Model Risk: The Challenges of Legacy Code and Best Practices

April 2014

David Eliezer, PhD, Vice President, Head of Model Validation, Numerix LLC

Numerix LLC
Corporate Headquarters
99 Park Avenue
5th Floor
New York, NY 10016
Tel: +1 212 302 2220

The model risk inherent in derivative pricing models has received a burst of renewed interest since the global financial crisis. Driven partly by regulators and partly by internal pressure to avoid losses due to poor modeling, the emerging trend is for financial institutions to utilize techniques for measuring and mitigating model risk. Although these ideas have existed in some form for many years, this revived attention to the field has lit the spark for a renewed inquiry into best practices for development of pricing models. In this paper, we will examine the most common types and sources of model risk, and then outline best practices that practitioners can utilize in their model validation processes.

In order to restore the public trust in the financial industry, the trust that we need to do our work, we need to approach model risk with a new seriousness of purpose.

SOURCES OF MODEL RISK: DETECTION AND PREVENTION

Model risk has become a topic of great interest, and deservedly so, after the credit crisis of 2008. A key component of the most recent credit crisis, though not the only one, included 'buggy' and inaccurate models.

The financial crisis 'post-mortem' revealed that the crisis was made worse by the misuse of models—from poor modeling and dynamics to bad coding and practices. We've come to see that the sources of model risk may be found all along the production line for models, as well as in their usage. The bugs relevant to model risk fall into a few well-known categories.

Categories of Bugs Leading to Model Risk
A Checklist

- ✔ Model pricer is not pricing according to the model
- ✔ Model pricer is pricing accurately, but greeks are far off
- ✔ Model calibrator fails to fit the model to the market well
- ✔ Model is not compatible with market dynamics
- ✔ Bad inputs to models, for example:
  - *Inputs inadequately labeled, and passed into the wrong function argument*
  - *Traders asked to set numerical parameters for which they are not knowledgeable*
  - *Change of regime makes the model's dynamics obsolete (i.e. Kurtosis in volatility surface after 1987 crash; Low interest rate regimes that occurred after 2008 mortgage crisis and other 'crashes')*

CHALLENGES WITH POORLY MAINTAINED LEGACY CODE

Code bases have a life of their own; and when code is not properly maintained, it begins to 'rot.' The end state of this process is a code base that is drowning in bugs and must be thrown away. This, in turn, leads to other attendant risks related to production and PnL. What these challenges look like, in the context of mathematical financial code, is likely very familiar to this audience. Let's take a closer look...

Symptoms of Unhealthy Code
A Checklist

- ✔ Too much time spent debugging, instead of developing
- ✔ Known bugs that no one knows how to solve
- ✔ Too much reliance on 'workarounds'
- ✔ Traders do not trust the model, or manually intervene to change the parameters significantly
- ✔ Lots of bleed/ hedged portfolio is very volatile
- ✔ Inaccurate re-pricings of calibration instruments
- ✔ Unstable calibrations, i.e. calibrations that sometimes change radically when inputs are changed by a small amount

HOW MODEL RISK ORIGINATES

Where does model risk originate? One cause is the manner in which the financial industry develops code, when compared to other types of software industries. This partially stems from the smaller set of users for the code, the constantly critical (i.e. short) delivery schedules and the complexity of numerical programming that exists in financial institutions. For example, financial industry developers often code alone, with only minimal reviews and testing. This is further exacerbated by the overall complexity of numerical and mathematical programming. What are the practices that exacerbate model risk?

Practices Causing Model Risk:*
A Checklist

- ✔ Insufficiently broad and thorough testing regime
- ✔ Weak or nonexistent code reviews. Code reviews should review coding methods, but also clarity of code and code documentation, as well as the testing cases, and testing methods.
- ✔ Market regime changes undetected/ Model dynamics and numerical approximations become obsolete.
  (For example, a new market regime with a much higher volatility might necessitate either a model with a different volatility structure, OR a numerical method that is more accurate at the new level of volatility. To detect new market regimes, there should be daily tests in place, such as PnL Attribution Tests, that measure hedging errors and send a warning when the errors grow too large over a sustained period. Additionally, the regions where numerical methods become inaccurate should be mapped out as part of the development process.)
- ✔ Too much "ownership" of code, not enough joint sharing of code causes institutional knowledge of the code base to leave when authors leave
- ✔ Inadequate documentation. Documentation should be adequate to fully explain both the model and numerical method, so that a reader can reproduce the exact numbers produced by the model.
- ✔ Large, complex code blocks are allowed to fester and grow. Left to themselves, developers under time pressure will always choose the quickest coding method, and this often means adding another case to an already opaque block of code. Such blocks of code must be monitored, and re-factored when they grow beyond what can be easily understood.

*(Note: This is not an exhaustive list.)

BEST PRACTICES FOR PREVENTING UNHEALTHY (MATHEMATICAL) CODE BASE

If bug detection is good, then bug prevention is divine.

A variety of tests can be performed to detect bugs and other problems, but prevention is always a far better and cheaper solution. We may prevent bugs by utilizing good coding practices, specifically tailored for the kind of code we are writing—which is primarily mathematical code. Mathematical code is more difficult to debug than other types of code, yet more easy to test, and the coding regime should take full advantage of this fact.

As most coders know, coding errors are suppressed by writing clear code. The inputs should be clearly labeled, and different algorithms should be separated into separate functions, rather than mixed. It is a common practice to save computation time by "inlining" certain numerical methods inside of other, larger ones (which mixes them together). But when this done, the code becomes more difficult to test. It also becomes more difficult for developers to isolate bugs. In all but the most extreme cases, the time saved by the practice of inlining is outweighed by the time lost in hunting for bugs. In the most extreme cases, where the mixed algorithm is a necessity, the risks must be mitigated by making an unmixed, slower version of the code that is tested in the usual way, and then writing a faster mixed version which can be tested against the slow version.

In the case of coding time, rushed coding always risks generating bugs. The only way to mitigate this is to have a ready library of components with clear meanings, which can be rapidly put together into any desired calculation. In this case, a higher-level language like Matlab may be more suitable, or a well-developed library of C++ tools. In the checklist that follows, we've compiled best practices for preventing unhealthy code.

Best Practices for Promoting and Maintaining Healthy Code
A Checklist

- ✔ Strong unit testing at multiple levels
- ✔ Test for and remove all memory errors/leaks
- ✔ A testing suite that can be run by developers anytime code is changed, as an aid to rapid development
  - *Tests must be run over many cases, thousands or millions where possible, with an automated failure condition*
    - Documentation must explain exactly what mathematical quantity is contained within each variable, at each stage of the code, and the exact quantities returned by each function, as well as any side effects or changes of state
    - Very often, an approximation method works well only in a certain part of the parameter space. This domain of applicability must be measured, and warnings or errors put in place in the function—so that the function does not just silently run and produce a wrong answer, without the user knowing.
- ✔ Code reviews, with partner developer(s) who knows they will have to debug the code in the author's absence, and so must be sure they understand the behavior of the code
- ✔ Enforcing of coding standards
  - *Don't allow opaque code*
    - Clear variable/function names: Each variable should have a single meaning, and should not be reused for other purposes
    - Each function or clause should have a single, identifiable purpose
      - *Failing this, heavy commenting making every calculation clear*
  - *Don't mix algorithms*
    - Separate each algorithm into its own function.
    - Mixed algorithm code cannot be subjected to unit testing. It also makes the option pricing code harder to understand.
    - Isolating separate algorithms in separate functions makes it easy to swap out old algorithms for newer, improved ones
  - *No non-local algorithms*
    - It is sometimes found that two calculations share a common fragment, and this fragment value is passed into a function as an argument
    - This prevents sensible unit testing, as the fragment obeys no known mathematical identity
    - The lack of a mathematical interpretation makes the code hard to understand
    - The function will not be re-usable
    - The algorithm cannot easily be swapped out for a new one

continues

Best Practices for Promoting and Maintaining Healthy Code *(continued)*

  - Users should only enter parameters that they are knowledgeable about. Traders, for example, should not have to enter parameters, neither approximation parameters, such as a number of paths, nor a calibration strategy—nothing except market information. These parameters should be set algorithmically, and the algorithm subjected to its own battery of tests.

- *As code becomes more complex, it must be re-factored regularly. In particular, large opaque blocks of code, especially if-then blocks, must not be allowed to fester and grow. Monitor them and break them up when they get too large.*
- *Strong labeling of inputs*
  - Very often a function will have a large number of inputs with similar types and similar names
  - It is impossible to predict the level of skill and experience of the developers who will maintain the code. They may be very junior, or lack mathematical skill or financial experience.
  - Inputs must be so clearly labeled that such developers cannot make the mistake of passing the wrong input to the function
  - To mitigate this, we can package up the large number of inputs into a larger object with well understood behavior, such as a yield curve or volatility surface. In this case, strong typing of the programming language will be very helpful.
  - If that is not possible, informative variable names, and commenting must be very elaborate

CONCLUSION

Model risk has gained renewed importance since the 2008 financial crisis. A strong contributor to model risk are the various coding errors that financial developers are prone to. But model risk can be mitigated through good coding practices, documentation, and a strong testing regime. This saves money and time, as debugging is always slower than testing, and discovering bugs by experiencing a financial loss is always the most expensive way.

This paper has touched on only a small part of where the errors come from, what can be done to detect them, and how we can prevent them. Financial practitioners must be vigilant, watching out for the health of their legacy code. Good coding practices prevent the decay of the health of a code base. In order to survive and thrive in today's competitive landscape, financial institutions must enforce and prioritize strong coding standards. Great improvements to our process and to our industry are possible through a smart approach to code maintenance and development.

AUTHOR BIOGRAPHY

David Eliezer, PhD, Vice President, Head of Model Validation, Numerix LLC

Mr. Eliezer has been a quant on Wall Street for 18 years, at Goldman-Sachs, Morgan Stanley, General Re Financial Products, and Bloomberg, among others. He has published work on option pricing, and on modeling liquidity in finance. He runs the internal testbed for Numerix models, and he leads the Model Validation group at Numerix.

Ready to elevate your business?

Get started →

More than 550 clients and 90 partners across 52 countries rely on Numerix analytics. Headquartered in the financial capital of the world, with offices around the globe, Numerix brings together unparalleled expertise combined with local market understanding.

Numerix LLC Corporate Headquarters

100 Park Avenue
15th Floor
New York, NY 10017
Tel: +1.212.302.2220
Fax: +1.212.302.6934
www.numerix.com

Copyright © Numerix LLC. All rights reserved. Numerix and the Numerix logos are registered trademarks or trademarks of Numerix LLC in the United States and/or other countries.