Section 10. Computational Finance

Disclaimer: unlike the other lectures, which have generally presented provable mathematical facts, this lecture is somewhat influenced by my own personal views.

Goals. There are many excellent books and courses on numerical methods for PDEs and monte carlo pricing methods, which are of course central to the field of computational finance. Fewer treatments actually cover software design issues – how to write design large, complex software systems which are nonetheless exceptionally stable and dependable, and which require minimal human intervention.

The following is our list of *software design goals*, ie, properties which we hope our software systems will have once completed. Some of these goals overlap, in the sense that one makes it easier to achieve the other:

(1) Readability

Upon examination of the source code, it should be abundantly clear what it is trying to do, even to someone who is not necessarily a subject matter expert.

(2) Modularity

The code should be organized into small, clearly defined "modules". Each module handles a specific task or closely-linked set of tasks. Code that is part of the internal calculations of the module and not intended for general use should not be accessible outside of the module.

(3) Maintainability

No great or unusual effort should be required to keep the code working.

(4) Extensibility

It should be possible to add features or functionality to a module without disturbing the rest of the code base. It should be possible for team members other than the original designer to extend the code.

(5) Stability

The production instance(s) should not have unpredictable behavior of any sort.

(6) Longevity

Code written today should continue to be useful for many years to come.

(7) Test Coverage

Each module should be accompanied by a set of tests which verify that the intended behavior is indeed satisfied for a fixed set of well-understood examples with known outcomes.

(8) Efficiency in Space and Time

This comes at the end of our list and is usually secondary to all of the goals above. If you can add 5% of efficiency to your code through extensive use of arrays, pointers, and shared memory, should you do so, at risk of failing in every other category listed above? This kind of efficiency comes at great cost, and hence should only be undertaken if there is a commensurately great benefit associated to it. The efficiency itself may be an illusion, for example, if it impedes thread-safety.

Many of these stylistic goals help achieve the others. For example, test coverage gives us the confidence to extend the code without impacting stability. Modularity tends to help with longevity – code written for a specific task might pass out of scope if that task is no longer needed, but if it's sufficiently modular, some of the modules might be adapted to other tasks and hence "live on."

Languages and Libraries. Our ability to achieve (or not) the above goals depends largely on the choice of language. Mostly, the same comments apply to choosing whether or not to depend on a particular add-on package, dependency, or external library. For example, it's hard to write code that can continue to be useful in 5 years, 10 years, or more if the language it's written in is probably going away before that. When beginning a new project or new business venture, you have the opportunity to choose what languages and frameworks you will depend on.

(1) Support and Critical Mass

It should be very widely used and supported by an active community including a large number (hundreds or thousands) of professional developers. There should be lots of examples of source code available on the internet covering most of the tasks or issues you're likely to encounter. There shouldn't be "key man risk" in the sense of one central person who would effectively kill the project if they were to move on.

(2) Activity and Longevity

It should be actively developed and supported, with every expectation that it will be actively developed and supported for all of the foreseeable future. There should not be a plan to discontinue it at some future point. Nothing that you rely on in production should restrict you to a particular version of anything (example: one firm wasn't able to upgrade to Java 8 because Matlab features they relied on only worked with Java versions ≤ 6 at the time.)

(3) Backwards Compatibility

This is related to longevity – code you write ideally would not suddenly stop working in the next version of the language or framework. The language should evolve more through new features and additional capabilities being added, than tried-and-true technologies being deprecated or substantially changed. Examples include Java, which adheres to a rather strict backwards compatibility standard. An inverse example is Python, in which case the entire language was essentially changed between versions 2 and 3.

(4) Openness

The language/framework should be as open as possible, in the sense that the core framework is open source (or at least, has open source implementations available).

(5) Standard Libraries

The language/framework should ship with a set of standard libraries which handle all common tasks such as file and network I/O, parsing common formats, working with dates, times, time zones, accessing databases, common computer-science data structures such as arrays, linked lists and hash tables, etc etc. If you find yourself needing to write your own implementation of any of the just-mentioned structures, you probably chose the wrong language!

(6) Ease of Development

The language (and its extensions) should encourage all of the desirable development practices listed above. There should be editors that facilitate rapid development.

Again, these are inter-related. Openness and Critical mass help to ensure longevity. Existence of good standard libraries contributes to ease of development. Adoption of a given language or framework by one or more very large software companies can be a good indicator: if Google, Apple and Facebook (or other similar industry peers) all rely heavily on a given language or

framework, it's probably safe for you to rely on it as well, assuming it meets the other criteria above. They can essentially ensure the longevity, activity, and critical mass of a language due to the resources they can deploy.

Side effects and Immutability.

Definition 10.1. a function or expression is said to have a side effect if it modifies some state or has an observable interaction with calling functions or the outside world.

For example, a particular function might modify a global variable or static variable, modify one of its arguments, raise an exception, write data to a display or file, read data, or call other side-effecting functions. In the presence of side effects, a program's behavior may depend on history; that is, the order of evaluation matters. Understanding and debugging a function with side effects requires knowledge about the context and its possible histories.

Definition 10.2. An immutable object is an object whose state cannot be modified after it is created.

Strings and other concrete objects are typically expressed as immutable objects to improve readability and runtime efficiency in object-oriented programming. Immutable objects are also *inherently thread-safe*. Immutable objects are generally simpler to understand and reason about and offer higher security than mutable objects. Immutable objects allow you to control side effects. In particular, no function can have a side effect on its arguments if all of its arguments are immutable objects.

Imagine a program that consisted solely of side-effect free functions acting on immutable objects to produce other immutable objects. Such a program would be very easy to parallelize and to reason about its behavior in a multi-threaded environment. No object can be modified by another thread while the current thread is reading it, because no object can be modified, period.

When working with immutable objects, a "modification" happens typically by creating a new object with the desired property. Consider immutable representations of a *Fill* as discussed in the lecture on slippage. For the purposes of this example, we assume the existence of an immutable implementation of *Order* which we will not write down here.

```
public class Fill {
    private final Order parent;
    private final Shares filled;
```

```
    private Fill(Order order, Shares n) {
        parent = order;
        filled = n;
    }

    public static Fill of(Order order, Shares n) {
        if(order == null)
          throw new IllegalArgumentException("the parent order " +
              "of a fill cannot be null");
        return new Fill(order, n);
    }

    public Fill withAdditionalQuantity(Shares additional) {
        return new Fill(parent, filled.plus(additional));
    }
}
```

There are several takeaways to learn here. First, note that it is impossible to create a Fill with a null parent order. This is because the constructor is private and actual construction is limited to the `of` method, which checks its argument. More importantly, note that when we add additional quantity to the `filled` member, the parent Order is not copied – it just comes along for the ride, which is perfectly safe because it's immutable. This is efficient because it avoids copying.

This sketch is incomplete, but hopefully suggests certain patterns which you will want to follow up on and extend.

Languages and Frameworks.

Java, C/C++, Python, Matlab and R are all heavily used in finance. These all have their strengths and weaknesses, but Java and C++ are probably the closest to achieving absolutely *all* of the desirable properties above. Python, Matlab, and R are *interpreted* or *scripting* languages, while Java and C++ are fully object oriented, strongly typed, compiled languages suitable for developing large and complex applications.

The main difference between Java and C++ is that in Java, memory management is automatic and handled by the virtual machine. In C++, memory management must be explicitly controlled by the program. The

second main difference is that C++ allows multiple inheritance. Java allows a class to implement multiple interfaces, but not to inherit from multiple superclasses. Both of these features of Java eliminate large classes of possible bugs.

Definition 10.3. The *Composition over inheritance Principle* in object-oriented programming is the principle that classes should achieve polymorphic behavior and code reuse by composition (containing other classes that implement the desired functionality), instead of through inheritance (being a subclass).

By disallowing multiple inheritance, Java does a better job than C++ of encouraging programmers towards the composition over inheritance principle.

There is an excellent free development environment for Java called *IntelliJ IDEA* which saves programmers a lot of time because it automates common tasks such as refactoring and supports automatic code completion.

Nowadays Python can do a lot of what previously would have been accomplished in Matlab and R. In particular, the *pandas* package is essentially a port of a subset of the core features in R.

Backtesting.

Backtesting, or *simulation* as I prefer to call it, is one of the most important tasks in creating a quantitative trading strategy.

Any data point has some time when it could have been knowable to the system. For example, close prices (the last price of the day for each security) are typically known to just about everyone at the time the exchange or market closes for the day. Importantly, this time is an *Instant*, as are all knowledge times of all data points. An *Instant* can be defined as a particular point on a linear timeline that is independent of any particular calendar system.

Definition 10.4. An *Event* e is any kind of data or information, together with an instant t_e called the *knowledge instant* of the event. This is defined so that the agent could have known the data or information contained in e at the time instant t_e.

Definition 10.5. A *Simulation* is a way of replaying events in time order, and ensuring that any calculations which depend on those events are triggered at the appropriate time(s).

Here we see a possible simplification due to immutability: if an *Event* object is fully immutable, then the same Event object can be passed to many calculators, without the controller or scheduler worrying about whether something has modified the internal state of the Event object (i.e. the data or information it contains).