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

Uwe Naumann

Software and Tools for Computational Engineering
Computer Science, RWTH Aachen University
Aachen, Germany
naumann@stce.rwth-aachen.de
www.stce.rwth-aachen.de

and

The Numerical Algorithms Group Ltd.
Oxford, United Kingdom
Uwe.Naumann@nag.co.uk
www.nag.co.uk
Contents

Motivation

Mind the Gap

Checkpointing
  Checkpointing Evolutions
  Checkpointing Ensembles
  Data Flow Reversal Problem

Symbolic Differentiation of Numerical Methods
  Symbolic Differentiation of Linear Solvers
  Symbolic Differentiation of Nonlinear Solvers
  Symbolic Differentiation of Nonlinear Optimizers

Preaccumulation

Embedded Adjoint Source Code

Finite Difference Smoothing
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 in the Chain Rule

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

\[
\frac{\partial y}{\partial x} = \frac{\partial f_2}{\partial v} \cdot \frac{\partial g}{\partial u} \cdot \frac{\partial f_1}{\partial x}
\]
Mind the Gap in the Chain Rule

... for Adjoint of \((y, \tilde{y}) = f_2(g(f_1(x, \tilde{x})))\)
Checkpointing
Live: Derivative of $10^9$ iterations of

\[ x = \sin(x \cdot p) \]

by

\[ \sim /checkpointing/gals_split/ \]

\[ \Rightarrow \text{std::bad_alloc} \]
Call Reversal Modes
Split [2]

\[
\begin{align*}
\frac{\partial x_i}{\partial x_{i-1}} &= (\text{MEM}, \text{OPS}) \\
(1, 1) &\leftarrow \text{push}(x_0); \text{compute}(x_1) \\
(2, 2) &\leftarrow \text{push}(x_1); \text{compute}(x_2) \\
(3, 3) &\leftarrow \text{push}(x_2); \text{compute}(x_3) \\
(4, 4) &\leftarrow \text{push}(x_3); \text{compute}(x_4) \\
(5, 5) &\leftarrow \text{push}(x_4); \text{compute}(x_5) \\
(6, 6) &\leftarrow \text{push}(x_5); \text{compute}(x_6) \\
(7, 7) &\leftarrow \text{push}(x_6); \text{compute}(x_7) \\
(8, ) &\leftarrow \text{save}(x_7) \\
(7, ) &\leftarrow \text{pop}(x_6); \text{adjoint}(x_6) \\
(6, ) &\leftarrow \text{pop}(x_5); \text{adjoint}(x_5) \\
(5, ) &\leftarrow \text{pop}(x_4); \text{adjoint}(x_4) \\
(4, ) &\leftarrow \text{pop}(x_3); \text{adjoint}(x_3) \\
(3, ) &\leftarrow \text{pop}(x_2); \text{adjoint}(x_2) \\
(2, ) &\leftarrow \text{pop}(x_1); \text{adjoint}(x_1) \\
(1, ) &\leftarrow \text{pop}(x_0); \text{adjoint}(x_0) \\
(0, ) &\leftarrow \text{recover}(x_7)
\end{align*}
\]
Call Reversal Modes

Joint [2]

\[
\frac{\partial x_i}{\partial x_{i-1}}
\]

\[
\frac{\partial x_i}{\partial p}
\]

(MEM,OPS)

\(1, 1 \leftarrow \text{push}(x_0); \text{compute}(x_1)\)

\(2, 5 \leftarrow \text{store}(x_1); \text{compute}(x_5)\)

\(3, 6 \leftarrow \text{push}(x_5); \text{compute}(x_6)\)

\(4, 7 \leftarrow \text{push}(x_6); \text{compute}(x_7)\)

\(5, \) \(\leftarrow \text{save}(x_7)\)

\(4, \) \(\leftarrow \text{pop}(x_6); \text{adjoint}(x_6)\)

\(3, \) \(\leftarrow \text{pop}(x_5); \text{adjoint}(x_5)\)

\(2, \) \(\leftarrow \text{restore}(x_1)\)

\(3, 8 \leftarrow \text{push}(x_1); \text{compute}(x_2)\)

\(4, 9 \leftarrow \text{push}(x_2); \text{compute}(x_3)\)

\(5, 10 \leftarrow \text{push}(x_3); \text{compute}(x_4)\)

\(6, 11 \leftarrow \text{push}(x_4); \text{compute}(x_5)\)

\(5, \) \(\leftarrow \text{pop}(x_4); \text{adjoint}(x_4)\)

\(4, \) \(\leftarrow \text{pop}(x_3); \text{adjoint}(x_3)\)

\(3, \) \(\leftarrow \text{pop}(x_2); \text{adjoint}(x_2)\)

\(2, \) \(\leftarrow \text{pop}(x_1); \text{adjoint}(x_1)\)

\(1, \) \(\leftarrow \text{pop}(x_0); \text{adjoint}(x_0)\)

\(0, \) \(\leftarrow \text{recover}(x_7)\)
Call Reversal Modes
Alternative View

\[(f,g,0)\]
MEM=8, OPS=7

\[(f,g,1)\]
MEM=6, OPS=10


[... read pre-order depth-first from left to right]
Mind the Gap in the Tape

... for Adjoint of \((y, \tilde{y}) = f_2(g(f_1(x, \tilde{x})))\)
Mind the Gap in the Tape

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

A gap \((v', \tilde{v}) = g(u', \tilde{u})\) 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 \(u\) in order to write \(u(1) := u(1) + u'_1\) correctly;
- adjoint gap input checkpoint \(\subset (u, \tilde{u}, v, \tilde{v})\) in order to initialize interpretation of the gap correctly;
- tape location of active gap outputs \(v\) in order to initialize \(v'(1) := v(1)\) 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 \(u(1)\) with \(\left(\frac{\partial v}{\partial u}\right)^T \cdot v(1)\).
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

\[ f_1 \]

\[ \tilde{x} \rightarrow \tilde{u} \rightarrow u \]

\[ \frac{\partial u}{\partial x} \]

\[ u' \rightarrow v'_1 \rightarrow v_1 \]

\[ g\_make\_gap \]

\[ u'_1 \]

\[ g\_make\_gap \]

\[ u'_N \rightarrow v'_N \rightarrow v_N \]

\[ \frac{\partial v'_N}{\partial u'_N} \]

\[ g\_fill\_gap \]

\[ x(1) \rightarrow u(1) \rightarrow v(1) \rightarrow y(1) \]

\[ \frac{\partial u}{\partial x} \]

\[ \frac{\partial v'_1}{\partial u'_1} \]

\[ \frac{\partial v'_N}{\partial u'_N} \]
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,

\[
\begin{align*}
    x_0 &= x \\
    x_2 &= F_2(F_1(x))
\end{align*}
\]

\[
\nabla F(x)^T \cdot y(1) = \nabla F_1(x_0)^T \cdot \left( \nabla F_2(F_1(x_0))^T \cdot \left( \nabla F_3(x_2)^T \cdot \left( \nabla F_4(F_3(x_2))^T \cdot y(1) \right) \right) \right)
\]

yields

\[
\begin{align*}
    OPS(F_{(1)}) &\approx \nu \cdot OPS(F) + 2 \cdot OPS(F_i) \\
    MEM(F_{(1)}) &\approx 2 \cdot MEM(F) + OPS(F_i)
\end{align*}
\]
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.
Theorem

1. never push checkpoint for left-most subtree at recursion level \( \geq 1 \)

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()
```cpp
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
Checkpointing Ensembles

\[f_1\]

\[g_{\text{make_gap}}\]

\[f_2\]

\[\frac{\partial y}{\partial v_1}\]

\[\frac{\partial y}{\partial v_N}\]
Checkpointing Ensembles (cont.)

\[
\begin{align*}
\mathbf{u}'_1(1) & \quad [\partial v'_1 / \partial u'_1] \\
\mathbf{v}'_1(1) & \quad [1] \\
\mathbf{v}'_N(1) & \quad [\partial v'_N / \partial u'_N] \\
\mathbf{v}_N(1) & \quad [1] \\
\mathbf{x}(1) & \quad [\partial u / \partial x] \\
\mathbf{u}(1) & \quad [1] \\
\mathbf{y}(1) & \quad [\partial y / \partial v_1] \\
\mathbf{y}(1) & \quad [\partial y / \partial v_N]
\end{align*}
\]
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
For given amount of memory find distribution of checkpoints s.th. overall run time of adjoint code is minimized.
Data Flow Reversal Problem: DAG Reversal
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
Call Tree Reversal
Abstract Perspective on Checkpointing

(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 $\text{MEM}(R)$

2. the number of arithmetic operations performed in addition to those required for recording, denoted by $\text{OPS}(R)$

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* $R_j \subseteq A \times \{0, 1\}$, $j = 1, \ldots, 4$. 
The reversal of a call to $g$ inside of $f$ in split [joint] mode is denoted as $(f, g, 0) [ (f, g, 1) ]$.

A subroutine $f$ is separated into $f_0, \ldots, f_k$ if it contains $k$ subroutine calls.

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

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

Fully Split Reversal: \( R_1 = \{ (f, g, 0), (g, h, 0) \} \)

\[
\begin{align*}
\text{MEM}(R_1) &= \text{MEM}(f) + \text{MEM}(g) + \text{MEM}(h) \\
\text{OPS}(R_1) &= \text{OPS}(f) + \text{OPS}(g) + \text{OPS}(h)
\end{align*}
\]
Call Tree Reversal
Joint over Split Reversal: $R_2 = ((f, g, 1), (g, h, 0))$

\[
\text{MEM}(R_2) = \max \left\{ \text{MEM}(f) + \text{MEM}(x^g), \text{MEM}(f_0) + \text{MEM}(g) + \text{MEM}(h) \right\}
\]

\[
\text{OPS}(R_2) = \text{OPS}(f) + 2 \cdot (\text{OPS}(g) + \text{OPS}(h))
\]
Call Tree Reversal

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

\[
\begin{align*}
| & a1.f(RECORD) \\
| & \quad | & a1.g(RECORD) \\
| & \quad | & a1.h(STORE_INPUTS) \\
| & a1.f(ADJOIN) \\
| & \quad | & a1.g(ADJOIN) \\
| & \quad | & a1.h(RESTORE_INPUTS) \\
| & \quad | & a1.h(RECORD) \\
| & \quad | & a1.h(ADJOIN)
\end{align*}
\]

\[
MEM(R_3) = \max \left\{ \begin{array}{c}
MEM(f) + MEM(g) + MEM(x^h) \\
MEM(f_0) + MEM(g_0) + MEM(h)
\end{array} \right\}
\]

\[
OPS(R_3) = OPS(f) + OPS(g) + 2 \cdot OPS(h)
\]
Call Tree Reversal

Fully Joint Reversal: $R_4 = ((f, g, 1), (g, h, 1))$

```
- 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)
```

\[
MEM(R_4) = \max \left\{ \begin{array}{l}
MEM(f) + MEM(x^g) \\
MEM(f_0) + MEM(g) + MEM(x^h) \\
MEM(f_0) + MEM(g_0) + MEM(h)
\end{array} \right\}
\]

\[
OPS(R_4) = OPS(f) + 2 \cdot OPS(g) + 3 \cdot OPS(h)
\]
The **Call Tree Reversal** problem aims to determine for a given call tree \( T = (N, A) \) and an integer \( K > 0 \) a reversal scheme \( R \subseteq A \times \{0, 1\} \) such that \( \text{OPS}(R) \to \min \) 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
$x^2 - p = 0$

$x^* = S(x^0, p)$

$x^* \approx \sqrt{p}$

$\frac{d}{dp} \cdot p^{(1)}$

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

$\bar{x}^{(1)} = S^{(1)}(x^0, p, p^{(1)})$

$\tilde{x}^{(1)} = \tilde{S}(x^*, p^{(1)})$

$x^{(1)} = \frac{dx}{dp} \cdot p^{(1)} = \frac{p^{(1)}}{2 \cdot \sqrt{p}} = \frac{p^{(1)}}{2 \cdot x} \approx \bar{x}^{(1)} \approx \tilde{x}^{(1)}$
Let $x = e^{\frac{3p}{2}}$ implemented as

\[
p := e^p
\]

\[
x := S(x, p) \quad // \text{Newton applied to } x^2 - p = 0
\]

\[
x := x \cdot p
\]

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
Symbolic Differentiation of Numerical Methods
Symbolic Adjoints: Motivation [7]

\[ x^2 - p = 0 \]

\[ x^* = S(x^0, p) \]

\[ x^* \approx \sqrt{p} \]

\[ p(1) = p(1) + x(1) \cdot \frac{dx}{dp} = \frac{x(1)}{2 \cdot \sqrt{p}} = \frac{x(1)}{2 \cdot x} \approx \tilde{p}(1) \approx \bar{p}(1) \]

\[ p(1) = 2 \cdot x \cdot p(1) - x(1) = 0 \]

\[ x(1) \cdot \left(2 \cdot x \cdot \frac{dx}{dp} - 1\right) = 2 \cdot x \cdot x(1) \cdot \frac{dx}{dp} - x(1) = 2 \cdot x \cdot p(1) - x(1) = 0 \]

\[ \bar{p}(1) = S(1)(x^0, x(1), p) \]

\[ \tilde{p}(1) = S(p(1), x^*, x(1)) \]
Symbolic Differentiation of Numerical Methods
SymbolicAdjoints: Example

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

\[
p := e^p \\
\]
\[
x := S(x, p) \quad // \text{Newton applied to } x^2 - p = 0 \\
x := x \cdot p
\]

Symbolic adjoint differentiation yields

\[
p(1) = p(1) + x(1) \cdot \frac{\partial x}{\partial p} = p(1) + x(1) \cdot \frac{3}{2} \cdot e^{\frac{3p}{2}}.
\]

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

\[(A, b) := P(z); \quad x := S(A, b); \quad y := p(x),\]

where \(A \cdot x = b\) and \(S\) denotes a linear solver.

**Context: NLP**

\[
\min_{z \in \mathbb{R}^m} f(z)
\]

requires \(\nabla f(z)\).
Symbolic Differentiation of Linear Solvers

Tangents

\[
\begin{align*}
\begin{pmatrix} A \\ b \end{pmatrix} &:= P(z) \\
\begin{pmatrix} A^{(1)} \\ b^{(1)} \end{pmatrix} &:= \begin{pmatrix} \langle \frac{\partial A}{\partial z}, z^{(1)} \rangle \\ \langle \frac{\partial b}{\partial z}, z^{(1)} \rangle \end{pmatrix}
\end{align*}
\]

\[
x := S(A, b)
\]

\[
x^{(1)} := \langle \frac{\partial x}{\partial A}, A^{(1)} \rangle + \langle \frac{\partial x}{\partial b}, b^{(1)} \rangle
\]

\[
y := p(x)
\]

\[
y^{(1)} := \langle \frac{\partial y}{\partial x}, x^{(1)} \rangle.
\]
A discrete tangent version of a direct linear solver induces computational overhead of $O(n^3)$. One can show [6] that

1. $A \cdot \left< \frac{\partial x}{\partial b}, b^{(1)} \right> = b^{(1)}$

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

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

\[
\begin{align*}
(x, L, U) &= S(A, b) \\
\begin{split}
x^{(1)} &= B(U, F(L, b^{(1)})) + B(U, F(L, -x^T \cdot A^{(1)})) \\
\end{split}
\end{align*}
\]

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

Adjoint

\[
\begin{align*}
\left( \begin{array}{c}
A \\
b
\end{array} \right), \tau_0 & = P_\downarrow(z) \\
x & = S(A, b) \\
(y, \tau_2) & = p_\downarrow(x)
\end{align*}
\]

\[
\begin{align*}
\mathbf{x}^{(1)} & := p^{(1)}(\tau_2, y^{(1)}) \\
\begin{pmatrix}
A^{(1)} \\
b^{(1)}
\end{pmatrix} & := \begin{pmatrix}
< \mathbf{x}^{(1)}, \frac{\partial \mathbf{x}}{\partial A} > \\
< \mathbf{x}^{(1)}, \frac{\partial \mathbf{x}}{\partial b} >
\end{pmatrix} \\
\mathbf{z}^{(1)} & := P^{(1)}(\tau_0, A^{(1)}, b^{(1)})
\end{align*}
\]
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. $A^T \cdot < x_{(1)}, \frac{\partial x}{\partial b} > = x_{(1)}$

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

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 $O(n)$ if the unit rank of $< x_{(1)}, \frac{\partial x}{\partial A} >$ is exploited.
Within the forward section of the adjoint code we get

\[
(x, L, U) = S(A, b)
\]

with corresponding code in the reverse section

\[
\begin{align*}
  b_{(1)} &= B(L^T, F(U^T, x_{(1)})) \\
  A_{(1)} &= -b_{(1)} \cdot x^T
\end{align*}
\]

Again, \( F \) and \( B \) denote forward and backward substitution, respectively.
Symbolic Differentiation of Nonlinear Solvers
Consider

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

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

Context: NLP

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

requires $\nabla f(z)$. 
Differentiation of the parameterized system of nonlinear equations $F(x, \lambda) = 0$ at the solution $x = x^*$ with respect to the parameters $\lambda$ yields

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

and hence

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

(1)
The computation of the directional derivative

\[ x^{(1)} = \frac{\partial x}{\partial \lambda} \cdot \lambda^{(1)} = - \frac{\partial F}{\partial x}(x, \lambda)^{-1} \cdot \frac{\partial F}{\partial \lambda}(x, \lambda) \cdot \lambda^{(1)} \]  

amounts to the solution of the linear system

\[ \frac{\partial F}{\partial x}(x, \lambda) \cdot x^{(1)} = - \frac{\partial F}{\partial \lambda}(x, \lambda) \cdot \lambda^{(1)} \]  

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 \( \frac{\partial F}{\partial x}(x, \lambda) \) which is preferably computed in tangent mode so that sparsity can be exploited.
Transposing (1) gives \((\frac{\partial x}{\partial \lambda})^T = - (\frac{\partial F}{\partial \lambda})^T \cdot (\frac{\partial F}{\partial x})^{-T}\) and hence

\[
\lambda(1) := \lambda(1) + \left(\frac{\partial x}{\partial \lambda}\right)^T \cdot x(1) = \lambda(1) - \frac{\partial F}{\partial \lambda}(x, \lambda)^T \cdot \frac{\partial F}{\partial x}(x, \lambda)^{-T} \cdot x(1).
\]

Consequently the adjoint solver needs to solve the linear system

\[
\frac{\partial F}{\partial x}(x, \lambda)^T \cdot z = -x(1)
\]

followed by a single call of the adjoint model of \(F\) seeded with the solution \(z\) which gives

\[
\lambda(1) = \lambda(1) + \frac{\partial F}{\partial \lambda}(x, \lambda)^T \cdot z.
\]

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

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

where $S$ denotes a solver for the NLP

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

Context: NLP

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

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

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

so that we obtain

$$\frac{\partial x}{\partial \lambda}(x, \lambda) = -\frac{\partial^2 F}{\partial x^2}(x, \lambda)^{-1} \cdot \frac{\partial^2 F}{\partial \lambda \partial x}(x, \lambda). \quad (6)$$
Computing the directional derivative $\mathbf{x}^{(1)} = \frac{\partial \mathbf{x}}{\partial \lambda} \cdot \lambda^{(1)}$ amounts to solving a linear system

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

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

$$
\lambda(1) := \lambda(1) + \frac{\partial x}{\partial \lambda}(x, \lambda)^T \cdot x(1)
$$

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

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

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

Generalizations for constrained optimization problems can be derived naturally, e.g. by treating the KKT\(^1\) system as above.

\(^1\)Karush-Kuhn-Tucker
Symbolic Differentiation of Nonlinear Optimizers

Adjoint

\[
\begin{align*}
\frac{\partial y}{\partial x} & \\
\frac{\partial \lambda}{\partial x} & \\
\frac{\partial \lambda}{\partial z} & \\
\end{align*}
\]
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:

```cpp
// Example implementation

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
```
Finite Difference Smoothing

Example: Function Values
Finite Difference Smoothing

Example: Exact Derivatives

"out"
Finite Difference Smoothing

Example: Derivatives of Finite Difference Smoothing

![Plot of Finite Difference Smoothing](image_url)
C. Bischof, M. BÜcker, P. Hovland, U. Naumann, and J. Utke, editors. 

A. Griewank and A. Walther.  

U. Naumann.  
Call tree reversal is NP-complete.  

U. Naumann.  
DAG reversal is NP-complete.  

U. Naumann.  
*The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation*.  
U. Naumann and J. Lotz.
Algorithmic differentiation of numerical methods: Tangent-linear and adjoint direct solvers for systems of linear equations.

Algorithmic differentiation of numerical methods: Tangent-linear and adjoint solvers for systems of nonlinear equations.

N. Safiran, J. Lotz, and U. Naumann.
Algorithmic differentiation of numerical methods: Second-order tangent and adjoint solvers for systems of parametrized nonlinear equations.