Patrick Hénaff

Topics in Empirical Finance

with R and Rmetrics

v 0.1

An Open Access Publication

Contents

Preface

This textbook is about empirical finance, and focusses on the pricing and risk management of financial assets: bonds, futures contracts, and other derivative securities.

The emphasis of this text is empirical. We present models, and verify their relevance by testing them of real data. We emphasize:

- an incremental approach to model building, starting from simple models, and building upon that foundation to construct more complex models, as needed,
- a data-driven approach: we will implement all the models that are presented, using the R statistical package and the Rmetrics libraries,
- the systematic use of simulation as a way of validating modeling decisions.

Last but not least, a particular attention is given to model estimation, in order to measure the tradeoff between model complexity and the challenges of a robust calibration.

This course would not be possible without the R statistical program and without the Rmetrics packages. We extend our deep appreciation to the R community and to Diethelm Wuertz and the Rmetrics team.

This book is open access (free as in free beer). It's also open source: feel free to clone and submit additions. You can download a PDF copy

Computational Framework

This first part is dedicated to the description of the experimental environment used in this book.

As mentioned earlier, our approach is data-driven and empirical. It is thus appropriate to start this text with a description of the sources of data that we will use, and how to fetch publicly available financial data from the Internet.

In the following chapters, we will make repeated use of the Rmetrics pricing libraries, and often compare different models applied to the same data. To facilitate this comparison, the Rmetrics pricing libraries have been wrapped in a object-oriented framework, which hides most of the implementation details, and allows us to focus on the key features of the financial instruments and models. The framework uses the S4 object model [@Genolini2008] of the R language, and is described in Chapter 1.

Simulation is our tool of choice to explore the features of pricing models and test their robustness. To that end, we have developped a framework that facilitates the definition and testing of simple risk management strategies. The core of this simulation framework is the `DataProvider` class, which is presented in Chapter 2.

In addition to the packages found on CRAN, data sets and code used in the text have been gathered into three packages:

empfin. contains all the data sets and several utility functions for simple bond pricing and the manipulation of dates,

fInstrument. provides the `fInstrument` class, that wraps the Rmetrics pricing library and the `DataProvider` class, that acts as a container for market data,

DynamicSimulation. contains tools needed to perform dynamic simulations, such as scenario generators and delta hedging simulators.

1 Basic Components

```
library(fOptions)
library(fExoticOptions)
library(fInstrument)
library(DynamicSimulation)
library(empfin)
library(plotly)
```

This chapter provides a tutorial and explains the design of the object framework that has been build on top of the Rmetrics library. As mentioned earlier, this layer is meant to hide most of the implementation details, and allows us to focus on the key features of the financial instruments and models.

The reader is encouraged to work through the examples, but the sections on design and implementation can be skipped.

The object framework involves two main entities:

- the `fInstrument` class models an abstract financial instrument, and exposes generic methods for computing the net present value (NPV) and the risk indicators (the "greeks' '). With this class, one can perform calculations on portfolio of instruments, without being concerned about the implementation details specific to each type of asset. This will be illustrated below with the detailed description of a delta-hedging algorithm.
- the `DataProvider` class is a container for market data, derived from the built-in R `environment`. It greatly simplifies the signature of pricing functions. Instead of passing as arguments all the necessary market data, we simply pass one `DataProvider` argument. The pricing methods fetch the necessary data from the `DataProvider`, as needed.

Each entity is now described and illustrated.

1.1 The `fInstrument` Class

As mentioned, the purpose of the `fInstrument` class is to create a layer of abstraction over the large variety of pricing models found in

Rmetrics. With this class, we can express calculation algorithms in a generic manner. This is best explained by an example.

1.1.1 Illustration

Consider a portfolio made of two options, a vanilla European call and a binary (cash-or-nothing) option, both written on the same underlying asset. We would like to compute the NPV and delta of this portfolio. Let's contrast the process, first performed with the Rmetrics functions, and then with the `fInstrument` class.

Starting with the Rmetrics functions, you first compute the price and delta of the European call:

```
p <- vector(mode = "numeric", length = 2)
d <- vector(mode = "numeric", length = 2)

p <- vector(mode = "numeric", length = 2)
cp <- "c"
Spot <- 100
Strike <- 100
Ttm <- 1
int.rate <- 0.02
div.yield <- 0.02
sigma <- 0.3
p[1] <- GBSOption(TypeFlag = cp, S = Spot, X = Strike,
    Time = Ttm, r = int.rate, b = int.rate - div.yield,
    sigma = sigma)@price
d[1] <- GBSGreeks(Selection = "delta", TypeFlag = cp,
    S = Spot, X = Strike, Time = Ttm, r = int.rate,
    b = int.rate - div.yield, sigma = sigma)
```

Perform the same calculation for the binary option. The delta is computed by finite difference.

```
p <- vector(mode = "numeric", length = 2)
K <- 1
p[2] <- CashOrNothingOption(TypeFlag = cp, S = Spot,
    X = Strike, K = K, Time = Ttm, r = int.rate, b = int.rate -
        div.yield, sigma = sigma)@price
h <- Spot * 0.001
dh <- CashOrNothingOption(TypeFlag = cp, S = c(Spot +
    h, Spot - h), X = Strike, K = K, Time = Ttm, r = int.rate,
    b = int.rate - div.yield, sigma = sigma)@price
d[2] <- diff(dh)/(2 * h)
```

Finally, sum both vectors to get the portfolio NPV and delta.

```
p <- vector(mode = "numeric", length = 2)
print(paste("Price:", round(sum(p), 2), "Delta:", round(sum(d),
    3)))
```

```
## [1] "Price: 0 Delta: 0.536"
```

With the `fInstrument` class, the calculation steps are quite different.

You first create a vanilla instrument with the `fInstrumentFactory` function:

```
dtExpiry <- myDate("01jan2011")
underlying <- "IBM"
Strike <- 100
K <- 1

b <- fInstrumentFactory("vanilla", quantity = 1, params = list(cp = "c",
    strike = Strike, dtExpiry = dtExpiry, underlying = underlying,
    discountRef = "USD.LIBOR", trace = FALSE))
```

Next, use again the `fInstrumentfactory` to create the binary option:

```
v <- fInstrumentFactory("binary", quantity = 1, params = list(cp = "c",
    strike = Strike, dtExpiry = dtExpiry, K = K, underlying = underlying,
    discountRef = "USD.LIBOR", trace = FALSE))
```

Insert the relevant market data into a `DataProvider` (this will be explained in the next section):

```
base.env <- DataProvider()
dtCalc <- myDate("01jan2010")
setData(base.env, underlying, "Price", dtCalc, 100)
setData(base.env, underlying, "DivYield", dtCalc, div.yield)
setData(base.env, underlying, "ATMVol", dtCalc, sigma)
setData(base.env, "USD.LIBOR", "Yield", dtCalc, int.rate)
```

Construct a portfolio, as a list of `fInstrument` objects:

```
portfolio = c(v, b)
```

and finally compute the price and delta of the portfolio:

```
price <- sum(sapply(portfolio, function(a) getValue(a,
    "Price", dtCalc, base.env)))
delta <- sum(sapply(portfolio, function(a) getValue(a,
    "Delta", dtCalc, base.env)))
print(paste("Price:", round(price, 2), "Delta:", round(delta,
    3)))
```

```
## [1] "Price: 13.31 Delta: 0.599"
```

1.1.2 *Design and Implementation*

`finstrument` objects are instantiated by the `fInstrumentfactory` class method, which takes as argument:

- the type of instrument being instantiated,
- the quantity or nominal amount of the instrument
- a list of parameters that define the instrument

The `fInstrumentFactory` method is simply a switch that delegates the object instantiation the the concrete subclasses of `fInstrument`. The following code fragment is extracted from the file `fInstrument.r` in package `fInstrument`:

```
fInstrumentFactory <- function(type, quantity, params) {
    switch(toupper(type), VANILLA = Vanilla(quantity,
        params), BINARY = Binary(quantity, params),
        ASIAN = Asian(quantity, params), STANDARDBARRIER = StandardBarrier(quantity,
            params))
}
```

There is only one method defined on `fInstrument` objects. This method is `getValue`, and takes as argument:

- the kind of calculation being requested (price, delta)
- the calculation date,
- the data container from which the required market data will be fetched.

Again, this method simply delegates to the concrete classes the requested calculation:

```
setMethod(f = "getValue", signature = signature("fInstrument"),
    definition = function(object, selection, dtCalc,
        env = NULL) {
        res <- NULL
        res <- switch(toupper(selection), PRICE = object@p(dtCalc,
            env), DELTA = object@d(dtCalc, env), GAMMA = object@g(dtCalc,
            env), VEGA = object@v(dtCalc, env))
        return(res * object@quantity)
    })
```

As an illustration, the price calculation for vanilla options is implemented as follows in `Vanilla.r`:

```
getP <- function(dtCalc, env) {
    Spot <- getData(env, Underlying, "Price", dtCalc)
    s <- getData(env, Underlying, "ATMVol", dtCalc)
    r <- getData(env, df, "Yield", dtCalc)
    b <- getData(env, Underlying, "DivYield", dtCalc)
    t <- tDiff(dtCalc, dtExpiry)
    if (trace) {
        print(paste("Calling GBSOption with Spot=",
            Spot, "Strike=", Strike, "t=", t, "r=",
            r, "b=", b, "sigma=", s))
    }
    GBSOption(TypeFlag = cp, S = Spot, X = Strike,
        Time = t, r = r, b = b, sigma = s)@price
}
```

The actual calculation being performed by the Rmetrics `GBSOption` function. The model can be easily extended to accommodate other instruments.

1.2 *The `DataProvider` Class*

The `DataProvider` class is a container of market data, from which the pricing algorithm will fetch the necessary market information, as illustrated in the code fragment above. We first describe the representation of market data, then the algorithm for searching data in a `DataProvider`.

1.2.1 *The Model for Market Data*

The model for market data is strongly inspired by @Fowler1996. To summarize, a piece of market data is modeled as an observed phenomenon on a financial instrument. Therefore, every market data item is identified by three attributes:

1. the financial instrument being observed (e.g. a stock)
2. the observed phenomenon (e.g. the implied volatility, or the price)
3. the observation date

In order to optimize storage, the data is stored in a hash table. The first two attributes are combined to created the key, and the data for all observation dates is stored as a time series, with one column for actual data, and many additional columns when performing a simulation.

1.2.2 *The Search Algorithm*

The `DataProvider` inherits from the built-in `environment` class. In particular, it inherits the parent/child relationship: if a `DataProvider` has a parent, the data not found in the child environment is fetched from the parent, if possible, or from the grand-parent, and so forth.

This is useful when performing a scenario analysis where only a few variables are modified: The data held constant is stored in the parent scenario, and the modified data is stored in the child scenario which is passed as argument. This scheme is illustrated by the following example.

Let's define a vanilla option:

```
dtExpiry <- myDate("01jan2011")
underlying <- "IBM"
K <- 100
a <- fInstrumentFactory("vanilla", quantity = 1, params = list(cp = "c",
    strike = K, dtExpiry = dtExpiry, underlying = underlying,
    discountRef = "USD.LIBOR", trace = FALSE))
```

and populate a DataProvider with the necessary market data:

```
base.env <- DataProvider()
dtCalc <- myDate("01jan2010")

setData(base.env, underlying, "Price", dtCalc, 100)
setData(base.env, underlying, "DivYield", dtCalc, 0.02)
setData(base.env, underlying, "ATMVol", dtCalc, 0.3)
setData(base.env, "USD.LIBOR", "Yield", dtCalc, 0.02)
```

The NPV of the derivative is obtained by:

```
getValue(a, "Price", dtCalc, base.env)
[1] 12.82158
```

Next, we investigate the relationship between the underlying price and the value of the derivative, by inserting a set of scenarios for the underlying asset price in a child `DataProvider`:

```
sce <- t(as.matrix(seq(80, 120, length.out = 30)))
sce.env <- DataProvider(parent = base.env)
setData(sce.env, underlying, "Price", dtCalc, sce)
```

and compute the NPV of the derivative for each scenario. The relationship between the underlying price and the value of the call is illustrated in figure 1.1.

```
p <- getValue(a, "Price", dtCalc, sce.env)
plot(sce, p, type = "l", lwd = 3, xlab = "Spot", ylab = "Price",
    bty = "n", col = "red")
```

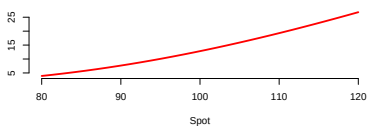

Figure 1.1: Call Price as a function of spot value. Strike: 100, maturity: 1 Year

2 *The Simulation Framework*

```
library(fInstrument)
library(empfin)
library(DynamicSimulation)
library(kableExtra)
```

The simulation framework has two main components:

1. A simulator, which can generate paths for the variables of interest (price of a asset, implied volatility, term structure of interest rate, etc.), according to a model that relates the variables to stochastic risk factors.

2. A framework for expressing trading strategies, and in particular dynamic hedging policies.

This is again best explained by an example, and we describe next the use of this framework for running a delta-hedging experiment. The main elements of the design are discussed afterwards.

2.1 Scenario Simulator

Assume that you want to simulate 500 price paths over a period of one year, with 100 time steps. The price process will be log-normal with annual volatility of 30%, starting from an initial value of $100.

```
dtStart <- myDate("01jan2010")
dtEnd <- myDate("01jan2011")
nbSteps <- 100
nbPaths <- 500
```

Next, define the sequence of simulation dates and the volatility of the simulated log-normal process:

```
dtSim <- seq(dtStart, dtEnd, length.out = nbSteps +
    1)
sigma <- 0.3
```

Use a sobol sequence as random number generator, with antithetic variates, standardized to unit variance:

```
tSpot <- pathSimulator(dtSim = dtSim, nbPaths = nbPaths,
    innovations.gen = sobolInnovations, path.gen = logNormal,
    path.param = list(mu = 0, sigma = sigma), S0 = 100,
    antithetic = F, standardization = TRUE, trace = FALSE)
print(head(tSpot[, 1:2]))
```

```
## GMT
##                          TS.1     TS.2
## 2010-01-01 00:00:00 100.0000 100.0000
## 2010-01-04 15:36:00 105.1741 100.4329
## 2010-01-08 07:12:00 106.1126 101.1069
## 2010-01-11 22:48:00 102.1089 102.3776
## 2010-01-15 14:24:00 101.5093 100.7461
## 2010-01-19 06:00:00 102.3337 104.7047
```

The output of the path simulator is a `timeSeries`. A plot of the first few paths is displayed in figure 2.1. The function can generate simulated values acording to various statistical processes; this is documented in the vignette of the package.

```
plot(tSpot[, 1:50], plot.type = "single", ylab = "Price",
    format = "%b %d")
```

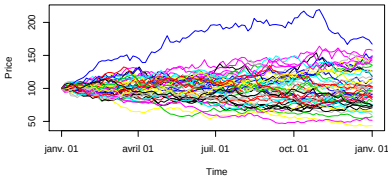

Figure 2.1: Simulated price paths under a log-normal diffusion process

2.2 *A Delta Hedging Experiment*

Having generated some scenarios for the stock price, let's now simulate the dynamic hedging of a European call option written on this stock, using the Black-Scholes pricing model, with the implied volatility and interest rate held constant.

Fist, we define the instrument to be hedged:

```
dtExpiry <- dtEnd

underlying <- "IBM"
K <- 100

a <- fInstrumentFactory("vanilla", quantity = 1, params = list(cp = "c",
    strike = K, dtExpiry = dtExpiry, underlying = underlying,
    discountRef = "USD.LIBOR", trace = FALSE))
```

Next, we define the market data that will be held constant during the simulation, and insert it in a `DataProvider`:

```
base.env <- DataProvider()
setData(base.env, underlying, "Price", dtStart, 100)
setData(base.env, underlying, "DivYield", dtStart,
    0.02)
setData(base.env, underlying, "ATMVol", dtStart, sigma)
setData(base.env, underlying, "discountRef", dtStart,
    "USD.LIBOR")
setData(base.env, "USD.LIBOR", "Yield", dtStart, 0.02)
```

At this stage, we can price the asset as of the start date of the simulation:

```
p <- getValue(a, "Price", dtStart, base.env)
```

which gives a value of $p = 12.82$.

Next, define the simulation parameters: we want to simulate a dynamic hedging policy over 500 paths, and 100 time steps per path:

We use a child `DataProvider` to store the simulated paths. Data not found in the child `DataProvider` will be searched for (and found) in the parent `base.env`.

```
sce.env <- DataProvider(parent = base.env)
setData(sce.env, underlying, "Price", time(tSpot),
    as.matrix(tSpot))
```

We can now run the delta-hedge strategy along each path:

```
assets = list(a)
res <- deltaHedge(assets, sce.env, params = list(dtSim = time(tSpot),
    transaction.cost = 0), trace = FALSE)
```

The result is a data structure that contains the residual wealth (hedging error) per scenario and time step. The distribution of hedging error at expiry is shown in Figure 2.2.

```
hist(tail(res$wealth, 1), 50, xlab = "Residual wealth at expiry",
    main = "")
```

Figure 2.2: Distribution of residual wealth at expiry: delta hedge of a 1 year call option

To better illustrate the hedging policy, let's run a toy example with few time steps. The function `deltaHedge()` produces a detailed log of the hedging policy, which is presented in Table 2.1. For each time step, the table show:

- The stock price,
- the option delta,
- the option value,

- the value of the replicating portfolio and the short bond position in that portfolio.

```
dtSim <- time(tSpot)[seq(1, dim(tSpot)[1], 10)]
res <- deltaHedge(assets, sce.env, params = list(dtSim = dtSim,
    transaction.cost = 0), trace = FALSE)
sim.table <- makeTable(1, res)
```

Table 2.1: Simulated value of a call option and its hedge portfolio over time

time	stock price	delta	option	bond pos	hedge port.
1	100.00000	0.5857659	12.8215847	-46.679318	12.8215847
2	97.00535	0.5393194	10.4505725	-42.091929	10.9758209
3	104.71501	0.6426285	14.2776535	-53.084419	15.0510232
4	105.29617	0.6505785	13.8692900	-53.933323	15.3199039
5	115.98565	0.7898772	20.7809251	-70.285023	22.1677804
6	94.43981	0.4536390	6.0882182	-38.211619	5.0101741
7	97.30819	0.4972939	6.5169379	-42.493719	6.2356476
8	87.15693	0.2364152	1.8346996	-19.619493	1.1030904
9	84.01967	0.1148928	0.5969689	-9.368302	0.3223531
10	80.58910	0.0136819	0.0356091	-1.195477	-0.0904753
11	74.32359	0.0000000	0.0000000	-0.178588	-0.1785880

2.2.1 *Design Considerations*

Two design features are worth mentioning.

The generation of the scenarios is independent from the expression of the dynamic trading strategies. Remember that every data element stored in a `DataProvider` is a `timeSeries`. Since all the calculations on `fInstrument` are vectorized, there is no difference between performing a calculation on a scalar, or performing a simulation on multiple scenarios.

The second aspect is the use of parent/child relationships among `DataProvider` objects. All the market data that is held constant in the simulation is stored in the parent `DataProvider`. The data that changes from simulation to simulation is stored in the child `DataProvider`, and this is the object that is passed to the simulator. When a piece of data is requested from the child `DataProvider`, the following logic is applied:

1. First look for the data in the current `DataProvider` (the object passed as argument to the simulator)
2. if not found, look for the data in the parent `DataProvider`.
3. and so forth: the logic is recursive.

This behavior is inherited from the built-in `environment`.