilde{p}_{(1)} = \tilde{S}(p^0_{(1)}, x^*, x_{(1)})"] \\
\boxed{x^* \approx \sqrt{p}} \arrow[r, dotted] & \boxed{p_{(1)} = p_{(1)} + x_{(1)} \cdot \frac{dx}{dp} = \frac{x_{(1)}}{2 \cdot \sqrt{p}} = \frac{x_{(1)}}{2 \cdot x} \approx \bar{p}_{(1)} \approx \tilde{p}_{(1)}}
\end{tikzcd}
$$

Let $x = e^{\frac{3p}{2}}$ implemented as

$$
\begin{aligned}
p &:= e^p \\
x &:= S(x, p) \quad // \text{ Newton applied to } x^2 - p = 0 \\
x &:= x \cdot p
\end{aligned}
$$

Symbolic adjoint differentiation yields

$$
\begin{aligned}
p_{(1)} &= p_{(1)} + x_{(1)} \cdot \frac{\partial x}{\partial p} \\
&= p_{(1)} + x_{(1)} \cdot \frac{3}{2} \cdot e^{\frac{3p}{2}} .
\end{aligned}
$$

Evaluation for $x_{(1)} = 1$, $p_{(1)} = 0$, and $p = 5$ yields $x = 1808.04\ldots$ and $p_{(1)} = 2712.06\ldots$.

```
template<class T>
void g(const T& p, T& x) {
  const T eps=1e-3;
  T x_old=x+1;
  while (fabs(x-x_old)>eps) {
    x_old=x;
    x=x_old-(x_old*x_old-p)/(2*x_old);
  }
}

template<class T>
void f(T p, T& x) {
  p=exp(p);
  g(p,x);
  x=x*p;
}
```

See

- `~/dco/ga1s_user_defined_adjoint`

Symbolic Differentiation of Linear Solvers

Consider

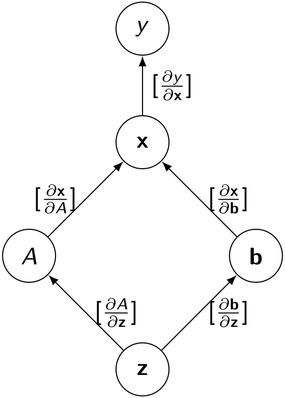
$$(A, \mathbf{b}) := P(\mathbf{z}); \; \mathbf{x} := S(A, \mathbf{b}); \; y := p(\mathbf{x}),$$

where $A \cdot \mathbf{x} = \mathbf{b}$ and S denotes a linear solver.

Context: NLP

$$\min_{\mathbf{z} \in \mathbb{R}^m} f(\mathbf{z})$$

requires $\nabla f(\mathbf{z})$.



$$\begin{pmatrix} A \\ \mathbf{b} \end{pmatrix} := P(\mathbf{z})$$

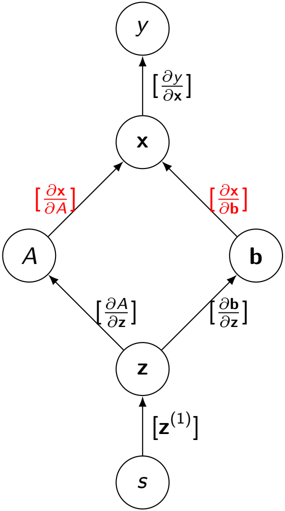
$$\begin{pmatrix} A^{(1)} \\ \mathbf{b}^{(1)} \end{pmatrix} := \begin{pmatrix} < \frac{\partial A}{\partial \mathbf{z}}, \mathbf{z}^{(1)} > \\ < \frac{\partial \mathbf{b}}{\partial \mathbf{z}}, \mathbf{z}^{(1)} > \end{pmatrix}$$

$$\mathbf{x} := S(A, \mathbf{b})$$

$$\mathbf{x}^{(1)} :=< \frac{\partial \mathbf{x}}{\partial A}, A^{(1)} > + < \frac{\partial \mathbf{x}}{\partial \mathbf{b}}, \mathbf{b}^{(1)} >$$

$$y := p(\mathbf{x})$$

$$y^{(1)} :=< \frac{\partial y}{\partial \mathbf{x}}, \mathbf{x}^{(1)} > .$$



A discrete tangent version of a direct linear solver induces computational overhead of $O(n^3)$. One can show [6] that

1. $A \cdot < \frac{\partial \mathbf{x}}{\partial \mathbf{b}}, \mathbf{b}^{(1)} >= \mathbf{b}^{(1)}$
2. $A \cdot < \frac{\partial \mathbf{x}}{\partial A}, A^{(1)} >= -A^{(1)} \cdot \mathbf{x}$

yielding a reduction of the computational overhead to $O(n^2)$ if the available factorization of A is reused.

For classical LU decomposition we get

$$
\begin{aligned}
&\dots \\
(\mathbf{x}, L, U) &= S(A, \mathbf{b}) \\
\mathbf{x}^{(1)} &= B(U, F(L, \mathbf{b}^{(1)})) + B(U, F(L, -\mathbf{x}^T \cdot A^{(1)})) \\
&\dots
\end{aligned}
$$

where $F, B : \mathbb{R}^{n\cdot(n+1)/2} \times \mathbb{R}^n \to \mathbb{R}^n$ denote solvers for lower and upper triangular systems by forward and backward substitution, respectively.

$$\left(\begin{pmatrix} A \\ \mathbf{b} \end{pmatrix}, \tau_0\right) = P_\downarrow(\mathbf{z})$$

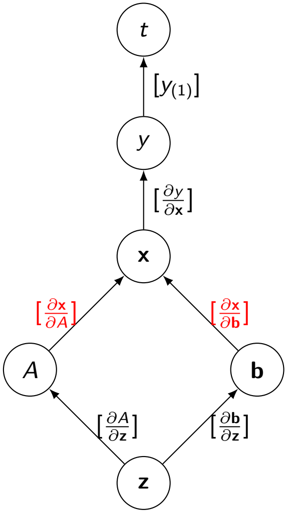
$$\mathbf{x} = S(A, \mathbf{b})$$

$$(y, \tau_2) = p_\downarrow(\mathbf{x})$$

$$\mathbf{x}_{(1)} := p_{(1)}(\tau_2, y_{(1)})$$

$$\begin{pmatrix} A_{(1)} \\ \mathbf{b}_{(1)} \end{pmatrix} := \begin{pmatrix} < \mathbf{x}_{(1)}, \frac{\partial \mathbf{x}}{\partial A} > \\ < \mathbf{x}_{(1)}, \frac{\partial \mathbf{x}}{\partial \mathbf{b}} > \end{pmatrix}$$

$$\mathbf{z}_{(1)} := P_{(1)}(\tau_0, A_{(1)}, \mathbf{b}_{(1)})$$



A discrete adjoint version of a direct linear solver induces computational and memory overheads of $O(n^3)$, respectively. One can show that [6]

1. $A^T \cdot < \mathbf{x}_{(1)}, \frac{\partial \mathbf{x}}{\partial \mathbf{b}} >= \mathbf{x}_{(1)}$
2. $< \mathbf{x}_{(1)}, \frac{\partial \mathbf{x}}{\partial A} >= -\mathbf{b}_{(1)} \cdot \mathbf{x}^T$

yielding a reduction of both the computational and memory overheads to $O(n^2)$ if the available factorization of A is reused. Memory overhead can even be reduced to $O(n)$ is unit rank of $< \mathbf{x}_{(1)}, \frac{\partial \mathbf{x}}{\partial A} >$ is exploited.

Within the forward section of the adjoint code we get

$$\begin{aligned}
&\dots \\
(\mathbf{x}, L, U) &= S(A, \mathbf{b}) \\
&\dots
\end{aligned}$$

with corresponding code in the reverse section

$$\begin{aligned}
&\dots \\
\mathbf{b}_{(1)} &= B(L^T, F(U^T, \mathbf{x}_{(1)})) \\
A_{(1)} &= -\mathbf{b}_{(1)} \cdot \mathbf{x}^T \\
&\dots
\end{aligned}$$

Again, F and B denote forward and backward substitution, respectively.

Symbolic Differentiation of Nonlinear Solvers

Consider

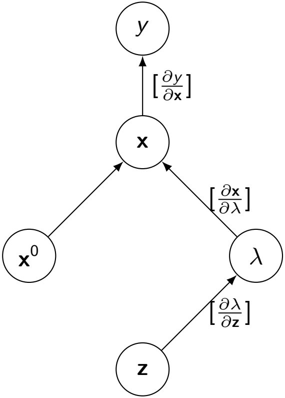
$$\lambda = P(\mathbf{z});\ \ \mathbf{x} = S(\mathbf{x}^0, \lambda);\ \ y = p(\mathbf{x})$$

where S denotes a solver for the nonlinear system $F(\mathbf{x}) = 0$

Context: NLP

$$\min_{\mathbf{z} \in \mathbb{R}^m} f(\mathbf{z})$$

requires $\nabla f(\mathbf{z})$.



Differentiation of the parameterized system of nonlinear equations $F(\mathbf{x}, \lambda) = 0$ at the solution $\mathbf{x} = \mathbf{x}^*$ with respect to the parameters λ yields

$$\frac{dF}{d\lambda}(\mathbf{x}, \lambda) = \frac{\partial F}{\partial \lambda}(\mathbf{x}, \lambda) + \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \lambda) \cdot \frac{\partial \mathbf{x}}{\partial \lambda} = 0$$

and hence

$$\frac{\partial \mathbf{x}}{\partial \lambda} = -\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \lambda)^{-1} \cdot \frac{\partial F}{\partial \lambda}(\mathbf{x}, \lambda). \tag{1}$$

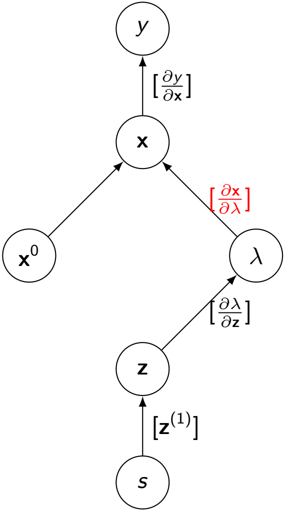
The computation of the directional derivative

$$\mathbf{x}^{(1)} = \frac{\partial \mathbf{x}}{\partial \lambda} \cdot \lambda^{(1)} = -\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \lambda)^{-1} \cdot \frac{\partial F}{\partial \lambda}(\mathbf{x}, \lambda) \cdot \lambda^{(1)} \quad (2)$$

amounts to the solution of the linear system

$$\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \lambda) \cdot \mathbf{x}^{(1)} = -\frac{\partial F}{\partial \lambda}(\mathbf{x}, \lambda) \cdot \lambda^{(1)} \quad (3)$$

the right-hand side of which can be obtained by a single evaluation of the tangent mode of F. The direct solution of (3) requires the $n \times n$ Jacobian $\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \lambda)$ which is preferably computed in tangent mode so that sparsity can be exploited.



Transposing (1) gives $\left(\frac{\partial \mathbf{x}}{\partial \lambda}\right)^T = -\left(\frac{\partial F}{\partial \lambda}\right)^T \cdot \left(\frac{\partial F}{\partial \mathbf{x}}\right)^{-T}$ and hence

$$\lambda_{(1)} := \lambda_{(1)} + \left(\frac{\partial \mathbf{x}}{\partial \lambda}\right)^T \cdot \mathbf{x}_{(1)} = \lambda_{(1)} - \frac{\partial F}{\partial \lambda}(\mathbf{x}, \lambda)^T \cdot \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \lambda)^{-T} \cdot \mathbf{x}_{(1)}. \tag{4}$$

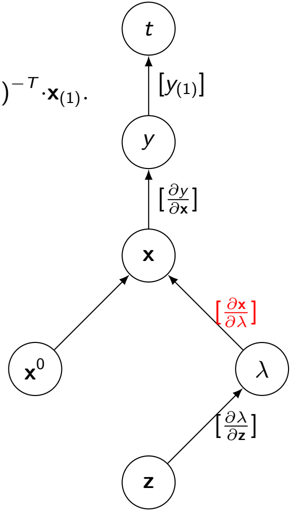
Consequently the adjoint solver needs to solve the linear system

$$\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \lambda)^T \cdot \mathbf{z} = -\mathbf{x}_{(1)} \tag{5}$$

followed by a single call of the adjoint model of F seeded with the solution $\mathbf{z}$ which gives

$$\lambda_{(1)} = \lambda_{(1)} + \frac{\partial F}{\partial \lambda}(\mathbf{x}, \lambda)^T \cdot \mathbf{z}.$$

The approach outlined above assumes availability of the exact primal solution of the nonlinear system.
See [8] for second order.



Symbolic Differentiation of Nonlinear Optimizers

Consider

$$\lambda = P(\mathbf{z});\ \ \mathbf{x} = S(\mathbf{x}^0, \lambda);\ \ y = p(\mathbf{x})$$

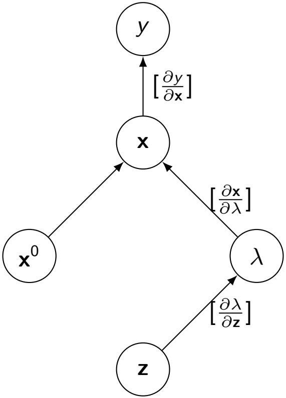
where S denotes a solver for the NLP

$$\min_{\mathbf{x}\in\mathbb{R}^n} F(\mathbf{x})$$

Context: NLP

$$\min_{\mathbf{z}\in\mathbb{R}^m} f(\mathbf{z})$$

requires $\nabla f(\mathbf{z})$.



An unconstrained convex optimisation problem can be regarded as root finding for the first-order optimality condition $\frac{\partial}{\partial \mathbf{x}} F(\mathbf{x}, \lambda) = 0$. Differentiating at the solution $\mathbf{x} = \mathbf{x}^*$ with respect to λ gives

$$\frac{\partial}{\partial \lambda} \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \lambda) = \frac{\partial^2 F}{\partial \lambda \partial \mathbf{x}}(\mathbf{x}, \lambda) + \frac{\partial^2 F}{\partial \mathbf{x}^2}(\mathbf{x}, \lambda) \cdot \frac{\partial \mathbf{x}}{\partial \lambda}(\mathbf{x}, \lambda) = 0$$

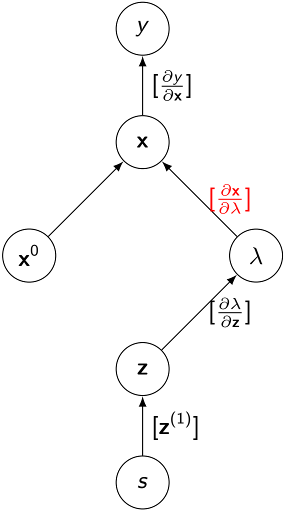
so that we obtain

$$\frac{\partial \mathbf{x}}{\partial \lambda}(\mathbf{x}, \lambda) = -\frac{\partial^2 F}{\partial \mathbf{x}^2}(\mathbf{x}, \lambda)^{-1} \cdot \frac{\partial^2 F}{\partial \lambda \partial \mathbf{x}}(\mathbf{x}, \lambda). \tag{6}$$

Computing the directional derivative $\mathbf{x}^{(1)} = \frac{\partial \mathbf{x}}{\partial \lambda} \cdot \lambda^{(1)}$ amounts to solving a linear system

$$-\frac{\partial^2 F}{\partial \mathbf{x}^2}(\mathbf{x}, \lambda) \cdot \mathbf{x}^{(1)} = \frac{\partial^2 F}{\partial \lambda \partial \mathbf{x}}(\mathbf{x}, \lambda) \cdot \lambda^{(1)} \qquad (7)$$

the right-hand side of which can be obtained by a single evaluation of the second-order adjoint version of F. The direct solution of (7) requires the $n \times n$ Hessian $\frac{\partial^2 f}{\partial \mathbf{x}^2}(\mathbf{x}, \lambda)$, which is preferably computed with second-order adjoint AD while exploiting potential sparsity.



Transposing (6) gives $\left(\frac{\partial \mathbf{x}}{\partial \lambda}\right)^T = -\left(\frac{\partial^2 F}{\partial \lambda \partial \mathbf{x}}\right)^T \cdot \left(\frac{\partial^2 F}{\partial \mathbf{x}^2}\right)^{-T}$ yielding the first-order adjoint model

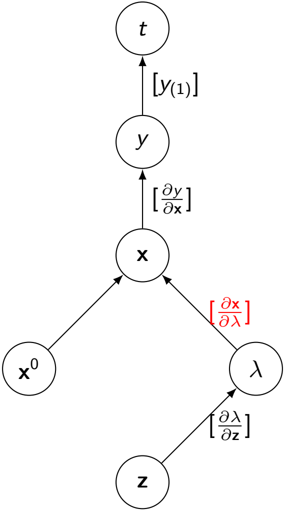
$$\begin{aligned}\lambda_{(1)} &:= \lambda_{(1)} + \frac{\partial \mathbf{x}}{\partial \lambda}(\mathbf{x}, \lambda)^T \cdot \mathbf{x}_{(1)} \\ &= \lambda_{(1)} - \frac{\partial^2 F}{\partial \lambda \partial \mathbf{x}}(\mathbf{x}, \lambda)^T \cdot \frac{\partial^2 F}{\partial \mathbf{x}^2}(\mathbf{x}, \lambda)^{-T} \cdot \mathbf{x}_{(1)}.\end{aligned}$$

Computing $\lambda_{(1)}$ amounts to solving the linear system $\frac{\partial^2 F}{\partial \mathbf{x}^2}^T \cdot \mathbf{z} = -\mathbf{x}_{(1)}$ followed by a single call of the second-order adjoint model of F at the solution $\mathbf{z}$ to obtain

$$\lambda_{(1)} := \lambda_{(1)} + \frac{\partial^2 F}{\partial \lambda \partial \mathbf{x}}(\mathbf{x}, \lambda)^T \cdot \mathbf{z}.$$

Generalizations for constrained optimization problems can be derived naturally, e.g. by treating the KKT[1] system as above.

[1] Karush-Kuhn-Tucker


t
$[y_{(1)}]$
y
$[\frac{\partial y}{\partial \mathbf{x}}]$
$\mathbf{x}$
$[\frac{\partial \mathbf{x}}{\partial \lambda}]$
$\mathbf{x}^0$
λ
$[\frac{\partial \lambda}{\partial \mathbf{z}}]$
$\mathbf{z}$

Preaccumulation

```
template<typename ATYPE>
void g( int n, ATYPE& x ) {
  for (int i=0;i<n;i++) x=sin(x);
}

template<typename ATYPE>
void f( int n, ATYPE& x ) {
  g(n/3,x);
  g(n/3,x); // preaccumulate in tangent mode
  g(n-n/3*2,x);
}
```

See

- ~/preaccumulation/ga1s_preaccumulation
- ~/preaccumulation/gt2s_ga1s_preaccumulation

Embedded Adjoint Source Code

```
template<typename AD_TYPE>
void g(std::vector<AD_TYPE> x, AD_TYPE& y)
{
  y=0;
  for (int i=0;i<x.size();i++) y+=x[i];
}

template<typename AD_TYPE>
void f(typename std::vector<AD_TYPE>& x, AD_TYPE& y) {
  for (int i=0;i<x.size();i++) x[i]*=x[i];
  g(x,y);
  y*=y;
}
```

See

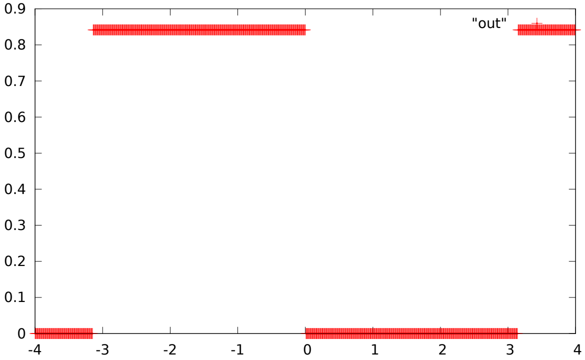
- ~/embedded_source_trafo/ga1s_external_manual
- ~/embedded_source_trafo/gt2s_ga1s_external_manual

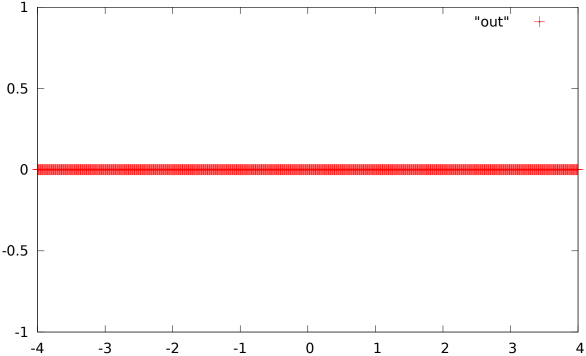
Finite Difference Smoothing

Finite differences on Heavyside step function:

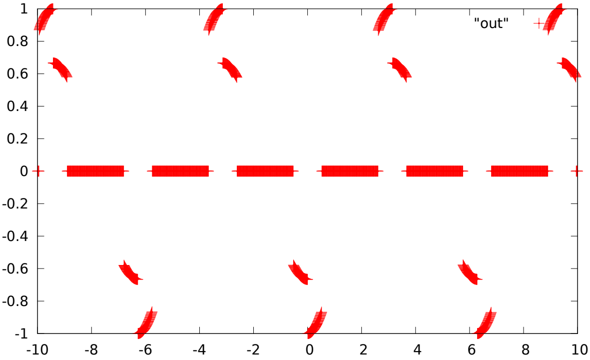
```
template<class T>
void g(T& x) {
  if (x>0)
    x=0;
  else
    x=1;
}

template<class T>
void f(T& x) {
  x=sin(x); g(x); x=sin(x);
}
```

- `~/fd_smoothing/gt1s_fd_smoothing`


"out"
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
-4
-3
-2
-1
0
1
2
3
4



"out"
1
0.5
0
-0.5
-1
-4
-3
-2
-1
0
1
2
3
4


"out"
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-10
-8
-6
-4
-2
0
2
4
6
8
10

C. Bischof, M. Bücker, P. Hovland, U. Naumann, and J. Utke, editors.
Advances in Automatic Differentiation, number 64 in Lecture Notes in Computational Science and Engineering (LNCSE). Springer, 2008.

A. Griewank and A. Walther.
Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.
Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition, 2008.

U. Naumann.
Call tree reversal is NP-complete.
In *[1]*, pages 13–22. Springer, 2008.

U. Naumann.
DAG reversal is NP-complete.
Journal of Discrete Algorithms, 7:402–410, 2009.

U. Naumann.
The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation.
Number 24 in Software, Environments, and Tools. SIAM, Philadelphia, PA, 2012.

U. Naumann and J. Lotz.
Algorithmic differentiation of numerical methods: Tangent-linear and adjoint direct solvers for systems of linear equations.
Technical Report AIB-2012-10, RWTH Aachen University, 2012.

U. Naumann, J. Lotz, K. Leppkes, and M. Towara.
Algorithmic differentiation of numerical methods: Tangent-linear and adjoint solvers for systems of nonlinear equations.
Technical Report AIB-2012-15, RWTH Aachen University, 2012.

N. Safiran, J. Lotz, and U. Naumann.
Algorithmic differentiation of numerical methods: Second-order tangent and adjoint solvers for systems of parametrized nonlinear equations.
Technical Report AIB-2014-07, RWTH Aachen University, 2014.