Analysis of Covid-19 Mathematical and Software Models

Or how NOT to set up a software project

Dr. Daniel J. Duffy, dduffy@datasim.nl

Abstract
This report discusses the open-source software that implements the Covid-19 model in [1] (announced March 16th 2020), led by Dr. Neil Ferguson of Imperial College London (ICL). My personal interest was to investigate how the model was implemented after having seen it described as a system of ordinary differential equations (ODEs) on BBC News on March 17th 2020). Anecdotal evidence suggests that the original program (written in C) is at least twenty years old and it is undocumented (nothing new in the software world; the programmers in this case probably thought it was not necessary to write readable and maintainable software). Furthermore, all the 15,000 lines of code were in a single file (sometimes called *balls of mud*). On April 22nd 2020 a modified version (called 0.7.0, seemingly produced by Microsoft) appeared consisting of approximately 12 separate source files. This is the version of the program that we review in this article. We do not investigate the fixes that have been made between April 22nd and the time of writing of this report. Finally, we note that version 0.7.0 is not an implementation of the ODEs that were announced on BBC News with great aplomb on the evening of Saint Patrick's Day 2020. For a discussion of ODEs in epidemiology, see [2]. Before I discovered that the ICL model was not ODE-based (seemingly contradicting the BBC announcement) I solved the MSEIR (iMmune, Susceptible, Exposed, Infective, Recovered) ODE model numerically in C++. The system of equations is relatively benign and we used the C++ Boost *odeint* library to solve them (A word of advice: it is tempting to use the Euler method but don't use it, not even for producing cute S-curves in your blogs).

Summarising, this report is a critique of the quality of 0.7.0 code based on my experience, background and how I view software. I was unable to analyse the underlying mathematical model because it is not documented and the code is unreadable. However, I do have something to say about the (lack of) quality of code, random number generation (RNG) and (univariate) statistical distributions as these are of central importance in computational finance, an area in which I am involved (see [3], [4], [5], [6]).

This critique is based on incomplete information, namely the bespoke open-source C code. I was unable to find further relevant documentation. Nonetheless, a robust review is still possible. Finally, this report has wider applicability than just the current software system. It pinpoints some of the things that can (and do) go badly wrong in software projects. I will distribute this report to my clients and students. I have been preaching the same "doctrine" for more than thirty years and now is a special opportunity to use it to shine a light to expose the shortcomings of such a high-profile project. It will be interesting to follow developments in the coming years.

One reason for writing this report was to counter some of the technically superficial and somewhat partisan blogs written by what we flippantly call *nameless internet warriors* (see [17]). It is never clear to me if there are ulterior or political motives behind these blogs. They are irrelevant here. On the other hand, these kinds of blogs may provide more information on some structural problems in software projects (see [18] for some horror stories).

Keywords C Language, *big balls of mud*, software crisis, software quality metrics, random number generation, parallel code (speedup, race conditions), and software maintenance, defined software process.

Disclaimer

No one has paid me for this work nor has anyone pushed me to write this report. I am not a member of any lobby or pressure group.

One reason for writing this report was to remove many half-baked and subjective random ideas floating on the internet about the COVID-19 software. Another follow-on reason is to show that the ICL code is by no means unique in the lack of attention that it pays to proper coding and taking a more professional engineering approach to software projects.

1 First Encounters with the Software; Kicking the Tires

I downloaded the code from github. Then I created an empty Console Project in Visual Studio 2017. The project did not compile because first, some of the over-used functions (such as `sscanf, fprintf`) are unsafe and are deprecated. Second, the code is using a little-known and somewhat arcane library called *Magick++* (I don't even know why it is needed in the first place; it reminds me of the early 90s when such libraries were popular.)

I was able to fix these problems but it was not a good *first impression*.

2 Analysis of the Covid-19 Software Product Version 0.7.0

This section contains the main results and findings of the report based on inspection of the undocumented code. The main goal of this section is to analyse the quality of the code as objectively as possible.

2.1 C Code Quality

The code is written in C, a general-purpose procedural programming language that was developed in 1972 [12]. C is one of the most popular languages but the syntax of the original language was "dangerous" and "ugly" (see [11]). In 1983 C++ was born and one of its goals was to create a "better C" in the sense that the new syntax would improve code readability and robustness. In this regard the ICL code has not been improved nor upgraded to C++ at all. Some of the major problems with the code are (see [12] for more on the C language):

1. Raw pointers (single and double), manual memory on the heap.
2. Much use of large STACK-based arrays (typically what novices do). For example, the following code triggers a crash in DEBUG mode while in RELEASE mode it runs OK
   ```
   int netbuf[4 * 1'000'000];
   ```
3. Use of `extern` and `static` variables.
4. Massive use of deprecated functions such as `sscanf, fprintf`. In 2020 these are not needed. File I/O used, this is ancient. Should use a more modern file format. It might be worthwhile to use a real database system.
5. char* all over the place.
6. Old school *for loops.*
7. Who would believe it, but (HARMFUL) GOTO is used, hundreds of them!
8. Using unsafe macros instead of standard C++ functions.
9. Unsafe C-style conversions.
10. Using pointers as input arguments, no true C++ programmer would ever do this.

It is not clear what the (negative) impact from these features are on the rest of the code.

It is probably safe to say that the percentage of present-day programmers who know C (or want to know C) is a small percentage of the total developer population. Rightly or wrongly, they do not wish to be associated with "legacy code".

2.2 Code Layout, Code Complexity

In general, the code layout and readability are a disaster. This is primarily caused by casual programmers with no experience of writing flexible and maintainable software as well as writing code that is understandable to others. What we see here is a great ball of mud. Professional and experienced software developers would never do this.

Where did it all go wrong? Inexperience and a possible lack of self-awareness. Some serious symptoms that add to the complexity are:

1. Large files with many lines of unfathomable code (LOC):
   Sweep 1770 LOC
   Covid 5405 LOC
   SetupModel 2547 LOC

2. Many functions consisting of hundreds of lines of code.
3. Many functions with 4-deep *for loops* and numerous *if-else* conditionals.
4. “OpenMP pepper spraying” all over the place.

These major (and possibly irreparable) problems are caused by the fact that the code was written in a bottom-up manner and that the developers do not use top-down functional decomposition techniques to break a problem into simpler components. In other words, modularity is missing. We discuss this issue in more detail in section 2.3.

The code is daunting to read and should be rewritten from the ground up. We can quantify the above criticisms by appealing to graph theory to produce a formula to compute the complexity of any piece of code. To this end, *cyclomatic complexity is a software metric used to indicate the complexity of a program. It is a quantitative measure of the number of linearly independent paths through a program's source code. It was developed by Thomas J. McCabe, Sr. in 1976.*

Cyclomatic complexity is computed using the control flow graph of the program: the nodes of the graph correspond to indivisible groups of commands of a program, and a directed edge connects two nodes if the second command might be executed immediately after the first command. Cyclomatic complexity may also be applied to individual functions, modules, methods or classes within a program.

One testing strategy, called basis path testing by McCabe who first proposed it, is to test each linearly independent path through the program; in this case, the number of test cases will equal the cyclomatic complexity of the program. (from Wikipedia).

Let us take an example. The following code is an implementation of a strongly connected control flow graph as depicted in Figure 1:

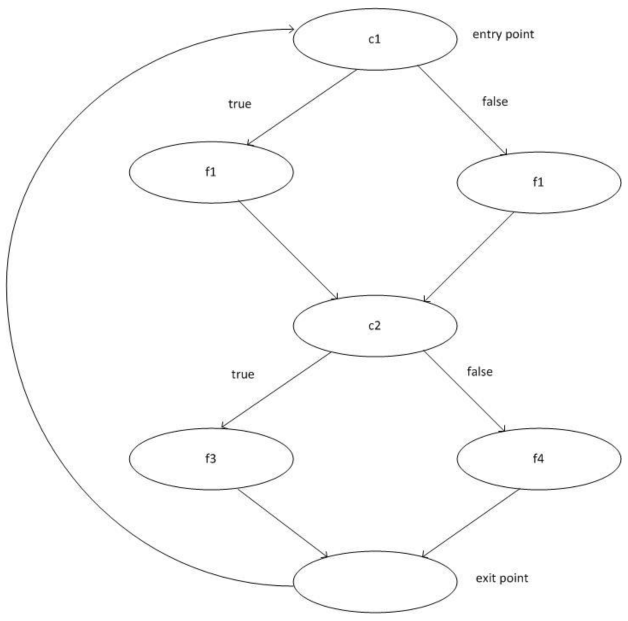

Figure 1 Control Flow Graph of CyclomaticFunction

```
#include <iostream>

// Use lambda functions, test cases
auto c1 = [](bool b) { return b; };
auto c2 = [](bool b) { return b; };
auto f1 = []() { std::cout << "f1\n"; };
auto f2 = []() { std::cout << "f2\n"; };
auto f3 = []() { std::cout << "f3\n"; };
auto f4 = []() { std::cout << "f4\n"; };

void CycloMaticFunction(bool b1, bool b2)
{
	if (c1(b1))
		f1();
	else
		f2();

	if (c2(b2))
		f3();
	else
		f4();
}

int main()
{
	bool b1 = true; bool b2 = false;
	CycloMaticFunction(b1, b2); // f1(), f4() called

}
```

In general, the complexity M is defined as:

$$M = E - N + 2P$$

where

E = number of edges in the graph

N = number of nodes in the graph

P = number of connected components.

In the case of Figure 1 we have $E = 9$, $N = 7$, $P = 1$. Hence, the complexity $M = E - N + 2P = 4$.

The relevance of cyclomatic complexity to the COVID-19 software system is that it can help us answer a number of questions:

1. It can help in determining the number of test cases to write in order to test all execution paths in a module or function.
2. It can be an incentive to split a module into smaller modules whenever the cyclomatic complexity of a modules exceeds 10.
3. Functions that have the highest complexity also tend to have the most defects.
4. There is a positive correlation between cyclomatic complexity and program size (see [21]).

Of course, you don't need to appeal to cyclomatics to realise that the COVID-19 is unduly complex. To quote Les Hatton:

I have been arguing since 1990 that Science is storing up some real problems for the future. The vast majority of scientific results involving significant computation are unquestionably tainted by unquantifiable software errors. The continuing widespread occurrence of software failures and lack of a scientific discipline for handling them, makes this unpleasant conclusion inevitable. Things might finally be changing for the better but we have a lot to do.

This state of affairs is not new; before the age of computers mathematical physicists had to rely on log tables that were riddled with errors. Weird and fallacious theories were invented and modified to fit the data (see [22]).

Where did the code complexity go wrong? The answer will now be discussed.

2.3 The Mathematical Models: The Achilles' Heels

The mathematical model is not evident, mainly because its data model is not documented (more worrying, I suspect that the model resides in someone's brain and has not been "externalised" on paper).

The lack of *architectural or design blueprints* in the COVID-19 software is its Achilles' heel. The code is someone's mental model of the entities of interest (household, persons, country, whatever) and the relationships between them. It would seem that this mental model was mapped directly into C!

And what about developers in general?

1. They prefer talking about code more than understanding the model/problem.
2. They usually don't like documenting their models.
3. No time taken to discuss real problems because they are too busy fixing the code.
4. Users and developers don't always understand each other.
5. Many developers are not familiar with *divide-and-conquer* and *functional decomposition techniques*.
6. They tend to have idiosyncratic (and often erroneous) ideas on what constitutes good software. In general, some developers are obsessed with performance, while others wish to use new language features at the earliest possible opportunity. The structural solution is to determine which ISO 9126 software product quality characteristics are important and then to ensure that they are realised in the code (see [10] for a discussion).
7. They tend to be oblivious of the risks in projects in general.

A serious software project needs to pay attention to the following project activities:

1. Create a data/object model using ERD (Entity Relationship Diagram) or UML (Unified Modeling Language) depicting the COVID-19 abstractions and their structural relationships.
2. Critically, high-level data flow diagrams (DFDs) and system decomposition methods are essential if we wish to write maintainable modular systems (see [23]), particularly if multithreading is used to improve speedup. In other words, we must discover *potential concurrency* ([15], [16]). Parallel programming features were added to COVID-19 after the serial code was written. This opens a new Pandora's Box: write once and debug many times. For example, a later version 0.14.0 has

   ```
   #pragma omp parallel for schedule(static,500)
   reduction(+:recovery_time_days,recovery_time_timesteps) default(none) \
   shared(P, Hosts)
   ```

   while the current version is 0.7.0

   ```
   #pragma omp parallel for private(i) schedule(static,500)
   reduction(+:recovery_time_days,recovery_time_timesteps)
   ```

   What does this suggest? To me, much trial-and-error testing in a short period of time (developers are busy..). I wonder if proper testing has been done. I have had the same experience with OpenMP code in the past; you never really know if it really works. Unfortunately, customers will let you know, sometimes even after the software product has been shipped.

3. If the model is based on mathematical formulae, write them up and integrate them with the software models. The *cognitive distance* between the maths and C++ code should be as small as possible.

2.4 Random Number Generation

This is probably the most important section of the report.

The functions in `Rand.h` and `Rand.cpp` are a bit of a motley crew, to put it mildly. It is a central component of the program because it contains functions for random number generation and functions to generate variates of several univariate probability distributions, for example normal, lognormal, gamma and exponential distributions. Some features are:

1. Most of the code is "home grown". Nowadays, standard and tested C++ and Python libraries would be more suitable and more reliable.
2. The algorithms on which the code is based are almost forty years old and in many cases they are a direct 'manual' translation from Fortran to C (while still preserving a few hundred "harmful" GOTO statements).

3. There are single and "multi-threaded" versions of each function, the latter in obvious expectation of the arrival of OpenMP in the simulation part of the code.
4. The Polar Marsaglia algorithm (which is a special case of Box-Muller algorithm) is used to generate pairs of random numbers; this method is very old and unreliable. It is possible that the generated pairs are not independent. This is the *Neave effect* and It has been known for almost fifty years (see [6], [7]). Colloquially, we say that the current software is behind the curve as there are faster and more robust algorithms for the job at hand (see, for example [8]).
5. The code is littered with undocumented magic numbers which presumably started life as Fortran entities and now they have been promoted to the status of `static` and `const` C data.
6. The caveats on code quality as introduced in Section 2.1 probably also apply in the current context.

I have many more questions that I cannot ask here due to scope limitations. Some of my concerns about this part of the code are:

A. Even in single-threaded mode, I doubt that the code is robust and reliable enough for all use cases.
B. Has this code been benchmarked against *DieHard*, for example?
C. Multi-threaded code is difficult to get right, and it should be written by expert programmers. Anyone can inject an OpenMP #pragma into the code without understanding what the consequence are. Threads are heavyweight entities. Nowadays, *tasks* and *futures* are a better option as well as being part of standard C++11.
D. Consider C++11 *<random>, Boost.Random* and Python's *numpy.random*.

I would put `Rand.h` and `Rand.cpp` out to pasture/make obsolete and replace them by external (fully tested) standardised libraries.

2.5 Parallelisation and OpenMP

OpenMP is a parallel programming model for shared and distributed shared memory multiprocessors. It was pioneered by Silicon Graphics (SGI), an American high-performance computing manufacturer, producing computer hardware and software. One view of OpenMP is that it is a set of compiler directives (language extensions), run-time library routines and environment variables that comprise OpenMP's application programming interface (API) (see [13], [14]). It is supported by Fortran, C and C++. The latest version at the moment of writing is 5.0 (November 2018) while we note that Microsoft stopped supporting it after version 2.0. In a sense it has had its time (in my opinion) because multithreading and multitasking are already supported in C++11.

I have used OpenMP when it was the only kid on the block (around twenty years ago) but I found it to be too rigid for computational finance applications (PDE and Monte Carlo methods). Some problems I encountered were:

1. Its syntax and that of C++11 don't mix well.
2. It only works well for loop-level parallelisation.
3. Races and other non-deterministic behaviours can and do occur (for example, when different threads access unprotected shared data). This will destroy the reliability of random number generation.
4. It can happen that a parallel program running on N multicores on a developer machine and seemingly race condition-free will produce race conditions on a customer machine with 2*N multicores. Each run produces a different result (at unpredictable times).
5. OpenMP is not easily adaptable to more complex problems that demand using *parallel design patterns* (see [15], [16]).
6. Using OpenMP *critical sections* in files such as *Sweep.cpp* is a blunt instrument/sledgehammer. My suspicion is that it is trial-and-error and also bad for performance. This does not bode well. High alert now.

I get the impression that OpenMP is used as afterthought to speed up the horrendous loops in the code. This is equivalent to putting the cart before the horse. It will never work in the long term.

2.5.1 OpenMP versus C++ Threads versus C++ Asynchronous Tasks

To give an idea of the possibilities we give an (edited) example of random number generation using C++11 threads and C++11 tasks to price options in computational finance (more detail can be found in [3]):

```cpp
// Random number generators
auto rng1 = std::shared_ptr<Rng<PolarMarsaglia>>(new PolarMarsaglia());
auto rng2 = std::shared_ptr<Rng<MTRng>>(new MTRng());
auto rng3 = std::shared_ptr<Rng<MTRng64>>(new MTRng64());
auto rng4 = std::shared_ptr<Rng<FibonacciRng>>(new FibonacciRng());


// sde and fdm are objects ..
SUD<GBM, EuropeanPricer, EulerFdm<GBM>, PolarMarsaglia>
                    s(sde, pricerPut, fdm, rng1, NSim, NT);
SUD<GBM, EuropeanPricer, EulerFdm<GBM>, MTRng>
                    s2(sde, pricerPut, fdm, rng2, NSim, NT);
SUD<GBM, EuropeanPricer, EulerFdm<GBM>, MTRng64>
                    s3(sde, pricerCall, fdm, rng3, NSim, NT);
SUD<GBM, EuropeanPricer, EulerFdm<GBM>, FibonacciRng>
                    s4(sde, pricerCall, fdm, rng4, NSim, NT);

auto fn1 = [&]() {s.start();std::cout << "\n Polar Marsaglia: " << s.Price(); };
auto fn2 = [&]() {s2.start(); std::cout << "\n MT: " << s2.Price() << '\n'; };
auto fn3 = [&]() {s3.start(); std::cout << "\n MT 64: " << s3.Price() << '\n';  };
auto fn4 = [&]() {s4.start(); std::cout << "\n Fibonacci: "<< s4.Price()<<'\n'; };


// Whole lot of threads
std::vector<std::shared_ptr<std::thread>> threadGroup;
std::vector <std::function<void()>> tGroupFunctions = { fn1, fn2, fn3, fn4 };
for (std::size_t i = 0; i < tGroupFunctions.size(); ++i)
{
       threadGroup.emplace_back(std::shared_ptr<std::thread>
                          (new std::thread(tGroupFunctions[i])));
}
for (std::size_t i = 0; i < tGroupFunctions.size(); ++i)
{
       if (threadGroup[i]->joinable())
       {
              threadGroup[i]->join();
       }
}

// OMP
omp_set_num_threads(16);
std::cout << "Number of processors: " << omp_get_num_procs() << '\n';
std::cout << "Number of threads: " << omp_get_num_threads() << '\n';

std::vector <std::function<void()>> tGroupFunctions = { fn1, fn2, fn3, fn4 };

#pragma omp parallel for
for (int i = 0; i < tGroupFunctions.size(); ++i)
{
       tGroupFunctions[i]();
}
```

Finally, the Python equivalent of the function `SampleWithoutReplacement()` is a simple one-liner:

```python
#create a new list whose k items are unique
k = 4
seq = [1,2,3,4,5,6,7,8,9,10,11,12,13]
selection = random.sample(seq,k); print("Sampling w/o replacement", selection)
print("Sampling w/o replacement, original list ", seq)
selection = random.sample(seq,k); print("Sampling w/o replacement", selection)
selection = random.sample(seq,k); print("Sampling w/o replacement", selection)
```

3. The Elephant in the Room: No One Understands the Requirements

Just as I was completing this report I was made aware of a note from the *International Institute of Forecasters (IIF)*. The authors discuss some of the reasons why attempts to forecast the impact of the COVID-19 pandemic on the world's population in terms of the number of deaths, the effect on the economy, the prospect of starvation and the rise in mental health problems is severely flawed (see [24]). The authors paint a bleak picture and lack of confidence in science and scientists. We summarise these attention points that have direct relevance to this report:

1. Poor data input of key features (parameters) that are used in the models (for example, SIR (Susceptible, Infective, Recovered)).
2. Poor data input for data-based forecasting, for example time series (no consensus even on seemingly hard-core data).
3. Wrong assumptions in modelling (many models assume *homogeneity*).
4. High sensitivity of estimates (small errors in input may result in major deviations from reality). In numerical analysis, this is called an *ill-posed problem*.
5. Lack of incorporation of epidemiological features ("almost all COVID-19 mortality models focused on number of deaths, without considering age structure and comorbidities"). It is possible to create demographic models with continuous age as independent variable. However, we now get an initial boundary value problem for a first-order *hyperbolic partial differential equation* ([2], [27]). Such problems do not have analytical solutions and hence numerical methods (for example Finite Difference Method (FDM) or the Method of Characteristics (MOC)) are employed (see [28]).
6. Poor past evidence (for example, "The core evidence to support "flatten-the-curve" efforts was based on observational data from the 1918 Spanish flu pandemic on 43 US cities. These data are more than 100 years old, of questionable quality.")
7. Lack of transparency (many models are not disclosed, not peer-reviewed and even not published).
8. Errors (there errors refer to code as discussed in this report).
9. Lack of determinacy (most models are *stochastic* in nature and results must be reproducible). Computational finance feels comfortable with these models. It is not rocket science anymore. The field of epidemiology seems to be behind the curve here. For some new stochastic-based models, see [29], which will be familiar to quantitative analysts working in computational finance.
10. Looking at only one or a few dimensions of the problem at hand, for example the number of deaths whereas multiple outputs should be considered. This is a form of *tunnel vision*.
11. Lack of expertise in crucial disciplines (in many cases the credentials are not always disclosed). Lead scientists do not have expertise in all areas. A multi-disciplinary team is needed, as seen in professional software projects.
12. Groupthink and bandwagon effects (models that depend on theory and speculation can give any desired results or output by tweaking parameters, code hacking and other ad-hoc methods).
13. Selective reporting (in charged environments, forecasts may be more likely to be published if they produced results that are "pleasing" to politicians, for example).

These problems crop up in many projects but in this case the conclusions in [24] are depressing. At face value, epidemiology models are small and easy to program. But we can safely conclude that these projects come off the rails before a single line of code has been written (see again, [18]). We then need to get into *requirements analysis* (see [25], [26]). Good luck.

Conclusions

In my opinion, and based on my analysis, this software cannot be salvaged in its present form. It is a lost cause. The original code cannot be trusted for the reasons mentioned above and there is no point trying to improve it in small increments. A rethink and a rewrite are needed. To quote Fred Brooks [9]:

"The management question, therefore, is not whether to build a pilot system and throw it away. You will do that. The only question is whether to plan in advance to build a throwaway, or to promise to deliver the throwaway to customers."

What is also clear from reading the code is those who programmed up the code are neither experienced software developers nor are they are familiar with modern libraries (e.g. random number generation), data structures, C++11 and software design. More personnel with a range of skills are needed. A brief examination of later github versions would suggest that the project is attracting participants and bloggers who do not have this prerequisite background.

As already mentioned, regarding the COVID-19 software the only hope I see is to throw it away and start again. I would have two versions (by two different teams):
1. C++11 and libraries
2. Python and libraries

All these github versions are a road to nowhere. They only address the symptoms. It is a waste of time and energy.

About the Author

Daniel J. Duffy (real name) has been working since 1988 with C++ and its applications to computational finance, process-control, Computer-Aided Design (CAD) and holography (optical technology). His company Datasim was the first to promote C++ and object-oriented technology in the Netherlands. In the period 1979-1987 he worked on a range of scientific, oil/gas and engineering applications in FORTRAN using supercomputers, mainframes and minicomputers. He has trained thousands of practitioners and MSc/MFE degree students in the areas of requirements analysis, design, programming and advanced applied and numerical mathematics as well as being MSc supervisor for several top US and UK universities. He is the originator of two very popular C++ courses in cooperation with www.quantnet.com and Baruch College NYC and is the author of ten books on mathematics, software design, C++ and C#. Some testimonials can be found on www.datasim.nl

Daniel J. Duffy has BA (Mod), MSc and PhD degrees from Trinity College (University of Dublin, Trinity College), all in mathematics. He does judo as a way to relax.

I can be reached at dduffy@datasim.nl

References

[1] N. M. Ferguson, D. Laydon, G. Nedjati-Gilani et al. 2020 Impact of non-pharmaceutical interventions (NPIs) to reduce COVID-19 mortality and healthcare demand. Imperial College London (16-03-2020), doi: https://doi.org/10.25561/77482.

[2] H. W. Hethcote 2000 the Mathematics of Infectious Diseases, SIAM Review is currently published by Society for Industrial and Applied Mathematics.

[3] D. J. Duffy 2018 Financial Instrument Pricing using C++ (second edition), John Wiley and Sons Chichester UK.

[4] D. J. Duffy and Joerg Kienitz 2009 Monte Carlo Frameworks in C++ John Wiley and Sons Chichester UK.

[5] D. J. Duffy Beyond Object-Orientation: C++ Application Design for Computational Finance, A Defined Process from Problem to Parallel Code Wilmott Magazine.

https://onlinelibrary.wiley.com/doi/pdf/10.1002/wilm.10647

[6] P. Jaeckel 2002 Monte Carlo methods in finance, John Wiley and Sons Chichester UK.

[7] H. R. Neave 1973 On Using the Box-Muller Transformation with Multiplicative Congruential Pseudo-Random Number Generators, Journal of the Royal Statistical Society. Series C (Applied Statistics) Vol. 24, No. 1 (1975), pp. 132-135.

[8] M. E. O'Neill 2014. PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation (PDF) (Technical report). Harvey Mudd College. HMC-CS-2014-0905. (5 September 2014).

[9] F. Brooks 1975. The Mythical Man-Month. Addison-Wesley.

[10] ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models.

[11] B. Stroustrup 1994 The Design and Evolution of C++, Addison-Wesley.

[12] B. W. Kernighan and D. M. Ritchie 1988 The C Programming Language, Prentice-Hall.

[13] R. Chandra et al 2001 Parallel Programming in OpenMP Morgan Kaufman.

[14] B. Chapman et al 2008 Using OpenMP MIT Press.

[15] C. Campbell, A. Miller 2011 Parallel Programming with C++ Microsoft.

[16] T. G. Mattson et al 2005 Patterns for Parallel Programming Addison-Wesley.

[17] S. Denim (not real name) 2020 Second Analysis of Ferguson's Model

https://lockdownsceptics.org/second-analysis-of-fergusons-model/

[18] E. Yourdon Death March 2004 , Yourdon Press.

[19] T. J. MacCabe Structured Testing IEEE Computer Society Press, 1983.

[20] G. M. Weinberg 1998 The Psychology of Computer Programming, Dorset House.

[21] L. Hatton 2008 The role of empiricism in improving the reliability of future software.

[22] H. H. Goldstine 1977 A History of Numerical Analysis from the 16th through the 19th Century Springer.

[23] D. Duffy 2004 Domain Architectures, John Wiley and Sons Chichester UK.

[24] International Institute of Forecasters 2020 Forecasting for COVID-19 has failed.

https://forecasters.org/blog/2020/06/14/forecasting-for-covid-19-has-failed/

[25] I. Sommerville and P. Sawyer 1997 Requirements Engineering, John Wiley and Sons Chichester UK.

[26] CMU/SEI 1993 Taxonomy-Based Risk Identification, Technical Report CMU/SEI-93-TR-6, ESC-TR-93-183.

[27] R. E. Plant and L.T. Wilson 1986 Models for age structured populations with distributed maturation rates, Journal of Mathematical Biology 23: 247-262.

[28] Daniel J. Duffy 2006 Finite difference methods in financial engineering, John Wiley and Sons Chichester UK.

[29] L. Allen 2017 A primer on stochastic epidemic models: Formulation, numerical simulation, and analysis, Infectious Disease Modelling 2(2017), 128-142.