FRE6871 R in Finance

Lecture#3, Spring 2025

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

April 7, 2025

NYU | TANDON SCHOOL OF ENGINEERING

Simulating Single-period Defaults

Consider a portfolio of credit assets (bonds or loans) over a single period of time.

At the end of the period, some of the assets default, while the rest don't.

The default probabilities are equal to p_i.

Individual defaults can be simulated by comparing the probabilities p_i with the uniform random numbers u_i.

Default occurs if u_i is less than the default probability p_i:

$$u_i < p_i$$

Simulations in R can be accelerated by pre-computing a vector of random numbers, instead of generatng them one at a time in a loop.

Vectors of random numbers allow using *vectorized* functions, instead of inefficient (slow) `for()` loops.

```
> # Calculate random default probabilities
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> nbonds <- 100
> probv <- runif(nbonds, max=0.2)
> mean(probv)
> # Simulate number of defaults
> unifv <- runif(nbonds)
> sum(unifv < probv)
> # Simulate average number of defaults using for() loop (inefficien
> nsimu <- 1000
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> defaultv <- numeric(nsimu)
> for (i in 1:nsimu) {  # Perform loop
+   unifv <- runif(nbonds)
+   defaultv[i] <- sum(unifv < probv)
+ }  # end for
> # Calculate average number of defaults
> mean(defaultv)
> # Simulate using vectorized functions (efficient way)
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> unifm <- matrix(runif(nsimu*nbonds), ncol=nsimu)
> defaultv <- colSums(unifm < probv)
> mean(defaultv)
> # Plot the distribution of defaults
> x11(width=6, height=5)
> plot(density(defaultv), main="Distribution of Defaults",
+      xlab="number of defaults", ylab="frequency")
> abline(v=mean(defaultv), lwd=3, col="red")
```

Asset Values and Default Thresholds

Defaults can also be simulated using normally distributed variables a_i called *asset values*, instead of the uniformly distributed variables u_i.

The asset values a_i are the *quantiles* corresponding to the uniform variables u_i: $a_i = \Phi^{-1}(u_i)$ (where $\Phi()$ is the cumulative *Standard Normal* distribution).

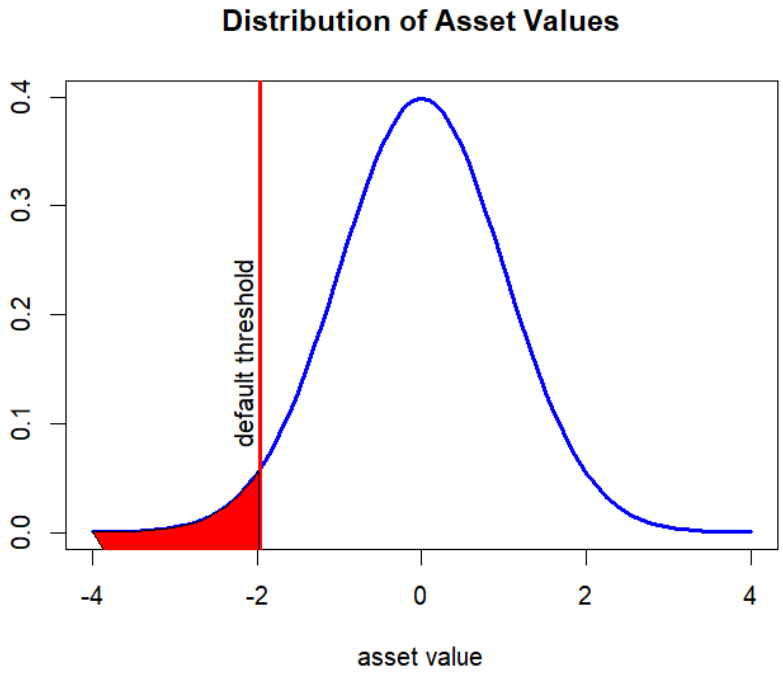
Similarly, the default probabilities p_i are also transformed into *default thresholds* t_i, which are the *quantiles*: $t_i = \Phi^{-1}(p_i)$.

Before, default occurred if u_i was less than the default probability p_i: $u_i < p_i$.

Now, default occurs if the *asset value* a_i is less than the *default threshold* t_i: $a_i < t_i$.

The asset values a_i are mathematical variables which can be negative, so they are not actual company asset values.

```
> # Calculate default thresholds and asset values
> threshv <- qnorm(probv)
> assetm <-qnorm(unifm)
> # Simulate defaults
> defaultv <- colSums(assetm < threshv)
> mean(defaultv)
```




```
> # Plot Standard Normal distribution
> x11(width=6, height=5)
> xlim <- 4; threshv <- qnorm(0.025)
> curve(expr=dnorm(x), type="l", xlim=c(-xlim, xlim),
+ xlab="asset value", ylab="", lwd=3,
+ col="blue", main="Distribution of Asset Values")
> abline(v=threshv, col="red", lwd=3)
> text(x=threshv-0.1, y=0.15, labels="default threshold",
+  lwd=2, srt=90, pos=3)
> # Plot polygon area
> xvar <- seq(-xlim, xlim, length=100)
> yvar <- dnorm(xvar)
> intail <- ((xvar >= (-xlim)) & (xvar <= threshv))
> polygon(c(xlim, xvar[intail], threshv),
+   c(-1, yvar[intail], -1), col="red")
```

Vasicek Model of Correlated Asset Values

So far, the asset values are independent from each other, but in reality default events are correlated.

The *Vasicek* model introduces correlation between the asset values a_i.

Under the *Vasicek* single factor model, the asset value a_i is equal to the sum of a *systematic* factor s, plus an *idiosyncratic* factor z_i:

$$a_i = \sqrt{\rho}\, s + \sqrt{1-\rho}\, z_i$$

Where ρ is the correlation between asset values.

The variables s, z_i, and a_i all follow the *Standard Normal* distribution $\phi(0,1)$.

The matrix of asset values is arranged with columns corresponding to simulations and rows corresponding to bonds or loans.

The *Vasicek* model resembles the *CAPM* model, with the asset value equal to the sum of a *systematic* factor plus an *idiosyncratic* factor.

The Bank for International Settlements (BIS) uses the *Vasicek* model as part of its regulatory capital requirements for bank credit risk:
http://bis2information.org/content/Vasicek_model
https://www.bis.org/bcbs/basel3.htm
https://www.bis.org/bcbs/irbriskweight.pdf

```
> # Define correlation parameters
> rho <- 0.2
> rhos <- sqrt(rho) ; rhosm <- sqrt(1-rho)
> nbonds <- 5 ; nsimu <- 10000
> # Calculate vector of systematic and idiosyncratic factors
> sysv <- rnorm(nsimu)
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> isync <- rnorm(nsimu*nbonds)
> dim(isync) <- c(nbonds, nsimu)
> # Simulate asset values using vectorized functions (efficient way)
> assetm <- t(rhos*sysv + t(rhosm*isync))
> # Asset values are standard normally distributed
> apply(assetm, MARGIN=1, function(x) c(mean=mean(x), sd=sd(x)))
> # Calculate correlations between asset values
> cor(t(assetm))
> # Simulate asset values using for() loop (inefficient way)
> # Allocate matrix of assets
> assetn <- matrix(nrow=nbonds, ncol=nsimu)
> # Simulate asset values using for() loop
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> for (i in 1:nsimu) {  # Perform loop
+   assetn[, i] <- rhos*sysv[i] + rhosm*rnorm(nbonds)
+ }  # end for
> all.equal(assetn, assetm)
> # benchmark the speed of the two methods
> library(microbenchmark)
> summary(microbenchmark(
+   forloop={for (i in 1:nsimu) {
+     rhos*sysv[i] + rhosm*rnorm(nbonds)}},
+   vectorized={t(rhos*sysv + t(rhosm*isync))},
+   times=10))[, c(1, 4, 5)]
```

Vasicek Model of Correlated Defaults

Under the *Vasicek* model, default occurs if the *asset value* a_i is less than the *default threshold* t_i:

$$a_i = \sqrt{\rho} s + \sqrt{1-\rho} z_i$$

$$a_i < t_i$$

The *systematic* factor s may be considered to represent the state of the macro economy, with positive values representing an economic expansion, and negative values representing an economic recession.

When the value of the *systematic* factor s is positive, then the asset values will all tend to be bigger as well, which will produce fewer defaults.

But when the *systematic* factor is negative, then the asset values will tend to be smaller, which will produce more defaults.

This way the *Vasicek* model introduces a correlation among defaults.

```
> # Calculate random default probabilities
> nbonds <- 5
> probv <- runif(nbonds, max=0.2)
> mean(probv)
> # Calculate default thresholds
> threshv <- qnorm(probv)
> # Calculate number of defaults using vectorized functions (efficie
> # Calculate vector of number of defaults
> rowMeans(assetm < threshv)
> probv
> # Calculate number of defaults using for() loop (inefficient way)
> # Allocate matrix of defaultm
> defaultm <- matrix(nrow=nbonds, ncol=nsimu)
> # Simulate asset values using for() loop
> for (i in 1:nsimu) {  # Perform loop
+   defaultm[, i] <- (assetm[, i] < threshv)
+ }  # end for
> rowMeans(defaultm)
> rowMeans(assetm < threshv)
> # Calculate correlations between defaults
> cor(t(defaultm))
```

Asset Correlation and Default Correlation

Default correlation is defined as the correlation between the `Boolean` vectors of default events.

The *Vasicek* model introduces correlation among default events, through the correlation of *asset values*.

If *asset values* have a positive correlation, then the defaults among credits are clustered together, and if one credit defaults then the other credits are more likely to default as well.

Empirical studies have found that the asset correlation ρ can vary between 5% to 20%, depending on the default risk.

Credits with higher default risk tend to also have higher asset correlation, since they are more sensitive to the economic conditions.

Default correlations are usually much lower than the corresponding asset correlations.

```
> # Define default probabilities
> nbonds <- 2
> defprob <- 0.2
> threshv <- qnorm(defprob)
> # Define correlation parameters
> rho <- 0.2
> rhos <- sqrt(rho) ; rhosm <- sqrt(1-rho)
> # Calculate vector of systematic factors
> nsimu <- 1000
> sysv <- rnorm(nsimu)
> isync <- rnorm(nsimu*nbonds)
> dim(isync) <- c(nbonds, nsimu)
> # Simulate asset values using vectorized functions
> assetm <- t(rhos*sysv + t(rhosm*isync))
> # Calculate number of defaults using vectorized functions
> defaultm <- (assetm < threshv)
> # Calculate average number of defaults and compare to defprob
> rowMeans(defaultm)
> defprob
> # Calculate correlations between assets
> cor(t(assetm))
> # Calculate correlations between defaults
> cor(t(defaultm))
```

Cumulative Defaults Under the Vasicek Model

A formula for the default distribution under the Vasicek Model can be derived under the assumption that the number of assets is very large and that they all have the same default probabilities $p_i = p$.

In that case the single default threshold is equal to $t = \Phi^{-1}(p)$.

If the systematic factor s is fixed, then the *asset value* a_i follows the *Normal* distribution with mean equal to $\sqrt{\rho}s$ and standard deviation equal to $\sqrt{1-\rho}$:

$$a_i = \sqrt{\rho}s + \sqrt{1-\rho}z_i$$

The conditional default probability $p(s)$, given the systematic factor s, is equal to:

$$p(s) = \Phi\left(\frac{t - \sqrt{\rho}s}{\sqrt{1-\rho}}\right)$$

Since the systematic factor s is fixed, then the defaults are all independent with the same default probability $p(s)$.

Because the number of assets is very large, the percentage x of the portfolio that defaults, is equal to the conditional default probability $x = p(s)$.

We can invert the formula $x = \Phi\left(\frac{t-\sqrt{\rho}s}{\sqrt{1-\rho}}\right)$ to obtain the systematic factor s:

$$s = \frac{\sqrt{1-\rho}\,\Phi^{-1}(x) - t}{\sqrt{\rho}}$$

Since the systematic factor s follows the *Standard Normal* distribution, then the portfolio cumulative default probability $P(x)$ is equal to:

$$P(x) = \Phi(s) = \Phi\left(\frac{\sqrt{1-\rho}\,\Phi^{-1}(x) - t}{\sqrt{\rho}}\right)$$

Cumulative Default Distribution And Correlation

The cumulative portfolio default probability $P(x)$:

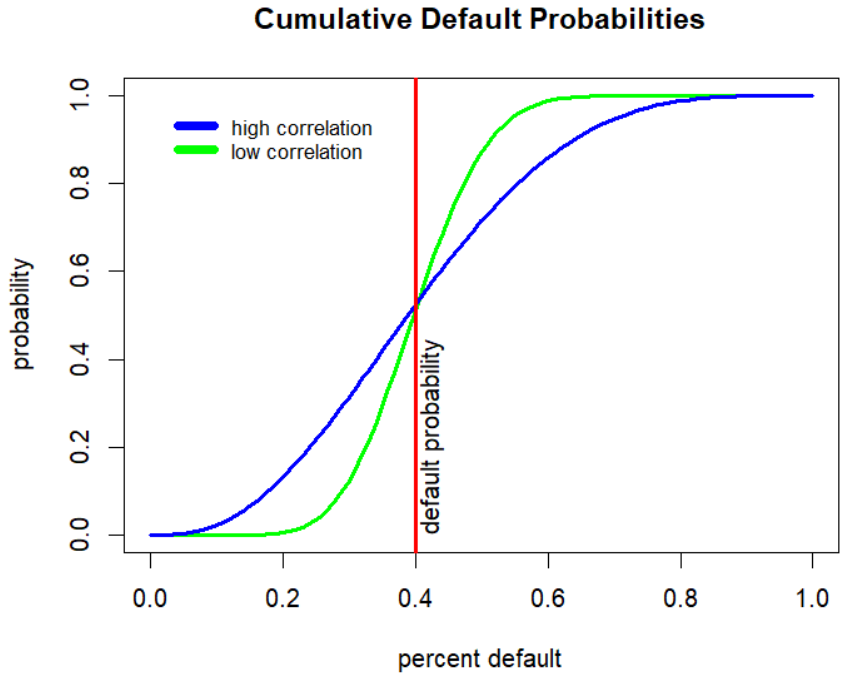
$$P(x) = \Phi\left(\frac{\sqrt{1-\rho}\,\Phi^{-1}(x) - t}{\sqrt{\rho}}\right)$$

Depends on the correlation parameter ρ.

If the correlation ρ is very low (close to 0) then the percentage x of the portfolio defaults is always very close to the default probability p, and the cumulative default probability curve is steep close to the expected value of p.

If the correlation ρ is very high (close to 1) then the percentage x of the portfolio defaults has a very wide dispersion around the default probability p, and the cumulative default probability curve is flat close to the expected value of p.

This is because with high correlation, the assets will tend to all default together or not default.




```
> # Define cumulative default distribution function
> cumdefdistr <- function(x, threshv=(-2), rho=0.2)
+   pnorm((sqrt(1-rho)*qnorm(x) - threshv)/sqrt(rho))
> defprob <- 0.4; threshv <- qnorm(defprob)
> cumdefdistr(x=0.2, threshv=qnorm(defprob), rho=rho)
> # Plot cumulative default distribution function
> curve(expr=cumdefdistr(x, threshv=threshv, rho=0.05),
+ xlim=c(0, 0.999), lwd=3, xlab="percent default", ylab="probabili
+ col="green", main="Cumulative Default Probabilities")
```

```
> # Plot default distribution with higher correlation
> curve(expr=cumdefdistr(x, threshv=threshv, rho=0.2),
+     xlim=c(0, 0.999), add=TRUE, lwd=3, col="blue", main="")
> # Add legend
> legend(x="topleft",
+    legend=c("high correlation", "low correlation"),
+    title=NULL, inset=0.05, cex=1.0, bg="white",
+    bty="n", lwd=6, lty=1, col=c("blue", "green"))
> # Add unconditional default probability
> abline(v=defprob, col="red", lwd=3)
> text(x=defprob, y=0.0, labels="default probability",
+  lwd=2, srt=90, pos=4)
```

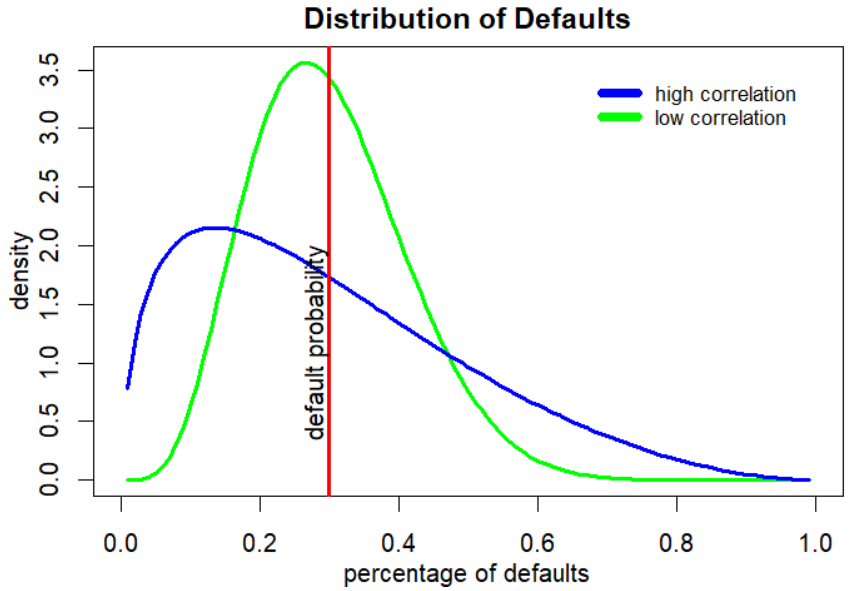
Distribution of Defaults Under the Vasicek Model

The probability density $f(x)$ of portfolio defaults is equal to the derivative of the cumulative default distribution $P(x)$:

$$f(x) = \frac{\sqrt{1-\rho}}{\sqrt{\rho}} \exp\left(-\frac{1}{2\rho}\left(\sqrt{1-\rho}\,\Phi^{-1}(x) - t\right)^2 + \frac{1}{2}\Phi^{-1}(x)^2\right)$$

If the correlation ρ is very low (close to 0) then the probability density $f(x)$ is centered around the default probability p.

If the correlation ρ is very high (close to 1) then the probability density $f(x)$ is wide, with significant probability of large portfolio defaults and also small portfolio defaults.




```
> # Define default probability density function
> defdistr <- function(x, threshv=(-2), rho=0.2)
+   sqrt((1-rho)/rho)*exp(-(sqrt(1-rho)*qnorm(x) -
+   threshv)^2/(2*rho) + qnorm(x)^2/2)
> # Define parameters
> rho <- 0.2 ; rhos <- sqrt(rho) ; rhosm <- sqrt(1-rho)
> defprob <- 0.3; threshv <- qnorm(defprob)
> defdistr(0.03, threshv=threshv, rho=rho)
> # Plot probability distribution of defaults
> curve(expr=defdistr(x, threshv=threshv, rho=0.1),
+ xlim=c(0, 1.0), lwd=3,
+ xlab="Default percentage", ylab="Density",
+ col="green", main="Distribution of Defaults")
```

```
> # Plot default distribution with higher correlation
> curve(expr=defdistr(x, threshv=threshv, rho=0.3),
+ xlab="default percentage", ylab="",
+ add=TRUE, lwd=3, col="blue", main="")
> # Add legend
> legend(x="topright",
+    legend=c("high correlation", "low correlation"),
+    title=NULL, inset=0.05, cex=1.0, bg="white",
+    bty="n", lwd=6, lty=1, col=c("blue", "green"))
> # Add unconditional default probability
> abline(v=defprob, col="red", lwd=3)
> text(x=defprob, y=2, labels="default probability",
+  lwd=2, srt=90, pos=2)
```

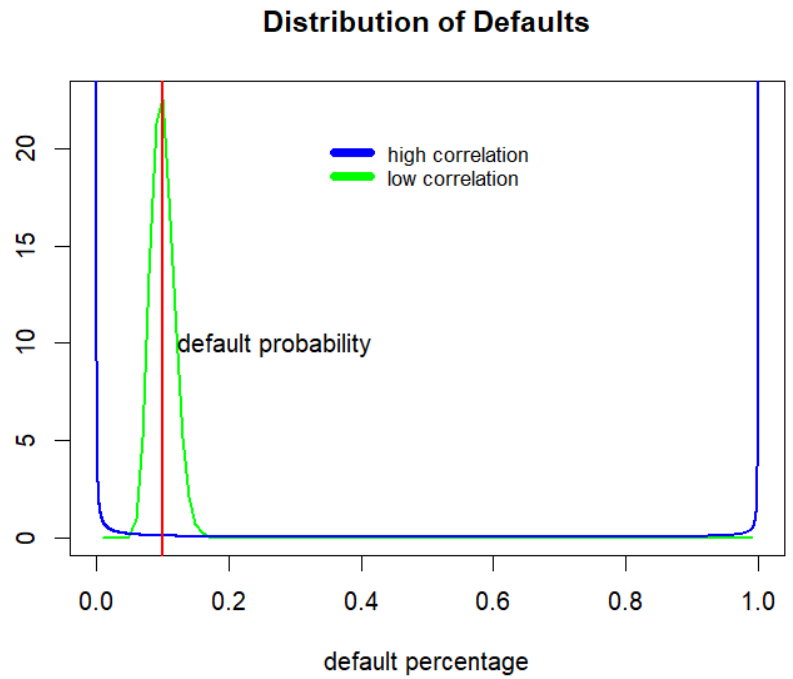
Distribution of Defaults Under Extreme Correlations

If the correlation ρ is close to 0, then the asset values a_i are independent from each other, and defaults are also independent, so that the percentage of portfolio defaults is very close to the default probability p.

In that case, the probability density of portfolio defaults is very narrow and is centered on the default probability p.

If the correlation ρ is close to 1, then the asset values a_i are almost the same, and defaults occur at the same time, so that the percentage of portfolio defaults is either 0 or 1.

In that case, the probability density of portfolio defaults becomes *bimodal*, with two peaks around *zero* and 1.




```
> # Plot default distribution with low correlation
> curve(expr=defdistr(x, threshv=threshv, rho=0.01),
+   xlab="default percentage", ylab="", lwd=2,
+   col="green", main="Distribution of Defaults")
> # Plot default distribution with high correlation
> curve(expr=defdistr(x, threshv=threshv, rho=0.99),
+   xlab="percentage of defaults", ylab="density",
+   add=TRUE, lwd=2, n=10001, col="blue", main="")
```

```
> # Add legend
> legend(x="top", legend=c("high correlation", "low correlation"),
+    title=NULL, inset=0.1, cex=1.0, bg="white",
+    bty="n", lwd=6, lty=1, col=c("blue", "green"))
> # Add unconditional default probability
> abline(v=0.1, col="red", lwd=2)
> text(x=0.1, y=10, lwd=2, pos=4, labels="default probability")
```

Numerical Integration of Functions

The function `integrate()` performs numerical integration of a function of a single variable, i.e. it calculates a definite integral over an integration interval.

Additional parameters can be passed to the integrated function through the dots `"..."` argument of the function `integrate()`.

The function `integrate()` accepts the integration limits `-Inf` and `Inf` equal to minus and plus infinity.

```
> # Get help for integrate()
> ?integrate
> # Calculate slowly converging integral
> func <- function(x) {1/((x+1)*sqrt(x))}
> integrate(func, lower=0, upper=10)
> integrate(func, lower=0, upper=Inf)
> # Integrate function with parameter lambdaf
> func <- function(x, lambdaf=1) {
+   exp(-x*lambdaf)
+ }  # end func
> integrate(func, lower=0, upper=Inf)
> integrate(func, lower=0, upper=Inf, lambdaf=2)
> # Cumulative probability over normal distribution
> pnorm(-2)
> integrate(dnorm, low=2, up=Inf)
> str(dnorm)
> pnorm(-1)
> integrate(dnorm, low=2, up=Inf, mean=1)
> # Expected value over normal distribution
> integrate(function(x) x*dnorm(x), low=2, up=Inf)
```

Portfolio Loss Distribution

The expected loss (EL) of a credit portfolio is equal to the sum of the default probabilities p_i multiplied by the loss given default LGD (aka the *loss severity* - equal to 1 minus the *recovery rate*):

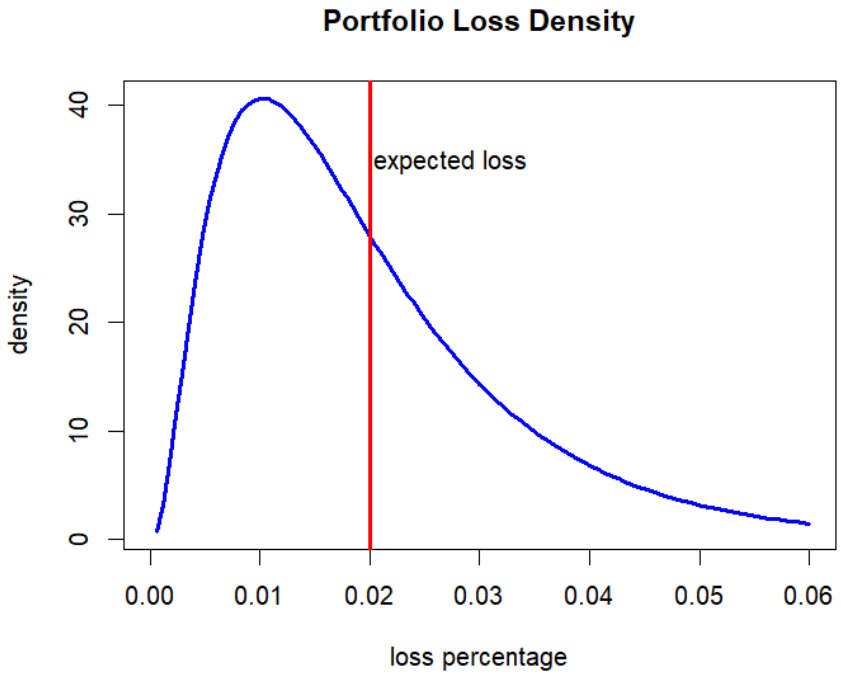
$$EL = \sum_{i=1}^{n} p_i LGD_i$$

Then the *cumulative loss distribution* is equal to:

$$P(x) = \Phi\left(\frac{\sqrt{1-\rho}\,\Phi^{-1}\left(\frac{x}{LGD}\right) - t}{\sqrt{\rho}}\right)$$

And the *default distribution* is the derivative, and is equal to:

$$f(x) = \frac{\sqrt{1-\rho}}{LGD\sqrt{\rho}} \exp\left(-\frac{1}{2\rho}\left(\sqrt{1-\rho}\Phi^{-1}\left(\frac{x}{LGD}\right) - t\right)^2 + \frac{1}{2}\Phi^{-1}\left(\frac{x}{LGD}\right)\right)^2$$




```
> # Vasicek model parameters
> rho <- 0.1; lgd <- 0.4
> defprob <- 0.05; threshv <- qnorm(defprob)
> # Define Vasicek cumulative loss distribution
> cumlossdistr <- function(x, threshv=(-2), rho=0.2, lgd=0.4)
+   pnorm((sqrt(1-rho)*qnorm(x/lgd) - threshv)/sqrt(rho))
> # Define Vasicek loss distribution function
> lossdistr <- function(x, threshv=(-2), rho=0.2, lgd=0.4)
+   sqrt((1-rho)/rho)*exp(-(sqrt(1-rho)*qnorm(x/lgd) - threshv)^2/(2*rho) + qnorm(x/lgd)^2/2)/lgd
```

```
> # Plot probability distribution of losses
> x11(width=6, height=5)
> curve(expr=lossdistr(x, threshv=threshv, rho=rho),
+ cex.main=1.8, cex.lab=1.8, cex.axis=1.5,
+ type="l", xlim=c(0, 0.06),
+ xlab="loss percentage", ylab="density", lwd=3,
+ col="blue", main="Portfolio Loss Density")
> # Add line for expected loss
> abline(v=lgd*defprob, col="red", lwd=3)
> text(x=lgd*defprob-0.001, y=35, labels="expected loss", lwd=3, pos
```

Collateralized Debt Obligations (*CDOs*)

Collateralized Debt Obligations (cash *CDOs*) are securities (bonds) collateralized by other debt assets.

The *CDO* assets can be debt instruments like bonds, loans, and mortgages.

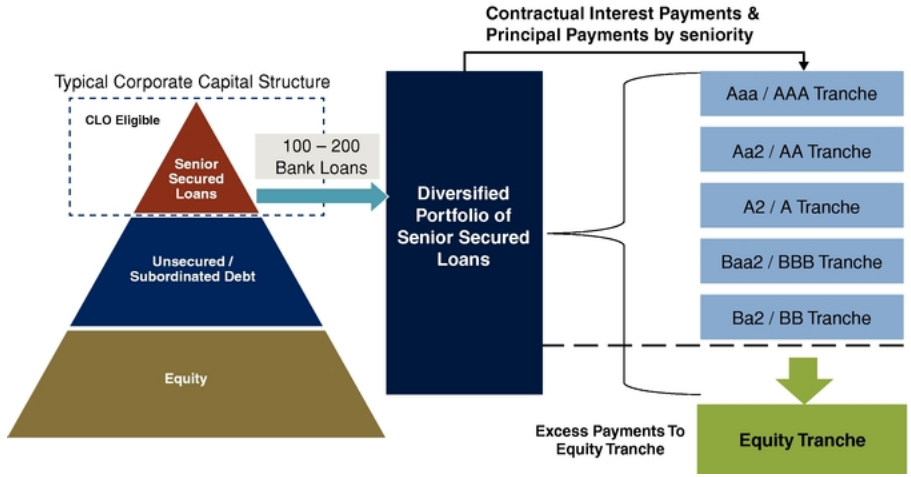
The *CDO* liabilities are *CDO* tranches, which receive cashflows from the *CDO* assets, and are exposed to their defaults.

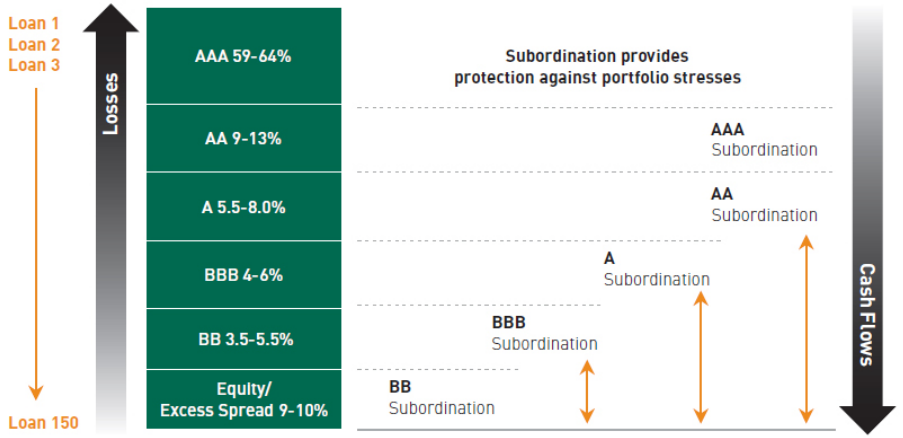
CDO tranches have an attachment point (subordination, i.e. the percentage of asset default losses at which the tranche starts absorbing those losses), and a detachment point when the tranche is wiped out (suffers 100% losses).

The *equity tranche* is the most junior tranche, and is the first to absorb default losses.

The *mezzanine tranches* are senior to the *equity tranche* and absorb losses ony after the *equity tranche* is wiped out.

The *senior tranche* is the most senior tranche, and is the last to absorb losses.





CDO Tranche Losses

Single-tranche (synthetic) *CDOs* are credit default swaps which reference credit portfolios.

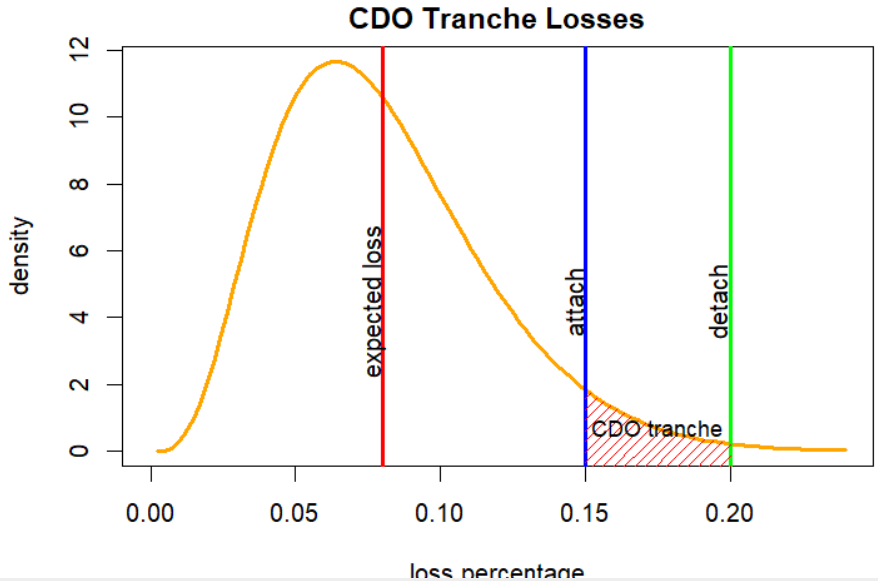
The expected loss *EL* on a *CDO* tranche is:

$$EL = \int_a^d \frac{x-a}{d-a} f(x)\,\mathrm{d}x + \int_d^{LGD} f(x)\,\mathrm{d}x$$

Where $f(x)$ is the density of portfolio losses, and a and d are the tranche attachment (subordination) and detachment points.

The difference $(d-a)$ is the tranche *thickness*, so that *EL* is the expected loss as a percentage of the tranche notional.

A single-tranche *CDO* can be thought of as a short option spread on the asset defaults, struck at the attachment and detachment points.




```
> # Define Vasicek cumulative loss distribution
> # (with error handling for x)
> cumlossdistr <- function(x, threshv=(-2), rho=0.2, lgd=0.4) {
+   qnormv <- ifelse(x/lgd < 0.999, qnorm(x/lgd), 3.1)
+   pnorm((sqrt(1-rho)*qnormv - threshv)/sqrt(rho))
+ }  # end cumlossdistr
> # Define Vasicek loss distribution function
> # (vectorized version with error handling for x)
> lossdistr <- function(x, threshv=(-2), rho=0.1, lgd=0.4) {
+   qnormv <- ifelse(x/lgd < 0.999, qnorm(x/lgd), 3.1)
+   sqrt((1-rho)/rho)*exp(-(sqrt(1-rho)*qnormv - threshv)^2/(2*rho)
+ }  # end lossdistr
```

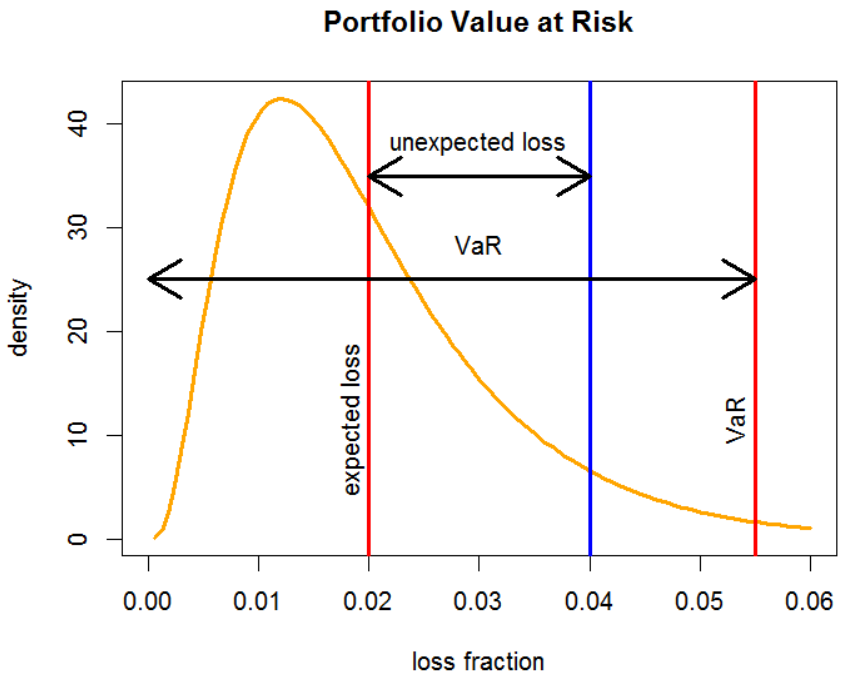
```
> defprob <- 0.2; threshv <- qnorm(defprob)
> rho <- 0.1; lgd <- 0.4
> attachp <- 0.15; detachp <- 0.2
> # Expected tranche loss is sum of two terms
> tranchel <-
+   # Loss between attachp and detachp
+   integrate(function(x, attachp) (x-attachp)*lossdistr(x,
+ threshv=threshv, rho=rho, lgd=lgd),
+ low=attachp, up=detachp, attachp=attachp)$value/(detachp-attachp)
+   # Loss in excess of detachp
+   (1-cumlossdistr(x=detachp, threshv=threshv, rho=rho, lgd=lgd))
> # Plot probability distribution of losses
> curve(expr=lossdistr(x, threshv=threshv, rho=rho),
+ cex.main=1.8, cex.lab=1.8, cex.axis=1.5,
+ type="l", xlim=c(0, 3*lgd*defprob),
+ xlab="loss percentage", ylab="density", lwd=3,
+ col="orange", main="CDO Tranche Losses")
> # Add line for expected loss
> abline(v=lgd*defprob, col="red", lwd=3)
> text(x=lgd*defprob-0.001, y=4, labels="expected loss",
+   lwd=2, srt=90, pos=3, cex=1.8)
```

Portfolio Value at Risk

Value at Risk (*VaR*) measures extreme portfolio loss (but not the worst possible loss), defined as the *quantile* of the loss distribution, corresponding to a given confidence level α.

A loss exceeding the *EL* is called the Unexpected Loss (*UL*), and can be calculated from the *portfolio loss distribution*.

```
> # Add lines for unexpected loss
> abline(v=0.04, col="blue", lwd=3)
> arrows(x0=0.02, y0=35, x1=0.04, y1=35, code=3, lwd=3, cex=0.5)
> text(x=0.03, y=36, labels="unexpected loss", lwd=2, pos=3)
> # Add lines for VaR
> abline(v=0.055, col="red", lwd=3)
> arrows(x0=0.0, y0=25, x1=0.055, y1=25, code=3, lwd=3, cex=0.5)
> text(x=0.03, y=26, labels="VaR", lwd=2, pos=3)
> text(x=0.055-0.001, y=10, labels="VaR", lwd=2, srt=90, pos=3)
```



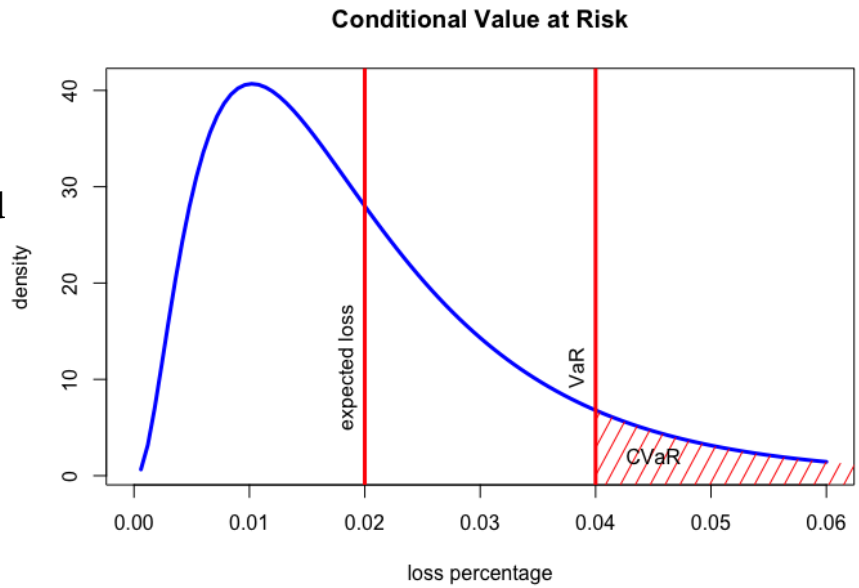
Portfolio Conditional Value at Risk

The *Conditional Value at Risk* (*CVaR*) is equal to the average of the *VaR* values for all the confidence levels greater than the given confidence level α:

$$\mathrm{CVaR} = \frac{1}{1-\alpha}\int_{\alpha}^{1}\mathrm{VaR}(p)\,\mathrm{d}p = \frac{1}{1-\alpha}\int_{\mathrm{VaR}}^{LGD} x\,f(x)\,\mathrm{d}$$

The *Conditional Value at Risk* is also called the Expected Shortfall (*ES*), or Expected Tail Loss (*ETL*).

```
> varisk <- 0.04; varmax <- 4*lgd*defprob
> # Calculate CVaR
> cvar <- integrate(function(x) x*lossdistr(x, threshv=threshv,
+   rho=rho, lgd=lgd), low=varisk, up=lgd)$value
> cvar <- cvar/integrate(lossdistr, low=varisk, up=lgd,
+    threshv=threshv, rho=rho, lgd=lgd)$value
> # Plot probability distribution of losses
> curve(expr=lossdistr(x, threshv=threshv, rho=rho),
+ type="l", xlim=c(0, 0.06),
+ xlab="loss percentage", ylab="density", lwd=3,
+ col="blue", main="Conditional Value at Risk")
> # Add line for expected loss
> abline(v=lgd*defprob, col="red", lwd=3)
> text(x=lgd*defprob-0.001, y=10, labels="expected loss", lwd=2, s
```




```
> # Add lines for VaR
> abline(v=varisk, col="red", lwd=3)
> text(x=varisk-0.001, y=10, labels="VaR",
+  lwd=2, srt=90, pos=3)
> # Add shading for CVaR
> vars <- seq(varisk, varmax, length=100)
> densv <- sapply(vars, lossdistr,
+   threshv=threshv, rho=rho)
> # Draw shaded polygon
> polygon(c(varisk, vars, varmax), density=20,
+   c(-1, densv, -1), col="red", border=NA)
> text(x=varisk+0.005, y=0, labels="CVaR", lwd=2, pos=3)
```

Value at Risk Under the Vasicek Model

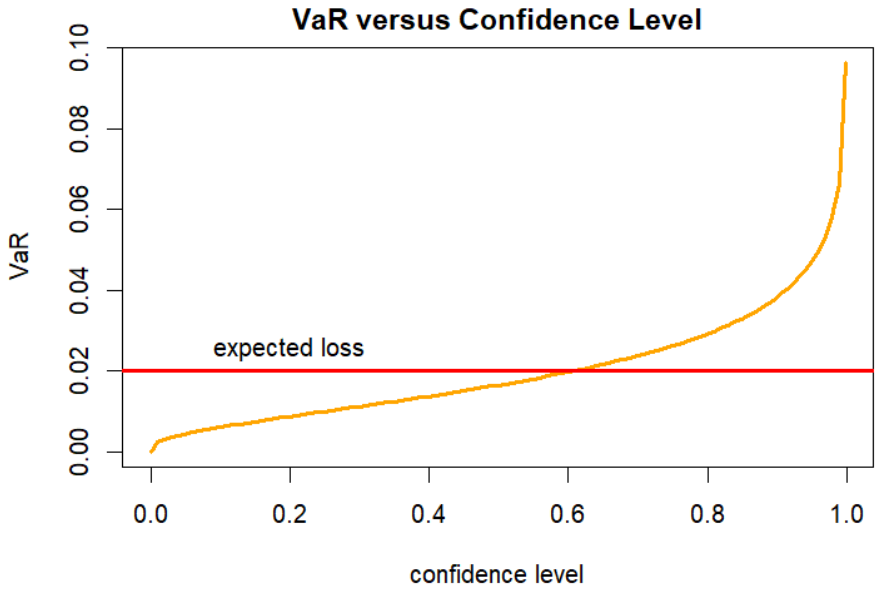
Value at Risk (*VaR*) measures extreme portfolio loss (but not the worst possible loss), defined as the *quantile* of the loss distribution, corresponding to a given confidence level α.

The *cumulative loss distribution* is equal to:

$$P(x) = \Phi\left(\frac{\sqrt{1-\rho}\,\Phi^{-1}\left(\frac{x}{LGD}\right) - t}{\sqrt{\rho}}\right)$$

Then the *quantile* of the loss distribution (x = *VaR*), for a given a confidence level $\alpha = P(x)$, is given by the inverse of the *cumulative loss distribution*:

$$VaR(\alpha) = LGD \cdot \Phi\left(\frac{\sqrt{\rho}\Phi^{-1}(\alpha) + t}{\sqrt{1-\rho}}\right)$$

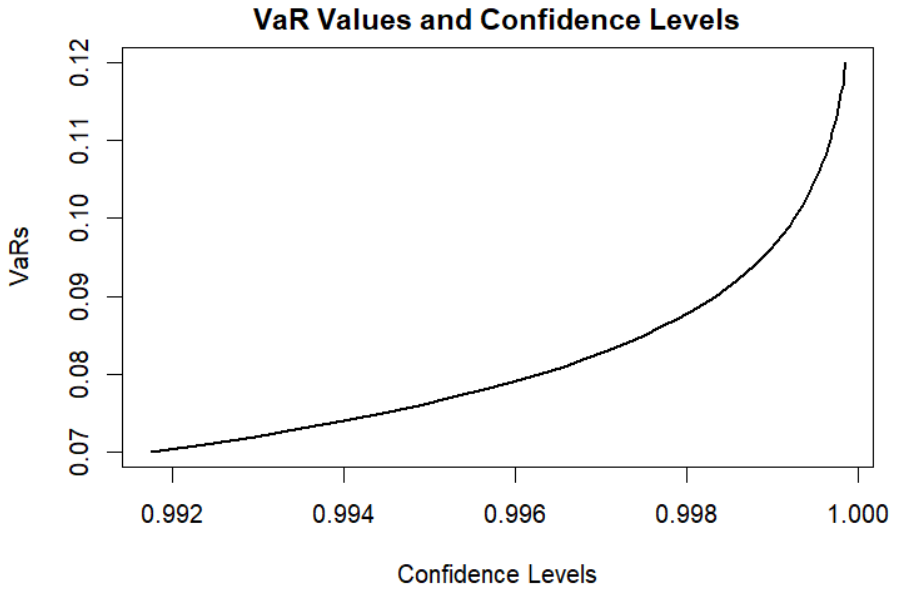



```
> # VaR (quantile of the loss distribution)
> varfun <- function(x, threshv=qnorm(0.1), rho=0.1, lgd=0.4)
+   lgd*pnorm((sqrt(rho)*qnorm(x) + threshv)/sqrt(1-rho))
> varfun(x=0.99, threshv=threshv, rho=rho, lgd=lgd)
> # Plot VaR
> curve(expr=varfun(x, threshv=threshv, rho=rho, lgd=lgd),
+ type="l", xlim=c(0, 0.999), xlab="confidence level", ylab="VaR", lwd=3,
+ col="orange", main="VaR versus Confidence Level")
> # Add line for expected loss
> abline(h=lgd*defprob, col="red", lwd=3)
> text(x=0.2, y=lgd*defprob, labels="expected loss", lwd=2, pos=3)
```

Value at Risk and Confidence Levels

The confidence levels of *VaR* values can also be calculated by integrating over the tail of the loss density function.

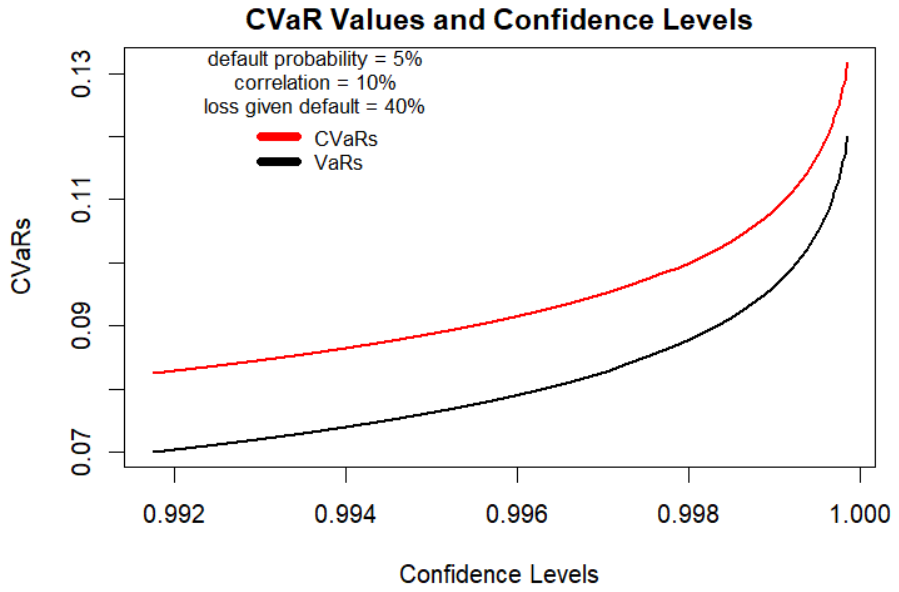
```
> # Integrate lossdistr() over full range
> integrate(lossdistr, low=0.0, up=lgd,
+     threshv=threshv, rho=rho, lgd=lgd)
> # Calculate expected losses using lossdistr()
> integrate(function(x) x*lossdistr(x, threshv=threshv,
+   rho=rho, lgd=lgd), low=0.0, up=lgd)
> # Calculate confidence levels corresponding to VaR values
> vars <- seq(0.07, 0.12, 0.001)
> conv <- sapply(vars, function(varisk) {
+   integrate(lossdistr, low=varisk, up=lgd,
+       threshv=threshv, rho=rho, lgd=lgd)
+ })  # end sapply
> conv <- cbind(as.numeric(t(conv)[, 1]), vars)
> colnames(conv) <- c("levels", "VaRs")
> # Calculate 95% confidence level VaR value
> conv[match(TRUE, conv[, "levels"] < 0.05), "VaRs"]
> plot(x=1-conv[, "levels"],
+      y=conv[, "VaRs"], lwd=2,
+      xlab="confidence level", ylab="VaRs",
+      t="l", main="VaR Values and Confidence Levels")
```



Conditional Value at Risk Under the Vasicek Model

The *CVaR* values can be calculated by integrating over the tail of the loss density function.

```
> # Calculate CVaR values
> cvars <- sapply(vars, function(varisk) {
+   integrate(function(x) x*lossdistr(x, threshv=threshv,
+ rho=rho, lgd=lgd), low=varisk, up=lgd)})  # end sapply
> conv <- cbind(conv, as.numeric(t(cvars)[, 1]))
> colnames(conv)[3] <- "CVaRs"
> # Divide CVaR by confidence level
> conv[, "CVaRs"] <- conv[, "CVaRs"]/conv[, "levels"]
> # Calculate 95% confidence level CVaR value
> conv[match(TRUE, conv[, "levels"] < 0.05), "CVaRs"]
> # Plot CVaRs
> plot(x=1-conv[, "levels"], y=conv[, "CVaRs"],
+      t="l", col="red", lwd=2,
+      ylim=range(conv[, c("VaRs", "CVaRs")]),
+      xlab="confidence level", ylab="CVaRs",
+      main="CVaR Values and Confidence Levels")
```




```
> # Add VaRs
> lines(x=1-conv[, "levels"], y=conv[, "VaRs"], lwd=2)
> # Add legend
> legend(x="topleft", legend=c("CVaRs", "VaRs"),
+    title="default probability = 5%
+ correlation = 10%
+ loss given default = 40%",
+    inset=0.1, cex=1.0, bg="white", bty="n",
+    lwd=6, lty=1, col=c("red", "black"))
```

Simulating Portfolio Losses Under the Vasicek Model

If the default probabilities p_i are not all the same, then there's no simple formula for the *portfolio loss distribution* under the Vasicek Model.

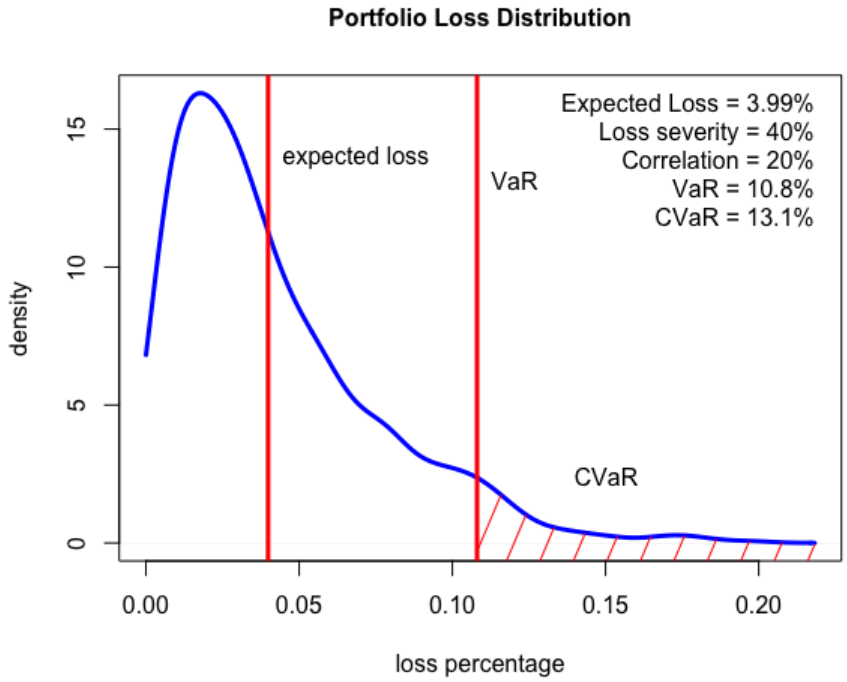
In that case the portfolio losses and *VaR* must be simulated.

```
> # Define model parameters
> nbonds <- 300; nsimu <- 1000; lgd <- 0.4
> # Define correlation parameters
> rho <- 0.2; rhos <- sqrt(rho); rhosm <- sqrt(1-rho)
> # Calculate default probabilities and thresholds
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> probv <- runif(nbonds, max=0.2)
> threshv <- qnorm(probv)
> # Simulate losses under the Vasicek model
> sysv <- rnorm(nsimu)
> assetm <- matrix(rnorm(nsimu*nbonds), ncol=nsimu)
> assetm <- t(rhos*sysv + t(rhosm*assetm))
> lossv <- lgd*colSums(assetm < threshv)/nbonds
```

VaR and *CVaR* Under the Vasicek Model

The function `density()` calculates a kernel estimate of the probability density for a sample of data, and returns a list with a vector of loss values and a vector of corresponding densities.

```
> # Calculate VaR from confidence level
> confl <- 0.95
> varisk <- quantile(lossv, confl)
> # Calculate the CVaR as the mean losses in excess of VaR
> cvar <- mean(lossv[lossv > varisk])
> # Plot the density of portfolio losses
> densv <- density(lossv, from=0)
> plot(densv, xlab="loss percentage", ylab="density",
+    cex.main=1.0, cex.lab=1.0, cex.axis=1.0,
+    lwd=3, col="blue", main="Portfolio Loss Distribution")
> # Add vertical line for expected loss
> exploss <- lgd*mean(probv)
> abline(v=exploss, col="red", lwd=3)
> xmax <- max(densv$x); ymax <- max(densv$y)
> text(x=exploss, y=(6*ymax/7), labels="expected loss",
+      lwd=2, pos=4, cex=1.0)
> # Add vertical line for VaR
> abline(v=varisk, col="red", lwd=3)
> text(x=varisk, y=4*ymax/5, labels="VaR", lwd=2, pos=4, cex=1.0)
```

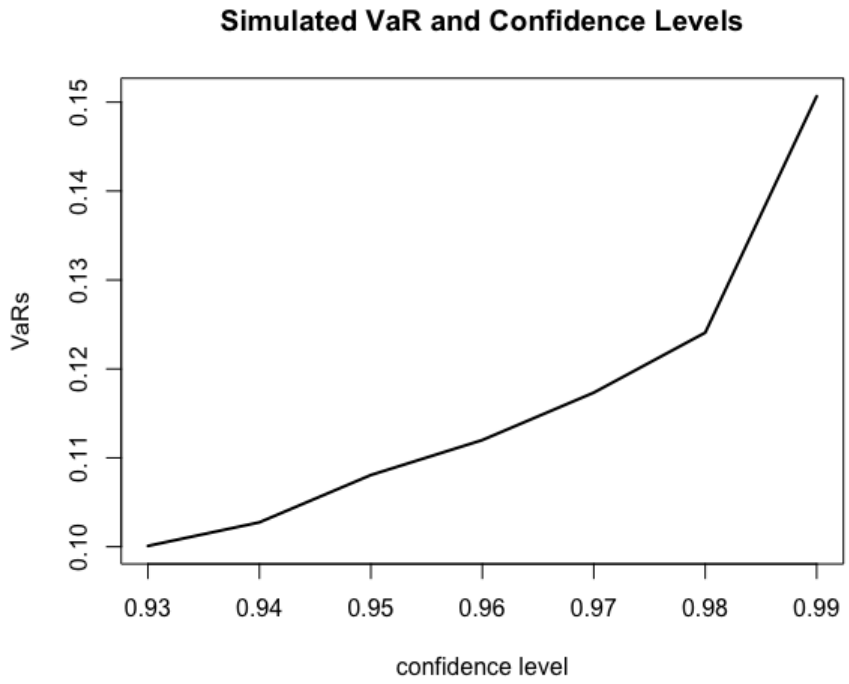



```
> # Draw shaded polygon for CVaR
> intail <- (densv$x > varisk)
> xvar <- c(min(densv$x[intail]), densv$x[intail], max(densv$x))
> polygon(xvar, c(-1, densv$y[intail], -1), col="red", border=NA, de
> # Add text for CVaR
> text(x=5*varisk/4, y=(ymax/7), labels="CVaR", lwd=2, pos=4, cex=1.
> # Add text with data
> text(xmax, ymax, labels=paste0(
+    "Expected Loss = ", format(100*exploss, digits=3), "%", "\n",
+    "Loss severity = ", format(100*lgd, digits=3), "%", "\n",
+    "Correlation = ", format(100*rho, digits=3), "%", "\n",
+    "VaR = ", format(100*varisk, digits=3), "%", "\n",
+    "CVaR = ", format(100*cvar, digits=3), "%"),
+    adj=c(1, 1), cex=1.0, lwd=2)
```

Simulating *VaR* Under the Vasicek Model

The *VaR* can be calculated from the simulated portfolio losses using the function `quantile()`.

The function `quantile()` calculates the sample quantiles. It uses interpolation to improve the accuracy. Information about the different interpolation methods can be found by typing `?quantile`.



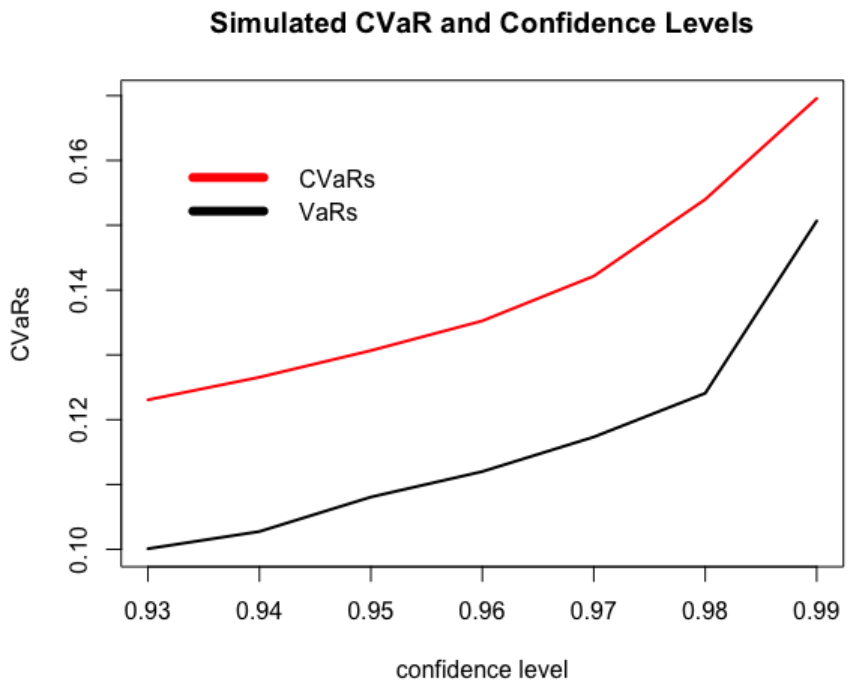

```
> # Calculate VaRs from confidence levels
> conv <- seq(0.93, 0.99, 0.01)
> vars <- quantile(lossv, probs=conv)
> plot(x=conv, y=vars, t="l", lwd=2,
+    xlab="confidence level", ylab="VaRs",
+    main="Simulated VaR and Confidence Levels")
```

Simulating *CVaR* Under the Vasicek Model

The *CVaR* can be calculated from the frequency of tail losses in excess of the *VaR*.

The function `table()` calculates the frequency distribution of categorical data.

```
> # Calculate CVaRs
> cvars <- sapply(vars, function(varisk) {
+   mean(lossv[lossv >= varisk])
+ })  # end sapply
> cvars <- cbind(cvars, vars)
> # Alternative CVaR calculation using frequency table
> # first calculate frequency table of losses
> # tablev <- table(lossv)/nsimu
> # Calculate CVaRs from frequency table
> # cvars <- sapply(vars, function(varisk) {
> #   tailrisk <- tablev[names(tablev) > varisk]
> #   tailrisk %*% as.numeric(names(tailrisk)) / sum(tailrisk)
> # })  # end sapply
```




```
> # Plot CVaRs
> plot(x=conv, y=cvars[, "cvars"],
+    t="l", col="red", lwd=2, ylim=range(cvars),
+    xlab="confidence level", ylab="CVaRs",
+    main="Simulated CVaR and Confidence Levels")
> # Add VaRs
> lines(x=conv, y=cvars[, "vars"], lwd=2)
> # Add legend
> legend(x="topleft", legend=c("CVaRs", "VaRs"), bty="n",
+    title=NULL, inset=0.05, cex=1.0, bg="white",
+    y.intersp=0.3, lwd=6, lty=1, col=c("red", "black"))
```

Function for Simulating *VaR* Under the Vasicek Model

The function `calc_var()` simulates default losses under the *Vasicek* model, for a vector of confidence levels, and calculates a vector of *VaR* and *CVaR* values.

```
> calc_var <- function(threshv, # Default thresholds
+    lgd=0.6, # loss given default
+    rhos, rhosm, # asset correlation
+    nsimu=1000, # number of simulations
+    conv=seq(0.93, 0.99, 0.01) # Confidence levels
+    ) {
+   # Define model parameters
+   nbonds <- NROW(threshv)
+   # Simulate losses under the Vasicek model
+   sysv <- rnorm(nsimu)
+   assetm <- matrix(rnorm(nsimu*nbonds), ncol=nsimu)
+   assetm <- t(rhos*sysv + t(rhosm*assetm))
+   lossv <- lgd*colSums(assetm < threshv)/nbonds
+   # Calculate VaRs and CVaRs
+   vars <- quantile(lossv, probs=conv)
+   cvars <- sapply(vars, function(varisk) {
+     mean(lossv[lossv >= varisk])
+   })  # end sapply
+   names(vars) <- conv
+   names(cvars) <- conv
+   c(vars, cvars)
+ }  # end calc_var
```

Standard Errors of *VaR* Using Bootstrap Simulation

The values of *VaR* and *CVaR* produced by the function `calc_var()` are subject to uncertainty because they're calculated from a simulation.

We can calculate the standard errors of *VaR* and *CVaR* by running the function `calc_var()` many times and repeating the simulation in a loop.

This bootstrap will only capture the uncertainty due to the finite number of trials in the simulation, but not due to the uncertainty of model parameters.

```
> # Define model parameters
> nbonds <- 300; nsimu <- 1000; lgd <- 0.4
> rho <- 0.2; rhos <- sqrt(rho); rhosm <- sqrt(1-rho)
> # Calculate default probabilities and thresholds
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> probv <- runif(nbonds, max=0.2)
> threshv <- qnorm(probv)
> conv <- seq(0.93, 0.99, 0.01)
> # Define number of bootstrap simulations
> nboot <- 500
> # Perform bootstrap of calc_var
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> bootd <- sapply(rep(lgd, nboot), calc_var,
+   threshv=threshv,
+   rhos=rhos, rhosm=rhosm,
+   nsimu=nsimu, conv=conv)  # end sapply
> bootd <- t(bootd)
> # Calculate standard errors of VaR and CVaR from bootd data
> varsd <- apply(bootd[, 1:7], MARGIN=2,
+     function(x) c(mean=mean(x), sd=sd(x)))
> cvarsd <- apply(bootd[, 8:14], MARGIN=2,
+     function(x) c(mean=mean(x), sd=sd(x)))
> # Scale the standard errors of VaRs and CVaRs
> varsds <- varsd[2, ]/varsd[1, ]
> cvarsds <- cvarsd[2, ]/cvarsd[1, ]
```

Standard Errors of *VaR* at High Confidence Levels

The standard errors of *VaR* and *CVaR* are inversely proportional to square root of the number of loss events in the simulation, that exceed the *VaR*.

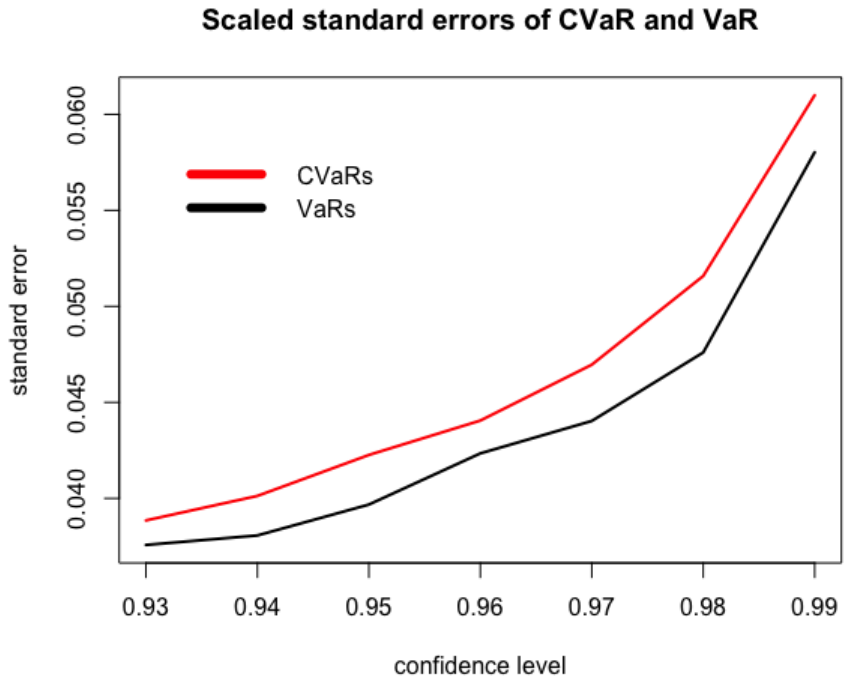
So the greater the number of loss events, the smaller the standard errors, and vice versa.

But as the confidence level increases, the *VaR* also increases, and the number of loss events decreases, causing larger standard errors.

So the as the confidence level increases, the standard errors of *VaR* and *CVaR* also increase.

The *scaled* (relative) standard errors of *VaR* and *CVaR* also increase with the confidence level, making them much less reliable at very high confidence levels.

The standard error of *CVaR* is even greater than that of *VaR*.

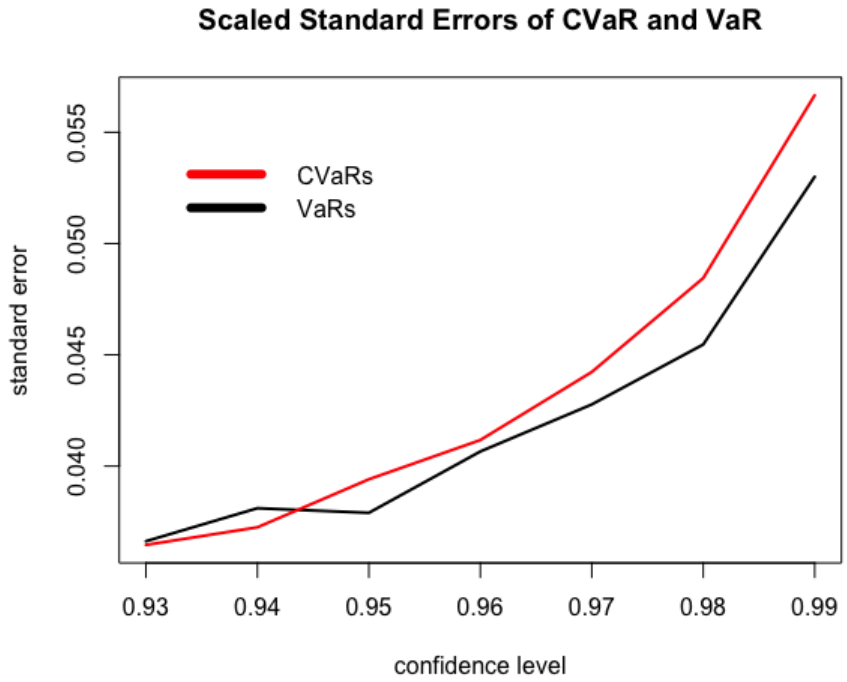



```
> # Plot the scaled standard errors of VaRs and CVaRs
> plot(x=names(varsds), y=varsds,
+   t="l", lwd=2, ylim=range(c(varsds, cvarsds)),
+   xlab="confidence level", ylab="standard error",
+   main="Scaled Standard Errors of CVaR and VaR")
> lines(x=names(cvarsds), y=cvarsds, lwd=2, col="red")
> legend(x="topleft", legend=c("CVaRs", "VaRs"), bty="n",
+    title=NULL, inset=0.05, cex=1.0, bg="white",
+    y.intersp=0.3, lwd=6, lty=1, col=c("red", "black"))
```

Standard Errors of *VaR* Using Parallel Bootstrap

The *scaled* standard errors of *VaR* and *CVaR* increase with the confidence level, making them much less reliable at very high confidence levels.

```
> library(parallel)  # load package parallel
> ncores <- detectCores() - 1  # number of cores
> compclust <- makeCluster(ncores)  # Initialize compute cluster
> # Perform bootstrap of calc_var for Windows
> clusterSetRNGStream(compclust, 1121)
> bootd <- parLapply(compclust, rep(lgd, nboot),
+   fun=calc_var, threshv=threshv,
+   rhos=rhos, rhosm=rhosm,
+   nsimu=nsimu, conv=conv)  # end parLapply
> stopCluster(compclust)  # Stop R processes over cluster
> # Bootstrap under Mac-OSX or Linux
> bootd <- mclapply(rep(lgd, nboot),
+   FUN=calc_var, threshv=threshv,
+   rhos=rhos, rhosm=rhosm,
+   nsimu=nsimu, conv=conv)  # end mclapply
> bootd <- rutils::do_call(rbind, bootd)
> # Calculate standard errors of VaR and CVaR from bootd data
> varsd <- apply(bootd[, 1:7], MARGIN=2,
+     function(x) c(mean=mean(x), sd=sd(x)))
> cvarsd <- apply(bootd[, 8:14], MARGIN=2,
+     function(x) c(mean=mean(x), sd=sd(x)))
> # Scale the standard errors of VaRs and CVaRs
> varsds <- varsd[2, ]/varsd[1, ]
> cvarsds <- cvarsd[2, ]/cvarsd[1, ]
```




```
> # Plot the standard errors of VaRs and CVaRs
> plot(x=names(varsds), y=varsds, t="l", lwd=2,
+   ylim=range(c(varsds, cvarsds)),
+   xlab="confidence level", ylab="standard error",
+   main="Scaled Standard Errors of CVaR and VaR")
> lines(x=names(cvarsds), y=cvarsds, lwd=2, col="red")
> legend(x="topleft", legend=c("CVaRs", "VaRs"), bty="n",
+    title=NULL, inset=0.05, cex=1.0, bg="white",
+    y.intersp=0.3, lwd=6, lty=1, col=c("red", "black"))
```

Vasicek Model With Uncertain Default Probabilities

The previous bootstrap only captured the uncertainty due to the finite simulation trials, but not due to the uncertainty of model parameters, such as the default probabilities and correlations.

The below function `calc_var()` can simulate the *Vasicek* model with uncertain default probabilities.

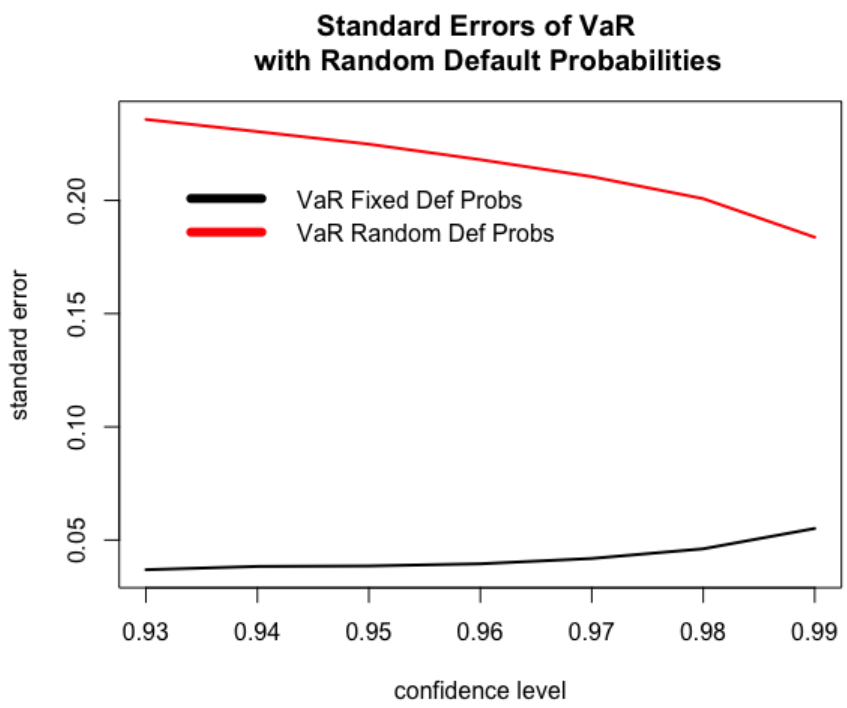
```
> calc_var <- function(probv, # Default probabilities
+    lgd=0.6, # loss given default
+    rhos, rhosm, # asset correlation
+    nsimu=1000, # number of simulations
+    conv=seq(0.93, 0.99, 0.01) # Confidence levels
+    ) {
+   # Calculate random default thresholds
+   threshv <- qnorm(runif(1, min=0.5, max=1.5)*probv)
+   # Simulate losses under the Vasicek model
+   nbonds <- NROW(probv)
+   sysv <- rnorm(nsimu)
+   assetm <- matrix(rnorm(nsimu*nbonds), ncol=nsimu)
+   assetm <- t(rhos*sysv + t(rhosm*assetm))
+   lossv <- lgd*colSums(assetm < threshv)/nbonds
+   # Calculate VaRs and CVaRs
+   vars <- quantile(lossv, probs=conv)
+   cvars <- sapply(vars, function(varisk) {
+     mean(lossv[lossv >= varisk])
+   })  # end sapply
+   names(vars) <- conv
+   names(cvars) <- conv
+   c(vars, cvars)
+ }  # end calc_var
```

Standard Errors Due to Uncertain Default Probabilities

The greatest contribution to the standard errors of *VaR* and *CVaR* is from the uncertainty of model parameters, such as the default probabilities, correlations, and loss severities.

For example, a 50% uncertainty in the default probabilities can produce a 20% uncertainty of the *VaR*.

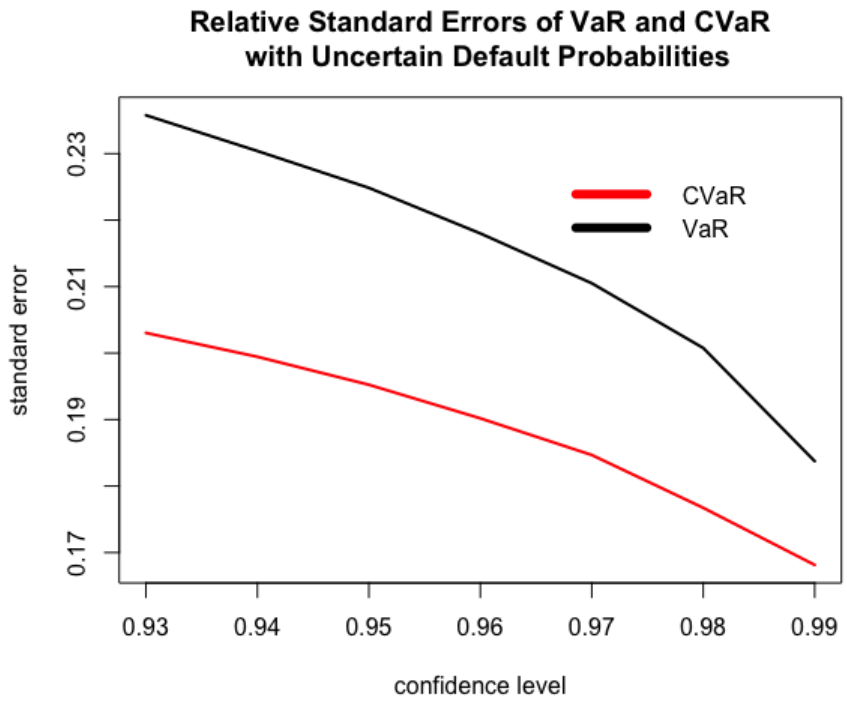
```
> library(parallel)  # load package parallel
> ncores <- detectCores() - 1  # number of cores
> compclust <- makeCluster(ncores)  # Initialize compute cluster
> # Perform bootstrap of calc_var for Windows
> clusterSetRNGStream(compclust, 1121)
> bootd <- parLapply(compclust, rep(lgd, nboot),
+   fun=calc_var, probv=probv,
+   rhos=rhos, rhosm=rhosm,
+   nsimu=nsimu, conv=conv)  # end parLapply
> stopCluster(compclust)  # Stop R processes over cluster
> # Bootstrap under Mac-OSX or Linux
> bootd <- mclapply(rep(lgd, nboot),
+   FUN=calc_var, probv=probv,
+   rhos=rhos, rhosm=rhosm,
+   nsimu=nsimu, conv=conv)  # end mclapply
> bootd <- rutils::do_call(rbind, bootd)
> # Calculate standard errors of VaR and CVaR from bootd data
> varsd <- apply(bootd[, 1:7], MARGIN=2,
+     function(x) c(mean=mean(x), sd=sd(x)))
> cvarsd <- apply(bootd[, 8:14], MARGIN=2,
+     function(x) c(mean=mean(x), sd=sd(x)))
> # Scale the standard errors of VaRs and CVaRs
> varsdsu <- varsd[2, ]/varsd[1, ]
> cvarsdsu <- cvarsd[2, ]/cvarsd[1, ]
```




```
> # Plot the standard errors of VaRs under uncertain default probabi
> plot(x=colnames(varsd), y=varsds, t="l",
+  col="black", lwd=2, ylim=range(c(varsds, varsdsu)),
+   xlab="confidence level", ylab="standard error",
+   main="Standard Errors of VaR
+   with Random Default Probabilities")
> lines(x=colnames(varsd), y=varsdsu, lwd=2, col="red")
> legend(x="topleft",
+    legend=c("VaR Fixed Def Probs", "VaR Random Def Probs"),
+    bty="n", title=NULL, inset=0.05, cex=1.0, bg="white",
+    y.intersp=0.3, lwd=6, lty=1, col=c("black", "red"))
```

Relative Errors Due to Uncertain Default Probabilities

The *scaled* (relative) standard errors of *VaR* and *CVaR* under uncertain default probabilities decrease with higher confidence level, because the standard errors are less dependent on the confidence level and don't increase as fast as the *VaR* does.




```
> # Plot the standard errors of VaRs and CVaRs
> plot(x=colnames(varsd), y=varsdsu, t="l", lwd=2,
+   ylim=range(c(varsdsu, cvarsdsu)),
+   xlab="confidence level", ylab="standard error",
+   main="Relative Standard Errors of VaR and CVaR
+   with Uncertain Default Probabilities")
> lines(x=colnames(varsd), y=cvarsdsu, lwd=2, col="red")
> legend(x="topright", legend=c("CVaR", "VaR"), bty="n",
+    title=NULL, inset=0.05, cex=1.0, bg="white",
+    y.intersp=0.3, lwd=6, lty=1, col=c("red", "black"))
```

Model Risk of Credit Portfolio Models

Credit portfolio models are subject to very significant *model risk* due to the uncertainties of model parameters, such as the default probabilities, correlations, and loss severities.

Model risk is the risk of incorrect model predictions due to incorrect model specification, and due to incorrect model parameters.

Jon Danielsson at the London School of Economics (LSE) has studied the model risk of *VaR* and *CVaR* in: Why Risk is So Hard to Measure, and in Model Risk of Risk Models.

Jon Danielsson has pointed out that there's not enough historical data to be able to accurately calculate the credit model parameters.

Jon Danielsson and Chen Zhou have demonstrated that accurately estimating *CVaR* at 5% confidence would require decades of price history, something that simply doesn't exist for many assets.

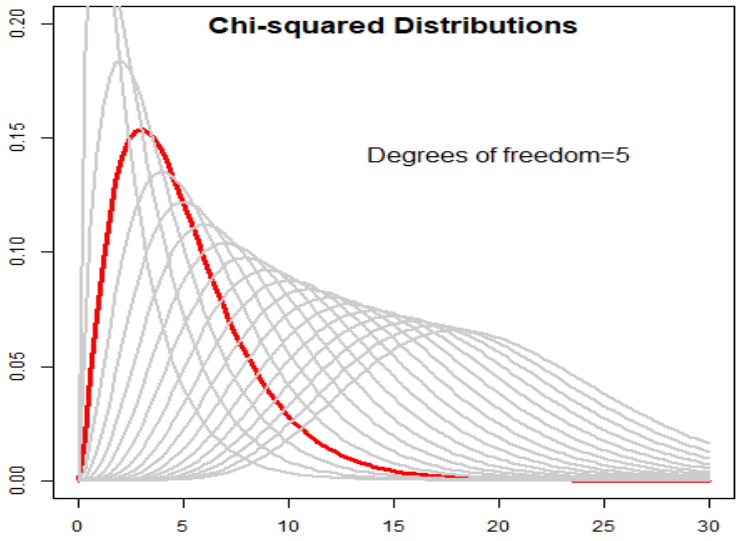
Plotting Using Expression Objects

It's sometimes convenient to create an *expression* object containing plotting commands, to be able to later create plots using it.

The function `quote()` produces an *expression* object without evaluating it.

The function `eval()` evaluates an *expression* in a specified *environment*.

```
> # Create a plotting expression
> expv <- quote({
+   degf <- 2:20
+   rangev <- (1:NROW(degf))
+   indeks <- 4
+   # Plot a curve
+   curve(expr=dchisq(x, df=degf[indeks]),
+ xlim=c(0, 30), ylim=c(0, 0.2),
+ xlab="", ylab="", lwd=3, col="red")
+   # Add grey lines to plot
+   for (it in rangev[-indeks]) {
+     curve(expr=dchisq(x, df=degf[it]),
+   xlim=c(0, 30), ylim=c(0, 0.2),
+   xlab="", ylab="", lwd=2, col="grey80", add=TRUE)
+   }  # end for
+   # Add title
+   title(main="Chi-squared Distributions", line=-1.5, cex.main=1.5)
+   # Add legend
+   text(x=20, y=0.15, labels=paste0("Degrees of freedom=",
+       degf[indeks]), pos=1, cex=1.3)
+ })  # end quote
```




```
> # View the plotting expression
> expv
> # Create plot by evaluating the plotting expression
> x11(width=6, height=4)
> eval(expv)
```

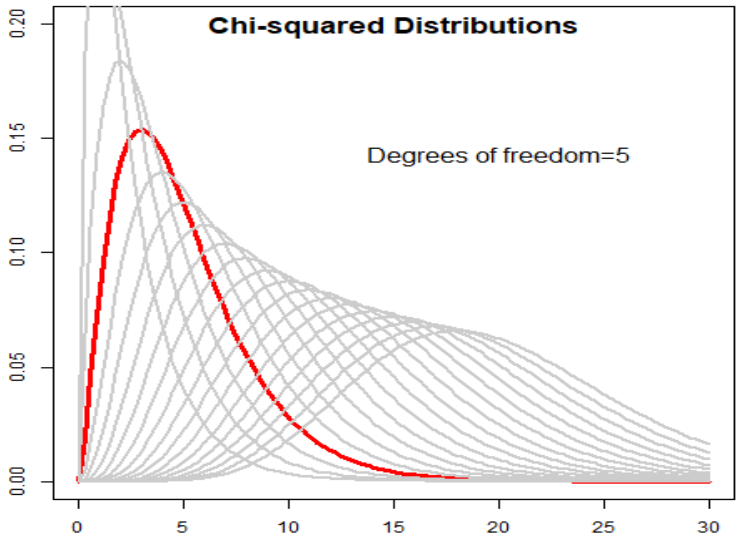
Animated Plots Using Package *animation*

The package *animation* allows creating animated plots in the form of *gif* and *html* documents.

The function `saveGIF()` produces a *gif* image with an animated plot.

The function `saveHTML()` produces an *html* document with an animated plot.

```
> library(animation)
> # Create an expression for creating multiple plots
> expv <- quote({
+   degf <- 2:20
+   rangev <- (1:NROW(degf))
+   # Set image refesh interval
+   animation::ani.options(interval=0.5)
+   # Create multiple plots with curves
+   for (indeks in rangev) {
+     curve(expr=dchisq(x, df=degf[indeks]),
+   xlim=c(0, 30), ylim=c(0, 0.2),
+   xlab="", ylab="", lwd=3, col="red")
+     # Add grey lines to plot
+     for (it in rangev[-indeks]) {
+       curve(expr=dchisq(x, df=degf[it]),
+     xlim=c(0, 30), ylim=c(0, 0.2),
+     xlab="", ylab="", lwd=2, col="grey80", add=TRUE)
+     }  # end for
+     # Add title
+     title(main="Chi-squared Distributions", line=-1.5, cex.main=
+     # Add legend
+     text(x=20, y=0.15, labels=paste0("Degrees of freedom=",
+       degf[indeks]), pos=1, cex=1.3)
+   }  # end for
+ })  # end quote
```




```
> # Create plot by evaluating the plotting expression
> x11(width=6, height=4)
> eval(expv)
> # Create gif with animated plot
> animation::saveGIF(expr=eval(expv),
+   movie.name="chi_squared.gif",
+   img.name="chi_squared")
> # Create html with animated plot
> animation::saveHTML(expr=eval(expv),
+   img.name="chi_squared",
+   htmlfile="chi_squared.html",
+   description="Chi-squared Distributions")  # end saveHTML
```

Dynamic Documents Using *R markdown*

markdown is a simple markup language designed for creating documents in different formats, including *pdf* and *html*.

R Markdown is a modified version of *markdown*, which allows creating documents containing *math formulas* and `R` code embedded in them.

An *R Markdown* document (with extension `.Rmd`) contains:

- A *YAML* header,
- Text in *R Markdown* code format,
- Math formulas (equations), delimited using either single "$" symbols (for inline formulas), or double "$$" symbols (for display formulas),
- `R` code chunks, delimited using either single "`" backtick symbols (for inline code), or triple "```" backtick symbols (for display code).

The packages *rmarkdown* and *knitr* compile `R` documents into either *pdf*, *html*, or *MS Word* documents.

````
---
title: "My First R Markdown Document"
author: Jerzy Pawlowski
date: ''`r format(Sys.time(), "%m/%d/%Y")`''
output: html_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)

# install package quantmod if it can't be loaded success
if (!require("quantmod"))
  install.packages("quantmod")
```

### R Markdown
This is an *R Markdown* document. Markdown is a simple f

One of the advantages of writing documents *R Markdown*

You can read more about publishing documents using *R* h
https://algoquant.github.io/r,/markdown/2016/07/02/Publi

You can read more about using *R* to create *HTML* docum
https://algoquant.github.io/2016/07/05/Interactive-Plots

Clicking the **Knit** button in *RStudio*, compiles the

Example of an *R* code chunk:
```{r cars}
summary(cars)
```

### Plots in *R Markdown* documents

Plots can also be embedded, for example:
```{r pressure, echo=FALSE}
plot(pressure)
````
````

Package *shiny* for Creating Interactive Applications

The package *shiny* creates interactive applications running in R, with their outputs presented as live visualizations.

Shiny allows changing the model parameters, recalculating the model, and displaying the resulting outputs as plots and charts.

A *shiny app* is a file with *shiny* commands and R code.

The *shiny* code consists of a *shiny interface* and a *shiny server*.

The *shiny interface* contains widgets for data input and an area for plotting.

The *shiny server* contains the R model code and the plotting code.

The function `shiny::fluidPage()` creates a GUI layout for the user inputs of model parameters and an area for plots and charts.

The function `shiny::renderPlot()` renders a plot from the outputs of a live model.

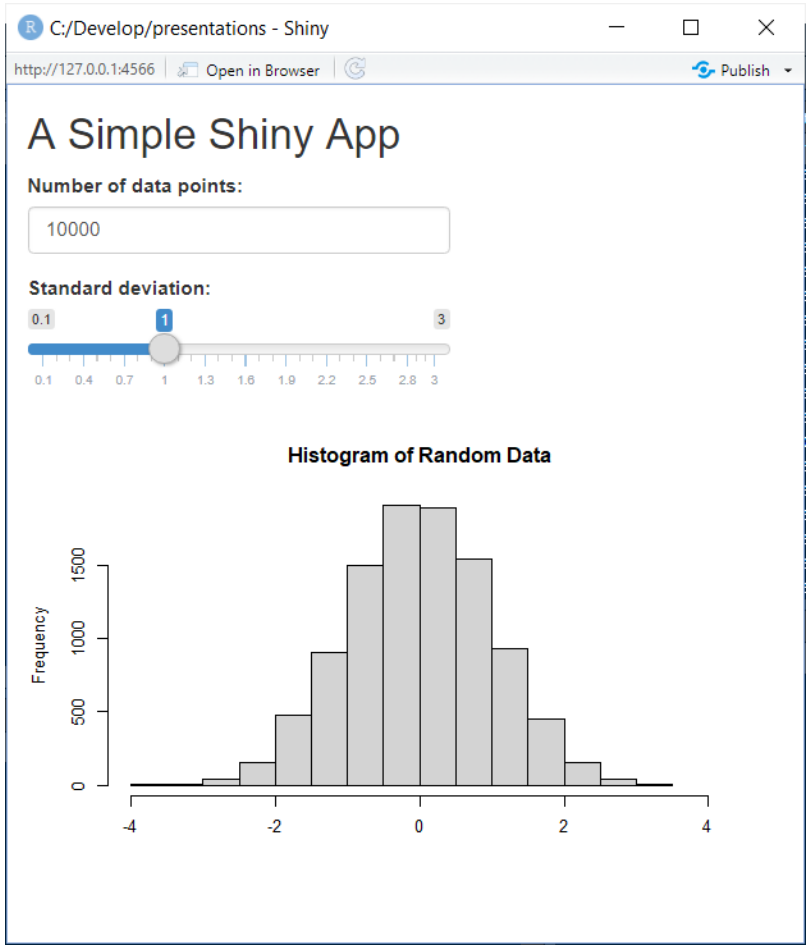
The function `shiny::shinyApp()` creates a `shiny` app from a *shiny interface* and a *shiny server*.

```
> ## App setup code that runs only once at startup.
> ndata <- 1e4
> stdev <- 1.0
>
> ## Define the user interface
> uiface <- shiny::fluidPage(
+   # Create numeric input for the number of data points.
+   numericInput("ndata", "Number of data points:", value=ndata),
+   # Create slider input for the standard deviation parameter.
+   sliderInput("stdev", label="Standard deviation:",
+         min=0.1, max=3.0, value=stdev, step=0.1),
+   # Render plot in a panel.
+   plotOutput("plotobj", height=300, width=500)
+ )  # end user interface
>
> ## Define the server function
> servfun <- function(input, output) {
+   output$plotobj <- shiny::renderPlot({
+     # Simulate the data
+     datav <- rnorm(input$ndata, sd=input$stdev)
+     # Plot the data
+     par(mar=c(2, 4, 4, 0), oma=c(0, 0, 0, 0))
+     hist(datav, xlim=c(-4, 4), main="Histogram of Random Data")
+   })  # end renderPlot
+ }  # end servfun
>
> # Return a Shiny app object
> shiny::shinyApp(ui=uiface, server=servfun)
```

Running Shiny Apps in *RStudio*

A *shiny app* can be run by pressing the "Run App" button in *RStudio*.

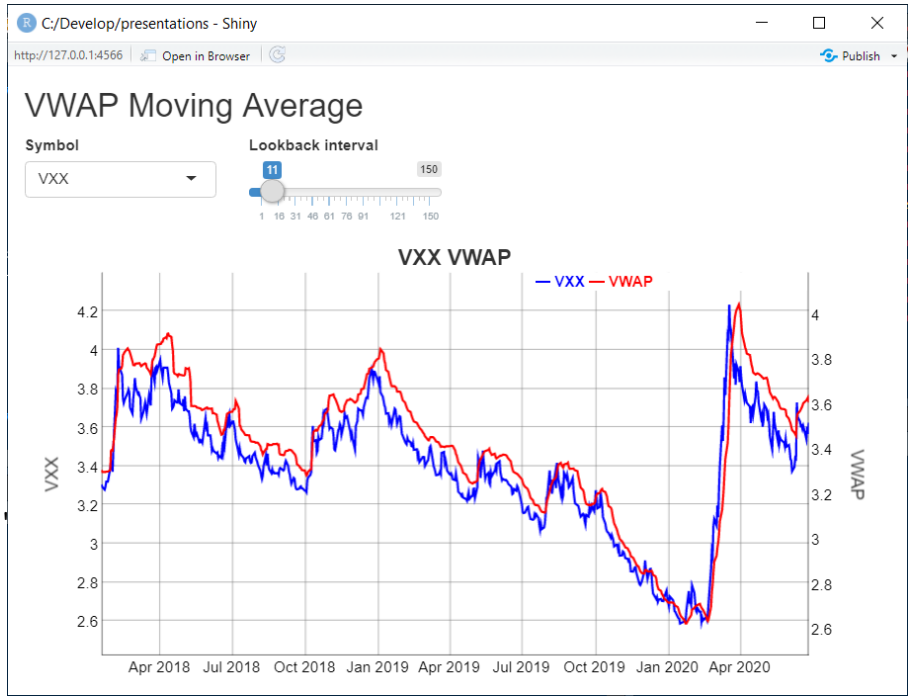
When the *shiny app* is run, the *shiny* commands are translated into *JavaScript* code, which creates a graphical user interface (GUI) with buttons, sliders, and boxes for data input, and also with the output plots and charts.



Positioning and Sizing Widgets Within the Shiny GUI

The functions `shiny::fluidRow()` and `shiny::column()` allow positioning and sizing widgets within the *shiny* GUI.

```
> ## Create elements of the user interface
> uiface <- shiny::fluidPage(
+   titlePanel("VWAP Moving Average"),
+   # Create single row of widgets with two slider inputs
+   fluidRow(
+     # Input stock symbol
+     column(width=3, selectInput("symbol", label="Symbol",
+                           choices=symbolv, selected=symbol)),
+     # Input look-back interval
+     column(width=3, sliderInput("lookb", label="Lookback interval"
+                           min=1, max=150, value=11, step=1))
+   ),  # end fluidRow
+   # Create output plot panel
+   mainPanel(dygraphs::dygraphOutput("dyplot"), width=12)
+ )  # end fluidPage interface
```



Shiny Apps With Reactive Expressions

The package *shiny* allows specifying reactive expressions which are evaluated only when their input data is updated.

Reactive expressions avoid performing unnecessary calculations.

If the reactive expression is invalidated (recalculated), then other expressions that depend on its output are also recalculated.

This way calculations cascade through the expressions that depend on each other.

The function `shiny::reactive()` transforms an expression into a reactive expression.

```
> ## Define the server function
> servfun <- shiny::shinyServer(function(input, output) {
+   # Get the close and volume data in a reactive environment
+   closep <- shiny::reactive({
+     # Get the data
+     ohlc <- get(input$symbol, data_env)
+     closep <- log(quantmod::Cl(ohlc))
+     volum <- quantmod::Vo(ohlc)
+     # Return the data
+     cbind(closep, volum)
+   })  # end reactive code
+
+   # Calculate the VWAP indicator in a reactive environment
+   vwapv <- shiny::reactive({
+     # Get model parameters from input argument
+     lookb <- input$lookb
+     # Calculate the VWAP indicator
+     closep <- closep()[, 1]
+     volum <- closep()[, 2]
+     vwapv <- HighFreq::roll_sum(tseries=closep*volum, lookb=lookb)
+     volumroll <- HighFreq::roll_sum(tseries=volum, lookb=lookb)
+     vwapv <- vwapv/volumroll
+     vwapv[is.na(vwapv)] <- 0
+     # Return the plot data
+     datav <- cbind(closep, vwapv)
+     colnames(datav) <- c(input$symbol, "VWAP")
+     datav
+   })  # end reactive code
+
+   # Return the dygraph plot to output argument
+   output$dyplot <- dygraphs::renderDygraph({
+     colv <- colnames(vwapv())
+     dygraphs::dygraph(vwapv(), main=paste(colv[1], "VWAP")) %>%
+ dyAxis("y", label=colv[1], independentTicks=TRUE) %>%
+ dyAxis("y2", label=colv[2], independentTicks=TRUE) %>%
+ dySeries(name=colv[1], axis="y", label=colv[1], strokeWidth=2, col
+ dySeries(name=colv[2], axis="y2", label=colv[2], strokeWidth=2, co
+   })  # end output plot
+ })  # end server code
```

Reactive Event Handlers

Event handlers are functions which evaluate expressions when an event occurs (like a button press).

The functions `shiny::observeEvent()` and `shiny::eventReactive()` are event handlers.

The function `shiny::eventReactive()` returns a value, while `shiny::observeEvent()` produces a side-effect, without returning a value.

The function `shiny::reactiveValues()` creates a list for storing reactive values, which can be updated by event handlers.

```
> ## Define the server function
> servfun <- shiny::shinyServer(function(input, output) {
+
+   # Create an empty list of reactive values.
+   value_s <- reactiveValues()
+
+   # Get input parameters from the user interface.
+   nrows <- reactive({
+     # Add nrows to list of reactive values.
+     value_s*nrows <- input$nrows
+     input$nrows
+   })  # end reactive code
+
+   # Broadcast a message to the console when the button is pressed.
+   observeEvent(eventExpr=input$button, handlerExpr={
+     cat("Input button pressed\n")
+   })  # end observeEvent
+
+   # Send the data when the button is pressed.
+   datav <- eventReactive(eventExpr=input$button, valueExpr={
+     # eventReactive() executes on input$button, but not on nrows()
+     cat("Sending", nrows(), "rows of data\n")
+     datav <- head(mtcars, input$nrows)
+     value_s$mpg <- mean(datav$mpg)
+     datav
+   })  # end eventReactive
+   #   datav
+
+   # Draw table of the data when the button is pressed.
+   observeEvent(eventExpr=input$button, handlerExpr={
+     datav <- datav()
+     cat("Received", value_s*nrows, "rows of data\n")
+     cat("Average mpg = ", value_s$mpg, "\n")
+     cat("Drawing table\n")
+     output$tablev <- renderTable(datav)
+   })  # end observeEvent
+
+ })  # end server code
>
```

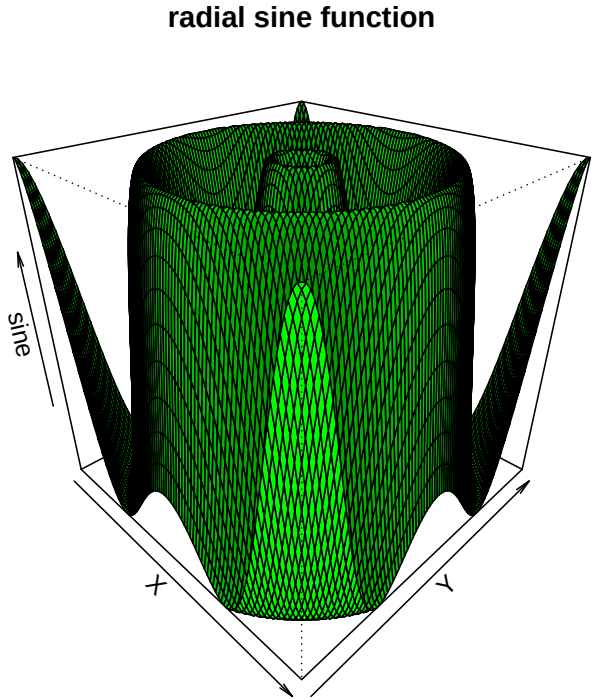
Plotting 3d Perspective Surface Plots

The function `persp()` plots a 3d perspective surface plot of a function specified over a grid of argument values.

The argument *z* accepts a matrix containing the function values.

`persp()` belongs to the base *graphics* package, and doesn't create interactive plots.

```
> # Define function of two variables
> fun2d <- function(x, y) sin(sqrt(x^2+y^2))
> # Calculate function over matrix grid
> xlim <- seq(from=-10, to=10, by=0.2)
> ylim <- seq(from=-10, to=10, by=0.2)
> # Draw 3d surface plot of function
> persp(z=outer(xlim, ylim, FUN=fun2d),
+ theta=45, phi=30, zlab="sine",
+ shade=0.1, col="green",
+ main="radial sine function")
```



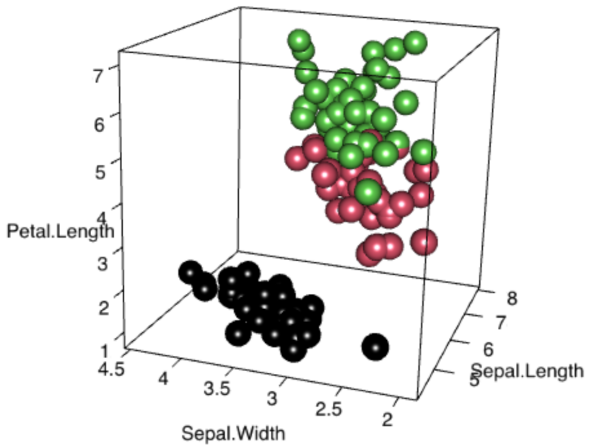
Interactive 3d Scatter Plots Using Package *rgl*

The package *rgl* creates *interactive* 3d scatter plots and surface plots by calling the *WebGL JavaScript* library.

WebGL is a *JavaScript* library which creates plots by calling graphics code written in the *OpenGL* language.

The function `rgl::plot3d()` plots an *interactive* 3d scatter plot from three vectors of data.

```
> # Load package rgl
> library(rgl)
> # Set rgl options
> options(rgl.useNULL=TRUE)
> # Create 3d scatter plot of function
> with(iris, rgl::plot3d(Sepal.Length, Sepal.Width, Petal.Length,
+             type="s", col=as.numeric(Species)))
> # Render the 3d scatter plot of function
> rgl::rglwidget(elementId="plot3drgl", width=1000, height=1000)
```

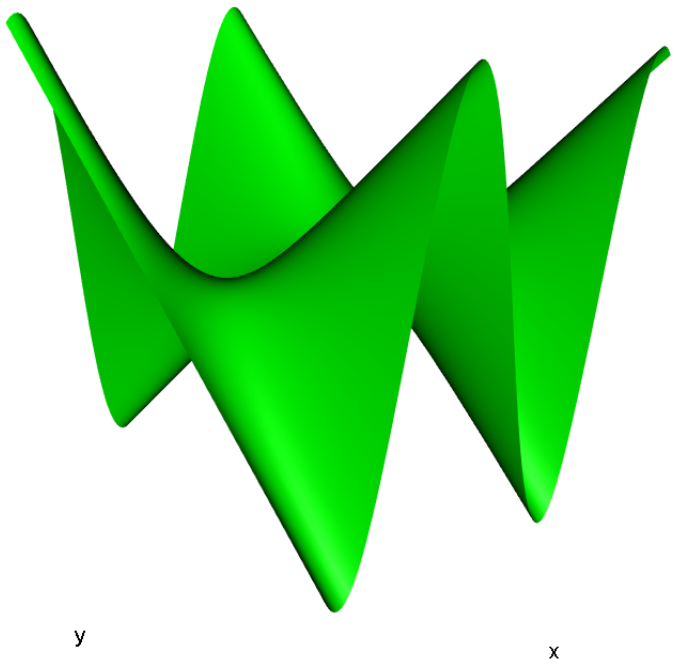


Interactive 3d Surface Plots Using Package *rgl*

The function `rgl::persp3d()` plots an *interactive* 3d surface plot of a *vectorized* function or a matrix.

rgl is an `R` package for 3d and perspective plotting, based on the *OpenGL* framework.

```
> library(rgl)  # Load rgl
> # Define function of two variables
> fun2d <- function(x, y) y*sin(x)
> # Create 3d surface plot of function
> rgl::persp3d(x=fun2d, xlim=c(-5, 5), ylim=c(-5, 5),
+   col="green", axes=FALSE)
> # Render the 3d surface plot of function
> rgl::rglwidget(elementId="surfacergl", width=500, height=500)
> # Draw 3d surface plot of matrix
> xlim <- seq(from=-5, to=5, by=0.1)
> ylim <- seq(from=-5, to=5, by=0.1)
> rgl::persp3d(z=outer(xlim, ylim, FUN=fun2d),
+   xlab="x", ylab="y", zlab="fun2d", col="green")
> # Render the 3d surface plot of function
> rgl::rglwidget(elementId="plot3drgl", width=1000, height=1000)
> # Save current view to png file
> rgl::rgl.snapshot("surface_plot.png")
> # Define function of two variables and two parameters
> fun2d <- function(x, y, lambdaf1=1, lambdaf2=1)
+   sin(lambdaf1*x)*sin(lambdaf2*y)
> # Draw 3d surface plot of function
> rgl::persp3d(x=fun2d, xlim=c(-5, 5), ylim=c(-5, 5),
+   col="green", axes=FALSE, lambdaf1=1, lambdaf2=2)
> # Render the 3d surface plot of function
> rgl::rglwidget(elementId="plot3drgl", width=1000, height=1000)
```



Homework Assignment

Required

- Study all the lecture slides in *FRE6871_Lecture_3.pdf*, and run all the code in *FRE6871_Lecture_3.R*
- Read about the *bootstrap technique* in:
  bootstrap_technique.pdf and *doBootstrap_primer.pdf*
- Read about applying the *importance sampling technique* for calculating *CVaR*:
  Muller CVAR Importance Sampling.pdf

Recommended

- Read about why *CVaR* is a coherent risk measure:
  https://en.wikipedia.org/wiki/Expected_shortfall
  https://en.wikipedia.org/wiki/Coherent_risk_measure#Value_at_risk
- Read about why *CVaR* has very large standard errors:
  Danielsson CVAR Estimation Standard Error.pdf
  http://www.bloomberg.com/view/articles/2016-05-23/big-banks-risk-does-not-compute