Introduction to
Financial Computing
with Python

Matthieu Mariapragassam
Why coding seems so easy?

But is actually not

**Sprezzatura:**
« It’s an art that doesn’t seem to be an art » -
The Book of the Courtier
Plan

• Introduction to programming languages
• Where do researchers and industry stand
• Fenics and Quantlib Libraries for PDEs and Financial Maths
• Python
How one writes code

1. Maths problem
2. Look for a good method
3. Find an article
4. Code it
5. Face another problem

Repeat the process to tackle another problem.
How one writes code

• It’s easy to:
  • Get lost
  • And REINVENT the wheel!
The Usual Suspects

• MatLab (Octave, SciLab)
• R-project
• C++
• Java
• C#
• Python
• SAS
• S
Different types of coding

- Tests
- Prototyping
- Enhanced-Prototyping
- Production ready code
Different types of coding

• Tests:
  • Code as fast as possible and get a not too horrible result

• Matlab / Python / R / VBA
Different types of coding

• Prototyping:
  • Transform an idea to code fast
  • Reliable and accurate results
  • Performance not a concern

• Matlab/ Python/ C#/ Java / R
Different type of coding

• Enhanced-Prototyping:
  • Researchers belong here
  • Scalability becomes more important
  • Performance can be a concern, but is usually not

• Matlab / Python
Different type of coding

• Production ready code:
  • Industry belong here
  • Code is shared
  • Scalability is of paramount importance
  • Performance is a top concern
  • Long-term goals

• C++
  • Eventually C# / Java
Example:
Numerical Solution of a Partial Differential Equation

• We would like to:
  • Solve the Forward Kolmogorov PDE for the density of the Black model
  • Use the Finite Element method to handle efficiently Dirac delta initial value
  • Compute Black-Scholes prices with the solution
  • Be as efficient as possible
Example:
Numerical Solution of a Partial Differential Equation

- The PDE Solver in Matlab:
  - Black-box
  - Lack of flexibility
  - Don’t handle Dirac IV
  - It’s quite slow

- Re-implementing is the only way in Matlab
Example:
Numerical Solution of a Partial Differential Equation

• Approximate figure after implementation:
  • A 1D problem takes 1M
  • A 2D problem takes 10M
  • A 3D problem takes 1H
  • A 3D problem with good refinement takes 1D
Example:
Numerical Solution of a Partial Differential Equation

• Conclusion:
  • Acceptable for a 1D problem
  • No scalability
  • Lot of potential bugs

• What about external libraries?
Open Source External Libraries

• Pros
  • No need to re-invent the wheel
  • Robust
  • Plenty of features
  • Very fast and optimized when mature
  • Transparent
  • Scalable
  • Usually coded in C++ only
Open Source External Libraries

- Cons
  - Installing it can be tough
  - **Entry cost is usually very (very) high**
  - Documentation can be sparse
  - Debugging is harder
  - Building upon it requires to master it
  - Usually coded in C++
  - Code can be “over-engineered”
  - Getting lost in features
Open Source External Libraries: Fenics

“FEniCS has an extensive list of features for automated, efficient solution of differential equations, including automated solution of variational problems, automated error control and adaptivity, a comprehensive library of finite elements, high performance linear algebra and many more.”
Open Source External Libraries: Fenics

- Coded in C++
- Can handle up to 3 dimensions
- Specific BC or Subdomain equations
- Can handle very complex problems
- Can either be called from C++ or Python
Introduction to Python and Library Interfacing
Python

- Interpreted language
- Object-Oriented
- Syntax and Readability are key
- A “Real” programming language
- Fast when vectorized
  - Better than MatLab when not
- Slower than C++
- Packages are not as good as in Matlab
Python

• For Matrices, rely on NumPy:
Python

For numerical toolbox, rely on SciPy:

- **constants**: physical
- **cluster**: hierarchical clustering, vector quantization
- **fftpack**: Discrete Fourier Transform algorithms
- **integrate**: numerical integration routines
- **interpolate**: interpolation tools
- **io**: data input and output
- **lib**: Python wrappers to external libraries
- **linalg**: linear algebra routines
- **misc**: miscellaneous utilities (e.g. image reading/writing)
- **ndimage**: various functions for multi-dimensional image processing
- **optimize**: optimization algorithms including linear programming
- **optimize**: optimization algorithms
- **signal**: signal processing tools
- **sparse**: sparse matrix and related algorithms
- **spatial**: KD-trees, nearest neighbors, distance functions
- **special**: special functions
- **stats**: statistical functions
- **weave**: tool for writing C/C++ code
Driving Example:

• Pricing a Call Option under Black-Scholes with:
  • **Pure Python** libraries SciPy/NumPy and Integration of the payoff x density
  • **Use Fenics C++ Lib** to solve the Fokker-Plank equation and integrate payoff x density
  • **Use QuantLib C++ Lib** (SWIG) and price with Monte-Carlo
Driving Example:

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
<th>Standard Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call Price ClosedForm Density</td>
<td>10.5195410637</td>
<td>± 1.2039562355e-09</td>
</tr>
<tr>
<td>Call Price MonteCarlo</td>
<td>10.515670586</td>
<td>± 0.0915125502167</td>
</tr>
<tr>
<td>Call Price FEM Density</td>
<td>10.5165959904</td>
<td>± 1.35274640423e-07</td>
</tr>
<tr>
<td>Call Price Black Formula</td>
<td>10.5195410637</td>
<td></td>
</tr>
</tbody>
</table>
Fenics FEM Library

• Fenics use **UFL** to define a PDE under weak form

• Fokker-Planck equation of for the density of $S$ under Black:

$$\frac{\partial \phi}{\partial t} + (r - q) \frac{\partial x \phi}{\partial x} - \frac{1}{2} \frac{\partial^2 x^2 \sigma^2 \phi}{\partial x} = 0$$

• Initial value: $\phi(s, 0) = \delta(s = s_0)$

• Boundary conditions: $\phi(0, t) = \phi(\infty, t) = 0$
Fenics FEM Library

- The space $\Omega$ on which we want to solve the PDE is $\Omega = [0; \infty[$

- Let’s assume no rates or dividends. Let $v$ be a test function. The weak formulation of the PDE is:

\[
\int_{\Omega} \frac{\partial \phi}{\partial t} v d\Omega + \int_{\Omega} \frac{1}{2} x^2 \sigma^2 \frac{\partial \phi}{\partial x} \frac{\partial v}{\partial x} d\Omega + \int_{\Omega} \frac{1}{2} \frac{\partial x^2 \sigma^2}{\partial x} \phi \frac{\partial v}{\partial x} d\Omega = 0
\]

- The dirac delta IV condition becomes:

\[
\int_{\Omega} \phi(t = 0) v d\Omega = v(s0)
\]
Full implicit time scheme:

- We note $\phi^n = \phi(t_n)$
- $\int_\Omega \phi(t = 0) v d\Omega = v(s_0)$
- $\int_\Omega \phi^n v d\Omega + dt \left[ \int_\Omega \frac{1}{2} x^2 \sigma^2 \frac{\partial \phi^n}{\partial x} \frac{\partial v}{\partial x} d\Omega + \int_\Omega \frac{1}{2} \frac{\partial x^2 \sigma^2}{\partial x} \phi^n \frac{\partial v}{\partial x} d\Omega \right] = \int_\Omega \phi^{n-1} v d\Omega$
# Initial condition
u0 = Dirac()
u_1 = interpolate(u0, V)

vol = Expression('x[0]*sigma', sigma=sigma)
b0 = Expression('x[0]*sigma*sigma', sigma=sigma)
dt = T/nt

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)

a = u*v*dx + 0.5*vol*vol*dt*inner(nabla_grad(u), nabla_grad(v))*dx +
   b0*v.dx(0)*u*dt*dx
L = (u_1)*v*dx
Scipy Toolbox

• Exercise:
  • Price a Call Option with the Scipy toolbox
  • $s_0=100.; K=95; T=1.0; \text{vol} = 0.20$
Scipy Toolbox

Idea:

- Using **SciPy** Integration combined with the log-normal density
- Use the documentation tool of Spyder
- The density function \( f \) of \( S_T \) under the Black-Scholes model

\[
E \left[ (S_T - K)^+ \right] = \int_0^\infty x f(x) \, dx
\]

\[
\ln(S_t) = \ln(S_0) - \frac{\sigma^2 t}{2} + \sigma W_t
\]

Density is log-normal with scale factor \( e^{\ln(S_0) - \frac{\sigma^2 t}{2}} \) and shape factor \( \sigma \)
QuantLib FinEng Library

- It’s a full C++ solution
- Very over-engineered
  - It makes it’s beauty, for a geek...
- Entry cost is high, **very high**:
  - In Python it’s okay
- It’s a **powerfull toolbox**!
• Exercise:
  • Price a Call Option with the QuantLib Monte-Carlo Engine
  • Use Sobol sequences
  • $s_0=100.; K=95; T=1.0; \text{vol}=0.20$
QuantLib FinEng Library

• Steps:

1. Access the SWIG mappings between C++/Python:
   https://github.com/lballabio/quantlib/tree/master/QuantLib-SWIG/SWIG
2. Define the Vanilla contract
3. Define the Random Number Generator
4. Define the Path Generator
5. Write the Monte-Carlo Loop
6. Compute Mean and StdDev using Numpy

Use the debugger to check the values
QuantLib FinEng Library

Exercise 2:

- Define a StochasticProcess python class
- Define a LocalVolatilityProcess deriving from it
- Price an Up-and-Out Call option with QuantLib MC engine and compare LocalVol and Black-Scholes prices.

\[ s_0 = 100.;\ K = 80;\ B = 120;\ T = 1.0; \]

- volBS = 0.20
- localVol = \( ax^2 + bx + c \), such that \( c - \frac{b^2}{4a} = 0.18 \)
  - Extrapolated flat after ± 6 Black-Scholes stdDev
- Test limit cases to validate the pricer
Conclusion

• Python is the perfect **Glue**:
  • Interface C++ libraries
  • Use Native Python libraries
  • Identify bottlenecks and compile them in C
  • Best of both worlds!

Python is a real Programming OO language!
Open, Free and Multi-Platform!
THANKS

- Questions?