List of Files

```julia
using Printf
include("src/printmat.jl");
```

```julia
files = filter(x -> endswith(lowercase(x),".ipynb"), readdir())  #all .ipynb files
filter!(x -> x != "Ch00_ListOfFiles.ipynb",files)

printmat(files)
```

```
Ch14_FX.ipynb
Ch15_Forwards.ipynb
Ch16_Bonds1.ipynb
Ch17_Bonds2.ipynb
Ch18_Bonds3.ipynb
Ch19_Options1.ipynb
Ch20_Options2.ipynb
Ch21_Options3.ipynb
Ch22_Options4.ipynb
```

FX

This notebook illustrates some basic aspects of FX pricing, for instance, the definition of currency returns and the idea of UIP. It also implements a carry trade strategy.

Load Packages and Extra Functions

```julia
using Printf, Statistics

include("src/printmat.jl");
```

```julia
using Plots
default(size = (480,320),fmt = :png)
```

Currency Returns

There are different ways to quote exchange rates, but this notebook uses S to denote the number of domestic currency units (say, CHFs if you are a Swiss investor) to buy one unit of foreign currency (say, one USD). That is, we treat foreign currency as any other asset.

The next cell calculates the return of the following simple strategy: - in $t = 0$: buy foreign currency (at the price S_0) and lend it on foreign money market (at the safe rate R_f^*). - in $t = 1$: sell the foreign currency (at the price S_1)

Since the strategy is financed by borrowing on the domestic money market (at the rate R_f), the excess return is

$R^e = (1 + R_f^*)S_1/S_0 - (1 + R_f)$

Notice that R_f and R_f^* are the safe rates over the investment period (for instance, one-month period). Conversion from annualized interest rates to these monthly rates is discussed under UIP (below).

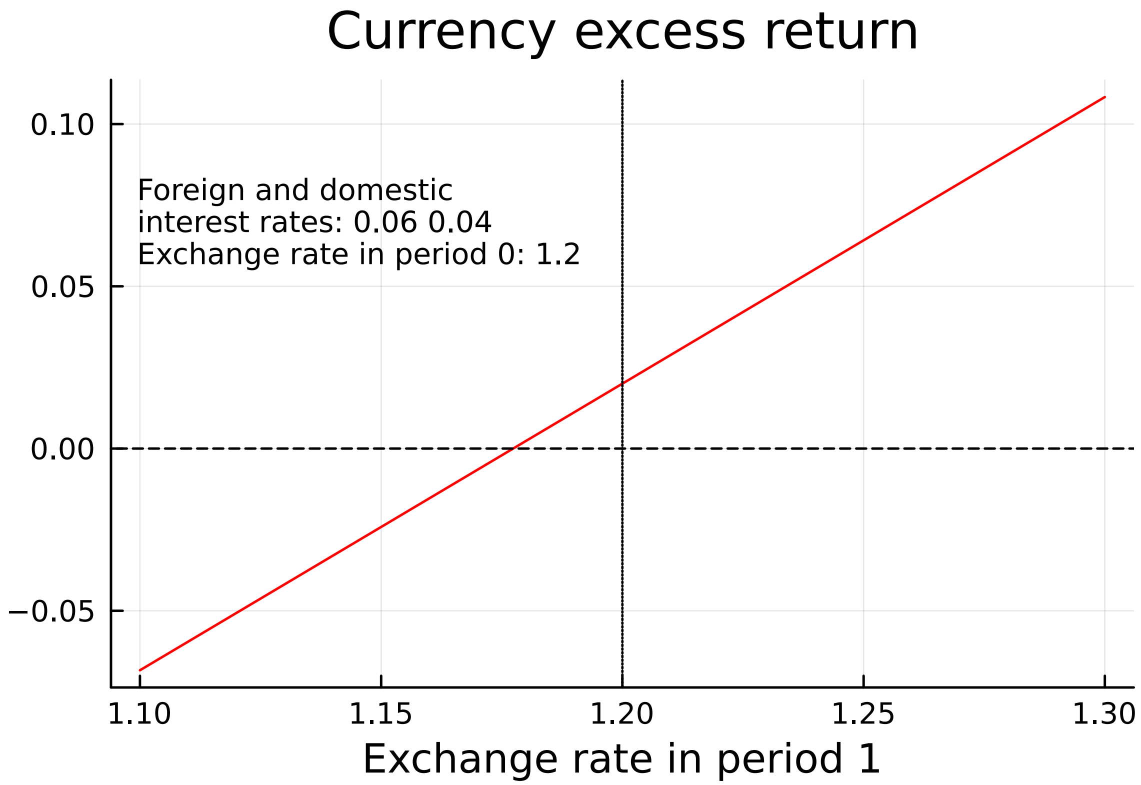
A Remark on the Code

The code uses `Rfˣ` to denote R_f^* (since there is no easy way to get a subscript `f` or a superscript $*$).

```
S₀   = 1.2          #current spot FX rate, t=0
S₁   = 1.25         #spot FX rate in t=1
Rfˣ  = 0.06         #safe ledning rate (foreign) between period 0 and 1
Rf   = 0.04         #safe domestic borrowing rate

Re = (1+Rfˣ)*S₁/S₀ - (1+Rf)

printblue("A simple example of how to calculate the excess return from investing in a foreign curre
xx = [S₀,Rf,Rfˣ,S₁,Re]
printmat(xx;rowNames=["S₀";"Rf";"Rfˣ";"S₁";"Currency excess return"])
```

```
A simple example of how to calculate the excess return from investing in a foreign currency:

S₀                          1.200
Rf                          0.040
Rfˣ                         0.060
S₁                          1.250
Currency excess return      0.064
```

```
S₁_range = 1.1:0.01:1.3           #range of different possible exchange rates in t=1

Re = (1+Rfˣ)*S₁_range/S₀ .- (1+Rf)  #corresponding returns

txt = "Foreign and domestic \ninterest rates: $Rfˣ $Rf \nExchange rate in period 0: $S₀"

p1 = plot( S₁_range,Re,
           legend = nothing,
           linecolor = :red,
           title = "Currency excess return",
           xlabel = "Exchange rate in period 1",
           annotation = (1.1,0.07,text(txt,8,:left)) )
vline!([S₀],linecolor=:black,line=(:dot,1))
hline!([0],linecolor=:black,line=(:dash,1))
display(p1)
```



Uncovered Interest Rate Parity (UIP)

UIP *assumes* that the expected future exchange rate ($\mathrm{E}_0 S_m$) is related to the current exchange rate and interest rates in such a way that the expected excess return of a foreign investment is zero.

In addition, interest rates are typically annualized (denoted Y and Y^* below). This means that the safe (gross) rate over an investment period of m years (eg. $m = 1/12$ for a month) is $(1 + Y)^m$.

```
S₀  = 1.2                        #current spot FX rate
Y   = 0.04                       #annualized interest rates
Yˣ  = 0.06
m   = 1/2                        #investment period
ESₘ = S₀ * (1+Y)^m/(1+Yˣ)^m      #implies E(excess return) = 0

printblue("Expected future exchange rate $m years ahead according to UIP:\n")
xx = [S₀,Y,Yˣ,m,ESₘ]
printmat(xx;rowNames=["S₀";"Y";"Yˣ";"m";"UIP 'expectation' of Sₘ"])
```

```
Expected future exchange rate 0.5 years ahead according to UIP:

S₀                          1.200
Y                           0.040
Yˣ                          0.060
m                           0.500
UIP 'expectation' of Sₘ     1.189
```

A Carry Trade Strategy

Means betting on high interest currencies and against low interest rate currencies.

```julia
using Dates, DelimitedFiles, Statistics
```

Load Data

Returns from FX investments (for a US investor) in percent are in `Data_FxReturns.csv` and log forward premia in percent are in `Data_FxForwardpremia.csv`.

```julia
CurrNames = ["AUD","CAD","EUR","JPY","NZD","NOK","SEK","CHF","GBP"]   #currency abbreviations

x  = readdlm("Data/Data_FxReturns.csv",',',skipstart=1)           #return data
(dN,R) = (Date.(x[:,1]),Float64.(x[:,2:end]))

x   = readdlm("Data/Data_FxForwardpremia.csv",',',skipstart=2)    #forward premia, skip 2 rows
(dN2,fp) = (Date.(x[:,1]),Float64.(x[:,2:end]))

(T,n) = size(R)               #number of data points, number of currencies

println("Same dates in the two files? ",dN == dN2)
```

```
Same dates in the two files? true
```

Sorting on Forward Premia

The `m=3` currencies with the highest forward premia (interest rate differential) in $t-1$ are given portfolio weights `w_CT[t,i] = 1/m`. These are the investment currencies. The `m` currencies with the lowest forward premia in $t-1$ are given portfolio weights `w_CT[t,i] = -1/m`. These are the funding currencies.

A Remark on the Code

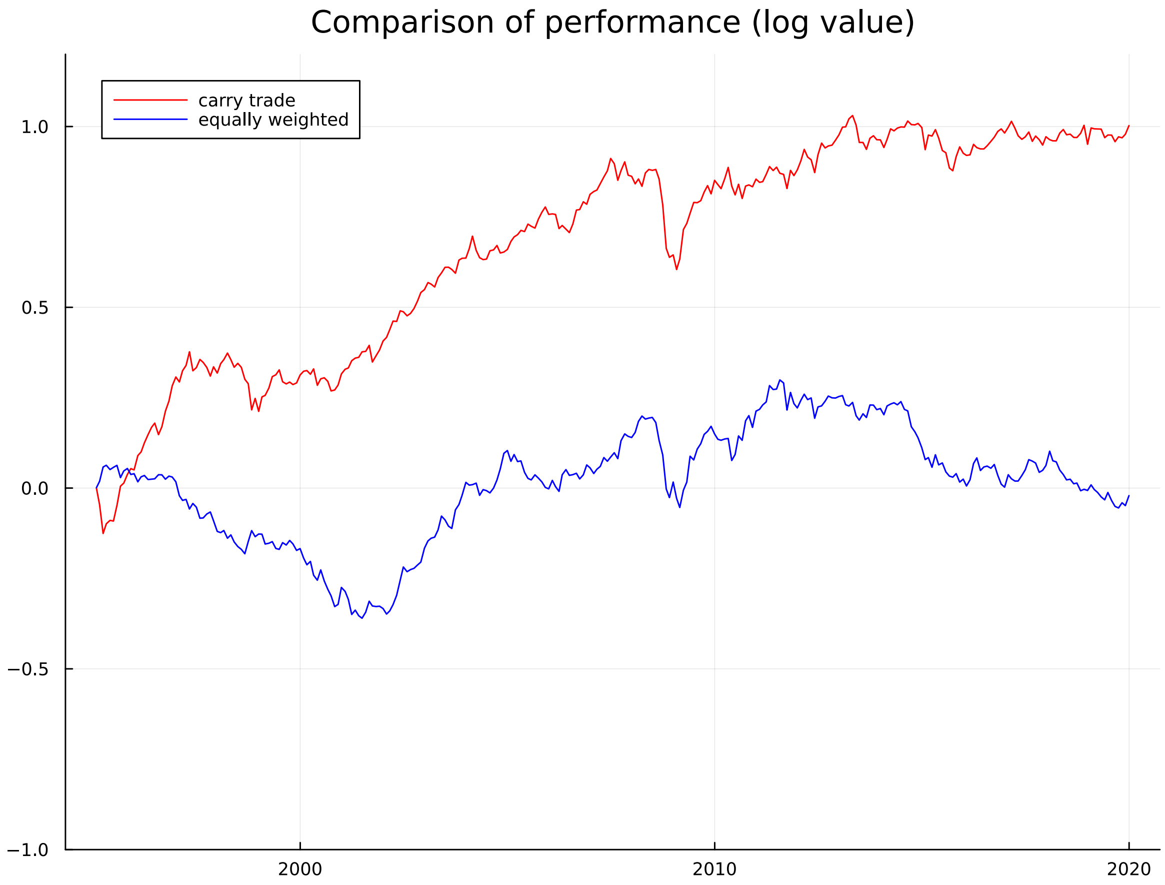
- With `x = [9,7,8]`, the `rankPs(x)` function (see below) gives the output `[3,1,2]`. This says, for instance, that `7` is the lowest number (rank 1).
- `R.*w_CT` creates a `Txn` matrix, `sum(,dims=2)` sums across columns (for each row).

```
"""
    rankPs(x)

Calculates the ordinal rank of eack element in a vector `x`. As an aternative,
use `ordinalrank` from the `StatsBase.jl` package.

"""
rankPs(x) = invperm(sortperm(x))
```

```
rankPs
```

```
m = 3                #number of long/short positions

w_CT = zeros(T,n)
for t = 2:T         #loop over periods, save portfolio returns
    #local r,wL,wH      #local/global is needed in script
    r         = rankPs(fp[t-1,:])
    wL        = (r.<=m)/m               #low interest rate currencies
    wH        = ((n-m+1).<=r)/m                #high interest rate currencies
    w_CT[t,:] = wH - wL
end

R_CT = sum(R.*w_CT,dims=2)              #return on carry trade portfolio

R_EW = mean(R,dims=2);                   #return on equally weighted portfolio
```

```
p_CT = cumsum(log.(1.0.+R_CT/100),dims=1)
p_EW = cumsum(log.(1.0.+R_EW/100),dims=1)

xTicksLoc = [Date(2000),Date(2010),Date(2020)]
xTicksLab = Dates.format.(xTicksLoc,"Y")

p1 = plot( dN,[p_CT p_EW],
           label = ["carry trade" "equally weighted"],
           size = (800,600),
           linecolor = [:red :blue],
           xticks = (xTicksLoc,xTicksLab),
           ylims = (-1,1.2),
           title = "Comparison of performance (log value)")
display(p1)
```



Forward Contracts

This notebook introduces forward contracts and applies the forward-spot parity to different assets.

Load Packages and Extra Functions

```
using Printf

include("src/printmat.jl");
```

```
using Plots
default(size = (480,320),fmt = :png)
```

Present Value

With a continuously compounded interest rate y, the present value of receiving Z in m years is $e^{-my}Z$.

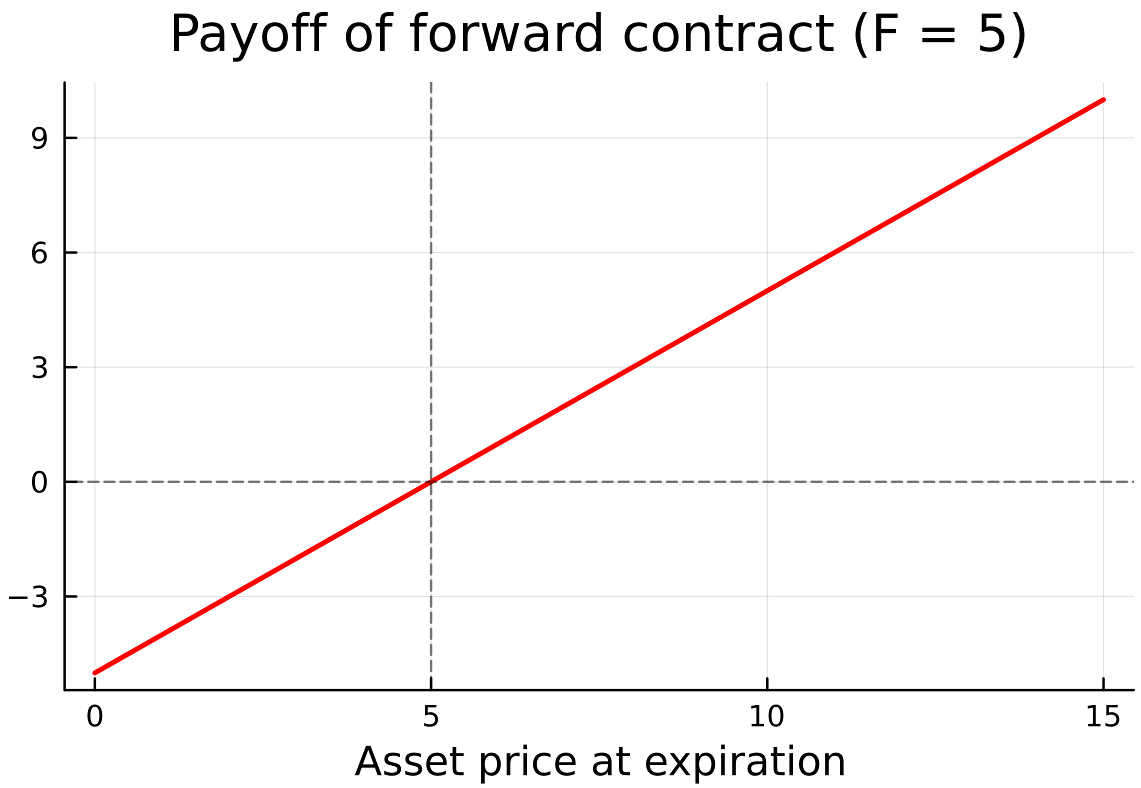
```
(y,m,Z)  = (0.05,3/4,100)
PV = exp(-m*y)*Z

printlnPs("PV of $Z when m=$m and y=$y: ",PV)
```

```
PV of 100 when m=0.75 and y=0.05:     96.319
```

Payoff of Forward Contract

The payoff of a forward contract (at expiration, m years ahead) is $S_m - F$, where S_m is the value of the underlying (at expiration) and F is the forward price (agreed upon at inception of the contract).

```julia
Sₘ_range = range(0,15,length=16)   #possible values of the underlying price at expiration
F   = 5
ForwardPayoff = Sₘ_range .- F

p1 = plot( Sₘ_range,ForwardPayoff,
           linecolor = :red,
           linewidth = 2,
           legend = false,
           title = "Payoff of forward contract (F = $F)",
           xlabel = "Asset price at expiration" )
hline!([0],linecolor=:black,line=(:dash,1,0.5))
vline!([F],linecolor=:black,line=(:dash,1,0.5))
#display(p1)
```



Forward-Spot Parity

For an asset without dividends (at least until expiration of the forward contract),

$F = e^{my}S$,

where F is the forward price (agreed upon today, to be paid m years ahead), S the *current* spot price, m the time to expiration of the forward contract and y the interest rate.

In contrast, for an asset with continuous dividends at the rate δ,

$F = e^{m(y-\delta)}S$.

```
(y,m,S) = (0.05,3/4,100)   #interest rate, time to expiration (in years), spot price now
F_A = exp(m*y)*S           #forward price

δ   = 0.01             #dividend rate
F_B = exp(m*(y-δ))*S   #forward price when there are dividends

printblue("Forward prices:")
printmat([F_A F_B];colNames=["no dividends";"with dividends"],width=16)
```

```
Forward prices:
    no dividends  with dividends
         103.821         103.045
```

Return on a Covered Strategy

Buy the underlying asset now (S) and issue a forward contract, and get the forward price (F) at expiration. This is a risk-free strategy with a gross return of

$1 + R = F/S = e^{my}$,

if the parity holds. Thus, $\ln(1 + R)/m = y$.

```
printmat(log(F_A/S)/m,y;colNames=["ln(1+R)/m","y"])
```

```
 ln(1+R)/m         y
     0.050     0.050
```

Application: Forward Price of a Bond

The forward price (in a forward contract with expiration m years ahead) of a bond that matures in $n \geq m$ is

$F = e^{my(m)}B(n)$,

where $y(m)$ denotes the (annualized) interest rate for an m-period loan.

By definition, $1/B(m) = e^{my(m)}$. Combine to get

$F = B(n)/B(m)$.

```
(m,n)   = (5,7)        #time to expiration of forward,time to maturity of bond
(ym,yn) = (0.05,0.06)  #interest rates
Bm = exp(-m*ym)     #bond price now, maturity m
Bn = exp(-n*yn)     #bond price, maturity n
F  = Bn/Bm          #forward price a bond maturing in n, delivered in m

printblue("Bond and forward prices, assuming a face value of 1:")
printmat([Bm,Bn,F];colNames=["price"],rowNames=["$(m)y-bond","$(n)y-bond","$(m)y->$(n)y forward"])
```

```
Bond and forward prices, assuming a face value of 1:
                   price
5y-bond            0.779
7y-bond            0.657
5y->7y forward     0.844
```

Application: Covered Interest Rate Parity

The "dividend rate" on foreign currency is the foreign interest rate y^* (since you can keep the foreign currency on a foreign bank/money market account). The forward-spot parity then gives

$F = e^{m(y-y^*)}S$.

```
(m,S)  = (1,1.2)            #time to expiration, #exchange rate now
(y,y˟) = (0.0665,0.05)      #domestic interest rate, foreign interest rate

F = exp(m*(y-y˟))*S

printlnPs("Forward price of foreign currency: ",F)
```

```
Forward price of foreign currency:      1.220
```

Bonds 1

This notebook discusses bond price, interest rates, forward rates, yield to maturity and (as extra material) estimates the yield curve.

The InterestRates.jl and FinanceModels.jl packages contain many more methods for working with bonds.

Load Packages and Extra Functions

```
using Printf, Roots

include("src/printmat.jl");
```

```
using Plots
default(size = (480,320),fmt = :png)
```

Interest Rate vs (Zero Coupon) Bond Price

Zero coupon bonds (also called bills or discount bonds) have very simple cash flows: buy the bond now and get the face value (here normalised to 1) at maturity. Clearly, it can be resold at any time.

The bond price (for maturity m years ahead) is a function of the effective interest rate

$B = (1 + Y)^{-m}$

and the inverse is $Y = B^{-1/m} - 1$. For instance, $m = 1/12$ is a month and $m = 2$ is two years.

Instead, with a continuously compounded interest rate we have

$B = e^{-my}$

and $y = -(\ln B)/m$.

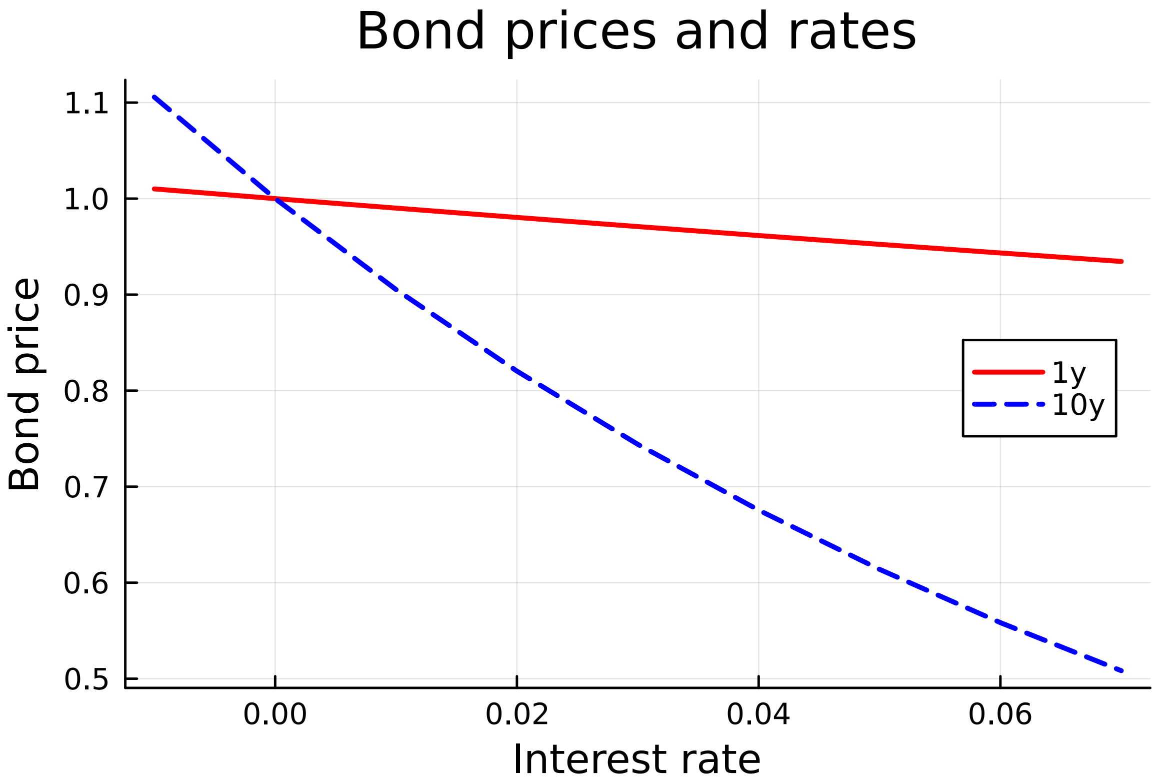
These expressions are coded up as short functions in the next cell.

```
"""
Zero coupon bond/bill price B as a function of Y and m
"""
BillPrice(Y,m) = (1+Y)^(-m)

"""
Effective interest rate Y as a function of B and m
"""
EffRate(B,m)   = B^(-1/m) - 1

"""
Zero coupon bond/bill price B as a function of y and m
"""
BillPrice2(y,m)   = exp(-m*y)

"""
Continuously compounded interest rate y as a function of B and m
"""
ContCompRate(B,m) = -log(B)/m
```

```
ContCompRate
```

Long-maturity bonds are more sensitive to interest rate changes than short-maturity bonds, as illustrated by the plot below.

```
Y_range = -0.01:0.01:0.07          #different interest rates
B1      = BillPrice.(Y_range,1)    #prices at different interest rates, 1-year zero coupon bond
B10     = BillPrice.(Y_range,10)   #10-year bond

p1 = plot( Y_range,[B1 B10],
           linecolor = [:red :blue],
           linestyle = [:solid :dash],
           linewidth = 2,
           label = ["1y" "10y"],
           legend = :right,
           title = "Bond prices and rates",
           xlabel = "Interest rate",
           ylabel = "Bond price" )
display(p1)
```



Effective and continuously compounded interest rates are fairly similar at low interest rates, but start to diverge at high rates.

```
m  = 0.5                              #maturity, in years
B  = 0.95                             #bond price

Y = EffRate(B,m)                      #effective interest rate
y = ContCompRate(B,m)                 #continuously compounded interest rate

printblue("Interest rates ($m-year bond with a price of $B):")
printmat([Y,y];rowNames= ["Effective","Continuously compounded"])
```

```
Interest rates (0.5-year bond with a price of 0.95):
Effective                    0.108
Continuously compounded      0.103
```

Bond Returns (Zero Coupon Bonds)

Let B_0 be the bond price in period 0 and B_s the price of the same bond s periods later. The *log* return of holding this bond is

$r_s = \ln(B_s/B_0)$.

In the computations below, the time to maturity is assumed to be the same in $t = 0$ and $t = s$. This is a reasonable *approximation* if the holding period s are a few days or perhaps weeks, while time to maturity is much longer.

A Remark on the Code

- The code below lets B_0 and B_s be vectors with prices for different maturities.

```
y₀ = 0.01                      #(continuously compounded) interest rate before
yₛ = 0.02                       #interest rate after

B₀ = BillPrice2.([y₀ y₀],[1 10])  #bond prices (1-year and 10 year) before, same interest rates, ve
Bₛ = BillPrice2.([yₛ yₛ],[1 10])    #bond prices (1-year and 10 year) after
rₛ = log.(Bₛ./B₀)                 #returns

xy       = vcat([y₀ y₀],B₀,[yₛ yₛ],Bₛ,rₛ)                #to table
rowNames = ["Rate, t=0";"Bond price, t=0";"Rate, t=s";"Bond price, t=s";"Return"]
printblue("Analysis of two bonds: 1y and 10y maturities")
printmat(xy*100;colNames=["1-year";"10-year"],rowNames=rowNames,width=15,prec=2)

printred("Notice that the return is -m*Δy")
```

```
Analysis of two bonds: 1y and 10y maturities
                        1-year        10-year
Rate, t=0                 1.00           1.00
Bond price, t=0          99.00          90.48
Rate, t=s                 2.00           2.00
Bond price, t=s          98.02          81.87
Return                   -1.00         -10.00

Notice that the return is -m*Δy
```

Forward Rates

From the forward-spot parity, the forward price of a bond (delivered in $t = m$ and maturing in $t = n$, with time again measured in years) is

$F = [1 + Y(m)]^m B(n) = B(n)/B(m)$.

A forward interest rate can then be defined as

$\Gamma(m, n) = F^{-1/(n-m)} - 1$.

This is the effective interest rate paid (or received) between $t = m$ and $t = n$, but agreed upon already in $t = 0$.

```
"""
    ForwardRate(Ym,m,Yn,n)

Calculate late forward rate for the investment period m to n (where m<n).

# Input
- Ym: interest rate for maturity m
- m: maturity m
- Yn: interest rate for maturity n
- n: maturity n

"""
function ForwardRate(Ym,m,Yn,n)           #forward rate
    if m >= n
        error("m<n is required")
    end
    Bm = (1+Ym)^(-m)
    Bn = (1+Yn)^(-n)
    F  = Bn/Bm
    Γ  = F^(-1/(n-m)) - 1
    return Γ
end
```

```
ForwardRate
```

```
(m,n,Ym,Yn) = (0.5,0.75,0.04,0.05)

Γ = ForwardRate(Ym,m,Yn,n)
printlnPs("\nImplied forward rate ($m-year -> $n-year): ",Γ)
```

```
Implied forward rate (0.5-year -> 0.75-year):      0.070
```

Coupon Bond Prices

Recall that a coupon bond price P is the portfolio value

$P = \sum_{k=1}^{K} B(m_k)cf_k$, or

$P = \sum_{k=1}^{K} \frac{cf_k}{[1+Y(m_k)]^{m_k}}$.

where cf_k is the cash flow m_k years ahead and $B(m_k)$ is the price of a zero-coupon bond maturing in the same period. The last cash flow includes also the payment of the face value (if any).

```
B  = [0.95,0.9]              #B(1),B(2)
c  = 0.06                    #coupon rate
cf = [c,1+c]                 #cash flows in m=1 and 2

P_components = B.*cf
P = sum(B.*cf)

printblue("\n2-year bond with $c coupon:")
rowNames = ["1-year zero coupon bond price","2-year zero coupon bond price","coupon bond price"]
xx = hcat([B;NaN],[P_components;P])
printmat(xx;rowNames,colNames=["zero-coupon price","zero-coupon price*cf"],width=25)
```

```
2-year bond with 0.06 coupon:
                                         zero-coupon price     zero-coupon price*cf
1-year zero coupon bond price                        0.950                    0.057
2-year zero coupon bond price                        0.900                    0.954
coupon bond price                                      NaN                    1.011
```

A Remark on the Code

- The `BondPrice()` function below can handle both the case when Y is a vector with different values for different maturities and when Y is a scalar (same interest rate for all maturities). It can also handle continously compounded interest rates by a keyword argument.

```
"""
    BondPrice(Y,cf,m;ContRateQ=false)

Calculate bond price as sum of discounted cash flows.

# Input:
- `Y`:  scalar or K vector of interest rates
- `cf`: K vector of cash flows
- `m`:  K vector of years to the cash flows
- `ContRateQ`: if true: `Y` is interpreted as a continously compounded rate
"""
function BondPrice(Y,cf,m;ContRateQ=false)              #cf is a vector of all cash flows at times
    if length(cf) != length(m)
        error("BondPrice: cf and m must have the same lengths")
    end
    if ContRateQ
        cdisc = cf./exp.(m.*Y)
    else
        cdisc = cf./((1.0.+Y).^m)        #c/(1+y[1])^t1 + c/(1+y[2])^t2 + ...+ c/(1+y[m])^tm
    end
    P = sum(cdisc)                       #price
    return P
end
```

```
BondPrice
```

```
Y = [0.053,0.054]
c = 0.06
P =  BondPrice(Y,[c,c+1],[1,2])
printblue("\n2-year bond with $c coupon:")
rowNames = ["1-year spot rate","2-year spot rate","bond price"]
printmat([Y;P];rowNames)

Y = [0.06,0.091]
c = 0.09
P =  BondPrice(Y,[c,c+1],[1,2])
printblue("\n2-year bond with $c coupon:")
printmat([Y;P];rowNames)
```

```
2-year bond with 0.06 coupon:
```

```
1-year spot rate     0.053
2-year spot rate     0.054
bond price           1.011


2-year bond with 0.09 coupon:
1-year spot rate     0.060
2-year spot rate     0.091
bond price           1.001
```

Bootstrapping

With information about coupons $c(m)$ and coupon bond price $P(m)$, we solve for the implied zero coupon bond prices $B(s)$ from a system like (here with just 2 maturities)

$$\begin{bmatrix} P(1) \\ P(2) \end{bmatrix} = \begin{bmatrix} c(1)+1 & 0 \\ c(2) & c(2)+1 \end{bmatrix} \begin{bmatrix} B(1) \\ B(2) \end{bmatrix}$$

Notice that each row refers to a specific bond and that each column (in matrix) to different times of cash flows.

A Remark on the Code

`B = cfMat\P` solves `P = cfMat*B` for `B`.

```
c     = [0,0.06]
P     = [0.95,1.01]                 #coupon bond prices
m     = [1,2]                       #time of coupon payments
cfMat = [1    0     ;               #cash flow matrix
         c[2] 1+c[2]]

println("The cash flow matrix")
printmat(cfMat,colNames=["year 1";"year 2"],rowNames=["Bond 1";"Bond 2"],colUnderlineQ=true)

println("B from solving P = cfMat*B (implied zero-coupon bond prices):")
B = cfMat\P
printmat(B,rowNames=["1-year";"2-year"])

Y = EffRate.(B,m)            #solve for the implied spot rates
```

```
println("Implied spot interest rates:")
printmat(Y,rowNames=["1-year";"2-year"])
```

```
The cash flow matrix
          year 1    year 2
          ------    ------

Bond 1     1.000     0.000
Bond 2     0.060     1.060

B from solving P = cfMat*B (implied zero-coupon bond prices):
1-year     0.950
2-year     0.899

Implied spot interest rates:
1-year     0.053
2-year     0.055
```

Yield to Maturity

The yield to maturity (ytm) is the θ that solves

$P = \sum_{k=1}^{K} \frac{cf_k}{(1+\theta)^{m_k}}$.

We typically have to find θ by a numerical method.

A Remark on the Code

- The Roots.jl package is used to find the ytm.
- The `find_zero(fn,(lower,upper))` finds a root of the function `fn` in the interval `(lower,upper)`.
- The function is here `θ->BondPrice(θ,cf,m)-P`. This expression creates an anonymous function that takes `θ` as the only argument, and calculates the difference between the price according to `BondPrice(θ,cf,m)` and the actual price `P`. The ytm `θ` makes this zero.

```
c  = 0.04                  #simple case
Y  = 0.03                  #all spot rates are 3%
m  = [1,2]                 #time of cash flows
cf = [c,c+1]               #cash flows

P = BondPrice(Y,cf,m)
```

```
ytm = find_zero(θ->BondPrice(θ,cf,m)-P,(-0.1,0.1))   #solving for ytm

printblue("Price and ytm of 2-year $c coupon bond when all spot rates are $Y:")
printmat([P,ytm],rowNames=["price","ytm"])
```

```
Price and ytm of 2-year 0.04 coupon bond when all spot rates are 0.03:
price     1.019
ytm       0.030
```

```
m  = [1,3]                #cash flow in year 1 and 3
cf = [1,1]                #cash flows
Y  = [0.07,0.10]          #spot interest rates for different maturities

P  = BondPrice(Y,cf,m)
ytm = find_zero(y->BondPrice(y,cf,m)-P,(-0.2,0.2))

printblue("'bond' paying 1 in both t=1 and in t=3")
printmat([Y;ytm];rowNames=["1-year spot rate","3-year spot rate","ytm"])
```

```
'bond' paying 1 in both t=1 and in t=3
1-year spot rate     0.070
3-year spot rate     0.100
ytm                  0.091
```

Estimating Yield Curve with Regression Analysis (extra)

Recall: with a quadratic discount function

$B(m) = a_0 + a_1 m + a_2 m^2$,

we can write the coupon bond price

$P(m_K) = \sum_{k=1}^{K} B(m_k)c + B(m_K)$ as

$P(m_K) = \sum_{k=1}^{K} (a_0 + a_1 m_k + a_2 m_k^2)c + (a_0 + a_1 m_K + a_2 m_K^2)$

Collect terms as

$P(m_K) = a_0(Kc + 1) + a_1(c\sum_{k=1}^{K} m_k + m_K) + a_2(c\sum_{k=1}^{K} m_k^2 + m_K^2)$

We estimate (a_0, a_1, a_2) by using the terms (within parentheses) as regressors.

A Remark on the Code

`a = x\P` solves `P = x*a` if `x` is square, and computes the least-squares solution when `x` is rectangular (as in the cell below).

```
c     = [0,0.06]
P     = [0.95,1.01]                 #coupon bond prices
m     = [1,2]                       #time of coupon payments

n = length(P)
x = zeros(n,3)                      #create regressors for quadratic model: 3 columns
for i in 1:n                         #x[i,j] is for bond i, regressor j
    x[i,1] = m[i]*c[i] + 1
    x[i,2] = c[i]*sum(m[1]:m[i]) + m[i]
    x[i,3] = c[i]*sum(abs2,m[1]:m[i]) + m[i]^2    #sum(abs2,x) is the same as sum(x.^2)
end
println("regressors:")
printmat(x,colNames=["term 0";"term 1";"term 2"],rowNames=["Bond 1";"Bond 2"],colUnderlineQ=true)

a = x\P                             #regress P on x
println("regression coefficients")
printmat(a)

m = 1:5
B = [ones(length(m)) m m.^2]*a      #fitted discount function

printmat(B,colNames=["fitted B"],rowNames=string.(m,"-year"))
printred("Btw. do B[4:5] make sense? If not, what does that teach us?\n")

Y = EffRate.(B[1:2],m[1:2])         #solve for the implied spot rates
printmat(Y,colNames=["fitted Y"],rowNames=string.(m[1:2],"-year"))
```

```
regressors:
          term 0    term 1    term 2
          ------    ------    ------

Bond 1     1.000     1.000     1.000
Bond 2     1.120     2.180     4.300

regression coefficients
     0.693
     0.411
    -0.154
```

```
        fitted B
1-year     0.950
2-year     0.899
3-year     0.540
4-year    -0.126
5-year    -1.101
```

Btw. do B[4:5] make sense? If not, what does that teach us?

```
        fitted Y
1-year     0.053
2-year     0.055
```

Bonds 2

This notebook discusses *duration hedging* as a way to immunize a bond portfolio.

Load Packages and Extra Functions

```
using Printf, Roots

include("src/printmat.jl");
```

```
using Plots
default(size = (480,320),fmt = :png)
```

A Remark on the Code

The file included below contains the function `BondPrice()` which calculates the present value of a cash flow stream. (It's an extract from the notebook on Bonds 1.)

```
include("src/BondCalculations.jl");
```

Value of a Bond Portfolio after a Sudden Interest Rate Change

The calculations below assume that the *yield curve is flat*, but that it can shift up or down in parallel. This assumption is similar to the classical literature on duration hedging.

The initial values are indicated by the subscript $_0$ and the values after the interest rate change by the subscript $_1$. It is assumed that the change is very sudden, so the time to the different cash flows is (virtually) the same before and after. For instance, the subscripts 0 and 1 could be interpreted as "day 0" and "day 1".

The next cell sets up the cash flow for a bond portolio L ("Liability") that pays the amount 0.2 each year for the next 10 years. The value of L is calculated at an yield to maturity (θ_0) and at a new yield to maturity (θ_1). Notice that with a flat yield curve, the yield to maturity equals the (one and only) interest rate.

Time to cash flow (m) is measured in years.

```
θ₀ = 0.05                     #initial interest rate
θ₁ = 0.03                     #interest rate after sudden change

cf = ones(10)*0.2             #cash flow of liability, 0.2 in year 1,2,...,10
m  = 1:10                     #time periods of the cash flows, years

PL₀ = BondPrice(θ₀,cf,m)     #value of bond L at initial interest rate
PL₁ = BondPrice(θ₁,cf,m)     #and after sudden change in interest rate
R   = (PL₁ - PL₀)/PL₀        #relative change of the value

printblue("Value of bond portfolio at θ₀=$θ₀ and θ₁=$θ₁:")
xy = [PL₀, PL₁, R]
rowNames = ["PL₀ (at θ₀=$θ₀)";"PL₁ (at θ₁=$θ₁)";"ΔPL/PL₀ (return)"]
printmat(xy;rowNames,width=15)

printred("Notice that the bond is worth more at the lower interest rate")
```

```
Value of bond portfolio at θ₀=0.05 and θ₁=0.03:
PL₀ (at θ₀=0.05)          1.544
PL₁ (at θ₁=0.03)          1.706
ΔPL/PL₀ (return)          0.105

Notice that the bond is worth more at the lower interest rate
```

Durations

Let D^M denote Macaulay's duration

$D^M = \sum_{k=1}^{K} m_k \frac{cf_k}{(1+\theta)^{m_k} P}$.

It can be interpreted as the weighted average time (years) to cash flow.

The modified duration is

$D = D^M/(1+\theta)$,

which can be used in a first-order Taylor approximation to get the (approximate) return $\frac{\Delta P}{P} \approx -D \times \Delta\theta$.

```
"""
    BondDuration(P,cf,m,θ)

Calculate Macaulay's and the modified  (bond) duration measures.

P:   scalar, bond price
cf:  scalar or K vector of cash flows
m:   K vector of times of cash flows
θ: scalar, yield to maturity
"""
function BondDuration(P,cf,m,θ)
    if length(cf) != length(m)
        error("cf and m must have the same lengths")
    end
    cdisc = cf.*m./((1+θ).^m)        #cf1/(1+y) + 2*cf2/(1+y)^2 + ...
    Dmac  = sum(cdisc)/P             #Macaulays duration
    D     = Dmac/(1+θ)               #modified duration
    return Dmac, D
end
```

```
BondDuration
```

```
(Dmac,DL) = BondDuration(PL₀,cf,m,θ₀)
printlnPs("Macaulay's and the modified durations of the bond portfolio L:",Dmac,DL)

Δθ       = θ₁ - θ₀              #change of ytm
R_approx = -DL*Δθ

printblue("\nRelative price change (return): ")
printmat([R,R_approx,];rowNames=["Exact","Approximate (from duration)"])
```

```
Macaulay's and the modified durations of the bond portfolio L:     5.099     4.856

Relative price change (return):
Exact                           0.105
Approximate (from duration)     0.097
```

Hedging a Liability Stream

Suppose we are short one bond portfolio L (we have a liability) as discussed above, which is worth P_L. To hedge against interest rate changes, we buy v units of bond (portfolio) H. The balance is put on a money market account M to make the initial value of the portfolio zero ($V = 0$)

$V = vP_H + M - P_L$.

Over a short time interval, the change (indicated by Δ) in the overall portfolio value is

$\Delta V = v\Delta P_H - \Delta P_L$.

In the cells below, we assume that the yield curve is flat and shifts in parallel. This means that the ytm of both L and H change from one common value (θ_0) to another common value (θ_1).

Duration Matching

Duration matching means that we choose a hedge bond with the *same duration* as the liability and invest same amount in each. Clearly, this gives $\frac{\Delta V}{P_L} \approx 0$.

The code below uses a zero coupon bond as bond H, but that is not important. It has the same maturity as Macaulay's duration of the liability.

```
PH₀ = BondPrice(θ₀,1,Dmac)      #value of hedge bond before, here a zero coupon bond
PH₁ = BondPrice(θ₁,1,Dmac)      #value of hedge bond, after

(_,DH) = BondDuration(PH₀,1,Dmac,θ₀)   #modified duration of hedge bond

v  = PL₀/PH₀                     #so v*PH₀ = PL₀, same amount in L and H

ΔV = v*(PH₁-PH₀) - (PL₁-PL₀)     #value change of total portfolio
R  = ΔV/PL₀                      #relative value change

txt = """Hedge bond: a zero coupon bond with m=$(round(Dmac,digits=2)) and face value of 1.\n
Recall that the interest rates change from θ₀=$θ₀ to θ₁=$θ₁.\n
Results of the calculations:"""
printblue(txt)

xy = [PL₀,PH₀,v,v*PH₀/PL₀,DL,DH,ΔV]
rowNames = ["PL₀","PH₀","v","v*PH₀/PL₀","Duration(liability)","Duration(hedge)","ΔV"]
printmat(xy;rowNames)

printred("The duration matching gives a return, ΔV/PL₀, of $(round(R*100,digits=1))%. Close to zero
Hedge bond: a zero coupon bond with m=5.1 and face value of 1.

Recall that the interest rates change from θ₀=0.05 to θ₁=0.03.

Results of the calculations:
PL₀                     1.544
PH₀                     0.780
v                       1.981
v*PH₀/PL₀               1.000
Duration(liability)     4.856
Duration(hedge)         4.856
ΔV                     -0.003

The duration matching gives a return, ΔV/PL₀, of -0.2%. Close to zero.
```

Naive Hedging

The "naive" hedging invests the *same amount in the hedge bond* as the value of the liability, that is, $vP_H = P_L$ so $v = P_L/P_H$, but pays no attention to the durations.

The effectiveness of this approach depends on the interest rate sensitivities of L and H. In particular, it can be shown that the relative value change of the overall portfolio is

$\frac{\Delta V}{P_L} \approx (D_L - D_H) \times \Delta\theta,$

so it depends on the difference between the durations (of the liability and hedge bond).

In particular, if $D_L > D_H$, and $\Delta\theta < 0$ (as in the example below), then we will lose money. More importantly, with $D_L \neq D_H$, we are taking a risk (of losses/gains), that is, we are not hedged.

In contrast, with $D_L = D_H$, then we are effectively doing duration matching (see above). Clearly, there are also scenarious

```
mH = 3                        #a mH-year zero coupon bond is used as hedge bond

PH₀  = BondPrice(θ₀,1,mH)
PH₁  = BondPrice(θ₁,1,mH)
(_,DH) = BondDuration(PH₀,1,mH,θ₀)   #modified duration of hedge bond

v = PL₀/PH₀                    #so v*PH₀ = PL₀

ΔV = v*(PH₁-PH₀) - (PL₁-PL₀)
R  = ΔV/PL₀                    #relative value change
```

```
printblue("Hedge bond: zero coupon bond with m=$mH and face value of 1\n")
xy = [PL₀,PH₀,v,v*PH₀/PL₀,DL,DH,ΔV]
rowNames = ["PL₀","PH₀","v","v*PH₀/PL₀","Dur(liability)","Dur(hedge)","ΔV"]
printmat(xy;rowNames)

printred("The naive hedge here gives a return, ΔV/PL₀, of $(round(R*100,digits=1))%, which is a bac
```

```
Hedge bond: zero coupon bond with m=3 and face value of 1

PL₀                 1.544
PH₀                 0.864
v                   1.788
v*PH₀/PL₀           1.000
Dur(liability)      4.856
Dur(hedge)          2.857
ΔV                 -0.070

The naive hedge here gives a return, ΔV/PL₀, of -4.5%, which is a bad hedge
```

Illustrating the Problem with the Naive Hedging

by plotting the value of the liability (P_L) and of the hedge bond position (vP_H) at different interest rates.

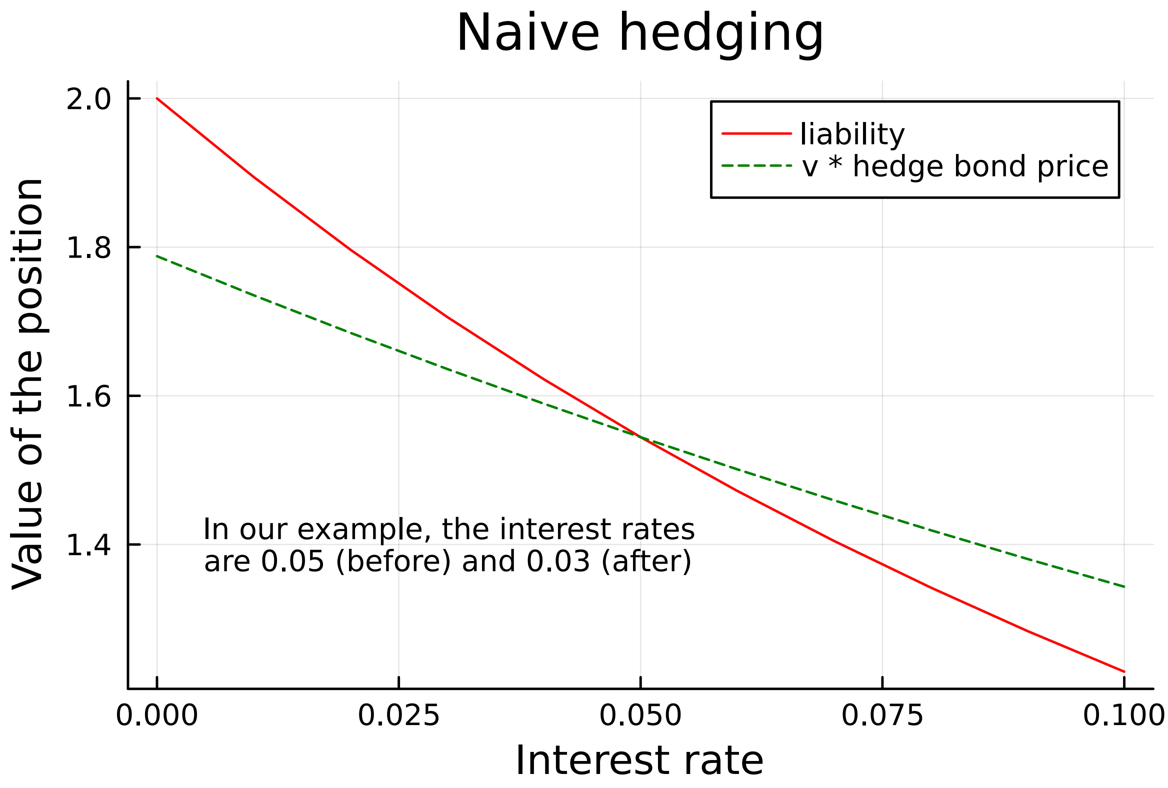
A Remark on the Code

The next cell calculates PL and PH by a list comprehension. To instead do it with a loop use something like

```
PL = fill(NaN,length(θ_range))
for i in 1:length(θ_range)
    PL[i] = BondPrice(θ_range[i],cf,m)
end
```

```
θ_range = 0:0.01:0.1        #possible ytm values

PL = [BondPrice(θ,cf,m) for θ in θ_range]  #list comprehension,
PH = [BondPrice(θ,1,mH) for θ in θ_range]  #an alternative to a loop
```

```
txt = "In our example, the interest rates \nare $θ₀ (before) and $θ₁ (after)"

p1 = plot( θ_range,[PL v*PH],
           linecolor = [:red :green],
           linestyle = [:solid :dash],
           label = ["liability" "v * hedge bond price"],
           title = "Naive hedging",
           xlabel = "Interest rate",
           ylabel = "Value of the position",
           annotation = (0.005,1.4,text(txt,8,:left)))
display(p1)
```



Duration Hedging

With $D_L \neq D_H$, we could *adjust the hedge ratio* v to compensate for the difference in interest rate sensitivity (duration). In particular, we could set

$v = \frac{D_L}{D_H} \times \frac{P_L}{P_H}$.

The balance $(P_L - vP_H)$ is kept on a money market account (M).

It can be shown that this gives an (approximate) hedge.

```
PH₀ = BondPrice(θ₀,1,mH)
PH₁ = BondPrice(θ₁,1,mH)
v   = DL/DH * PL₀/PH₀
M   = PL₀ - v*PH₀              #on money market account

ΔV = v*(PH₁-PH₀) - (PL₁-PL₀)
R  = ΔV/PL₀                    #relative value change

printblue("Hedge bond: zero coupon bond with m=$mH and face value of 1\n")
xy = [PL₀,PH₀,v,v*PH₀/PL₀,DL,DH,M,ΔV]
rowNames = ["PL₀","PH₀","v","v*PH₀/PL₀","Dur(liability)","Dur(hedge)","M","ΔV"]
printmat(xy;rowNames)

printred("The duration hedging gives a return, ΔV/PL₀, of $(round(R*100,digits=1))%. Close to zero.
```

```
Hedge bond: zero coupon bond with m=3 and face value of 1

PL₀                   1.544
PH₀                   0.864
v                     3.039
v*PH₀/PL₀             1.700
Dur(liability)        4.856
Dur(hedge)            2.857
M                    -1.081
ΔV                   -0.006

The duration hedging gives a return, ΔV/PL₀, of -0.4%. Close to zero.
```

Convexity (extra)

A second-order Taylor approximation gives that

$$\frac{\Delta P}{P} \approx -D \times \Delta\theta + \frac{1}{2}C \times (\Delta\theta)^2,$$

where $C = \frac{1}{P}\frac{d^2P}{d\theta^2}$.

The function below calculates C.

```
"""
    BondConvexity(P,cf,m,ytm)

Calculate the convexity term `C`. The effect on the bond is 0.5C*(Δytm)^2
"""
function BondConvexity(P,cf,m,ytm)
    cdisc = cf.*m.*(m.+1)./((1+ytm).^(m.+2))
    C     = sum(cdisc)/P
    return C
end
```

```
BondConvexity
```

```
C = BondConvexity(PL₀,cf,m,θ₀)

printblue("Convexity of the liability")
xy = [C;Δθ;0.5*C*Δθ^2]
printmat(xy,rowNames=["C";"Δθ";"0.5*C*(Δθ)^2"])

printred("Compare this magnitude to ΔPL/PL: ",round((PL₁-PL₀)/PL₀,digits=3),". It seems convexity i
```

```
Convexity of the liability
C                35.602
Δθ               -0.020
0.5*C*(Δθ)^2      0.007

Compare this magnitude to ΔPL/PL: 0.105. It seems convexity is not important in this case.
```

Bonds 3

This notebook presents a yield curve model (a simplified Vasicek model) and then uses it to improve the duration hedging approach.

Load Packages and Extra Functions

```
using Printf

include("src/printmat.jl")
include("src/BondCalculations.jl");
```

```
using Plots, LaTeXStrings
default(size = (480,320),fmt = :png)
```

Predictions from an AR(1)

Consider an AR(1) with mean μ

$$r_{t+1} - \mu = \rho\left(r_t - \mu\right) + \varepsilon_{t+1}$$

The forecast (based on information in t) for $t + s$ is

$$\mathrm{E}_t r_{t+s} = \left(1 - \rho^s\right)\mu + \rho^s r_t.$$

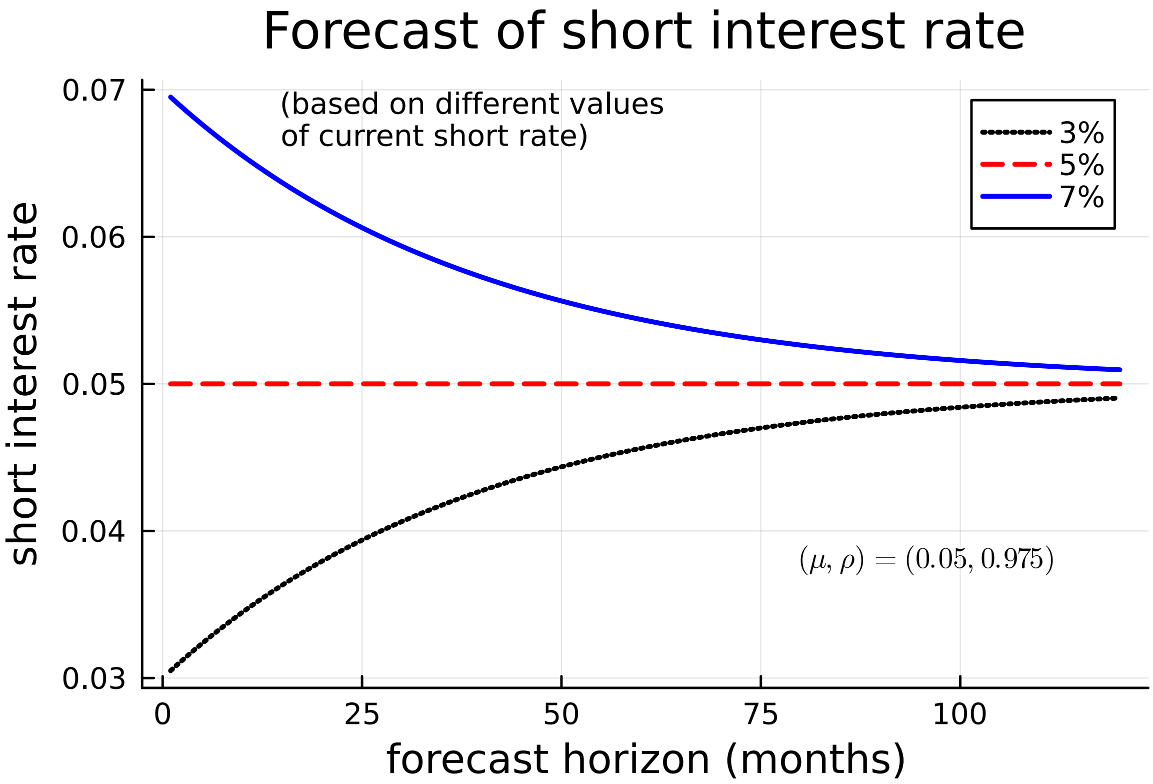
In the the cells below, we code a function for these forecasts and then plot the results (for many forecasting horizons and also for several different starting values).

```
"""
    AR1Prediction(r0,ρ,μ,s)

Calculate forecast from AR(1)

# Input
r0,ρ,μ,s: scalars
"""
AR1Prediction(r0,ρ,μ,s) = (1-ρ^s)*μ + ρ^s*r0;
```

```
(μ,ρ) = (0.05,0.975)

printlnPs("Prediction for t+50, assuming current r=0.07: ",AR1Prediction(0.07,ρ,μ,50))
```

```
Prediction for t+50, assuming current r=0.07:      0.056
```

```
sMax    = 120                             #make forecasts for many horizons and initial values
s_range = 1:sMax                          #different horizons, months
r₀      = [0.03,0.05,0.07]

xPred = fill(NaN,sMax,length(r₀))             #xPred[maturity,r₀ value]
for s in s_range, j = 1:length(r₀)
    xPred[s,j] = AR1Prediction(r₀[j],ρ,μ,s)
end


txt = text(L"(\mu,\rho) = (%$μ,%$ρ)",8,:left)   #laTeX string to plot, notice %$
p1 = plot( s_range,xPred,
           linecolor = [:black :red :blue],
           linewidth = 2,
           linestyle = [:dot :dash :solid],
           label = ["3%" "5%" "7%"],
           title = "Forecast of short interest rate",
           xlabel = "forecast horizon (months)",
           ylabel = "short interest rate",
           annotation = [(80,0.038,txt),
                    (15,0.068,text("(based on different values\nof current short rate)",8,:left))]
display(p1)
```



The Vasicek Model

The simpified (because of no risk premia) Vasicek model implies that

$y_t(n) = a(n) + b(n)r_t$, where

$b(n) = (1 - \rho^n)/[(1 - \rho)n]$,

$a(n) = \mu\,[1 - b(n)]$.

In the example, $y_t(36)$ which corresponds to `y[36]` in the code, is the (annualized) continuously compounded interest rate for a bond maturing in 36 months (3 years).

In the cells below, we code a function for this model and then plots the results for many different maturities (horizons) and several initial values of the short interest rate.

A Remark on the Code

The code (used below) `b = ρ == 1.0 ? 1.0 : (1-ρ^n)/((1-ρ)*n)` is the same as

```
if p == 1.0
    b = 1.0
else
    b = (1-ρ^n)/((1-ρ)*n)
end
```

```
"""
    Vasicek(r,ρ,μ,n)

Vasicek model: calculate interest rate and (a,b) coeffs

# Input
r,ρ,μ,n: scalars

n is the maturity, but this could be measured in any unit (months, years, etc),
but changing the units require recalibrating ρ
"""
function Vasicek(r,ρ,μ,n)
    b = ρ == 1.0 ? 1.0 : (1-ρ^n)/((1-ρ)*n)   #if-else-end
    a = μ*(1-b)
    y = a + b*r
    return y,a,b
end
```

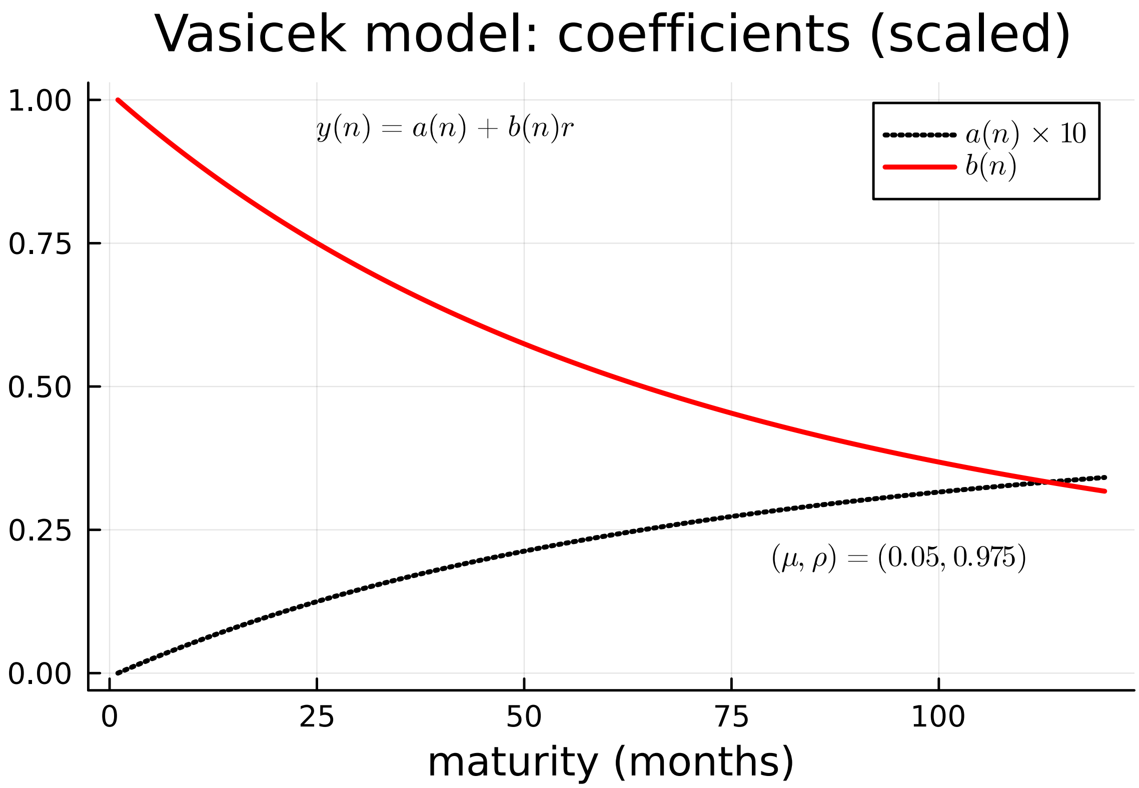
```
Vasicek
```

```
y = fill(NaN,sMax,length(r₀))         #interest rates, different starting values in columns
ab = fill(NaN,sMax,2)                 #a and b coefs
for n in s_range, j = 1:length(r₀)
    #local a, b                       #local/global is needed in script
    (y[n,j],a,b) = Vasicek(r₀[j],ρ,μ,n)
    if j == 1
        ab[n,:] = [a,b]               #update ony if j==1, the same across r₀ values
    end
end

printblue("a and b for the first few horizons (months):")
printmat([ab[1:4,1]*10 ab[1:4,2]];colNames=["a*10","b"],rowNames=string.(s_range[1:4]),cell00="hori
```

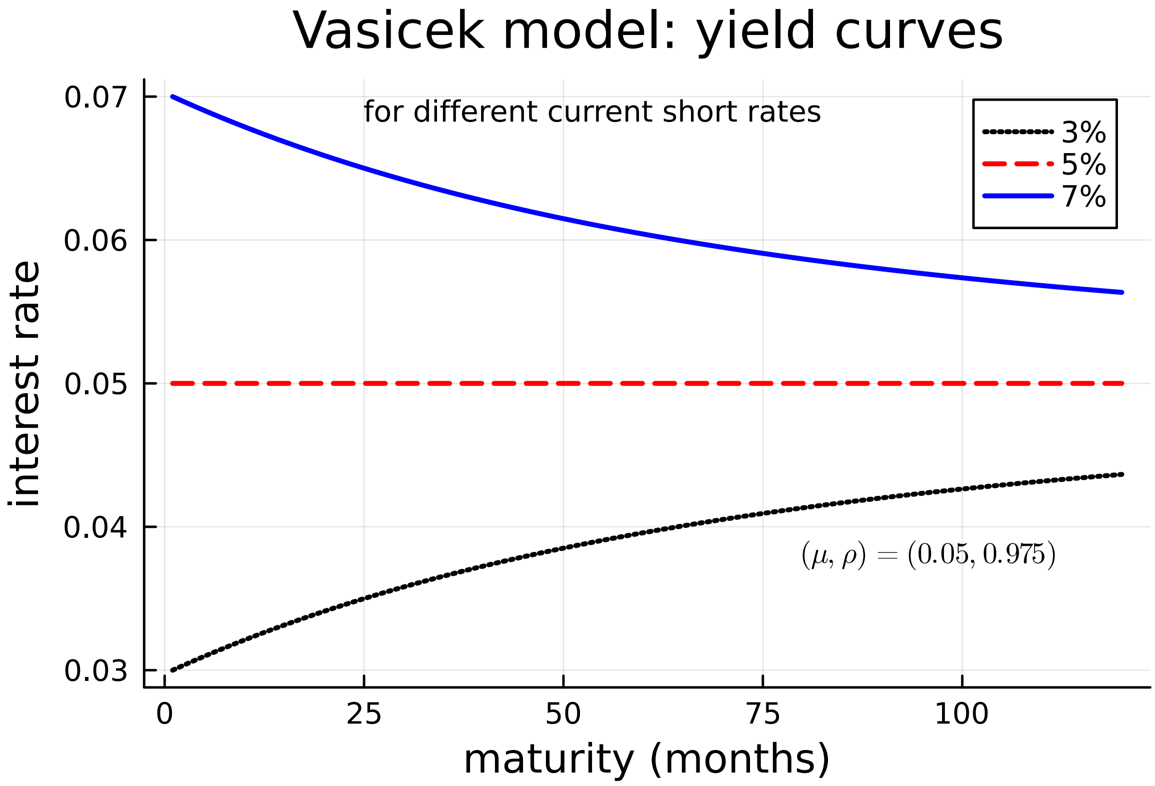
```
a and b for the first few horizons (months):
horizon      a*10         b
```

```
1            0.000     1.000
2            0.006     0.988
3            0.012     0.975
4            0.018     0.963
```

```
p1 = plot( s_range,ab.*[10 1],
           linecolor = [:black :red],
           linestyle = [:dot :solid],
           linewidth = 2,
           label = [L"a(n) \times 10" L"b(n)"],
           title = "Vasicek model: coefficients (scaled)",
           xlabel = "maturity (months)",
           annotation = [(80,0.2,txt),
                    (25,0.95,text(L"y(n) = a(n) + b(n)r",8,:left))] )
display(p1)
```



```
p1 = plot( s_range,y,
           linecolor = [:black :red :blue],
           linewidth = 2,
           linestyle = [:dot :dash :solid],
           label = ["3%" "5%" "7%"],
           title = "Vasicek model: yield curves",
           xlabel = "maturity (months)",
           ylabel = "interest rate",
           annotation = [(80,0.038,txt),
                    (25,0.069,text("for different current short rates",8,:left))] )
display(p1)
```



Hedging Using the Vasicek Model

The change of the (value of the) overall portfolio is

$\Delta V = v\Delta P_H - \Delta P_L$

The code below calculates the v value that makes $\Delta V \approx 0$, that is, $v = \Delta P_L / \Delta P_H$.

Information about the Two Bonds

In the example below, the liability pays 0.2 every 12 months for 10 years. The hedge bond is a 3-year zero coupon bond.

```
(cfL,mL) = (fill(0.2,10),1:10)     #liability, 0.2 in year 1,2...,10

(cfH,mH) = (1,3);                   #hedge bond, 3-year zero-coupon bond
```

Calculate Yield Curves

at two different values of the current short rate: r_0 and r_1. This is similar to the previous figure on the Vasicek yield curves, but here for the horizons that we need in order to (later) calculate the prices of the liability and hedge bond.

Motice that the periods in the Vasicek model are months, that is, corresponding to the maturity of the short rate. This means, for instance, that the 3-year spot rate is element 36 of the `y₀` vector calculated below.

```
nMax = maximum([mL;mH])*12        #number of months to calculate spot rates for
ρ = 0.975                         #experiment with this
μ = 0.05

r₀ = 0.05                         #initial (day 0) short interest rate
r₁ = 0.03                         #another possible short rate

nM   = 1:nMax                     #time to maturity (months)

y₀ = [Vasicek(r₀,ρ,μ,n)[1] for n in nM]    #yield curve starting at r₀
y₁ = [Vasicek(r₁,ρ,μ,n)[1] for n in nM]    #yield curve starting at r₁
b  = [Vasicek(r₁,ρ,μ,n)[3] for n in nM]    #b, from y=a*br

nL = indexin(mL,nM/12)    #indices in nM (months) that fit with mL (years), used later
nH = indexin(mH,nM/12)

printred("similar to the previous figure of yield curves. Plot y₀ and y₁ to verify that.")
```

```
similar to the previous figure of yield curves. Plot y₀ and y₁ to verify that.
```

Brute Force Calculation of Hedge Ratio

As a preliminary exercise, we calculate the bond prices (of both the liability and the hedge bond) at the initial yield curve and also at the new yield curve,

$$P = \sum_{k=1}^{K} \frac{cf_k}{\exp[m_k y(m_k)]},$$

where $y(m_k)$ is the continuously compounded (annualized) interest rate on a bond maturing in m_k years.

In a second step, we calculate a hedge ratio (v) as

$$v = \Delta P_L / \Delta P_H,$$

where ΔP is the change in the bond price (as the short rate changes).

Remark on the Code

- `y₀[nL]` picks out those (monthly) maturities in the yield curve that correspond to the annual maturities `mL` used in the definition of the liability cash flow.

```
PL₀ = BondPrice(y₀[nL],cfL,mL;ContRateQ=true)       #liability, before, /12 to get years
PL₁ = BondPrice(y₁[nL],cfL,mL;ContRateQ=true)       #after
ΔPL = PL₁ - PL₀                                     #change of liability value

PH₀ = BondPrice(y₀[nH],cfH,mH;ContRateQ=true)       #hedge bond
PH₁ = BondPrice(y₁[nH],cfH,mH;ContRateQ=true)
ΔPH = PH₁ - PH₀                                     #change of hedge bond value
v = ΔPL/ΔPH

printblue("Bond prices (according to the Vasicek model) at different r values")
xy = [PL₀ PH₀;PL₁ PH₁;(PL₁-PL₀) (PH₁-PH₀)]
printmat(xy;colNames=["PL","PH"],rowNames=["at $r₀","at $r₁","Δ"])

printlnPs("Hedge ratio implied by (finite difference) approach")
xy = [v;v*PH₀/PL₀]
printmat(xy,rowNames=["v","v*PH₀/PL₀"])
```

```
Bond prices (according to the Vasicek model) at different r values
               PL         PH
at 0.05     1.535      0.861
at 0.03     1.609      0.896
Δ           0.074      0.035
```

```
Hedge ratio implied by (finite difference) approach
v              2.106
v*PH₀/PL₀      1.181
```

Hedge Ratio from Spot Durations

We now apply a spot duration approach instead.

The `BondDurationSpot()` function calculates spot rate durations, D_k for $k = 1, ..., K$, that is, derivatives of the bond price wrt. the (vector of) spot rates. This can be used to calculate (approximate) bond price values as

$\Delta P = -P[\sum_{k=1}^{K} D_k b(m_k)]\Delta r,$

where $b(m_k)$ is the coefficient in the Vasicek model for horizon m_k. As before, remember that the output from the Vasicek model is indexed by months (not years). Apply the equation to the liability and to the hedge bond (which have different D_k values).

As before, the hedge ratio is calculated as

$v = \Delta P_L / \Delta P_H,$

so the Δr terms cancel.

```
"""
    BondDurationSpot(Y,cf,m;ContRateQ=false)

Calculate (modified) spot rate durations.

# Input
- `Y::Vector`:            spot rates
- `cf::Vector`:           cash flows
- `m::Vector`:            time to cash flows
- `P::Number`:            bond price
- `ContRateQ::Bool`:      if `true`: continuously compounded rates

"""
function BondDurationSpot(Y,cf,m;ContRateQ=false)
  P = BondPrice(Y,cf,m;ContRateQ)         #implied bond price
  if ContRateQ
    D = m.*cf.*exp.(-m.*Y)
  else
    D = m.*cf./(1.0.+Y).^(m.+1)
  end
  D = D/P
  return D, P
end
```

```
BondDurationSpot
```

```
(DurSpotL,PL) = BondDurationSpot(y₀[nL],cfL,mL;ContRateQ=true)
ΔPL = -PL₀*sum(b[nL].*DurSpotL)
(DurSpotH,PH) = BondDurationSpot(y₀[nH],cfH,mH;ContRateQ=true)
ΔPH = -PH₀*sum(b[nH].*DurSpotH)
v = ΔPL/ΔPH

printblue("Hedge ratio from spot durations combined with the Vasicek model")
xy = [v;v*PH₀/PL₀]
printmat(xy,rowNames=["v","v*PH₀/PL₀"])

printblue("The hedge ratio is similar (but not identitical)
to the one obtained by the 'brute force' approach.\n")

printred("Notice: ρ is important for the hedge ratio v.
Try also ρ=1 (change in one of the first cells in the notebook)
to see how it affects the hedge ratio v.")
```

```
Hedge ratio from spot durations combined with the Vasicek model
v              2.093
v*PH₀/PL₀      1.174

The hedge ratio is similar (but not identitical)
to the one obtained by the 'brute force' approach.

Notice: ρ is important for the hedge ratio v.
Try also ρ=1 (change in one of the first cells in the notebook)
to see how it affects the hedge ratio v.
```

Options 1

This notebook shows profit functions and pricing bounds for options.

Load Packages and Extra Functions

```
using Printf

include("src/printmat.jl");
```

```
using Plots, LaTeXStrings
default(size = (480,320),fmt = :png)
```

Profits of Options

Let K be the strike price, S_m the price of the underlying at expiration (m years ahead) of the option contract and y the continously compounded interest rate.

The call and put profits (at expiration) are

call profit$_m$ $= \max(0, S_m - K) - e^{my}C$

put profit$_m$ $= \max(0, K - S_m) - e^{my}P$,

where C and P are the call and put option prices (paid today). In both cases the first term ($\max()$) represents the payoff at expiration, while the second term ($e^{my}C$ or $e^{my}P$) subtracts the capitalised value of the option price (premium) paid at inception of the contract.

The profit of a straddle is the sum of those of a call and a put.

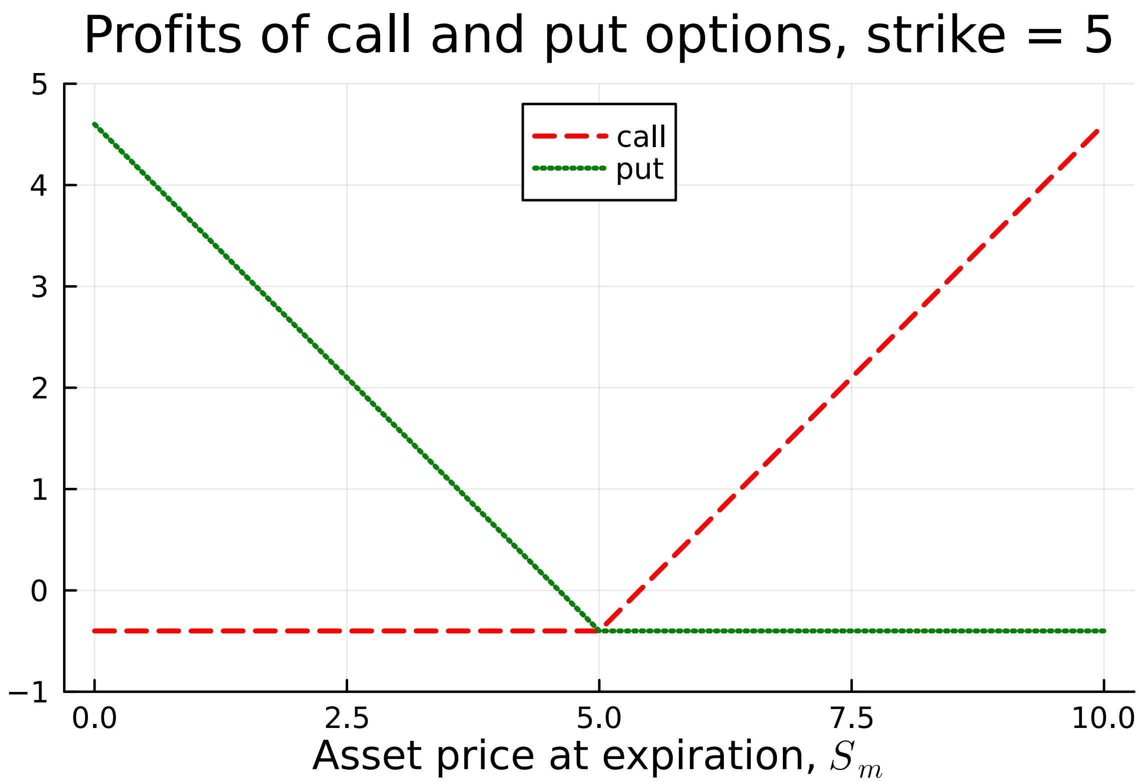
A Remark on the Code

- `Sₘ_range` is a vector (or range) of S_m values. The idea is to show the payoff (or profit) at different possible outcomes of the final price of the underlying.
- `Sₘ_range .> K` creates a vector of false/true. Notice that the dot (`.`) is needed to compare each element in `Sₘ_range` to the number `K`.
- `ifelse.(Sₘ_range.>K,"yes","no")` creates a vector of "yes" or "no" depending on whether `Sₘ_range.>K` or not. This is one of several possible ways of writing an `if` statement
- To get LaTeX in the graphs use `L"some text, $\cos x$"` (Notice the `L` which indicates that this is a string with some LaTeX.)

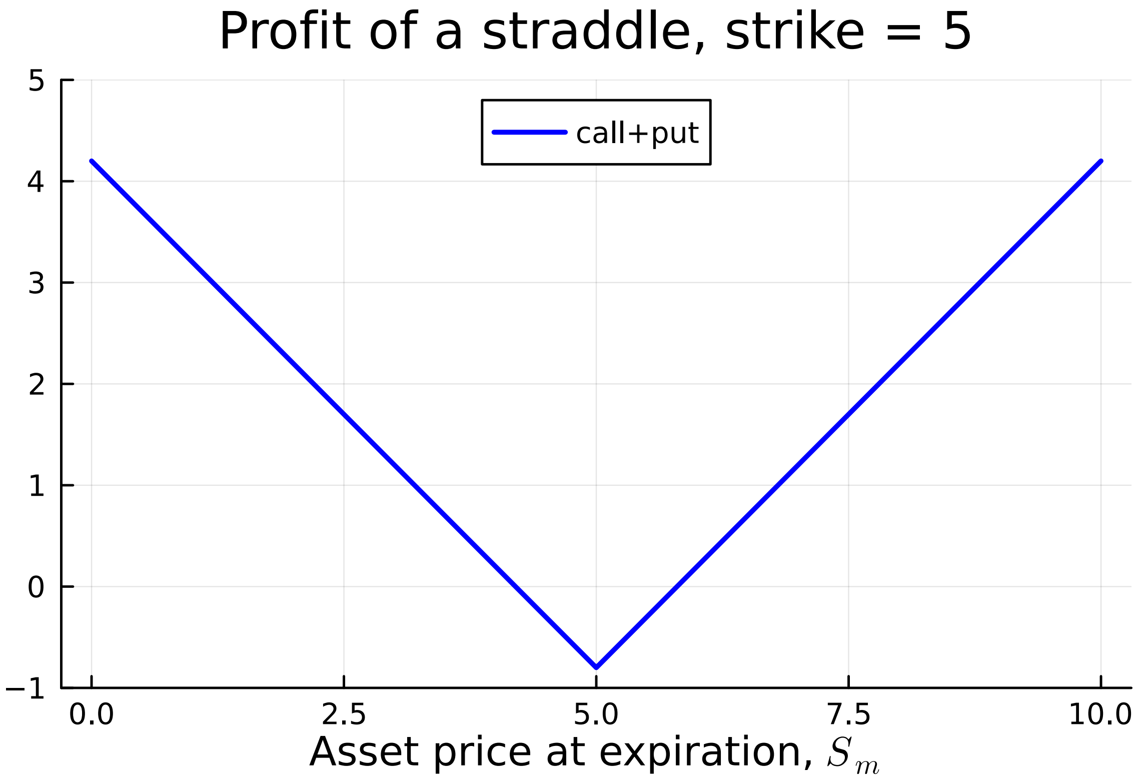
```
Sₘ_range = [4.5,5.5]   #possible values of underlying at expiration
K   = 5                #strike price
C   = 0.4              #call price (just a number that I made up)
P   = 0.4              #put price
(y,m) = (0,1)          #zero interest to keep it simple, 1 year to expiration

CallPayoff = max.(0,Sₘ_range.-K)                #payoff at expiration
CallProfit = CallPayoff .- exp(m*y)*C     #profit at expiration

ExerciseIt  = ifelse.(Sₘ_range.>K,"yes","no")  #"yes"/"no" for exercise

printblue("Payoff and profit of a call option with strike price $K, price (premium) of $C and inter
printmat(Sₘ_range,ExerciseIt,CallPayoff,CallProfit;colNames=["Sₘ","Exercise","Payoff","Profit"])
```

```
Payoff and profit of a call option with strike price 5, price (premium) of 0.4 and interest rate 0:

        Sₘ  Exercise    Payoff    Profit
     4.500        no     0.000    -0.400
     5.500       yes     0.500     0.100
```

```
Sₘ_range = 0:0.1:10          #more possible outcomes, for plotting

CallProfit = max.(0,Sₘ_range.-K) .- exp(m*y)*C
PutProfit  = max.(0,K.-Sₘ_range) .- exp(m*y)*P

p1 = plot( Sₘ_range,[CallProfit PutProfit],
           linecolor = [:red :green],
           linestyle = [:dash :dot],
           linewidth = 2,
           label = ["call" "put"],
           ylim = (-1,5),
           legend = :top,
           title = "Profits of call and put options, strike = $K",
           xlabel = L"Asset price at expiration, $S_m$" )
display(p1)
```



```
StraddleProfit = CallProfit + PutProfit   #a straddle: 1 call and 1 put

p1 = plot( Sₘ_range,StraddleProfit,
           linecolor = :blue,
           linewidth = 2,
           label = "call+put",
           ylim = (-1,5),
           legend = :top,
           title = "Profit of a straddle, strike = $K",
           xlabel = L"Asset price at expiration, $S_m$" )
display(p1)
```



Put-Call Parity for European Options

A no-arbitrage condition says that

$C - P = e^{-my}(F - K)$

must hold, where F is the forward price (on a forward contract with the same time to expiration, m, as the option). This is the *put-call parity*.

When the underlying asset has no dividends (until expiration of the option), then the forward-spot parity says that $F = e^{my}S$, which can be used in the put-call partity to substitute for F.

```
(S,K,m,y) = (42,38,0.5,0.05)    #current price of underlying etc

C = 5.5                   #assume this is the price of a call option(K)

F = exp(m*y)*S            #forward-spot parity
P = C - exp(-m*y)*(F-K)   #put price implied by the parity
```

```
printblue("Put-Call parity when (C,S,y,m)=($C,$S,$y,$m):\n")

printmat([C,exp(-m*y),F-K,P],rowNames=["C","exp(-m*y)","F-K","P (implied)"])
```

```
Put-Call parity when (C,S,y,m)=(5.5,42,0.05,0.5):

C                5.500
exp(-m*y)        0.975
F-K              5.063
P (implied)      0.562
```

Pricing Bounds

The pricing bounds for (American and European) call options are

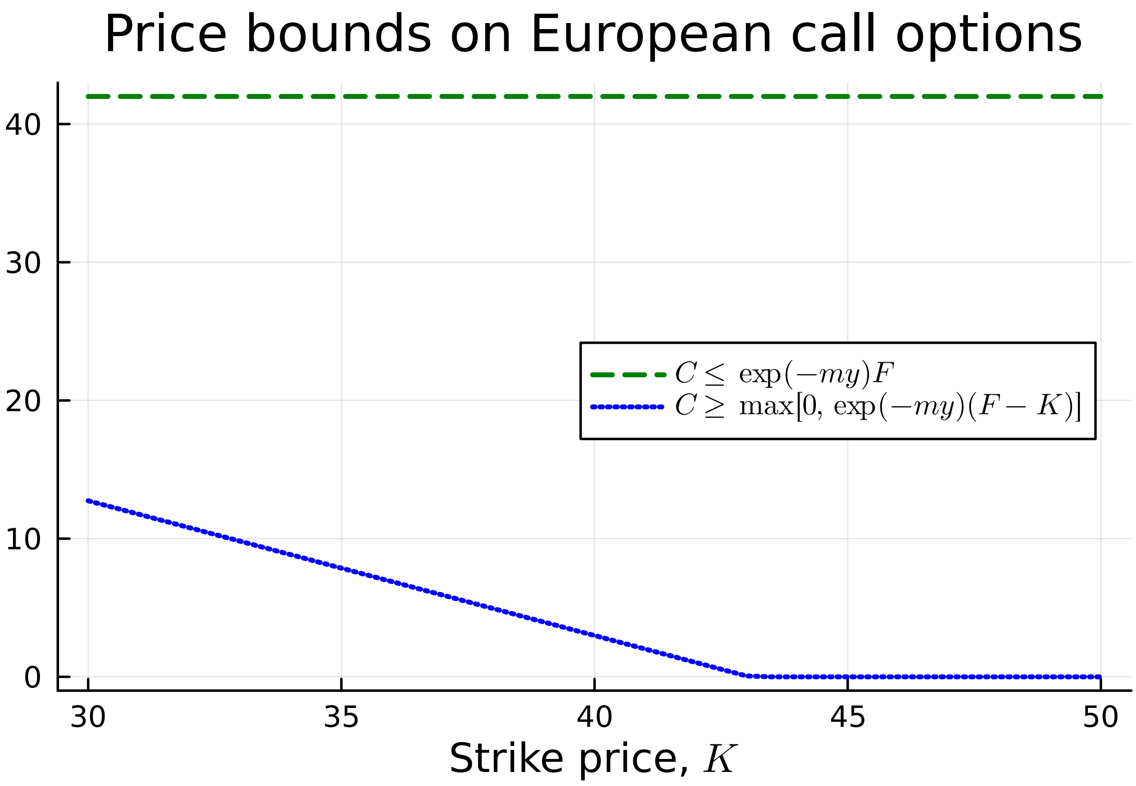
$\max[0, e^{-my}(F-K)] \leq C \leq e^{-my}F$

We plot these bounds as functions of the strike price K.

```
(S,K,m,y) = (42,38,0.5,0.05)        #current price of underlying etc

F = exp(m*y)*S

C_Upper = exp(-m*y)*F
C_Lower = max.(0,exp(-m*y)*(F-K))  #pricing bounds for a (single) strike price

printlnPs("Pricing bounds for European call option with strike $K: ",C_Lower,C_Upper)
```

```
Pricing bounds for European call option with strike 38:      4.938    42.000
```

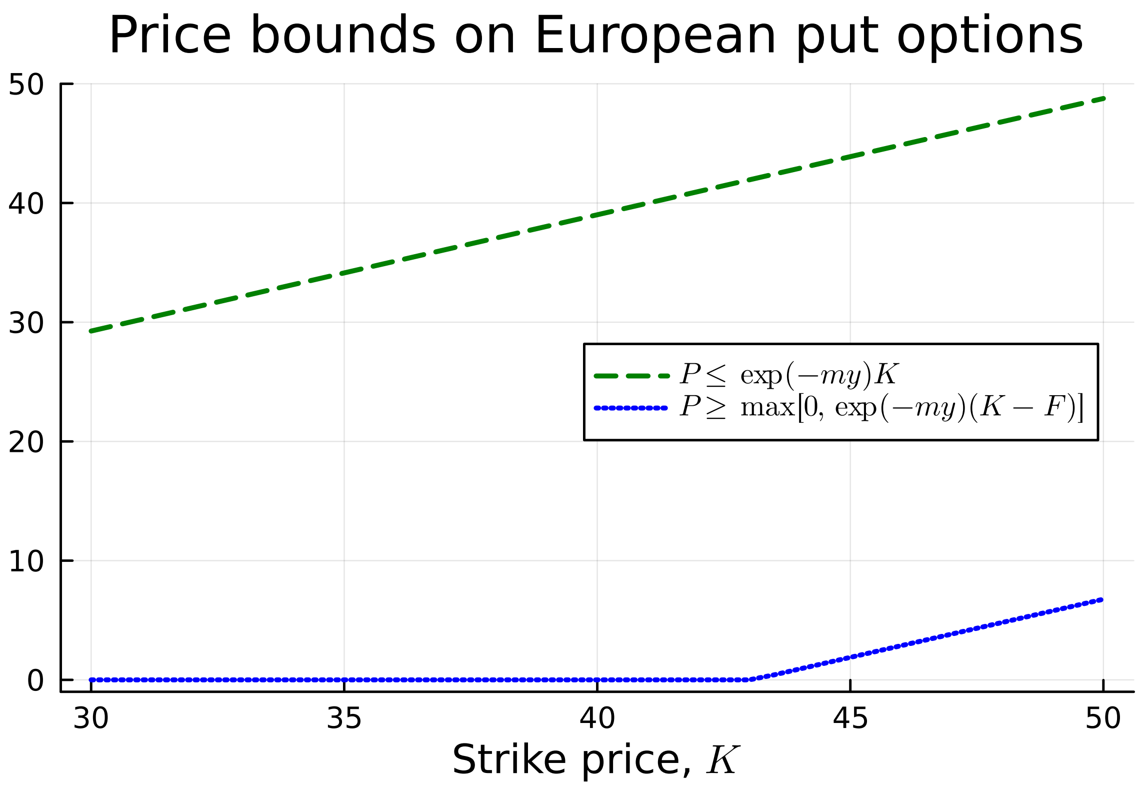
```
K_range = 30:0.5:50             #pricing bounds for many strike prices
n = length(K_range)

C_Upper = exp(-m*y)*F
C_Lower = max.(0,exp(-m*y)*(F.-K_range));

p1 = plot( K_range,[C_Upper*ones(n) C_Lower],
           linecolor = [:green :blue],
           linewidth = 2,
           linestyle = [:dash :dot],
           label = [L"C \leq \exp(-my)F " L"C \geq \max[0,\exp(-my)(F-K)]"],
           ylim = (-1,S+1),
           legend = :right,
           title = "Price bounds on European call options",
           xlabel = L"Strike price, $K$" )
display(p1)
```



The pricing bounds for (European) put options are

$e^{-my}(K-F) \leq P_E \leq e^{-my}K$

```
P_Upper = exp(-m*y)*K_range
P_Lower = max.(0,exp(-m*y)*(K_range.-F))

p1 = plot( K_range,[P_Upper P_Lower],
           linecolor = [:green :blue],
           linewidth = 2,
           linestyle = [:dash :dot],
           label = [L" P \leq \exp(-my)K " L"P \geq \max[0,\exp(-my)(K-F)]"],
           ylim = (-1,50),
           legend = :right,
           title = "Price bounds on European put options",
           xlabel = L"Strike price, $K$" )
display(p1)
```



Options 2: The Binomial Option Pricing Model

This notebook implements the binomial option pricing model for European and American style options. The appraoch is to set up, fill and store all important arrays (price of underlying asset, exercise decision, option value…in all periods and states) so we can analyze them. Code aimed at performance (rather than teaching) would avoid that.

Load Packages and Extra Functions

The OffsetArrays.jl package allows flexible indexing of an array. In particular, it allows us to refer to the first element of a vector as element 0. This is useful in order to stick close to lecture notes on binomial trees where period 0 is the starting point and we take n time steps.

```
using Printf, OffsetArrays

include("src/printmat.jl");
```

```
using Plots
default(size = (480,320),fmt = :png)
```

The Binomial Model for *One* Time Step

In a binomial model the price of the underlying asset can change from S today to either Su or Sd in the next period (which is h years from now).

Let f_u and f_d be the values of the derivative in the up- and down states (next period). Then, the value of the derivative today (f) is

$f = e^{-yh}\left[pf_u + (1-p)f_d\right]$ where $p = \frac{e^{yh}-d}{u-d}$

Notice that p denotes the "risk-neutral probability" of an up move. (Don't mix it with notation for a put price.)

As an example, for a call option that expires next period, the payoffs in the two states are

$f_u = \max(Su - K, 0)$

$f_d = \max(Sd - K, 0)$

For these equations to make sense, we need $u > e^{yh} > d$, which needs to be checked in any numerical implementation.

```
(S,K,y) = (10,10,0)             #underlying price today, strike pricem interest rate
(u,d,h) = (1.1,0.95,1/12)       #up move, down move, length of time period

fu = max(S*u-K,0)               #value of call option in up node
fd = max(S*d-K,0)               #in down node

p = (exp(y*h)-d)/(u-d)         #risk-neutral probability of "up" move
CallPrice = exp(-y*h)*(p*fu+(1-p)*fd)  #call option price today

printblue("Pricing of a call option with strike $K, one time step:\n")
printmat([fu,fd,p,CallPrice];rowNames=["Payoff 'up'","Payoff 'down'","p","Call price now"])
```

```
Pricing of a call option with strike 10, one time step:

Payoff 'up'        1.000
Payoff 'down'      0.000
p                  0.333
Call price now     0.333
```

A CRR Tree for *Many* (Short) Time Steps

We now build a tree with n steps of (time), each of length h, to reach expiration m (so $nh = m$).

The CRR approach to construct (u, d) is

$u = e^{\sigma\sqrt{h}}$ and $d = 1/u$,

where σ is the annualized standard deviation of the return on the underlying asset. Notice that p depends on the choice of (u, d) and also on e^{yh}

$p = \frac{e^{yh}-d}{u-d}$.

With these choices, $u > e^{yh} > d$ holds if h is sufficiently small.

```
"""
CRRparams(σ,m,n,y)

    BOPM parameters according to CRR
"""
function CRRparams(σ,m,y,n)
    h = m/n                    #time step size (in years)
    u = exp(σ*sqrt(h))         #up move
    d = 1/u                    #down move
    p = (exp(y*h) - d)/(u-d) #rn prob of up move
    return h,u,d,p
end
```

```
CRRparams
```

```
(m,y,σ,n) = (0.5,0.05,0.2,50)
(h,u,d,p) = CRRparams(σ,m,y,n)

printblue("CRR parameters in tree when m=$m, y=$y, σ=$σ and n=$n:\n")
xx = [h,u,d,p,exp(y*h)]
printmat(xx;rowNames=["h","u","d","p","exp(yh)"])

printblue("Checking if u > exp(y*h) > d: ", u > exp(y*h) > d)
```

```
CRR parameters in tree when m=0.5, y=0.05, σ=0.2 and n=50:

h            0.010
u            1.020
d            0.980
p            0.508
exp(yh)      1.001

Checking if u > exp(y*h) > d: true
```

Build a Tree for the Underlying Asset

The next few cells explain how we build a tree with n time steps for the underlying asset.

Creating a Vector of Vectors

of different lengths.

A Remark on the Code

The next cell illustrates how we can create a vector of vectors (of different lengths). We use the OffsetArrays.jl package to set up the outer vector so that the first element has index 0. All the inner vectors are traditional, that is, the first index is 1.

In this example, `x[0]` is a vector `[0]`, while `x[1]` is a vector `[0,0]`, etc.

Using a vector of vector is more straightforward than filling half a matrix, and wastes less memory space.

```
x = [zeros(i+1) for i = 0:2]       #a vector or vectors (of different lengths)
x = OffsetArray(x,0:2)             #convert so the indices of x are 0:2

printblue("Illustrating a vector of vectors (of different lengths):\n")
for i in 0:2
    printblue("x[$i]:")
    printmat(x[i])
end
```

```
Illustrating a vector of vectors (of different lengths):

x[0]:
     0.000

x[1]:
     0.000
     0.000

x[2]:
     0.000
     0.000
     0.000
```

Steps (Up and Down)

We create a tree by starting at the current spot price S. The subsequent nodes are then created by multiplying by u or d. That is, the tree (vector of vectors) is built from the first (0) time step to the last (n) time step.

step $0 : (S)$ as `STree[0]`

step $1 : (Su, Sd)$ as `STree[1]`

step $2 : (Suu, Sud, Sdd)$ (since $Sud = Sdu$) as `STree[2]`

…

A Remark on the Code

Each time step is a vector and the vector of those vectors is the entire tree. For instance, for step 2, the code in the next cell creates the vector (Suu, Sud, Sdd) by first multiplying the step 1 vector by the up factor (to get $u(Su, Sd)$) and then attaching (as the last element) the last element of the step 1 vector times the down factor (to get dSd).

```
"""
    BuildSTree(S,n,u,d)

Build binomial tree, starting at `S` and having `n` steps with up move `u` and down move `d`

# Output
- `STree:: Vector of vectors`: each (sub-)vector is for a time step. `STree[0] = [S]` and `STree[n]

"""
function BuildSTree(S,n,u,d)
    STree = [fill(NaN,i) for i = 1:n+1]  #vector of vectors (of different lengths)
    STree = OffsetArray(STree,0:n)       #convert so the indices are 0:n
    STree[0][1] = S                      #step 0 is in STree[0], element 1
    for i in 1:n                          #move forward in time
        STree[i][1:end-1] = u*STree[i-1]       #most elements: up move from STree[i-1][1:end]
        STree[i][end]     = d*STree[i-1][end]  #last element: down move from STree[i-1][end]
    end
    return STree
end
```

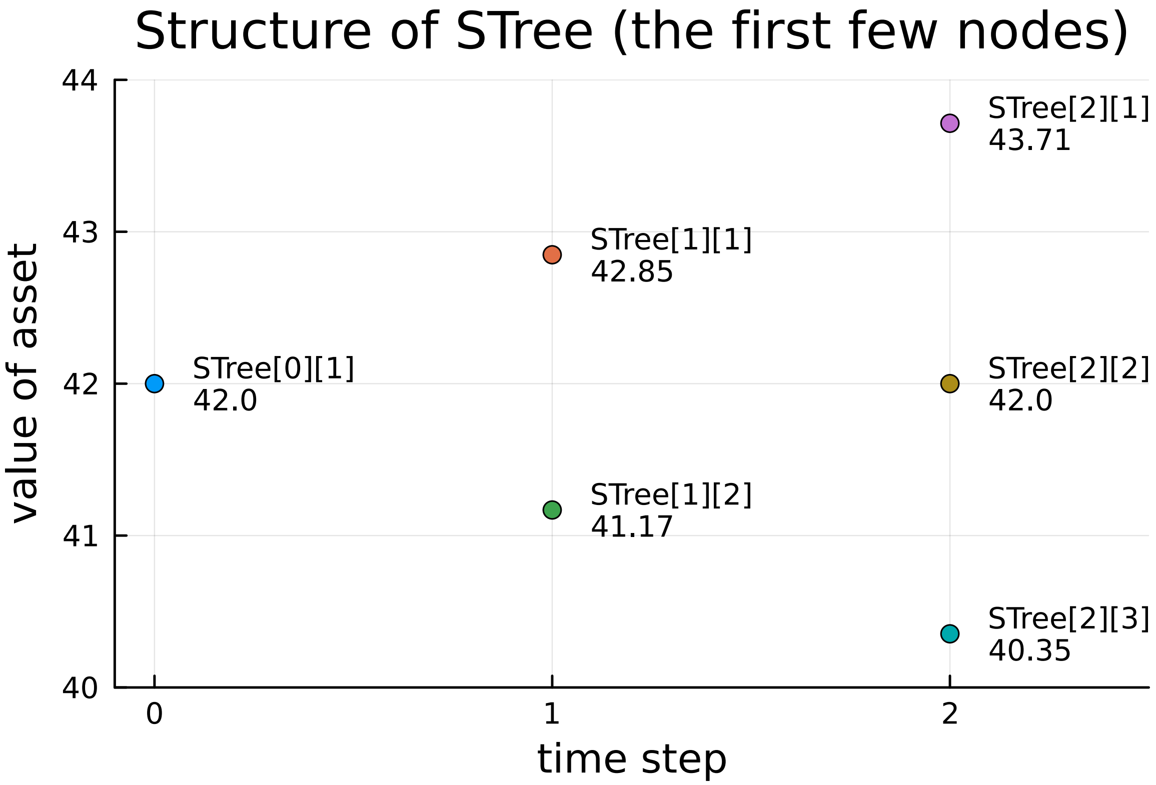
```
BuildSTree
```

```
S     = 42.0
STree = BuildSTree(S,n,u,d);
```

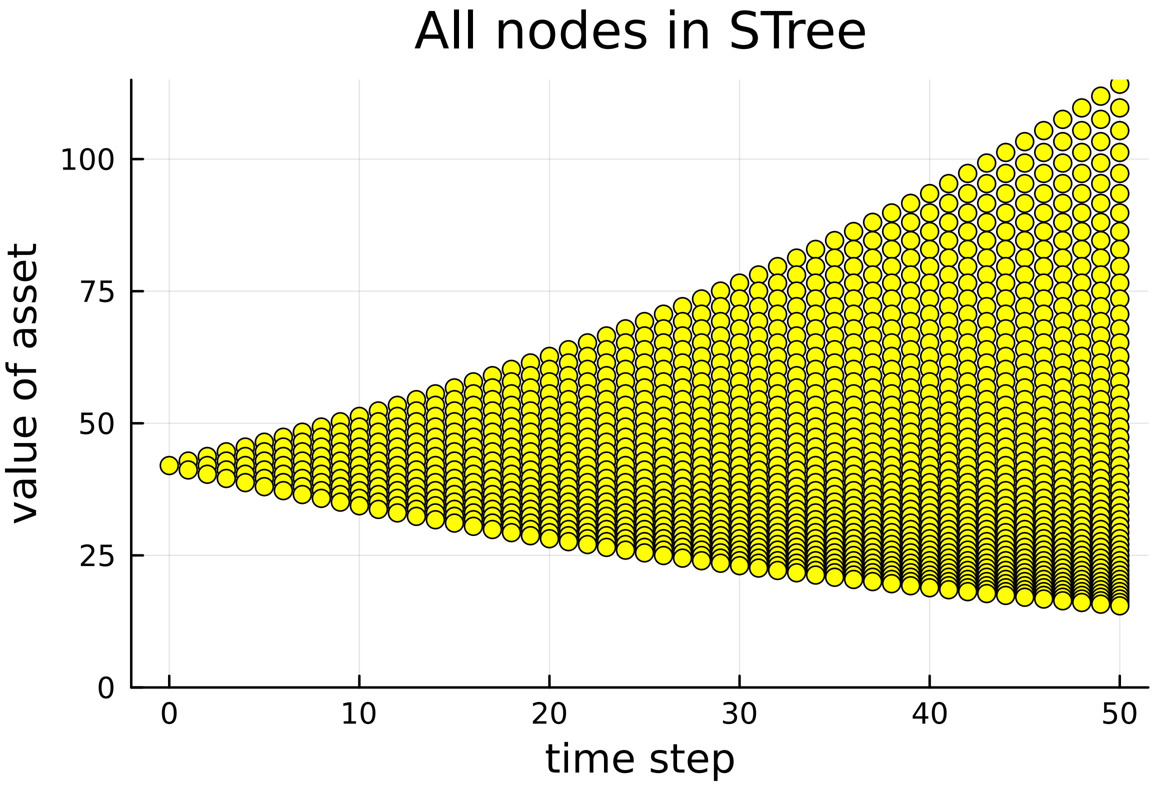
Showing the Tree for the Underlying Asset

The next few cells illustrate the tree. Notice that `STree[i][1]` is the highest value of the underlying for time step `i`, and that `STree[i][2]` is the second highest, etc.

```
p1 = plot( legend = false,
           xlim = (-0.1,2.5),
           ylim = (40,44),
           xticks = 0:2,
           title = "Structure of STree (the first few nodes)",
           xlabel = "time step",
           ylabel = "value of asset" )
for i in 0:2, j in 1:length(STree[i])
    local txt1,txt2,txt
    txt1 = "STree[$i][$j]"
    txt2 = string(round(STree[i][j],digits=2))
    txt  = text(string(txt1,"\n",txt2),8,:left)    #adding this info to plot
    scatter!( [i],[STree[i][j]],annotation = (i+0.1,STree[i][j],txt) )
end
display(p1)           #needed since plot() does not have data points to plot
```



```
p1 = scatter( [0.0],STree[0],
              markercolor = :yellow,
              xlim = (-2,n+1.5),
              ylim = (0,115),
              legend = false,
              title = "All nodes in STree",
              xlabel = "time step",
              ylabel = "value of asset" )

for i in 1:n
    scatter!(fill(i,i+1),STree[i],color=:yellow)
end
display(p1)
```



Calculating the Option Price

Let f_{ij} be the option price at time step i when the underlying price is S_{ij}. (We use S_{ij} as a shorthand notation, to avoid writing things like $Sudd$.)

For a European call option, the call price at the *last time step n* is $f_{nj} = \max(0, S_{nj} - K)$ for each node j. Similarly, the put price is $f_{nj} = \max(0, K - S_{nj})$.

For all *earlier time steps*, the value for a European option is

$f_{ij} = e^{-yh}[pf_{i+1,j} + (1-p)f_{i+1,j+1}]$.

In the code below, $f_{i+1,1}$ refers to the highest node (of the underlying) in time step $i + 1$, $f_{i+1,2}$ to the second highest in the same time step, etc. Notice that we calculate the option price from the last time step (n) to the first (0).

A Remark on the Code

- Inside the function `EuOptionPrice()`, `Value = similar(STree)` is used to create a new vector (of vectors) called `Value` with the same structure as `STree`.
- The `;isPut=false` creates a keyword argument which defaults to `false`. To calculate a put price, call the function as `EuOptionPrice(STree,K,y,h,p;isPut=true)`.
- The code `Value[n] = isPut ? max.(0,K.-STree[n]) : max.(0,STree[n].-K)` is a short form of an `if...else...end`.
- `only(Pₑ[0])` creates a scalar from an array with one element. (Similar to `Pₑ[0][1]` but with error checking.)

European Options

```
"""
    EuOptionPrice(STree,K,y,h,p;isPut=false)

Calculate price of European option from binomial model

# Output
- `Value:: Vector of vectors`: option values at different nodes, same structure as STree

"""
function EuOptionPrice(STree,K,y,h,p;isPut=false)
    Value    = similar(STree)                            #tree for derivative, to fill
    n        = length(STree) - 1                         #number of steps in STree
    Value[n] = isPut ? max.(0,K.-STree[n]) : max.(0,STree[n].-K)  #last time node
    for i in n-1:-1:0                                   #move backward in time
        Value[i] = exp(-y*h)*(p*Value[i+1][1:end-1] + (1-p)*Value[i+1][2:end])
    end                                           #p*up + (1-p)*down, discount
    return Value
end
```

```
EuOptionPrice
```

```
K = 42.0                                  #strike price

Pₑ = EuOptionPrice(STree,K,y,h,p;isPut=true)        #Pₑ[0] is a 1-element vector with the put price
Cₑ = EuOptionPrice(STree,K,y,h,p)

Cₑ_parity = only(Pₑ[0]) + S - exp(-m*y)*K           #put-call parity, only(Pₑ[0]) makes it a scalar
```

```
printblue("European option prices at K=$K and S=$S: ")
printmat([only(Pₑ[0]),only(Cₑ[0]),Cₑ_parity];rowNames=["put","call","call from parity"])
```

```
European option prices at K=42.0 and S=42.0:
put                  1.844
call                 2.881
call from parity     2.881
```

Routines for the Same Calculations, but Using Matrices to Store the Results (extra)

If you want to port the code to another language where a vector of vectors is tricky, then you might consider starting with the functions included below.

```
include("src/OptionsBopmMatrix.jl")
STreeM = BuildSTreeM(S,n,u,d)

println("first upper 4x4 block of the matrix:")
printmat(STreeM[1:4,1:4])

Pₑ₂ = EuOptionPriceM(STreeM,K,y,h,p;isPut=true)
printlnPs("put price: ",Pₑ₂[1,1])
```

```
first upper 4x4 block of the matrix:
    42.000    42.848    43.714    44.597
       NaN    41.168    42.000    42.848
       NaN       NaN    40.353    41.168
       NaN       NaN       NaN    39.554

put price:      1.844
```

American Options

The calculations are similar to thosw for and European option, except that that the option value is

$f_{ij} = \max(\text{value if exercised now, continuation value})$

The *continuation value* has the same form as in the European case, and thus assumes that the option has not been exercised before the next period.

The *value of exercising* now is $S_{ij} - K$ for a call and $K - S_{ij}$ for a put.

```
"""
    AmOptionPrice(STree,K,y,h,p;isPut=false)

Calculate price of American option from binomial model

# Output
- `Value:: Vector of vectors`: option values at different nodes, same structure as STree
- `Exerc:: Vector of vectors`: true if early exercise at the node, same structure as STree

"""
function AmOptionPrice(STree,K,y,h,p;isPut=false)     #price of American option
    Value = similar(STree)                            #tree for derivative, to fill
    n     = length(STree) - 1
    Exerc = similar(Value,BitArray)               #same structure as STree, but BitArray, empty
    Value[n] = isPut ? max.(0,K.-STree[n]) : max.(0,STree[n].-K)  #last time node
    Exerc[n] = Value[n] .> 0                      #exercise
    for i in n-1:-1:0                                    #move backward in time
        fa  = exp(-y*h)*(p*Value[i+1][1:end-1] + (1-p)*Value[i+1][2:end])
        Value[i] = isPut ? max.(K.-STree[i],fa) : max.(STree[i].-K,fa)    #put or call
        Exerc[i] = Value[i] .> fa                   #true if early exercise
    end
    return Value, Exerc
end
```

```
AmOptionPrice
```

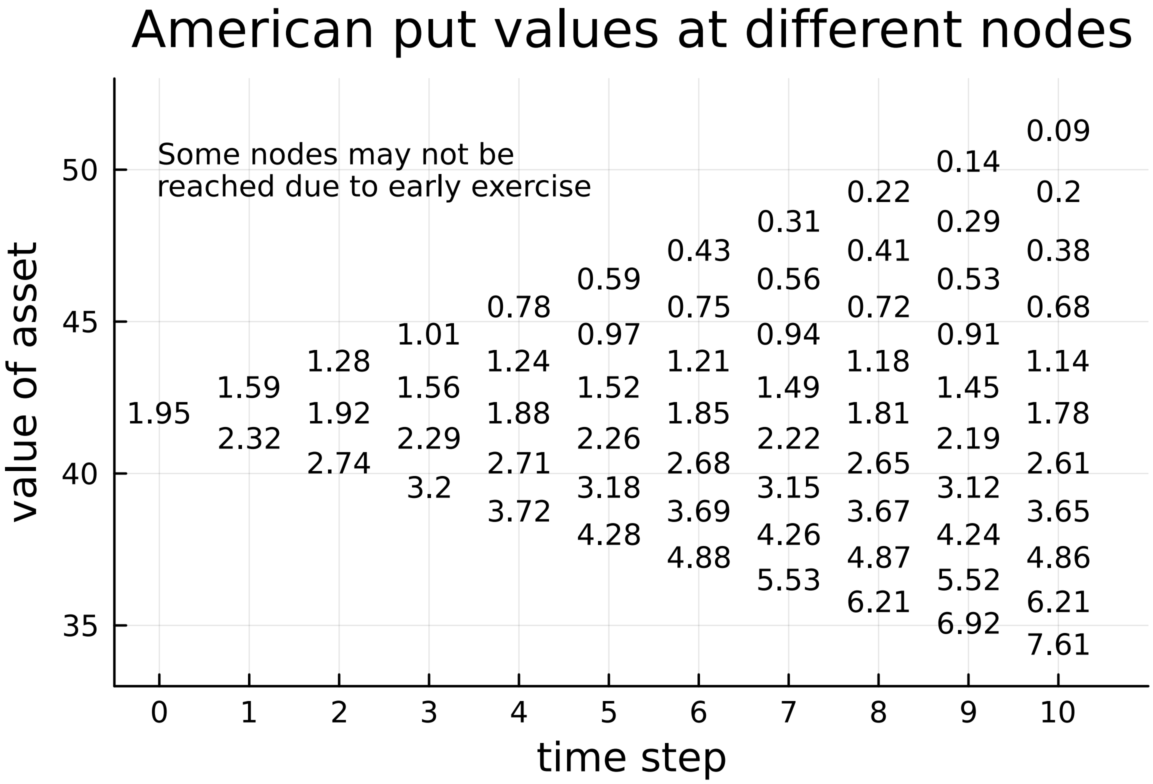
```
K = 42.0                                  #strike price

(Pₐ,Exerc)  = AmOptionPrice(STree,K,y,h,p;isPut=true)
(Cₐ,ExercC) = AmOptionPrice(STree,K,y,h,p)

printblue("Put and call prices at K=$K and S=$S: ")
xx = [ only(Pₐ[0]) only(Pₑ[0]);only(Cₐ[0]) only(Cₑ[0]) ]
printmat(xx;colNames=["American","European"],rowNames=["put","call"])

printred("When is the American option worth more? When the same?")
```

```
Put and call prices at K=42.0 and S=42.0:
      American  European
put      1.950     1.844
call     2.881     2.881
```

```
When is the American option worth more? When the same?
```

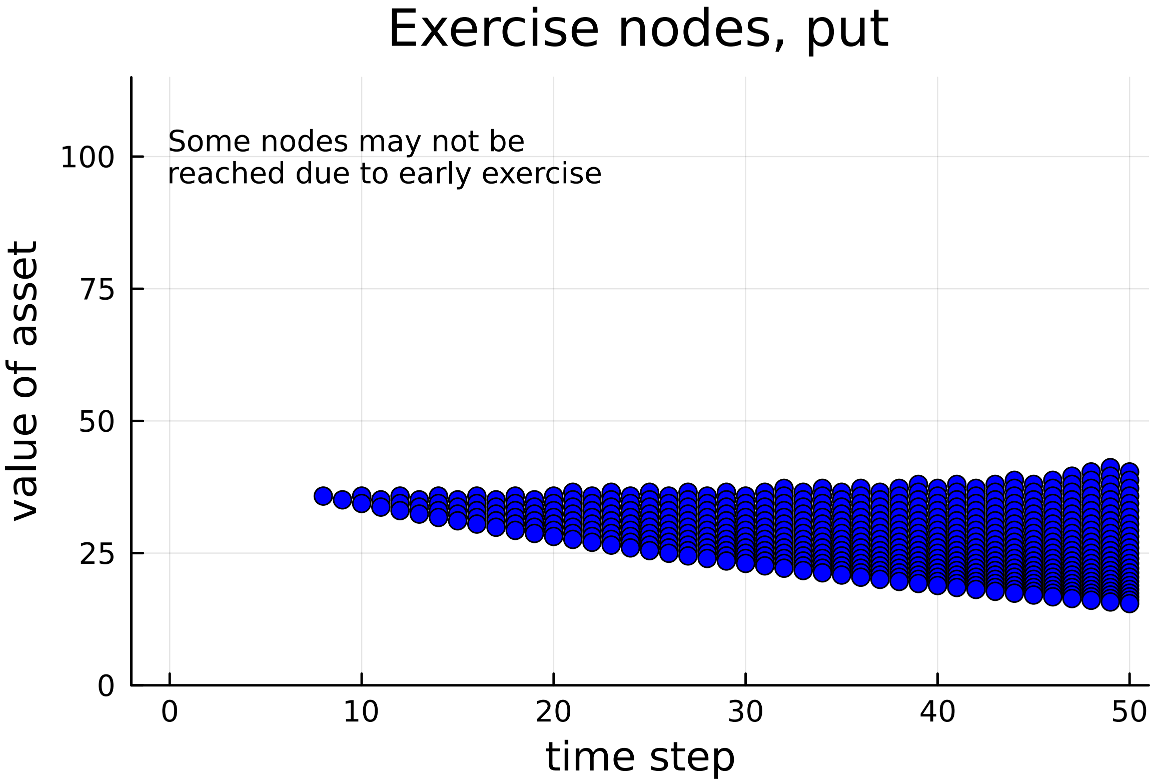
Plotting the Tree of American Option Values

```
p1 = plot( xlim = (-0.5,11),
           ylim = (33,53),
           xticks = 0:10,
           legend = false,
           title = "American put values at different nodes",
           xlabel = "time step",
           ylabel = "value of asset",
           annotation = (0,50,text("Some nodes may not be\nreached due to early exercise",8,:left)))
for i in 0:10, j in 1:length(STree[i])
    annotate!(i,STree[i][j],text(string(round(Pₐ[i][j],digits=2)),8))
end
display(p1)
```

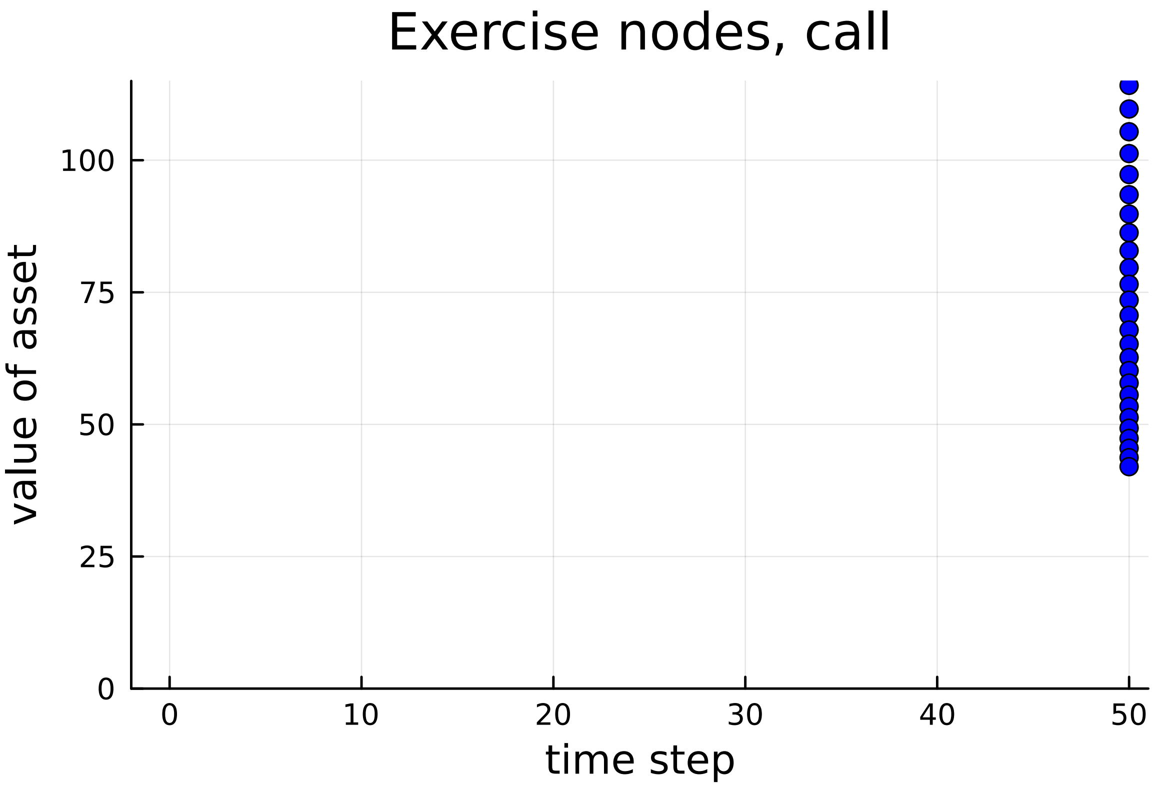


Plotting where Exercise Happens

```
p1 = plot( xlim = (-2,n+1),
           ylim = (0,115),
           legend = false,
           title = "Exercise nodes, put",
           xlabel = "time step",
           ylabel = "value of asset",
           annotation = (0,100,text("Some nodes may not be\nreached due to early exercise",8,:left)
for i in 0:n, j in 1:length(STree[i])
    if Exerc[i][j]
        scatter!([i],[STree[i][j]],markercolor=:blue)
    end
end
display(p1)
```



```
p1 = plot( xlim = (-2,n+1),
           ylim = (0,115),
           legend = false,
           title = "Exercise nodes, call",
           xlabel = "time step",
           ylabel = "value of asset")
for i in 0:n, j in 1:length(STree[i])
    if ExercC[i][j]
        scatter!([i],[STree[i][j]],markercolor=:blue)
    end
end
display(p1)
```



Options 3: The Black-Scholes Model

This notebook introduces the Black-Scholes option pricing model. It also discusses *(a)* implied volatility; *(b)* how to calculate the Black-Scholes model by numerical integration and *(c)* how the binomial model converges to the Black-Scholes model.

Load Packages and Extra Functions

```julia
using Printf, Distributions, Roots, QuadGK

include("src/printmat.jl");
```

```julia
using Plots, LaTeXStrings
default(size = (480,320),fmt = :png)
```

Black-Scholes

The Black-Scholes formula for a European call option on an (underlying) asset with a continuous dividend rate δ is

$C = e^{-\delta m} S \Phi(d_1) - e^{-ym} K \Phi(d_2)$, where

$d_1 = \frac{\ln(S/K)+(y-\delta+\sigma^2/2)m}{\sigma\sqrt{m}}$ and $d_2 = d_1 - \sigma\sqrt{m}$

and where $\Phi(d)$ denotes the probability of $x \leq d$ when x has an $N(0,1)$ distribution. In other words, $\Phi(d)$ is the cumulative distribution function of the $N(0,1)$ distribution.

A Remark on the Code

- `Φ(x)= cdf(Normal(0,1),x)` is a one-line definition of a function.
- The `;isPut=false` creates a keyword argument which defaults to `false` which means that we calculate a call option price. To calculate a put option price, use the function as `OptionBlackSPs(...;isPut=true)`.

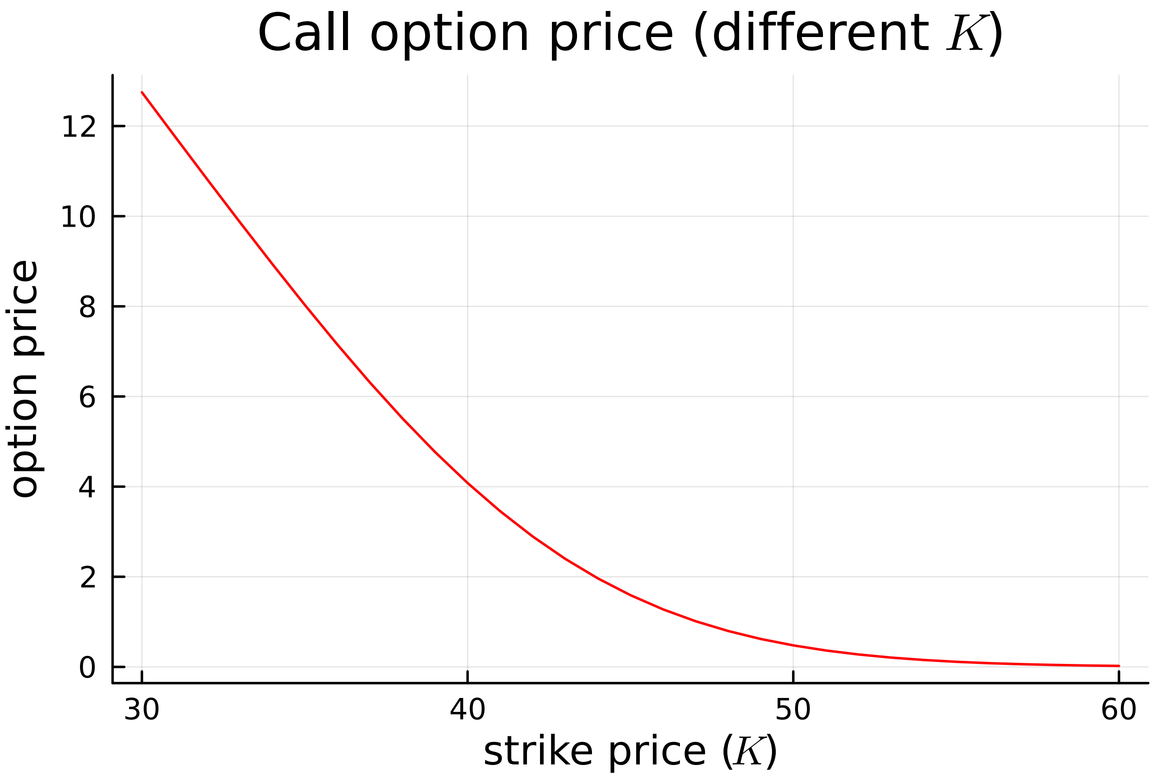
```
"""
    Φ(x)
Calculate Pr(z<=x) for N(0,1) variable z. Convenient short hand notation. Used in `OptionBlackSPs()
"""
Φ(x)= cdf(Normal(0,1),x)



"""
Calculate Black-Scholes European call or put option price, continuous dividends of δ
"""
function OptionBlackSPs(S,K,m,y,σ,δ=0;isPut=false)
    d1 = ( log(S/K) + (y-δ+0.5*σ^2)*m ) / (σ*sqrt(m))
    d2 = d1 - σ*sqrt(m)
    c  = exp(-δ*m)*S*Φ(d1) - K*exp(-y*m)*Φ(d2)
    price = isPut ? c - exp(-δ*m)*S + exp(-y*m)*K : c
    return price
end
```

```
OptionBlackSPs
```

```
(S,K,m,y,σ) = (42,42,0.5,0.05,0.2)           #some parameter values

C = OptionBlackSPs(S,K,m,y,σ)                #call
P = OptionBlackSPs(S,K,m,y,σ,0;isPut=true)   #put

δ = 0.03
Cδ = OptionBlackSPs(S,K,m,y,σ,δ)             #call, with dividends
Pδ = OptionBlackSPs(S,K,m,y,σ,δ;isPut=true)  #put

printblue("Option prices at S=$S and K=$K: ")
xx = [P Pδ;C Cδ]
printmat(xx;rowNames=["put","call"],colNames=["no dividends","δ=$δ"],width=15,colUnderlineQ=true)
```

```
Option prices at S=42 and K=42:
       no dividends         δ=0.03
       ------------         ------

put           1.856          2.121
call          2.893          2.532
```

```
K_range = 30:60           #different strike prices
C_K     = OptionBlackSPs.(S,K_range,m,y,σ)

p1 = plot( K_range,C_K,
           linecolor = :red,
           legend = false,
           title  = L"Call option price (different $K$)",
           xlabel = L"strike price ($K$)",
           ylabel = "option price" )
display(p1)
```



How the Black-Scholes Depends on Volatility

The Black-Scholes option price is an increasing function of the volatility (σ), as illustrated below.

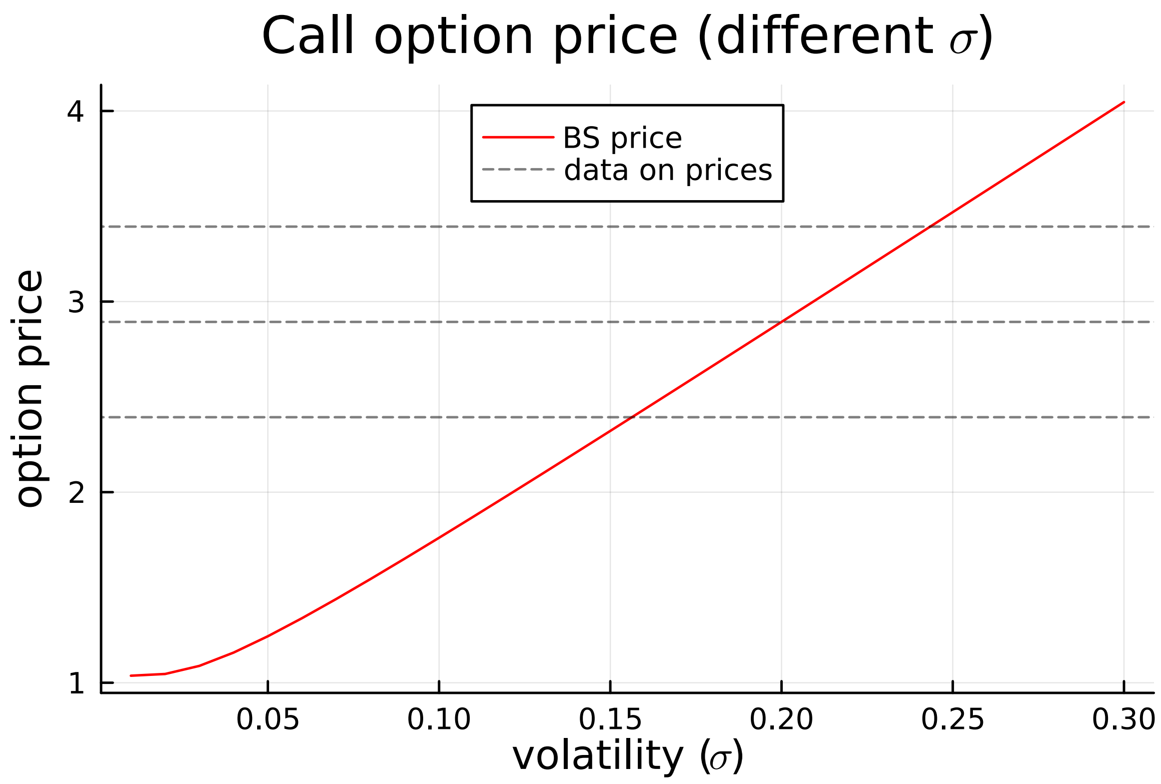
Implied Volatility

In the cell below we use an observed option price and solve the BS formula for σ. This is the "implied volatility," which could be interpreted as the market belief about (annualised) standard deviation of the underlying until expiration of the option. We do this for several option prices: the Black-Scholes price and also higher and lower prices.

A Remark on the Code

- To solve for the implied volatility, the next cells use the Roots.jl package.
- We solve for the root of the anonymous function `σ->OptionBlackSPs(S,K,m,y,σ)-C` which finds σ so that the model based price from `OptionBlackSPs()` minus the actual price `C` equals 0. We loop over a few different values of `C`.
- `iv = [... for Cᵢ in C_range]` creates a vector with one element per element in `C_range`.

```
σ_range = 0.01:0.01:0.3           #S,K,m,y are scalars, σ_range is a vector
C_σ     = OptionBlackSPs.(S,K,m,y,σ_range)    #at different σ values

p1 = plot( σ_range,C_σ,
           linecolor = :red,
           label = "BS price",
           title = L"Call option price (different $\sigma$)",
           xlabel = L"volatility ($\sigma$)",
           ylabel = "option price",
           legend = :top )
hline!( [C,C+0.5,C-0.5],linetype=:hline,linecolor=:black,line=(:dash,0.5),label="data on prices")
display(p1)
```



```
C_range = [C,C-0.5,C+0.05]     #B-S price, cheaper, more expensive

iv = [find_zero(σ->OptionBlackSPs(S,K,m,y,σ)-Cᵢ,(0,5)) for Cᵢ in C_range]  #create a vector of resu
                                                                           #for value in C_range
printblue("implied volatility:")
printmat(iv;rowNames=["Benchmark","Cheap option","Expensive option"])

printred("Compare the results with the previous graph\n")
```

```
implied volatility:
Benchmark            0.200
Cheap option         0.156
Expensive option     0.204

Compare the results with the previous graph
```

BS from an Explicit Integration

The price of a European a call option is

$C = e^{-ym} \mathrm{E}^* \max(0, S_m - K)$,

which can be written

$C = e^{-ym} \int_K^\infty \max(0, S_m - K) f^*(S_m) dS_m$,

where $f^*(S_m)$ is the risk neutral density function of the asset price at expiration (S_m). The integration starts at $S_m = K$ since $\max(0, S_m - K) = 0$ for $S_m < K$. See the figure below.

In the Black-Scholes model, the risk neutral distribution of $\ln S_m$ is

$\ln S_m \sim^* \mathrm{N}(\ln S + my - m\sigma^2/2, m\sigma^2)$,

where S is the current asset price. This means that $f^*(S_m)$ is the pdf of a lognormally distributed variable. Notice that this is the pdf of S_m, not its logarithm.

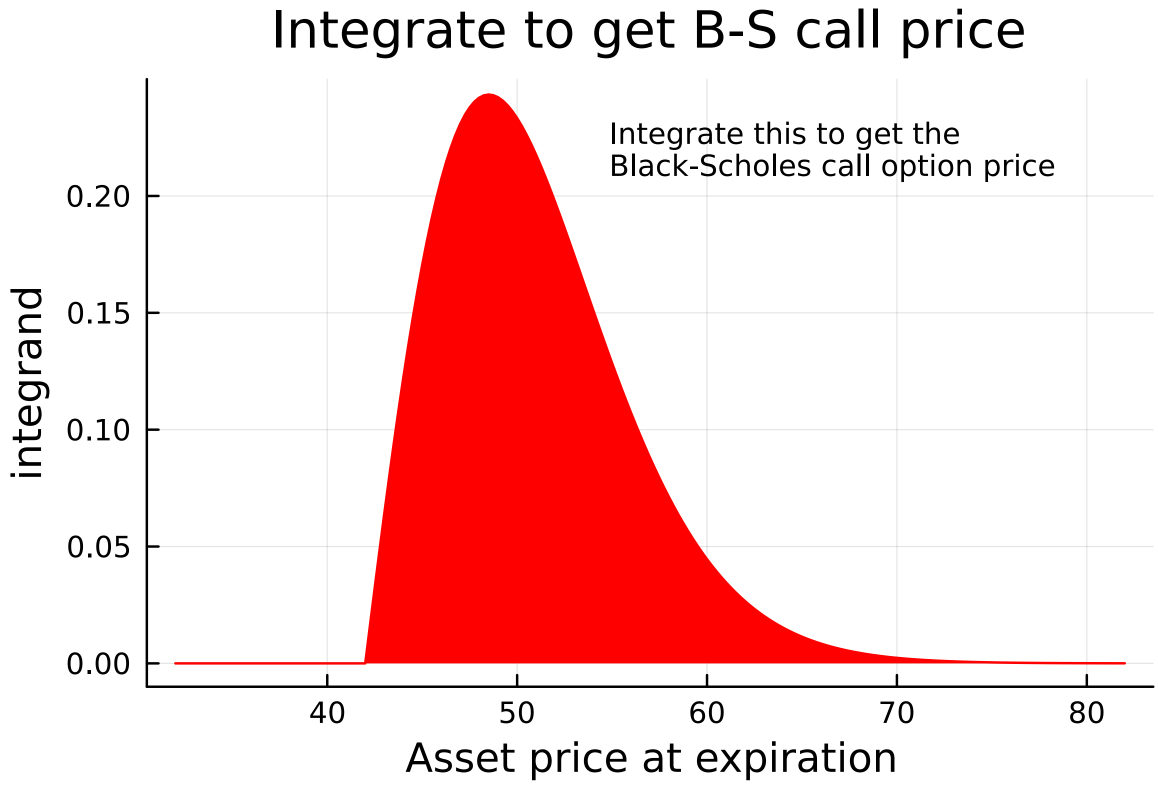
A Remark on the Code

- The `LogNormal` in the `Distributions.jl` package wants the mean and standard deviation of $\ln S_m$ as inputs, but calculates the pdf of S_m.
- The numerical integration is done by the QuadGK.jl package.
- The integration is of the anonymous function `x->BSintegrand(x,S,K,y,m,σ)` over the interval `[K,Inf]`. Here, `x` represents possible values of S_m.

```
"""
BSintegrand(Sₘ,S,K,y,m,σ)

Constructs the integrand for the Black-Scholes call price
"""

function BSintegrand(Sₘ,S,K,y,m,σ)
    μ = log(S) + m*y - m*σ^2/2            #"mean"
    λ = sqrt(m)*σ                         #"std"
    f = pdf(LogNormal(μ,λ),Sₘ)            #log-normal pdf(mean,std)
    z = exp(-y*m)*max(0,Sₘ-K)*f
    return z
end
```

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

```
Sₘ_range = (K-10):0.25:(K+40)      #possible outcomes of underlying price

txt = text("Integrate this to get the\nBlack-Scholes call option price",8,:left)

p1 = plot( Sₘ_range,BSintegrand.(Sₘ_range,S,K,y,m,σ),
           linecolor = :red,
           ylim = (-0.01,0.25),
           fill = (0,:red),
           legend = false,
           title = "Integrate to get B-S call price",
           xlabel = "Asset price at expiration",
           ylabel = "integrand",
           annotation = (55,0.22,txt) )
display(p1)
```



```
C1, = QuadGK.quadgk(x->BSintegrand(x,S,K,y,m,σ),K,Inf)    #numerical integration over (K,Inf)

printblue("Call option price:")
printmat([C1,C],rowNames=["from numerical integration","from BS formula"])
```

```
Call option price:
from numerical integration     2.893
from BS formula                2.893
```

Convergence of BOPM to BS

The next few cells calculate the option price according to binomial model with a CRR calibration where

$u = e^{\sigma\sqrt{h}}, d = e^{-\sigma\sqrt{h}}$ and $p = \frac{e^{yh}-d}{u-d}$.

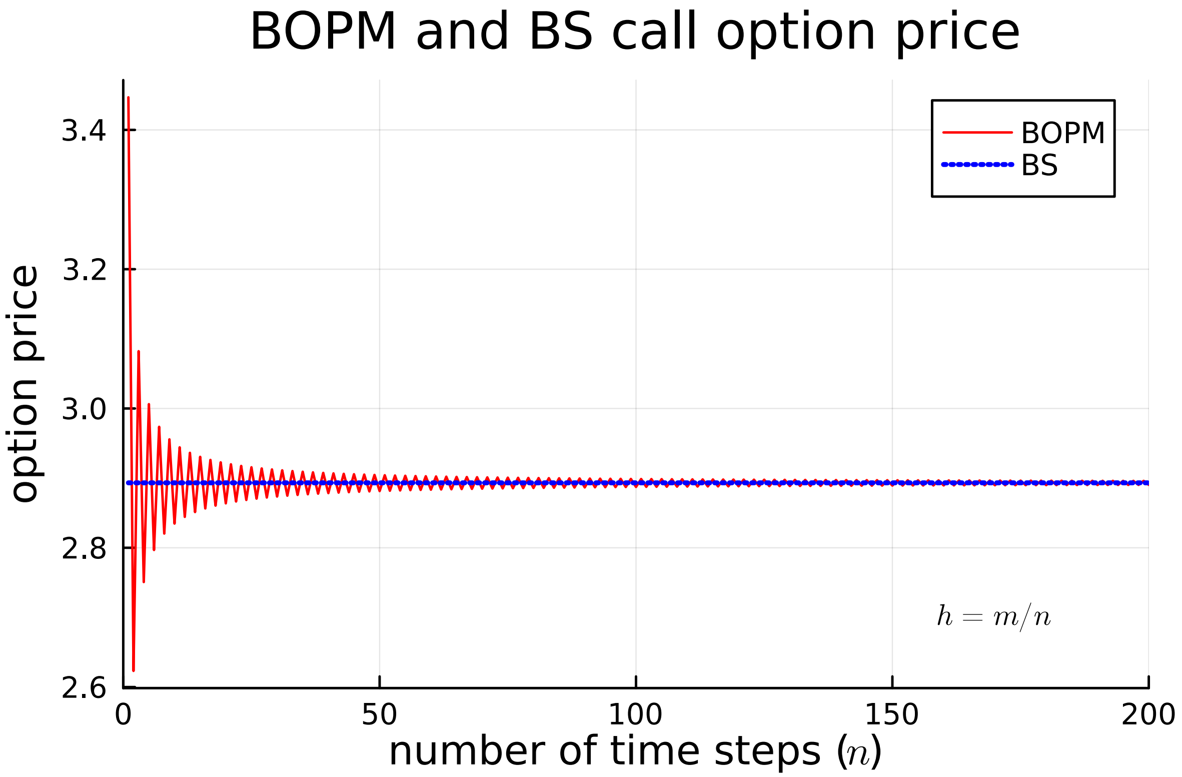
This is done repeatedly, using more and more time steps (n) with $h = m/n$ where m is the fixed time to expiration.

The file included below contains, among other things, the functions `BuildSTree()` and `EuOptionPrice()` from the chapter on the binomial model.

```
using OffsetArrays
include("src/OptionsCalculations.jl");
```

```
#(S,K,m,y,σ) = (42,42,0.5,0.05,0.2)       #these parameters were defined before

nMax = 200

C_bopm = fill(NaN,nMax)
for n in 1:nMax                         #calculate option price nMax times
    #local h, u, d, p, STree, Ce       #local/global is needed in script
    (h,u,d,p) = CRRparams(σ,m,y,n)
    STree     = BuildSTree(S,n,u,d)
    Ce        = EuOptionPrice(STree,K,y,h,p)
    C_bopm[n] = Ce[0][]        #pick out the call price at the starting node
end
```

```
p1 = plot( 1:nMax,[C_bopm fill(C,nMax)],
           linecolor = [:red :blue],
           linestyle = [:solid :dot],
           linewidth = [1 2],
           xlims = [0,nMax],
           label = ["BOPM" "BS"],
           title = "BOPM and BS call option price",
           xlabel = L"number of time steps ($n$)",
           ylabel = "option price",
            annotation = (170,2.7,text(L"h = m/n",8)) )
display(p1)
```



Options 4: Hedging Options

This notebook illustrates how to hedge an option by holding a position in the underlying asset (delta hedging).

Load Packages and Extra Functions

```
using Printf, Distributions

include("src/printmat.jl");
```

```
using Plots, LaTeXStrings
default(size = (480,320),fmt = :png)
```

A First-Order Approximation of the Option Price Change

"Delta hedging" is based on the idea that we can approximate the change in the option price by

$C_{t+h} - C_t \approx \Delta \left(S_{t+h} - S_t\right)$,

where Δ is the derivative of the call option price wrt. the underlying asset price, called the *delta*. (It does *not* indicate a first difference.)

In the Black-Scholes model, the delta of a call option is

$\Delta_c = \frac{\partial C}{\partial S} = e^{-\delta m}\Phi\left(d_1\right)$,

where d_1 is the usual term in Black-Scholes and δ is the continuous dividend rate (possibly 0).

Similarly, the delta of a put option is

$\Delta_p = e^{-\delta m}[\Phi\left(d_1\right) - 1]$.

The file included in the next cell contains the functions `Φ()` and `OptionBlackSPs()` from the chapter on the Black-Scholes model.

The subsequent cell defines a function for the Δ of the Black-Scholes model.

```
include("src/OptionsCalculations.jl");
```

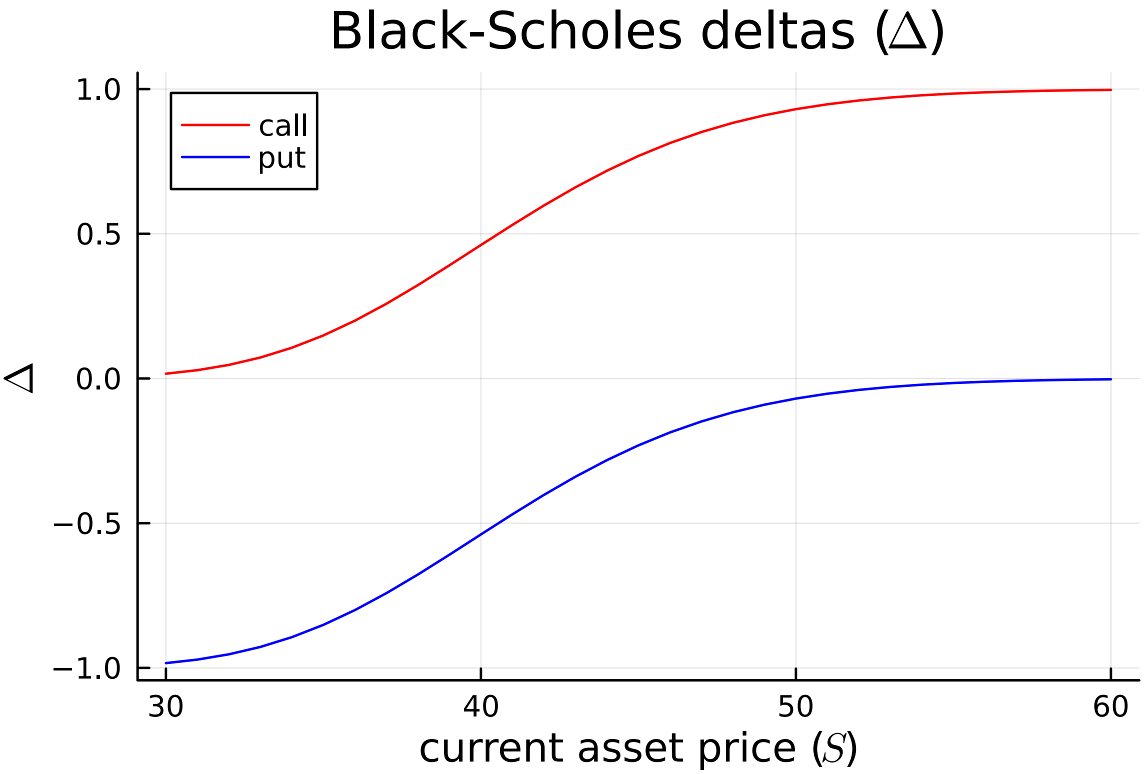
```
"""
Calculate the Black-Scholes delta
"""
function OptionDelta(S,K,m,y,σ,δ=0;isPut=false)
    d1 = ( log(S/K) + (y-δ+0.5*σ^2)*m ) / (σ*sqrt(m))
    d2 = d1 - σ*sqrt(m)
    Δ = isPut ? exp(-δ*m)*(Φ(d1)-1) : exp(-δ*m)*Φ(d1)   #put or call
    return Δ
end
```

```
OptionDelta
```

```
(S,K,m,y,σ) = (42,42,0.5,0.05,0.2)

Δ_c = OptionDelta(S,K,m,y,σ)                #call
Δ_p = OptionDelta(S,K,m,y,σ,0;isPut=true)   #put

printblue("Δ when the underlying asset has no dividends:\n")
printmat([Δ_c Δ_p (Δ_c-Δ_p)];colNames=["call","put","difference"],width=12,colUnderlineQ=true)
```

```
Δ when the underlying asset has no dividends:

        call         put  difference
        ----         ---  ----------

       0.598      -0.402       1.000
```

```
S_range = 30:60          #different spot prices
Δ_c_S   = OptionDelta.(S_range,K,m,y,σ)
Δ_p_S   = OptionDelta.(S_range,K,m,y,σ;isPut=true)

p1 = plot( S_range,[Δ_c_S Δ_p_S],
           linecolor = [:red :blue],
           label = ["call" "put"],
           title = L"Black-Scholes deltas ($\Delta$)",
           xlabel = L"current asset price ($S$)",
           ylabel = L"\Delta" )
display(p1)
```



Hedging an Option

The example below shows how delta hedging works for a European call option when the price of the underlying asset changes (from 42 on day 0 to 43 on day 1). For simplicity, we assume that the Black-Scholes model is a good description of how the option price is set.

```
(S₀,S₁,K,m,y,σ) = (42,43,42,0.5,0.05,0.2)   #prices before, after, parameters

C₀ = OptionBlackSPs(S₀,K,m,y,σ)   #option price at S₀
Δ₀ = OptionDelta(S₀,K,m,y,σ)      #Delta at S₀
M₀ = C₀ - Δ₀*S₀                   #on money market account

C₁ = OptionBlackSPs(S₁,K,m-1/252,y,σ)   #option price at S₁ (it's one day later)
```

```
dC = C₁ - C₀                                #change of option value
dV = Δ₀*(S₁-S₀) - (C₁-C₀)                   #change of hedge portfolio value

xy = [S₀,Δ₀,C₀,M₀,S₁,C₁,dC,dV]
printmat(xy;rowNames=["S₀","Δ₀","C₀","M₀","S₁","C₁","dC","dV"])

printred("\nV changes much less in value than the option, that is, abs(dV) < abs(dC),
so the hedge helps")
```

```
S₀     42.000
Δ₀      0.598
C₀      2.893
M₀    -22.212
S₁     43.000
C₁      3.509
dC      0.616
dV     -0.018


V changes much less in value than the option, that is, abs(dV) < abs(dC),
so the hedge helps
```

Hedging an Option Portfolio

In this case, we have issued *nc* call options and *np* put options with strike *K* and want to know how many units of the underlying asset that we need in order to be hedged. The delta of this portfolio is $nc \cdot \Delta(call) + np \cdot \Delta(put)$.

The example uses `(nc,np) = (3,-2)`. Change to `(1,1.5)` to see what happens. (Maybe the hedge does not work so well in this case…although that is not evaluated here.)

```
(S₀,K,m,y,σ) = (42,42,0.5,0.05,0.2)

(nc,np) = (3,-2)
#(nc,np) = (1,1.5)                              #try this too

Δ_call = OptionDelta.(S₀,K,m,y,σ)               #Delta of call option
Δ_put  = OptionDelta.(S₀,K,m,y,σ,0;isPut=true)  #Delta of put option
Δ      = nc*Δ_call + np*Δ_put

xy = [Δ_call,Δ_put,Δ]
```

```
printmat(xy;rowNames=["Δ of call","Δ of put","Δ of option portfolio"])

printred("We need to buy $(round(Δ,digits=3)) units of the underlying")
```

```
Δ of call                  0.598
Δ of put                  -0.402
Δ of option portfolio      2.598

We need to buy 2.598 units of the underlying
```