AD Masterclass: Part 1

Viktor Mosenkis

The Numerical Algorithms Group

- Founded in 1970 as a co-operative project in UK
- Operates as a commercial, not-for-profit organization
  - Funded entirely by customer income

- NAG Portfolio
  - NAG Numerical Libraries
  - NAG Fortran Compiler
  - HPC services
  - Consultancy work for bespoke application development
  - Algorithmic Differentiation (AD) in collaboration with Prof. Naumann at Aachen University

AD Portfolio and AD work/consulting

- AD Portfolio
  - dco/c++ - AD tool based on operator overloading
  - AD consultancy
  - NAG AD Library (AD versions of NAG Library routines)

- AD work/consulting
  - Several Tier 1 & 2 investment banks have global licences for our AD software
  - Similar arrangements with others across automotive
  - We've helped a number of banks implement and optimise their AD codes
  - We've delivered bespoke AD training for many organisations

Remarks

- Please submit your questions via the questions panel at any time during this session, these will be addressed at the end.

- A recording of this session, along with the slides will be shared with you in a day or two.

Dialogue

We want this webinar series to be interactive (even though it's hard to do)

- We want your feedback, we want to adapt material to your feedback
- Please feel free to contact us via email to ask questions at any time
- We'd love to reach out offline, discuss what's working, what to spend more time on
- For some orgs, may make sense for us to do a few bespoke sessions

Outcomes

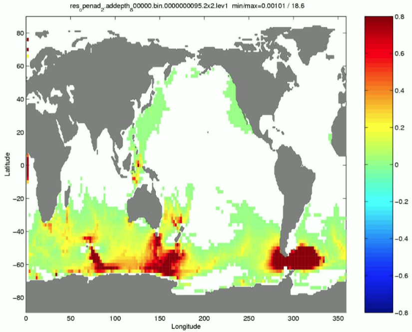
- Recall on the problems arising when computing the derivatives by finite differences (bumping)

- Introduce Algorithmic Differentiation (AD) esp. the two AD models
  - tangent-linear and
  - adjoint model

- Demonstrate how AD can help to address the problems that arise with finite differences

Real-world examples

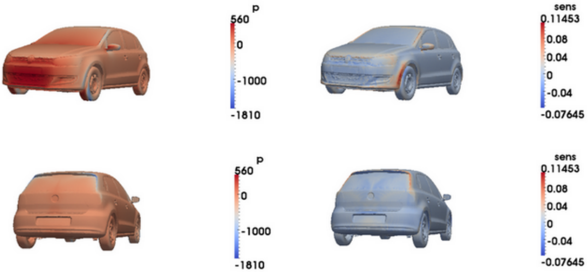
Figure shows the sensitivity of the amount of water flowing through the Drake passage to changes in the topography of the ocean floor. The simulation was performed with the AD-enabled MIT Global Circulation Model (MITgcm) run on a supercomputer. The ocean was meshed with 64,800 grid points.



Obtaining the gradient through finite differences took a month and a half. The adjoint AD code obtained the gradient in less than 10 minutes.

Real-world examples

AD enables sensitivity analyses of huge simulations, enabling shape optimization, intelligent design and comprehensive risk studies.



The figure shows sensitivities of the drag coefficient to each point on a car's surface when it moves at high speed (left) and low speed (right). The simulation was performed with AD-enabled OpenFOAM built on top of dco/c++. The normal simulation took 44s, while the AD-enabled simulation took 273s, to obtain the same gradient by finite differences would take roughly 5 years (surface mesh had 5.5 million cells).

AD in Finance

AD has been used heavily in the finance industry for many years now to compute Greeks:

- Applied to PDE and Monte Carlo codes
- Also been applied to XVA codes
- Adjoint chained backwards through calibration to get sensitivities to market instruments
- Speedups of 10x or more are common (higher the more inputs): full gradient in seconds rather than minutes. Combined with GPUs it can even give full XVA sensitivities in seconds
- Main challenge is controlling memory use

How to compute derivatives?

Typically $F : \mathbb{R}^n \to \mathbb{R}^m, \quad \boldsymbol{y} = F(\boldsymbol{x})$ is not given in closed form but rather implemented in some programming language.

Writing derivative code by hand is difficult E.g.

```
void foo(int n, double *x, double &y){
  for (int i=0; i<n; i++){
    if (i == 0)
      y = sin(x[i]);
    else
      y *= x[i];
  }
}
```

How to compute derivatives?

We first need to understand that `foo` is computing

$$y = F(\boldsymbol{x}) = \sin(x_0) \cdot \prod_{i=1}^{n-1} x_i.$$

We then need to differentiate this function

$$F'(x)^T = \begin{pmatrix} \cos(x_0) \cdot \prod_{i=1}^{n-1} x_i \\ \sin(x_0) \cdot \prod_{i=1,i\neq 1}^{n-1} x_i \\ \cdots \\ \sin(x_0) \cdot \prod_{i=1,i\neq 1}^{n-1} x_i \end{pmatrix}$$

and then implement the corresponding derivative code.

```
void d_foo(int n, double *x, double *Jac){
  double prod = 1.0;
  for (int i=1; i<n; i++) prod=x[i];
  Jac[0] = cos(x[0])*prod;
  for (int i=0; i<n; i++)
        Jac[i] = sin(x[0]) * prod / x[i];
 }
```

How to compute derivatives?

Writing derivative code by hand is very error prone job and leads to software engineering problem of maintaining two sources as any changes to function `foo` must be ported to `d_foo`.

That is why practitioners tend to use more automatic approaches to compute derivatives

- Finite differences (also know as bumping)
- Algorithmic (or Automatic) Differentiation (AD)

Finite Difference

Allows to approximate the derivative of
$F : \mathbb{R}^n \to \mathbb{R}^m, \quad \boldsymbol{y} = F(\boldsymbol{x}).$

Example (Forward finite difference Error: $O(h)$)

$$\tilde{F}(\boldsymbol{x}, \dot{\boldsymbol{x}}) = F'(\boldsymbol{x})\dot{\boldsymbol{x}} \approx \frac{F(\boldsymbol{x} + h\dot{\boldsymbol{x}}) - F(\boldsymbol{x})}{h}$$

Example (Centred finite difference Error: $O(h^2)$)

$$\mathring{F}(\boldsymbol{x}, \dot{\boldsymbol{x}}) = F'(\boldsymbol{x})\dot{\boldsymbol{x}} \approx \frac{F(\boldsymbol{x} + h\dot{\boldsymbol{x}}) - F(\boldsymbol{x} - h\dot{\boldsymbol{x}})}{2h}$$

Automatic approach, no need to maintain two sources.

Finite Difference: Problems

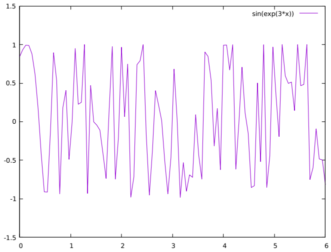
The finite differences approach suffer from two problems

- Accuracy:
  How to find a proper perturbation parameter h.

- Complexity when computing the full Jacobian of a function

Finite Difference: Example Oscillating function

```
template <typename T>
T foo(T &x){
  return sin(exp(3*x));
}
```

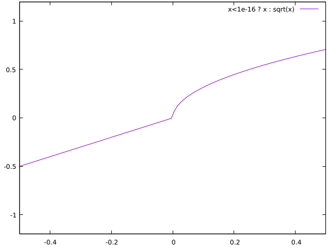


Derivative at $x = 5.5$

- with centred finite difference: -35439200.1768122
- exact: -38105558.7965109

Finite Difference: Example different code branches

```
template <typename T>
T foo(T &x){
  if (x < 100*epsilon)
    return x;
  else
    return sqrt(x);
}
```

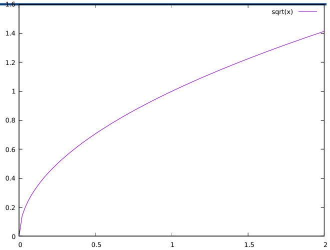


Derivative at $x = 0$

- with centered finite difference: 4096.5
- exact: 1

Finite Difference: Example highly non-linear function

```
template <typename T>
T foo(T &x){
  return sqrt(x);
}
```



Derivative at $x = 10^{-10}$ (exact solution: 50000)

- with centered finite difference:
  - ☐ NAN for $h > 10^{-10}$
  - ☐ 50140.1439388931 for $h \approx 10^{-11}$
  - ☐ 50000.0138777977 for $h \approx 10^{-13}$
  - ☐ 50000.0000016598 for $h \approx 10^{-15}$ $\quad (10^{-7} \cdot (|x| + 1) \cdot \sqrt{\epsilon})$

But for $x = 10000$ similar choice for h is failing to provide good results

Example: Dense Jacobian with forward finite difference

Consider that `void foo(int n, double *x, double &y)` implements the function $F : \mathbb{R}^n \to \mathbb{R}, \quad y = F(\boldsymbol{x})$. The following code demonstrates the computation of approximated Jacobian (gradient) of the function F.

```
...
foo(n, x, y);
...
for (int i = 0; i<n; i++) {
  x[i] += h;
  foo(n, x, yp);
  J[i] = (yp - y)/h;
  x[i] -= h;
}
```

The function `foo` is executed $n + 1$ times

Example: Dense Jacobian with centred finite difference

Consider that `void foo(int n, double *x, double &y)` implements the function $F : \mathbb{R}^n \to \mathbb{R}, \quad y = F(\boldsymbol{x})$. The following code demonstrates the computation of approximated Jacobian (gradient) of the function F.

```
...
for (int i = 0; i<n; i++) {
  tmp = x[i]
  x[i] = tmp + h;
  foo(n, x, yp);
  x[i] = tmp - h;
  foo(n, x, ym);
  J[i] = (yp - ym)/(2*h);
  x[i] = tmp;
}
```

The function `foo` is executed $2 \cdot n$ times

nag®

Algorithmic Differentiation: Tangent-Linear Model

$$F : \mathbb{R}^n \to \mathbb{R}^m, \quad \boldsymbol{y} = F(\boldsymbol{x})$$

- Tangent-Linear Model (TLM) $\dot{F}$ (forward mode)
  $\dot{F}(\boldsymbol{x}, \dot{\boldsymbol{x}}) = \underbrace{F'(\boldsymbol{x})}_{\in \mathbf{R}^{m \times n}} \cdot \underbrace{\dot{\boldsymbol{x}}}_{\in \mathbf{R}^n} = \dot{\boldsymbol{y}}$
  - $F'(x)$ at $O(n) \cdot Cost(F)$
  - exact derivatives
  - $\frac{Cost(\dot{F})}{Cost(F)} \approx 2$

Jacobian with the Tangent-Linear Model

Consider

$$F : \mathbb{R}^n \to \mathbb{R}, \quad \boldsymbol{y} = F(\boldsymbol{x})$$

and $[y, dy] = \text{foo\_dot}(x, dx)$ contains the code that implements the tangent-linear model of F, where $\boldsymbol{x}$ corresponds to x, $\dot{\boldsymbol{x}}$ to dx, $\boldsymbol{y}$ to y and $\dot{\boldsymbol{y}}$ to dy. Then the Jacobian (F') of F can be computed as follows

$dx = 0.0$
for $i = 0; \quad i < n; \quad i++$ do
  $dx_i = 1.0;$
  $[y, dy] = \text{foo\_dot}(x, dx)$
  $\frac{\partial y}{\partial x_i} = dy$
  $dx_i = 0.0$
end for

The tangent-linear model of the function `foo` is executed n times

Algorithmic Differentiation: Adjoint Model

$$F : \mathbb{R}^n \rightarrow \mathbb{R}^m, \quad \boldsymbol{y} = F(\boldsymbol{x})$$

- Adjoint Model (ADM) $\bar{F}$ (reverse mode)

$$\bar{F}(\boldsymbol{x}, \bar{\boldsymbol{y}}) = \underbrace{\bar{\boldsymbol{y}}}_{\in \mathbf{R}^m} \cdot \underbrace{F'(\boldsymbol{x})}_{\in \mathbf{R}^{m \times n}} = F'(\boldsymbol{x})^T \cdot \bar{y} = \bar{\boldsymbol{x}}$$

  - $F'(x)$ at $O(m) \cdot Cost(F)$
  - exact derivatives
  - $\frac{Cost(\bar{F})}{Cost(F)} < 30$

Jacobian with the Adjoint Model

Consider

$$F : \mathbb{R}^n \to \mathbb{R}, \quad \boldsymbol{y} = F(\boldsymbol{x})$$

and $[y, dx] =$ foo_bar(x, dy) contains the code that implements the adjoint model of F, where $\boldsymbol{x}$ corresponds to x, $\bar{\boldsymbol{x}}$ to dx, $\boldsymbol{y}$ to y and $\bar{\boldsymbol{y}}$ to dy. Then the Jacobian (F') of F can be computed as follows

$dy = 1.0;$
$[y, dx] =$ foo_bar(x, dy)
for $i = 0; \quad i < n; \quad i++$ do
  $\frac{\partial y}{\partial x_i} = dx_i$
end for

The adjoint model of the function `foo` is executed only once. Cost for bad implementation of foo_bar may be much higher than 30. The challenge is to make a good implementation.

Live demo: Race!

Summary

Finite difference suffers from two main problems

- Accuracy (due to approximation)
- $O(n)$ complexity for computing dense Jacobian

Both problems can be addressed by AD. Tangent-linear as well as the adjoint model compute exact derivatives with machine accuracy.

With the adjoint model the complexity of Jacobian computation is independent from the number of inputs of the function ($O(m)$).

Outlook: AD Masterclass Part 2

In the next part we will

- Learn how AD works (basic approach behind AD),
- Learn how to apply operator overloading AD tool to a code
- Learn how to write the driver computing the Jacobian for both
  - tangent-linear and
  - adjoint model
- Understand how to choose the best model (tangent-linear or adjoint) for our problem (also Masterclass Part 3)
- Understand the caveats of using AD especially the adjoint model.

You will see a survey on your screen after exiting from this session.

We would appreciate your feedback.

We are no moving on the Q&A Session

References

- A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, *Automatic differentiation in machine learning: a survey*, arXiv preprint arXiv:1502.05767 (2015).
- Felix Gremse, Benjamin Theek, Sijumon Kunjachan, Wiltrud Lederle, Alessa Pardo, Stefan Barth, Twan Lammers, Uwe Naumann, and Fabian Kiessling, *Absorption reconstruction improves biodistribution assessment of fluorescent nanoprobes using hybrid fluorescence-mediated tomography*, Theranostics 4 (2014), 960–971.

References (cont.)

- Mathias Luers, Max Sagebaum, Sebastian Mann, Jan Backhaus, David Grossmann, and Nicolas Gauger, *Adjoint-based volumetric shape optimization of turbine blades*, AIAA 2018-3638 (2018).

- Uwe Naumann, Jean Utke, Carl Wunsch, Patrick Heimbach, Chris Hill, Derya Ozyurt, Carl Wunschyy, Mike Fagan, Nathan Tallent, and Michelle Strout, *Adjoint Code by Source Transformation with OpenAD/F*, Proceedings / European Conference on Computational Fluid Dynamics : Egmond aan Zee, the Netherlands, 5 - 8 September 2006 / ECCOMAS, European Community on Computational Methods in Applied Sciences. Ed.: P. Wesseling ... (Delft), TU Delft, 2006, Datentraeger: 1 CD-ROM, p. 19 S.

References (cont.)

- Andreas Püttmann, Sebastian Schnittert, Uwe Naumann, and Eric von Lieres, *Fast and accurate parameter sensitivities for the general rate model of column liquid chromatography*, Computers & Chemical Engineering 56 (2013), 46 – 57.
- Florian Rauser, Jan Riehme, Klaus Leppkes, Peter Korn, and Uwe Naumann, *On the use of discrete adjoints in goal error estimation for shallow water equations*, Procedia Computer Science 1 (2010), no. 1, 107 – 115, ICCS 2010.
- Antoine Savine, *Modern computational finance: Aad and parallel simulations*, Wiley, 2018.

References (cont.)

Markus Towara, Michel Schanen, and Uwe Naumann, *MPI-Parallel Discrete Adjoint OpenFOAM*, Computational science at the gates of nature : International Conference on Computational Science (ICCS 2015) ; Reykjavik, Iceland, 1 - 3 June 2015 / [organised by: Háskólinn í Reykjavík ...]. Ed.: Slawomir Koziel ... - Pt. 1 (Red Hook, NY), Procedia computer science, vol. 51, International Conference On Computational Science, Reykjavík (Iceland), 1 Jun 2015 - 3 Jun 2015, Curran, Jun 2015, pp. 19–28.

References (cont.)

- J. Ungermann, J. Blank, J. Lotz, K. Leppkes, L. Hoffmann, T. Guggenmoser, M. Kaufmann, P. Preusse, U. Naumann, and M. Riese, *A 3-D tomographic retrieval approach with advection compensation for the air-borne limb-imager GLORIA*, Atmospheric measurement techniques 4 (2011), 2509 – 2529, Record converted from VDB: 12.11.2012.
- Andrey Vlasenko, Peter Korn, Jan Riehme, and Uwe Naumann, *Estimation of Data Assimilation Error: A Shallow-Water Model Study*, Monthly Weather Review 142 (2014), no. 7, 2502–2520.

References (cont.)

V. Vassiliadis, J. Riehme, J. Deussen, K. Parasyris, C. D. Antonopoulos, N. Bellas, S. Lalis, and U. Naumann, *Towards automatic significance analysis for approximate computing*, 2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2016, pp. 182–193.