

Slide 1 - JQuantLib

JQuantLib ist eine finanzmathematische Java Bibliothek.

Slide 2 – Quantitative Finance

Finanzmathematik, Synonyme

- Mathematical Finance
- Financial Engineering
- Finance Computing

Berechnen der Preise von Finanzinstrumenten

- Aktien
- Optionen
- Obligationen
- Futures
- Wechselkurse
- usw.

Anwendung von mathematischen Methoden und Konzepten

- Statistik
- Analysis
- Lineare Algebra
- Interpolationen
- Näherungen
- Wahrscheinlichkeitsrechnungen

Einige dieser Berechnungen brauchen viel Rechenzeit, speziell numerische Integration und Matrizenberechnungen (grosse Matrizen).

Für was wird die Finanzmathematik verwendet?

- Modellieren von Handelsstrategien
- Berechnen von optimalen Investment Portfolios in Abhängigkeit des eingegangenen Risikos
- Neue Modelle und Strategien mit historischen Daten testen
- Risikio management

Die Finanzmathematik wird für Java-Entwickler immer interessanter da Java in diesem Bereich immer mehr Anwendung findet. Die bevorzugte Sprache ist heute aber nach wie vor C++. Firmen die bereits Java anwenden: Goldman Sachs, CMC Markets, CS Group).

Slide 3 – Background

- JQuantLib ist keine Neuerfindung und basiert auf QuantLib, die in C++ implementiert ist

QuantLib

- Start im 2000
- Ca. 35 Entwickler
- Ca. 2 Millionen Codezeilen
- Kompiliert mit MSVC++ und GCC
- Aktuelle Version: 0.9.7 (November 2008)
- Portierungen verfügbar für:
 - C# (QLNet)
 - Python
 - Perl
 - Ruby
 - ...

Die meisten dieser Portierungen verwenden SWIG-Wrappers.

Slide 3 – The Need of JQuantLib

QuantLib wurde bis heute nicht zufriedenstellend mit Java integriert, zumindest was öffentlich verfügbare Bibliotheken und Schnittstellen betrifft.

Eine Evaluation im 2007 zeigte:

- SWIG Wrappers: kompliziert, falsche Resultate, unvollständige Integration
- JNI: Technisch sehr komplex, Anwender von JQuantLib oder QuantLib haben keine Zeit sich damit auseinanderzusetzen
- SWIG/JNI: Die Erweiterung und Anpassbarkeit ist nicht flexibel genug

Alternativen

- Integration über J2EE und CORBA: Performanceprobleme, kompliziert
- Übersetzung von QuantLib nach Java
 - Ca. 1300 Klassen
 - 2 Millionen Codezeilen

Entscheid

- Übersetzung, auch wenn das eine grosse Aufgabe ist und sehr komplex. Jedoch benötigen wir eine Bibliothek, die so funktioniert, wie es ein Java-Entwickler erwartet

Slide 5 – Objectives

- 100% Übersetzung
- Syntax und Semantik wie sie ein Java-Entwickler erwartet
- Top Code-Qualität, Korrektheit, sehr gut dokumentiert
- Verwendung von Java-Funktionalitäten (Sprache, Plattform, verfügbare APIs)

Slide 6 – Challenges

QuantLib basiert auf einem sehr guten Design, aber

- Das Objektmodell ist sehr komplex
- Templates werden z.T. so verwendet, dass sie in Java nicht 1:1 implementiert werden können
- Nicht in Java verfügbare Funktionen aus boost und stl

Weitere Herausforderungen

- Maximale Genauigkeit ohne Performanceverlust
- Strenge Typenprüfung während der Kompilierung (um Fehler während der Laufzeit zu verhindern)
- Relativ schlechte Performance von Java verglichen mit C++
- Performanceprobleme wegen der Objekterzeugung und den Operationen der GC

Slide 7 – Current Status

- Start der Evaluation: September 2007
- Start mit der Codierung: Januar 2008
- 10 aktive Entwickler
- Erster Release: Juni 2008
- Heute: 40% aller Klassen sind übersetzt

Slide 8 –Features

Der Grossteil der Funktionalität in JQuantLib ist deckungsgleich mit QuantLib

- Finanzinstrumente: Aktien, Optionen, Futures, Swaps, Swap-Optionen, Obligationen usw.
- Formeln / Numerische Methoden: Black Scholes, Binomial Model, Monte Carlo, low-discrepancy numbers (Sobol) und vieles mehr

Spezifische JquantLib-Funktionalitäten

- OSGi: Wichtig für die Hochverfügbarkeit und flexible Konfiguration im laufenden Betrieb
- Unterstützung für Parallelität. Wichtig für die Skalierung und Performance
- JQuantLib wird ‚Grid-enabled‘ sein

OSGi

- Funktionalitäten können während dem laufenden Betrieb hinzugefügt, aktualisiert oder gestoppt werden dank der Bundles (Modularität)
- Beispiel 1: Ein Anwender kann den Randomizer während dem Betrieb austauschen, ohne das System zu unterbrechen

- Beispiel 2: Switch zwischen primitiven Datentypen für Berechnungen und BigDecimal

Slide 9 – Architecture

- Build-System
- Quality Assurance
- Correctness
- Strenge Typenprüfung
- Performance

Bei finanzmathematischen Berechnungen kommt es letztlich auf zwei Dinge an:

- Performance
- Genauigkeit und Korrektheit der Berechnungen

Slide 10 – Build Environment

- Linux (Debian)
- Java 6
- Eclipse 3.3
- Umbrello (UML)
- SVN
- Maven als Build-Tool
 - Eclipse-Integration
 - Kann Ant-Skripte ausführen
- Continuum:
 - Für Continuous Integration
 - Gute Integration in Maven
- Archiva:
 - Für das Artifact Management
 - Integriert sich sehr gut in Continuum

Slide 11 – Quality Assurance

- JUnit4 und Cobertura (Code Testabdeckung) sind in Maven integriert. Reports werden von Cobertura generiert und über eine Website verfügbar gemacht
- PMD und FindBugs um die Verletzung von Programmierrichtlinien aufzuzeigen. In Eclipse integriert
 - Beispiel: Erkennen von der Verwendung von JCF Klassen anstatt fastutil (primitive Datentypen)
- Eclemma: Code coverage Analyse

Slide 12 – Correctness

- Strenge Typenprüfung (JSR-308)
- Verwendung von enums, generischen Typen und Annotationen
- JQuantLib muss die gleichen Resultate liefern wie QuantLib
- Floating Point Rundungsfehler

Slide 13 – Annotations

Strenge Typenprüfung kann mit Annotationen erreicht werden:

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

In diesem Beispiel werden zwei double-Variablen vertauscht weil keine semantische Prüfung während der Kompilation stattfindet.

Bemerkung: Selbst typedefs in C++ bieten nicht dieselbe strenge Typenprüfung an.

Wie bekommen wir die gewünschte Typenprüfung?

Slide 14 – JSR-308

Diesen Problem wird mit der JSR-308 für das JDK7 (ab Anfang 2010 verfügbar) gelöst (Typenprüfung während der Kompilierung):

```
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;  
  
Double result1 = calc(rate, time); // This call will pass  
  
// This call can give us a compiler error  
Double result2 = calc(time, rate);
```

Slide 15 – Accuracy

Hier sehen wir ein einfaches Beispiel einer mathematischen Ungenauigkeit die wegen floating point Rundungsfehlern entstehen:

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

```
Resultat: 0.30000000000000004
```

Das liegt nicht an der Programmiersprache sondern an der Art und Weise wie Computer floating point Daten handhaben.

Wichtig zu wissen: Je mehr Rechenoperationen durchgeführt werden, desto grösser wird der Fehler (Beispiel: Monte Carlo Simulationen)

Anforderungen

- Gleiche Resultate wie QuantLib
- So leichtgewichtig wie möglich
- So schnell wie möglich: Objekthallokationen und GC sollen die Berechnungen nicht beeinflussen

Lösung

- Wie bei QuantLib: Berechnung von € nach einer Folge von Berechnungen um die Grösse des Fehlers festzustellen. Dieser Ansatz erlaubt die Verwendung von primitiven Datentypen
- Benötigte Stellen in QuantLib (und daher auch in JQuantLib):
 - 10^{-15} – 10^{-6}

Slide 16 – Documentation

Die Originaldokumentation von QuantLib soll mit JavaDoc nachgebaut werden. Dazu gehören neben der üblichen Code-Dokumentation auch UML-Diagramme und Formeln

- UMLGraph: Geschrieben in Java, kann einfach in JavaDoc integriert werden
- LaTeXtaglet: Einbettung von mathematischen Formeln in JavaDoc basierend auf der Latex-Syntax (Beispiel)

Slide 17 – Performance

Performance ist ein sehr kritisches Bedürfnis für ein Finanzpaket.

Wir wollen hier zeigen, dass es mit Java heute möglich ist um hochperformante Finanzapplikationen mit niedriger Latenz zu entwickeln.

Es geht um verschiedene Technologien, Tools und Produkte die uns helfen diese hochgesteckten Ziele zu erreichen.

Themen:

- Vergleich mit C++
- Collections
- Mathematik APIs
- Parallelität
- JVM und GC
- Profiling

Slide 18 – Comparison with C++

Die Resultate stammen vom Dr. Dobbs Journal und vergleichen C++ mit Java 5 Performancemessungen sind abhängig von der verwendeten Software bzw. der JVM, der Hardware und wie gut die Tester die Programmiersprache kennen.

- Arithmetische 32/64bit Operationen: mehr oder weniger gleich schnell (am häufigsten und wichtigsten)
- Sort Algos / Matrizen: Java ist 2-3 x langsamer als C++. Gründe: Objekterstellung, GC, autoboxing. Optimierungsmöglichkeiten später
- Trigonometrische Berechnungen: Java ist klar langsamer als C++

Die JVM bietet keine ‚unsigned‘ integer Operationen. In den meisten Situationen ist das kein Problem. In Ausnahmefällen sind Zusatzberechnungen notwendig die dann die Performance negativ beeinflussen.

Slide 18 – Collections

Das JCF ist sehr nützlich. Leider ist die Standardimplementierung nicht für High-Perfomancesysteme ausgelegt. Jedes Element in einer Collection ist ein Objekt das erstellt, referenziert und wieder entfernt werden muss. Die alternative dazu ist die Verwendung von primitiven Typen.

JQuantLib verwendet dazu die API fastutil, die eine Implementation der JCF Interfaces anbietet, aber auf Arrays von primitiven Typen aufgebaut ist.

Beispiel:

```
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
```

Slide 19 – Math Package

Colt wurde am CERN entwickelt für die Anwendung in der Hochenergiephysik und für grosse Datenmengen (Arrays/Matrizen) optimiert.

- Sehr stabil und zuverlässig
- Ausgezeichnete Dokumentation
- Vector und Matrix-Operationen
- Lineare Algebra
- Statistische Methoden
- Letzter Release: v1.2, Sept. 04

Slide 20 – Parallelism

Parallelität ist die Grundvoraussetzung für gute Performance und nutzt die Ressourcen voll aus.

Parallelität kann so erreicht werden

- JQuantLib ist tread-safe
- JVM: Eine angepasste JVM kann die Hardware voll auslasten und die GC steuern
- APIs: Parallel Colt
 - Eine spezielle Version von Colt
 - JQuantLib wurde zuerst mit Colt implementiert. Nun sind wir daran, auf Parallel Colt umzustellen
 - Noch immer in Entwicklung, aber keine Probleme bekannt
 - Letzter Release: v0.6.1, Dez. 2008

Slide 21 – Java and GC

Die JVM muss für hochperformante Anwendungen angepasst werden.

Azul Systems: Java Applikationen in Umgebungen mit niedriger Latenz

- Bis zu 860 Kerne
- 768 Gb Memory

Slide 22 – Profiling

Die Codebasis heute erlaubt noch kein sinnvolles Profiling weil wir uns noch auf die Coderstellung konzentrieren. Die Optimierung folgt später.

„Make it run – optimize later“

Die hier gezeigte Exceltabelle (siehe Datei Quantlib vs. JQuantLib.xls) vergleicht die Performance von europäischen Optionspreisberechnungen zwischen QuantLib und JQuantLib.

Die Performance von JQuantLib ist beachtlich, jedoch ist das Resultat mit Vorsicht zu geniessen.

JQuantLib ist beim Start klar langsamer und mit weiteren Ausreissern zwischendurch. Wir vermuten, dass dies an Operationen der GC liegt.

Slide 23 – Next Releases

3. Release

- 12. Februar 2009 / Eclipse Banking Day, London
- American Options mit Finit Differences
- American Options mit Integral Engine
- Asian Options
- Aktualisierte Kalenderklassen (35) von 2004 bis 2012

4. Release

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

Slide 24 – Future

- OSGi
- Grid-enabled
- Abgleich mit QuantLib 0.9.7 (oder der dann aktuellen Version)
- Plug-ins/Add-ins für Excel, OpenOffice, Gnumeric, Quantrix

Markplatz

- Produkte / Service Fallstudien
- Zusammenarbeit
- Forum, Wiki
- Projektideen

Slide 24 – Thank you!

Wer möchte mitarbeiten?