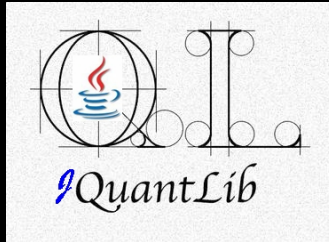


JQuantLib

A framework for Quantitative Finance written in Java



==JQuantLib==

JQuantLib is a Quantitative Finance framework written in Java.

Quantitative Finance

What is Quantitative Finance?

- Valuation of financial instruments
- Involves complex mathematics and statistics

How it is used?

- Model trading strategies
- Determine optimal investment portfolios
- Testing of new models and strategies
- Risk valuation and management

===Quantitative Finance===

But...

What is Quantitative Finance?

Other wordings:

- * Mathematical Finance
- * Financial Engineering
- * Finance Computing
- * ...

It's something which...

- * Allows price valuation of financial instruments, such as stocks, options, bonds, futures, etc;
- * Involves complex mathematics and statistics like linear algebra, interpolations, extrapolations, probability distributions, stochastic calculus, etc. Some calculations are CPU intensive and may consume a lot of resources, like numerical integrations and big matrix operations, to mention a few.

How Quantitative Finance is used?

It can be used to...

- * Model trading strategies and determine optimal investment portfolios needed to forecast return on investment;
- * Risk Valuation risk associated to investments
- * Test new products, models and strategies

Who's interested on Quantitative Finance?

- * Investment banks and hedge funds
- * High skilled investors in general
- * Academics

How Quantitative Finance affects us, Java developers?

The world of Quantitative Finance is very rewarding, offering very good salaries. Most companies opt for C++ but Java is slowly gaining more market share.

Companies which adopting Java are Goldman Sachs and CMC Markets, CS Group, to mention a few.

Background

JQuantLib is based on QuantLib (implemented in C++)

QuantLib

- Started in 2000
- ~35 developers
- MSVC++ / GCC
- Production quality v0.9.7 (November 18th, 2008)
- Near 2m lines of code
- Ported several languages (C# (QLNet), Python, Perl, Ruby...)

===Background===

JQuantLib is based on QuantLib, which is written in C++.

Some words about QuantLib

- * It started in 2000;
- * It has more than 35 contributors and near 2 million lines of code;
- * Current v0.9.7 (Nov/2008) is a near-production quality product;
- * It compiles under MSVC and GCC;
- * It has ports to several languages. Most ports are based on SWIG wrappers. There are some initiatives to translate source code to other languages, such as C#;

More about how to mimic templates in Java will be explored later.

The Need of JQuantLib

QuantLib misses integration with Java

Evaluation of QuantLib showed

- SWIG wrappers: inconvenient, incomplete, wrong
- JNI: inconvenient, complex, counterproductive
- SWIG/JNI: Difficult customization / extensibility

Alternatives

- Integration via J2EE and CORBA containers
- Translate QuantLib to Java : ~1300 classes

===The Need of JQuantLib===

During 3 months QuantLib was evaluated from the perspective of integration with a Java front-end.

* We discovered that SWIG wrappers offered by QuantLib are inconvenient, incomplete and sometimes even wrong;

* An alternative would be to use JNI as integration technology. This idea was abandoned because JNI is inconvenient, complex, error prone, slow and counterproductive.

Then some alternatives were evaluated:

* Integrate Java and C++ worlds via J2EE and CORBA containers. This idea was quickly abandoned due to exaggerated complexity and performance concerns.

* Translate QuantLib to Java, which involves migrating around 1300 classes of near 2 million lines of code.

We opted by the 2nd approach, because of the results we aim to obtain, in spite it is certainly a very challenging task.

Objectives

JQuantLib aims...

- Translate QuantLib, which is written in C++
- Offer syntax and semantics Java developers expect but keeping JQuantLib API as close as possible to QuantLib API
- Be exceptionally well coded, accurate and well documented
- Take advantage of features Java can offer

===Objectives===

JQuantLib aims to:

- * Translate QuantLib to 100% pure Java;
- * Offer syntax and semantics Java developers expect but keeping JQuantLib API as close as possible to QuantLib API;
- * Be exceptionally well coded, accurate and well documented;
- * Take advantage of features Java can offer as a language, as a execution environment and as a platform in general.

Challenges

QuantLib (C++)

- Very complex object model (example: Monte Carlo)
- Abuse of templates and other idioms

Other challenges

- Maximum accuracy
- Very strong type checking at compile time
- Relative bad performance of Java
- Latency due to Objects and GC

===Challenges===

QuantLib (C++) is very well designed and implemented but it imposes some challenges:

- * The object model is over complicated
- * Abuse of templates, which are difficult to mimic properly in Java
- * Use of some C++ idioms, specially templates which can be difficult to translate to Java

In addition, there are other challenges:

- * Obtain maximum accuracy without imposing performance penalties
- * Obtain very strong type checking at compile time in order to avoid performance penalties and run time errors
- * Relative bad performance when comparing to C++
- * Performance penalties imposed by Objects and operation of GC

Current Status - Jan/2009

- Started September 2007
- Coding started January 2008
- ~10 active developer
- First release in June 2008
- 40% of classes translated (January 2009)

===Current status of JQuantLib===

This is the status as for sep/2008:

- * Started on sep/2007
- * Coding started on jan/2008
- * About 10 active developers;
- * First released in jun/2008
- * 30% of translation task by number of classes

Features

From QuantLib

- Day counters, calendars, IMM
- Term structures, yield structures
- Instruments: stocks, options, bonds, swaps, etc
- Methods: Black-Scholes, binomial, LMM, MonteCarlo, low-discrepancy numbers, etc

JQuantLib specific (planned)

- OSGi support
- Support for parallelism (Parallel Colt, <http://piotr.wendykier.googlepages.com/parallelcolt>)
- Grid enabled

===Features===

Most features of JQuantLib are simply borrowed from QuantLib, such as

- * Financial instruments: stocks, options, futures, future options, swaps, swap options, bonds, currencies, etc
- * Methods: Black-Scholes, Binomial, Monte Carlo, low-discrepancy numbers (Sobol) etc
- * and much more

Other features are specific to JQuantLib

- * OSGi support is important for providing high availability and configuration flexibility in production
- * Support for parallelism is critical for scalability and performance. JQuantLib aims to provide parallelism via support libraries which support parallelism, including mathematical libraries. Also, being thread-safe, JQuantLib can provide parallelism itself
- * JQuantLib can also deliver tasks to be run on other nodes of a grid. Doing so, more scalability and more parallelism can be obtained.

Back to OSGi, functionalities can be selected and adapted to underlying infrastructure by the choice of adequate modules (called OSGi bundles). Example: The user can use what implementation of a randomizer he or she wants to use without the need of restarting the system.

Architecture

- Build Environment
- Quality Assurance
- Documentation
- Accuracy
- Strong Type Checking
- Performance

===Archicecture===

Here we will expose how JQuantLib addresses requirements and objectives. Also, we will present some details about build tools, etc.

This is our agenda:

- * Build Environment
- * Quality Assurance
- * Documentation
- * Accuracy
- * Strong Type Checking
- * Performance

In this presentation we will explore important aspects taken into account aiming to reach features we need whilst aiming to obtain the best performance as possible.

In the world of quantitative finance, performance is a critical factor of success. Other critical factors are correctness and accuracy.

Build Environment

- Linux
- Java6
- Eclipse
- KDE
- Umbrello
- SVN
- Maven
- Continuum
- Archiva

====Build Environment====

A choice for open source solutions, flexibility, automation and integration between all the components involved defined the preferred platform and tools.

- * Linux because it's open and free;
- * Java6 because it provides additional functionalities over Java5;
- * KDE because it runs on Linux, BSD, Windows and Mac.
- * Eclipse because it's a de-facto standard and provides an integrated OSGi container;
- * Umbrello is a good enough UML modelling tool which runs on KDE and has a version for Windows;
- * Source code management: Subversion (SVN) because it is the natural successor of CVS and it is mature enough;
- * Build tool: Maven because it provides dependency management integrated with ordinary build tasks. It can also run Ant tasks if needed.
- * Continuous integration: Continuum because it integrates seamlessly with Maven;

* Artifact management: Archiva because it integrates well with Continuum.

Quality Assurance

- JUnit4
- PMD
- FindBugs
- EclEmma
- Cobertura
- Mantis

====Quality Assurance====

We use these tools:

* JUnit4 and Cobertura are integrated with Maven in order to collect results of test cases. Reports generated by Cobertura can be seen in the website generated by Maven

* PMD and FindBugs are also integrated with Maven and generate reports containing violation to coding standards, bad practices and so on. FindBugs is also integrated with Eclipse as a plugin

* EclEmma provides code coverage integrated with Eclipse which is extremely helpful for testing purposes

* Macker has a plugin for Maven which can be used to look for regular expressions we'd like to avoid in the source code. A good example is that we'd like to erradicate the use of `java.lang.Double` in order to avoid autoboxing.

Correctness

Strong type checking

- Enumerations
- Generic types
- Annotation on Java types (JSR-308)

Accuracy

- Floating point rounding errors

====Correctness====

JQuantLib must be correct.

The compiler is our best friend on this difficult task.

By using idioms like enumerations, generic types and annotations on types we can help the compiler to help us. We will discuss this topic further on.

Correctness implies that calculations must be correct and, in particular, must match results provided by QuantLib(C++). We will discuss this topic soon.

Annotations on Java types

```
double calc(double rate, double year) {  
    return Math.exp(1+rate,year);  
}
```

```
double rate = 0.45;  
double year = 0.5;
```

```
double result1 = calc(rate, year);  
double result2 = calc(year, rate); // Wrong, not recognized by the compiler
```

Need for semantic checking

- Erradicate errors at compile time
- Test cases are not 100% guaranteed :(

```
C++  
typedef double Rate;  
typedef double Year;  
double calc(Rate rate, Year year);
```

====Annotations on Java types====

In this slide we will talk about how very strong type checking can be obtained using annotations. First of all, lets examine what is the need for very strong type checking.

Consider the piece of code you can see on top:

```
double calc(double rate, double year) {  
    return Math.exp(1+rate,year);  
}
```

```
double rate = 0.45;  
double year = 0.5;
```

```
double result1 = calc(rate, year);  
double result2 = calc(year, rate);
```

It shows a situation where 2 double variables where swaped because there's no semantic enforcement at compile time of what the parameters are and which variables should be accept or reject for each parameter.

""About test cases

In spite test cases can point out most of these situations, test cases which pass "do not mean" that our code is "certainly right": they only "mean" that our code is "not certainly wrong".

""What C++ provides

Let's examine the piece of code on the bottom.

```
typedef double Rate;  
typedef double Year;  
  
double calc(Rate rate, Year year);
```

Notice that typedefs improves how code can be understood but it does not prevent the compiler for accepting a Rate where an Year is expected. This is because both Rate and Year are simply double variables, in fact.

JSR-308

```
private Double calc(@Rate double rate, @Time double time) {  
    return new Double (Mathc.exp(1+rate, time));  
}
```

```
@Rate double rate = 0.45;  
@Time double time = 0.5;
```

```
// This call will pass  
Double result1 = calc(rate, time);
```

```
// This call can give us a compiler error  
Double result2 = calc(time, rate);
```

- Available in JDK7 (target date for JDK7: Early 2010)
- Annotations wherever a type is accepted
- Annotation processor plugged into compilation phase

Link

<http://groups.csail.mit.edu/pag/jsr308/>

====JSR-308====

How JQuantLib addresses this issue:

Going straight to the solution, we are using a very interesting feature of JDK7 known as JSR-308. Let's see the example:

```
private Double calc(@Rate double rate, @Time double time) {  
    return new Double( Math.exp(1+rate, time) );  
}
```

```
@Rate double rate = 0.45;  
@Time double time = 0.5;
```

```
// This call pass  
Double result1 = calc(rate, time);
```

```
// This call *can* give us a compiler error  
Double result2 = calc(time, rate);
```

It consists of allowing annotations wherever a type is allowed. Annotations add semantic meaning

to types. By the use of annotation processors, which can be plugged into javac (Java Compiler) it is possible to provoke compilation errors.

JSR-308 allows Java to provide even stronger and more flexible type checkings than C++.

Accuracy

```
System.out.println(0.1 + 0.1 + 0.1);
```

```
0.30000000000000004
```

Requirements

- As accurate as QuantLib (C++)
- As lightweight as possible
- As fast as possible

Solution

- Primitive types
- Calculate epsilon when needed

===Accuracy===

On the right top we can see the simplest example of mathematical inaccuracy due to floating point rounding errors:

This kind of error happens not due to the programming language but due to the way computers represent floating point data.

It's good to mention that more operations are done with inaccurate data the bigger the error becomes (example: Monte Carlo simulations).

'''Requirements

- * To be as accurate as QuantLib is;
- * To be as lightweight as possible;
- * To be as fast as possible, in particular: avoid impacts of object allocation and garbage collection.

'''Solution

JQuantLib takes the same approach as QuantLib. It consists of calculation of epsilon after a sequence of mathematical operations which gives us the order of magnitude of the error.

No Objects are used, only primitive types.

Documentation

Requirements

1. Replace doxygen

Solution

1. UMLGraph
2. LaTeXtaglet

Links

<http://www.umlgraph.org/>

Tool for testing Latex formulas: Laeqed

====Documentation====

""What are the requirements?

We decided to mimic original QuantLib documentation, which is generated by doxygen containing UML diagrams and mathematical formulas in addition to ordinary code documentation.

""JQuantLib approach is

1. UMLGraph <http://www.umlgraph.org/> is tool written in Java which can be easily integrated with javadoc and produces various UML diagrams. In particular, we can say that the default configuration of UMLGraph is what we need regarding UML diagrams.

2. Embed mathematical formulas in Javadocs via taglets.

We modified a tool called LaTeXtaglet in order to better integrate with Linux as it had Windows-dependent code.

Tool for testing Latex formulas: Laeqed

""Links

[<http://www.umlgraph.org/> UMLGraph]

[http://www.jquantlib.org/index.php/Building_JQuantLib Building JQuantLib]

Performance

- Critical requirement
- Needed for low latency
- Needed for scalability

Topics

- Comparison with C++
- Numbers and Objects
- Collections
- Math Packages
- Parallelism
- JVM and gc
- Profiling

===Performance===

One of the most important requirements for a financial package is high performance. It guarantees that results are calculated on time and it potentially enables applications to scale.

Performance is a difficult matter and comparisons are subjective in general. Anyway, we will try to show that Java is ready for serious high performance, low latency financial applications.

We will talk about some techniques, tools and products which can help us to get most of language, JVM and hardware.

These are the items we will cover:

- * Comparison with C++
- * Numbers and Objects
- * Collections
- * Math Packages
- * Parallelism
- * JVM and gc
- * Profiling

Comparison with C++

Benchmarks in DDJ (Java 5)

- 32bit integer arithmetic: as fast as
- 64bit double arithmetic: as fast as
- sort algorithms: 50% slower
- list operations: 2x slower
- matrix operations: 2x to 3x slower
- nested loops: 2x slower
- trigonometric functions: deadly slow!

<http://www.jquantlib.org/index.php/DesignPerformance>

====Comparison with C++====

The results we present here were taken from Dr. Dobbs Journal and compares C++ with Java5. We have to remember that performance comparisons are always subjective but these figures can help us build an overall scenario.

* 32 and 64 bit arithmetic present roughly the same performance. These are the most important operations from JQuantLib point of view because they are the most common mathematical operations we will perform.

* sort algorithms, list operations and matrix operations may be 2 to 3 times slower in Java. Most of slowness is due to object creation, garbage collection and autoboxing. This is a "region in our domain" which can be potentially improved in Java.

* Trigonometric functions are deadly slow, in general.

Actually, these results may vary a lot depending on JVM implementation and hardware platform.

The link below ...

<http://www.jquantlib.org/index.php/DesignPerformance>

... contains links to this study taken from Dr. Dobbs Journal and also some other interesting links.

Another point to mention is that Java does not provide "unsigned" integer arithmetic. It only provides "signed" integer arithmetic. In spite of this issue can be circumvented in most situations, there are certain situations where you will have to perform additional operations in order to obtain the correct result. It obviously has impacts on performance. This is not only a lack of Java, but a lack in JVM: it means that Groovy, Scala and all other JVM based languages will present the same issue.

Collections

Issues

- Not optimised for high performance systems
- Expansive object management and reference

Alternatives

- Arrays of primitive types
- Optimised JCF implementation

fastutil `List list = DoubleArrayList(); // backed by an array of doubles
list.add(1.0); // autoboxing :: list.add(new Double(1.0));
(DoubleArrayList)list.add(2.0); // no autoboxing :)
double d = (DoubleArrayList)list.getDouble(0); // no autoboxing`

<http://fastutil.dsi.unimi.it/>

====Collections====

Java Collections Framework is a collection of data structures which are certainly very useful. But standard JCF is not optimised for high performance systems. Every element of a collection is an object which demands to be created, referenced and released at collection destruction. A large collection which involves several operations may be too expensive from the performance point of view.

As an alternative, we can use regular arrays of primitive types instead of Collections. This alternative can be good on several circumstances but certainly does not offer the flexibility JCF provides.

JQuantLib uses fastutil, which is an implementation of JCF interfaces backed on arrays of primitive types. From the user point of view, it's pretty much JCF but there are certain methods intended to avoid autoboxing.

The snippet of code shows the usage of DoubleArrayList, which is a List but backed by an array of primitive type doubles, not class Double, I mean. FastUtil manages the growth of this array, as you would expect.

The second line does not have anything different from what you would expect but it inserts an object of type Double which involves autoboxing.

The third line shows an extension to the well known List interface. It intends to offer you the possibility of retrieving a primitive type double directly from the underlying array of primitive doubles. No autoboxing.

```
List list = DoubleArrayList(); // backed by an array of doubles
list.add(1.0); // autoboxing :: list.add(new Double(1.0))
(DoubleArrayList)list.add(2.0); // no autoboxing
double d = (DoubleArrayList)list.getDouble(0);
```

Due to these improvements, only a few objects are created and no need for autoboxing.

For more info about fastutil, please have a look at
<http://fastutil.dsi.unimi.it/>

Math Packages

Colt was developed at CERN

- Stable and Reliable
- Optimised / High performance
- Production grade
- Vector and Matrix operations
- Linear Algebra
- Statistical methods
- Last release: v1.2, Sept/04



<http://acs.lbl.gov/~hoschek/colt/>

====Math Packages====

JQuantLib uses external libraries wherever possible.

In the specific case of mathematical and statistical stuff, JQuantLib uses Colt, which was developed at CERN and is used on high energy physics problems. Colt is optimised to be used in production, solving problems which involve thousands or even millions data points.

There are other packages around but Colt was selected by its completeness and high performance.

For more information about Colt, please have a look at the link shown below:

<http://acs.lbl.gov/~hoschek/colt/>

Parallelism

Levels of parallelism

- JQuantLib is thread-safe
- Parallel Colt takes advantage of multiple CPUs
- Customized JVMs

Parallel Colt

- The natural evolution for Colt
- Important factor for selecting Colt
- Still in development but no major issues
- Last release: v0.6.1, Dec. 2008

<http://piotr.wendykier.googlepages.com/parallelcolt>

====Paralellism====

Paralellism is a key factor which enables applications to perform well, taking full advantage of hardware resources, multiprocessors or even grid environments.

Parallelism can be obtained in several ways:

* Application level: JQuantLib is designed to be thread-safe, which potentially enables it to solve several different problems at the same time. No rocket science here: only what you would expect.

* JVM level: A customised JVM can take advantage of special hardware features in order to minimize gc latency and other bottlenecks. We will talk more about this item soon.

* Library level: Actually JQuantLib will use Parallel Colt and not Colt.

Parallel Colt is a a parallelised version of Colt, which takes advantage of multiprocessing. The existence of a parallel version of Colt was another reason for selecting Colt at first place.

In fact, JQuantLib was initially written for using Colt and focus changed to Parallel Colt by the time. Parallel Colt keeps compatibility with Colt APIs wherever possible. We plan to keep compatibility with both via proxies.

For more information about Parallel Colt, please have a look at

JVM and gc



JVMs need to be optimised for specific hardware

Some info about Azul appliances

- Getting rid of the JVM scaling issues
- Customised JVM
- Up to 54 cores per CPU
- Up to 16 processors, 860 cores, 768Gb
- Grid enabled: can scale even more
- Hardware assisted GC
- Hardware assisted Java locking
- Performance increase: order of hundred times

<http://www.azulsystems.com>

====JVM and gc====

JVMs need to be optimised for specific hardware in order to perform well.

A good example is how IBM JVM support BigDecimal on AIX boxes.

Another good example is Azul Systems appliances.

I'd like to mention something about Azul appliances as they are certainly very interesting for those willing to deploy applications written in Java onto critical, low latency environments.

* These appliances can have up to 54 cores per processors and reach 860 cores, 768Gb memory in a single box.

* A customized JVM offers low latency due to hardware assisted garbage collector and hardware assisted Java locking.

For more information about Azul Systems, please have a look at <http://www.azulsystems.com>

Profiling

- No memory profiling yet
- Performance tests are still incipient

====Profiling====

We cannot say we have enough focus on performance at the moment because our main focus is on translation. We have only some incipient performance tests.

In the following chart we compare performance of European Options calculation between QuantLib/C++ and JQuantLib/Java implementations.

In spite JQuantLib shows better performance... which is excellent!... JQuantLib is still in its childhood and we cannot say anything about performance yet.

JQuantLib

mean = 52.2 :: stddev = 72.4

QuantLib

mean = 68.8 :: stddev = 7.0

We can see that stddev from JQuantLib is much higher, mainly during start-up. Also, performance alternates highs and lows, for unknown reasons at this time.

Next releases

3rd release

- Date: 12th February 2009 / Eclipse Banking Day, London
- American Options with Finite Differences
- American Options with Integral Engine
- Asian Options
- Translation and tests of all 35 calendar classes , from 2004 to 2012

4th release - tbd

- Monte Carlo method
- Sobol (Quasi Monte Carlo, low-discrepancy numbers)
- Bonds

===Next Releases===

The 2nd Release , which is planned to happen late October/2008 will have support for American Options and Finite differences methods.

The 3rd Release, which is planned to happen late December/2008 will support Monte Carlo simulation and bonds.

Future

Pluggable OSGi blundles

- Purpose specific implementations
- Hot swap
- 24x7x365

Marketplace

- Products and services show case
- Cooperation
- Forum, wiki
- Room for inovation

Thanks :)

<http://www.jquantlib.org/>

Wer möchte mitarbeiten?

Thanks!

Please have a look at our website tomorrow for obtaining presentation notes.