Miletus.jl

February 9, 2017

Contents

Part I

Introduction

Chapter 1

Introduction

1.1 Overview

Miletus is a financial contract definition, modeling language, and valuation framework written in Julia. The implementation of the contract definition language is based on papers by Peyton Jones and Eber [PJ&E2000],[PJ&E2003].

As originally conceived in the referenced papers, complex financial contracts can often be deconstructed into combinations of a few simple primitive components and operations. When viewed through the lens of functional programming, this basic set of primitive objects and operations form a set of ”combinators” that can be used in the construction of more complex financial constructs.

Miletus provides both basic the primitives for the construction of financial contract payoffs as well as a decoupled set of valuation model routines that can be applied to various combinations of contract primitives. In Miletus, these ”combinators” are implemented through the use of Julia's *user-defined types*, *generic programming*, and *multiple dispatch* capabilities.

Unlike some existing implementations of financial contract modeling environments (created in languages such as Haskell or OCaml) that rely heavily on pure functional programming for the contract definition language, but may then switch to a second language (e.g. C++, Java, APL) for implementation of valuation processes, Miletus leverages Julia's strong type system and multiple dispatch capabilities to both express these contract primitive constructs and provide for generation of efficient valuation code. As seen in other parts of the Julia ecosystem, Miletus solves the two language problem with regards to defining and modeling financial contracts.

Miletus provides functionality for teams across an organization, both front office and back office, to use a common language for structuring, valuing and managing complex financial instruments. Whether a sales team needs to structure new products, an operations team deploying infrastructure for batch processing, a risk management team must determine exposure of a firm's portfolio, or regulators are evaluating capital requirements for solvency, Miletus allows for everyone to use the same language effectively and efficiently.

1.2 Installation

Installation of Miletus can be performed by obtaining an installer from Julia Computing as part of your JuliaFin purchase.

Once installed, to load the Miletus package in your current Julia session, use the following command:

```
using Miletus
```

At this point you can use any of the primitive Miletus types for defining new contracts, constructing and manipulating either your own contracts or a set of pre-existing option contracts included with Miletus, as well as executing valuation operations against any combination of built-in and user-defined primitives that comprise your contract.

Part II

Tutorial

Chapter 2

Tutorial

2.1 Motivating example

In the example code below we show, without detailed explanation, how to construct and value a European call option on a single stock using combinations of the basic primitive types. Each of the primitive types and operations utilized will be explained in more detail in subsequent sections.

```
using Miletus
using Base.Dates
using Miletus.TermStructure
using Miletus.DayCounts
using BusinessDays

import Miletus: When, Give, Receive, Pay, Buy, Both, At, Either, Zero
import Miletus: YieldModel, maturitydate
```

Acquire the rights to a contract with 100 units

```
x = Receive(100)
```

```
Amount└─
100
```

Acquire the rights to a contract with 100 units as an obligation

```
x = Pay(100)
```

```
Give└─
Amount
  └─100
```

Acquire the rights to a contract with 100 USD as an obligation

```
x = Pay(100USD)
```

```
Give└─
Amount
  └─100USD
```

Construct an object containing the core properties of our stock model including the start price, yield curve and carry curve

```
s = SingleStock()
```

```
SingleStock
```

The functional definition for buying a stock at a given price

```
x = Both(s, Pay(100USD))
```

```
Both├─
SingleStock└─
Give
  └─Amount
    └─100USD
```

Calling the Buy method defined as in the previous operation

```
x = Buy(s, 100USD)
```

```
Both├─
SingleStock└─
Give
  └─Amount
    └─100USD
```

Defining the acquisition of rights to a contract on a given date

```
x = When(At(Date("2016-12-25")), Receive(100USD))
```

```
When├─
{==}│
 ├─DateObs│
 └─2016-12-25└─
Amount
  └─100USD
```

Constructing a zero coupon bond with a function having the same components as in the previous operation

```
z = ZCB(Date("2016-12-25"), 100USD)
```

```
When├─
{==}│
 ├─DateObs│
 └─2016-12-25└─
Amount
  └─100USD
```

One of the most basic of option structures, acquisition of either a stock or an empty contract having no rights and no obligations

```
x = Either(SingleStock(), Zero())
```

```
Either├─
SingleStock└─
Zero
```

Combining all of the above concepts into the definition of a European call option

```
x = When(At(Date("2016-12-25")), Either(Buy(SingleStock(), 100USD), Zero()))
```

```
When├─
{==}│
 ├─DateObs│
 └─2016-12-25└─
Either
  ├─Both
  │ ├─SingleStock
  │ └─Give
  │   └─Amount
  │     └─100USD
  └─Zero
```

Calling the functional form of a European Call option defined using the same components as in the previous operation

```
eucall = EuropeanCall(Date("2016-12-25"), SingleStock(), 100USD)
```

```
When├─
{==}│
 ├─DateObs│
 └─2016-12-25└─
Either
  ├─Both
  │ ├─SingleStock
  │ └─Give
  │   └─Amount
  │     └─100USD
  └─Zero
```

Construction of a Geometric Brownian Motion Model used for describing the price dynamics of a stock

```
gbmm = GeomBMModel(Date("2016-01-01"), 100.0USD, 0.1, 0.05, .15)
```

```
Miletus.GeomBMModel{Miletus.CoreModel{Miletus.Currency.CurrencyQuantity{Miletus.Currency.CurrencyUnit{:USD
    },Float64},Miletus.TermStructure.ConstantYieldCurve,Miletus.TermStructure.ConstantYieldCurve}}(
    Miletus.CoreModel{Miletus.Currency.CurrencyQuantity{Miletus.Currency.CurrencyUnit{:USD},Float64},
    Miletus.TermStructure.ConstantYieldCurve,Miletus.TermStructure.ConstantYieldCurve}(100.0USD,Miletus.
    TermStructure.ConstantYieldCurve(Miletus.DayCounts.Actual365(),0.1,:Continuous,-1,2016-01-01),Miletus
    .TermStructure.ConstantYieldCurve(Miletus.DayCounts.Actual365(),0.05,:Continuous,-1,2016-01-01))
    ,0.15)
```

Valuation of our European call option whose underlying stock model uses a Geometric Brownian Motion Model for its price dynamics

```
value(gbmm, eucall)
```

```
8.09128105913761USD
```

2.2 Building Contracts with Primitive and Derived Types

Most of the types defined in Miletus are built upon a small set abstract types (`Contract`, `Observable{T}`, `Process{T}`, `TermStruct`, `DayCount`, `AbstractModel`), and each of the primitive combinators described in the original PJ&E papers are implemented as a `typealias` of a set of Julia types having one of these abstract types as a super type.

Contract primitives

The set of `Contract` primitives includes the following types:

- `Zero()`
  - A "null" contract
- `Amount(o::Observable)`
  - Receive an amount of the observable object `o`
- `Scale(s::Observable, c::Contract)`
  - Scale the contract `c` by `s`
- `Both(c1::Contract, c2::Contract)`
  - Acquire both contracts `c1` and `c2`
  - This type corresponds to the `and` combinator in the PJ&E papers.
- `Either(c1::Contract, c2::Contract)`
  - Acquire either contract `c1` or `c2`
  - This type corresponds to the `or` combinator in the PJ&E papers.
- `Give(c::Contract)`
  - Take the opposite side of contract `c`
  - Acquires the rights to contract `c` as an obligation
- `Cond(p::Observable{Bool}, c1::Contract, c2::Contract)`
  - If expression `p` is true at the point of acquisition, then acquire contract `c1`, otherwise acquire contract `c2`
- `When(p::Observable{Bool}, c::Contract)`
  - Acquire the contract `c` at the point when observable quantity `p` becomes `true`.
- `Anytime(p::Observable{Bool}, c::Contract)`
  - May acquire the contract `c` at any point when observable quantity `p` is `true`.
- `Until(p::Observable{Bool}, c::Contract)`
  - A contract that acts like contract `c` until `p` is `true`, at which point the object is abandoned, and hence becomes worthless.

Primative Observables

Like `Contract`, `Observable{T}` is defined as an abstract type. Specific instances of an `Observable` type are objects, possibly time-varying, and possibly unknown at contracting time, for which a direct measurement can be made. Example observable quantities include date, price, temperature, population or other objects that can be objectively measured.

Built-in primitive `Observable` types include the following:

- `DateObs() <: Observable{Date}`
  - A singleton type representing the ”free” date observable
- `AcquisitionDateObs() <: Observable{Date}`
  - The acquisition date of the contract
- `ConstObs{T} <: Observable{T}`
  - A constant observable quantity
  - `ConstObs(x)` - Constructor function for a constant observable of value `x`

Derived Observables

Built-in derived observable types include the following:

- `AtObs(t::Date) <: Observable{Bool}`
  - `AtObs(t::Date) = LiftObs(==,DateObs(),ConstObs(t))`
  - An observable that is `true` when the date is `t`
  - This type of observable is used as part of the construction of the derived contract primitives `ZCB`, `WhenAt`, `Forward`, and `European`
- `BeforeObs(t::Date) <: Observable{Bool}`
  - `BeforeObs(t::Date) = LiftObs(<=,DateObs(),ConstObs(t))`
  - An observable that is `true` when the date is before or equal to `t`
  - This type of observable is useds as part of the construction of the derived contract primitives `AnytimeBefore` and `American`

Each of these derived `Observable` types makes use of a `LiftObs` operation.

`LiftObs` is defined as an `immutable` type whose type constructor applies a function to one or more existing `Observable` quantities to produce a new `Observable`.

2.3 Constructing Observables and Contracts

To provide an example of how one goes about using the above primitive and derived `Observable` types, let's return to one of the operations from the opening "Motivting Example" section. We will break apart each piece of the constructed zero coupon bond, to point out the specific `Contract` and `Observable` components utilized.

Defining the acquisition of rights to a contract on a given date

```
x = When(At(Date("2016-12-25")), Receive(100USD))
```

```
When├─
{==}│
 ├─DateObs│
 └─2016-12-25└─
Amount
  └─100USD
```

Constructing a zero coupon bond with a function having the same components as in the previous operation

```
z = ZCB(Date("2016-12-25"), 100USD)
```

```
When├─
{==}│
 ├─DateObs│
 └─2016-12-25└─
Amount
  └─100USD
```

The most basic primitives in the above zero coupon bond construction are the `Amount` primitive `Contract` type used for representing the value of 100, the `CurrencyUnit` and `CurrencyQuantity` types used when representing USD, and the `DateObs` primitive `Observable` type used for representing the a `Date`.

The expression `Receive(100USD)` creates a `Contract` object that provides acquisition rights to 100USD.

The expression `At(Date("2016-12-25"))` creates a new `LiftObs` observable object that is `true` when the current date in the valuation model is "2016-12-25". The implementation of the `At` observable type constructor includes the following operations:

```
typealias AtObs LiftObs{typeof(==),Tuple{DateObs,ConstObs{Date}},Bool}

AtObs(t::Date) = LiftObs(==,DateObs(),ConstObs(t))

typealias At AtObs
```

```
WARNING: Method definition (::Type{Miletus.LiftObs{Base.#==, Tuple{Miletus.DateObs, Miletus.ConstObs{Base.
     Dates.Date}}, Bool}})(Base.Dates.Date) in module Miletus at /Users/aviks/.julia/v0.5/Miletus/src/
     observables.jl:65 overwritten in module ex-lift at none:2.
```

The arguments to `LiftObs` in the definition of `AtObs` include:

- The == function that will be applied to two observable values on date quantities
- A `DateObs` object that acts as a reference observable quantity for the "Current Date" when valuing a model

- An input date `t` which becomes a constant observable quantity `ConstObs(t)` to which the reference observable is compared when valuing a contract.

The commands below show both the hierarchy of observables and the type of the result returned by a call to `At`.

```
At(Date("2016-12-25"))
```

```
{==}├─
DateObs└─
2016-12-25
```

```
typeof(At(Date("2016-12-25")))
```

```
Miletus.LiftObs{Base.#==,Tuple{Miletus.DateObs,Miletus.ConstObs{Date}},Bool}
```

With use of the `When` primitive `Contract`, the combination of our defined `Receive(100USD)` `Contract` object with the above `At(Date("2016-12-25"))` `Observable` object constructs new a zero coupon bond `Contract` that defines a payment of 100USD to the holder on December 25th, 2016.

The concept of optionality provides a contract acquirer with a choice on whether to exercise particular rights embedded in that contract. The most basic `Contract` primitives representing optionality in Miletus are the `Either` and `Cond` primitives described previously.

Adjusting the zero coupon bond example above to incorporate the `Either`, `Both` and `AtObs` `Contract` and `Observable` primitives allow for implementing a European Call option as repeated below.

```
x = When(At(Date("2016-12-25")), Either(Both(SingleStock(), Pay(100USD)), Zero()))
```

```
When├─
{==}│
 ├─DateObs│
 └─2016-12-25└─
Either
  ├─Both
  │ ├─SingleStock
  │ └─Give
  │   └─Amount
  │     └─100USD
  └─Zero
```

The above operations are defined as the typealias EuropeanCall

```
eucall = EuropeanCall(Date("2016-12-25"), SingleStock(), 100USD)
```

```
When├─
{==}│
 ├─DateObs│
 └─2016-12-25└─
Either
  ├─Both
  │ ├─SingleStock
  │ └─Give
  │   └─Amount
  │     └─100USD
  └─Zero
```

By combining various `Contract` and `Observable` primitives, contract payoffs of arbitrary complexity can be constructed easily.

The next section lists a number of built-in derived contracts that combine the above primitives in the defintion of various types of options instruments.

2.4 Built-in Derived Contracts

By combining these contract primitives, a set of `typealias` quantities are defined that allow for more compact syntax when creating various derived contracts. Using these type aliases, a set of constructors for these derived contracts are defined as shown below:

- `Receive(x::Union{Real,CurrencyQuantity}) = Amount(ConstObs(x))`
  - Receive an amount of a particular real valued object or currency
- `Pay(x::Union{Real,CurrencyQuantity}) = Give(Receive(x))`
  - Pay an amount of a particular real valued object or currency
- `Buy(c::Contract, x::Union{Real,CurrencyQuantity}) = Both(c, Pay(x))`
  - Purchase a contract `c` for an amount of a particular real valued object or currency
- `Sell(c::Contract, x::Union{Real,CurrencyQuantity}) = Both(Give(c), Receive(x))`
  - Sell a contract `c` for an amount of a particular real valued object or currency
- `ZCB(date::Date, x::Union{Real,CurrencyQuantity}) = When(AtObs(date), Receive(x))`
  - A "Zero Coupon Bond" that provides for obtaining a particular amount of a real valued object or currency on a particular maturity `date`
- `WhenAt(date::Date, c::Contract) = When(AtObs(date), c)`
  - Activate the contract `c` on the particular maturity `date`
- `Forward(date::Date, c::Contract, strike::Union{Real,CurrencyQuantity}) = WhenAt(date, Buy(c, strike))`
  - Purchase a contract `c` for a particular amount of a real valued object or currency (`strike`) on a particular maturity `date`
- `Option(c::Contract) = Either(c, Zero())`
  - Activate either contract `c` or nothing
- `European(date::Date, c::Contract) = WhenAt(date, Option(c))`
  - On a particular maturity `date` acquire either contract `c` or nothing
- `EuropeanCall(date::Date, c::Contract, strike::Union{Real,CurrencyQuantity}) = European(date, Buy(c, strike))`
  - A European call contract, with maturity `date`, on underlying contract `c` at price `strike`

- `EuropeanPut(date::Date, c::Contract, strike::Union{Real,CurrencyQuantity}) = European(date, Sell(c, strike))`
  - A European put contract, with maturity `date`, on underlying contract `c` at price `strike`
- `AnytimeBefore(date::Date, c::Contract) = Anytime(BeforeObs(date), c)`
  - Activate the contract `c` anytime before a particular maturity `date`
- `American(date::Date, c::Contract) = AnytimeBefore(date, Option(c))`
  - Either activate the contract `c` or nothing anytime before a particular maturity `date`
- `AmericanCall(date::Date, c::Contract, strike::Union{Real,CurrencyQuantity}) = American(date, Buy(c, strike))`
  - An American call contract, with maturity `date`, on underlying contract `c` at price `strike`
- `AmericanPut(date::Date, c::Contract, strike::Union{Real,CurrencyQuantity}) = American(date, Sell(c, strike))`
  - An American put contract, with maturity `date`, on underlying contract `c` at price `strike`
- `AsianFixedStrikeCall(dt::Date, c::Contract, period::Period, strike) = European(dt, Buy(MovingAveragePrice(c, period), strike))`
  - An Asian option contract where the strike price is constant and whose pay off is based on the moving average price of the underlying over the life of the contract.
- `AsianFloatingStrikeCall(dt::Date, c::Contract, period::Period, strike) = European(dt, Both(c, Give(MovingAveragePrice(c, period))))`
  - An Asian option contract where the strike price and payoff are based on the moving average price of the underlying over the life of the contract.

2.5 DayCounts

Miletus provides implementations of a number of separate calendar implementations that take into consideration day count conventions from different countries and financial organizations worldwide. Each day count type is an instance of an abstract `DayCount` type.

Specific `DayCount` instances present in Miletus include:

- Actual360 - Uses a coupon factor equal to the number of days between two dates in a Julian calendar divided by 360.
- Actual365 - Uses a coupon factor equal to the number of days between two dates in a Julian calendar divided by 365.
- BondThirty360 / USAThirty360 - Uses a coupon factor equal to the number of days between two dates assuming 30 days in any month and 360 days in a year. Used for the pricing of US Corporate Bonds and many US agency bond issues.
- EuroBondThirty360 / EuroThirty360 - Uses a coupon factor equal to the number of days between two dates assuming 30 days in any month and 360 dates in a year.

- ItalianThirty360
- ISMAActualActual
- ISDAActualActual
- AFBActualActual

For each `DayCount` type, the `yearfraction` function provides the fractional position within the associated year for a provided input date.

Modifying a particular date in the course of a calculation often needs to take into account the above `DayCount` convention, as well as a relevant holiday calendar. The `adjust` function takes into account holidays through functionality used from the `HolidayCalendar` included `BusinessDays` package.

2.6 Processes

In the context of the contract definition language implemented by Miletus, a `Process`, `p(t)`, is a mapping from time to a random variable of a particular type. Both `Contract` objects and `Observable` objects can be modeled as a `Process`. Like `Contract` and `Observable`, a `Process` is defined in Miletus as an abstract type, where subtypes of `Process` are implemented as immutable types.

The following `Process` types are available for operating on `Contract` and `Observable` objects

- `DateProcess()` - maps an `Observable` date to the given date.
- `ConstProcess(val::T)` - maps an `Observable` value to a constant value (`val::T`) for all times.
- `CondProcess(cond::Process{Bool}, a::Process{T}, b::Process{T})` - based on first `Process` boolean value, maps to one of two distinct `Process` values.

2.7 Term Structures

Term Structures provide a framework for representing how interest rates for a given set of modeling assumptions change through time.

- TermStruct - An abstract type that is a super type to all Term Structures implmented in Miletus
- YieldTermStructure - An abstract type that encompasses various interest rate term structure models
- VolatilityTermStructure - An abstract type that encompasses various volatility term structure models
- ConstantYieldCurve - A concrete type encompassing a constant interest rate model
- ConstantVolatilityCurve - A concrete type encompassing a constant volatility model
- compound_factor - Multiplicative factor using the frequency and method by accumulated interest is included in principle for the purporses of interest rate calculations
- discount_factor - Inverse of the above compound_factor
- implied_rate - Determination of the current interest rate implied from the compounding factor
- forward_rate - A rate of interest as implied by the current zero rate of a given YieldTermStructure for periods of time in the future.
- zero_rate - The implied spot interest rate for a given YieldTermStructure and time horizon
- par_rate - A coupon rate for which a bond price equals its nominal value

Chapter 3

Models

A valuation model encompasses both the analytical mathematical description of the dynamics involved in how an observable quantity changes through time, as well as a numerical method used for discretizing and solving those analytical equations.

There are a wide variety of different analytical models for describing the value dynamics of interest rates, stocks, bonds, credit instruments (e.g. mortgages, credit cards, other loans) and other securities. With regards to numerical methods, most techniques fall into one of four distinct categories; Analytical Methods (closed-form equations), Lattice Methods (e.g. trees), Monte Carlo Methods, and Partial Differential Equation solvers (e.g. finite difference, finite element).

The `Contract` and `Observable` primitives described previously are used for setting up payoffs that act as boundary conditions and final conditions on the use of a model to value an instrument.

3.1 Implemented Models and Valuation methods

- Core Model (objective assumptions underlying the model. everything except volatility. objective parameters that can be observed in the market)
- Core Forward Model
- Yield Curves and Dates
- Geometric Brownian Motion
  - `GeomBMModel(startdate, startprice, interestrate, carryrate, volatility)`
    - A model for a `SingleStock`, following a geometric Brownian motion that includes the following fields:
      - `startdate`
      - `startprice`: initial price at `startdate`
      - `interestrate`: risk free rate of return.
      - `carryrate`: the carry rate, i.e. the net return for holding the asset:
      - for stocks this is typically positive (i.e. dividends)
      - for commodities this is typically negative (i.e. cost-of-carry)
    - `volatility`:
      - The `interestrate`, `carryrate` and `volatility` are all specified on a continously compounded, Actual/365 basis.
      - The price is assumed to follow the PDE:

$dS_t = (\kappa - \sigma^2/2)S_t dt + \sigma S_t dW_t$ * where W_t is a Wiener process, and `κ = interestrate - carryrate`.

* Associated valuation routines make use of analytical methods for solving the Black-Scholes equation, or when determining implied volatitilies based on the Black-Scholes equation.

- Binomial Geometric Random Walk

  - `BinomialGeomRWModel(startdate, enddate, nsteps, S₀, Δt, iR, logu, logd, p, q)`

    * A model for a Binomial Geometric Random Walk (aka Binomial tree)
    * The valuation routines for binomial trees are initialized using the payoff condition of the associated contract at expiry(`enddate`) and subsequently work backward in time through the tree to determine the value of the contract at the initial time (`startdate`).
    * Includes the following fields (or the `log` of those values)
      - `startdate` : start date of process
      - `enddate` : end date of process
      - `nsteps` : number of steps in the tree
      - `S₀` : inital value
      - `Δt` : the time-difference between steps, typically `days(startdate - enddate) / (365*nsteps)`
      - `iR` : discount rate, `exp(-Δt*interestrate)`
      - `u` : scale factor for up
      - `d` : scale factor for down
      - `p` : up probability
      - `q` : down probability, `1-p`

```
Plot of the underlying stock price dynamics on the binomial tree.

![](../media/binomial_tree1.png)

* Cox-Ross-Rubenstein Model

   * Makes use of a risk-neutral valuation principle wherein the expected return from the traded security
     is the risk-free interest rate, and all future cash flows can be valued by discounting their
     respective cashflows at that risk-free interest rate.
   * Imposes the condition that d = 1/u
   * u = expσΔ(*t)
   * d = expσΔ(-*t)
   * p = (exp(rΔ*t)-d)/(u-d)
   * q = (u-exp(rΔ*t))/(u-d)
```

```
Plot of the underlying stock price dynamics on the binomial tree for the Cox-Ross-Rubenstein Model.

![](../media/binomial_tree2.png)

* Jarrow-Rudd Model

  * u = exp((rσ-^2/2)Δ*t + σΔ*t)
  * d = exp((rσ-^2/2)Δ*t - σΔ*t)
  * p = q = 0.5
  * NOTE: not risk-neutral
```

```
* Jarrow-Rudd Risk Neutral

  * u = exp((rσ-^2/2)Δ*t + σΔ*t)
  * d = exp((rσ-^2/2)Δ*t - σΔ*t)
  * p = (exp(rΔ*t)-d)/(u-d)
  * q = (u-exp(rΔ*t))/(u-d)

* Tian

  * u = 1/2*exp(rΔ*t)*v*(v+1+sqrt(v^2+2v-3)), where v = expσΔ(^2*t)
  * d = 1/2*exp(rΔ*t)*v*(v+1-sqrt(v^2+2v-3)), where v = expσΔ(^2*t)
  * p = (exp(rΔ*t)-d)/(u-d)
  * q = (u-exp(rΔ*t))/(u-d)
```

- Monte Carlo Model
  - montecarlo(m::GeomBMModel, dates, n)
    - Accepts a Geometrical Brownian Motion model of the underlying asset dynamics.
    - Samples `n` Monte Carlo paths of the model `m`, at time `dates`.
    - Returns a `MonteCarloModel`
  - MonteCarloModel(core, dates, paths)
    - A `MonteCarloModel` is a type that represents the result of a simulation of a series of asset prices and includes the following fields:
    - `core`: a reference `CoreModel`
    - `dates`: an `AbstractVector{Date}`
    - `paths`: a matrix of the scenario paths: the rows are the scenarios, and the columns are the values at each date in `dates`.
  - MonteCarloScenario(core, dates, path)
    - A `MonteCarloScenario` is a single simulation scenario of a `MonteCarloModel` and includes the following fields:
    - `core`: a reference `CoreModel`
    - `dates`: an `AbstractVector{Date}`
    - `paths`: an `AbstractVector` of the values at each date in `dates`.

3.2 Functions available for operating on a Model

- `value()`
- `valueAt()`
- `forwardprice()`
- `yearfraction()`
- `yearfractionto()`
- `numeraire()`
- `startdate()`
- `ivol()`

3.3 Implied Volatlity calculations

- SplineInterpolation
  - This is used to model interpolation between any two discrete points on a discrete convex curve. This implements double quadratic interpolation.

```
* `x`: An Array of the discrete values on the x axis
* `y`: An Array of the discrete values on the y axis
* `weights`: An Array of tuples of the weights of every quadratic curve modelled between two
    discrete points on the curve
```

- SplineVolatilityModel
  - `ivol(m::SplineInterpolation, c::European)`
    - Compute the implied Black-Scholes volatility of an option `c` under the SplineVolatilityModel `m`.
  - `fit(SplineVolatilityModel, mcore::CoreModel, contracts, prices)`
    - Fit a `SplineVolatilityModel` using from a collection of contracts (`contracts`) and their respective prices (`prices`), under the assumptions of `mcore`.
  - `fit_ivol(SplineVolatilityModel, mcore::CoreModel, contracts, ivols)`
    - Fit a `SplineVolatilityModel` using from a collection of contracts (`contracts`) and their respective implied volatilities (`ivols`), under the assumptions of `mcore`.
- GeomBMModel
  - `ivol(m::CoreModel, c::Contract, price)`
    - Compute the Black-Scholes implied volatility of contract `c` at `price`, under the assumptions of model `m` (ignoring the volatility value of `m`).
  - `fitvol(GeomBMModel, m::CoreModel, c::Contract, price)`
    - Fit a `GeomBMModel` using the implied volatility of `c` at `price`, using the parametersof the `CoreModel` `m`.
- SABRModel
  - `ivol(m::SABRModel, c::European)`
    - Compute the implied Black-Scholes volatility of an option `c` under the SABR model `m`.
  - `fit_ivol(SABRModel, mcore::CoreModel, contracts, ivols)`
    - Fit a `SABRModel` using from a collection of contracts (`contracts`) and their respective implied volatilities (`ivols`), under the assumptions of `mcore`.
  - `sabr_alpha(F, t, σATM, β, ν, ρ)` - Not currently exported
    - Compute the α parameter (initial volatility) for the SABR model from the Black-Scholes at-the-money volatility. * `F`: Forward price * `t`: time to maturity * `σATM`: Black-Scholes at-the-money volatility * β, ν, ρ: parameters from SABR model.

Part III

Examples

Chapter 4

Extended Examples

4.1 Spread Options

Below we define a set of various spread options that show how one can combine vanilla options into more complex payoffs.

```
using Miletus, Gadfly, Colors

import Miletus: Both, Give, Contract, WhenAt, value
import Base.strip
```

First define parameters for use in various contract and model definitions.

```
expirydate = Date("2016-12-25")
startdate  = Date("2016-12-1")
interestrate = 0.05
carryrate    = 0.1
volatility   = 0.15
K₁ = 98.0USD
K₂ = 100.0USD
K₃ = 102.0USD
L  = 11 # Layers in the binomial lattice / Number of time steps
```

Next we define a range of prices to use for plotting payoff curves.

```
price = K₁-1USD:0.1USD:K₃+1USD
```

Then we construct example analytical, binomial, and Monte Carlo test models that will be used when valuing the various vanilla and spread options at the defined start date.

```
gbmm = GeomBMModel(startdate, K₂, interestrate, carryrate, volatility)
crr  = CRRModel(startdate, expirydate, L, K₂, interestrate, carryrate, volatility)
mcm  = Miletus.montecarlo(gbmm, startdate:expirydate, 10_000)
```

Next let's define a function for calculating the payoff curve of each spread at expiry over a range of asset prices. This function assumes that the provided date is the expiry date of all components within the contract c.

Figure 4.1:

```
function payoff_curve(c, d::Date, prices)
    payoff = [value(GeomBMModel(d, x, 0.0, 0.0, 0.0), c) for x in prices]
    p = [x.val for x in payoff]
    r = [x.val for x in prices]
    return r, p
end
```

Next we will create a set of vanilla call and put options at separate strikes that will be used for construction of the different spread options. The included plots show the payoff of the option at the middle strike, K_2.

Vanilla Call Option

```
call₁ = EuropeanCall(expirydate, SingleStock(), K₁)
call₂ = EuropeanCall(expirydate, SingleStock(), K₂)
call₃ = EuropeanCall(expirydate, SingleStock(), K₃)
s₁,cp₁ = payoff_curve(call₁, expirydate, price)
s₂,cp₂ = payoff_curve(call₂, expirydate, price)
s₃,cp₃ = payoff_curve(call₃, expirydate, price)
plot(x = s₂, y = cp₂, Geom.line,
     Theme(default_color=colorant"blue", line_width = 1.0mm),
     Guide.title("Vanilla Call Payoff Curve at Expiry"),
     Guide.xlabel("Stock Price"), Guide.ylabel("Payoff"))
```

```
value(gbmm, call₂)
```

```
1.3688351717682052USD
```

```
value(crr, call₂)
```

```
1.4034866337776404USD
```

```
value(mcm, call₂)
```

```
1.3849767880288002USD
```

Vanilla Put Option

```
put₁ = EuropeanPut(expirydate, SingleStock(), K₁)
put₂ = EuropeanPut(expirydate, SingleStock(), K₂)
put₃ = EuropeanPut(expirydate, SingleStock(), K₃)
s₁,pp₁ = payoff_curve(put₁, expirydate, price)
s₂,pp₂ = payoff_curve(put₂, expirydate, price)
s₃,pp₃ = payoff_curve(put₃, expirydate, price)
plot(x = s₂, y = pp₂, Geom.line,
     Theme(default_color=colorant"blue", line_width = 1.0mm),
     Guide.title("Vanilla Put Payoff Curve at Expiry"),
     Guide.xlabel("Stock Price"), Guide.ylabel("Payoff"))
```

```
value(gbmm, put₂)
```

```
1.6959851162778619USD
```

```
value(crr, put₂)
```

```
1.7306365782873316USD
```

```
value(mcm, put₂)
```

```
1.6588648682783513USD
```

Now we will start to combine these vanilla calls and puts into various spread options.

Figure 4.2:

Butterfly Call Spread

```
# Buy two calls at the high and low strikes
# Sell two calls at the middle strike
function butterfly_call(expiry::Date, K₁, K₂, K₃)
    @assert K₁ < K₂ < K₃
    c₁ = EuropeanCall(expiry, SingleStock(), K₁)
    c₂ = EuropeanCall(expiry, SingleStock(), K₂)
    c₃ = EuropeanCall(expiry, SingleStock(), K₃)
    Both(Both(c₁,c₃), Give(Both(c₂,c₂)))
end

bfly₁ = butterfly_call(expirydate, K₁, K₂, K₃)

s,p_bfly₁ = payoff_curve(bfly₁, expirydate, price)
blk = colorant"black"
red = colorant"red"
grn = colorant"green"
blu = colorant"blue"
plot(layer( x=s ,y=p_bfly₁,Geom.line,Theme(default_color=blk,line_width=1.5mm)),
     layer( x=s₁,y=  cp₁  ,Geom.line,Theme(default_color=red,line_width=1.0mm)),
     layer( x=s₃,y=  cp₃  ,Geom.line,Theme(default_color=grn,line_width=1.0mm)),
     layer( x=s₂,y=-2cp₂  ,Geom.line,Theme(default_color=blu,line_width=1.0mm)),
     Guide.manual_color_key("",["Butterfly Call", "call₁", "call₃", "-2call₂"],
     ["black", "red", "green", "blue"]),
     Guide.title("Butterfly Call Payoff Curve at Expiry"),
```

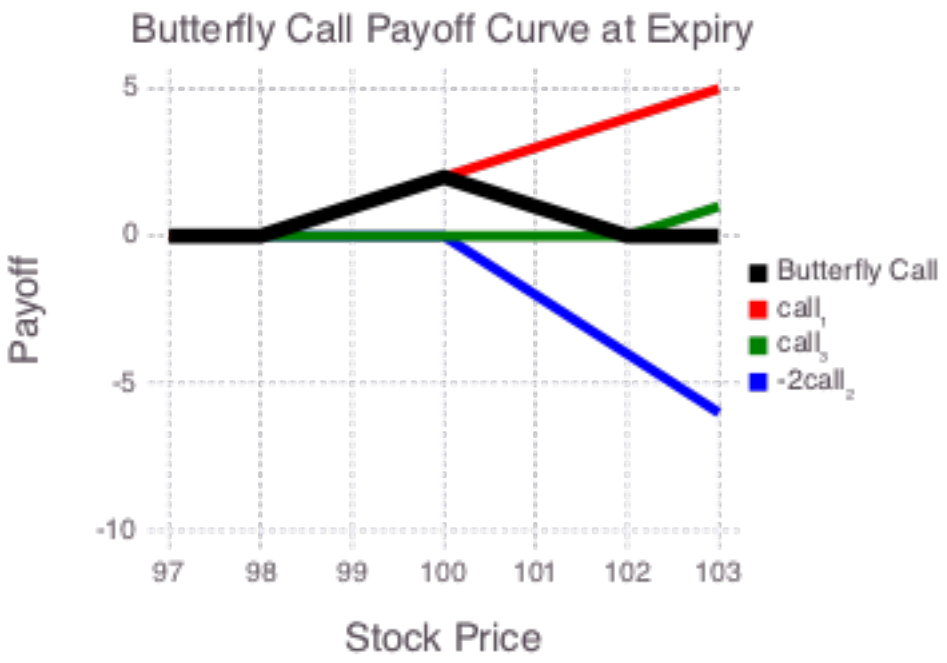

Figure 4.3:

```
    Guide.xlabel("Stock Price"), Guide.ylabel("Payoff"))
```

```
value(gbmm, bfly₁)
```

```
0.40245573232657295USD
```

```
value(crr, bfly₁)
```

```
0.3760697909383439USD
```

```
value(mcm, bfly₁)
```

```
0.40403957435376725USD
```

Butterfly Put Spread

```
# Buy two puts at the high and low strikes
# Sell two puts at the middle strike
function butterfly_put(expiry::Date, K₁, K₂, K₃)
    @assert K₁ < K₂ < K₃
    p₁ = EuropeanPut(expiry, SingleStock(), K₁)
```

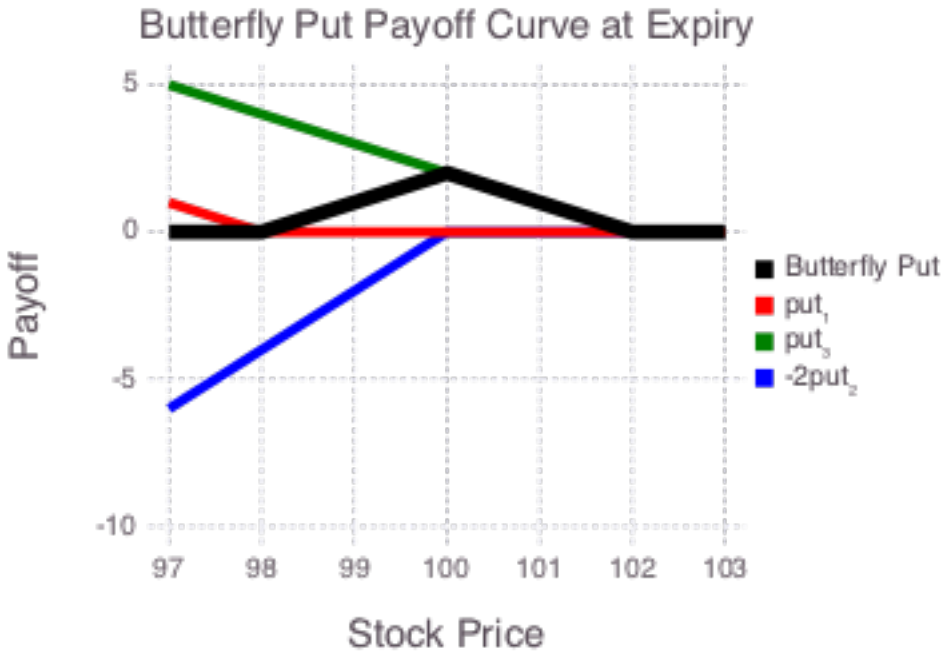

Figure 4.4:

```
    p₂ = EuropeanPut(expiry, SingleStock(), K₂)
    p₃ = EuropeanPut(expiry, SingleStock(), K₃)
    Both(Both(p₁,p₃), Give(Both(p₂,p₂)))
end

bfly₂ = butterfly_put(expirydate, K₁, K₂, K₃)
s,p_bfly₂ = payoff_curve(bfly₂, expirydate, price)
blk = colorant"black"
red = colorant"red"
grn = colorant"green"
blu = colorant"blue"
plot(layer( x=s ,y=p_bfly₂,Geom.line,Theme(default_color=blk,line_width=1.5mm)),
     layer( x=s₁,y=  pp₁  ,Geom.line,Theme(default_color=red,line_width=1.0mm)),
     layer( x=s₃,y=  pp₃  ,Geom.line,Theme(default_color=grn,line_width=1.0mm)),
     layer( x=s₂,y=-2pp₂  ,Geom.line,Theme(default_color=blu,line_width=1.0mm)),
     Guide.manual_color_key("",["Butterfly Put", "put₁", "put₃", "-2put₂"],
     ["black", "red", "green", "blue"]),
     Guide.title("Butterfly Put Payoff Curve at Expiry"),
     Guide.xlabel("Stock Price"), Guide.ylabel("Payoff"))
```

```
value(gbmm, bfly₂)
```

```
0.40245573232657295USD
```

```
value(crr, bfly₂)
```

```
0.37606979093834303USD
```

```
value(mcm, bfly₂)
```

```
0.40403957435377524USD
```

Bear Call Spread

```
# Buy a call at the high strike
# Sell a call at the low strike
function bear_call(expiry::Date, K₁, K₂)
    @assert K₁ != K₂
    c₁ = EuropeanCall(expiry, SingleStock(), K₁)
    c₂ = EuropeanCall(expiry, SingleStock(), K₂)
    Both(Give(c₁), c₂)
end

bear₁ = bear_call(expirydate, K₁, K₂)
s,p_bear₁ = payoff_curve(bear₁, expirydate, price)
blk = colorant"black"
red = colorant"red"
blu = colorant"blue"
plot(layer( x=s, y=p_bear₁,Geom.line,Theme(default_color=blk,line_width=1.5mm)),
     layer( x=s₁,y=-cp₁   ,Geom.line,Theme(default_color=red,line_width=1.0mm)),
     layer( x=s₂,y= cp₂   ,Geom.line,Theme(default_color=blu,line_width=1.0mm)),
     Guide.manual_color_key("",["Bear Call", "-call₁", "call₂"],
     ["black", "red", "blue"]),
     Guide.title("Bear Call Payoff Curve at Expiry"),
     Guide.xlabel("Stock Price"), Guide.ylabel("Payoff"))
```

```
value(gbmm, bear₁)
```

```
-1.1196804045372653USD
```

```
value(crr, bear₁)
```

```
-1.1071634323407784USD
```

```
value(mcm, bear₁)
```

```
-1.136626875669708USD
```

Figure 4.5:

Bear Put Spread

```
# Buy a put at the low strike
# Sell a put at the high strike
function bear_put(expiry::Date, K₁, K₂)
    @assert K₁ != K₂
    p₁ = EuropeanPut(expiry, SingleStock(), K₁)
    p₂ = EuropeanPut(expiry, SingleStock(), K₂)
    Both(p₁, Give(p₂))
end

bear₂ = bear_put(expirydate, K₁, K₂)
r,p_bear₂ = payoff_curve(bear₂, expirydate, price)
blk = colorant"black"
red = colorant"red"
blu = colorant"blue"
plot(layer( x=s,  y=p_bear₂,Geom.line,Theme(default_color=blk,line_width=1.5mm)),
     layer( x=s₁, y= pp₁   ,Geom.line,Theme(default_color=red,line_width=1.0mm)),
     layer( x=s₂, y=-pp₂   ,Geom.line,Theme(default_color=blu,line_width=1.0mm)),
     Guide.manual_color_key("",["Bear Put", "call₁", "-call₂"],
     ["black", "red", "blue"]),
     Guide.title("Bear Put Payoff Curve at Expiry"),
     Guide.xlabel("Stock Price"), Guide.ylabel("Payoff"))
```

```
value(gbmm, bear₂)
```

Figure 4.6:

```
-0.873755049943609USD
```

```
value(crr, bear₂)
```

```
-0.886272022140092USD
```

```
value(mcm, bear₂)
```

```
-0.8568085788110785USD
```

Bull Call Spread

```
# Buy a call at the low strike
# Sell a call at the high strike
function bull_call(expiry::Date, K₁, K₂)
    @assert K₁ != K₂
    c₁ = EuropeanCall(expiry, SingleStock(), K₁)
    c₂ = EuropeanCall(expiry, SingleStock(), K₂)
    Both(c₁, Give(c₂))
end

bull₁ = bull_call(expirydate, K₁, K₂)
r,p_bull₁ = payoff_curve(bull₁, expirydate, price)
```

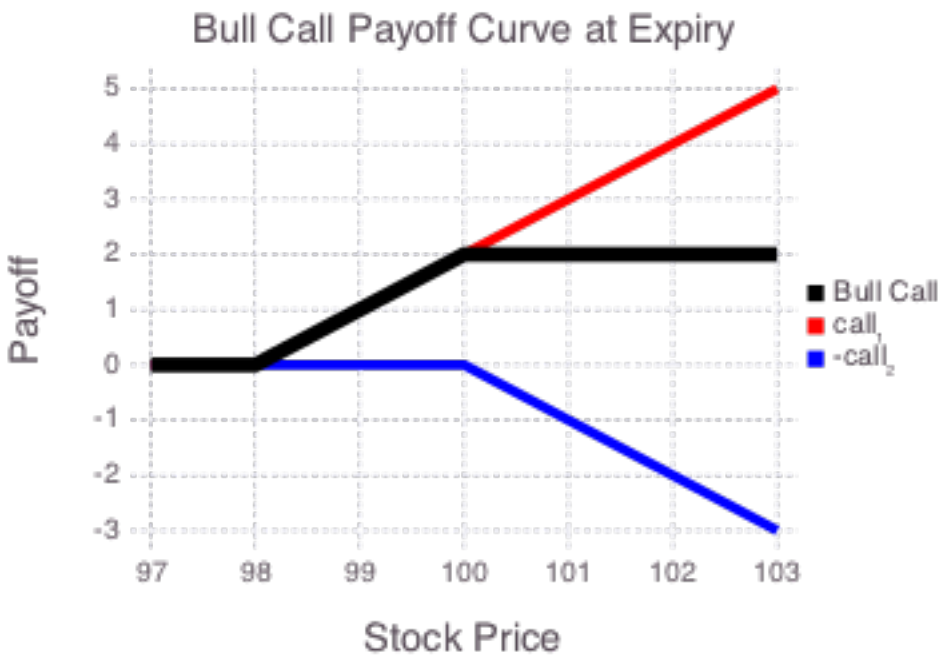

Figure 4.7:

```
blk = colorant"black"
red = colorant"red"
blu = colorant"blue"
plot(layer( x=s ,y=p_bull₁,Geom.line,Theme(default_color=blk,line_width=1.5mm)),
     layer( x=s₁,y= cp₁   ,Geom.line,Theme(default_color=red,line_width=1.0mm)),
     layer( x=s₂,y=-cp₂   ,Geom.line,Theme(default_color=blu,line_width=1.0mm)),
     Guide.manual_color_key("",["Bull Call", "call₁", "-call₂"],
     ["black", "red", "blue"]),
     Guide.title("Bull Call Payoff Curve at Expiry"),
     Guide.xlabel("Stock Price"), Guide.ylabel("Payoff"))
```

```
value(gbmm, bull₁)
```

```
1.1196804045372653USD
```

```
value(crr, bull₁)
```

```
1.1071634323407784USD
```

```
value(mcm, bull₁)
```

```
1.1366268756697908USD
```

Figure 4.8:

Bull Put Spread

```
# Buy a put at the high strike
# Sell a put at the low strike
function bull_put(expiry::Date, K₁, K₂)
    @assert K₁ != K₂
    p₁ = EuropeanPut(expiry, SingleStock(), K₁)
    p₂ = EuropeanPut(expiry, SingleStock(), K₂)
    Both(Give(p₁), p₂)
end

bull₂ = bull_put(expirydate, K₁, K₂)
r,p_bull₂ = payoff_curve(bull₂, expirydate, price)
blk = colorant"black"
red = colorant"red"
blu = colorant"blue"
plot(layer( x=s ,y=p_bull₂,Geom.line,Theme(default_color=blk,line_width=1.5mm)),
     layer( x=s₁,y=-pp₁   ,Geom.line,Theme(default_color=red,line_width=1.0mm)),
     layer( x=s₂,y= pp₂   ,Geom.line,Theme(default_color=blu,line_width=1.0mm)),
     Guide.manual_color_key("",["Bear Put", "-put₁", "put₂"],
     ["black", "red", "blue"]),
     Guide.title("Bear Put Payoff Curve at Expiry"),
     Guide.xlabel("Stock Price"), Guide.ylabel("Payoff"))
```

```
value(gbmm, bull₂)
```

```
0.873755049943609USD
```

```
value(crr, bull₂)
```

```
0.886272022140092USD
```

```
value(mcm, bull₂)
```

```
0.8568085788110785USD
```

Straddle Spread

```
# Buy a put and a call at the same strike
function straddle(expiry::Date, K)
    p = EuropeanPut(expiry, SingleStock(), K)
    c = EuropeanCall(expiry, SingleStock(), K)
    Both(p, c)
end

strd₁ = straddle(expirydate, K₂)
r,p_strd₁ = payoff_curve(strd₁, expirydate, price)
blk = colorant"black"
red = colorant"red"
blu = colorant"blue"
plot(layer( x=s ,y=p_strd₁,Geom.line,Theme(default_color=blk,line_width=1.5mm)),
     layer( x=s₁,y=cp₂    ,Geom.line,Theme(default_color=red,line_width=1.0mm)),
     layer( x=s₂,y=pp₂    ,Geom.line,Theme(default_color=blu,line_width=1.0mm)),
     Guide.manual_color_key("",["Straddle", "call₂", "put₂"],
     ["black", "red", "blue"]),
     Guide.title("Straddle Payoff Curve at Expiry"),
     Guide.xlabel("Stock Price"), Guide.ylabel("Payoff"))
```

```
value(gbmm, strd₁)
```

```
3.064820288046067USD
```

```
value(crr, strd₁)
```

```
3.134123212064972USD
```

```
value(mcm, strd₁)
```

```
3.0438416563071513USD
```

Figure 4.9:

Strip Spread

```
# Buy one call and two puts at the same strike
function strip(expiry::Date, K)
    p = EuropeanPut(expiry, SingleStock(), K)
    c = EuropeanCall(expiry, SingleStock(), K)
    Both(c, Both(p, p))
end

strip₁ = strip(expirydate, K₂)
r,p_strip = payoff_curve(strip₁, expirydate, price)
blk = colorant"black"
red = colorant"red"
blu = colorant"blue"
plot(layer( x=s ,y=p_strip,Geom.line,Theme(default_color=blk,line_width=1.5mm)),
     layer( x=s₁,y=cp₂    ,Geom.line,Theme(default_color=red,line_width=1.0mm)),
     layer( x=s₂,y=2pp₂   ,Geom.line,Theme(default_color=blu,line_width=1.0mm)),
     Guide.manual_color_key("",["Strip", "call₂", "2put₂"],
     ["black", "red", "blue"]),
     Guide.title("Strip Payoff Curve at Expiry"),
     Guide.xlabel("Stock Price"), Guide.ylabel("Payoff"))
```

```
value(gbmm, strip₁)
```

```
4.7608054043239285USD
```

Figure 4.10:

```
value(crr, strip₁)
```

```
4.864759790352304USD
```

```
value(mcm, strip₁)
```

```
4.702706524585503USD
```

Strap Spread

```
# Buy one put and two calls at the same strike
function strap(expiry::Date, K)
    p = EuropeanPut(expiry, SingleStock(), K)
    c = EuropeanCall(expiry, SingleStock(), K)
    Both(p, Both(c, c))
end

strap₁ = strap(expirydate, K₂)
r,p_strap = payoff_curve(strap₁, expirydate, price)
blk = colorant"black"
red = colorant"red"
blu = colorant"blue"
plot(layer( x=s ,y=p_strap,Geom.line,Theme(default_color=blk,line_width=1.5mm)),
     layer( x=s₁,y=2cp₂   ,Geom.line,Theme(default_color=red,line_width=1.0mm)),
```

Figure 4.11:

```
    layer( x=s₂,y=pp₂    ,Geom.line,Theme(default_color=blu,line_width=1.0mm)),
    Guide.manual_color_key("",["Strap", "2call₂", "put₂"],
    ["black", "red", "blue"]),
    Guide.title("Strap Payoff Curve at Expiry"),
    Guide.xlabel("Stock Price"), Guide.ylabel("Payoff"))
```

```
value(gbmm, strap₁)
```

```
4.433655459814272USD
```

```
value(crr, strap₁)
```

```
4.537609845842613USD
```

```
value(mcm, strap₁)
```

```
4.428818444335952USD
```

Figure 4.12:

Strangle Spread

```
# Buy a put at the low strike and a call at the high strike
function strangle(expiry::Date, K₁, K₂)
    p = EuropeanPut(expiry, SingleStock(), K₁)
    c = EuropeanCall(expiry, SingleStock(), K₂)
    Both(p, c)
end

strangle₁ = strangle(expirydate, K₁, K₃)
r,p_strangle = payoff_curve(strangle₁, expirydate, price)
blk = colorant"black"
red = colorant"red"
blu = colorant"blue"
plot(layer( x=s ,y=p_strangle,Geom.line,Theme(default_color=blk,line_width=1.5mm)),
     layer( x=s₁,y=cp₃       ,Geom.line,Theme(default_color=red,line_width=1.0mm)),
     layer( x=s₂,y=pp₁       ,Geom.line,Theme(default_color=blu,line_width=1.0mm)),
     Guide.manual_color_key("",["Strangle", "call₃", "put₁"],
     ["black", "red", "blue"]),
     Guide.title("Strangle Payoff Curve at Expiry"),
     Guide.xlabel("Stock Price"), Guide.ylabel("Payoff"))
```

```
value(gbmm, strangle₁)
```

```
1.473840565891766USD
```

```
value(crr, strangle,)
```

```
1.5167575485224454USD
```

```
value(mcm, strangle,)
```

```
1.4544457761800493USD
```

4.2 Coupon Bearing Bonds

Unlike a zero coupon bond, a coupon bearing bond pays the holder a specified amount at regular intervals up to the maturity date of the bond. These coupon payments, and the interest that can accumulate on those payments must be taken into account when pricing the coupon bond. The structuring of a coupon bond with Miletus provides an example of how to construct a product with multiple observation dates.

```
using Miletus, BusinessDays
using Miletus.TermStructure
using Miletus.DayCounts

import Miletus: Both, Receive, Contract, When, AtObs, value
import Miletus: YieldModel
import BusinessDays: USGovernmentBond
import Base.Dates: today, days, Day, Year
```

First let's show an example of the creation of a zero coupon bond. For this type of bond a payment of the par amount occurs only on the maturity date.

```
zcb = When(AtObs(today()+Day(360)), Receive(100USD))
```

```
When├─
{==}│
 ├─DateObs│
 └─2018-02-04└─
Amount
  └─100USD
```

Next let's define a function for our coupon bearing bond. The definition of multiple coupon payments and the final par payment involves a nested set of `Both` types, with each individual payment constructed from a `When` of an date observation and a payment contract.

```
function couponbond(par,coupon,periods::Int,start::Date,expiry::Date)
    duration = expiry - start
    bond = When(AtObs(expiry), Receive(par))
    for p = periods-1:-1:1
        coupondate = start + duration*p/periods
        bond = Both(bond,When(AtObs(coupondate), Receive(coupon)))
    end
    return bond
end
```

```
couponbond (generic function with 1 method)
```

To construct an individual coupon bond, we first define necessary parameters for the par, coupon, number of periods, start date and expiry date.

```
par = 100USD
coupon = 1USD
periods = 12
startdate = today()
expirydate = today() + Day(360)
```

```
2018-02-04
```

Now we can construct an instance of a coupon bearing bond.

```
cpb = couponbond(par,coupon,periods,startdate,expirydate)
```

```
Both├─
Both│
 ├─Both│
 │ ├─Both│
 │ │ ├─Both│
 │ │ │ ├─Both│
 │ │ │ │ ├─Both│
 │ │ │ │ │ ├─Both│
 │ │ │ │ │ │ ├─Both│
 │ │ │ │ │ │ │ ├─Both│
 │ │ │ │ │ │ │ │ ├─When│
 │ │ │ │ │ │ │ │ │ ├─{==}│
 │ │ │ │ │ │ │ │ │ │ ├─DateObs│
 │ │ │ │ │ │ │ │ │ │ └─2018-02-04│
 │ │ │ │ │ │ │ │ │ └─Amount│
 │ │ │ │ │ │ │ │ │   └─100USD│
 │ │ │ │ │ │ │ │ └─When│
 │ │ │ │ │ │ │ │   ├─{==}│
 │ │ │ │ │ │ │ │   │ ├─DateObs│
 │ │ │ │ │ │ │ │   │ └─2018-01-05│
 │ │ │ │ │ │ │ │   └─Amount│
 │ │ │ │ │ │ │ │     └─1USD│
 │ │ │ │ │ │ │ └─When│
 │ │ │ │ │ │ │   ├─{==}│
 │ │ │ │ │ │ │   │ ├─DateObs│
 │ │ │ │ │ │ │   │ └─2017-12-06│
 │ │ │ │ │ │ │   └─Amount│
 │ │ │ │ │ │ │     └─1USD│
 │ │ │ │ │ │ └─When│
 │ │ │ │ │ │   ├─{==}│
 │ │ │ │ │ │   │ ├─DateObs│
 │ │ │ │ │ │   │ └─2017-11-06│
 │ │ │ │ │ │   └─Amount│
 │ │ │ │ │ │     └─1USD│
 │ │ │ │ │ └─When│
 │ │ │ │ │   ├─{==}│
```

```
 │ │ │ │ │ │   │ ├─DateObs│
 │ │ │ │ │ │   │ └─2017-10-07│
 │ │ │ │ │ │   └─Amount│
 │ │ │ │ │ │     └─1USD│
 │ │ │ │ │ └─When│
 │ │ │ │ │   ├─{==}│
 │ │ │ │ │   │ ├─DateObs│
 │ │ │ │ │   │ └─2017-09-07│
 │ │ │ │ │   └─Amount│
 │ │ │ │ │     └─1USD│
 │ │ │ │ └─When│
 │ │ │ │   ├─{==}│
 │ │ │ │   │ ├─DateObs│
 │ │ │ │   │ └─2017-08-08│
 │ │ │ │   └─Amount│
 │ │ │ │     └─1USD│
 │ │ │ └─When│
 │ │ │   ├─{==}│
 │ │ │   │ ├─DateObs│
 │ │ │   │ └─2017-07-09│
 │ │ │   └─Amount│
 │ │ │     └─1USD│
 │ │ └─When│
 │ │   ├─{==}│
 │ │   │ ├─DateObs│
 │ │   │ └─2017-06-09│
 │ │   └─Amount│
 │ │     └─1USD│
 │ └─When│
 │   ├─{==}│
 │   │ ├─DateObs│
 │   │ └─2017-05-10│
 │   └─Amount│
 │     └─1USD│
 └─When│
   ├─{==}│
   │ ├─DateObs│
   │ └─2017-04-10│
   └─Amount│
     └─1USD└─
When
  ├─{==}
  │ ├─DateObs
  │ └─2017-03-11
  └─Amount
    └─1USD
```

Finally we can value this bond by constructing a yield curve and associated yield model and operating on the coupon bond contract with the defined yield model.

```
yc = ConstantYieldCurve(Actual360(), .1, :Continuous, :NoFrequency, Dates.today())
```

```
Miletus.TermStructure.ConstantYieldCurve(Miletus.DayCounts.Actual360(),0.1,:Continuous,-1,2017-02-09)
```

```
ym = YieldModel(yc, ModFollowing(), USGovernmentBond())
```

```
Miletus.YieldModel{Miletus.TermStructure.ConstantYieldCurve,Miletus.DayCounts.ModFollowing,BusinessDays.
    USGovernmentBond}(Miletus.TermStructure.ConstantYieldCurve(Miletus.DayCounts.Actual360(),0.1,:
    Continuous,-1,2017-02-09),Miletus.DayCounts.ModFollowing(),BusinessDays.USGovernmentBond())
```

```
value(ym,cpb)
```

```
100.92417167207167USD
```

4.3 Asian Option pricing

Asian options are structures whose payoff depends on the average price of an underlying security over a specific period of time, not just the price of the underlying at maturity. To price an Asian option, we will make use of a Monte Carlo pricing model, as well as a contract that considers a `MovingAveragePrice`

```
using Miletus
using Gadfly
using Colors

d1 = Dates.today()
d2 = d1 + Dates.Day(120)
```

```
2017-06-09
```

Structing the model without currency units

```
m = GeomBMModel(d1, 100.00, 0.05, 0.0, 0.3)
mcm = montecarlo(m, d1:d2, 100_000)
```

```
Miletus.MonteCarloModel{Miletus.CoreModel{Float64,Miletus.TermStructure.ConstantYieldCurve,Miletus.
    TermStructure.ConstantYieldCurve},StepRange{Date,Base.Dates.Day},Float64}(Miletus.CoreModel{Float64,
    Miletus.TermStructure.ConstantYieldCurve,Miletus.TermStructure.ConstantYieldCurve}(100.0,Miletus.
    TermStructure.ConstantYieldCurve(Miletus.DayCounts.Actual365(),0.05,:Continuous,-1,2017-02-09),
    Miletus.TermStructure.ConstantYieldCurve(Miletus.DayCounts.Actual365(),0.0,:Continuous,-1,2017-02-09)
    ),2017-02-09:1 day:2017-06-09,[100.0 100.665 … 118.557 119.593; 100.0 99.409 … 88.9926 89.6832; … ;
    100.0 100.144 … 84.2522 85.7512; 100.0 101.474 … 148.781 146.818])
```

We can view the underlying simulation paths used for our Geometric Brownian Motion Model using Gadfly as follows:

```
theme=Theme(default_color=Colors.RGBA{Float32}(0.1, 0.1, 0.7, 0.1))
p = plot([layer(x=mcm.dates,y=mcm.paths[i,:],Geom.line,theme) for i = 1:200]...)
```

Now let's value a vanilla European call option using a Geometric Brownian Motion Model.

```
o = EuropeanCall(d2, SingleStock(), 100.00)
value(m, o)
```

```
7.644207157741412
```

And value that same vanilla European call using a Monte Carlo Model

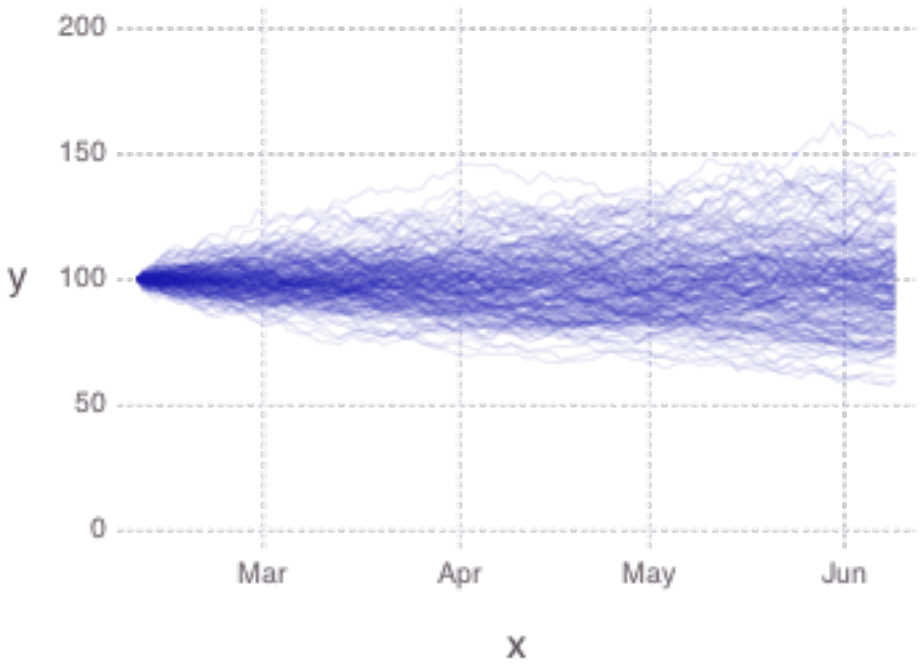

Figure 4.13:

```
value(mcm, o)
```

```
7.611532298314782
```

Next we construct a fixed strike Asian Call option. Note the `MovingAveragePrice` embedded in the definition.

```
oa1 = AsianFixedStrikeCall(d2, SingleStock(), Dates.Month(1), 100.00)
```

```
When├─
{==}│
 ├─DateObs│
 └─2017-06-09└─
Either
  ├─Both
  │ ├─MovingAveragePrice
  │ │ ├─SingleStock
  │ │ └─1 month
  │ └─Give
  │   └─Amount
  │     └─100.0
  └─Zero
```

```
value(mcm, oa1)
```

```
6.434610669797414
```

Similarly, we can construct a floating strike Asian Call option.

```
oa2 = AsianFloatingStrikeCall(d2, SingleStock(), Dates.Month(1), 100.00)
```

```
When├─
{==}│
 ├─DateObs│
 └─2017-06-09└─
Either
  ├─Both
  │ ├─SingleStock
  │ └─Give
  │   └─MovingAveragePrice
  │     ├─SingleStock
  │     └─1 month
  └─Zero
```

```
value(mcm, oa2)
```

```
3.690794638247629
```