MATH 676

–

Finite element methods in scientific computing

Wolfgang Bangerth, Texas A&M University

Lecture 18:

Debug vs. optimized builds

Performance vs ease of development

Scientific computing lives in a tough spot:

- Many computations are long and expensive
- We thus care about performance

- But simulators are also expensive to develop
- **Recall:** Nothing is as expensive as your own work time!
- We need to make it simpler to develop codes

General rule: These two goals can not be achieved at the same time. But we can get there partway.

Performance vs ease of development

Approach 1: Dimension independent programming

- 3d computations are far more expensive than 2d
- They are consequently difficult to debug

- With deal.II's dimension templates, develop code once
- Debug in 2d where this is cheap
- Run debugged production code in 3d

Performance vs ease of development

Approach 2: Debug vs optimized builds

- Aid programmer by making sure that deal.II functions check arguments, verify consistent internal state, etc
- Abort program when a condition is not verified
- This allows much simpler debugging (see the lecture on Eclipse)
- Technical implementation via "*assertions*"

Note: deal.II currently (early 2013) has ~8,000 such assertions.

Preconditions

Preconditions:

- Usefulness of input arguments can be expressed in the form of "*preconditions*"
- Preconditions are often recorded as part of the documentation of a function
- The vast majority of bugs are violations of preconditions

Example:

- When accessing element i of a vector of length N then
  $0 <= i < N$

Preconditions

Example:

- When accessing element *i* of a vector of length *N* then
  0 <= i < N
- Typical implementation:

```
class Vector {
    double *data;
    unsigned int size;
public:
    double operator[] (const unsigned int i) const {
        assert (i < size);
        return data[i];
      }
};
```

Preconditions

Example:

- When accessing element *i* of a vector of length *N* then
  $0 <= i < N$
- Implementation in deal.II:

```
class Vector {
    double *data;
    unsigned int size;
public:
    double operator[] (const unsigned int i) const {
        Assert (i < size, ExcIndexRange (i, 0, size));
        return data[i];
      }
};
```

Postconditions

Postconditions:

- State things about the return value of a function
- State things about the state of an object after a call
- Postconditions are often recorded as part of the documentation of a function
- Violations often indicate programming bugs

Example:

- When computing the norm of a vector, the result must be non-negative

Postconditions

Example:

- When computing the norm of a vector, the result must be non-negative
- Implementation in deal.II:

```
class Vector {          /* … rest as before … */
   double norm() const {
        double sum = 0;
        for (unsigned int i=0; i<size; ++i)
            sum += data[i]*data[i];
        const double return_value = std::sqrt(sum);
        Assert (return_value >= 0, ExcMessage("..."));
        return return_value;
      }
```

Internal conditions

Internal conditions:

- In complex functions, one often knows that at a particular point (say after step 3 of the algorithm) a particular condition has to hold
- Verify this via an assertion!
- Violations often indicate programming bugs

Example:

- We only tolerate one hanging node per cell
- During mesh refinement we should never refine a cell whose neighbor is coarser!

Pre-, post and internal conditions

Within this taxonomy:

- We can never get rid of precondition assertions because there will always be new calls to functions
- One *could* get rid of post- and internal condition assertions once the function has been verified
- But this is a bad idea: we might have to modify it later

General rule from years of experience: There can never be enough assertions. You should always add as many as you can and never delete them!

The problem with assertions

Assertions have drawbacks:

- They are slow: they make code 4–10 slower
- They lead to to very large executables

Debug vs optimized mode

Solution:

- deal.II is built twice:
  - libdeal_II.g.so: contains all of the assertions ("*debug mode library*")
  - libdeal_II.so: does not contain assertions, uses compiler optimizations ("*optimized/release mode library*")
- When you develop your program:
  - compile in debug mode with all assertions
  - link against debug mode lib
  - find all your bugs :-)
- Do production runs without assertions and in release mode

MATH 676

–

Finite element methods in scientific computing

Wolfgang Bangerth, Texas A&M University