Instructions

The numerical results of the thesis [1] have been computed on Linux (version 5.10.64) using the GCC compiler (version 4.8.5) with the options `-O3 -std=c++11`. The results can be reproduced as follows.

Article PIV

`example1()` reproduces Example 7.1
`example2()` reproduces Example 7.2
`example3()` reproduces Example 7.3
`example4()` reproduces Example 7.4
`example5()` reproduces Example 7.6
`example6()` reproduces Example 7.7
`example7()` reproduces Example 7.8
`example8()` reproduces Example 7.9
`example9()` reproduces Example 7.10
`example10()` reproduces Example 7.11
`example11()` reproduces Example 7.12

Article PII

The numerical results of Article PII were obtained with `example12()` and `example13()`, but the computations were computed in quadruple precision with an older version of the code. One may replace the code in `SmallSimplexPartition3D::getBodyNodesCustom` with `return mesh_old_ptr->getBodyNodes(b);` to restore the old behaviour; this is optional but enables one to reproduce exactly the same results.

Quadruple precision was enabled with the libquadmath library for the GCC compiler, which was used with the options `-O3 -fext-numeric-literals -std=c++11`. To reproduce the results in quadruple precision, the following changes in code are required.

In GFD/Types/Types.hpp, include `<quadmath.h>` and `<sstream>` and add the following code after the type definitions:

```
typedef __float128 quadruple_t; //rename the quadruple precision type
//for printing __float128
inline std::ostream& operator<<(std::ostream& out, __float128 f) {
   char buf[200];
   std::ostringstream format;
   if (out.flags() & out.scientific)
      format << "%." << (std::min)(190L, out.precision()) << "Qe";
   else
      format << "%." << (std::min)(190L, out.precision()) << "Qf";
   quadmath_snprintf(buf, 200, format.str().c_str(), f);
   out << buf;
```

```
   return out;
}
```

Replace all instances of `double` with `quadruple_t`.

Change all instances of the following functions:
`std::abs` → `fabsq`,
`std::pow` → `powq`,
`std::sqrt` → `sqrtq`,
`std::sin` → `sinq`,
`std::cos` → `cosq`,
`std::exp` → `expq`.

Add the suffix `q` to all numeric literals in NumericalIntegration.cpp (lines 7-590) and in the polynomial test functions in
`polynomialTestWhitney0Forms3D`,
`polynomialTestWhitney1Forms3D`,
`polynomialTestWhitney2Forms3D`, and
`polynomialTestWhitney3Forms3D`
in ErrorAnalysisFunctions.cpp (for example, `64.0 / 75.0` → `64.0q / 75.0q`).

Remarks

The same code should work on any system supporting the C++11 standard, but please note that the exact output may vary slightly between different systems.

Integrals required in computations may optionally be saved to files by preparing a directory structure as follows:

```
for i in {1..12}
do
    mkdir -p Files/2D/order$i/barycentric/DefaultTriangle
    mkdir -p Files/2D/order$i/barycentric/RightTriangle
    mkdir -p Files/2D/order$i/circumcentric/DefaultTriangle
    mkdir -p Files/2D/order$i/circumcentric/RightTriangle
    mkdir -p Files/3D/order$i/barycentric/BccTetrahedron
    mkdir -p Files/3D/order$i/barycentric/CubeTetrahedron
    mkdir -p Files/3D/order$i/circumcentric/BccTetrahedron
    mkdir -p Files/3D/order$i/circumcentric/CubeTetrahedron
done
```

When appropriate, the integrals that have been saved to files can be reused by setting the parameter values `loadMatricesFromFile = true` or `loadIntegralsFromFile = true` when using the SmallSimplexPartition class. Especially for high orders, the use of precomputed integrals results in significant speed-ups.