FRE6871 R in Finance

Lecture#3, Fall 2023

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

September 25, 2023

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)
> nbonds <- 100
> defprobs <- runif(nbonds, max=0.2)
> mean(defprobs)
> # Simulate number of defaults
> unifv <- runif(nbonds)
> sum(unifv < defprobs)
> # Simulate average number of defaults using for() loop (inefficien
> nsimu <- 1000
> set.seed(1121)
> defaultv <- numeric(nsimu)
> for (i in 1:nsimu) {  # Perform loop
+   unifv <- runif(nbonds)
+   defaultv[i] <- sum(unifv < defprobs)
+ }  # end for
> # Calculate average number of defaults
> mean(defaultv)
> # Simulate using vectorized functions (efficient way)
> set.seed(1121)
> unifm <- matrix(runif(nsimu*nbonds), ncol=nsimu)
> defaultv <- colSums(unifm < defprobs)
> mean(defaultv)
> # Plot the distribution of defaults
> x11(width=6, height=5)
> plot(density(defaultv), main="Distribution of Defaults",
+      xlab="number of defaults", ylab="frequqncy")
> 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).

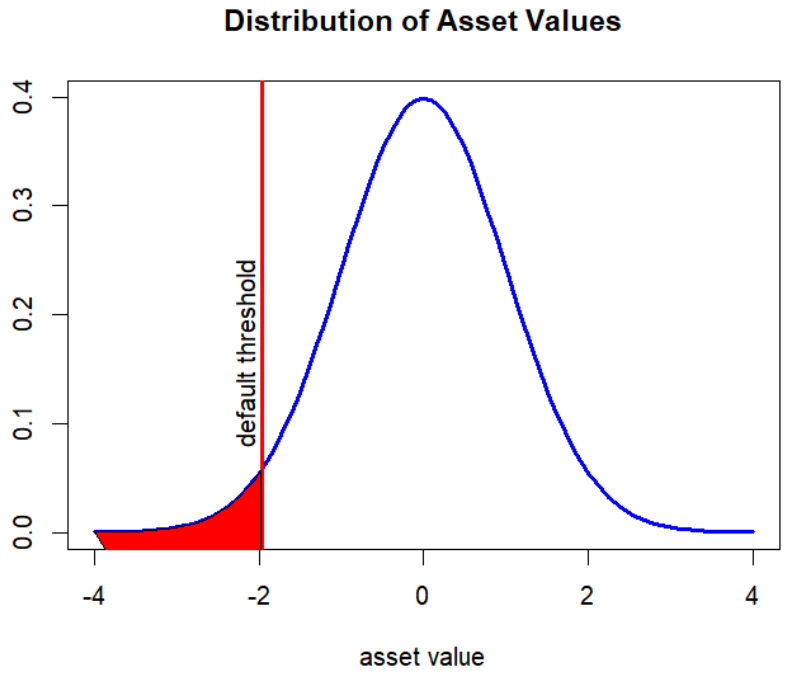
Similarly, the default probabilities p_i are also transformed into *default thresholds* t_i, which are the *quantiles*: $t_i = \Phi^{-1}(p_i)$.

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
> defthresh <- qnorm(defprobs)
> assetm <-qnorm(unifm)
> # Simulate defaults
> defaultv <- colSums(assetm < defthresh)
> mean(defaultv)
```




```
> # Plot Standard Normal distribution
> x11(width=6, height=5)
> xlim <- 4; defthresh <- 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=defthresh, col="red", lwd=3)
> text(x=defthresh-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 <= defthresh))
> polygon(c(xlim, xvar[intail], defthresh),
+   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)
> 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)
> 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
> defprobs <- runif(nbonds, max=0.2)
> mean(defprobs)
> # Calculate default thresholds
> defthresh <- qnorm(defprobs)
> # Calculate number of defaults using vectorized functions (efficie
> # Calculate vector of number of defaults
> rowMeans(assetm < defthresh)
> defprobs
> # 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] < defthresh)
+ }  # end for
> rowMeans(defaultm)
> rowMeans(assetm < defthresh)
> # 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
> defthresh <- 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 < defthresh)
> # 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:

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

Cumulative Default Distribution And Correlation

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

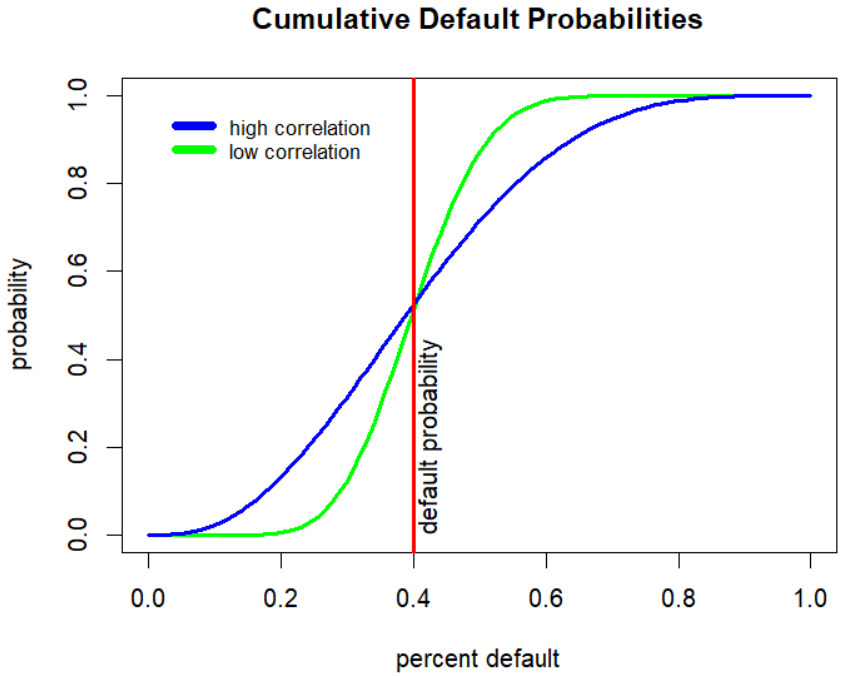
$$P(x) = \Phi\left(\frac{\sqrt{1-\rho}\,\Phi^{-1}(x) - t}{\sqrt{\rho}}\right)$$

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, defthresh=(-2), rho=0.2)
+   pnorm((sqrt(1-rho)*qnorm(x) - defthresh)/sqrt(rho))
> defprob <- 0.4; defthresh <- qnorm(defprob)
> cumdefdistr(x=0.2, defthresh=qnorm(defprob), rho=rho)
> # Plot cumulative default distribution function
> curve(expr=cumdefdistr(x, defthresh=defthresh, 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, defthresh=defthresh, 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)
```

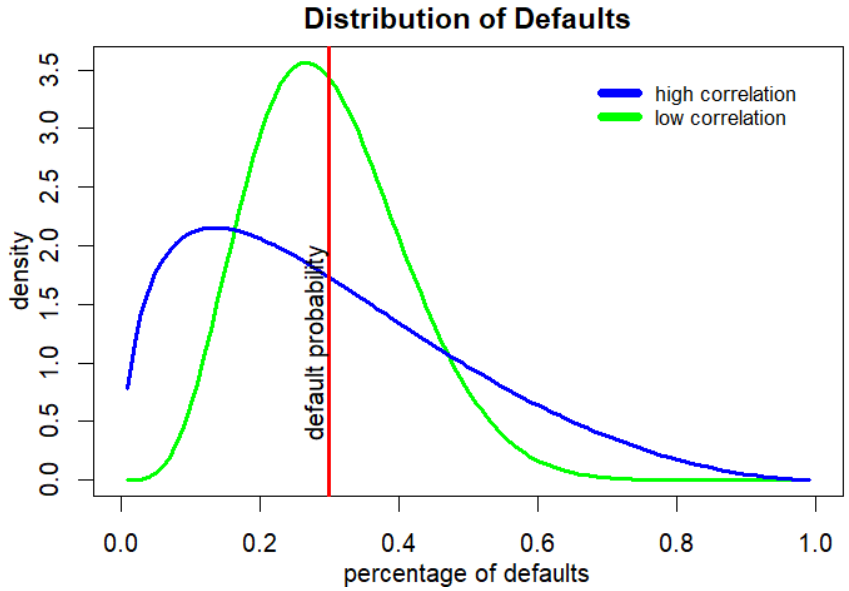
Distribution of Defaults Under the Vasicek Model

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, defthresh=(-2), rho=0.2)
+   sqrt((1-rho)/rho)*exp(-(sqrt(1-rho)*qnorm(x) -
+   defthresh)^2/(2*rho) + qnorm(x)^2/2)
> # Define parameters
> rho <- 0.2 ; rhos <- sqrt(rho) ; rhosm <- sqrt(1-rho)
> defprob <- 0.3; defthresh <- qnorm(defprob)
> defdistr(0.03, defthresh=defthresh, rho=rho)
> # Plot probability distribution of defaults
> curve(expr=defdistr(x, defthresh=defthresh, 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, defthresh=defthresh, 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)
```

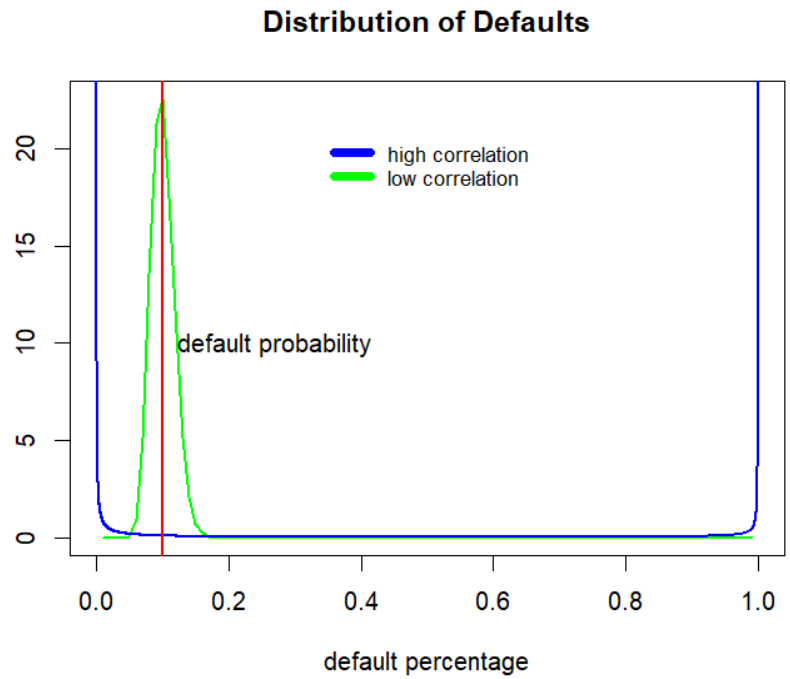
Distribution of Defaults Under Extreme Correlations

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, defthresh=defthresh, 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, defthresh=defthresh, 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 lambda
> func <- function(x, lambda=1) {
+   exp(-x*lambda)
+ }  # end func
> integrate(func, lower=0, upper=Inf)
> integrate(func, lower=0, upper=Inf, lambda=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*):

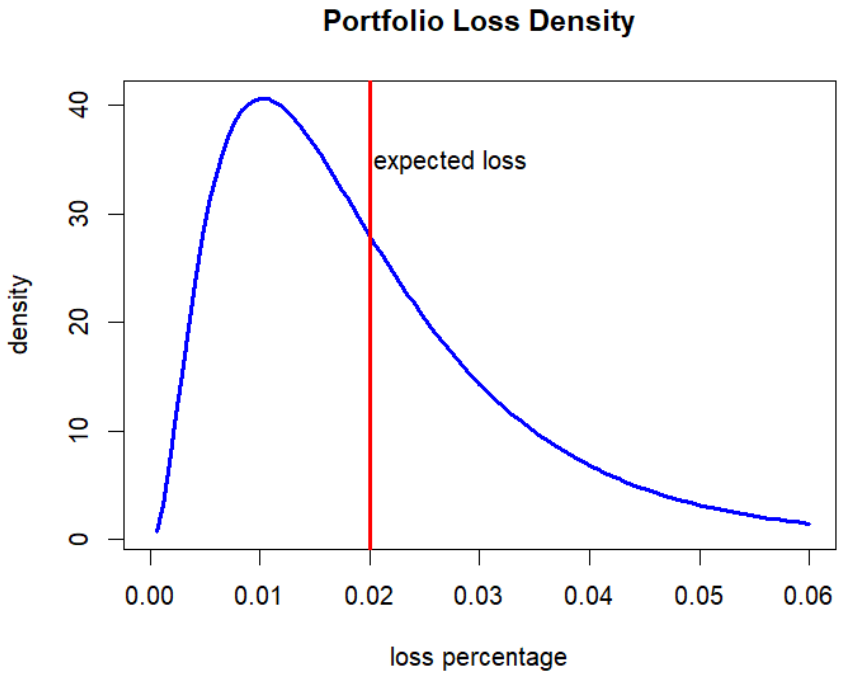
$$EL = \sum_{i=1}^{n} p_i LGD_i$$

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

```
> # Plot probability distribution of losses
> x11(width=6, height=5)
> curve(expr=lossdistr(x, defthresh=defthresh, 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.

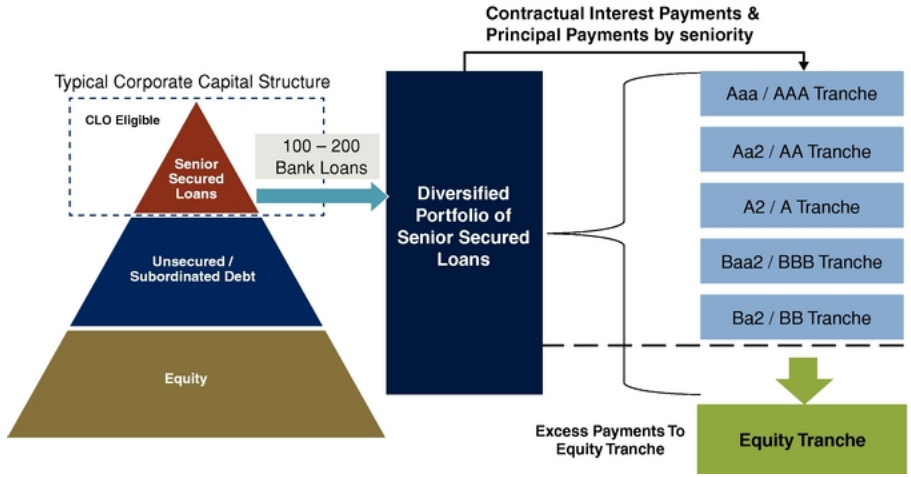
The *CDO* liabilities are *CDO* tranches, which receive cashflows from the *CDO* assets, and are exposed to their defaults.

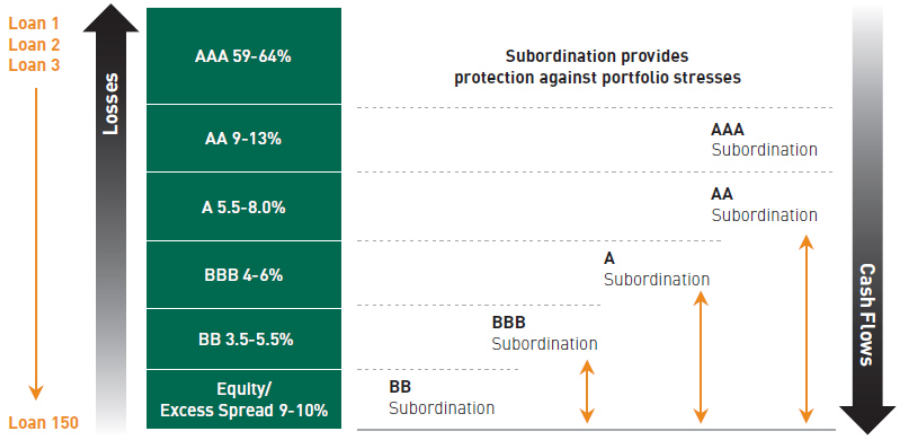
CDO tranches have an attachment point (subordination, i.e. the percentage of asset default losses at which the tranche starts absorbing those losses), and a detachment point when the tranche is wiped out (suffers 100% losses).

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.

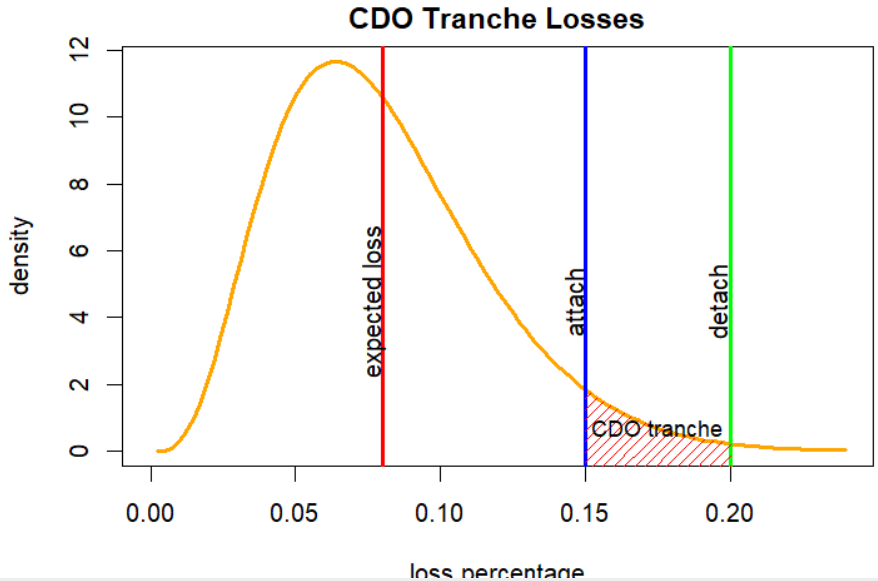
The expected loss *EL* on a *CDO* tranche is:

$$EL = \frac{1}{d-a}\int_a^d (x-a)\,f(x)\,\mathrm{d}x + \int_d^{LGD} f(x)\,\mathrm{d}x$$

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

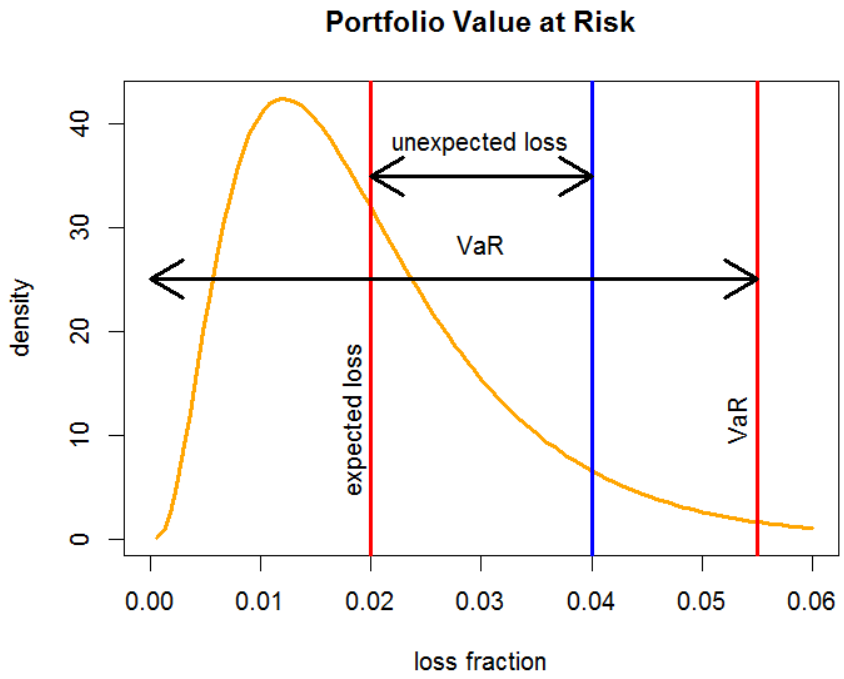
```
> defprob <- 0.2; defthresh <- qnorm(defprob)
> rho <- 0.1; lgd <- 0.4
> attachp <- 0.15; detachp <- 0.2
> # Expected tranche loss is sum of two terms
> tranchel <-
+   # Loss between attachp and detachp
+   integrate(function(x, attachp) (x-attachp)*lossdistr(x,
+ defthresh=defthresh, rho=rho, lgd=lgd),
+ low=attachp, up=detachp, attachp=attachp)$value/(detachp-attachp)
+   # Loss in excess of detachp
+   (1-cumlossdistr(x=detachp, defthresh=defthresh, rho=rho, lgd=lgc
> # Plot probability distribution of losses
> curve(expr=lossdistr(x, defthresh=defthresh, 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",
```

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)
```

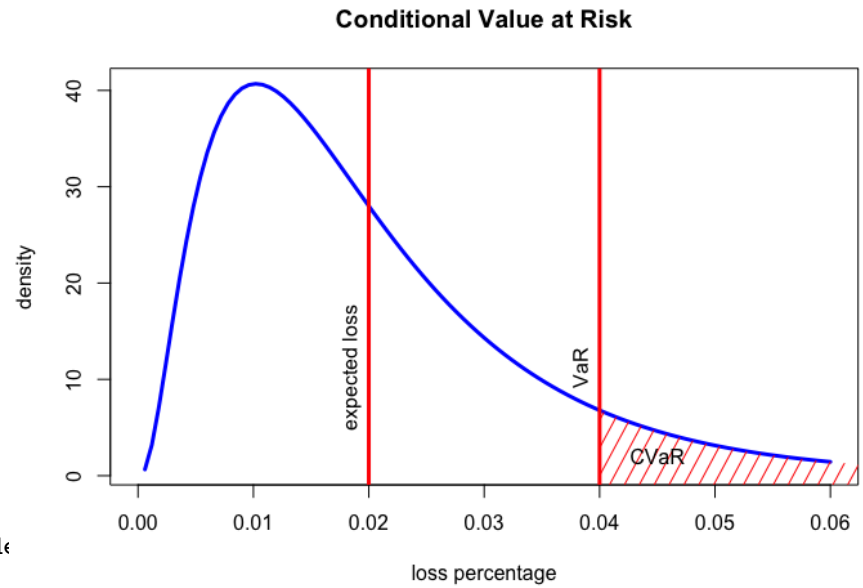


Conditional Value at Risk

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

$$\mathrm{CVaR} = \frac{1}{\alpha}\int_0^{\alpha} \mathrm{VaR}(p)\,\mathrm{d}p = \frac{1}{\alpha}\int_{\mathrm{VaR}}^{LGD} x\,f(x)\,\mathrm{d}x$$

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, defthresh=defthresh,
+   low=varisk, up=lgd)$value
> cvar <- cvar/integrate(lossdistr, low=varisk, up=lgd, defthresh=de
> # Plot probability distribution of losses
> curve(expr=lossdistr(x, defthresh=defthresh, 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
```

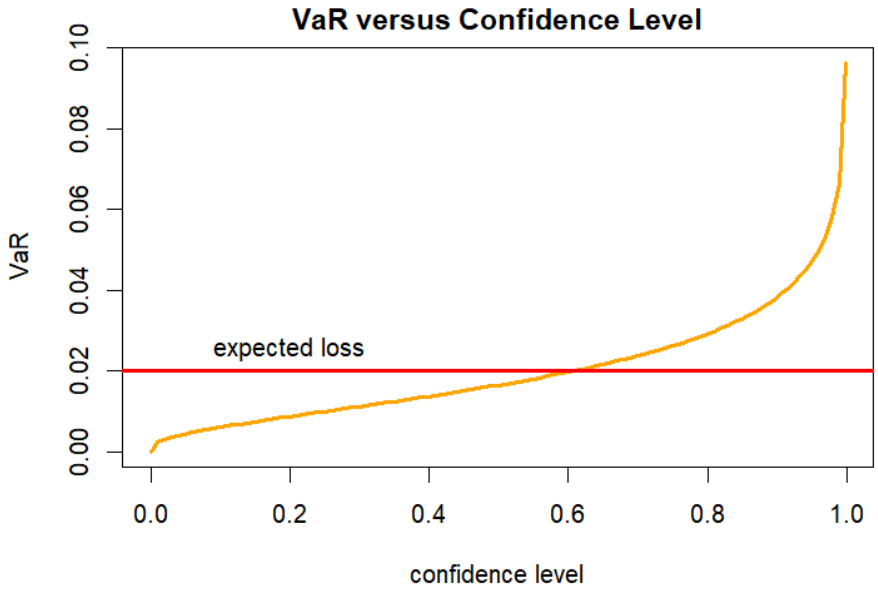
```
> # Add lines for VaR
> abline(v=varisk, col="red", lwd=3)
> text(x=varisk-0.001, y=10, labels="VaR",
+  lwd=2, srt=90, pos=3)
> # Add shading for CVaR
> vars <- seq(varisk, varmax, length=100)
> densv <- sapply(vars, lossdistr,
+   defthresh=defthresh, rho=rho)
> # Draw shaded polygon
> polygon(c(varisk, 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 *quantile* of the loss distribution (the *VaR*), for a given a confidence level α, is given by the inverse of the cumulative loss distribution:

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

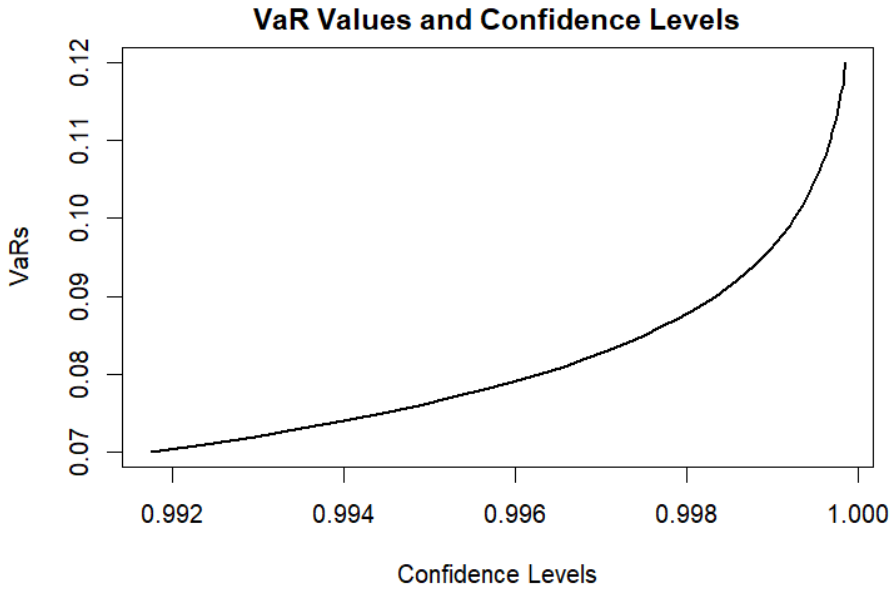



```
> # VaR (quantile of the loss distribution)
> varfun <- function(x, defthresh=qnorm(0.1), rho=0.1, lgd=0.4)
+   lgd*pnorm((sqrt(rho)*qnorm(x) + defthresh)/sqrt(1-rho))
> varfun(x=0.99, defthresh=defthresh, rho=rho, lgd=lgd)
> # Plot VaR
> curve(expr=varfun(x, defthresh=defthresh, rho=rho, lgd=lgd),
+ 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.

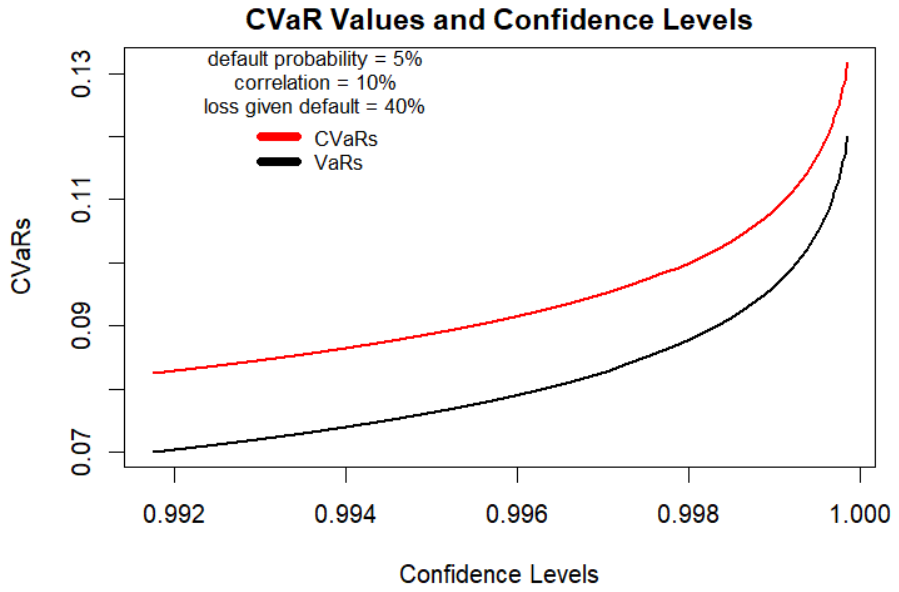
```
> # Integrate lossdistr() over full range
> integrate(lossdistr, low=0.0, up=lgd,
+     defthresh=defthresh, rho=rho, lgd=lgd)
> # Calculate expected losses using lossdistr()
> integrate(function(x) x*lossdistr(x, defthresh=defthresh,
+   rho=rho, lgd=lgd), low=0.0, up=lgd)
> # Calculate confidence levels corresponding to VaR values
> vars <- seq(0.07, 0.12, 0.001)
> confls <- sapply(vars, function(varisk) {
+   integrate(lossdistr, low=varisk, up=lgd,
+       defthresh=defthresh, rho=rho, lgd=lgd)
+ })  # end sapply
> confls <- cbind(as.numeric(t(confls)[, 1]), vars)
> colnames(confls) <- c("levels", "VaRs")
> # Calculate 95% confidence level VaR value
> confls[match(TRUE, confls[, "levels"] < 0.05), "VaRs"]
> plot(x=1-confls[, "levels"],
+      y=confls[, "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, defthresh=defthresh,
+ rho=rho, lgd=lgd), low=varisk, up=lgd)})  # end sapply
> confls <- cbind(confls, as.numeric(t(cvars)[, 1]))
> colnames(confls)[3] <- "CVaRs"
> # Divide CVaR by confidence level
> confls[, "CVaRs"] <- confls[, "CVaRs"]/confls[, "levels"]
> # Calculate 95% confidence level CVaR value
> confls[match(TRUE, confls[, "levels"] < 0.05), "CVaRs"]
> # Plot CVaRs
> plot(x=1-confls[, "levels"], y=confls[, "CVaRs"],
+      t="l", col="red", lwd=2,
+      ylim=range(confls[, c("VaRs", "CVaRs")]),
+      xlab="confidence level", ylab="CVaRs",
+      main="CVaR Values and Confidence Levels")
```




```
> # Add VaRs
> lines(x=1-confls[, "levels"], y=confls[, "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.

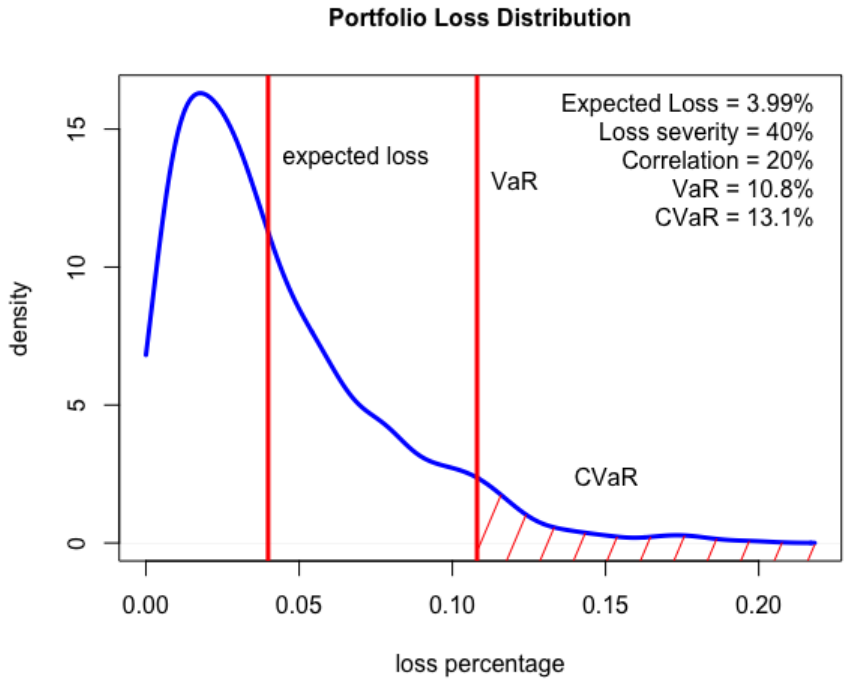
In that case the portfolio losses and *VaR* must be simulated.

```
> # 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)
> defprobs <- runif(nbonds, max=0.2)
> defthresh <- qnorm(defprobs)
> # Simulate losses under the Vasicek model
> sysv <- rnorm(nsimu)
> assetm <- matrix(rnorm(nsimu*nbonds), ncol=nsimu)
> assetm <- t(rhos*sysv + t(rhosm*assetm))
> lossm <- lgd*colSums(assetm < defthresh)/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(lossm, confl)
> # Calculate the CVaR as the mean losses in excess of VaR
> cvar <- mean(lossm[lossm > varisk])
> # Plot the density of portfolio losses
> densv <- density(lossm, 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(defprobs)
> 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)
```

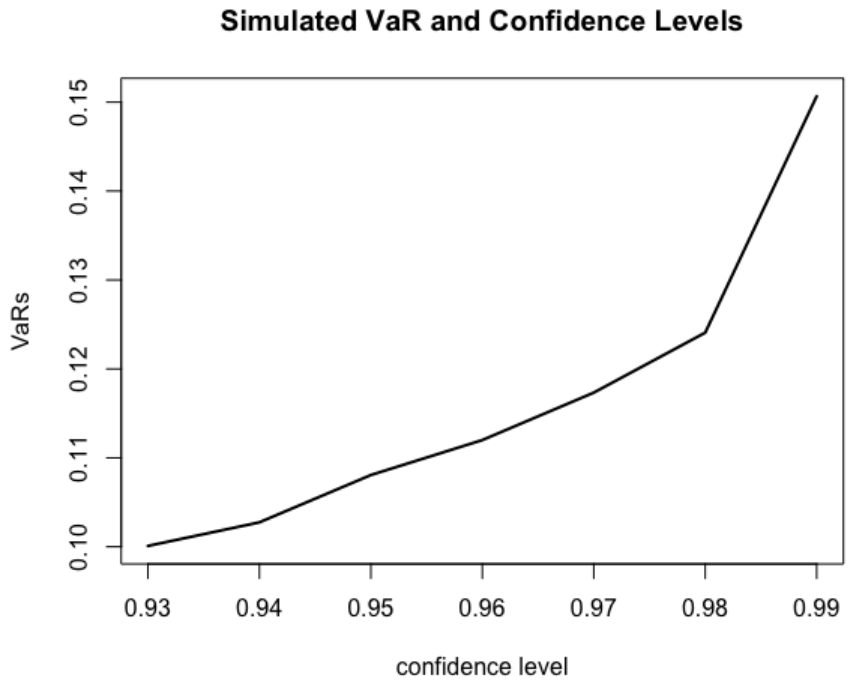



```
> # Draw shaded polygon for CVaR
> intail <- (densv$x > varisk)
> xvar <- c(min(densv$x[intail]), densv$x[intail], max(densv$x))
> polygon(xvar, c(-1, densv$y[intail], -1), col="red", border=NA, de
> # Add text for CVaR
> text(x=5*varisk/4, y=(ymax/7), labels="CVaR", lwd=2, pos=4, cex=1.
> # Add 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`.



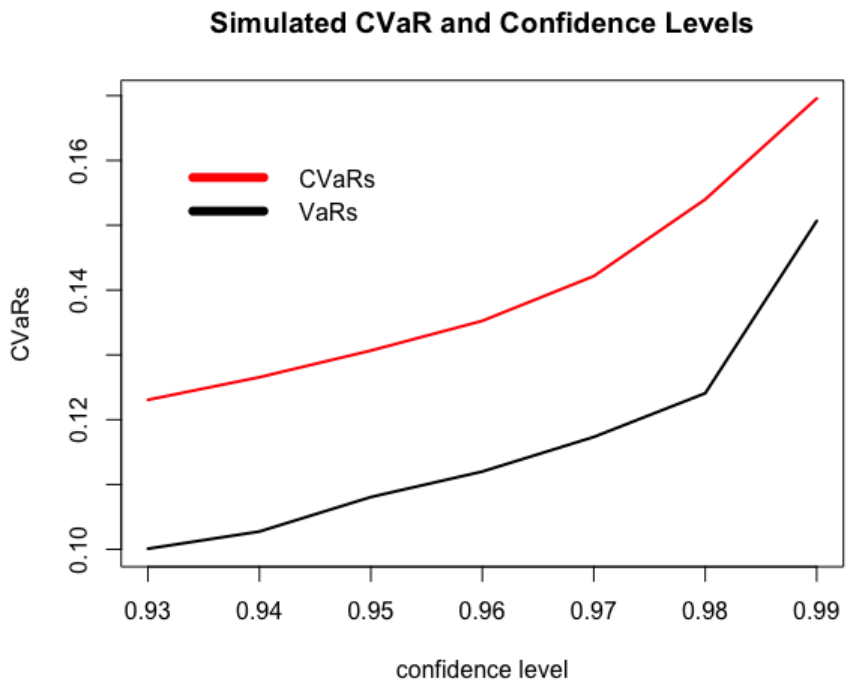

```
> # Calculate VaRs from confidence levels
> confls <- seq(0.93, 0.99, 0.01)
> vars <- quantile(lossm, probs=confls)
> plot(x=confls, y=vars, t="l", lwd=2,
+    xlab="confidence level", ylab="VaRs",
+    main="Simulated VaR and Confidence Levels")
```

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(lossm[lossm >= varisk])
+ })  # end sapply
> cvars <- cbind(cvars, vars)
> # Alternative CVaR calculation using frequency table
> # first calculate frequency table of losses
> # tablev <- table(lossm)/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=confls, 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=confls, 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(defthresh, # Default thresholds
+    lgd=0.6, # loss given default
+    rhos, rhosm, # asset correlation
+    nsimu=1000, # number of simulations
+    confls=seq(0.93, 0.99, 0.01) # Confidence levels
+    ) {
+   # Define model parameters
+   nbonds <- NROW(defthresh)
+   # Simulate losses under the Vasicek model
+   sysv <- rnorm(nsimu)
+   assetm <- matrix(rnorm(nsimu*nbonds), ncol=nsimu)
+   assetm <- t(rhos*sysv + t(rhosm*assetm))
+   lossm <- lgd*colSums(assetm < defthresh)/nbonds
+   # Calculate VaRs and CVaRs
+   vars <- quantile(lossm, probs=confls)
+   cvars <- sapply(vars, function(varisk) {
+     mean(lossm[lossm >= varisk])
+   })  # end sapply
+   names(vars) <- confls
+   names(cvars) <- confls
+   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)
> defprobs <- runif(nbonds, max=0.2)
> defthresh <- qnorm(defprobs)
> confls <- seq(0.93, 0.99, 0.01)
> # Define number of bootstrap simulations
> nboot <- 500
> # Perform bootstrap of calc_var
> set.seed(1121)
> bootd <- sapply(rep(lgd, nboot), calc_var,
+   defthresh=defthresh,
+   rhos=rhos, rhosm=rhosm,
+   nsimu=nsimu, confls=confls)  # 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*.

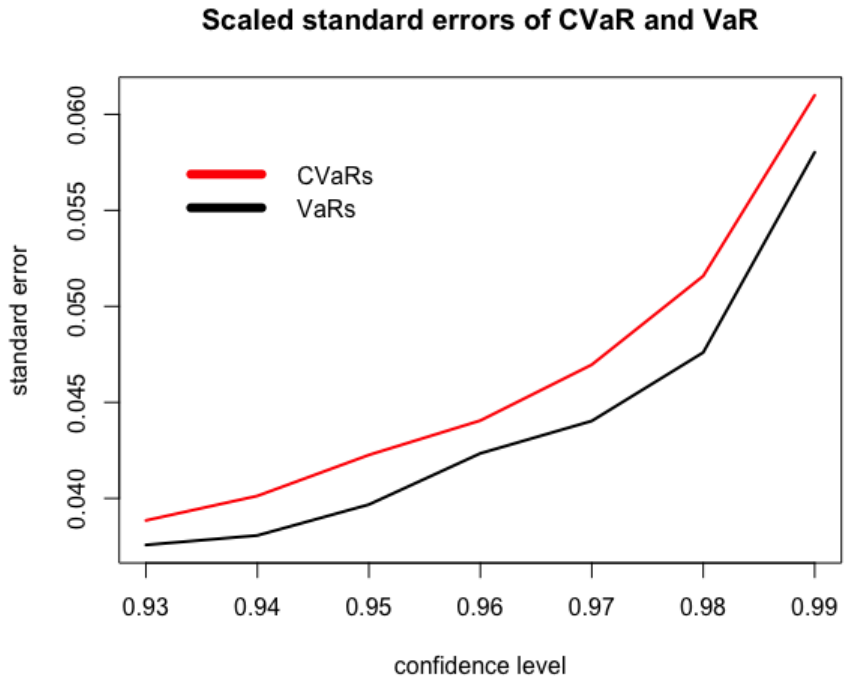
So the greater the number of loss events, the smaller the standard errors, and vice versa.

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*.

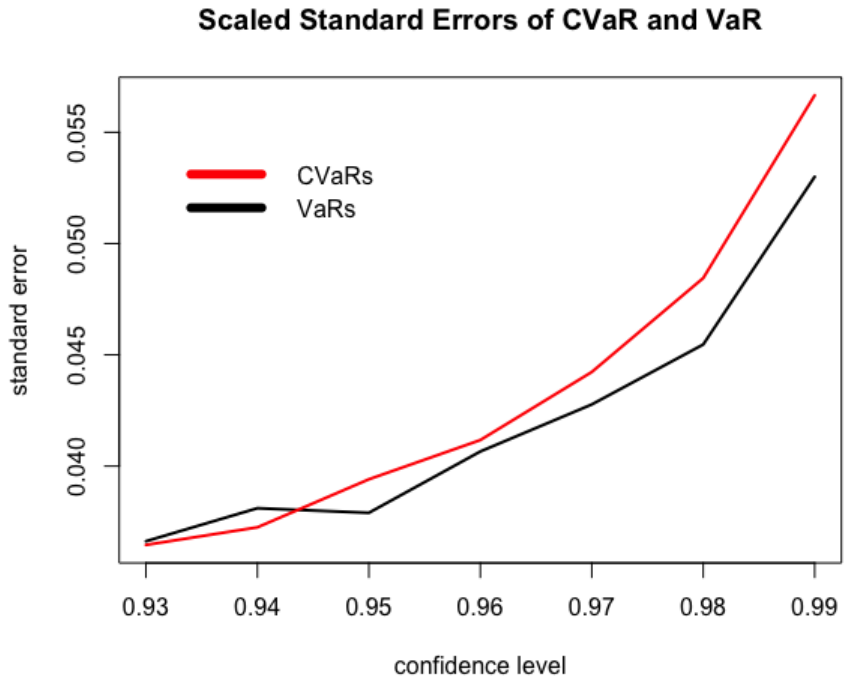



```
> # Plot the scaled standard errors of VaRs and CVaRs
> plot(x=names(varsds), y=varsds,
+   t="l", lwd=2, ylim=range(c(varsds, cvarsds)),
+   xlab="confidence level", ylab="standard error",
+   main="Scaled Standard Errors of CVaR and VaR")
> lines(x=names(cvarsds), y=cvarsds, lwd=2, col="red")
> 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
> cluster <- makeCluster(ncores)  # Initialize compute cluster
> # Perform bootstrap of calc_var for Windows
> clusterSetRNGStream(cluster, 1121)
> bootd <- parLapply(cluster, rep(lgd, nboot),
+   fun=calc_var, defthresh=defthresh,
+   rhos=rhos, rhosm=rhosm,
+   nsimu=nsimu, confls=confls)  # end parLapply
> stopCluster(cluster)  # Stop R processes over cluster
> # Bootstrap under Mac-OSX or Linux
> bootd <- mclapply(rep(lgd, nboot),
+   FUN=calc_var, defthresh=defthresh,
+   rhos=rhos, rhosm=rhosm,
+   nsimu=nsimu, confls=confls)  # 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.

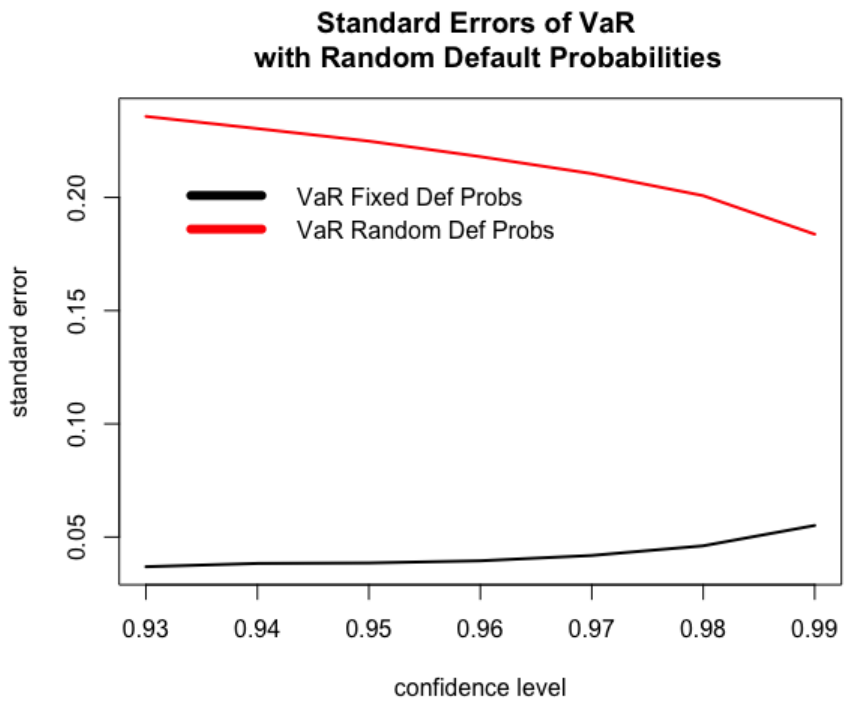
```
> calc_var <- function(defprobs, # Default probabilities
+    lgd=0.6, # loss given default
+    rhos, rhosm, # asset correlation
+    nsimu=1000, # number of simulations
+    confls=seq(0.93, 0.99, 0.01) # Confidence levels
+    ) {
+   # Calculate random default thresholds
+   defthresh <- qnorm(runif(1,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

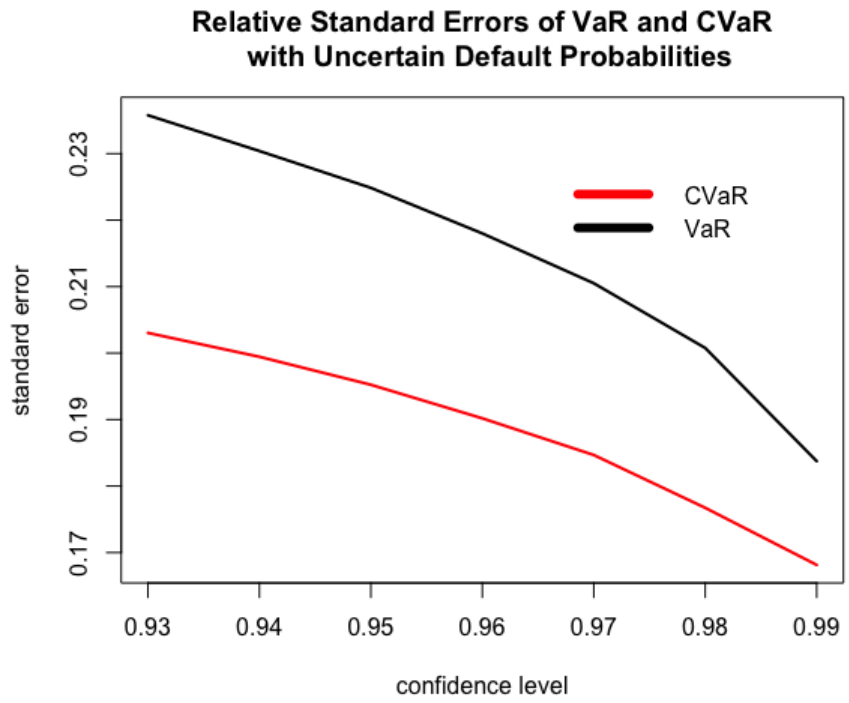
```
> library(parallel)  # load package parallel
> ncores <- detectCores() - 1  # number of cores
> cluster <- makeCluster(ncores)  # Initialize compute cluster
> # Perform bootstrap of calc_var for Windows
> clusterSetRNGStream(cluster, 1121)
> bootd <- parLapply(cluster, rep(lgd, nboot),
+   fun=calc_var, defprobs=defprobs,
+   rhos=rhos, rhosm=rhosm,
+   nsimu=nsimu, confls=confls)  # end parLapply
> stopCluster(cluster)  # Stop R processes over cluster
> # Bootstrap under Mac-OSX or Linux
> bootd <- mclapply(rep(lgd, nboot),
+   FUN=calc_var, defprobs=defprobs,
+   rhos=rhos, rhosm=rhosm,
+   nsimu=nsimu, confls=confls)  # 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.

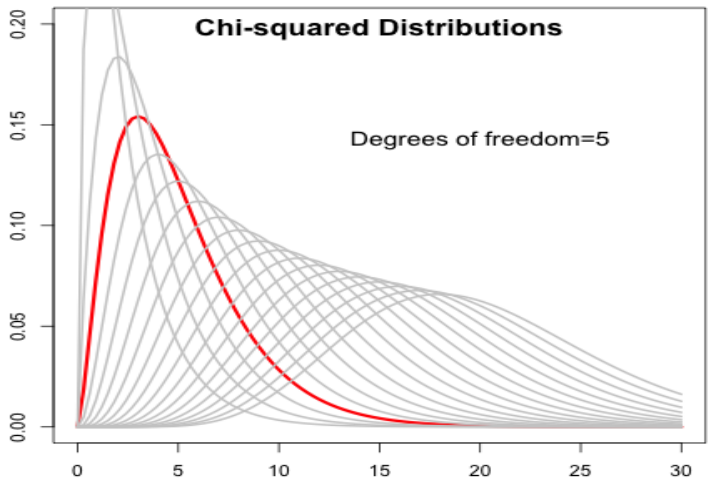
Plotting Using Expression Objects

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)
```

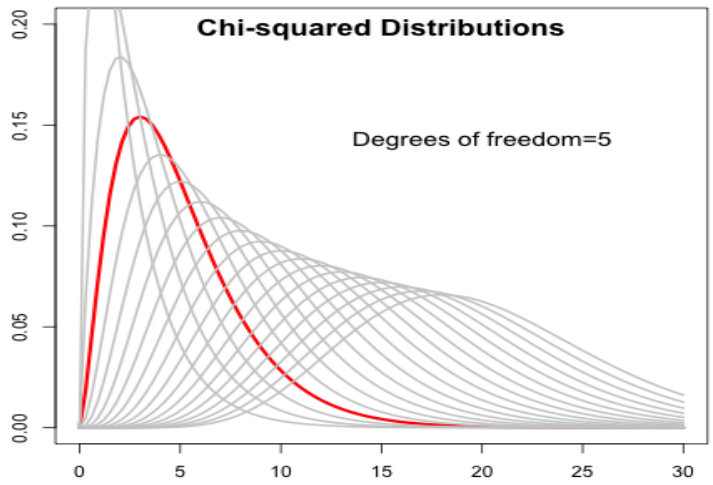
Animated Plots Using Package *animation*

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.

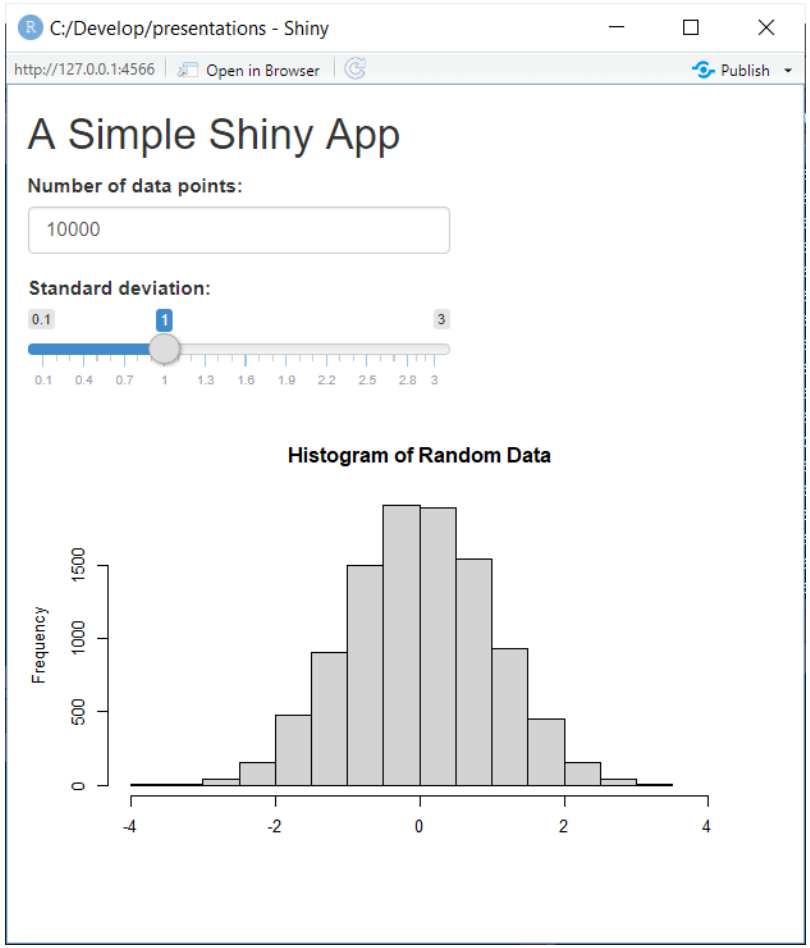
The function `shiny::shinyApp()` creates a `shiny` app from a *shiny interface* and a *shiny server*.

```
> ## 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("look_back", label="Lookback inter
+                           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$symbol, data_env)
+     closep <- log(quantmod::Cl(ohlc))
+     volum <- quantmod::Vo(ohlc)
+     # Return the data
+     cbind(closep, volum)
+   })  # end reactive code
+
+   # Calculate the VWAP indicator in a reactive environment
+   vwapv <- shiny::reactive({
+     # Get model parameters from input argument
+     look_back <- input$look_back
+     # Calculate the VWAP indicator
+     closep <- closep()[, 1]
+     volum <- closep()[, 2]
+     vwapv <- HighFreq::roll_sum(tseries=closep*volum, look_back=lo
+     volumroll <- HighFreq::roll_sum(tseries=volum, look_back=look_
+     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({
+     colnamev <- colnames(vwapv())
+     dygraphs::dygraph(vwapv(), main=paste(colnamev[1], "VWAP")) %>
+ dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+ dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+ dySeries(name=colnamev[1], axis="y", label=colnamev[1], strokeWidt
+ dySeries(name=colnamev[2], axis="y2", label=colnamev[2], strokeWid
+   })  # 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
>
```

Vector and Matrix Calculus

Let $\mathbf{v}$ and $\mathbf{w}$ be vectors, with $\mathbf{v} = \{v_i\}_{i=1}^{i=n}$, and let $\mathbb{1}$ be the unit vector, with $\mathbb{1} = \{1\}_{i=1}^{i=n}$.

Then the inner product of $\mathbf{v}$ and $\mathbf{w}$ can be written as $\mathbf{v}^T\mathbf{w} = \mathbf{w}^T\mathbf{v} = \sum_{i=1}^{n} v_i w_i$.

We can then express the sum of the elements of $\mathbf{v}$ as the inner product: $\mathbf{v}^T\mathbb{1} = \mathbb{1}^T\mathbf{v} = \sum_{i=1}^{n} v_i$.

And the sum of squares of $\mathbf{v}$ as the inner product: $\mathbf{v}^T\mathbf{v} = \sum_{i=1}^{n} v_i^2$.

Let $\mathbb{A}$ be a matrix, with $\mathbb{A} = \{A_{ij}\}_{i,j=1}^{i,j=n}$.

Then the inner product of matrix $\mathbb{A}$ with vectors $\mathbf{v}$ and $\mathbf{w}$ can be written as:

$$\mathbf{v}^T\mathbb{A}\,\mathbf{w} = \mathbf{w}^T\mathbb{A}^T\mathbf{v} = \sum_{i,j=1}^{n} A_{ij} v_i w_j$$

The derivative of a scalar variable with respect to a vector variable is a vector, for example:

$$\frac{d(\mathbf{v}^T\mathbb{1})}{d\mathbf{v}} = d_v[\mathbf{v}^T\mathbb{1}] = d_v[\mathbb{1}^T\mathbf{v}] = \mathbb{1}^T$$

$$d_v[\mathbf{v}^T\mathbf{w}] = d_v[\mathbf{w}^T\mathbf{v}] = \mathbf{w}^T$$

$$d_v[\mathbf{v}^T\mathbb{A}\,\mathbf{w}] = \mathbf{w}^T\mathbb{A}^T$$

$$d_v[\mathbf{v}^T\mathbb{A}\,\mathbf{v}] = \mathbf{v}^T\mathbb{A} + \mathbf{v}^T\mathbb{A}^T$$

Eigenvectors and Eigenvalues of Matrices

The vector w is an *eigenvector* of the matrix $\mathbb{A}$, if it satisfies the *eigenvalue* equation:

$$\mathbb{A}\, w = \lambda\, w$$

Where λ is the *eigenvalue* corresponding to the *eigenvector* w.

The number of *eigenvalues* of a matrix is equal to its dimension.

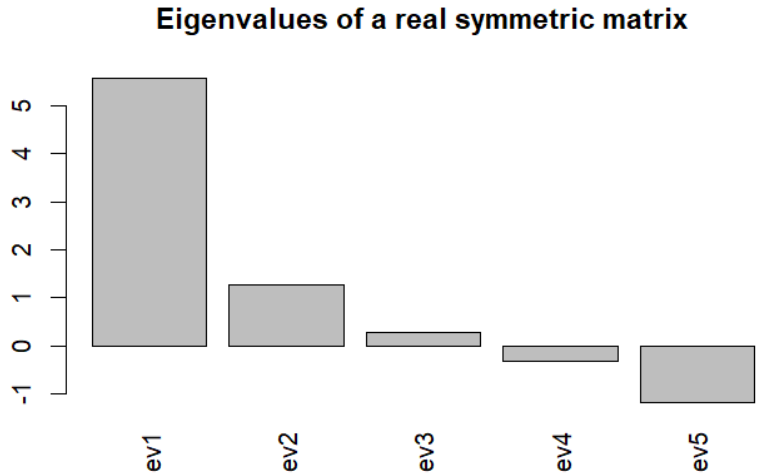
Real symmetric matrices have real *eigenvalues*, and their *eigenvectors* are orthogonal to each other.

The *eigenvectors* can be normalized to 1.

The *eigenvectors* form an *orthonormal basis* in which the matrix $\mathbb{A}$ is diagonal.

The function `eigen()` calculates the *eigenvectors* and *eigenvalues* of numeric matrices.

An excellent interactive visualization of *eigenvectors* and *eigenvalues* is available here:
http://setosa.io/ev/eigenvectors-and-eigenvalues/




```
> # Create a random real symmetric matrix
> matv <- matrix(runif(25), nc=5)
> matv <- matv + t(matv)
> # Calculate the eigenvalues and eigenvectors
> eigend <- eigen(matv)
> eigenvec <- eigend$vectors
> dim(eigenvec)
> # Plot eigenvalues
> barplot(eigend$values, xlab="", ylab="", las=3,
+   names.arg=paste0("ev", 1:NROW(eigend$values)),
+   main="Eigenvalues of a real symmetric matrix")
```

Eigen Decomposition of Matrices

Real symmetric matrices have real *eigenvalues*, and their *eigenvectors* are orthogonal to each other.

The *eigenvectors* form an *orthonormal basis* in which the matrix $\mathbb{A}$ is diagonal:

$$\Sigma = \mathbb{O}^T \mathbb{A} \mathbb{O}$$

Where Σ is a *diagonal* matrix containing the *eigenvalues* of matrix $\mathbb{A}$, and $\mathbb{O}$ is an *orthogonal* matrix of its *eigenvectors*, with $\mathbb{O}^T \mathbb{O} = \mathbb{1}$.

Any real symmetric matrix $\mathbb{A}$ can be decomposed into a product of its *eigenvalues* and its *eigenvectors* (the *eigen decomposition*):

$$\mathbb{A} = \mathbb{O} \Sigma \mathbb{O}^T$$

The *eigen decomposition* expresses a matrix as the product of a rotation, followed by a scaling, followed by the inverse rotation.

```
> # eigenvectors form an orthonormal basis
> round(t(eigenvec) %*% eigenvec, digits=4)
> # Diagonalize matrix using eigenvector matrix
> round(t(eigenvec) %*% (matv %*% eigenvec), digits=4)
> eigend$values
> # eigen decomposition of matrix by rotating the diagonal matrix
> matrixe <- eigenvec %*% (eigend$values * t(eigenvec))
> # Create diagonal matrix of eigenvalues
> # diagmat <- diag(eigend$values)
> # matrixe <- eigenvec %*% (diagmat %*% t(eigenvec))
> all.equal(matv, matrixe)
```

Orthogonal matrices represent rotations in *hyperspace*, and their inverse is equal to their transpose: $\mathbb{O}^{-1} = \mathbb{O}^T$.

The *diagonal* matrix Σ represents a scaling (stretching) transformation proportional to the *eigenvalues*.

The %*% operator performs *inner* (*scalar*) multiplication of vectors and matrices.

Inner multiplication multiplies the rows of one matrix with the columns of another matrix, so that each pair produces a single number.

Positive Definite Matrices

Matrices with positive *eigenvalues* are called *positive definite* matrices.

Matrices with non-negative *eigenvalues* are called *positive semi-definite* matrices (some of their *eigenvalues* may be zero).

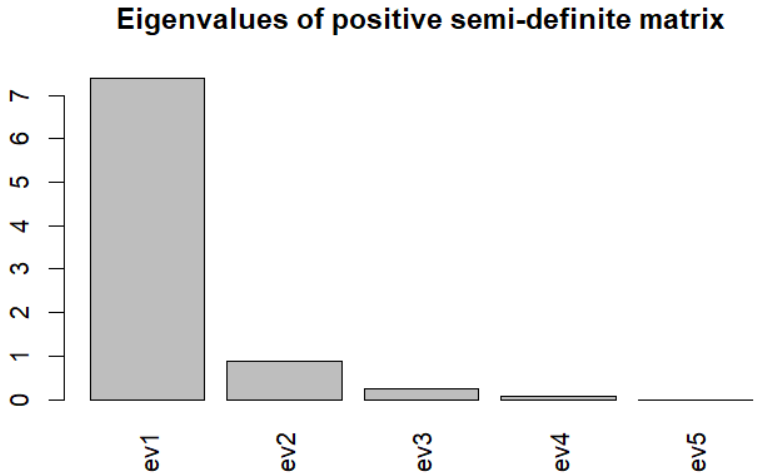
An example of *positive definite* matrices are the covariance matrices of linearly independent variables.

But the covariance matrices of linearly dependent variables have some *eigenvalues* equal to zero, in which case they are *singular*, and only *positive semi-definite*.

All covariance matrices are *positive semi-definite* and all *positive semi-definite* matrices are the covariance matrix of some multivariate distribution.

Matrices which have some *eigenvalues* equal to zero are called *singular* (degenerate) matrices.

For any real matrix $\mathbb{A}$, the matrix $\mathbb{A}^T\mathbb{A}$ is *positive semi-definite*.




```
> # Create a random positive semi-definite matrix
> matv <- matrix(runif(25), nc=5)
> matv <- t(matv) %*% matv
> # Calculate the eigenvalues and eigenvectors
> eigend <- eigen(matv)
> eigend$values
> # Plot eigenvalues
> barplot(eigend$values, las=3, xlab="", ylab="",
+   names.arg=paste0("ev", 1:NROW(eigend$values)),
+   main="Eigenvalues of positive semi-definite matrix")
```

Singular Value Decomposition (*SVD*) of Matrices

The *Singular Value Decomposition* (*SVD*) is a generalization of the *eigen decomposition* of square matrices.

The *SVD* of a rectangular matrix $\mathbb{A}$ is defined as the factorization:

$$\mathbb{A} = \mathbb{U}\,\mathbf{\Sigma}\,\mathbb{V}^T$$

Where $\mathbb{U}$ and $\mathbb{V}$ are the left and right *singular matrices*, and $\mathbf{\Sigma}$ is a diagonal matrix of *singular values*.

If $\mathbb{A}$ has `m` rows and `n` columns and if (`m > n`), then $\mathbb{U}$ is an (`m x n`) *rectangular* matrix, $\mathbf{\Sigma}$ is an (`n x n`) *diagonal* matrix, and $\mathbb{V}$ is an (`n x n`) *orthogonal* matrix, and if (`m < n`) then the dimensions are: (`m x m`), (`m x m`), and (`m x n`).

The left $\mathbb{U}$ and right $\mathbb{V}$ singular matrices consist of columns of *orthonormal* vectors, so that $\mathbb{U}^T\mathbb{U} = \mathbb{V}^T\mathbb{V} = \mathbb{1}$.

In the special case when $\mathbb{A}$ is a square matrix, then $\mathbb{U} = \mathbb{V}$, and the *SVD* reduces to the *eigen decomposition*.

The function `svd()` performs *Singular Value Decomposition* (*SVD*) of a rectangular matrix, and returns a list of three elements: the *singular values*, and the matrices of left-*singular* vectors and the right-*singular* vectors.

```
> # Perform singular value decomposition
> matv <- matrix(rnorm(50), nc=5)
> svdec <- svd(matv)
> # Recompose matv from SVD mat_rices
> all.equal(matv, svdec$u %*% (svdec$d*t(svdec$v)))
> # Columns of U and V are orthonormal
> round(t(svdec$u) %*% svdec$u, 4)
> round(t(svdec$v) %*% svdec$v, 4)
```

The Left and Right Singular Matrices

The left $\mathbb{U}$ and right $\mathbb{V}$ singular matrices define rotation transformations into a coordinate system where the matrix $\mathbb{A}$ becomes diagonal:

$$\mathbf{\Sigma} = \mathbb{U}^T \mathbb{A} \mathbb{V}$$

The columns of $\mathbb{U}$ and $\mathbb{V}$ are called the *singular* vectors, and they are only defined up to a reflection (change in sign), i.e. if vec is a singular vector, then so is -vec.

The left singular matrix $\mathbb{U}$ forms the *eigenvectors* of the matrix $\mathbb{A}\mathbb{A}^T$.

The right singular matrix $\mathbb{V}$ forms the *eigenvectors* of the matrix $\mathbb{A}^T\mathbb{A}$.

```
> # Dimensions of left and right matrices
> nrows <- 6 ; ncols <- 4
> # Calculate left matrix
> leftmat <- matrix(runif(nrows^2), nc=nrows)
> eigend <- eigen(crossprod(leftmat))
> leftmat <- eigend$vectors[, 1:ncols]
> # Calculate right matrix and singular values
> rightmat <- matrix(runif(ncols^2), nc=ncols)
> eigend <- eigen(crossprod(rightmat))
> rightmat <- eigend$vectors
> singval <- sort(runif(ncols, min=1, max=5), decreasing=TRUE)
> # Compose rectangular matrix
> matv <- leftmat %*% (singval * t(rightmat))
> # Perform singular value decomposition
> svdec <- svd(matv)
> # Recompose matv from SVD
> all.equal(matv, svdec$u %*% (svdec$d*t(svdec$v)))
> # Compare SVD with matv components
> all.equal(abs(svdec$u), abs(leftmat))
> all.equal(abs(svdec$v), abs(rightmat))
> all.equal(svdec$d, singval)
> # Eigen decomposition of matv squared
> retsq <- matv %*% t(matv)
> eigend <- eigen(retsq)
> all.equal(eigend$values[1:ncols], singval^2)
> all.equal(abs(eigend$vectors[, 1:ncols]), abs(leftmat))
> # Eigen decomposition of matv squared
> retsq <- t(matv) %*% matv
> eigend <- eigen(retsq)
> all.equal(eigend$values, singval^2)
> all.equal(abs(eigend$vectors), abs(rightmat))
```

Inverse of Symmetric Square Matrices

The inverse of a square matrix $\mathbb{A}$ is defined as a square matrix $\mathbb{A}^{-1}$ that satisfies the equation:

$$\mathbb{A}^{-1}\mathbb{A} = \mathbb{A}\mathbb{A}^{-1} = \mathbb{1}$$

Where $\mathbb{1}$ is the identity matrix.

The inverse $\mathbb{A}^{-1}$ of a *symmetric* square matrix $\mathbb{A}$ can also be expressed as the product of the inverse of its *eigenvalues* ($\mathbf{\Sigma}$) and its *eigenvectors* ($\mathbb{O}$):

$$\mathbb{A}^{-1} = \mathbb{O}\,\mathbf{\Sigma}^{-1}\,\mathbb{O}^{T}$$

But *singular* (degenerate) matrices (which have some *eigenvalues* equal to zero) don't have an inverse.

The inverse of *non-symmetric* matrices can be calculated using *Singular Value Decomposition* (*SVD*).

The function `solve()` solves systems of linear equations, and also inverts square matrices.

```
> # Create a random positive semi-definite matrix
> matv <- matrix(runif(25), nc=5)
> matv <- t(matv) %*% matv
> # Calculate the inverse of matv
> invmat <- solve(a=matv)
> # Multiply inverse with matrix
> round(invmat %*% matv, 4)
> round(matv %*% invmat, 4)
> # Calculate the eigenvalues and eigenvectors
> eigend <- eigen(matv)
> eigenvec <- eigend$vectors
> # Calculate inverse from eigen decomposition
> inveigen <- eigenvec %*% (t(eigenvec) / eigend$values)
> all.equal(invmat, inveigen)
> # Decompose diagonal matrix with inverse of eigenvalues
> # diagmat <- diag(1/eigend$values)
> # inveigen <- eigenvec %*% (diagmat %*% t(eigenvec))
```

Generalized Inverse of Rectangular Matrices

The generalized inverse of an (`m x n`) rectangular matrix $\mathbb{A}$ is defined as an (`n x m`) matrix $\mathbb{A}^{-1}$ that satisfies the equation:

$$\mathbb{A}\mathbb{A}^{-1}\mathbb{A} = \mathbb{A}$$

The generalized inverse matrix $\mathbb{A}^{-1}$ can be expressed as a product of the inverse of its *singular values* ($\mathbb{\Sigma}$) and its left and right *singular* matrices ($\mathbb{U}$ and $\mathbb{V}$):

$$\mathbb{A}^{-1} = \mathbb{V}\,\mathbb{\Sigma}^{-1}\,\mathbb{U}^T$$

The generalized inverse $\mathbb{A}^{-1}$ can also be expressed as the *Moore-Penrose pseudo-inverse*:

$$\mathbb{A}^{-1} = (\mathbb{A}^T\mathbb{A})^{-1}\mathbb{A}^T$$

In the case when the inverse matrix $\mathbb{A}^{-1}$ exists, then the *pseudo-inverse* matrix simplifies to the inverse: $(\mathbb{A}^T\mathbb{A})^{-1}\mathbb{A}^T = \mathbb{A}^{-1}(\mathbb{A}^T)^{-1}\mathbb{A}^T = \mathbb{A}^{-1}$

The function `MASS::ginv()` calculates the generalized inverse of a matrix.

```
> # Random rectangular matrix: nrows > ncols
> nrows <- 6 ; ncols <- 4
> matv <- matrix(runif(nrows*ncols), nc=ncols)
> # Calculate generalized inverse of matv
> invmat <- MASS::ginv(matv)
> round(invmat %*% matv, 4)
> all.equal(matv, matv %*% invmat %*% matv)
> # Random rectangular matrix: nrows < ncols
> nrows <- 4 ; ncols <- 6
> matv <- matrix(runif(nrows*ncols), nc=ncols)
> # Calculate generalized inverse of matv
> invmat <- MASS::ginv(matv)
> all.equal(matv, matv %*% invmat %*% matv)
> round(matv %*% invmat, 4)
> round(invmat %*% matv, 4)
> # Perform singular value decomposition
> svdec <- svd(matv)
> # Calculate generalized inverse from SVD
> invsvd <- svdec$v %*% (t(svdec$u) / svdec$d)
> all.equal(invsvd, invmat)
> # Calculate Moore-Penrose pseudo-inverse
> invmp <- MASS::ginv(t(matv) %*% matv) %*% t(matv)
> all.equal(invmp, invmat)
```

Regularized Inverse of Singular Matrices

Singular matrices have some *singular values* equal to zero, so they don't have an inverse matrix which satisfies the equation: $\mathbb{A}\mathbb{A}^{-1}\mathbb{A} = \mathbb{A}$

But if the *singular values* that are equal to zero are removed, then a *regularized inverse* for *singular* matrices can be specified by:

$$\mathbb{A}^{-1} = \mathbb{V}_n \, \mathbf{\Sigma}_n^{-1} \, \mathbb{U}_n^T$$

Where $\mathbb{U}_n$, $\mathbb{V}_n$ and $\mathbf{\Sigma}_n$ are the *SVD* matrices with the rows and columns corresponding to zero *singular values* removed.

```
> # Create a random singular matrix
> # More columns than rows: ncols > nrows
> nrows <- 4 ; ncols <- 6
> matv <- matrix(runif(nrows*ncols), nc=ncols)
> matv <- t(matv) %*% matv
> # Perform singular value decomposition
> svdec <- svd(matv)
> # Incorrect inverse from SVD because of zero singular values
> invsvd <- svdec$v %*% (t(svdec$u) / svdec$d)
> # Inverse property doesn't hold
> all.equal(matv, matv %*% invsvd %*% matv)
```

```
> # Set tolerance for determining zero singular values
> precv <- sqrt(.Machine$double.eps)
> # Check for zero singular values
> round(svdec$d, 12)
> notzero <- (svdec$d > (precv*svdec$d[1]))
> # Calculate regularized inverse from SVD
> invsvd <- svdec$v[, notzero] %*%
+   (t(svdec$u[, notzero]) / svdec$d[notzero])
> # Verify inverse property of matv
> all.equal(matv, matv %*% invsvd %*% matv)
> # Calculate regularized inverse using MASS::ginv()
> invmat <- MASS::ginv(matv)
> all.equal(invsvd, invmat)
> # Calculate Moore-Penrose pseudo-inverse
> invmp <- MASS::ginv(t(matv) %*% matv) %*% t(matv)
> all.equal(invmp, invmat)
```

Diagonalizing the Inverse of Singular Matrices

The left-*singular* matrix $\mathbb{U}$ combined with the right-*singular* matrix $\mathbb{V}$ define a rotation transformation into a coordinate system where the matrix $\mathbb{A}$ becomes diagonal:

$$\mathbf{\Sigma} = \mathbb{U}^T \mathbb{A} \mathbb{V}$$

The generalized inverse of *singular* matrices doesn't satisfy the equation: $\mathbb{A}^{-1}\mathbb{A} = \mathbb{A}\mathbb{A}^{-1} = \mathbb{1}$, but if it's rotated into the same coordinate system where $\mathbb{A}$ is diagonal, then we have:

$$\mathbb{U}^T (\mathbb{A}^{-1}\mathbb{A}) \mathbb{V} = \mathbb{1}_n$$

So that $\mathbb{A}^{-1}\mathbb{A}$ is diagonal in the same coordinate system where $\mathbb{A}$ is diagonal.

```
> # Diagonalize the unit matrix
> unitmat <- matv %*% invmat
> round(unitmat, 4)
> round(matv %*% invmat, 4)
> round(t(svdec$u) %*% unitmat %*% svdec$v, 4)
```

Solving Linear Equations Using solve()

A system of linear equations can be defined as:

$$\mathbb{A}\,x = b$$

Where $\mathbb{A}$ is a matrix, b is a vector, and x is the unknown vector.

The solution of the system of linear equations is equal to:

$$x = \mathbb{A}^{-1} b$$

Where $\mathbb{A}^{-1}$ is the *inverse* of the matrix $\mathbb{A}$.

The function solve() solves systems of linear equations, and also inverts square matrices.

The %*% operator performs *inner* (*scalar*) multiplication of vectors and matrices.

Inner multiplication multiplies the rows of one matrix with the columns of another matrix, so that each pair produces a single number:

```
> # Define a square matrix
> matv <- matrix(c(1, 2, -1, 2), nc=2)
> vecv <- c(2, 1)
> # Calculate the inverse of matv
> invmat <- solve(a=matv)
> invmat %*% matv
> # Calculate solution using inverse of matv
> solutionv <- invmat %*% vecv
> matv %*% solutionv
> # Calculate solution of linear system
> solutionv <- solve(a=matv, b=vecv)
> matv %*% solutionv
```

Fast Matrix Inverse Using C++

The *Armadillo* C++ functions can be several times faster than R functions - even those that are compiled from C++ code.

That's because the *Armadillo* C++ library calls routines optimized for fast numerical calculations.

The package *RcppArmadillo* allows calling from R the high-level *Armadillo* C++ linear algebra library.

The C++ *Armadillo* function `arma::inv()` calculates the matrix inverse several times faster than the function `solve()`.

The function `solve()` calculates the matrix inverse several times faster than the function `MASS::ginv()`.

```
> # Create a random matrix
> matv <- matrix(rnorm(100), nc=10)
> # Calculate the matrix inverse using solve()
> invmatr <- solve(a=matv)
> round(invmatr %*% matv, 4)
> # Compile the C++ file using Rcpp
> Rcpp::sourceCpp(file="/Users/jerzy/Develop/Rcpp/test_fun.cpp")
> # Calculate the matrix inverse using C++
> invmat <- calc_invmat(matv)
> all.equal(invmat, invmatr)
> # Compare the speed of RcppArmadillo with R code
> library(microbenchmark)
> summary(microbenchmark(
+   ginv=MASS::ginv(matv),
+   solve=solve(matv),
+   cpp=calc_invmat(matv),
+   times=10))[, c(1, 4, 5)]
```

```
// Rcpp header with information for C++ compiler
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h> // include RcppArmadillo header file
using namespace arma; // use Armadillo C++ namespace

// [[Rcpp::export]]
arma::mat calc_invmat(arma::mat& matv) {

  return arma::inv(matv);

}  // end calc_invmat
```

Cholesky Decomposition

The *Cholesky* decomposition of a *positive definite* matrix $\mathbb{A}$ is defined as:

$$\mathbb{A} = \mathbb{L}^T \mathbb{L}$$

Where $\mathbb{L}$ is an upper triangular matrix with positive diagonal elements.

The matrix $\mathbb{L}$ can be considered the square root of $\mathbb{A}$.

The vast majority of random *positive semi-definite* matrices are also *positive definite*.

The function `chol()` calculates the *Cholesky* decomposition of a *positive definite* matrix.

The functions `chol2inv()` and `chol()` calculate the inverse of a *positive definite* matrix two times faster than `solve()`.

```
> # Create large random positive semi-definite matrix
> matv <- matrix(runif(1e4), nc=100)
> matv <- t(matv) %*% matv
> # Calculate the eigen decomposition
> eigend <- eigen(matv)
> eigenval <- eigend$values
> eigenvec <- eigend$vectors
> # Set tolerance for determining zero singular values
> precv <- sqrt(.Machine$double.eps)
> # If needed convert to positive definite matrix
> notzero <- (eigenval > (precv*eigenval[1]))
> if (sum(!notzero) > 0) {
+   eigenval[!notzero] <- 2*precv
+   matv <- eigenvec %*% (eigenval * t(eigenvec))
+ }  # end if
> # Calculate the Cholesky matv
> cholmat <- chol(matv)
> cholmat[1:5, 1:5]
> all.equal(matv, t(cholmat) %*% cholmat)
> # Calculate inverse from Cholesky
> invchol <- chol2inv(cholmat)
> all.equal(solve(matv), invchol)
> # Compare speed of Cholesky inversion
> library(microbenchmark)
> summary(microbenchmark(
+   solve=solve(matv),
+   cholmat=chol2inv(chol(matv)),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Simulating Correlated Returns Using Cholesky Matrix

The *Cholesky* decomposition of a covariance matrix can be used to simulate correlated *Normal* returns following the given covariance matrix: $\mathbb{C} = \mathbb{L}^T\mathbb{L}$

Let $\mathbb{R}$ be a matrix with columns of *uncorrelated* returns following the *Standard Normal* distribution.

The *correlated* returns $\mathbb{R}_c$ can be calculated from the *uncorrelated* returns $\mathbb{R}$ by multiplying them by the *Cholesky* matrix $\mathbb{L}$:

$$\mathbb{R}_c = \mathbb{L}^T\mathbb{R}$$

```
> # Calculate random covariance matrix
> covmat <- matrix(runif(25), nc=5)
> covmat <- t(covmat) %*% covmat
> # Calculate the Cholesky matrix
> cholmat <- chol(covmat)
> cholmat
> # Simulate random uncorrelated returns
> nassets <- 5
> nrows <- 10000
> retp <- matrix(rnorm(nassets*nrows), nc=nassets)
> # Calculate correlated returns by applying Cholesky
> retscorr <- retp %*% cholmat
> # Calculate covariance matrix
> covmat2 <- crossprod(retscorr) /(nrows-1)
> all.equal(covmat, covmat2)
```

Eigenvalues of Singular Covariance Matrices

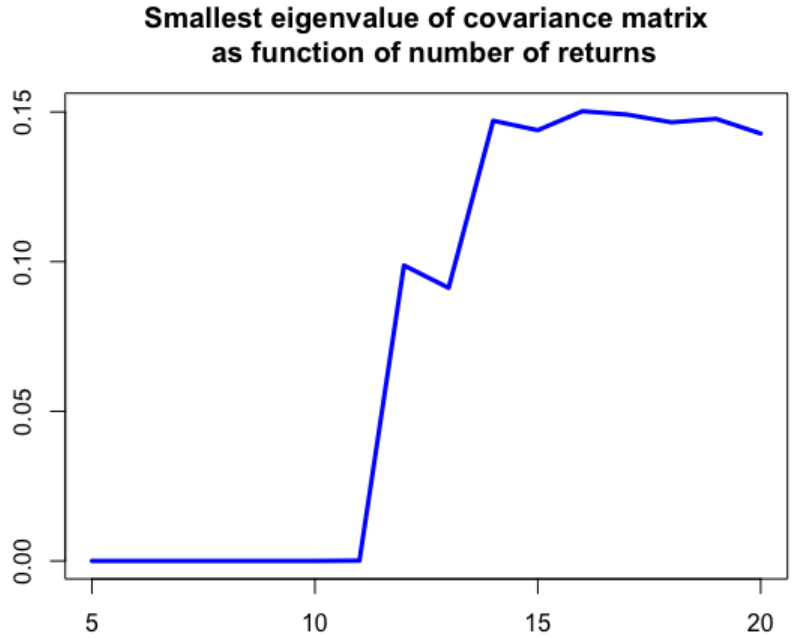
If $\mathbb{R}$ is a matrix of returns (with zero mean) for a portfolio of `k` stocks (columns), over `n` time periods (rows), then the sample covariance matrix is equal to:

$$\mathbb{C} = \mathbb{R}^T \mathbb{R}/(n-1)$$

If the number of rows is less than the number of stocks, then the returns are *collinear*, and the sample covariance matrix is *singular*, with some *eigenvalues* equal to zero.

The function `crossprod()` performs *inner* (*scalar*) multiplication, exactly the same as the %*% operator, but it is slightly faster.

```
> # Simulate random stock returns
> nassets <- 10
> nrows <- 100
> set.seed(1121)  # Initialize random number generator
> retp <- matrix(rnorm(nassets*nrows), nc=nassets)
> # Calculate centered (de-meaned) returns matrix
> retp <- t(t(retp) - colMeans(retp))
> # Or
> retp <- apply(retp, MARGIN=2, function(x) (x-mean(x)))
> # Calculate covariance matrix
> covmat <- crossprod(retp) /(nrows-1)
> # Calculate the eigenvalues and eigenvectors
> eigend <- eigen(covmat)
> eigend$values
> barplot(eigend$values, # Plot eigenvalues
+   xlab="", ylab="", las=3,
+   names.arg=paste0("ev", 1:NROW(eigend$values)),
+   main="Eigenvalues of Covariance Matrix")
```




```
> # Calculate the eigenvalues and eigenvectors
> # as function of number of returns
> ndata <- ((nassets/2):(2*nassets))
> eigenval <- sapply(ndata, function(x) {
+   retp <- retp[1:x, ]
+   retp <- apply(retp, MARGIN=2, function(y) (y - mean(y)))
+   covmat <- crossprod(retp) / (x-1)
+   min(eigen(covmat)$values)
+ })  # end sapply
> plot(y=eigenval, x=ndata, t="l", xlab="", ylab="", lwd=3, col="blu
+   main="Smallest eigenvalue of covariance matrix
+   as function of number of returns")
```

Regularized Inverse of Singular Covariance Matrices

The *regularization* technique allows calculating the inverse of *singular* covariance matrices while reducing the effects of statistical noise.

If the number of time periods of returns is less than the number of assets (columns), then the covariance matrix of returns is *singular*, and some of its *eigenvalues* are zero, so it doesn't have an inverse.

The *regularized* inverse $\mathbb{C}_n^{-1}$ is calculated by removing the higher order eigenvalues that are almost zero, and keeping only the first *n eigenvalues*:

$$\mathbb{C}_n^{-1} = \mathbb{O}_n \, \mathbf{\Sigma}_n^{-1} \, \mathbb{O}_n^T$$

Where $\mathbf{\Sigma}_n$ and $\mathbb{O}_n$ are matrices with the higher order eigenvalues and eigenvectors removed.

The function `MASS::ginv()` calculates the *regularized* inverse of a matrix.

```
> # Create rectangular matrix with collinear columns
> matv <- matrix(rnorm(10*8), nc=10)
> # Calculate covariance matrix
> covmat <- cov(matv)
> # Calculate inverse of covmat - error
> invmat <- solve(covmat)
> # Calculate regularized inverse of covmat
> invmat <- MASS::ginv(covmat)
> # Verify inverse property of matv
> all.equal(covmat, covmat %*% invmat %*% covmat)
> # Perform eigen decomposition
> eigend <- eigen(covmat)
> eigenvec <- eigend$vectors
> eigenval <- eigend$values
> # Set tolerance for determining zero singular values
> precv <- sqrt(.Machine$double.eps)
> # Calculate regularized inverse matrix
> notzero <- (eigenval > (precv * eigenval[1]))
> invreg <- eigenvec[, notzero] %*%
+   (t(eigenvec[, notzero]) / eigenval[notzero])
> # Verify that invmat is same as invreg
> all.equal(invmat, invreg)
```

The Bias-Variance Tradeoff of the Regularized Inverse

Removing the very small higher order eigenvalues can also be used to reduce the propagation of statistical noise and improve the signal-to-noise ratio.

Removing a larger number of eigenvalues further reduces the noise, but it increases the bias of the covariance matrix.

This is an example of the *bias-variance tradeoff*.

Even though the *regularized* inverse $\mathbb{C}_n^{-1}$ does not satisfy the matrix inverse property, its out-of-sample forecasts may be more accurate than those using the actual inverse matrix.

The parameter `dimax` specifies the number of eigenvalues used for calculating the *regularized* inverse of the covariance matrix of returns.

The optimal value of the parameter `dimax` can be determined using *backtesting* (*cross-validation*).

```
> # Calculate regularized inverse matrix using cutoff
> dimax <- 3
> invmat <- eigenvec[, 1:dimax] %*%
+   (t(eigenvec[, 1:dimax]) / eigend$values[1:dimax])
> # Verify that invmat is same as invreg
> all.equal(invmat, invreg)
```

Shrinkage Estimator of Covariance Matrices

The estimates of the covariance matrix suffer from statistical noise, and those noise are magnified when the covariance matrix is inverted.

In the *shrinkage* technique the covariance matrix $\mathbb{C}_s$ is estimated as a weighted sum of the sample covariance estimator $\mathbb{C}$ plus a target matrix $\mathbb{T}$:

$$\mathbb{C}_s = (1 - \alpha)\,\mathbb{C} + \alpha\,\mathbb{T}$$

The target matrix $\mathbb{T}$ represents an estimate of the covariance matrix subject to some constraint, such as that all the correlations are equal to each other.

The shrinkage intensity α determines the amount of shrinkage that is applied, with $\alpha = 1$ representing a complete shrinkage towards the target matrix.

The *shrinkage* estimator reduces the estimate variance at the expense of increasing its bias (known as the *bias-variance tradeoff*).

```
> # Create a random covariance matrix
> set.seed(1121)
> matv <- matrix(rnorm(5e2), nc=5)
> covmat <- cov(matv)
> cormat <- cor(matv)
> stdev <- sqrt(diag(covmat))
> # Calculate target matrix
> cormean <- mean(cormat[upper.tri(cormat)])
> targetmat <- matrix(cormean, nr=NROW(covmat), nc=NCOL(covmat))
> diag(targetmat) <- 1
> targetmat <- t(t(targetmat * stdev) * stdev)
> # Calculate shrinkage covariance matrix
> alpha <- 0.5
> covshrink <- (1-alpha)*covmat + alpha*targetmat
> # Calculate inverse matrix
> invmat <- solve(covshrink)
```

Recursive Matrix Inverse

The inverse of a square matrix $\mathbb{A}$ can be calculated approximately using the recursive *Schulz formula*:

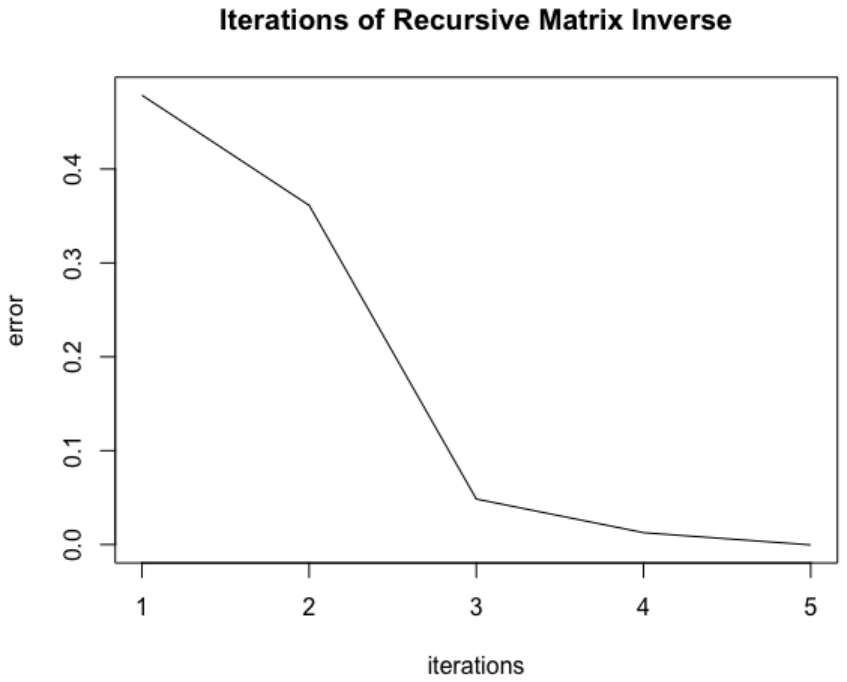
$$\mathbb{A}_{i+1}^{-1} \leftarrow 2\mathbb{A}_i^{-1} - \mathbb{A}_i^{-1}\mathbb{A}\mathbb{A}_i^{-1}$$

The *Schulz formula* requires a good initial value for the inverse matrix $\mathbb{A}_1^{-1}$ or else the recursion diverges.

If the initial inverse matrix $\mathbb{A}_1^{-1}$ is very close to the actual inverse $\mathbb{A}^{-1}$, then the *Schulz formula* produces a very good approximation with just a few iterations.

The *Schulz formula* is useful for updating the inverse when the matrix $\mathbb{A}$ changes only slightly. For example, for updating the inverse of the covariance matrix as it changes slowly over time.

The super-assignment operator "<<-" modifies variables in the *enclosing* environment in which the function was *defined* (*lexical* scoping).




```
> # Create a random matrix
> matv <- matrix(rnorm(100), nc=10)
> # Calculate the inverse of matv
> invmat <- solve(a=matv)
> # Multiply inverse with matrix
> round(invmat %*% matv, 4)
> # Calculate the initial inverse
> invmatr <- invmat + matrix(rnorm(100, sd=0.1), nc=10)
> # Calculate the approximate recursive inverse of matv
> invmatr <- (2*invmatr - invmatr %*% matv %*% invmatr)
> # Calculate the sum of the off-diagonal elements
> sum((invmatr %*% matv)[upper.tri(matv)])
```

```
> # Calculate the recursive inverse of matv in a loop
> invmatr <- invmat + matrix(rnorm(100, sd=0.1), nc=10)
> iterv <- sapply(1:5, function(x) {
+ # Calculate the recursive inverse of matv
+   invmatr <<- (2*invmatr - invmatr %*% matv %*% invmatr)
+ # Calculate the sum of the off-diagonal elements
+   sum((invmatr %*% matv)[upper.tri(matv)])
+ })  # end sapply
> # Plot the iterations
> plot(x=1:5, y=iterv, t="l", xlab="iterations", ylab="error",
+      main="Iterations of Recursive Matrix Inverse")
```

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