A template-based C++ library for automatic differentiation and hull consistency enforcing

Bartłomiej Jacek Kubica
Department of Applied Informatics
Warsaw University of Life Sciences

SCAN 2016, Uppsala
Automatic differentiation (AD) is a good alternative to symbolic and numeric differentiation.

Idea:
- Computations are performed by a program.
- This program can be enhanced to compute the derivative(s) together with the original function.
- We attach a derivative to each term and subsequently compute derivatives of the whole expression.

There are a few approaches (forward mode, reverse mode, using surreal numbers, etc.)

We shall concentrate on forward mode, based on operator overloading.
AD arithmetic

\( u, u', v, v' \in \mathbb{R}, \)

\[
\langle u, u' \rangle + \langle v, v' \rangle = \langle u + v, u' + v' \rangle \\
\langle u, u' \rangle - \langle v, v' \rangle = \langle u - v, u' - v' \rangle \\
\langle u, u' \rangle \cdot \langle v, v' \rangle = \langle u \cdot v, u \cdot v' + u' \cdot v \rangle \\
\langle u, u' \rangle / \langle v, v' \rangle = \langle u / v, (u' \cdot v - u \cdot v') / v^2 \rangle
\]

Formulae for other functions can be obtained analogously, e.g.:

\[
\langle u, u' \rangle^n = \langle u^n, n u^{n-1} u' \rangle \\
\exp (\langle u, u' \rangle) = \langle \exp(u), \exp(u) \cdot u' \rangle
\]
A univariate example

\[ f(x) = x^2 + 2x + 1 \]

\[ x = [1, 2] \]

\[ \langle x, x' \rangle = \langle [1, 2], [1, 1] \rangle \]

\[ f(\langle x, x' \rangle) = \langle x, x' \rangle^2 + 2 \langle x, x' \rangle + 1 \]

\[ f(\langle x, x' \rangle) = \langle [1, 2], [1, 1] \rangle^2 + 2 \langle [1, 2], [1, 1] \rangle + 1 \]

\[ f(\langle x, x' \rangle) = \langle [1, 4], [2, 4] \rangle + \langle [2, 4], [2, 2] \rangle + \langle [1, 1], [0, 0] \rangle \]

\[ f(\langle x, x' \rangle) = \langle [4, 9], [4, 6] \rangle \]
Automatic differentiation for multivariate functions

- First derivatives (gradients) are vectors.
- Second derivatives are (Hesse) matrices.
- Derivatives of higher order are tensors aka 3D matrices (or higher dimensional ones).

\[ f(x_1, x_2) = x_1^2 + \sin(x_2) \]
\[ \langle x_1, x_1' \rangle = \langle x_1, \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \rangle, \quad \langle x_2, x_2' \rangle = \langle x_2, \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} \rangle \]
\[ f(\langle x, x' \rangle) = \langle x_1, x_1' \rangle^2 + \sin(\langle x_2, x_2' \rangle) \]
\[ f(\langle x, x' \rangle) = \langle x_1^2 + \sin(x_2), \begin{bmatrix} 2x_1 \\ \cos(x_2) \end{bmatrix} \rangle \]
State-of-the-art software

- C-XSC (eXtended Scientific Computing) – the library I'm using.
- Interval calculus and various auxiliary tools:
  - classes for interval (and non-interval) vectors and matrices,
  - classes for sparse vector and matrices,
  - multiprecision arithmetic,
  - ...
- In particular, we have automatic differentiation.
- But the implementation is far from satisfying...
Drawbacks of C-XSC automatic differentiation procedures

- Distinct classes for computing $1^{\text{st}}$ or $2^{\text{nd}}$ derivatives and for uni- or multivariate functions.
- Computing higher derivatives would require to implement a separate (but analogous) class.
- The developers of C-XSC provide several useful classes for sparse matrices and vectors, but their AD code makes no use of it.
  - The Hesse matrix is stored as a collection of vectors. The procedures are aware of its symmetry, but:
    - All operations are done elementwise.
    - No use of BLAS for matrix-vector multiplication.
    - No use of potential sparsity.
What we need is...

- a generic technique to provide classes for:
  - uni- and multivariate functions,
  - dense and sparse representations,
  - computing original functions, its gradients, Hesse matrices, etc.
What we need is...

• ... a generic technique to provide classes for:
  – uni- and multivariate functions,
  – dense and sparse representations,
  – computing original functions, its gradients, Hesse matrices, etc.

• But, hey! We do have such a technique. It is template meta-programming.
C++ templates in a nutshell

- Generic "classes" of classes (and functions; the C++14 standard adds variable templates, also).
- Parameterized by an ordinal value or by a type.
- Some kind of pre-processor creates distinct binary codes for distinct specializations of a template.
- Tools:
  - Partial specialization of templates.
  - Templates to choose the type depending on a logical condition.
  - Typelists, allowing, e.g., generation of tuples.
My library

- ADHC – **Automatic Differentiation and Hull Consistency.**

- The basic type:
  ```cpp
template<int level, sparsity_t sparse_mode, int num_vars>
struct adhc_ari {
    // ...
};
```

- The `sparsity_t` is an enumeration type:
  ```cpp
typedef enum sparsity_t {dense, sparse, highly_sparse};
```

- Using typelists, we choose proper types for various derivatives: scalar values or vectors, matrices, etc.; sparse or dense representations...
• "Sparse" vs "highly sparse" mode:
  – dense gradient representation (**ivector**) and a sparse Hesse matrix (**simatrix**),
  – sparse representations for both (**sivector** and **simatrix**).

• The **level** template param. can take four values:
  – 0 – compute the function value itself,
  – 1 – compute the function and gradient values,
  – 2 – function, gradient and Hesse matrix values,
  – -1 – create a **syntactic tree**, that can be used to enforce **hull consistency**...

... BUT THAT'S QUITE ANOTHER STORY.
Using sparse vectors as gradient representations

- Surprisingly, using sparse representations is often worthwhile even for non-sparse problems!

\[ f(x_1, \ldots, u_n) = \sum_{i=0}^{n} x_i^2 \]

- The gradient seems dense:

\[ \nabla f(x_1, \ldots, x_n) = \begin{pmatrix} 2x_1 \\ \vdots \\ 2x_n \end{pmatrix} \]
Using sparse vectors as gradient representations

- Surprisingly, using sparse representations is often worthwhile even for non-sparse problems!

\[ f(x_1, \ldots, u_n) = \sum_{i=0}^{n} x_i^2 \]

- The gradient seems dense:

\[ \nabla f(x_1, \ldots, x_n) = \begin{pmatrix} 2x_1 \\ \vdots \\ 2x_n \end{pmatrix} \]

- But it is computed as:

\[ \nabla f(x_1, \ldots, x_n) = (2x_1) \cdot \nabla x_1 + \cdots + (2x_n) \cdot \nabla x_n \]
Using sparse vectors as gradient representations

- Surprisingly, using sparse representations is often worthwhile even for non-sparse problems!

\[ f(x_1, \ldots, u_n) = \sum_{i=0}^{n} x_i^2 \]

- The gradient seems dense:

\[ \nabla f(x_1, \ldots, x_n) = \begin{pmatrix} 2x_1 \\ \vdots \\ 2x_n \end{pmatrix} \]

- But it is computed as:

\[ \nabla f(x_1, \ldots, x_n) = (2x_1) \cdot \nabla x_1 + \cdots + (2x_n) \cdot \nabla x_n \]
Using sparse vectors as gradient representations

- Surprisingly, using sparse representations is often worthwhile even for non-sparse problems!

\[ f(x_1, \ldots u_n) = \sum_{i=0}^{n} x_i^2 \]

- The gradient seems dense:

\[ \nabla f(x_1, \ldots, x_n) = \begin{pmatrix} 2x_1 \\ \vdots \\ 2x_n \end{pmatrix} \]

Dense representation:
- \( n^2 \) interval multiplications,
- \((n - 1) \cdot n\) interval additions.

Sparse representation:
- \( n \) interval multiplications,
- Some cheap index checking.
- No interval additions!

- But it is computed as:

\[ \nabla f(x_1, \ldots, x_n) = (2x_1) \cdot \nabla x_1 + \cdots + (2x_n) \cdot \nabla x_n \]
My library

• Being integrated to my solvers for nonlinear systems, Nash equilibria, optimization, MPC...

• More efficient than the AD package from C-XSC:

  – A spectacular example: HIBA_USNE solver applied to the Extended Freudenstein problem (200 equations in 200 variables):
    • in 2 min. 15 sec. – using AD from C-XSC,
    • in 2 min. 5 sec. – using ADHC with dense mode,
    • in 1 min. 50 sec. – using ADHC with sparse mode,
    • in 35 sec. – using ADHC with highly sparse mode.
  – But the speedup is noticeable already for 8 variables.
  – Operations on Hesse matrices are less efficient, but also efficient enough.
Limitations & possible improvements

- Some transcendental functions not available yet.
  - Nothing challenging, just needs to be done.
  - But writing code that is not challenging is no fun, aww...
  - Also, there might be a generic way...

- No derivatives of order higher than 2 can be computed, as for now.
  - I don't have a tensor representation.
  - What is more, I have not been able to obtain general enough formulae!
  - It is a serious drawback, but gradients and Hesse matrices are already useful, aren't they?
Example of formulae

- Consider, e.g., the multiplication operation...
- For univariate functions we have a relatively simple formula:

\[ f^{(n)}(u_1, u_2) = \sum_{i=0}^{n} \binom{n}{i} u_1^{(i)} \cdot u_2^{(n-i)} \]

- But not so for multivariate functions:

\[ f(u_1, u_2) = u_1 \cdot u_2, \]
\[ \nabla f(u_1, u_2) = u_1 \cdot \nabla u_2 + u_2 \cdot \nabla u_1, \]
\[ \nabla^2 f(u_1, u_2) = u_1 \cdot \nabla^2 u_2 + \nabla u_1 \cdot (\nabla u_2)^T + \nabla u_2 \cdot (\nabla u_1)^T + u_2 \cdot \nabla^2 u_1, \]

...
Example of formulae

- Providing formulae for higher derivatives is all but simple.
- Yes, we have the Faa di Bruno's formula (and other similar ones), but it gives us the way to compute **single components** and not the whole derivative tensors.
- And using using specific components would require us to have distinct codes for dense and sparse representations, making the whole effort almost worthless...
Limitations & possible improvements

- Dependency on Loki (typelists and tuples).
  - It was my deliberate decision not to use C++11 innovations (variadic templates and `std::tuple` classes).
  - But it might have been better to do so – hey, it's 2016; most environments probably have it...
  - Going to be changed in future versions.

- Extended interval division and infinite intervals.
  - How to evaluate expressions, like \([0, 1] / [0, 1]\) or \([0, 1] \cdot [1, +\infty]\)? (IEEE P1788 Standard?)
  - Efficiency vs versatility.
  - It might be chosen by one of the template arguments.
Limitations & possible improvements

● Other interval libraries (e.g., Profil/BIAS or GAOL).
  – Relatively simple to do, but wearisome.

● Basic types other than `cxsc::interval`.
  – Plain floating-point computations (`cxsc::real`)?
  – Staggered arithmetic (`cxsc::lx_interval`)?
  – Complex numbers/intervals (`cxsc::complex`, `cxsc::cinterval`, etc.) and holomorphic functions?
  – The extension could be done, relatively easily.
  – As for now, I wanted to keep the templates as simple as possible, but...
ADHC library

- Version **Alfa 0.1** of **ADHC** is already available from my ResearchGate profile

- Is has been integrated, i.a., to a new version (Beta 2.0) of **HIBA_USNE** (**Heuristical Interval Branch-and-prune Algorithm for solving Underdetermined Systems of Nonlinear Equations**) nonlinear solver, also available from the page.

- Does anyone want to cooperate? Contribute? Co-whatever? Do not hesitate, you're very welcome!
Conclusions

- The developed library works and seems useful.
- It is more efficient than a state-of-the-art competitor.
- It occurred particularly worthwhile to use sparse vectors to represent gradients in the AD process.
- Operations on sparse Hesse matrices did not turn out that well, but they are efficient enough, also.
- Hull consistency enforcing will be ready, soon.
- Further development in progress...