COMPUTATIONAL FINANCE & RISK MANAGEMENT

UNIVERSITY *of* WASHINGTON

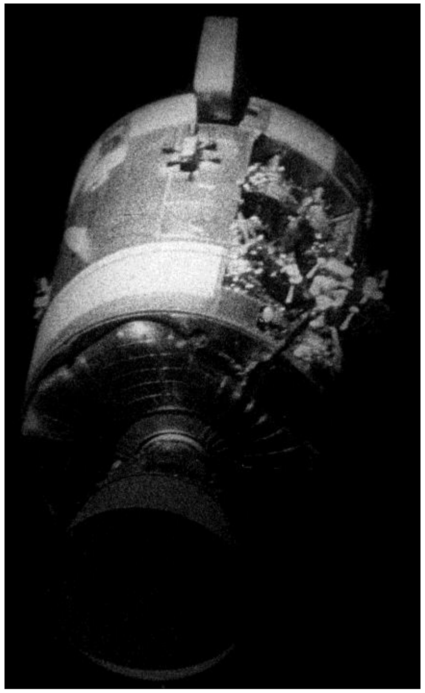
Department of Applied Mathematics

Thoughts on Teaching C++ to Students in the Applied Sciences

NWCPP Meeting, 15 April 2020

Dedication

To the crew of Apollo 13, and all the members of Mission Control who worked tirelessly to get the men home safely, 50 years ago this month.



Background

- Single quarter introductory/intermediate course (10 weeks) in C++ for MSc students in Computational Finance & Risk Management (CFRM) at the University of Washington
- Emphasis on applications
- ~90% of this material would be transferable to general applied sciences
- Orders of magnitude more efficient than Python
- But can be almost as rapid to develop, thanks to ***modern*** C++
- Currently in sixth year of teaching this course

Background

- Companion lecture notes: *C++ for the Rest of Us!*

Background

- Introductory:
  - ➢ Fundamentals of C++
  - ➢ Object-Oriented Programming
  - ➢ Templates
  - ➢ STL Containers and Algorithms
- Intermediate:
  - ➢ Prior programming experience in other languages is assumed
    - Functions
    - Conditional branching
    - Iteration
    - We move very quickly through these topics
  - ➢ Open source mathematical libraries
  - ➢ By the end of the quarter, students are able to implement some fairly sophisticated mathematical routines and models

Goals

- Proficiency in implementing common mathematical models in C++
- Emphasis on modern (through C++17) language and Standard Library features for problem solving
    - “No-cost” abstractions
    - Don’t reinvent what we already have
    - Don’t discourage students with gratuitously complicated code or legacy C constructs
    - Use modern C++ for practical work!
    - Bust the myth that C++ is “too difficult”

We Don’t Care About…

- C language programming
  - Not a prerequisite
  - **`std::string`**, not **`char*`**
  - **`std::vector<.>`**, not dynamic C-arrays
- Minute details about strings and output formatting
  - Computational course, so we care more about numerical results
  - In practice, results are not output to the console with **`std::cout`**
- The vast multitude of numerical types in C++; we only require the following for our work
  - **`double`**
  - **`int`**
  - **`size_t`** and **`long`** where necessary

We Don't Care About…

- Memory allocation with **`new`** and **`delete`**
  - We use **`std::unique_ptr<.>`** instead
  - **`new`** and **`delete`** are covered during the last week
- Implementing sort/search algorithms or doubly-linked lists
  - They're in the Standard, so use them
  - We use **`std::vector<.>`** anyway

Week 1

- A brief history of C++
    - “The Creator” Bjarne Stroustrup
    - C with Classes
    - Turn of the century heyday
    - Stagnation: C++98/C++03 and the emergence of Java
    - The Beast is Back (Jon Kalb): C++11 and the post-2011 world
- IDE project setup
    - Visual Studio 2019 is preferred
    - How to build a simple executable
    - Compiler warnings and errors

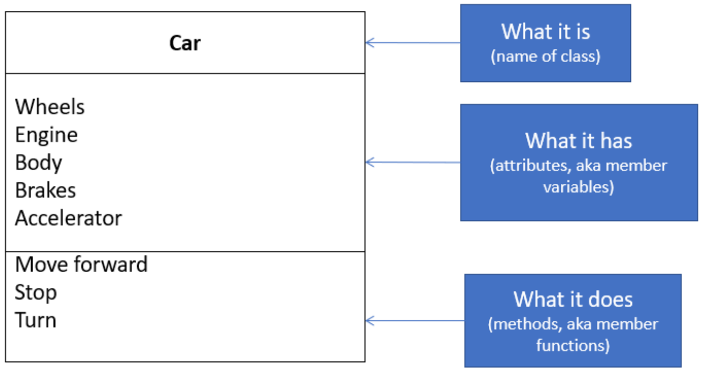
Week 1 (continued)

- Math
  - Numerical variables: **`int`** and **`double`**
  - Math operators
  - Mathematical functions in **`<cmath>`**
- User-defined functions
  - Declaration and implementation
  - Write own functions using **`<cmath>`**

Week 2

- Classes and objects
  - Car is a class, a Dino Ferrari 308 GT4 and a Honda Accord are objects

  - Member variables and member functions
  - UML class diagrams


Car
Wheels
Engine
Body
Brakes
Accelerator
Move forward
Stop
Turn
What it is (name of class)
What it has (attributes, aka member variables)
What it does (methods, aka member functions)

Week 2

- Classes and objects (continued)
  - **`std::string`** as an example
    - ➢ What it means to create an instance
    - ➢ Call and use the member functions
      - **`at(.)`**
      - **`size(.)`**
      - **`push_back(.)`**
      - These also then become more familiar for when we cover STL containers
- Conditional and iterative statements
  - **`if/else if/else`**
  - **`switch`** statements
  - **`for`**/**`while`** loops

Week 2

- Arrays
  - **`std::array`** and **`std::vector`**
  - We don't care about dynamic C-style arrays
- Function overloading
- Aliases
  - **`using`**
  - prefer **`using`** over **`typedef`**
  - (lvalue) references
- Pointers
  - To variables on the stack
  - Only introduced here because we will soon discuss
    - ➢ The **`this`** pointer
    - ➢ STL iterators
    - ➢ "Just-in-Time" approach to introducing new concepts

Week 3: Object-Based Programming and user-defined classes

- Modified version of the Fraction example in Josuttis – C++ Object Oriented Programming: Incrementally build up a class with
  - Private numerator and denominator members, and public accessors and mutators for each
  - Overloaded constructors
  - Meaning of the **`this`** pointer
  - Refactor error checking for zero in the denominator
    - ➢ Constructor
    - ➢ Mutator for denominator

Week 3: Object-Based Programming and user-defined classes

- Fraction example (continued): Incrementally build up a class with
  - Operators
    - Multiplication **`*`** and **`*=`**, plus a common private simplifying function (uses **`std::gcd(.)`** in C++17)
    - Inequality **`<`**
    - Equality and non-equality **`==`** and **`!=`**
    - Prefix and postfix increment operators **`++`**
    - Returning **`*this`**
  - Functors

Week 3 Assignment

- Implement a more complete set of operators on the Fraction class
  - **`+   +=   -   -=`** (and use **`std::lcm(.)`** from C++17)
  - **`/   /=`**
  - **`<=   >   >=`**
  - Prefix and postfix decrement
  - Other member functions

Week 4: Object-Oriented Programming

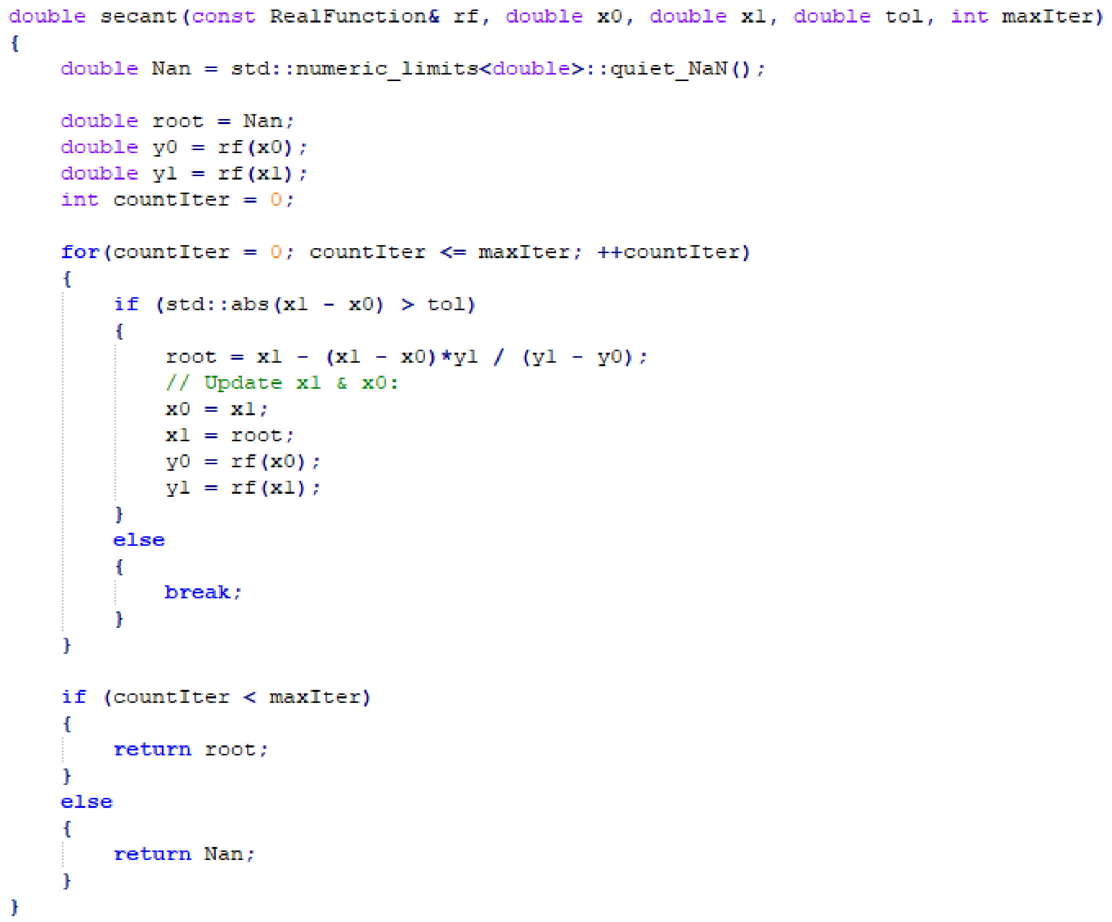
- **Inheritance**
    - Virtual functions
    - Virtual default destructor on base class
    - **`virtual`** and **`override`** keywords
    - Order of instantiation and destruction
    - Emphasis on abstract base interface classes
        - ➢ Pure virtual functions only
        - ➢ Restrict to one level of derivation
        - ➢ Pitfalls of extended inheritance chains
    - Polymorphism

Week 4: Object-Oriented Programming

- Assignment: inheritance and polymorphism for root-finding
  - Implement a pure abstract base class representing a function $f(x)$
    - **`virtual double operator()(double x) = 0 const;`**
  - Implement several derived concrete classes; eg
    - $ax^3 + bx^2 + cx + d$
    - $a\log(x - b)$
      - Introduce and use **`std::numeric_limits<double>::epsilon()`** , **`infinity()`**, and **`quiet_NaN()`**
    - $\gamma \sin(ax) - \eta \cos(bx)$
  - Write a function implementing a root-finding method (eg Secant) that takes an abstract base object const reference as its argument and computes the roots for any derived class object

Week 4: Object-Oriented Programming

- Assignment: inheritance and polymorphism for root-finding (continued)



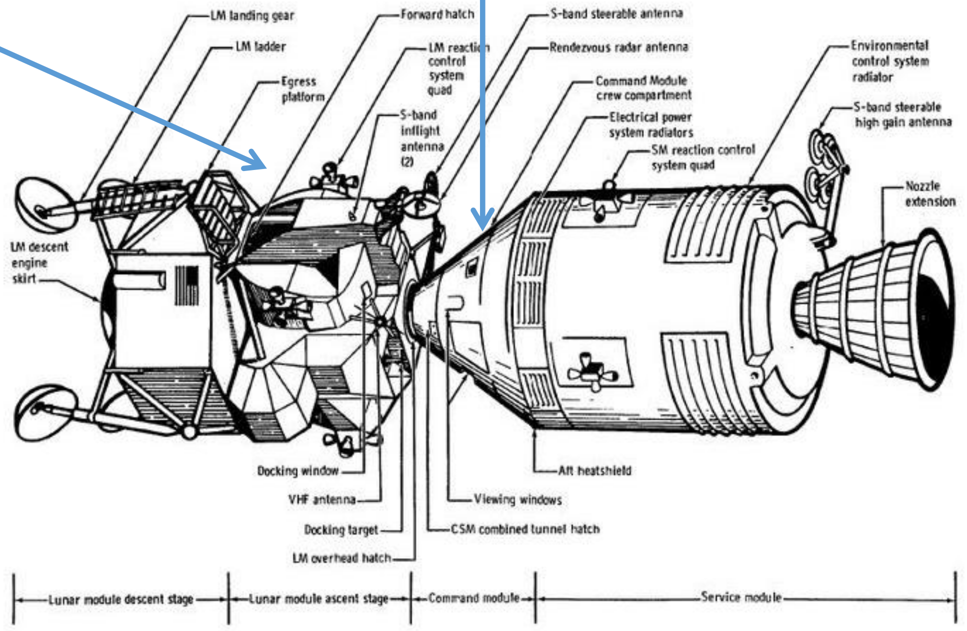
```cpp
double secant(const RealFunction& rf, double x0, double x1, double tol, int maxIter)
{
    double Nan = std::numeric_limits<double>::quiet_NaN();

    double root = Nan;
    double y0 = rf(x0);
    double y1 = rf(x1);
    int countIter = 0;

    for(countIter = 0; countIter <= maxIter; ++countIter)
    {
        if (std::abs(x1 - x0) > tol)
        {
            root = x1 - (x1 - x0)*y1 / (y1 - y0);
            // Update x1 & x0:
            x0 = x1;
            x1 = root;
            y0 = rf(x0);
            y1 = rf(x1);
        }
        else
        {
            break;
        }
    }

    if (countIter < maxIter)
    {
        return root;
    }
    else
    {
        return Nan;
    }
}
```

Week 4: Object-Oriented Programming

- **Composition** with a member object, and avoid object copy
  - Pass by and store as const *lvalue* reference
    - ➢ Avoid object copy
    - ➢ But not flexible
      - Lack control over lifetime of the member variable
      - Can't modify member object after containing object initialized
      - Or copy the containing object
  - Pass by move/*rvalue* reference and store the actual object
    - ➢ Containing object has full control over member object lifetime
    - ➢ Can be modified after containing object construction
    - ➢ Deep copy of containing object possible by default

Week 4: Composition Example

- An Apollo moon landing mission *has-a* command module and a lunar module


LM landing gear
LM ladder
Egress platform
Forward hatch
LM reaction control system quad
S-band inflight antenna (2)
S-band steerable antenna
Rendezvous radar antenna
Command Module crew compartment
Electrical power system radiators
SM reaction control system quad
Environmental control system radiator
S-band steerable high gain antenna
Nozzle extension
LM descent engine skirt
Docking window
VHF antenna
Docking target
LM overhead hatch
Aft heatshield
Viewing windows
CSM combined tunnel hatch
Lunar module descent stage
Lunar module ascent stage
Command module
Service module

- Case 1: Store CM and LM as const lvalue reference
- Case 2: Store CM and LM by value, by passing by move semantics

Week 4: Composition Examples

Header file: lvalue case

```
#include "CommandModule.h"
#include "LunarModule.h"

class ApolloRef
{
public:
    ApolloRef(const CommandModule& cm, const LunarModule& lm);
    void printModuleNames() const;

private:
    const CommandModule& commandModule_;
    const LunarModule& lunarModule_;
};
```

Week 4: Composition Examples

Header file: rvalue case

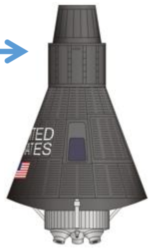
```
#include "CommandModule.h"
#include "LunarModule.h"


class Apollo
{
public:
    Apollo(CommandModule&& cm, LunarModule&& lm);
    void printModuleNames() const;

private:
    // Stored by value; Apollo object now has sole ownership
    // of these member objects.  They and their state can also
    // be modified as they are not restricted as const.
    CommandModule commandModule_;
    LunarModule lunarModule_;
};
```

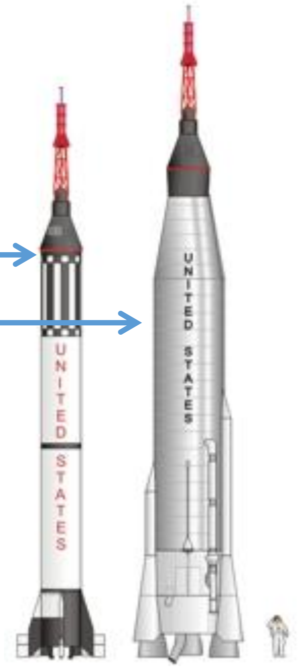
Week 4: Object-Oriented Programming

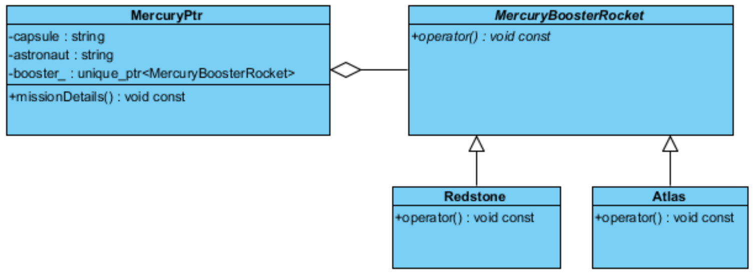
- Composition *with a polymorphic member object*
  - Pass by and store as const *lvalue* reference (but again, has limitations)
  - Pass by move/**`std::unique_ptr<.>`** and store as **`unique_ptr<.>`**
    - ➢ Full control over lifetime of member
    - ➢ Can replace old **`clone()`** methods that required object copy
    - ➢ Blessed by The Creator (*A Tour of C++*, 2E)

Week 4: Composition Examples (polymorphic member)

- A Mercury mission (first NASA manned missions with single astronaut) was comprised of
  - A capsule
  - And a booster rocket
    - ➢ Redstone, for the first up-and-down missions
    - ➢ Atlas, with more power for the later orbital missions
    - ➢ The booster rocket is therefore a polymorphic member







Week 4: Composition Examples (polymorphic member)

- Header file:

```
class Mercury
{
public:
    Mercury(std::unique_ptr<MercuryBoosterRocket> booster,
        const std::string& capsule, const std::string& astronaut);

    // Rule of Five:
    // Explicitly disable copy operations:
    Mercury(const Mercury& rhs) = delete;                  // 1
    Mercury& operator = (const Mercury& rhs) = delete;      // 2

    // Since we explicitly declared copy ctor and assignment
    // should also explicitly allow default move operations:
    Mercury(Mercury&& rhs) = default;                   // 3
    Mercury& operator = (Mercury&& rhs) = default;        // 4
    ~Mercury() = default;                               // 5


    void missionDetails() const;

private:
    std::unique_ptr<MercuryBoosterRocket> booster_;
    std::string capsule_;
    std::string astronaut_;
};
```

Week 5 – STL Containers

- A gentle introduction to templates
  - functions
  - classes
- STL containers
  - sequential
  - associative
  - **`std::vector<.>`**
    - ➢ Is our sequential container of choice
    - ➢ In-depth coverage of member functions
    - ➢ **`push_back(.)`** vs **`emplace_back(.)`** - avoid object copy
  - std::map<.> is useful for handling data input and reporting out results, with enum key types
- STL iterators and iterator-based **`for`** loops
- Range-based **`for`** loops

Week 6 – STL Algorithms

- STL Algorithms
  - Focus on **`for_each(.)`** and **`transform(.)`** to start
  - Algorithms in **`<numeric>`**
  - More from **`<algorithm>`**
  - Assignment: Use various algorithms to avoid loops (including calculations of means and dot products)
- Lambda Expressions
  - Very convenient for math!
  - As auxiliary functions in STL algorithms
- Exceptions

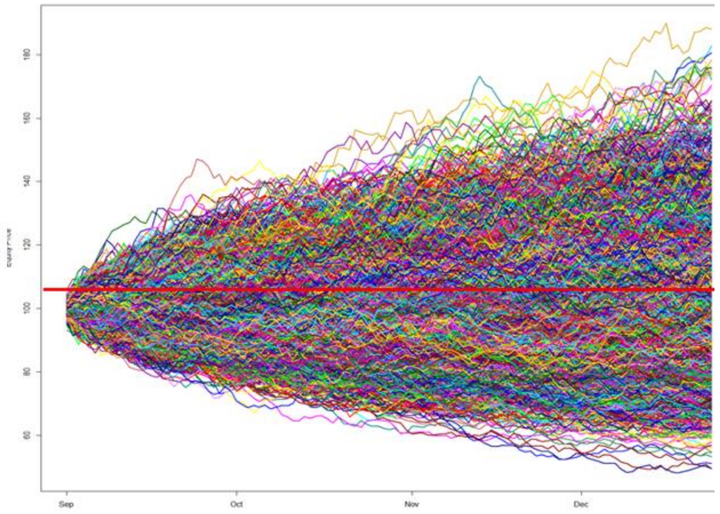
Week 7: Numerical C++

- Random number generation with **`<random>`**
  - Engines: We use Mersenne Twister 64 bit
  - Distributions: We mainly use
    - ➢ Uniform
    - ➢ Normal
    - ➢ Student's t

- Parallel STL algorithms from C++17

Week 7: Numerical C++

- Task-based concurrency with **`std::future`** and **`std::async(.)`**
  - Concurrency vs Parallelism
  - **`std::async(.)`** gets a bad rap sometimes, but
    - ➢ The performance improvement is fantastic…
    - ➢ Considering how easy and fool-proof it is to use
    - ➢ 90%+ cut in run-time on a 20-processor virtual server
    - ➢ Commonly available in modern practice

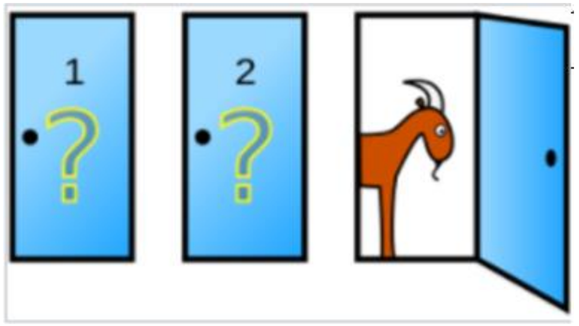


Week 7: Assignment (`<random>`)

- The Monty Hall Paradox (Let's Make a Deal)
  - Three doors marked number 1, 2, and 3
  - Behind one of the doors would be the prize, such as a new car
  - Behind the other two doors were "zonk" prizes (ie, you lose), eg a goat.

Week 7: Assignment

- The Monty Hall Paradox (continued)
  - When the contestant would make a choice, say Door #1, Monty would have one of the other doors opened eg Door #3, revealing a goat



  - He would then ask the contestant whether s/he would like to switch the choice to the remaining door
  - The "Paradox": If you were given the chance to switch, should you?
  - It turns out the answer is ***yes***!
  - The assignment is to implement a model of the outcomes in C++, using simulations of a uniform distribution on {1, 2, 3}, available in **`<random>`**

Week 8

- Eigen: Matrix Algebra Library
  - Templated/header only
  - Dynamic **`MatrixXd`** and **`VectorXd`** objects
    - ➢ All standard matrix and vector operations are supported
    - ➢ STL compliant
  - Matrix decompositions
    - ➢ LU (Lower/Upper Triangular)
    - ➢ Cholesky
    - ➢ SVD (Singular Value Decomposition)

Week 8

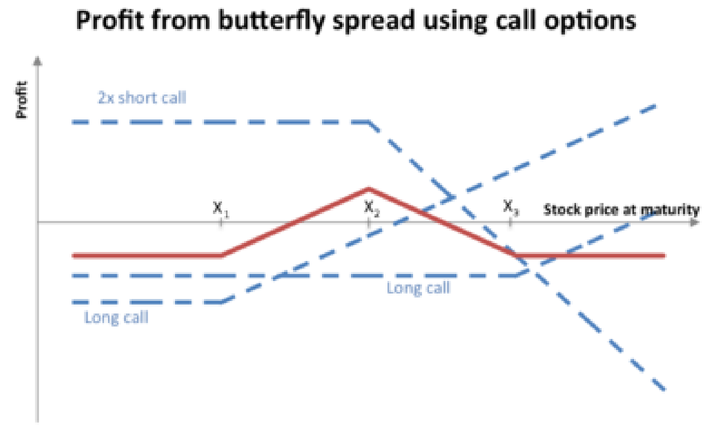
- Examples and exercises (Eigen: Matrix Algebra Library)
    - Solve system of linear equations (LU)
    - Correlated random scenarios (Cholesky + Monte Carlo)
    - Calculation of regression coefficients (SVD)
    - Rolling window predictions of portfolio returns (matrix algebra)

Week 8

- Math-related components in the Boost Libraries
  - Boost Math Toolkit
    - ➢ Probability Distributions: cdf, pdf, quantile functions
    - ➢ Numerical Integration
    - ➢ Root Solving
  - Circular Buffers
    - ➢ Useful for time series and live data feeds
    - ➢ STL compliant
  - Assignment: EGARCH(1, 1) model of market volatility
    - ➢ Circular buffer
    - ➢ **`std::normal_distribution<>`** from **`<random>`**

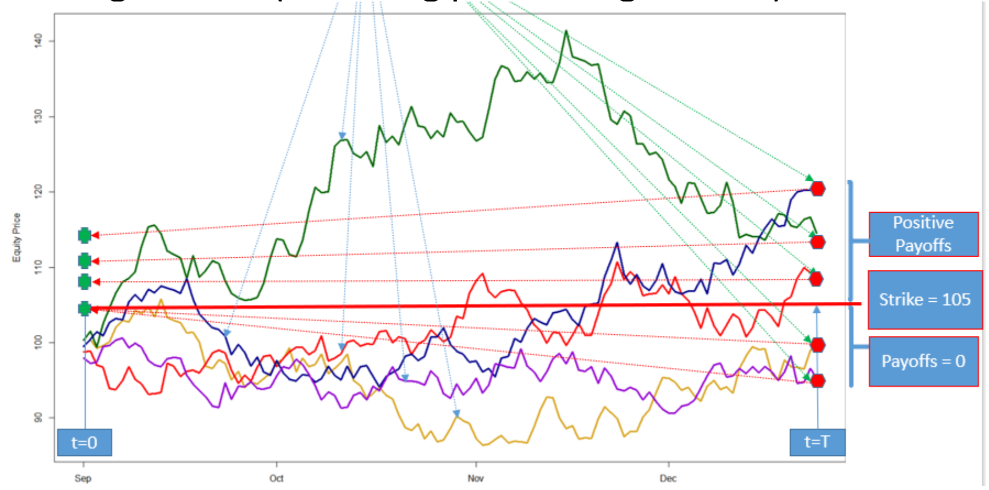
Week 9: Option Pricing

- Confluence of topics covered: Use these to code realistic examples
- Pricing of financial option contracts
  - Closed form models (Derived from Black-Scholes)
    - ➢ European options
    - ➢ Single barrier options
    - ➢ Calculate implied volatility (Boost root finding)
    - ➢ Boost Math Library: Statistical Distributions (N(0, 1) cdf)



Week 9: Option Pricing

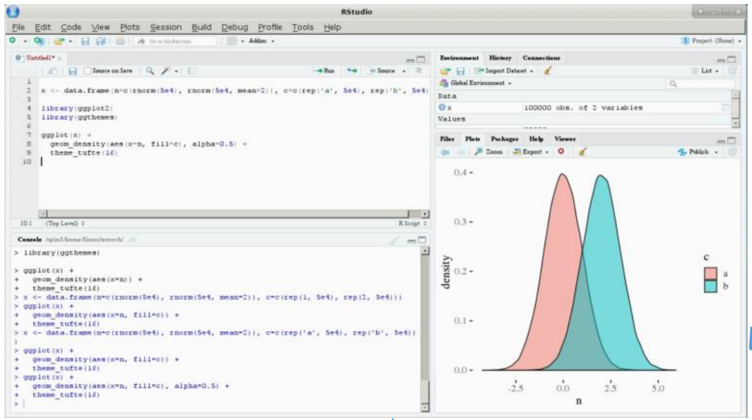
- Pricing of financial option contracts (continued)
  - Monte Carlo-based models
    - ➢ European options (compare results with closed form)
    - ➢ Single barrier options (compare results with closed form)
    - ➢ Featured concepts:
      - Projection of asset price simulations using <random>
      - Generation of these simulations in parallel using **`std::async`** and **`std::future`**
      - STL algorithms (including parallel algorithms)



Week 10

- Rcpp: An R package that provides interfaces to C++ (Dirk Eddelbuettel)
  - RcppEigen
  - BH (Boost Headers)
  - Interface to
    - ➢ One-off C++ code as a faster alternative to R
    - ➢ Reusable C++ code and libraries
  - For R users, calling a function in C++ is the same as calling an R function
  - Can use the powerful visualization capabilities available in R

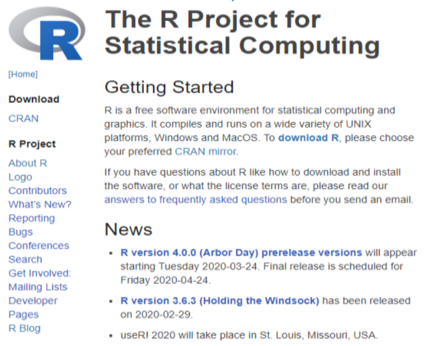
Week 10 (Rcpp Interface)


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Untitled1*
Source on Save
Run
Source
Environment History Connections
Import Dataset
Global Environment
List
Files Plots Packages Help Viewer
Zoom
Export
Publish
density
n
c
a
b

```
x <- data.frame(n=c(rnorm(5e4), rnorm(5e4, mean=2)), c=c(rep('a', 5e4), rep('b', 5e4)

library(ggplot2)
library(ggthemes)

ggplot(x) +
  geom_density(aes(x=n, fill=c), alpha=0.5) +
  theme_tufte(16)
```

```
> library(ggthemes)

> ggplot(x) +
+   geom_density(aes(x=n)) +
+   theme_tufte(16)
> x <- data.frame(n=c(rnorm(5e4), rnorm(5e4, mean=2)), c=c(rep(1, 5e4), rep(2, 5e4)))
> ggplot(x) +
+   geom_density(aes(x=n, fill=c)) +
+   theme_tufte(16)
> x <- data.frame(n=c(rnorm(5e4), rnorm(5e4, mean=2)), c=c(rep('a', 5e4), rep('b', 5e4))
)
> ggplot(x) +
+   geom_density(aes(x=n, fill=c)) +
+   theme_tufte(16)
> ggplot(x) +
+   geom_density(aes(x=n, fill=c), alpha=0.5) +
+   theme_tufte(16)
> 
```

Data
x   100000 obs. of 2 variables
Values



The R Project for Statistical Computing

[Home]

Download

CRAN

R Project

About R
Logo
Contributors
What's New?
Reporting Bugs
Conferences
Search
Get Involved: Mailing Lists
Developer Pages
R Blog

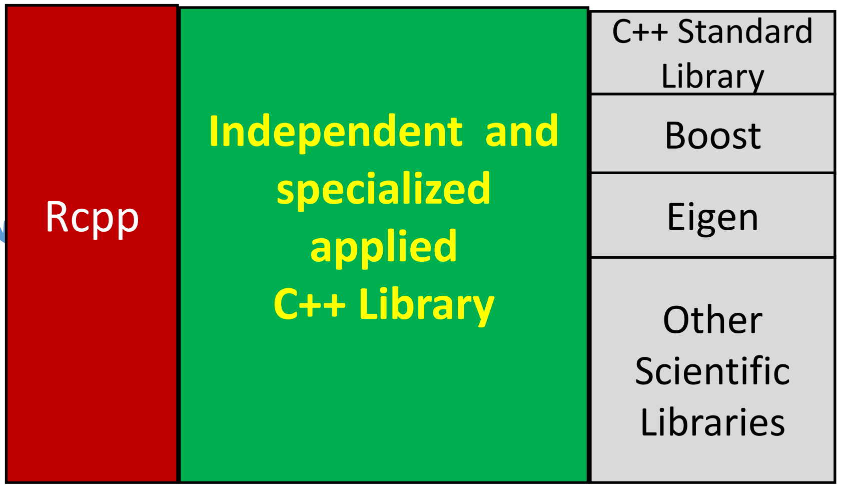
Getting Started

R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS. To **download R**, please choose your preferred CRAN mirror.

If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

News

- **R version 4.0.0 (Arbor Day) prerelease versions** will appear starting Tuesday 2020-03-24. Final release is scheduled for Friday 2020-04-24.
- **R version 3.6.3 (Holding the Windsock)** has been released on 2020-02-29.
- useR! 2020 will take place in St. Louis, Missouri, USA.


Rcpp
Independent and specialized applied C++ Library
C++ Standard Library
Boost
Eigen
Other Scientific Libraries

Summary

- At the end of the course, students can program in *modern* C++:
  - Object-oriented code
  - Efficient iterative code with STL algorithms
  - Distribution-based random number generation
  - Probability functions
  - Standard numerical methods
  - Parallelized code
  - Matrix algebra and decompositions
  - Regression models
  - Reusable library code with graphics capabilities in R
  - Real world scientific applications
    - ➢ Faster than Python
    - ➢ But not much more difficult

- With significantly better efficiency than Python
- And with convenient abstractions for rapid development
- Without ever needing:
  - **`*char`**
  - **`new`** and **`delete`**
  - Dynamic C arrays
  - Implementation of a doubly-linked list
  - Coding a binary search
  - Manual thread management or non-standard threading library
- And without the frustration of legacy constructs

Summary

Thank you very much for attending!

hansondj (at) uw.edu

Sample code will also be made available on GitHub