Add numerical differentiation support in clad

Google Summer of Code 2021 - CERN-HSF Garima Singh

Project Overview

In mathematics and computer algebra, automatic differentiation (AD) is a set of techniques to numerically evaluate the derivative of a function specified by a computer program. Automatic differentiation is an alternative technique to Symbolic differentiation and Numerical differentiation (the method of finite differences). Clad is based on Clang which provides the necessary facilities for code transformation. The AD library can differentiate non-trivial functions, find a partial derivative for trivial cases, and has good unit test coverage. In several cases, due to different limitations, it is either inefficient or impossible to differentiate a function. For example, clad cannot differentiate declared-but-not-defined functions. In that case, it issues an error. Instead, clad should fall back to its future numerical differentiation facilities.

Project Objectives

- Add numerical differentiation support for forward mode.
- Add numerical differentiation support for reverse mode.
- Provide numerical differentiation through a dedicated interface.
- Provide error estimates for the numerical differentiation.

Final Results

PR (Merged): https://github.com/vgvassilev/clad/pull/261

Initial Stage

The main aim of the project was to be able to have a backup plan in case clad fails to differentiate a given function call expression. Since clad is a source transformation tool, source visibility is a very important aspect of whether clad can differentiate a function call or not. Earlier, the only way for clad to differentiate functions whose source was *not visible* (i.e. external library functions) was through having the user define a custom derivative by hand. Even though clad does cover a lot of the frequently used math functions and provides in-built custom derivatives for the same, there is still a large blindspot for clad in terms of these `invisible` functions. One of the ways to overcome this was simply defining a numerical differentiation function as follows:

```
template <typename F> double central_difference(F f, double x) {
    // A simple 2 point central difference numerical method here.
}
```

This was the initial idea of the implementation and this is also a very similar way of how numerical differentiation is implemented in libraries like gnu_gsl and boost. However, as we can see, this heavily limits the kind of functions that can be differentiated. A list of the "eligibility" criteria of a function to undergo numerical differentiation is listed below:

- Functions with single arguments (or equivalent).

- Function with arguments that can be converted to a `double`. This means functions with user-defined types as inputs are largely not supported.
- Function with scalar inputs only, which means no arrays or pointer types (even to built-in base types).

Current Stage

The project’s current state eliminates all of the problems faced above by heavily using templates and parameter packs. We currently provide two interfaces packaged in a single template header file which allows uses to easily use the provided functions as standalone, without any extra link dependencies. The two interfaces and their usage are mentioned as follows:

- `forward_central_difference` - The numerical differentiation function which differentiates a multi-argument function with respect to a single argument only. The position of the argument is specified by the user or Clad. This interface is mainly used in clad's forward mode for call expressions with single arguments. However, it can easily be extended for jacobian-vector products as well. The signature of this method is as follows:

```
template < typename F, typename T, typename... Args>
  precision forward_central_difference(F f, T arg, std::size_t n, bool printErrors,
Args&&... args){
    // Here as you can tell, we have enough type generality that we can accept
    // functions with a variety of input types.
    // Here:
    // f(args...) - is our target function.
    // n - is the position of the parameter with respect to which we want a
derivative.
    // printErrors - A flag to enable printing of error estimates.
}
```

- `central_difference` - The numerical differentiation function which differentiates a multi-argument function with respect to all the input arguments. This function returns the partial derivative of the function with respect to every input, making it a suitable candidate to use in clad's reverse mode. The signature of the method is as follows:

```
template <typename F, std::size_t... Ints,
            typename RetType = typename clad::return_type<F>::type,
            typename... Args>
  void central_difference(F f, clad::tape_impl<clad::array_ref<RetType>>& _grad, bool
printErrors, Args&&... args) {
    // Similar to the above method, here:
    // f(args...) - is our target function.
    // grad - is a 2D data structure to store all our derivatives as
grad[paramPosition][indexPosition]
    // printErrors - A flag to enable printing of error estimates.
}
```

Since we use some `math` functions for numerical differentiation, we have to link against the library. To avoid this and give the users a choice not to enable numerical differentiation, we have the functionality to define a macro `CLAD_NO_NUM_DIFF` at the target program's compile time.

How it works behind the scenes:

Before we can get a gist of how things work, let us look at the general formula for calculating derivatives of a function `f`. In clad we use the five-point stencil method, but for simplicity we assume the vanilla central difference method here:

> *$df/dx_i = (f(\ldots, x_i + h, \ldots) - f(\ldots, x_i - h, \ldots)) / (2 * h)$*

the above formula gives a good estimate of the partial derivative of `f` with respect to $\mathbf{x_i}$. Now what remains is to see how this comes together in code.

So we have one main function that comes into play:

```
// This function basically enables us to 'select' the correct parameter to update.
// Without this function, we will not be able to figure out which x should be updated
to x ± h.
template <typename T>
T updateIndexParamValue(T arg, std::size_t idx, std::size_t currIdx, int multiplier,
precision& h_val,...) {
    if (idx == currIdx) {
        // selects the correct ith term.
        // assigns it an h_val (h)
        // and returns arg + multiplier * h_val essenially.
    }
    return arg;
  }
```

Here, `Idx` is the current parameter we are on and `currIdx` is the parameter we want to differentiate with respect to in this pass. If the indices do not match, we return the argument unchanged.

Now, we apply this function to all the arguments in our `args` parameter pack and forward the same to our target function `f`.

```
fxh = f(updateIndexParamValue(args, indexSeq/*integer index sequence for the parameter
pack,
                Args allows us to give an index to each parameter in the pack.*/,
                i /*index to be differentiated wrt*/,
                /*±1*/,
                h/*we expect this to be returned*/,
                /*other params omitted for brevity*/)...);
```

The above line results in the calculation of $f(\ldots, x_i \pm h, \ldots)$. Finally the whole algorithm for calculating gradient of a function is as follows:

> *for each `i` in `args`, do:*
>
> *`fx1 := f(updateIndexParamValue(args, idexSeq, i, 1, h, /*other params*/)...)`*
>
> *`fx2 := f(updateIndexParamValue(args, idexSeq, i, -1, h, /*other params*/)...)`*
>
> *`grad[i][0] := (fx1 - fx2)/(2 * h)`*
>
> *end for*

Challenges/Objectives Achieved

We achieved some really interesting results that are enumerated below:

- **Differentiating multi-arg function calls with minimum code duplication**

Due to the use of templates, we were able to write concise code to implement both forward and reverse numerical difference functions.

- **Differentiating calls with pointer/array input**

A big problem with non-scalar inputs such as arrays or vectors is that they are implicitly pass-by-reference, which means that may not be reusable as input to the same function and are expected to result in the same output. Since numerical differentiation by clad requires a function to be executed at least `4*num_arguments`, we have to make a copy of the non-scalar input and use that to pass to the function instead. The way we achieved this is by implementing a simple memory buffer manager that is responsible to allocate and free temporary memory allocated for this purpose.

Use case:

A potential use case here would be differentiating tensors or structures such as queues/linked lists/stacks. Taking an example: suppose we want to support differentiating a linked-list input, we would have to overload the `updateIndexParamValue` as follows:

```
MyLinkedList* updateIndexParamValue(MyLinkedList* arg, std::size_t idx, std::size_t
currIdx,
int multiplier, precision& h_val, std::size_t  n = 0, std::size_t  i = 0) {
    if (idx == currIdx) {
        // Malloc n pointers of type MyLinkedList using BufferManager. Connect them
all together.
        // copy all values from arg to temp.
        temp.copy(arg, n);
        // Now get the ith value
        MyLinkedList* val = temp.getAtPos(i);
        // update the data
        val->data += multiplier * h_val;
        // Return the copy.
        return temp;
    }
    return arg;
  }
```

- **Differentiating user-defined types**

Another significant issue that clad does not currently handle is user-defined types. In the standalone mode, we can tackle both scalar (structs, classes, etc.) and non-scalar (struct arrays/pointers, etc.) user-defined data types. We provide a function that can be specialized by the user for their data type. These specialized functions will then tell clad how the user-defined data type has to be differentiated.

Use case: An interesting use case for this can be numeric types that have custom implementations or are 'concealed' in a way. That means, we can handle things such as:

```
// Essentially a typedef over double but cannot be implicitly
// converted to double.
struct MyDouble {
    double x;
};
```

Or more complex things such as fixed-length floating-point numbers (eg. java's BigInteger). Another interesting example handling such a case is described in a demo here.

- **Lightweight dedicated interface**

The standalone numerical diff interface is very lightweight and consists of only a single header file. Since we currently have very low dependence on clad, it is super easy to port these codes.

- **Printing of error estimates**

Since numerical differentiation is a way to *estimate* the derivative, it is essential to keep track of any associated errors. Users can choose to enable printing of error estimates for numerical differentiation calls in clad using a command-line flag. Doing so will enable the printing of error information on standard output.

Related future work: These estimates may also be propagated further through the function to see what kind of effects they have on the final output. They may also be coupled with clad's floating-point error estimation framework to get interesting insights into the stability of functions. Issue #295 tracks the progress for this.

Roadblocks

The roadblocks faced during development as listed as below:

- **Developing a general interface**

The biggest roadblock of this project was probably how to build a generic interface that could differentiate functions with any signature. We wanted the interface to be simple and central enough that users could use it standalone. There was discussion on multiple design choices we could have taken but most of them seemed to either be very elaborate or very specialized.

- **Handling pass-by-reference input arguments**

Pass-by-reference arguments may be modified by the function and hence may result in that modification propagating into other function calls for numerical differentiation. Hence it became necessary to pass on a clone of these arguments each time a function call is requested. This also implied that we had to have a way to manage memory that was independent of type. We achieved this by creating a `BufferManager` that allocates and destroys memory related to those objects. As of now, we can only handle allocating types that are trivially destructible.

Miscellaneous Contributions

I was able to make a few miscellaneous contributions to clad which included major memory leak fixes and general improvement on clad.

You can find all my created PRs and issues here.

Conclusions

I was able to complete the major objectives set by the project mentors. In the process, I was also able to further my knowledge in the field of computational mathematics, compilers and improved my coding skills.

Acknowledgement

I would like to acknowledge my mentors Vassil Vassilev and Alexander Penev without whose guidance and help the completion of this project would not be possible. I would also like to thank the CERN-HSF organization mentors and the GSoC team to have provided me with such an amazing opportunity!

> *You may contact me @ garimasingh0028@gmail.com!*