Report - Taylor-based reachability analysis of continuous and hybrid systems

Jules Pénuchot

The goal of this project is to implement a Taylor-Based reachability analysis program for continuous and hybrid systems. These analyses provide bounding boxes for variables over time given their behaviors described as differential equations.

Introduction

My approach for this project was to provide a set of tools to efficiently **implement** Taylor-based analyses as C++ code : having these informations at compile-time allow us to use **template meta-programming** to specialize functions and therefore having analyses as fast as handmade implementations, with very high-level and thus clear and concise code.

The project was split in 3 major parts :

- `include/interval` : A generic interval algebra library
- `include/differentiation` : An automatic differentiation library for Taylor coefficients
- `(not implemented)` : The actual analysis library

We'll start off with the interval algebra library, then we'll have a look at the differentiation library.

Interval algebra

The interval algebra library provides a simple interface to manipulate generic intervals :

```
template<typename T>
struct interval
{
  T l, r;
};
```

Arithmetic operators such as `+`, `-`, `/` and `*` have been overloaded to allow writing operations with them. Therefore declaring and manipulating intervals is as simple as this :

- Type inference for easier declaration (C++11)

```
//  Using make_interval allows simple declaration using type inference :
//  a's type will be inferred as interval<double>
auto a = make_interval(20., 21.);
```

- Structured binding (C++17)

```
//  as_const_ref_tuple returns a const ref tuple to the interval's bounds,
//  allowing us to do structured binding on intervals easily and therefore
//  access bound values in a more concise and readable manner.
const auto& [l, r] = as_const_ref_tuple(a);
```

- Algebra

```
//  Simple algebra mixed with type inferrence,
//  b's type will be inferred as the expression's return type,
//  ie. interval<double>
auto b = (2. + make_interval(l + 1, r + 2)) * 2.;
```

- Printing

```
//  Operator << has been overloaded too, making intervals easy to print
cout << b << '\n';
```

All these high level features come at strictly zero cost in terms of performance, following C++'s zero-cost abstraction principle. Bounds can be represented by any kind of structure that has overloads for the 4 arithmetic operators, and a specialization for `std::numeric_limits` (required by the `/` operator).

Differenciation

The differenciation library aims to implement two strategies :

- A strategy based on the construction of an abstract template structure for expressions, that delivers derivates of functions at **compile-time** thanks to template meta-

programming (recommended way)
- A more naive strategy based on function sampling, that was times faster to implement but not as precise as the first one

Both are incomplete as they cannot derivate n-arguments functions (yet).

Before talking about these strategies, I wanted to talk first about how meta-programming, including concepts like parameter packs, can help making a very efficient library for automatic differenciation.

Current libraires like FADBAD++ generate DAGs at runtime to represent expressions and finally derivate them, still at runtime, by generating runtime representations of the derivatives. These representations are stored in structures that have to be interpreted by the program itself. The interpretation actually takes more time than the computation itself as it adds up to a series of jump that take way longer than arithmetic operations.

To eliminate the time taken by the interpretation, another more efficient approach would be to store expressions as template type representations, and derivate the functions at compile-time.

Meta-programming : a few examples

Let's first have a look at template meta-programming with `std::tuple` :

```
using namespace std;

auto test = make_tuple(10.2, "Hello", "another string", '\n');
```

Tuples in C++ are an abstraction on structs. The elements can be accessed using `std::get` by providing indexes as **template parameters** that way :

```
auto first  = get<0>(test); // 10.2
auto second = get<1>(test); // "Hello"
```

Or by using **structured binding**, introduced by C++17 (analogous to pattern matching in functional languages):

```
// first, second, third and fourth now contain the values stored in test
auto [first, second, third, fourth] = test;
```

That is when you want to access elements separately and distinctively. However, you might want to write code that will remain the same whatever you put in your tuple, for example a pretty printer :

```
auto pretty_print = [](auto... Args)
{
  ( (std::cout << Args << ' ') , ... );
};

//  We can call pretty_print with as many arguments as we want,
//  each of different type
pretty_print("Hello", "World", '!');

//  Or by applying a tuple using std::apply, that gives each
//  element of the tuple as parameter a to the function
std::apply(pretty_print, make_tuple("Hello", "World", '!'));
```

This is done by using the parameter pack expansion syntax :

`( expr , ... )` where `expr` is an expression that contains an unexpanded parameter pack.

In this case, calling pretty_print is **exactly the same** as writing a series of `std::cout` ; each argument is known at the compile time, so the compiler just unfolds the expression then compiles it for each case.

What can be done

Similar methods can be used in our case to represent expressions as template types (ie. the expression is stored in the **type** and not the actual runtime value of the variable that represents it).

```
//  Arithmetic operations
enum arith_op { add, sub, mul, div };

template<arith_op, typename... Ts>
struct op
{
  std::tuple<Ts...> ts;
};

template<arith_op Op, typename T1, typename T2>
using binop = op<Op, T1, T2>;
```

Here, `op` can represent any operator within an expression : its nature is represented by an enum as a template parameter, and its children are also types (T1 and T2) which can eventually store a value at runtime; this will be useful if we want to refer to other values (references/pointers) in our expressions.

At the moment, only the construction of these expressions is partially implemented using operator overloading, which means that building up will eventually be as easy as writing an arithmetic expression in C++.

However the derivation part isn't implemented due to lack of time.

Luckily I had enough time to come up with another way to implement automatic differentiation for any function of arity 1 :

```
template<typename F, typename D>
inline D derivate(F f, D i, D dt)
{
  return (f(i + dt) - f(i)) / dt;
}
```

`derivate` can be used to derivate any function at point `i` by sampling it on a delta-time of `dt` , and can be used like that :

```
derivate([](auto x)
{
  return x * x;
}, 10., .0001);
```

Thanks to templates, it supports any kind of function, even polymorphic lambda functions (C++11) as shown above.

Conclusion

A lot of work remains for this project, the example shown in the course was easy to understand as it features a 1-D function but the examples for the project were much more complex and I didn't understand how the analysis would work in these cases until today (the last day).

I didn’t come up with an implementation at the end of the project, but I spent a lot of time digging for resources that helped me improve my comprehension of Taylor models. I strongly believe that the approach I introduced could lead in significantly faster simulations and I am motivated to work further to eventually end up with an implementation to benchmark it against existing implementations.