

Programming Techniques for Scientific Simulations I: A Detailed Textbook

Based on lecture slides

October 27, 2025

Contents

1	Introduction	3
2	Error Handling and Exceptions	4
2.1	The Fundamental Problem: What Happens When Code Fails?	4
2.2	Classic (and Clunky) Strategies for Error Handling	4
2.3	The Modern C++ Approach: Exceptions	6
2.4	How to <code>throw</code> an Exception	7
2.5	The Standard Exception Library: <code><stdexcept></code>	7
2.6	How to <code>catch</code> an Exception	8
2.7	Advanced Exception Details	9
2.8	A Complete Example Walkthrough	10
3	Software Testing: Gaining Confidence in Your Code	13
3.1	Types of Testing	13
3.2	Example: Testing a Fibonacci Function	14
3.3	CTest: The CMake Test Runner	14
3.4	Catch2: A Real Testing Framework	15
4	Timing Your Code: The Science of Speed	16
4.1	How to Time Code with <code><chrono></code>	16
5	Monte Carlo Methods: Solving Problems with Randomness	18
5.1	The Problem: Multidimensional Integration	18
5.2	The Solution: Monte Carlo Integration	19
5.3	Example: Calculating π with Monte Carlo	19
5.4	What is a "Random" Number?	20
5.5	How PRNGs Work: The LCG	21
5.6	The Modern C++ <code><random></code> Library	21
5.7	Step 1: Choose Your Engine	21
5.8	Step 2: Seed Your Engine	22
5.9	Step 3: Choose Your Distribution (The "Refinery")	22

5.10	Step 4: Put It All Together (The π Example)	23
5.11	Best Practices for Random Numbers	23
6	Documentation: Writing Code for Humans	24
6.1	The "Contract" for a Function	24
6.2	Doxxygen: The Automatic Documentation Generator	24
6.3	Doxxygen Syntax Example	25
6.4	Integrating with CMake	25
6.5	A Practical Guide to Using Doxygen	26
7	Conclusion	29

1 Introduction

Welcome to this course on programming techniques for scientific simulations. A **scientific simulation** is a powerful tool that allows us to build a "virtual laboratory" inside a computer. We use it to model complex real-world phenomena, such as the orbit of planets, the intricate dance of molecules in a chemical reaction, or the dynamics of a biological population.

To build these simulations, just "knowing the physics" isn't enough. We also need to write code that is:

- **Robust:** It doesn't crash or fail unexpectedly. It can gracefully handle errors.
- **Correct:** It produces the right answer. We need to be able to prove and verify this.
- **Professional:** It is easy for others (and our future selves) to read, understand, and modify.

In this first part of our study, we will cover five fundamental topics that form the bedrock of high-quality scientific programming:

1. **Exceptions:** The modern C++ way to handle errors and exceptional situations.
2. **Testing:** The process of verifying that our code actually does what we think it does.
3. **Random Numbers:** Understanding how computers generate and use randomness, which is the heart of many advanced simulation methods.
4. **Timing:** How to measure the speed of our code to find and eliminate bottlenecks.
5. **Documentation:** The art of writing explanations for our code so that it can be understood and used by others.

Let's begin with the first, and perhaps most critical, topic: what to do when things go wrong.

2 Error Handling and Exceptions

2.1 The Fundamental Problem: What Happens When Code Fails?

Imagine you've written a function, a self-contained "worker" designed to perform a specific task. For example, a function `calculate_step()` that computes the next position of a planet in its orbit. During its execution, this function detects a critical error—perhaps the input data is corrupt, or a calculation results in an impossible number (like dividing by zero).

The function is now in a state where it **cannot complete its task**.

This is not a hypothetical problem; it happens constantly in scientific computing.

A Real-World Example: A Failing Time Step In physics and engineering, many simulations involve solving differential equations. These are equations that describe *change* over time. We often write this as:

$$\dot{y} = f(t, y)$$

This notation simply means "The rate of change of some value y (written as \dot{y}) depends on the current time t and the current value y ."

To solve this on a computer, we can't get a continuous answer. Instead, we take tiny, discrete "time steps" to find the *next* value (y^{n+1}) based on the *current* value (y^n). One common method is the **implicit midpoint method**, which has the formula:

$$y^{n+1} = y^n + \Delta t \cdot f\left(t^n + \frac{\Delta t}{2}, \frac{y^n + y^{n+1}}{2}\right)$$

Notice the problem: the value we want to find, y^{n+1} , appears on *both sides* of the equation! We can't just compute it directly. To solve this, we must use a numerical algorithm called a **root solver**.

A root solver is like a "guess-and-check" machine. It tries a value for y^{n+1} , sees how "wrong" the equation is, and then makes a better guess, repeating until it "converges" on the right answer.

But what if the root solver fails? Maybe no solution exists, or the algorithm just gives up after 1000 guesses. If the root solver fails, our `timeStep()` function *cannot* calculate y^{n+1} . It has failed. It must report this critical error to whatever part of the program *called* it. What should it do?

2.2 Classic (and Clunky) Strategies for Error Handling

For decades, programmers have used several strategies to deal with this.

Strategy 1: Terminate the Program This is the simplest and most drastic option. Just crash the program.

```
1 #include <iostream>
2 #include <cstdlib> // for std::abort
3
```

```

4 void my_function() {
5     std::cout << "Error: Root solver failed to converge!" << std::endl
6     ;
7     std::abort(); // Halts the program. Immediately.

```

- **Analogy:** A chef, realizing they're out of salt, burns down the entire restaurant.
- **Pros:** Very easy to write. You can't miss the error.
- **Cons:** Catastrophic for the user. No chance to save work, no graceful shutdown. Unacceptable for almost any real application.

Strategy 2: Return Status/Error Codes This is the most common method used in older C-style libraries (like BLAS, LAPACK, GSL). The function's return value is used to signal success or failure.

```

1 // Convention: 0 means success, non-zero means error
2 int do_stuff() {
3     // ... try to do work ...
4     if /* something bad happened */) {
5         return 1; // Error code 1: "Solver failed"
6     }
7     if /* something else bad happened */) {
8         return 2; // Error code 2: "Input was invalid"
9     }
10    return 0; // Success!
11 }
12
13 // The *caller* must check the return code
14 int status = do_stuff();
15 if (status != 0) {
16     // Handle the error based on the value of 'status'
17 }

```

- **Analogy:** The chef hands the dish back to the waiter (the caller) with a small, printed note. "Code 1: Out of salt."
- **Pros:** Gives the caller a *chance* to handle the error.
- **Cons:** It's *easy to forget* to check the return value. If the caller doesn't check status, the program continues in a broken state. It also "pollutes" the function's return value; a function that should return a calculation (e.g., a double) now has to return an int or some complex "status-and-value" object.

Strategy 3: Global Error Flags This method, used by C's `errno`, stores the error status in a global variable.

```

1 // A global error variable (this is bad practice)
2 int global_error_flag = 0;
3
4 void do_stuff() {
5     // ... work ...
6     if /* error */) {
7         global_error_flag = 1;
8         return;
9     }
10 }
11
12 // The *caller* must check this global flag
13 global_error_flag = 0; // Must reset it first!
14 do_stuff();
15 if (global_error_flag != 0) {
16     // Handle error
17 }
```

- **Analogy:** The chef flips a "Problem" switch in the kitchen. The waiter has to remember to look at that switch *immediately* after making an order.
- **Cons:** This is widely considered terrible practice. Global variables are dangerous; any part of the program can change them, leading to "spooky" bugs. It's not "thread-safe" (if multiple functions run at once, they'll all try to use the same flag).

This leads to the ideal philosophy: **The detection of an error (in the callee) should be separate from the handling of an error (in the caller).**

2.3 The Modern C++ Approach: Exceptions

Exceptions are the modern C++ mechanism for handling errors, designed to solve the problems above.

The Core Idea Think of an exception as an emergency signal.

1. A function (the *callee*) runs into a problem it **does not know how to deal with** (e.g., the root solver fails).
2. It **"throws"** an exception. This is like pulling a "Solver Failed" fire alarm.
3. When an exception is thrown, the function's **normal execution immediately stops**.
4. The C++ runtime system takes over, looking for an "alarm handler" (a `catch` block).
5. If the calling function has a `catch` block, it can "catch" the exception and handle the emergency.
6. If the calling function *doesn't* have a handler, the exception travels *up* the call stack to its caller, and so on.*

- If the exception reaches the very top (`main()`) and is *still* not caught, the program terminates (crashes).

This is a good thing! It's a "safety net." The program crashes instead of continuing to run in a broken, undefined state.

- Analogy:** The chef (callee) is out of salt. They can't fix this. They pull the "Out of Salt" alarm (`throw`). All cooking stops. The alarm rings. The waiter (caller) hears it, but they don't know how to handle it, so they let the alarm ring up to the restaurant manager. The manager (*their* caller) *does* have a plan. They "catch" the alarm, and their "handler" code runs: "Apologize to the customer, offer them a different dish." The crisis is handled gracefully.

2.4 How to `throw` an Exception

The syntax is simple: you use the `throw` keyword. An exception is just an **object**. You can throw (almost) anything.

```

1 // Example 1: Throwing a C-style string (generally bad practice)
2 if (n < 0) {
3     throw "n too small"; // Throws a 'const char*'
4 }
5
6 // Example 2: Throwing a standard exception object (good practice!)
7 #include <stdexcept>
8 if (index > max_size) {
9     throw std::range_error("index is out of range");
10 }
```

When you throw an object, the C++ runtime system looks for a `catch` block that matches the *type* of the object thrown. This is why throwing a `std::range_error` object is better than a simple string—it's much easier to catch specifically.

Stack Unwinding This is a crucial concept. When an exception is thrown, the program must "unwind" the stack.

- Analogy:** As the fire alarm rings and propagates *up* from the kitchen to the manager, every worker on each floor (local objects) must follow fire safety rules. They clean up their stations (destructors are called) and evacuate their floor (the function stack frame is destroyed) in an orderly fashion.

This "automatic cleanup" is one of C++'s most powerful features. It ensures that resources (like open files or allocated memory) are properly released even when an error occurs.

2.5 The Standard Exception Library: `<stdexcept>`

You should not invent your own error types. C++ provides a rich library of standard exceptions in the `<stdexcept>` header. They all "inherit" from a base "parent" class called `std::exception`.

- **Analogy:** Instead of one generic "Fire Alarm," the standard library gives you specific alarms for "Grease Fire," "Electrical Fire," and "Gas Leak." This lets the handler know *what kind* of emergency it is.

There are two main families:

1. **std::logic_error:** Represents *bugs in your code*. Problems that, in theory, could be prevented.
 - std::invalid_argument: You passed a bad value to a function.
 - std::out_of_range: You tried to access an element that doesn't exist (e.g., my_vector[100] when the vector only has 10 elements).
 - std::length_error: You tried to create something (like a std::vector) that is too big.
2. **std::runtime_error:** Represents errors that happen *during execution* due to external factors you can't always prevent.
 - std::overflow_error: A calculation resulted in a number too *large* to store.
 - std::underflow_error: A calculation resulted in a number too *small* to store.
 - std::range_error: A calculation result was invalid.

All these objects are created with a string message. You can retrieve this message by calling the .what() member function.

2.6 How to catch an Exception

To handle potential errors, you place "risky" code inside a **try...catch** block.

- **Analogy:** The try block is the "Dangerous Work Area" where you know an alarm might be pulled. The catch blocks are the "Emergency Responders" waiting just outside, each one trained to handle a *specific type* of alarm.

```

1 #include <iostream>
2 #include <stdexcept>
3
4 void do_something_risky(int x) {
5     if (x == 1) {
6         throw std::invalid_argument("x cannot be 1!");
7     }
8     if (x == 2) {
9         throw "Some other unknown error"; // Bad practice, but
10        possible
11    }
12
13 int main() {
14     try {

```

```

15     // --- This is the "Dangerous Work Area" ---
16     std::cout << "Trying risky code..." << std::endl;
17     do_something_risky(1);
18     std::cout << "This line will not be printed." << std::endl;
19
20 } catch (std::invalid_argument& e) {
21     // --- Responder for "invalid_argument" alarms ---
22     std::cerr << "Caught a specific logic error: " << e.what() <<
23     std::endl;
24
25 } catch (std::exception& e) {
26     // --- Responder for *any other standard* alarm ---
27     // Catches all logic_errors, runtime_errors, etc.
28     std::cerr << "Caught a standard exception: " << e.what() <<
29     std::endl;
30
31 } catch (...) {
32     // --- Responder for *ANYTHING ELSE* (the "catch-all") ---
33     // This catches the "Some other unknown error" string.
34     std::cerr << "Caught an unknown, non-standard error!" << std:::
35     endl;
36 }
37 return 0;
38 }
```

Key Syntax Points:

- **Order Matters:** The `catch` blocks are checked in order. You must put the most *specific* handlers first (e.g., `std::invalid_argument`) and the most *general* handlers last. `catch (...)` must always be last.
- **Catch by Reference (& e):** We use `std::exception& e` to "catch by reference." This avoids making an unnecessary *copy* of the exception object, which is more efficient.

2.7 Advanced Exception Details

Re-throwing Exceptions Sometimes, a handler can only *partially* deal with an error. It might log the error, but it can't fully resolve it. In this case, it can "re-throw" the *same* exception up to the next handler.

- **Analogy:** The restaurant manager hears the "Grease Fire" alarm. They log it in their incident report, but then they pull the *building-wide* fire alarm to evacuate everyone, passing the problem up to the fire department.

```

1 catch (std::exception& e) {
2     std::cerr << "Logging error: " << e.what() << std::endl;
3     // We logged it, but we can't fix it. Let the caller deal with it.
4     throw; // A plain 'throw;' re-throws the *original* exception
5 }
```

The `noexcept` Specifier There is one place an exception must *never* be thrown: in a **destructor**. A destructor is the cleanup function that runs when an object is destroyed (e.g., during stack unwinding). If an exception is *already* in flight (during stack unwinding), and a destructor throws a *second* exception, C++ has no way to handle this "double emergency." It will immediately call `std::terminate()` and crash your program.

By default, destructors are implicitly `noexcept(true)`. You can also "promise" that a normal function will *never* throw by marking it `noexcept`. This can help the compiler optimize your code.

```
1 void my_safe_function() noexcept {
2     // This function promises to never throw.
3 }
```

Exception Guarantee When you write a function, you must document its "exception guarantee" as part of its "contract" with the user:

- **No-throw guarantee:** The strongest promise. "This function will never throw."
- **Basic guarantee:** "If this function throws, the program state will remain valid, but the object might be changed."
- **Strong guarantee:** "If this function throws, the program state will be rolled back to *exactly* how it was before the function was called."

2.8 A Complete Example Walkthrough

Let's trace the execution of this program from the slides:

```
102 // (Main function from slide)
103 int main() {
104     simulation(10, 0.1); // Run simulation for 10 steps
105     return 0;
106 }
107 // (Simulation function from slide)
108 void simulation(int n_steps, double dt) {
109     double t(0.);
110     for (int step = 0; step < n_steps; ++step) {
111         std::cout << "Step " << step << std::endl;
112         try {
113             timeStep(step, dt); // Risky operation
114         }
115         catch (std::exception& e) {
116             std::cerr << "Error: " << e.what() << std::endl;
117         }
118         catch (...) {
119             std::cerr << "Error: I cannot handle... bailing out." <<
120             std::endl;
121             throw; // Re-throw the exception
122         }
123         t += dt;
124     }
125 }
```

```

124 }
125 // (timeStep function from slide)
126 void timeStep(int step, double dt) {
127     if (step == 3 || step == 6) {
128         throw std::domain_error("Solver failed at step " + std::
129             to_string(step));
130     }
131     else if (step == 9) {
132         throw "YOLO!!!!"; // Throwing a string
133     }
134     std::cout << "  step " << step << " successful" << std::endl;

```

Execution Trace:

- main() calls simulation(10, ...).
- **step = 0, 1, 2:** The try block calls timeStep(). The if conditions are false. "step X successful" is printed.
- **step = 3:** try calls timeStep(3, ...).
- timeStep() hits the first if. It **throws a std::domain_error object**.
- Execution in timeStep() *stops*.
- The runtime looks for a handler in simulation().
- It finds catch (std::exception& e). Since std::domain_error is a std::exception, this block runs.
- It prints "Error: Solver failed at step 3". The catch block finishes. The loop continues.
- **step = 4, 5:** Normal execution.
- **step = 6:** Same as step 3. A std::domain_error is thrown and caught.
- **step = 7, 8:** Normal execution.
- **step = 9:** try calls timeStep(9, ...).
- timeStep() hits the else if. It **throws "YOLO!!!!"** (a const char* string).
- Execution in timeStep() *stops*.
- The runtime looks for a handler in simulation().
- It checks catch (std::exception& e). A string is *not* a std::exception, so this block is **skipped**.
- It checks catch (...). This "catch-all" *does* catch the string.

- It prints "Error: I cannot handle... bailing out."
- It then executes `throw;`. This **re-throws** the "YOLO!!!!" exception.
- The exception now leaves the `simulation()` function and goes back up to `main()`.
- `main()` has **no `try...catch` block**.
- The exception is uncaught. The program **terminates** (crashes).

3 Software Testing: Gaining Confidence in Your Code

Writing the code is only the first step. The second, equally important step is verifying that it's *correct*.

"Testing shows the presence, not the absence, of bugs."
— Edsger Dijkstra

This famous quote is the foundation of software testing.

- **Analogy:** Think of testing as a doctor screening for a disease.
- Finding symptoms (a failed test) *proves* the presence of the disease (a bug).
- *Not* finding symptoms (all tests pass) does *not* prove you are perfectly healthy. It only **increases your confidence** that you are healthy.

Our goal is not to achieve impossible "proof of correctness," but to gain a high degree of confidence that our code behaves as we expect.

3.1 Types of Testing

Unit Testing A "unit" is the smallest testable piece of your program, typically a single function. **Unit Testing** means testing this function *in isolation*.

- **Analogy:** A chef performing a "unit test" on their ingredients. They taste *just* the flour. Then they taste *just* the sugar. They are testing each "unit" in isolation before combining them into a cake.
- **Example:** For a `fibonacci(n)` function, you would write separate tests:
 - Does `fibonacci(0)` return 0?
 - Does `fibonacci(1)` return 1?
 - Does `fibonacci(5)` return 5?
 - Does `fibonacci(10)` return 55?

Automated and Regression Testing You shouldn't have to manually run your tests every time. An **automated test suite** is *another program* whose only job is to run all your unit tests automatically.

This is critical for **regression testing**. A "regression" is when a new change to the code accidentally *breaks* old, existing features.

- **Analogy:** The chef creates a new recipe for cookies. A *regression* would be if this new cookie recipe somehow *broke* the old, perfectly good cake recipe.
- By running the *entire* automated test suite after *every* change, the chef (programmer) gets immediate feedback if their new cookie recipe (new feature) accidentally broke the cake (an old feature).

Testing Frameworks You don't have to build this "robot taster" from scratch. You can use a **testing framework**—a library that provides all the tools you need to write, run, and report on tests. Popular C++ frameworks include **GoogleTest** and **Catch2**.

3.2 Example: Testing a Fibonacci Function

Let's look at the "unit" we want to test. Good code practice separates the *declaration* from the *implementation*.

File 1: fibonacci.hpp (The Header / Interface) This file is the "public menu." It tells other files that the `fibonacci` function *exists* and what its "signature" (inputs and output) is.

```
1 #pragma once // Prevents this file from being #included twice
2
3 // Declares the function. Does not implement it.
4 unsigned int fibonacci(unsigned int n);
```

File 2: fibonacci.cpp (The Implementation / Logic) This file contains the "kitchen secrets"—the actual *code* that does the work.

```
1 #include "fibonacci.hpp" // Include the declaration
2
3 unsigned int fibonacci(unsigned int n) {
4     if (n == 0) {
5         return 0; // Base case
6     }
7
8     unsigned int a = 0;
9     unsigned int b = 1;
10
11    for (unsigned int i = 1; i < n; ++i) {
12        unsigned int next = a + b;
13        a = b;
14        b = next;
15    }
16
17    return b;
18 }
```

Our test program (e.g., `fibonacci_test.cpp`) would `#include "fibonacci.hpp"` and check if this implementation works as advertised.

3.3 CTest: The CMake Test Runner

The **CTest** tool is part of the CMake build system. It is *not* a testing framework (like GoogleTest). It is a **test runner**.

- **Analogy:** CTest is the restaurant manager who runs the "robot taster" program. CTest doesn't *do* the tasting. It just runs the `run_fib_tests` executable and waits for a "thumbs up" or "thumbs down."

- "**Thumbs Up**": The test program exits with **return code 0**.
- "**Thumbs Down**": The test program exits with a **non-zero return code**.

To use it, you add these commands to your CMakeLists.txt file:

```

1 # 1. Turn on the testing features
2 include(CTest)
3 enable_testing()
4
5 # 2. Tell CMake to build your test program executable
6 add_executable(run_fib_tests fibonacci_test.cpp fibonacci.cpp)
7
8 # 3. Register that executable as a test with CTest
9 # NAME: The human-readable name of the test
10 # COMMAND: The executable to run
11 add_test(NAME MyFibonacciTest COMMAND run_fib_tests)
```

After you `cmake` and `make` your project, you can now just type `make test`. CTest will run your `run_fib_tests` program and report "PASSED" or "FAILED" based on its exit code.

3.4 Catch2: A Real Testing Framework

Writing a test program that correctly returns 0 or 1 can be tedious. A framework like **Catch2** makes it much easier.

Catch2 is a "header-only" library. You just `#include "catch.hpp"`. It provides a set of powerful "macros" for defining and checking tests. Here is what `fibonacci_test.cpp` would look like using Catch2:

```

1 // This one line creates the 'main' function for our test program!
2 #define CATCH_CONFIG_MAIN
3 #include "catch.hpp" // The Catch2 framework
4
5 #include "fibonacci.hpp" // The code we want to test
6
7 // TEST_CASE creates a new test.
8 TEST_CASE("Fibonacci numbers are computed", "[fibonacci]") {
9
10    // REQUIRE is the "assertion".
11    // If this check is 'false', Catch2 marks the test as "FAILED"
12    // and prints a detailed report.
13
14    REQUIRE( fibonacci(0) == 0 );
15    REQUIRE( fibonacci(1) == 1 );
16    REQUIRE( fibonacci(5) == 5 );
17    REQUIRE( fibonacci(10) == 55 );
18
19    // We could add more sections
20    SECTION("testing larger numbers") {
21        REQUIRE( fibonacci(20) == 6765 );
22    }
23}
```

This is much cleaner! The `REQUIRE` macro handles all the "if-then-else-print-error-and-return-1" logic for you.

4 Timing Your Code: The Science of Speed

A common trap for new programmers is "premature optimization." They *guess* what part of their code is slow and spend hours trying to optimize it, often making the code more complex and harder to read. This is almost always a waste of time.

*"Discussions about efficiency are meaningless
in the absence of measurements!!!"*

- **Analogy:** You don't "guess" you have a fever. You *measure* it with a thermometer. You don't "guess" your code is slow. You *measure* it with a "profiler" or a "timer." The results will almost always surprise you. The "bottleneck" is rarely where you think it is.

To measure time in C++, you should *not* use the old `<ctime>` library. You should *always* use the modern, high-precision library: `<chrono>`.

4.1 How to Time Code with `<chrono>`

The `<chrono>` library gives you access to different "clocks." The one you want for performance measurement is `std::chrono::high_resolution_clock`.

The pattern for timing a piece of code is simple:

1. Get the time *before* the code.
2. Run the code.
3. Get the time *after* the code.
4. Calculate the difference.

```
1 #include <iostream>
2 #include <chrono>
3
4 // A function that just wastes some time
5 void long_running_function() {
6     for (volatile int i = 0; i < 100000000; ++i);
7 }
8
9 int main() {
10     // 1. Get the time *before*
11     auto start = std::chrono::high_resolution_clock::now();
12
13     // 2. Run the code you want to time
14     long_running_function();
15
16     // 3. Get the time *after*
17     auto end = std::chrono::high_resolution_clock::now();
18
19     // 4. Calculate the difference
20     // We ask for the duration as a <double> in units of (seconds).
```

```
21     std::chrono::duration<double> elapsed = end - start;
22
23     std::cout << "The code took: "
24         << elapsed.count() << " seconds." << std::endl;
25 }
```

In this code, `auto` is a C++ keyword that automatically figures out the (very complex) type of the `start` and `end` time-points. The `elapsed.count()` function returns the final duration as a simple `double`.

The Most Important Advice **Never trust a single measurement.** Your computer is a chaotic place. The operating system is constantly running other tasks in the background (checking for emails, updating files, etc.). These "other activities" can interfere with your measurement, making your code seem slower than it is.

- **Analogy:** You don't just take your temperature once. You take it a few times to make sure you get a stable reading.

To get a reliable result, you *must* run your timing test many times (e.g., 100 times in a loop) and then calculate the **average** or (often better) the **minimum** time. The minimum is often preferred because it represents the "purest" run, the one with the least interference from the operating system.

5 Monte Carlo Methods: Solving Problems with Randomness

We now move to a powerful class of algorithms that are essential to modern simulation: **Monte Carlo Methods**.

These are algorithms that use **random sampling** to get numerical results. They are named after the famous casino in Monaco, as they are based on the same principles of chance and randomness as games like roulette or dice.

The Core Analogy: The Area of a Lake Imagine you need to find the area of a large, complexly-shaped lake.

- **The "Calculus" Method:** You could try to find a mathematical function $f(x)$ that describes the lake's shore, and then integrate it. This would be incredibly difficult, maybe impossible.
- **The "Monte Carlo" Method:**
 1. Build a perfectly square fence (e.g., 1km \times 1km) around the entire lake. You know the area of this square: 1 km^2 .
 2. Fly over the square in a helicopter and drop 1,000,000 random "markers" (e.g., paintballs).
 3. Wait for the markers to land, then count how many landed *in the lake*.
 4. Let's say you find that 400,000 markers (i.e., 40%) landed in the lake.
 5. You can now estimate the lake's area: it must be 40% of the square's area.

$$\text{Area}_{\text{lake}} \approx 0.40 \times \text{Area}_{\text{square}} = 0.40 \times 1 \text{ km}^2 = 0.4 \text{ km}^2$$

This technique uses *randomness* to solve a completely *deterministic* problem. This simple idea is used everywhere: physics, finance, weather forecasting, and, as we'll see, integration. But it all hinges on one thing: a good source of **random numbers**.

5.1 The Problem: Multidimensional Integration

In calculus, "integration" is just finding the area under a curve. For a 1-dimensional function $f(x)$, we can use traditional methods like the **Simpson Method**.

- **Analogy (Traditional):** These methods work by laying down a *fixed, regular grid* of "tiles" (like tiny rectangles) under the curve and summing their areas.

In 1D, these methods are *excellent*. Their error (how "wrong" the answer is) decreases as $O(N^{-4})$, where N is the number of tiles. If you double the tiles, your error gets $2^4 = 16$ times smaller!

The Curse of Dimensionality The problem comes when we move to higher dimensions (d). What if we want to integrate a function in a 10-dimensional "hyper-volume"?

- **Analogy (The Curse):** Try to tile a 1-dimensional line. Easy. Now try to tile a 2D floor. You need N^2 tiles. Now a 3D room? N^3 tiles. A 10D hyper-room? You need N^{10} tiles. The number of tiles you need **explodes exponentially**. This is the "Curse of Dimensionality."

The error for the Simpson method gets exponentially worse with dimension: $O(N^{-4/d})$.

- For $d = 1$ (1D): Error is $O(N^{-4})$. (Amazing!)
- For $d = 8$ (8D): Error is $O(N^{-4/8}) = O(N^{-0.5})$. (Okay.)
- For $d = 10$ (10D): Error is $O(N^{-4/10}) = O(N^{-0.4})$. (Terrible!)

5.2 The Solution: Monte Carlo Integration

Monte Carlo integration saves us from this curse. It's the "lake" analogy applied to a function.

- **Traditional (Top Image):** Uses a fixed, regular grid.
- **Monte Carlo (Bottom Image):** Instead of a grid, it picks N **random points** x_i . It calculates the *average value* of the function at these random points, and multiplies by the width of the interval.

The Punchline (Error Scaling) The error for Monte Carlo integration is a "statistical" error. It scales as:

$$\text{Error} = O(N^{-1/2}) = O\left(\frac{1}{\sqrt{N}}\right)$$

Crucially, this error rate is **independent of the number of dimensions d !**

Let's compare again for a 10-dimensional integral:

- **Simpson's Error:** $O(N^{-0.4})$
- **Monte Carlo Error:** $O(N^{-0.5})$

The Monte Carlo error is *better* (it decreases faster). As the slide notes, Monte Carlo beats grid-based methods for any $d > 8$. In scientific simulations, d can be thousands or millions, so Monte Carlo is the *only* method that works.

5.3 Example: Calculating π with Monte Carlo

This is the "Hello, World!" of Monte Carlo methods. It's the lake analogy, but with a circle.

1. Imagine a 1x1 square (Total Area = 1).
2. Inside it, draw a quarter-circle of radius 1 (Area = $\pi \cdot r^2 / 4 = \pi/4$).
3. The *ratio* of the circle's area to the square's area is $(\pi/4)/1 = \pi/4$.

The Algorithm:

1. Generate two random numbers, x and y , both between 0 and 1. This is your "random marker" at coordinate (x, y) .
2. Check if the marker is *inside* the circle. The equation for a circle is $x^2 + y^2 = r^2$. Since $r = 1$, we just check if: $x^2 + y^2 \leq 1$.
3. Repeat this N times (e.g., 1,000,000 times). Count how many markers landed *inside* (N_{inside}).
4. The ratio of "hits" to "total" is an estimate for the ratio of the areas:

$$\frac{N_{\text{inside}}}{N} \approx \frac{\text{Area}_{\text{circle}}}{\text{Area}_{\text{square}}} = \frac{\pi}{4}$$

5. Therefore, we can estimate π :

$$\pi \approx 4 \times \frac{N_{\text{inside}}}{N}$$

This entire method depends on a good source of random x and y numbers.

5.4 What is a "Random" Number?

We need to understand what we mean by "random."

1. **"True" Random Numbers:** These come from unpredictable *physical processes*—atmospheric noise, radioactive decay, thermal noise in a circuit.
 - **Analogy:** Rolling a *real*, physical, perfectly balanced die.
 - **Pros:** Truly unpredictable.
 - **Cons:** Slow to generate, requires special hardware.
2. **"Pseudo-Random" Numbers (PRNG):** This is what 99.9% of computing uses. They are generated by a **deterministic algorithm**.
 - **Analogy:** A giant, 10-million-page book full of pre-recorded die rolls. The "algorithm" is just reading the next number from the book.
 - They are **not random at all!** They are 100% deterministic.
 - If you give the algorithm the same starting value (the "seed"), it will produce the **exact same sequence** of numbers, every single time.

This determinism is actually a *feature*, not a bug. It means our simulations are **reproducible**. A scientist who finds a bug can re-run the simulation with the *exact same* "random" numbers to debug it. The goal is just to make the algorithm's output *look* random and pass statistical tests for randomness.

5.5 How PRNGs Work: The LCG

A simple (and often flawed) example of a PRNG is the **Linear Congruential Generator (LCG)**. It uses a simple formula to get the *next* number (x_{n+1}) from the *current* one (x_n):

$$x_{n+1} = (a \cdot x_n + c) \pmod{m}$$

- x_0 is the "seed" (the starting page in the book).
- a is the "multiplier".
- c is the "increment".
- m is the "modulus" (this "wraps around" the number, keeping it from growing forever).
- **Analogy:** This is a "scrambler" machine. You put in a number x_n . It multiplies, adds, and then "wraps it around" (\pmod{m}) to get a new, "scrambled" number x_{n+1} .

All PRNGs eventually repeat. The "period" (how long the sequence is before it repeats) is at most m . A good generator has an *enormous* m . The plots in the slides (23-28) show how this deterministic sequence *appears* to be random.

5.6 The Modern C++ `<random>` Library

You *must not* write your own LCG. It's very easy to get wrong. Instead, you *must* use the professional, high-quality tools provided in the C++11 `<random>` header.

This library has a very important two-part design.

- **Analogy:** Think of generating random numbers as an "oil-refining" process.
 1. **Engines (Generators):** This is the "crude oil pump." Its only job is to pump out raw, unrefined, uniformly-distributed *unsigned integers* (e.g., big numbers between 0 and 4,294,967,295).
 2. **Distribution Functions:** This is the "oil refinery." It takes the "crude oil" from the engine and *transforms* it into the "refined fuel" you actually want (e.g., gasoline, diesel, jet fuel).

You *always* need both parts: an **Engine** and a **Distribution**.

5.7 Step 1: Choose Your Engine

C++ provides several "crude oil pumps."

- `std::linear_congruential_engine`: The LCG we just saw. (A "hand-cranked, leaky pump").

- `std::mersenne_twister_engine`: A much more powerful, high-quality generator with a *massive* period. **This is the one you should use.**
- `std::mt19937`: This is the standard, pre-packaged 32-bit Mersenne Twister. It's the "industrial-grade" pump. Use this one.

All engines share the same "buttons":

- `std::mt19937 my_engine;`: Creates the engine.
- `my_engine.seed(42);`: Sets the "seed" (the starting point).
- `my_engine()`: "Turns the crank" to get the *next* raw integer.

5.8 Step 2: Seed Your Engine

This is a *critical* step. If you don't seed your engine, it will use the *same default seed* every time, giving you the *same "random" sequence* every time you run your program.

```
1 #include <random>
2 std::mt19937 my_engine; // Create the engine
3 my_engine.seed(42); // Seed it
```

- **Analogy:** The seed is the *starting page number* in the 10-million-page book of die rolls. `seed(42)` tells the machine to "start reading from page 42."

5.9 Step 3: Choose Your Distribution (The "Refinery")

Now you choose the "refinery" to transform the engine's raw integers into the numbers you *actually* want.

- `std::uniform_int_distribution<int> die_roll(1, 6);`
 - **The "Die Roller" refinery.** Takes crude oil and gives you an integer, with equal probability, from 1 to 6.
- `std::uniform_real_distribution<double> unit_dist(0.0, 1.0);`
 - **The "0-to-1" refinery.** Takes crude oil and gives you a double (a decimal), with equal probability, between 0.0 and 1.0. **This is what you need for the π example.**
- `std::normal_distribution<double> bell_curve(100.0, 15.0);`
 - **The "Bell Curve" refinery.** Takes crude oil and gives you a double centered around a *mean* value (e.g., 100.0) with a *standard deviation* or "spread" (e.g., 15.0). This is vital for modeling natural phenomena like measurement errors or human heights.

5.10 Step 4: Put It All Together (The π Example)

Here is the full, correct, modern C++ code to generate the random (x, y) pairs for our π calculation.

```
1 #include <iostream>
2 #include <random>
3
4 int main() {
5     // === 1. SETUP (Do this ONCE at the start) ===
6
7     // 1a. Create and seed the ENGINE (the "crude oil pump")
8     std::mt19937 my_engine;
9     my_engine.seed(42); // Use a fixed seed for reproducibility
10
11    // 1b. Create the DISTRIBUTION (the "0-to-1 refinery")
12    std::uniform_real_distribution<double> my_dist(0.0, 1.0);
13
14    // === 2. USAGE (Do this inside your loop) ===
15
16    // To get a number, you pass the ENGINE *to* the DISTRIBUTION
17    // This is the "crank" that runs the whole machine.
18    double x = my_dist(my_engine);
19    double y = my_dist(my_engine);
20
21    std::cout << "Random pair: (" << x << ", " << y << ")" << std::endl;
22
23    // ... then check if (x*x + y*y <= 1.0) ...
24}
25 }
```

5.11 Best Practices for Random Numbers

This is a summary of the most important rules.

1. **Rule 1: Create ONE engine, and seed it ONCE.** A *very* common mistake is to create a new `std::mt19937` engine *inside* the loop.
 - **Analogy:** Don't build a brand-new, multi-million dollar oil pump every time you need one drop of oil. Build *one* pump at the start, and just keep *cranking* it.
2. **Rule 2: PRINT YOUR SEED.** If your simulation uses a seed from the user or from `std::random_device` (a "true random" source), *print it to your log file!*
 - **Analogy:** If your simulation discovers a "miracle" (a bug or a Nobel-prize-winning result), but you used a "secret" seed and didn't write it down, **you can never reproduce it.** It is lost forever. Always log your seed: "Simulation running with seed: 8675309".
3. **Rule 3: Try several generators.** If your scientific conclusion *changes* when you swap `std::mt19937` for `std::ranlux48`, you have a serious problem. Your result should not be an artifact of the specific PRNG you chose.

6 Documentation: Writing Code for Humans

The final topic is documentation. It's often overlooked, but it's what separates a "disposable script" from "professional, reusable software."

You are not just writing code for the compiler. You are writing it for other people. And the most important "other person" you're writing for is **yourself, six months from now**, when you've forgotten everything about this code.

6.1 The "Contract" for a Function

Good documentation for a function acts as a "contract" between the function (the *callee*) and the user (the *caller*). It should answer these questions:

- **Synopsis:** The function's signature. (What is its name and what are its parameters?)
- **Semantics:** What does it *do*? (A plain-English summary.)
- **Requirements:** (For templates) What are the rules for the template types?
- **Preconditions:** What *must* be true *before* I am called? (This is the caller's part of the contract. e.g., "The pointer must not be null," "bins must be > 0").
- **Postconditions:** What do I *promise* will be true *after* I finish? (This is the function's part of the contract. e.g., "The vector will be sorted.")
- **Exception guarantees:** What "alarms" (exceptions) might I *throw* if you (the caller) break your preconditions?
- **References:** Any links to papers or web pages that explain the algorithm.

6.2 Doxygen: The Automatic Documentation Generator

You don't just write this documentation in a random text file. You write it as *special comments directly in your source code*.

A tool called **Doxygen** can then parse your entire project.

- **Analogy:** Doxygen is a "robot librarian." You put special "comment cards" in your code, right above your functions. When you run Doxygen, the robot scans your *entire* codebase, collects all these cards, and automatically builds a beautiful, hyperlinked, searchable **HTML website** (the "card catalog") that documents your entire project.

This is how *all* professional C++ libraries (like Boost, Eigen, etc.) create their online API documentation.

6.3 Doxygen Syntax Example

Doxygen comments start with `/**` or `///`. Its commands start with an `@` or symbol.

Here is the `integrate` function from a previous lecture, fully documented for Doxygen:

```
1 /**
2 * @brief Computes the integral of a 1D function using Simpson's rule.
3 *
4 * @details
5 * This function approximates the integral of the given callable 'func'
6 * ,
7 * from 'a' to 'b' using the composite Simpson rule with 'bins'
8 * intervals.
9 *
10 * @tparam F The type of the callable object (e.g., a function, a
11 * lambda).
12 * Must be callable with a 'T' and return a 'T'.
13 * @tparam T The arithmetic type (e.g., double, float).
14 *
15 * @param a The lower integration limit.
16 * @param b The upper integration limit.
17 * @param bins The number of subintervals (must be > 0).
18 * @param func The callable object representing the function f(x).
19 *
20 * @pre
21 * The function 'func' must be valid on the interval [min(a,b), max(a,
22 * b)].
23 * @pre
24 * 'bins' must be greater than 0.
25 *
26 * @post
27 * The return value will approximate the integral.
28 *
29 * @return An approximation of the integral of func(x) from a to b.
30 *
31 * @throws Nothing. (This function is no-throw)
32 *
33 * @see
34 * Standard composite Simpson rule, e.g., Numerical CSE course notes.
35 */
```

```
32 template <typename F, typename T>
33 T integrate(const T a, const T b, const unsigned bins, const F& func)
34 {
35     // ... implementation goes here ...
36 }
```

When you run Doxygen, it will parse this comment and build a beautiful web-page for your `integrate` function.

6.4 Integrating with CMake

Your `CMakeLists.txt` file can be configured to find the Doxygen program and add a new "target." This means you can simply type `make doc`, and CMake will automatically run Doxygen to build or update your documentation website.

Doxygen can also use **Markdown** files (like `mainpage.md`) to create the "homepage" for your documentation, where you can explain what the project is, how to compile it, and how to run it.

6.5 A Practical Guide to Using Doxygen

As we've discussed, Doxygen is a "robot librarian" that builds a professional website from comments in your code. Let's walk through the exact, practical steps to make this work, from writing the comments to automating the process.

Where to Write Doxygen Comments The short answer: **Write your documentation in your header files (`.hpp` or `.h`)**, directly above the code you are documenting.

- **Analogy:** Your header file (`.hpp`) is the **public menu** for your restaurant. Your source file (`.cpp`) is the **private kitchen**.
- Customers (other programmers) should only ever look at the menu. They don't need to see the messy details in the kitchen.
- By documenting the header, you are writing the "contract" or "menu description" for your class, telling users *what it does*, not *how it does it*.

How to Write Doxygen Comments: An Example Let's fully document the `Timer` class we designed earlier. Doxygen comments start with `/**` or `///`. Here is what your `Timer.hpp` file should look like:

```
1 /**
2  * @file Timer.hpp
3  * @brief Declares a high-resolution timer class for performance
4  * monitoring.
5  * @author (Your Name)
6  * @date 2023-10-27
7  */
8 #pragma once
9
10 #include <chrono>
11 #include <stdexcept> // For std::runtime_error
12
13 // Define aliases to make the types clean
14 using Clock = std::chrono::high_resolution_clock;
15 using TimePoint = std::chrono::time_point<Clock>;
16
17 /**
18 * @class Timer
19 * @brief A simple stopwatch-style timer.
20 *
21 * @details This class models a stopwatch. It starts in a "stopped"
22 * state. You must call start() before you can call end().
23 * This is a practical example of a "state machine" (it has
24 * two states: "running" and "stopped").
```

```

25  */
26 class Timer {
27 public:
28     /**
29      * @brief Default constructor.
30      * Initializes the timer to a valid "stopped" state.
31      */
32     Timer() : m_startTime(), m_isRunning(false) {}
33
34     /**
35      * @brief Starts the timer.
36      * Captures the current time and sets the state to "running".
37      */
38     void start() {
39         m_startTime = Clock::now();
40         m_isRunning = true;
41     }
42
43     /**
44      * @brief Stops the timer and returns the elapsed time.
45      *
46      * @return The elapsed time in seconds as a 'double'.
47      *
48      * @throws std::runtime_error If the timer was not in the
49      * "running" state (i.e., if start() was not called).
50      */
51     double end() {
52         if (!m_isRunning) {
53             throw std::runtime_error("Timer::end() called before Timer"
54             "::start()");
55         }
56
57         TimePoint endTime = Clock::now();
58         std::chrono::duration<double> diff = endTime - m_startTime;
59         m_isRunning = false; // Return to "stopped" state
60         return diff.count();
61     }
62 private:
63     TimePoint m_startTime; // < The timepoint when start() was called
64     bool m_isRunning; // < Flag to track the timer's state.
65 };

```

How to Generate the Documentation (Manually) Doxygen is a command-line tool. You can run it manually to see the results.

1. **Generate a config file:** Open your terminal in your project's main directory and run:

```

1 doxygen -g Doxyfile
2

```

This creates a (very large) configuration file named `Doxyfile`.

2. **Edit the Doxyfile:** Open this text file. You only need to change a few key settings. Find these lines and edit them:

- PROJECT_NAME = "My Scientific Simulation"
- OUTPUT_DIRECTORY = ./doc (This is where the HTML site will go)
- INPUT = . (Tell Doxygen to scan the current directory)
- RECURSIVE = YES (Tell Doxygen to look inside all subfolders)
- GENERATE_HTML = YES (We want a website)
- EXTRACT_PRIVATE = NO (We only want to document the public API)

3. **Run Doxygen:** Now, just run the doxygen command in that same directory:

```
1 doxygen Doxyfile
2
```

You will now have a new folder named doc. Open doc/html/index.html in your web browser. You will see a beautiful, searchable website documenting your Timer class!

How to Integrate with CMake (The Automatic Way) Running Doxygen by hand is annoying. We want to just type make doc. This is a perfect job for CMake (as mentioned on Slide 42).

Add the following code to your CMakeLists.txt file:

```
1 # --- Doxygen Integration ---
2 # Find the Doxygen program on your system
3 find_package(Doxygen REQUIRED)
4
5 # This assumes you have a "Doxyfile" in your project's root folder
6 # (the one containing this CMakeLists.txt)
7 if(DOXYGEN_FOUND)
8     # Add a new "target" (a new command) to your Makefile.
9     # This command will not be run by default.
10    add_custom_target(doc
11        # The command to run:
12        COMMAND ${DOXYGEN_EXECUTABLE} Doxyfile
13
14        # Where to run the command from:
15        WORKING_DIRECTORY ${CMAKE_SOURCE_DIR}
16
17        # A nice message to print:
18        COMMENT "Generating API documentation with Doxygen"
19
20        # This makes 'doc' a "phony" target, not a real file
21        VERBATIM
22    )
23
24    # Optional: Print a helpful message
```

```
25     message(STATUS "Doxygen found - 'make doc' target is available.")
26 endif()
27 # --- End of Doxygen Integration ---
```

Now, after you run `cmake` and `make` to build your project, you can simply run:

```
1 wslview html/index.html
```

This will open the index of your documentation in an `html` viewer such as Chrome.

Adding a Main Page (Slide 43) Your documentation needs a "homepage." The easiest way is to use a Markdown file.

1. **Create a file** in your project's root directory named `mainpage.md`.
2. **Write your homepage content** in this file:

```
1 # Welcome to My Scientific Simulation!
2
3 This is the main page for the documentation of our project.
4
5 This project simulates... (your description here).
6
7 ## How to Compile
8 To compile the project, run the following commands:
9
10 mkdir build
11 cd build
12 cmake ..
13 make
14
15 ## How to Run
16 The main executable is simulation:
17
18 ./simulation
```

3. **Tell Doxygen to use it:** Open your `Doxyfile` and edit this line:

```
1 # This tells Doxygen to use your .md file as the index.html page
2 USE_MDFILE_AS_MAINPAGE = mainpage.md
3
```

Now, when you run `make doc`, Doxygen will generate the same website as before, but the homepage will be your beautifully formatted `mainpage.md` file. This provides a professional entry-point for anyone using your code.

7 Conclusion

This chapter has covered five essential skills for writing professional scientific software. We learned how to:

- **Handle Errors** gracefully using the C++ `try/throw/catch` exception system, which is far more robust than old C-style error codes.
- **Gain Confidence** in our code's correctness by writing `unit tests` and using frameworks like CTest and Catch2 to automate them.
- **Measure Performance** accurately using the `<chrono>` library, allowing us to find and fix real bottlenecks instead of guessing.
- **Use Randomness** safely and effectively using the C++ `<random>` library's Engine/Distribution model, which is the key to Monte Carlo methods.
- **Write for Humans** by documenting our code's "contract" using tools like Doxygen, ensuring our code is readable and maintainable.

These techniques form the foundation upon which all complex, reliable, and efficient scientific simulations are built.