P2072R0: Differentiable programming for C++

Date: 2020-01-13 (Pre-Prague mailing)

Project: ISO JTC1/SC22/WG21: Programming Language C++

Audience: SG19, WG21

Authors: Marco Foco, Max Rietmann, Vassil Vassilev, Michael Wong

Contributors Dmitry Duka, Vinod Grover

Emails: mfoco@nvidia.com, mrietmann@nvidia.com, v.g.vassilev@gmail.com, michael@codeplay.com

Reply to: mfoco@nvidia.com, v.g.vassilev@gmail.com

Introduction

Derivatives are vital a wide variety of computing applications, including numerical optimization, solution of nonlinear equations, sensitivity analysis, and nonlinear inverse problems. Virtually every process could be described with a mathematical function. A mathematical function can be thought of as an association between elements from different sets. Derivatives can track how a varying quantity depends on another quantity, for example how the position of a planet as the time varies. Derivatives and gradients allows us to explore the properties of a function and thus the described process as a whole. Also, gradients are an essential component in gradient-based optimization methods, that have become more and more important in recent years, in particular with its application training of (Deep) Neural Networks.

Derivatives can be computed numerically, but unfortunately the finite differences methods are problematic due to the approximation operated and the finite precision of floating point values used, and so the implementation of the method faces precision and round off problems which can affect the overall precision of the computation. This problem becomes worse with higher order derivatives.

The computational complexity of many problems depend on the number of input variables which poses scalability issues. This paper describes a broad set of domains where scalable derivative computations are essential. We make an overview of the major techniques in computing derivatives, and finally, we introduce the flagman of computational differential calculus -- algorithmic (also known as automatic) differentiation (AD). AD makes a clever use of the ‘nice’ mathematical properties of the chained rule and generative programming to solve the scalability issues by inverting the dependence on the number of input variables to the number of output variables. It gives us a tool to augment the regular function computation with instructions calculating its derivatives.

Differentiable programming is a programming paradigm in which the programs can be differentiated throughout, usually via automatic differentiation. The main scope of this document is to open a discussion on the possible scenarios enabling differential programming in C++. It briefly introduces possible approaches to implement AD such as a library solution, a language solution, and a library solution using future language features.

Background

Derivatives and gradients can be computed in several different ways [2]:

- Derivation by hand is a tedious and error-prone process. In case the initial function changes we need to remember to invalidate the derivative and manually derive it again. The manual derivation usually assumes derivation of a math expression in the math domain and translating it to code. It is virtually impossible to manually derive an algorithm because of its multi-level dependencies.
- Symbolic Differentiation (SD) is a method for automatically applying the chain rule to mathematical functions. This is the approach implemented in languages like Mathematica [15] or Maple [14]. It is limited to closed form expressions, that is, it cannot handle control flow [17].
- Divided (or Finite) Differentiation (DD/FD), a numerical method to approximate a derivative. Its implementation quickly reaches the limitations of the machine epsilon and the floating point representation issues. Usually, the algorithm can compute the derivative with a user-defined precision. This includes iteration and poses problems in convergence. Another problem for this method is that it's hard to find a good perturbation with the right tradeoff between maintaining numerical stability and accuracy.
- Automatic Differentiation (AD), is a set of methods that allows to efficiently differentiate *Algorithms* (as opposed to just *mathematical functions* in symbolic differentiation). Two main modes are used, Forward Automatic Differentiation (FAD) and Reverse Automatic Differentiation (RAD).

Automatic Differentiation and Symbolic differentiation are often confused. They are similar when dealing with single expressions (e.g. `f(x) = x*x`, in both cases give the same result, `f'(x) = 2*x`), but the automatic differentiation method(s) include rules for efficiently differentiating *sequences* of instructions, exploiting the existent code structure to optimize the use of intermediate variables and (to some degree) control flow.

Historically, Computer Algebra Systems (CAS) such as Macsyma (now Maxima [13]), Maple [14], and Mathematica [15] natively supported derivation as built-in (typically in the form of Symbolic Differentiation).

In the last decades, several other libraries [2], implemented in many different languages, have been developed to solve the problem of differentiation, in the form of Automatic Differentiation (implementing forward mode or, more recently reverse mode).

In the last few years, graph-based packages used in Deep Learning, such as Tensorflow, (py)Torch and mxNet, supported automatic differentiation too, and this gave them a competitive advantage to packages that didn't. These packages are used to implement the "backpropagation" step in training, that is equivalent to Automatic Differentiation in reverse mode ([9], [6]).

More recently, some languages (e.g. Swift [7], Julia [4], Halide [8], DiffTaichi [16]) introduced differentiation as a first class citizen in the language, allowing differentiation of code under specific conditions (in a form more similar to AD than to SD).

Numerical Differentiation

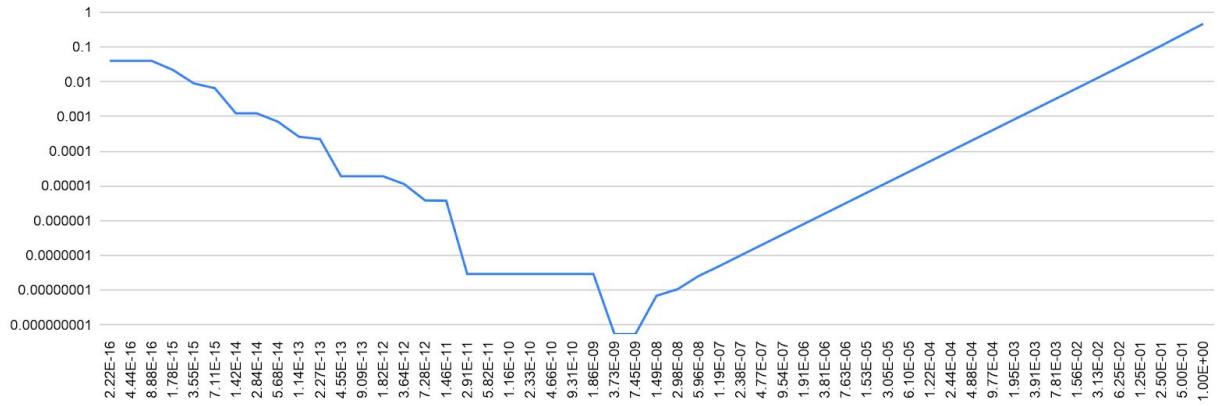
By definition the first derivative is

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

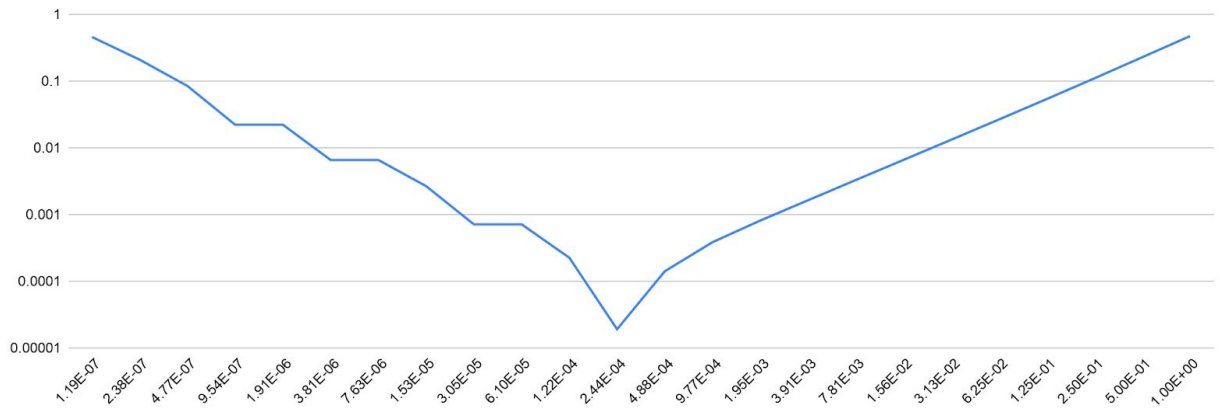
The simplest finite differences method consists in applying the definition, but with a finite value for h. A naïve implementation follows directly from the definition, by replacing the limit with a small number taken as a parameter:

```
template<typename F>
double asymmetric_diff(F f, double h, double x) {
    return (f(x+h) - f(x))/h;
}
```

Other methods use more sample points and different sampling schemes to achieve more precise results.
Unfortunately, due to the nature of floating point representations used in modern computers, we cannot choose an arbitrarily small number for h, as on one side we need to have a value of `h` small enough to consider the finite difference a good enough approximation, but on the other side h need not to be too small to avoid introducing roundings and cancelations in our computation.
The following figure shows the error of computing the derivative of the function `sin(x)` in `x=1` as a function of the displacement h and type **`double`**.



And this is the same experiment with **`float`**.



Empirically, the "sweet spot" is at `h=5E-9` for **`double`** and `h=2.5E-4` for **`float`**, but in other tests cases the optimal value can move by some orders of magnitude. Given T, ε_T is the value of `std::numeric_limits<T>::epsilon()`, some text report that the optimal value would be $h = 2\sqrt{\varepsilon_T \left|f(x)/f''(x)\right|}$, but that's impossible to calculate in general, as it depends both on the value of x and the specific value of the function and its second derivative. In most cases, a good approximation is choosing $h = \sqrt{\varepsilon_T}$.

Automatic Differentiation

Algorithm and Transformation

The AD transformation uses the properties of the Chain Rule of differential calculus:

$$y = f(g(h(x))) = f(g(h(w_0))) = f(g(w_1)) = f(w_2) = w_3$$
$$w_0 = x$$
$$w_1 = h(w_0)$$
$$w_2 = g(w_1)$$
$$w_3 = f(w_2) = y$$

$$\frac{dy}{dx} = \frac{dy}{dw_2}\ \frac{dw_2}{dw_1}\ \frac{dw_1}{dx}$$

A straightforward interpretation of the mathematical properties is that each function can be split into smaller, atomic operations where the differentiation rules can be applied. We can automatically visit every expression and transform it. The canonical form of the chained rule expresses the derivative with respect to function's input parameters, that is, the independent variable (seed) is the function parameter. This is useful when we compute a derivative in wrt a single parameter (a single direction). The complexity of the derivative computation depends on the number of the input parameters. This approach is called forward/tangent mode AD. Let's consider:

$$f(x_1, x_2) = \begin{cases} (x_1 - x_2)^2 + x_2 & \text{when } x_1 > x_2 \\ x_2 & \text{when } x_1 \le x_2 \end{cases}$$

$f(x_1, x_2)$	$df/dx1$	$df/dx2$

f(x1, x2) {
  x1 = x1
  x2 = x2
  if (x1 > x2)
  a = (x1 - x2)
  b = a*a
  return a + x2
  return x2
}
	f_dx1(x1, x2) {
	  dx1 = 1
	  dx2 = 0
	  if (x1 > x2)
	  da = dx1 - dx2
	  db = a*da + da*a
	  return da + dx2
	  return dx2
	}
		f_dx2(x1, x2) {
		  dx1 = 0
		  dx2 = 1
		  if (x1 > x2)
		  da = dx1 - dx2
		  db = a*da + da*a
		  return da + dx2
		  return dx2
		}

When we start differentiating in multiple directions we will notice that a lot of the intermediary results can be shared between different directions of the derivatives with a little tuning. The chained rule is symmetrical. This means that we can express the canonical form of the chained rule with respect to the function's output parameters instead. This means that the complexity of the algorithm will depend on the number of output parameters which in many cases is significantly smaller than the number of input parameters. This approach is called reverse/adjoint mode. It's harder to implement but it reuses many intermediary computations and reduces the algorithm complexity in most of the interesting cases, i.e. when the number of input parameters is much larger than the number of outputs. Let's consider again:

$$f(x_1, x_2) = \begin{cases} (x_1 - x_2)^2 + x_2 & \text{when } x_1 > x_2 \\ x_2 & \text{when } x_1 \leq x_2 \end{cases}$$

$f(x_1, x_2)$	$\nabla f(x_1, x_2)$

f(x1, x2) {
  x1 = x1
  x2 = x2
  if (x1 > x2)
  a = (x1 - x2)
  b = a*a
  return a + x2
  return x2
}
	f_grad(x1, x2)
	  gz = 1
	  if (x1 > x2) {
	  a = x1 - x2
	  gx2 = gz
	  ga = a*gz + gz*a
	  gx2 += -ga
	  gx1 = ga
	  return {gx1, gx2}
	  gx2 = dz
	  gx1 = 0
	  return {gx1, gx2};
	}

Implementation Approaches

Forward mode (aka Tangent Linear)

In forward mode we compute derivatives of the output(s) with respect to each input independently, using the traditional chaining rules from analytical differentiation at each step.

The complexity of this method depends thus on the number of inputs, and so it should be applied when the number of input is relatively small, or when the number of outputs is very large.

There are two ways to implement the Forward mode.

Forward mode with Dual Numbers

One method is using *dual numbers* (see appendix A). When an input variable x is replaced with a dual number of the form x + eps (i.e. having the epsilon part set to 1), the function will produce a result of type dual containing the function value at x together with its partial derivative with respect to x.

For example, let's consider this function:

```
double f(double x1, double x2) {
    if (x1 > x2) {
        double a = x1 - x2;
        return a*a + x2;
    }
    return x2;
}
```

But when we substitute the first type with a dual number, the result is:

```
dual f(dual x1, double x2) {
    if (x1.real() > x2) {
        dual a = x1 - x2; // the dual number a has the
                          // form {x1.real() - x2, x1.eps()};
        return a*a + x2; // the result is dual, and has the form
                         // {(x1.real() - x2)*(x1.real() - x2) + x2,
                         //       (x1.real() - x2)*x1.eps() +
                         //             x1.eps()*(x1.real() - x2)}
      }
      return x2; // the result is a real, which can be
                 // converted to a dual of the form {x2, 0}
}
```

so `f({x1, 1}, x2).real()` is the same as `f(x1, x2)`, while `f({x1, 1}, x2).eps()` is `2*(x1-x2)` for `x1>x2` and `0` for `x1<=x2`, which is exactly the partial derivative of `f` with respect to `x1`.
We can use a different *kind* of dual numbers (with one real component and multiple epsilon parts, one for each variable) for handling multiple parameters, and calculate all the partial

derivatives in a single pass. The complexity of this method, in any of these cases, depends linearly on the number of input variables.

Forward mode (AST Transformation)

Another method is to produce one single piece of code that can then be customized in different ways, is also the method implemented in CLAD which performs an AST to AST transformation to produce a differentiated form.
Let's use the same example as above:

```
double f(double x1, double x2) {
    if (x1 > x2) {
        double a = x1 - x2;
        return a*a + x2;
    }
    return x2;
}
```

The function df produced by clang looks like:

```
double df(double x1, double x2) {
  double dx1 = ?;
  double dx2 = ?;
  if (x1 > x2) {
    double a = x1 - x2;
    double da = dx1 - dx2;
    return a*da + da*a + dx2;
  }
  return dx2;
}
```

In **df** the two components `dx1` and `dx2` (and any other differential) form a versor (i.e. a vector having norm 1) along which the derivative is computed. The easiest way, which incidentally produces a lot of obvious optimization, is choosing dx and dy (and any other differential) to form a trivial orthonormal base, i.e. in this case (`dx1 = 1, dx2 = 0`) and (`dx1 = 0, dx2 = 1`). The two sets produce the two partial derivatives with respect to `x` and `y` respectively. Appendix C shows the AST of both **f** and **df**.

Reverse mode (Adjoint)

In reverse (adjoint) mode, the computation of derivatives proceeds from the outputs to the inputs, following the usual derivation chain rules. In this way the complexity of the final result is independent from the number of inputs.

```
double f(double x1, double x2) {
    if (x1 > x2) {
```

```
        double a = x1 - x2;
        return a*a + x2;
    }
    return x2;
}


std::tuple<double, double> gradf(double x1, double x2) {
    double gz = 1;      // return statement(s)
    if (x1 > x2) {
        double a = x1 - x2;      // From forward pass
        double gx2 = gz;         // +x2 part of "return a*a+x2;"
        double ga = a*gz + gz*a; // a*a part of "return a*a+x2;"
        gx2 += -ga;              // -x2 part of "a = x1 - x2;"
        double gx1 = ga;         // x part of "a = x1 - x2;"
        return {gx1, gx2};
    }
    double gx2 = gz;     // x2 part of "return x2"
    return {0, gx2};     // x1 isn't involved in the computation
}
```

Library vs Language solution

In the section about Automatic Differentiation we've shown some possible transformations of the source code in order to produce partial derivatives and gradients. Some of those solutions could be implemented as a Library in *current* C++ (e.g. forward differentiation with dual numbers, see [12]), and in general AD has been implemented in several libraries in modern C++ [2].
We found three main reasons to prefer a language solution to a library solution:

- **Type safety**
  Most of these solutions make extensive use of the type system (TMP and Expression Templates) to achieve differentiation. As a result, little is left to enforce types in base and differentiated expressions.
  A language solution would also allow to differentiate with respect to complex types (e.g. structs, vectors) easily, without the need to re-specify the type.
- **Efficiency**
  A library solution will have to make use of techniques like TMP and expression templates, which can end up being expensive for the compiler, as it will have to maintain all these intermediate types. It can also get less efficient when automatic inlining limits are reached. The compiler, on the other hand, is already aware of the AST representation of the original function, and can perform the differentiation tasks without burden to the (already abused) type system.
- **Completeness**

> Differentiating control flow code could be impossible for a library solution (or requires changing the code significantly, affecting readability), while it's feasible in the language [4]

Here are some examples of the problems mentioned above:

Example 1: Forward differentiation with Boost.Math

First of all, let's consider the impact of using a library solution by evaluating the impact of including the Boost.Math autodiff header (`boost/math/differentiation/autodiff.hpp`) in an otherwise empty file.

		int main() {}		+#include	
		Time (ms)	**Size (KB)**	**Time (ms)**	**Size (KB)**
GCC 9.2	**No option**	48	~1	2218	220
	-O3	49	~1	2281	48
Clang 9.0.0	**No option**	89	~1	2404	158
	-O3	91	~1	2488	23

So in every compiler tested, just including the autodiff header adds several seconds to the compilation and multiple hundreds of kilobytes to the final executable. We didn't measure impact on memory allocated by the compiler, but we expect a significant difference there too.

Let's now consider the same example we used in the previous section, and show how to implement it in boost.math:

```
double f(double x1, double x2) {
    if (x1 > x2) {
        double a = x1 - x2;
        return a*a + x2;
    }
    return x2;
}
```

First of all, this function needs to be rewritten as generic, at least in the parameter we want to differentiate, as it cannot be consumed from boost.math otherwise.
We could be tempted to write the function like this:

```
template<typename X1, typename X2>
```

```
auto f(X x1, X2 x2) {
    if (x1 > x2) {
        auto a = x1 - x2;
        return a*a + x2;
    }
    return x2;
}
```

But that wouldn't work, as the two `return` statements are returning two different types. In case of multiple input parameters, we also have to use a special return type, `promote<...>`:

```
template<typename X1, typename X2>
promote<X1, X2> f(X1 x1, X2 x2) {
    if (x1 > x2) {
        auto a = x1 - x2;
        return a*a + x2;
    }
    return x2;
}
```

Thus making the language in which we're writing the function more and more distant than plain C++. If we want to get the partial derivative w.r.t. x1, we have to write this code:

```
auto x1 = make_fvar<double, 1>(2.0);
auto z = f(x1, 1.0);
```

Example 2: Reverse differentiation with Enoki

Since Boost.Math doesn't support reverse-mode automatic differentiation, we'll show this example in Enoki [19].

We also have a similar table for compilation times, this time size increase is less relevant, as Enoki also links against a dynamic library of more than 500KB.

		int main() {}		**+#include**	
		Time (ms)	**Size (KB)**	**Time (ms)**	**Size (KB)**
GCC 9.2	**No option**	48	~1	1049	18
	-O3	49	~1	1180	7
Clang 9.0.0	**No option**	89	~1	1190	8
	-O3	91	~1	1017	7

Reverse differentiation needs to know the computational graph in order to reverse the order of the operations when computing the gradient, and thus cannot traverse all the control flows. For this reason, reverse-mode enabled frameworks usually provide a custom alternative to the condition `if`. In Enoki this function is called `select`. For this reason, our function becomes less and less readable:

```
template<typename Value>
Value f(Value x1, Value x2) {
    return select(x1>x2, (x1-x2)*(x1-x2) + x2, x2);
}
```

For completeness, we report the code used to produce the full gradient:

```
FloatD x1 = 2.0;
FloatD x2 = 1.0;
set_requires_gradient(x1);
set_requires_gradient(x2);
FloatD z = f(x1, x2);
backward(z);
return {gradient(x1), gradient(x2)};
```

Implementation

In the compiler, the differentiation can be implemented in different ways, we know of two different ways that can be applied to a C++ compiler (specifically, to clang).

- Transforming the AST (Appendix C)
  This is implemented in CLAD [10][20]
- Transforming the SSA IR (Appendix B)
  This is implemented by Julia's Zygote [4]

This leaves some freedom to compiler implementers to decide the strategy that best suits their product.

Appendix A: Dual Numbers

Dual numbers (Clifford, 1873) are defined in a way similar to complex numbers, with a real component and another component multiplied by a base ε. ε is not a real number, but has the property $\varepsilon^2 = 0$ (as opposed as $i^2 = -1$ for the imaginary base of complex numbers).

Properties

$a, b \in \Re, \quad \varepsilon \notin \Re, \quad \varepsilon^2 = 0 \quad \Rightarrow \quad a + \varepsilon b$ is a dual number

Interactions with $\mathbb{R}$

Given a, b, c $\in \mathbb{R}$
(a+εb) + c = (a+c) + εb
(a+εb) * c = (ac) + ε(bc)

Interactions between dual numbers

1. **Sum of two dual numbers**
   Given a, b, c, d $\in \mathbb{R}$
   (a+εb) + (c+εd) = (a+c) + ε(b+d)
   The sum of two dual number is the sum of their components

2. **Product of two dual numbers**
   Given a, b, c, d $\in \mathbb{R}$
   (a+εb) * (c+εd) = (ac) + ε(ad+bc) ~~+ ε^2bd~~
   The real part of the product of two dual numbers is the product of the real parts. The epsilon part, on the other hand, is the sum of the products of the real part of the first number by the epsilon component of the second number, and the real part of the second number and the epsilon component of the first. The ε^2 part is ignored as, by definition ε^2 is zero.

3. **Square of a dual number**
   Given a, b $\in \mathbb{R}$
   $(a+\varepsilon b)^2 = a^2 + \varepsilon(ab + ba)$ ~~+ $\varepsilon^2 b^2$~~ $= a^2 + 2\varepsilon(ab)$
   The square of a dual number is equivalent to multiplying the dual number by itself, so from the previous case we have that the real part is squared and the epsilon part is two times the product of the real and epsilon part.
   Note that if you consider the case b=1, the real part is the real part squared, and the coefficient of epsilon is 2a.

4. **n-th power of a dual number**
   Given a, b $\in \mathbb{R}$
   $(a+\varepsilon b)^n = a^n + \varepsilon(a\ldots ab + aba\ldots a + a\ldots ab) + \ldots = a^n + \varepsilon(na^{n-1}b)$
   The n-th power of a dual number follows similar rules, but in this case the real part is a^n and the epsilon part is $na^{n-1}b$.
   Note again that when b=1, the epsilon part is na^{n-1}.

5. For a given function f(x), switching the parameter x with z=x+ε, the function obtained f(z) has an interesting property: the real part of f(z) (denoted as Re[f(z)]) is the same as f(x), while the epsilon part (denoted as Eps[f(z)]) is its derivative. Given a $\in \mathbb{R}$

f(x)	f(z) = f(x+ε)	Re[f(z)]	Eps[f(z)]
a	a	a	0
x	x + ε	x	1
x+a	(x + a) + ε	x+a	1
ax	ax + εa	ax	a
x^n	$x^n + \varepsilon(nx^{n-1})$	x^n	nx^{n-1}

6. We will now show that given any function that can be expressed as a McLaurin series, the property at 5 is maintained. Given $a_k \in \Re$

$$f(x) = \sum_{k=0}^{\infty} a_k x^k = a_0 + \sum_{k=1}^{\infty} a_k x^k$$ McLaurin expansion of function $f(x)$

$$f(z) = a_0 + \sum_{k=1}^{\infty} a_k z^k$$ z is dual

$$f(x+\varepsilon) = a_0 + \sum_{k=1}^{\infty} a_k (x+\varepsilon)^k$$ $z = x + \varepsilon$

$$f(x+\varepsilon) = a_0 + \sum_{k=1}^{\infty} a_k x^k + \varepsilon \sum_{k=1}^{\infty} a_k k x^{k-1}$$ Splitting the sum in two sums.

$$f(x+\varepsilon) = f(x) + \varepsilon f'(x)$$

Reference implementation can be found in [12] path pan/include/pan/bases, in particular files dual.hpp, epsilon.hpp, base.hpp.

Appendix B: Result of Julia Zygote JIT to LLVM IR

Zygote.jl [4] example:

```
function P1(a::Float64, b::Float64) :: Float64
    return a^2 + a*b
end
```

```
@code_llvm P1(0.1, 1.0)
; a*a
%2 = fmul double %0, %0
; a*b
%3 = fmul double %0, %1
; a*a + a*b
%4 = fadd double %2, %3
ret double %4

@code_llvm gradient(P1, 0.1, 1.0)
; gradient = [2*a + b, a]
; a + a
 %3 = fadd double %1, %1
; 2*a + b
 %4 = fadd double %3, %2
 %.sroa.0.0..sroa_idx = getelementptr inbounds [2 x double], [2 x double]*
%0, i64 0, i64 0
; Res[0] = 2*a + b
store double %4, double* %.sroa.0.0..sroa_idx, align 8
 %.sroa.2.0..sroa_idx4 = getelementptr inbounds [2 x double], [2 x double]*
%0, i64 0, i64 1
; Res[1] = a
store double %1, double* %.sroa.2.0..sroa_idx4, align 8
ret void
```

Appendix C: CLAD forward differentiation

Original function f:

```
double f(double x1, double x2) {
    if (x1 > x2) {
        double a = x1 - x2;
        return a*a + x2;
    }
    return x2;
}
```

AST form of f:

```
FunctionDecl f 'double (double, double)'
|-ParmVarDecl x1 'double'
```

```
|-ParmVarDecl x2 'double'
`-CompoundStmt
  |-IfStmt
  | |-BinaryOperator 'bool' '>'
  | | |-ImplicitCastExpr 'double' <LValueToRValue>
  | | | `-DeclRefExpr 'double' lvalue ParmVar 'x1' 'double'
  | | `-ImplicitCastExpr 'double' <LValueToRValue>
  | |   `-DeclRefExpr 'double' lvalue ParmVar 'x2' 'double'
  | `-CompoundStmt
  |   |-DeclStmt
  |   | `-VarDecl a 'double' cinit
  |   |   `-BinaryOperator 'double' '-'
  |   |     |-ImplicitCastExpr 'double' <LValueToRValue>
  |   |     | `-DeclRefExpr 'double' lvalue ParmVar 'x1' 'double'
  |   |     `-ImplicitCastExpr 'double' <LValueToRValue>
  |   |       `-DeclRefExpr 'double' lvalue ParmVar 'x2' 'double'
  |   `-ReturnStmt
  |     `-BinaryOperator 'double' '+'
  |       |-BinaryOperator 'double' '*'
  |       | |-ImplicitCastExpr 'double' <LValueToRValue>
  |       | | `-DeclRefExpr 'double' lvalue Var 'a' 'double'
  |       | `-ImplicitCastExpr 'double' <LValueToRValue>
  |       |   `-DeclRefExpr 'double' lvalue Var 'a' 'double'
  |       `-ImplicitCastExpr 'double' <LValueToRValue>
  |         `-DeclRefExpr 'double' lvalue ParmVar 'x2' 'double'
  `-ReturnStmt
    `-ImplicitCastExpr 'double' <LValueToRValue>
      `-DeclRefExpr 'double' lvalue ParmVar 'x2' 'double'
```

AST form of df:

```
FunctionDecl df 'double (double, double)'
|-ParmVarDecl x1 'double'
|-ParmVarDecl x2 'double'
`-CompoundStmt
  |-DeclStmt
  | `-VarDecl dx1 'double' cinit
  |   `-FloatingLiteral 'double' ?
  |-DeclStmt
  | `-VarDecl dx2 'double' cinit
  |   `-FloatingLiteral 'double' ?
```

```
|-IfStmt
| |-BinaryOperator 'bool' '>'
| | |-ImplicitCastExpr 'double' <LValueToRValue>
| | | `-DeclRefExpr 'double' lvalue ParmVar 'x1' 'double'
| | `-ImplicitCastExpr 'double' <LValueToRValue>
| |   `-DeclRefExpr 'double' lvalue ParmVar 'x2' 'double'
| `-CompoundStmt
|   |-DeclStmt
|   | `-VarDecl a 'double' cinit
|   |   `-BinaryOperator 'double' '-'
|   |     |-ImplicitCastExpr 'double' <LValueToRValue>
|   |     | `-DeclRefExpr 'double' lvalue ParmVar 'x1' 'double'
|   |     `-ImplicitCastExpr 'double' <LValueToRValue>
|   |       `-DeclRefExpr 'double' lvalue ParmVar 'x2' 'double'
|   |-DeclStmt
|   | `-VarDecl da 'double' cinit
|   |   `-BinaryOperator 'double' '-'
|   |     |-ImplicitCastExpr 'double' <LValueToRValue>
|   |     | `-DeclRefExpr 'double' lvalue Var 'dx1' 'double'
|   |     `-ImplicitCastExpr 'double' <LValueToRValue>
|   |       `-DeclRefExpr 'double' lvalue Var 'dx2' 'double'
|   `-ReturnStmt
|     `-BinaryOperator 'double' '+'
|       |-BinaryOperator 'double' '+'
|       | |-BinaryOperator 'double' '*'
|       | | |-ImplicitCastExpr 'double' <LValueToRValue>
|       | | | `-DeclRefExpr 'double' lvalue Var 'a' 'double'
|       | | `-ImplicitCastExpr 'double' <LValueToRValue>
|       | |   `-DeclRefExpr 'double' lvalue Var 'da' 'double'
|       | `-BinaryOperator 'double' '*'
|       |   |-ImplicitCastExpr 'double' <LValueToRValue>
|       |   | `-DeclRefExpr 'double' lvalue Var 'da' 'double'
|       |   `-ImplicitCastExpr 'double' <LValueToRValue>
|       |     `-DeclRefExpr 'double' lvalue Var 'a' 'double'
|       `-ImplicitCastExpr 'double' <LValueToRValue>
|         `-DeclRefExpr 'double' lvalue Var 'dx2' 'double'
`-ReturnStmt
  `-ImplicitCastExpr 'double' <LValueToRValue>
    `-DeclRefExpr 'double' lvalue Var 'dx2' 'double'
```

References

[1] Various Authors
Wikipedia entry: Differentiable Programming

[2] Various Authors,
Community Portal for Automatic Differentiation

[3] Christian Bischof and Martin Bücker,
Computing Derivatives of Computer Programs
Modern Methods and Algorithms of Quantum Chemistry: Proceedings, Second Edition
NIC-Directors, 2000

[4] Michael Innes
Don't Unroll Adjoint: Differentiating SSA-Form Programs

[5] Cristian Homescu
Adjoints and Automatic (Algorithmic) Differentiation in Computational Finance
arXiv, 10 Jul 2011

[6] Bert Speelpenning
Compiling fast partial derivatives of functions given by algorithms
Technical report, Illinois Univ., Urbana (USA). Dept. of Computer Science, 1980

[7] Swift for Tensorflow
(retrieved May 2019)

[8] Tzu-Mao Li
Differentiable Visual Computing
PhD Thesis at MIT, June 2019

[9] Phil Ruffwind
Reverse-mode automatic differentiation: a tutorial

[10] Vassilev, V., Vassilev, M., Penev, A., Moneta, L. and Ilieva, V., 2015. Clad—automatic differentiation using Clang and LLVM. In *Journal of Physics: Conference Series* (Vol. 608, No. 1, p. 012055). IOP Publishing.

[11] Dan Piponi
Automatic Differentiation, C++ Templates and Photogrammetry

[12] Marco Foco
PAN repository - experiments in differentiability

[13] Maxima Open Source Project Website

[14] Maplesoft's Maple Website

[15] Wolfram's Mathematica Website

[16] Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, Frédo Durand
DiffTaichi: Differentiable Programming for Physical Simulation

[17] A. Baydin, B. Pearlmutter, A. Radul, and J. Siskind
Automatic Differentiation in Machine Learning: a Survey
Journal of Machine Learning Research, 2018

[18] Various Authors (Matthieu Pulver for `autodiff.hpp`)
Boost.Math repository
[19] Wenzel Jakob
Enoki library
[20] Vassil Vassilev, et al
CLAD repository