Plugins
Expression templates
Generic programming

Matteo Caldana

18/05/2023

Exercise 1: quadrature rules with plugins

Let us consider the following simple example compiled with `g++ 01-main.cpp -o main`

```cpp
void func() {}

int main() {
  func();
  return 0;
}
```

We can use the `nm` command to show information about symbols in the executable we obtained (it works also with object files, and object-file libraries). In particular, by running the command `nm -a main | grep func` we will obtain something like:

```
00000000004010f6 T _Z4funcv
```

In order to support overloading, the C++ compiler has mangled the name of the function `func` with the type of the input parameters (v for `void`).

Exercise 1: quadrature rules with plugins

This is more evident when we consider an example with function overloading

```
void func() {}
int func(int x) { return x; }
double func(const int x, double y) { return y; }

int main() {
  func();
  return 0;
}
```

By using again the `nm` we will obtain something like:

```
00000000004010fd T _Z4funci
0000000000401109 T _Z4funcid
00000000004010f6 T _Z4funcv
```

We have three different symbols, one for each overload. However, notice how only the parameter types (and not the return type) affect the name mangling. Indeed, you can't overload methods based on return type.

Exercise 1: quadrature rules with plugins

What happens in C where there is no function overload? We can test it by using the `extern "C"` directive.

```
void func() {}
extern "C" { void another_func() {} }

int main() {
  func();
  return 0;
}
```

Now, the symbols defined in the executable concerning the functions are

```
00000000004010f6 T _Z4funcv
00000000004010fd T another_func
```

Notice how `another_func` has no name mangling. This feature will come in handy when dynamically loading plug-ins since we do not want to read a mangled name from the shared library.

Exercise 1: quadrature rules with plugins

- Implement a code that enables the user to compute the integral of a scalar function by selecting the quadrature rule as the name of a dynamically loadable object;
- The dynamically loadable object should define a function with the following signature: `double integrate(const std::function<double (double)> &, double a, double b)`.
- Implement plugins for midpoint, trapezoidal and Simpson's rule.
- Implement a plugin for quadrature with adaptive refinement.

Exercise 2: expression templates

This exercise is inspired by
`https://www.modernescpp.com/index.php/expression-templates`.

The starting code provided implements a custom `MyVector` class.

- Given two vectors $\mathbf{x}$ and $\mathbf{y}$, implement the sum and the multiplication operators to compute the quantity $2\mathbf{x} + \mathbf{y}^2$, where the operations are intended component-wise.
- Provide an expression template implementation and compare the computational costs with respect to a non-template implementation.

See also: `https://devtut.github.io/cpp/expression-templates.html` and
`https://en.wikipedia.org/wiki/Expression_templates`.

Exercise 3: 1D interpolator using generic programming

This exercise (inspired by `Examples/src/Interp1D`) shows an example of a generic piecewise linear 1D interpolator, with an application to vectors of couples of values (representing the interpolation nodes and the corresponding values) and to two separate vectors (one for the nodes and one for the values).
It is rather generic and accepts as an input iterators to any container with bi-directional iterators. Bi-directionality is indeed only needed for extrapolation on the right.