FRE6871 R in Finance
Lecture#4, Fall 2022

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

September 26, 2022

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:

$$\begin{aligned} \frac{d(\mathbf{v}^T\mathbb{1})}{d\mathbf{v}} = d_v[\mathbf{v}^T\mathbb{1}] = d_v[\mathbb{1}^T\mathbf{v}] &= \mathbb{1}^T \\ d_v[\mathbf{v}^T\mathbf{w}] = d_v[\mathbf{w}^T\mathbf{v}] &= \mathbf{w}^T \\ d_v[\mathbf{v}^T\mathbb{A}\,\mathbf{w}] &= \mathbf{w}^T\mathbb{A}^T \\ d_v[\mathbf{v}^T\mathbb{A}\,\mathbf{v}] &= \mathbf{v}^T\mathbb{A} + \mathbf{v}^T\mathbb{A}^T \end{aligned}$$

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.

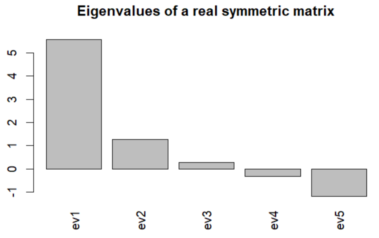
Real symmetric matrices have real *eigenvalues*, and their *eigenvectors* are orthogonal to each other.

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 random real symmetric matrix
> matrixv <- matrix(runif(25), nc=5)
> matrixv <- matrixv + t(matrixv)
> # Calculate eigenvectors and eigenvalues
> eigend <- eigen(matrixv)
> 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:

$$\mathbb{D} = \mathbb{O}^T \mathbb{A} \, \mathbb{O}$$

Where $\mathbb{D}$ 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*):

$$\mathbb{A} = \mathbb{O} \, \mathbb{D} \, \mathbb{O}^T$$

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) %*% (matrixv %*% 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(matrixv, matrixe)
```

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

The *diagonal* matrix $\mathbb{D}$ 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).

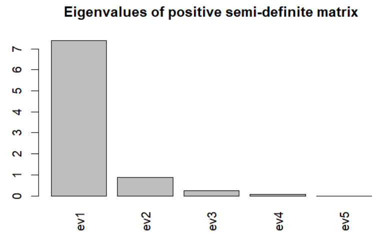
An example of *positive definite* matrices are the covariance matrices of linearly independent variables.

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 random positive semi-definite matrix
> matrixv <- matrix(runif(25), nc=5)
> matrixv <- t(matrixv) %*% matrixv
> # Calculate eigenvectors and eigenvalues
> eigend <- eigen(matrixv)
> 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:

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

Where $\mathbb{U}$ and $\mathbb{V}$ are the left and right *singular matrices*, and Σ 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, Σ 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
> matrixv <- matrix(rnorm(50), nc=5)
> svdec <- svd(matrixv)
> # Recompose matrixv from SVD mat_rices
> all.equal(matrixv, 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:

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

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
> nleft <- 6 ; nright <- 4
> # Calculate left matrix
> leftmat <- matrix(runif(nleft^2), nc=nleft)
> eigend <- eigen(crossprod(leftmat))
> leftmat <- eigend$vectors[, 1:nright]
> # Calculate right matrix and singular values
> rightmat <- matrix(runif(nright^2), nc=nright)
> eigend <- eigen(crossprod(rightmat))
> rightmat <- eigend$vectors
> sing_values <- sort(runif(nright, min=1, max=5), decreasing=TRUE)
> # Compose rectangular matrix
> matrixv <- leftmat %*% (sing_values * t(rightmat))
> # Perform singular value decomposition
> svdec <- svd(matrixv)
> # Recompose matrixv from SVD
> all.equal(matrixv, svdec$u %*% (svdec$d*t(svdec$v)))
> # Compare SVD with matrixv components
> all.equal(abs(svdec$u), abs(leftmat))
> all.equal(abs(svdec$v), abs(rightmat))
> all.equal(svdec$d, sing_values)
> # Eigen decomposition of matrixv squared
> retsq <- matrixv %*% t(matrixv)
> eigend <- eigen(retsq)
> all.equal(eigend$values[1:nright], sing_values^2)
> all.equal(abs(eigend$vectors[, 1:nright]), abs(leftmat))
> # Eigen decomposition of matrixv squared
> retsq <- t(matrixv) %*% matrixv
> eigend <- eigen(retsq)
> all.equal(eigend$values, sing_values^2)
> all.equal(abs(eigend$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* ($\mathbb{D}$) and its *eigenvectors* ($\mathbb{O}$):

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

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 random positive semi-definite matrix
> matrixv <- matrix(runif(25), nc=5)
> matrixv <- t(matrixv) %*% matrixv
> # Calculate the inverse of matrixv
> invmat <- solve(a=matrixv)
> # Multiply inverse with matrix
> round(invmat %*% matrixv, 4)
> round(matrixv %*% invmat, 4)
>
> # Calculate eigenvectors and eigenvalues
> eigend <- eigen(matrixv)
> eigenvec <- eigend$vectors
>
> # Perform eigen decomposition of inverse
> inveigen <- eigenvec %*% (t(eigenvec) / eigend$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* (Σ) and its left and right *singular* matrices ($\mathbb{U}$ and $\mathbb{V}$):

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

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: nleft > nright
> nleft <- 6 ; nright <- 4
> matrixv <- matrix(runif(nleft*nright), nc=nright)
> # Calculate generalized inverse of matrixv
> invmat <- MASS::ginv(matrixv)
> round(invmat %*% matrixv, 4)
> all.equal(matrixv, matrixv %*% invmat %*% matrixv)
> # Random rectangular matrix: nleft < nright
> nleft <- 4 ; nright <- 6
> matrixv <- matrix(runif(nleft*nright), nc=nright)
> # Calculate generalized inverse of matrixv
> invmat <- MASS::ginv(matrixv)
> all.equal(matrixv, matrixv %*% invmat %*% matrixv)
> round(matrixv %*% invmat, 4)
> round(invmat %*% matrixv, 4)
> # Perform singular value decomposition
> svdec <- svd(matrixv)
> # 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(matrixv) %*% matrixv) %*% t(matrixv)
> 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:

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

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

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

```
> # 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 matrixv
> all.equal(matrixv, matrixv %*% invsvd %*% matrixv)
> # Calculate regularized inverse using MASS::ginv()
> invmat <- MASS::ginv(matrixv)
> all.equal(invsvd, invmat)
> # Calculate Moore-Penrose pseudo-inverse
> invmp <- MASS::ginv(t(matrixv) %*% matrixv) %*% t(matrixv)
> 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:

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

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 <- matrixv %*% invmat
> round(unitmat, 4)
> round(matrixv %*% 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
> matrixv <- matrix(c(1, 2, -1, 2), nc=2)
> vectorv <- c(2, 1)
> # Calculate the inverse of matrixv
> invmat <- solve(a=matrixv)
> invmat %*% matrixv
> # Calculate solution using inverse of matrixv
> solutionv <- invmat %*% vectorv
> matrixv %*% solutionv
> # Calculate solution of linear system
> solutionv <- solve(a=matrixv, b=vectorv)
> matrixv %*% solutionv
```

One-dimensional Optimization Using The Functional `optimize()`

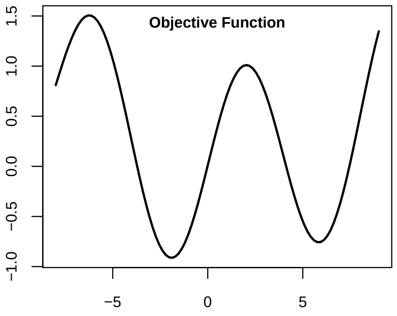
The functional `optimize()` performs *one-dimensional* optimization over a single independent variable.

`optimize()` searches for the minimum of the objective function with respect to its first argument, in the specified interval.

`optimize()` returns a list containing the location of the minimum and the objective function value,

The argument `tol` specifies the numerical accuracy, with smaller values of `tol` requiring more computations.

```
> # Display the structure of optimize()
> str(optimize)
> # Objective function with multiple minima
> objfun <- function(input, param1=0.01) {
+   sin(0.25*pi*input) + param1*(input-1)^2
+ }  # end objfun
> optiml <- optimize(f=objfun, interval=c(-4, 2))
> class(optiml)
> unlist(optiml)
> # Find minimum in different interval
> unlist(optimize(f=objfun, interval=c(0, 8)))
> # Find minimum with less accuracy
> accl <- 1e4*.Machine$double.eps^0.25
> unlist(optimize(f=objfun, interval=c(0, 8), tol=accl))
> # Microbenchmark optimize() with less accuracy
> library(microbenchmark)
> summary(microbenchmark(
+   more_accurate = optimize(f=objfun, interval=c(0, 8)),
+   less_accurate = optimize(f=objfun, interval=c(0, 8), tol=accl),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
```



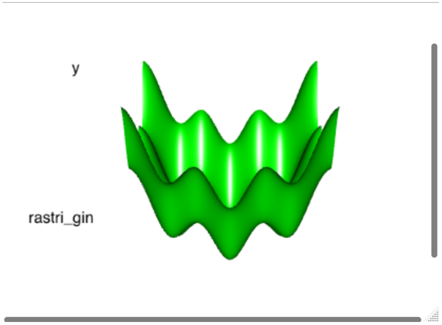
```
> # Plot the objective function
> curve(expr=objfun, type="l", xlim=c(-8, 9),
+ xlab="", ylab="", lwd=2)
> # Add title
> title(main="Objective Function", line=-1)
```

Package *rgl* for Interactive 3d Surface Plots

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

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

```
> # Rastrigin function
> rastrigin <- function(x, y, param=25) {
+   x^2 + y^2 - param*(cos(x) + cos(y))
+ }  # end rastrigin
> # Rastrigin function is vectorized!
> rastrigin(c(-10, 5), c(-10, 5))
> # Set rgl options and load package rgl
> library(rgl)
> options(rgl.useNULL=TRUE)
> # Draw 3d surface plot of function
> rgl::persp3d(x=rastrigin, xlim=c(-10, 10), ylim=c(-10, 10),
+   col="green", axes=FALSE, param=15)
> # Render the 3d surface plot of function
> rgl::rglwidget(elementId="plot3drgl", width=400, height=400)
```



Multi-dimensional Optimization Using optim()

The function optim() performs *multi-dimensional* optimization.

The argument fn is the objective function to be minimized.

The argument of fn that is to be optimized, must be a vector argument.

The argument par is the initial vector argument value.

optim() accepts additional parameters bound to the dots "..." argument, and passes them to the fn objective function.

The arguments lower and upper specify the search range for the variables of the objective function fn.

method="L-BFGS-B" specifies the quasi-Newton *gradient* optimization method.

optim() returns a list containing the location of the minimum and the objective function value.

The *gradient* methods used by optim() can only find the local minimum, not the global minimum.

```
> # Rastrigin function with vector argument for optimization
> rastrigin <- function(vectorv, param=25) {
+   sum(vectorv^2 - param*cos(vectorv))
+ }  # end rastrigin
> vectorv <- c(pi, pi/4)
> rastrigin(vectorv=vectorv)
> # Draw 3d surface plot of Rastrigin function
> rgl::persp3d(
+   x=Vectorize(function(x, y) rastrigin(vectorv=c(x, y))),
+   xlim=c(-10, 10), ylim=c(-10, 10),
+   col="green", axes=FALSE, zlab="", main="rastrigin")
> # Optimize with respect to vector argument
> optiml <- optim(par=vectorv, fn=rastrigin,
+         method="L-BFGS-B",
+         upper=c(14*pi, 14*pi),
+         lower=c(pi/2, pi/2),
+         param=1)
> # Optimal parameters and value
> optiml$par
> optiml$value
> rastrigin(optiml$par, param=1)
```

Downloading Treasury Bond Rates from *FRED*

The constant maturity Treasury rates are yields of hypothetical fixed-maturity bonds, interpolated from the market yields of actual Treasury bonds.

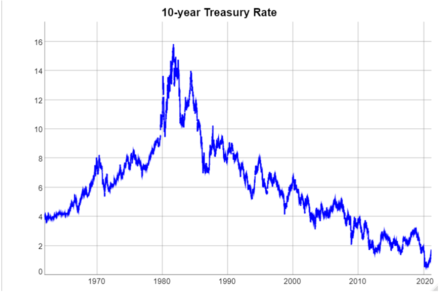
The *FRED* database contains current and historical constant maturity Treasury rates,

https://fred.stlouisfed.org/series/DGS5

`quantmod::getSymbols()` creates objects in the specified *environment* from the input strings (names).

It then assigns the data to those objects, without returning them as a function value, as a *side effect*.

```
> # Symbols for constant maturity Treasury rates
> symbolv <- c("DGS1", "DGS2", "DGS5", "DGS10", "DGS20", "DGS30")
> # Create new environment for time series
> ratesenv <- new.env()
> # Download time series for symbolv into ratesenv
> quantmod::getSymbols(symbolv, env=ratesenv, src="FRED")
> # List files in ratesenv
> ls(ratesenv)
> # Get class of all objects in ratesenv
> sapply(ratesenv, class)
> # Get class of all objects in R workspace
> sapply(ls(), function(name) class(get(name)))
> # Save the time series environment into a binary .RData file
> save(ratesenv, file="/Users/jerzy/Develop/lecture_slides/data/rat
```



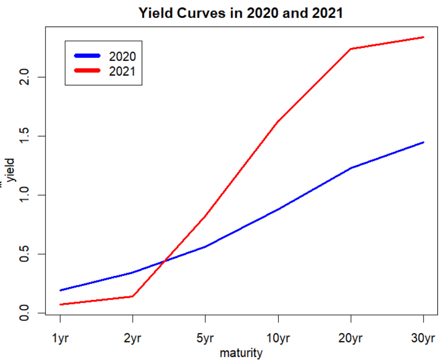
```
> # Get class of time series object DGS10
> class(get(x="DGS10", envir=ratesenv))
> # Another way
> class(ratesenv$DGS10)
> # Get first 6 rows of time series
> head(ratesenv$DGS10)
> # Plot dygraphs of 10-year Treasury rate
> dygraphs::dygraph(ratesenv$DGS10, main="10-year Treasury Rate") %>
+   dyOptions(colors="blue", strokeWidth=2)
> # Plot 10-year constant maturity Treasury rate
> x11(width=6, height=5)
> par(mar=c(2, 2, 0, 0), oma=c(0, 0, 0, 0))
> chart_Series(ratesenv$DGS10["1990/"], name="10-year Treasury Rate'
```

Treasury Yield Curve

The *yield curve* is a vector of interest rates at different maturities, on a given date.

The *yield curve* shape changes depending on the economic conditions: in recessions rates drop and the curve flattens, while in expansions rates rise and the curve steepens.

```
> # Load constant maturity Treasury rates
> load(file="/Users/jerzy/Develop/lecture_slides/data/rates_data.RDa
> # Get most recent yield curve
> ycnow <- eapply(ratesenv, xts::last)
> class(ycnow)
> ycnow <- do.call(cbind, ycnow)
> # Check if 2020-03-25 is not a holiday
> date2020 <- as.Date("2020-03-25")
> weekdays(date2020)
> # Get yield curve from 2020-03-25
> yc2020 <- eapply(ratesenv, function(x) x[date2020])
> yc2020 <- do.call(cbind, yc2020)
> # Combine the yield curves
> rates <- c(yc2020, ycnow)
> # Rename columns and rows, sort columns, and transpose into matr
> colnames(rates) <- substr(colnames(rates), start=4, stop=11)
> rates <- rates[, order(as.numeric(colnames(rates)))]
> colnames(rates) <- paste0(colnames(rates), "yr")
> rates <- t(rates)
> colnames(rates) <- substr(colnames(rates), start=1, stop=4)
```

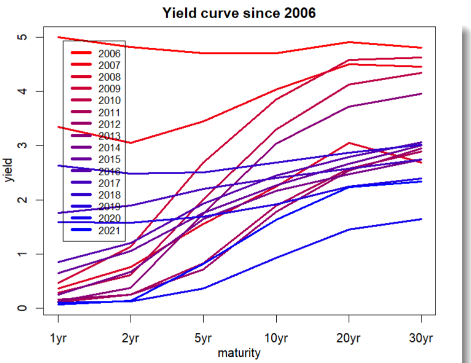


```
> # Plot using matplot()
> colors <- c("blue", "red")
> matplot(rates, main="Yield Curves in 2020 and 2021", xaxt="n", lwd
+   type="l", xlab="maturity", ylab="yield", col=colors)
> # Add x-axis
> axis(1, seq_along(rownames(rates)), rownames(rates))
> # Add legend
> legend("topleft", legend=colnames(rates),
+  col=colors, lty=1, lwd=6, inset=0.05, cex=1.0)
```

Treasury Yield Curve Over Time

The *yield curve* has changed shape dramatically depending on the economic conditions: in recessions rates drop and the curve flattens, while in expansions rates rise and the curve steepens.

```
> # Load constant maturity Treasury rates
> load(file="/Users/jerzy/Develop/lecture_slides/data/rates_data.RDa
> # Get end-of-year dates since 2006
> dates <- xts::endpoints(ratesenv$DGS1["2006/"], on="years")
> dates <- zoo::index(ratesenv$DGS1["2006/"][dates])
> # Create time series of end-of-year rates
> rates <- eapply(ratesenv, function(ratev) ratev[dates])
> rates <- rutils::do_call(cbind, rates)
> # Rename columns and rows, sort columns, and transpose into matrix
> colnames(rates) <- substr(colnames(rates), start=4, stop=11)
> rates <- rates[, order(as.numeric(colnames(rates)))]
> colnames(rates) <- paste0(colnames(rates), "yr")
> rates <- t(rates)
> colnames(rates) <- substr(colnames(rates), start=1, stop=4)
> # Plot matrix using plot.zoo()
> colors <- colorRampPalette(c("red", "blue"))(NCOL(rates))
> plot.zoo(rates, main="Yield curve since 2006", lwd=3, xaxt="n",
+    plot.type="single", xlab="maturity", ylab="yield", col=colors)
> # Add x-axis
> axis(1, seq_along(rownames(rates)), rownames(rates))
> # Add legend
> legend("topleft", legend=colnames(rates),
+  col=colors, lty=1, lwd=4, inset=0.05, cex=0.8)
```



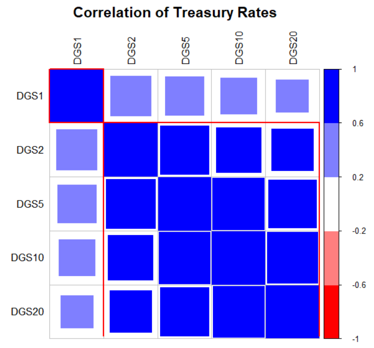
```
> # Alternative plot using matplot()
> matplot(rates, main="Yield curve since 2006", xaxt="n", lwd=3, lty
+   type="l", xlab="maturity", ylab="yield", col=colors)
> # Add x-axis
> axis(1, seq_along(rownames(rates)), rownames(rates))
> # Add legend
> legend("topleft", legend=colnames(rates),
+  col=colors, lty=1, lwd=4, inset=0.05, cex=0.8)
```

Covariance Matrix of Interest Rates

The covariance matrix $\mathbb{C}$, of the interest rate matrix $\mathbf{r}$ is given by:

$$\mathbb{C} = \frac{(\mathbf{r} - \bar{\mathbf{r}})^T (\mathbf{r} - \bar{\mathbf{r}})}{n - 1}$$

```
> # Extract rates from ratesenv
> symbolv <- c("DGS1", "DGS2", "DGS5", "DGS10", "DGS20")
> rates <- mget(symbolv, envir=ratesenv)
> rates <- rutils::do_call(cbind, rates)
> rates <- zoo::na.locf(rates, na.rm=FALSE)
> rates <- zoo::na.locf(rates, fromLast=TRUE)
> # Calculate daily percentage rates changes
> retsp <- rutils::diffit(log(rates))
> # De-mean the returns
> retsp <- lapply(retsp, function(x) {x - mean(x)})
> retsp <- rutils::do_call(cbind, retsp)
> sapply(retsp, mean)
> # Covariance and Correlation matrices of Treasury rates
> covmat <- cov(retsp)
> cormat <- cor(retsp)
> # Reorder correlation matrix based on clusters
> library(corrplot)
> ordern <- corrMatOrder(cormat, order="hclust",
+   hclust.method="complete")
> cormat <- cormat[ordern, ordern]
```



```
> # Plot the correlation matrix
> x11(width=6, height=6)
> colors <- colorRampPalette(c("red", "white", "blue"))
> corrplot(cormat, title=NA, tl.col="black",
+     method="square", col=colors(NCOL(cormat)), tl.cex=0.8,
+     cl.offset=0.75, cl.cex=0.7, cl.align.text="l", cl.ratio=0.25)
> title("Correlation of Treasury Rates", line=1)
> # Draw rectangles on the correlation matrix plot
> corrRect.hclust(cormat, k=NROW(cormat) %/% 2,
+   method="complete", col="red")
```

Principal Component Vectors

Principal components are linear combinations of the k return vectors $\mathbf{r}_i$:

$$\mathbf{pc}_j = \sum_{i=1}^{k} w_{ij} \, \mathbf{r}_i$$

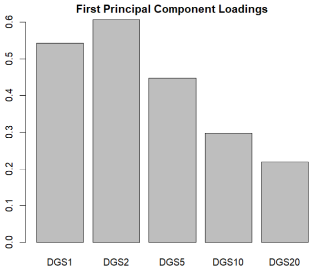
Where $\mathbf{w}_j$ is a vector of weights (loadings) of the *principal component* j, with $\mathbf{w}_j^T \mathbf{w}_j = 1$.

The weights $\mathbf{w}_j$ are chosen to maximize the variance of the *principal components*, under the condition that they are orthogonal:

$$\mathbf{w}_j = \arg \max \left\{ \mathbf{pc}_j^T \, \mathbf{pc}_j \right\}$$

$$\mathbf{pc}_i^T \, \mathbf{pc}_j = 0 \; (i \neq j)$$

```
> # Create initial vector of portfolio weights
> nweights <- NROW(symbolv)
> weights <- rep(1/sqrt(nweights), nweights)
> names(weights) <- symbolv
> # Objective function equal to minus portfolio variance
> objfun <- function(weights, retsp) {
+   retsp <- retsp %*% weights
+   -1e7*var(retsp) + 1e7*(1 - sum(weights*weights))^2
+ }  # end objfun
> # Objective function for equal weight portfolio
> objfun(weights, retsp)
> # Compare speed of vector multiplication methods
> library(microbenchmark)
> summary(microbenchmark(
+   transp=t(retsp) %*% retsp,
+   sumv=sum(retsp*retsp),
```



```
> # Find weights with maximum variance
> optiml <- optim(par=weights,
+   fn=objfun,
+   retsp=retsp,
+   method="L-BFGS-B",
+   upper=rep(5.0, nweights),
+   lower=rep(-5.0, nweights))
> # Optimal weights and maximum variance
> weights1 <- optiml$par
> objfun(weights1, retsp)
> # Plot first principal component loadings
> x11(width=6, height=5)
> par(mar=c(3, 3, 2, 1), oma=c(0, 0, 0, 0), mgp=c(2, 1, 0))
> barplot(weights1, names.arg=names(weights1),
+   xlab="", ylab="", main="First Principal Component Loadings")
```

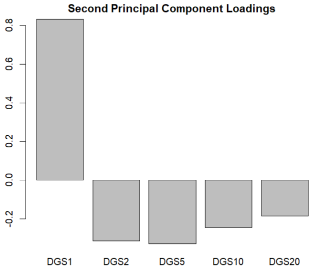
Higher Order Principal Components

The *second principal component* can be calculated by maximizing its variance, under the constraint that it must be orthogonal to the *first principal component*.

Similarly, higher order *principal components* can be calculated by maximizing their variances, under the constraint that they must be orthogonal to all the previous *principal components*.

The number of principal components is equal to the dimension of the covariance matrix.

```
> # pc1 weights and returns
> pc1 <- drop(retsp %*% weights1)
> # Redefine objective function
> objfun <- function(weights, retsp) {
+   retsp <- retsp %*% weights
+   -1e7*var(retsp) + 1e7*(1 - sum(weights^2))^2 +
+     1e7*sum(weights1*weights)^2
+ }  # end objfun
> # Find second principal component weights
> optiml <- optim(par=weights,
+              fn=objfun,
+              retsp=retsp,
+              method="L-BFGS-B",
+              upper=rep(5.0, nweights),
+              lower=rep(-5.0, nweights))
```



```
> # pc2 weights and returns
> weights2 <- optiml$par
> pc2 <- drop(retsp %*% weights2)
> sum(pc1*pc2)
> # Plot second principal component loadings
> barplot(weights2, names.arg=names(weights2),
+   xlab="", ylab="", main="Second Principal Component Loadings")
```

Eigenvalues of the Covariance Matrix

The portfolio variance: $\mathbf{w}^T \mathbb{C} \, \mathbf{w}$ can be maximized under the *quadratic* weights constraint $\mathbf{w}^T \mathbf{w} = 1$, by maximizing the *Lagrangian* $\mathcal{L}$:

$$\mathcal{L} = \mathbf{w}^T \mathbb{C} \, \mathbf{w} \; - \; \lambda \, (\mathbf{w}^T \mathbf{w} - 1)$$

Where λ is a *Lagrange multiplier*.

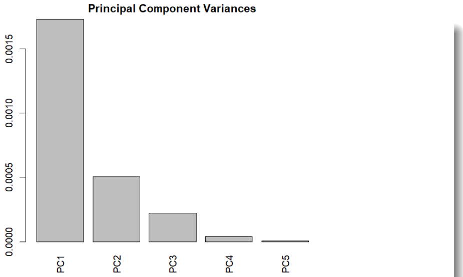
The maximum variance portfolio weights can be found by differentiating $\mathcal{L}$ with respect to $\mathbf{w}$ and setting it to zero:

$$\mathbb{C} \, \mathbf{w} = \lambda \, \mathbf{w}$$

The above is the *eigenvalue* equation of the covariance matrix $\mathbb{C}$, with the optimal weights $\mathbf{w}$ forming an *eigenvector*, and λ is the *eigenvalue* corresponding to the *eigenvector* $\mathbf{w}$.

The *eigenvalues* are the variances of the *eigenvectors*, and their sum is equal to the sum of the return variances:

$$\sum_{i=1}^{k} \lambda_i = \frac{1}{1-k} \sum_{i=1}^{k} \mathbf{r}_i^T \mathbf{r}_i$$



```
> eigend <- eigen(covmat)
> eigend$vectors
> # Compare with optimization
> all.equal(sum(diag(covmat)), sum(eigend$values))
> all.equal(abs(eigend$vectors[, 1]), abs(weights1), check.attribute
> all.equal(abs(eigend$vectors[, 2]), abs(weights2), check.attribute
> all.equal(eigend$values[1], var(pc1), check.attributes=FALSE)
> all.equal(eigend$values[2], var(pc2), check.attributes=FALSE)
> # Eigenvalue equations are satisfied approximately
> (covmat %*% weights1) / weights1 / var(pc1)
> (covmat %*% weights2) / weights2 / var(pc2)
> # Plot eigenvalues
> barplot(eigend$values, names.arg=paste0("PC", 1:nweights),
+   las=3, xlab="", ylab="", main="Principal Component Variances")
```

Principal Component Analysis Versus *Eigen Decomposition*

Principal Component Analysis (*PCA*) is equivalent to the *eigen decomposition* of either the correlation or the covariance matrix.

If the input time series *are* scaled, then *PCA* is equivalent to the eigen decomposition of the *correlation matrix*.

If the input time series *are not* scaled, then *PCA* is equivalent to the eigen decomposition of the *covariance matrix*.

Scaling the input time series improves the accuracy of the *PCA dimension reduction*, allowing a smaller number of *principal components* to more accurately capture the data contained in the input time series.

The function `prcomp()` performs *Principal Component Analysis* on a matrix of data (with the time series as columns), and returns the results as a list of class `prcomp`.

The `prcomp()` argument `scale=TRUE` specifies that the input time series should be scaled by their standard deviations.

```
> # Eigen decomposition of correlation matrix
> eigend <- eigen(cormat)
> # Perform PCA with scaling
> pcad <- prcomp(retsp, scale=TRUE)
> # Compare outputs
> all.equal(eigend$values, pcad$sdev^2)
> all.equal(abs(eigend$vectors), abs(pcad$rotation),
+     check.attributes=FALSE)
> # Eigen decomposition of covariance matrix
> eigend <- eigen(covmat)
> # Perform PCA without scaling
> pcad <- prcomp(retsp, scale=FALSE)
> # Compare outputs
> all.equal(eigend$values, pcad$sdev^2)
> all.equal(abs(eigend$vectors), abs(pcad$rotation),
+     check.attributes=FALSE)
```

Principal Component Analysis of the Yield Curve

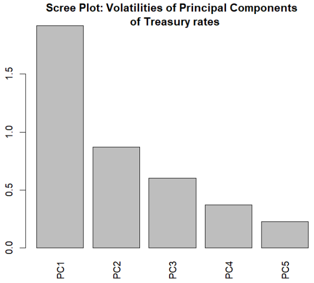
Principal Component Analysis (*PCA*) is a *dimension reduction* technique, that explains the returns of a large number of correlated time series as linear combinations of a smaller number of principal component time series.

The input time series are often scaled by their standard deviations, to improve the accuracy of *PCA dimension reduction*, so that more information is retained by the first few *principal component* time series.

If the input time series are not scaled, then *PCA* analysis is equivalent to the *eigen decomposition* of the covariance matrix, and if they are scaled, then *PCA* analysis is equivalent to the *eigen decomposition* of the correlation matrix.

The function prcomp() performs *Principal Component Analysis* on a matrix of data (with the time series as columns), and returns the results as a list of class prcomp.

The prcomp() argument scale=TRUE specifies that the input time series should be scaled by their standard deviations.



A *scree plot* is a bar plot of the volatilities of the *principal components*.

```
> # Perform principal component analysis PCA
> pcad <- prcomp(retsp, scale=TRUE)
> # Plot standard deviations
> barplot(pcad$sdev, names.arg=colnames(pcad$rotation),
+   las=3, xlab="", ylab="",
+   main="Scree Plot: Volatilities of Principal Components
+   of Treasury rates")
```

Yield Curve Principal Component Loadings (Weights)

Principal component loadings are the weights of portfolios which have mutually orthogonal returns.

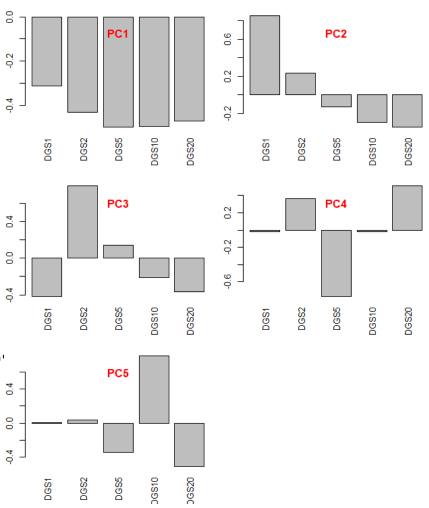
The *principal component* portfolios represent the different orthogonal modes of the data variance.

The first *principal component* of the *yield curve* is the correlated movement of all rates up and down.

The second *principal component* is *yield curve* steepening and flattening.

The third *principal component* is the *yield curve* butterfly movement.

```
> # Calculate principal component loadings (weights)
> pcad$rotation
> # Plot loading barplots in multiple panels
> par(mfrow=c(3,2))
> par(mar=c(3.5, 2, 2, 1), oma=c(0, 0, 0, 0))
> for (ordern in 1:NCOL(pcad$rotation)) {
+   barplot(pcad$rotation[, ordern], las=3, xlab="", ylab="", main='
+   title(paste0("PC", ordern), line=-2.0, col.main="red")
+ }  # end for
```



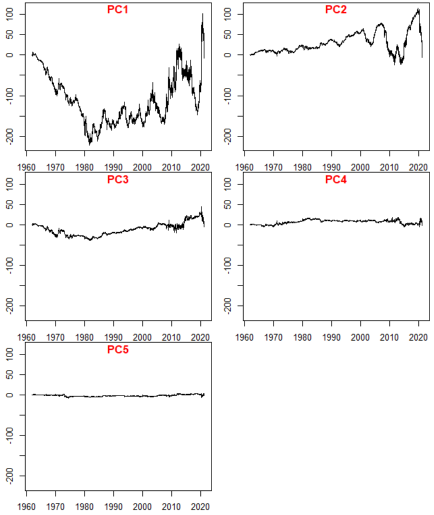
Yield Curve Principal Component Time Series

The time series of the *principal components* can be calculated by multiplying the loadings (weights) times the original data.

The *principal component* time series have mutually orthogonal returns.

Higher order *principal components* are gradually less volatile.

```
> # Standardize (de-mean and scale) the returns
> retsp <- lapply(retsp, function(x) {(x - mean(x))/sd(x)})
> retsp <- rutils::do_call(cbind, retsp)
> sapply(retsp, mean)
> sapply(retsp, sd)
> # Calculate principal component time series
> pcacum <- retsp %*% pcad$rotation
> all.equal(pcad$x, pcacum, check.attributes=FALSE)
> # Calculate products of principal component time series
> round(t(pcacum) %*% pcacum, 2)
> # Coerce to xts time series
> pcacum <- xts(pcacum, order.by=zoo::index(retsp))
> pcacum <- cumsum(pcacum)
> # Plot principal component time series in multiple panels
> par(mfrow=c(3,2))
> par(mar=c(2, 2, 0, 1), oma=c(0, 0, 0, 0))
> rangev <- range(pcacum)
> for (ordern in 1:NCOL(pcacum)) {
+   plot.zoo(pcacum[, ordern], ylim=rangev, xlab="", ylab="")
+   title(paste0("PC", ordern), line=-1, col.main="red")
+ }  # end for
```



Inverting Principal Component Analysis

The original time series can be calculated *exactly* from the time series of all the *principal components*, by inverting the loadings matrix.

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

```
> # Invert all the principal component time series
> retspca <- retsp %*% pcad$rotation
> solved <- retspca %*% solve(pcad$rotation)
> all.equal(coredata(retsp), solved)
```

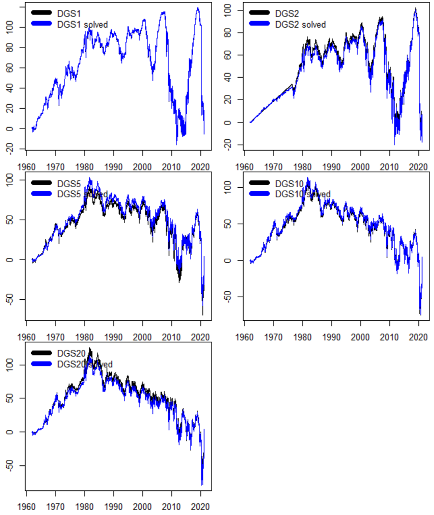
Dimension Reduction Using Principal Component Analysis

The original time series can be calculated *approximately* from just the first few *principal components*, which demonstrates that *PCA* is a form of *dimension reduction*.

A popular rule of thumb is to use the *principal components* with the largest variances, which sum up to 80% of the total variance of returns.

The *Kaiser-Guttman* rule uses only *principal components* with variance greater than 1.

```
> # Invert first 3 principal component time series
> solved <- retspca[, 1:3] %*% solve(pcad$rotation)[1:3, ]
> solved <- xts::xts(solved, zoo::index(retsp))
> solved <- cumsum(solved)
> retc <- cumsum(retsp)
> # Plot the solved returns
> par(mfrow=c(3,2))
> par(mar=c(2, 2, 0, 1), oma=c(0, 0, 0, 0))
> for (symbol in symbolv) {
+   plot.zoo(cbind(retc[, symbol], solved[, symbol]),
+     plot.type="single", col=c("black", "blue"), xlab="", ylab="")
+   legend(x="topleft", bty="n",
+    legend=paste0(symbol, c("", " solved")),
+    title=NULL, inset=0.0, cex=1.0, lwd=6,
+    lty=1, col=c("black", "blue"))
+ }  # end for
```



Calibrating Yield Curve Using Package *RQuantLib*

The package *RQuantLib* is an interface to the *QuantLib* open source C/C++ library for quantitative finance, mostly designed for pricing fixed-income instruments and options.

The function `DiscountCurve()` calibrates a *zero coupon yield curve* from *money market* rates, *Eurodollar* futures, and *swap* rates.

The function `DiscountCurve()` interpolates the *zero coupon* rates into a vector of dates specified by the `times` argument.

```
> library(RQuantLib)  # Load RQuantLib
> # Specify curve parameters
> curve_params <- list(tradeDate=as.Date("2018-01-17"),
+                settleDate=as.Date("2018-01-19"),
+                dt=0.25,
+                interpWhat="discount",
+                interpHow="loglinear")
> # Specify market data: prices of FI instruments
> market_data <- list(d3m=0.0363,
+               fut1=96.2875,
+               fut2=96.7875,
+               fut3=96.9875,
+               fut4=96.6875,
+               s5y=0.0443,
+               s10y=0.05165,
+               s15y=0.055175)
> # Specify dates for calculating the zero rates
> disc_dates <- seq(0, 10, 0.25)
> # Specify the evaluation (as of) date
> setEvaluationDate(as.Date("2018-01-17"))
> # Calculate the zero rates
> disc_curves <- DiscountCurve(params=curve_params,
+                          tsQuotes=market_data,
+                          times=disc_dates)
> # Plot the zero rates
> x11()
> plot(x=disc_curves$zerorates, t="l", main="zerorates")
```

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:

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

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

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

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

Formula Objects

Formulas in R are defined using the "~" operator followed by a series of terms separated by the "+" operator.

Formulas can be defined as separate objects, manipulated, and passed to functions.

The formula "z ~ x" means the *response vector z* is explained by the *predictor x* (also called the *explanatory variable* or *independent variable*).

The formula "z ~ x + y" represents a linear model: z = ax + by + c.

The formula "z ~ x - 1" or "z ~ x + 0" represents a linear model with zero intercept: $z = ax$.

The function `update()` modifies existing `formulas`.

The "." symbol represents either all the remaining data, or the variable that was in this part of the formula.

```
> # Formula of linear model with zero intercept
> formulav <- z ~ x + y - 1
> formulav
>
> # Collapse vector of strings into single text string
> paste0("x", 1:5)
> paste(paste0("x", 1:5), collapse="+")
>
> # Create formula from text string
> formulav <- as.formula(
+   # Coerce text strings to formula
+   paste("z ~ ",
+   paste(paste0("x", 1:5), collapse="+")
+   )  # end paste
+ )  # end as.formula
> class(formulav)
> formulav
> # Modify the formula using "update"
> update(formulav, log(.) ~ . + beta)
```

Simple *Linear Regression*

A Simple Linear Regression is a linear model between a *response vector* y and a single *predictor* x, defined by the formula:

$$y_i = \alpha + \beta x_i + \varepsilon_i$$

α and β are the unknown *regression coefficients*.

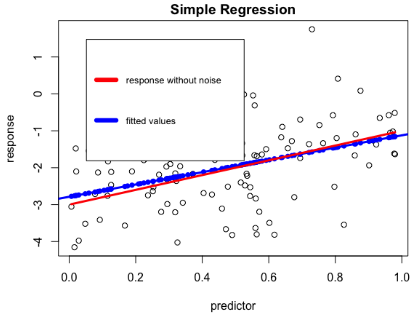
ε_i are the *residuals*, which are usually assumed to be standard normally distributed $\phi(0, \sigma_\varepsilon)$, independent, and stationary.

In the Ordinary Least Squares method (*OLS*), the regression parameters are estimated by minimizing the *Residual Sum of Squares* (*RSS*):

$$RSS = \sum_{i=1}^{n} \varepsilon_i^2 = \sum_{i=1}^{n} (y_i - \alpha - \beta x_i)^2$$
$$= (y - \alpha \mathbb{1} - \beta x)^T (y - \alpha \mathbb{1} - \beta x)$$

Where $\mathbb{1}$ is the unit vector, with $\mathbb{1}^T \mathbb{1} = n$ and $\mathbb{1}^T x = x^T \mathbb{1} = \sum_{i=1}^{n} x_i$

The data consists of n pairs of observations (x_i, y_i) of the response and predictor variables, with the index i ranging from 1 to n.



```
> # Define explanatory (predictor) variable
> nrows <- 100
> set.seed(1121)  # Initialize random number generator
> predictor <- runif(nrows)
> noise <- rnorm(nrows)
> # Response equals linear form plus random noise
> response <- (-3 + 2*predictor + noise)
```

The *response vector* and the *predictor matrix* don't have to be normally distributed.

Solution of *Linear Regression*

The *OLS* solution for the *regression coefficients* is found by equating the *RSS* derivatives to zero:

$$RSS_\alpha = -2(y - \alpha\mathbb{1} - \beta x)^T \mathbb{1} = 0$$

$$RSS_\beta = -2(y - \alpha\mathbb{1} - \beta x)^T x = 0$$

The solution for α is given by:

$$\alpha = \bar{y} - \beta\bar{x}$$

The solution for β can be obtained by manipulating the equation for RSS_β as follows:

$$(y - (\bar{y} - \beta\bar{x})\mathbb{1} - \beta x)^T (x - \bar{x}\mathbb{1}) =$$

$$((y - \bar{y}\mathbb{1}) - \beta(x - \bar{x}\mathbb{1}))^T (x - \bar{x}\mathbb{1}) =$$

$$(\hat{y} - \beta\hat{x})^T \hat{x} = \hat{y}^T\hat{x} - \beta\hat{x}^T\hat{x} = 0$$

Where $\hat{x} = x - \bar{x}\mathbb{1}$ and $\hat{y} = y - \bar{y}\mathbb{1}$ are the de-meaned variables. Then β is given by:

$$\beta = \frac{\hat{y}^T\hat{x}}{\hat{x}^T\hat{x}} = \frac{\sigma_y}{\sigma_x}\rho_{xy}$$

β is proportional to the correlation coefficient ρ_{xy} between the response and predictor variables.

If the response and predictor variables have zero mean, then $\alpha = 0$ and $\beta = \frac{y^T x}{x^T x}$.

The *residuals* $\varepsilon = y - \alpha\mathbb{1} - \beta x$ have zero mean: $RSS_\alpha = -2\varepsilon^T\mathbb{1} = 0$.

The *residuals* ε are orthogonal to the *predictor* x: $RSS_\beta = -2\varepsilon^T x = 0$.

The expected value of the *RSS* is equal to the *degrees of freedom* $(n-2)$ times the variance σ_ε^2 of the *residuals* ε_i: $\mathbb{E}[RSS] = (n-2)\sigma_\varepsilon^2$.

```
> # Calculate de-meaned explanatory (predictor) and response vectors
> predictor_zm <- predictor - mean(predictor)
> response_zm <- response - mean(response)
> # Calculate the regression beta
> betav <- cov(predictor, response)/var(predictor)
> # Calculate the regression alpha
> alpha <- mean(response) - betav*mean(predictor)
```

Linear Regression Using Function `lm()`

Let the data generating process for the response variable be given as: $z = \alpha_{lat} + \beta_{lat} x + \varepsilon_{lat}$

Where α_{lat} and β_{lat} are latent (unknown) coefficients, and ε_{lat} is an unknown vector of random noise (error terms).

The error terms are the difference between the measured values of the response minus the (unknown) actual response values.

The function `lm()` fits a linear model into a set of data, and returns an object of class `"lm"`, which is a list containing the results of fitting the model:

- call - the model formula,
- coefficients - the fitted model coefficients (α, β_j),
- residuals - the model residuals (response minus fitted values),

The regression *residuals* are not the same as the error terms, because the regression coefficients are not equal to the coefficients of the data generating process.

```
> # Specify regression formula
> formulav <- response ~ predictor
> model <- lm(formulav)  # Perform regression
> class(model)  # Regressions have class lm
[1] "lm"
> attributes(model)
$names
 [1] "coefficients" "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"       "qr"            "df.residual"
 [9] "xlevels"      "call"          "terms"         "model"

$class
[1] "lm"
> eval(model$call$formula)  # Regression formula
response ~ predictor
> model$coeff  # Regression coefficients
(Intercept)   predictor
      -2.79        1.67
> all.equal(coef(model), c(alpha, betav),
+   check.attributes=FALSE)
[1] TRUE
```

The *Fitted Values* of Linear Regression

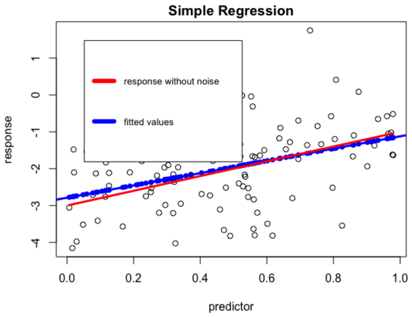
The *fitted values* y_{fit} are the estimates of the *response vector* obtained from the regression model:

$$y_{fit} = \alpha + \beta x$$

The *generic function* `plot()` produces a scatterplot when it's called on the regression formula.

`abline()` plots a straight line corresponding to the regression coefficients, when it's called on the regression object.

```
> fittedv <- (alpha + betav*predictor)
> all.equal(fittedv, model$fitted.values, check.attributes=FALSE)
> x11(width=5, height=4)  # Open x11 for plotting
> # Set plot parameters to reduce whitespace around plot
> par(mar=c(5, 5, 2, 1), oma=c(0, 0, 0, 0))
> # Plot scatterplot using formula
> plot(formulav, xlab="predictor", ylab="response")
> title(main="Simple Regression", line=0.5)
> # Add regression line
> abline(model, lwd=3, col="blue")
> # Plot fitted (predicted) response values
> points(x=predictor, y=model$fitted.values, pch=16, col="blue")
```



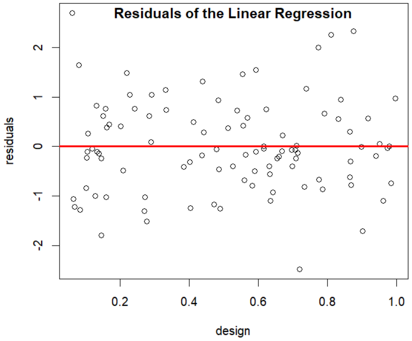
```
> # Plot response without noise
> lines(x=predictor, y=(response-noise), col="red", lwd=3)
> legend(x="topleft", # Add legend
+        legend=c("response without noise", "fitted values"),
+        title=NULL, inset=0.08, cex=0.8, lwd=6,
+        lty=1, col=c("red", "blue"))
```

Linear Regression Residuals

The *residuals* ε_i of a *linear regression* are defined as the *response vector* minus the fitted values:

$$\varepsilon_i = y_i - y_{fit}$$

```
> # Calculate the residuals
> fittedv <- (alpha + betav*predictor)
> residuals <- (response - fittedv)
> all.equal(residuals, model$residuals, check.attributes=FALSE)
[1] TRUE
> # Residuals are orthogonal to the predictor
> all.equal(sum(residuals*predictor), target=0)
[1] TRUE
> # Residuals are orthogonal to the fitted values
> all.equal(sum(residuals*fittedv), target=0)
[1] TRUE
> # Sum of residuals is equal to zero
> all.equal(mean(residuals), target=0)
[1] TRUE
```



```
> x11(width=6, height=5)  # Open x11 for plotting
> # Set plot parameters to reduce whitespace around plot
> par(mar=c(5, 5, 1, 1), oma=c(0, 0, 0, 0))
> # Extract residuals
> datav <- cbind(predictor, model$residuals)
> colnames(datav) <- c("predictor", "residuals")
> # Plot residuals
> plot(datav)
> title(main="Residuals of the Linear Regression", line=-1)
> abline(h=0, lwd=3, col="red")
```

Standard Errors of Regression Coefficients

The *residuals* are the source of error in the regression model, producing uncertainty in the *response vector y* and in the regression coefficients: $y_i = \alpha + \beta x_i + \varepsilon_i$.

The standard errors of the regression coefficients are equal to their standard deviations, given the *residuals* as the source of error.

Since $\beta = \frac{\hat{y}^T \hat{x}}{\hat{x}^T \hat{x}}$, then its variance is equal to:

$$\sigma_\beta^2 = \frac{1}{(n-2)} \frac{E[(\varepsilon^T \hat{x})^2]}{(\hat{x}^T \hat{x})^2} = \frac{1}{(n-2)} \frac{E[\varepsilon^2]}{\hat{x}^T \hat{x}} = \frac{\sigma_\varepsilon^2}{\hat{x}^T \hat{x}}$$

Since $\alpha = \bar{y} - \beta \bar{x}$, then its variance is equal to:

$$\sigma_\alpha^2 = \frac{\sigma_\varepsilon^2}{n} + \sigma_\beta^2 \bar{x}^2 = \sigma_\varepsilon^2 (\frac{1}{n} + \frac{\bar{x}^2}{\hat{x}^T \hat{x}})$$

```
> # Degrees of freedom of residuals
> degf <- model$df.residual
> # Standard deviation of residuals
> residsd <- sqrt(sum(residuals^2)/degf)
> # Standard error of beta
> betasd <- residsd/sqrt(sum(predictor_zm^2))
> # Standard error of alpha
> alphasd <- residsd*
+   sqrt(1/nrows + mean(predictor)^2/sum(predictor_zm^2))
```

Linear Regression Summary

The function `summary.lm()` produces a list of regression model diagnostic statistics:

- coefficients: matrix with estimated coefficients, their *t*-statistics, and *p*-values,
- r.squared: fraction of response variance explained by the model,
- adj.r.squared: r.squared adjusted for higher model complexity,
- fstatistic: ratio of variance explained by the model divided by unexplained variance,

The regression `summary` is a list, and its elements can be accessed individually.

```
> modelsum <- summary(model)  # Copy regression summary
> modelsum  # Print the summary to console

Call:
lm(formula = formulav)

Residuals:
   Min     1Q Median     3Q    Max
-2.133 -0.649  0.106  0.590  3.321

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   -2.787      0.196  -14.20  < 2e-16 ***
predictor      1.665      0.357    4.67  9.8e-06 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.988 on 98 degrees of freedom
Multiple R-squared:  0.182,Adjusted R-squared:  0.173
F-statistic: 21.8 on 1 and 98 DF,  p-value: 9.75e-06
> attributes(modelsum)$names  # get summary elements
 [1] "call"          "terms"          "residuals"     "coefficients"
 [5] "aliased"       "sigma"          "df"            "r.squared"
 [9] "adj.r.squared" "fstatistic"     "cov.unscaled"
```

Regression Model Diagnostic Statistics

The *null hypothesis* for regression is that the coefficients are *zero*.

The *t*-statistic (*t*-value) is the ratio of the estimated value divided by its standard error.

The *p*-value is the probability of obtaining values exceeding the *t*-statistic, assuming the *null hypothesis* is true.

A small *p*-value means that the regression coefficients are very unlikely to be zero (given the data).

The key assumption in the formula for the standard error is that the *residuals* are normally distributed, independent, and stationary.

If they are not, then the standard error and the *p*-value may be much bigger than reported by `summary.lm()`, and therefore the regression may not be statistically significant.

Asset returns are very far from normal, so the small *p*-values shouldn't be automatically interpreted as meaning that the regression is statistically significant.

```
> modelsum$coeff
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    -2.79      0.196  -14.20 1.61e-25
predictor       1.67      0.357    4.67 9.75e-06
> # Standard errors
> modelsum$coefficients[2, "Std. Error"]
[1] 0.357
> all.equal(c(alphasd, betasd),
+   modelsum$coefficients[, "Std. Error"],
+   check.attributes=FALSE)
[1] TRUE
> # R-squared
> modelsum$r.squared
[1] 0.182
> modelsum$adj.r.squared
[1] 0.173
> # F-statistic and ANOVA
> modelsum$fstatistic
value numdf dendf
 21.8   1.0  98.0
> anova(model)
Analysis of Variance Table

Response: response
          Df Sum Sq Mean Sq F value  Pr(>F)
predictor  1   21.3   21.25    21.8 9.8e-06 ***
Residuals 98   95.7    0.98
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

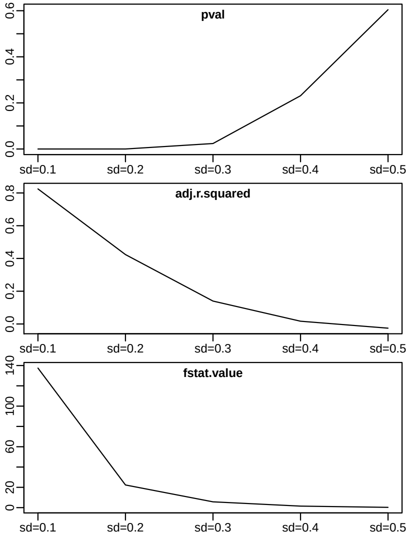
Weak Regression

If the relationship between the response and predictor variables is weak compared to the error terms (noise), then the regression will have low statistical significance.

```
> # High noise compared to coefficient
> response <- (-3 + 2*predictor + rnorm(nrows, sd=8))
> model <- lm(formulav)  # Perform regression
> # Values of regression coefficients are not
> # Statistically significant
> summary(model)

Call:
lm(formula = formulav)

Residuals:
    Min      1Q  Median      3Q     Max 
-16.430  -4.325   0.735   4.365  16.720 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    -1.65       1.44   -1.14     0.26
predictor      -1.70       2.62   -0.65     0.52

Residual standard error: 7.25 on 98 degrees of freedom
Multiple R-squared:  0.0043,Adjusted R-squared:  -0.00586 
F-statistic: 0.423 on 1 and 98 DF,  p-value: 0.517
```

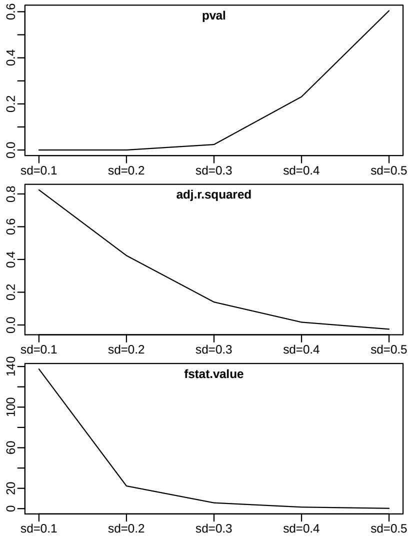
Influence of Noise on Regression

```
> reg_stats <- function(stdev) {  # Noisy regression
+   set.seed(1121)  # initialize number generator
+ # Define explanatory (predictor) and response variables
+   predictor <- rnorm(100, mean=2)
+   response <- (1 + 0.2*predictor +
+   rnorm(NROW(predictor), sd=stdev))
+ # Specify regression formula
+   formulav <- response ~ predictor
+ # Perform regression and get summary
+   modelsum <- summary(lm(formulav))
+ # Extract regression statistics
+   with(modelsum, c(pval=coefficients[2, 4],
+    adj_rsquared=adj.r.squared,
+    fstat=fstatistic[1]))
+ }  # end reg_stats
> # Apply reg_stats() to vector of std dev values
> vecsd <- seq(from=0.1, to=0.5, by=0.1)
> names(vecsd) <- paste0("sd=", vecsd)
> statsmat <- t(sapply(vecsd, reg_stats))
> # Plot in loop
> par(mfrow=c(NCOL(statsmat), 1))
> for (it in 1:NCOL(statsmat)) {
+   plot(statsmat[, it], type="l",
+  xaxt="n", xlab="", ylab="", main="")
+   title(main=colnames(statsmat)[it], line=-1.0)
+   axis(1, at=1:(NROW(statsmat)), labels=rownames(statsmat))
+ }  # end for
```



Influence of Noise on Regression Another Method

```
> reg_stats <- function(datav) {  # get regression
+ # Perform regression and get summary
+   colnamev <- colnames(datav)
+   formulav <- paste(colnamev[2], colnamev[1], sep="~")
+   modelsum <- summary(lm(formulav, data=datav))
+ # Extract regression statistics
+   with(modelsum, c(pval=coefficients[2, 4],
+    adj_rsquared=adj.r.squared,
+    fstat=fstatistic[1]))
+ }  # end reg_stats
> # Apply reg_stats() to vector of std dev values
> vecsd <- seq(from=0.1, to=0.5, by=0.1)
> names(vecsd) <- paste0("sd=", vecsd)
> statsmat <- t(sapply(vecsd, function(stdev) {
+     set.seed(1121)  # initialize number generator
+ # Define explanatory (predictor) and response variables
+     predictor <- rnorm(100, mean=2)
+     response <- (1 + 0.2*predictor +
+ rnorm(NROW(predictor), sd=stdev))
+     reg_stats(data.frame(predictor, response))
+     }))
> # Plot in loop
> par(mfrow=c(NCOL(statsmat), 1))
> for (it in 1:NCOL(statsmat)) {
+   plot(statsmat[, it], type="l",
+  xaxt="n", xlab="", ylab="", main="")
+   title(main=colnames(statsmat)[it], line=-1.0)
+   axis(1, at=1:(NROW(statsmat)),
+  labels=rownames(statsmat))
+ }  # end for
```



Linear Regression Diagnostic Plots

`plot()` produces diagnostic scatterplots for the *residuals*, when called on the regression object.

The diagnostic scatterplots allow for visual inspection to determine the quality of the regression fit.

"Residuals vs Fitted" is a scatterplot of the residuals vs. the predicted responses.

"Scale-Location" is a scatterplot of the square root of the standardized residuals vs. the predicted responses.

The residuals should be randomly distributed around the horizontal line representing zero residual error.

A pattern in the residuals indicates that the model was not able to capture the relationship between the variables, or that the variables don't follow the statistical assumptions of the regression model.

"Normal Q-Q" is the standard Q-Q plot, and the points should fall on the diagonal line, indicating that the residuals are normally distributed.

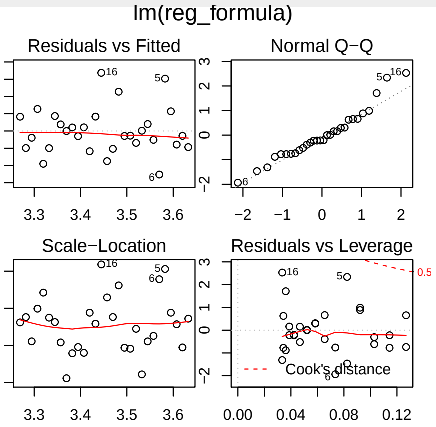
"Residuals vs Leverage" is a scatterplot of the residuals vs. their leverage.

Leverage measures the amount by which the fitted values would change if the response values were shifted by a small amount.

Cook's distance measures the influence of a single observation on the fitted values, and is proportional to the sum of the squared differences between predictions made with all observations and predictions made without the observation.

Points with large leverage, or a Cook's distance greater than 1 suggest the presence of an outlier or a poor model,

```
> par(mfrow=c(2, 2))  # Plot 2x2 panels
> plot(model)  # Plot diagnostic scatterplots
> plot(model, which=2)  # Plot just Q-Q
```



Durbin-Watson Test of Autocorrelation of Residuals

The *Durbin-Watson* test is designed to test the *null hypothesis* that the autocorrelations of regression *residuals* are equal to zero.

The test statistic is equal to:

$$DW = \frac{\sum_{i=2}^{n} (\varepsilon_i - \varepsilon_{i-1})^2}{\sum_{i=1}^{n} \varepsilon_i^2}$$

Where ε_i are the regression *residuals*.

The value of the *Durbin-Watson* statistic *DW* is close to zero for large positive autocorrelations, and close to four for large negative autocorrelations.

The *DW* is close to two for autocorrelations close to zero.

The *p*-value for the reg_model regression is large, and we conclude that the *null hypothesis* is TRUE, and the regression *residuals* are uncorrelated.

```
> library(lmtest)  # Load lmtest
> # Perform Durbin-Watson test
> lmtest::dwtest(model)

Durbin-Watson test

data:  model
DW = 2, p-value = 0.7
alternative hypothesis: true autocorrelation is greater than 0
```

The *Leverage* for Univariate Regression

We can add an extra unit column to the *predictor matrix* $\mathbb{X}$ so that the univariate regression can be written in *homogeneous form* as:

$$y = \mathbb{X}\beta + \varepsilon$$

With two *regression coefficients*: $\beta = (\alpha, \beta_1)$, and a *predictor matrix* $\mathbb{X}$ with two columns, with the first column equal to a unit vector.

After the second column of the *predictor matrix* $\mathbb{X}$ is de-meaned, its *covariance matrix* is given by:

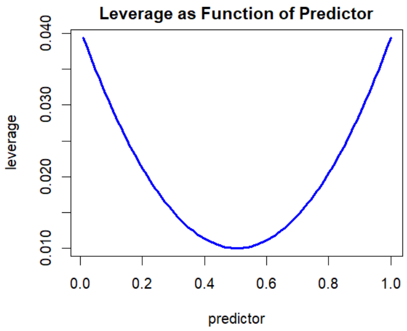
$$\mathbb{X}^T\mathbb{X} = \begin{pmatrix} n & 0 \\ 0 & \sum_{i=1}^n (x_i - \bar{x})^2 \end{pmatrix}$$

And the *influence matrix* $\mathbb{H}$ is given by:

$$\mathbb{H}_{ij} = [\mathbb{X}(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T]_{ij} = \frac{1}{n} + \frac{(x_i - \bar{x})(x_j - \bar{x})}{\sum_{i=1}^n (x_i - \bar{x})^2}$$

The first term above is due to the influence of the regression intercept α, and the second term is due to the influence of the regression slope β_1.

The diagonal elements of the *influence matrix* $\mathbb{H}_{ii}$ form the *leverage vector*.



```
> # Add unit column to the predictor matrix
> predictor <- cbind(rep(1, nrows), predictor)
> # Calculate generalized inverse of the predictor matrix
> invpred <- MASS::ginv(predictor)
> # Calculate the influence matrix
> influencem <- predictor %*% invpred
> # Plot the leverage vector
> ordern <- order(predictor[, 2])
> plot(x=predictor[ordern, 2], y=diag(influencem)[ordern],
+      type="l", lwd=3, col="blue",
+      xlab="predictor", ylab="leverage",
+      main="Leverage as Function of Predictor")
```

Covariance Matrix of Fitted Values in Univariate Regression

The *fitted values* y_{fit} can be considered to be *random variables* $\hat{y}_{fit}$:

$$\hat{y}_{fit} = \mathbb{H}\hat{y} = \mathbb{H}(y_{fit} + \hat{\varepsilon}) = y_{fit} + \mathbb{H}\hat{\varepsilon}$$

The *covariance matrix* of the *fitted values* $\hat{y}_{fit}$ is:

$$\sigma^2_{fit} = \frac{\mathbb{E}[\mathbb{H}\hat{\varepsilon}(\mathbb{H}\hat{\varepsilon})^T]}{d_{free}} = \frac{\mathbb{E}[\mathbb{H}\,\hat{\varepsilon}\hat{\varepsilon}^T\,\mathbb{H}^T]}{d_{free}} =$$

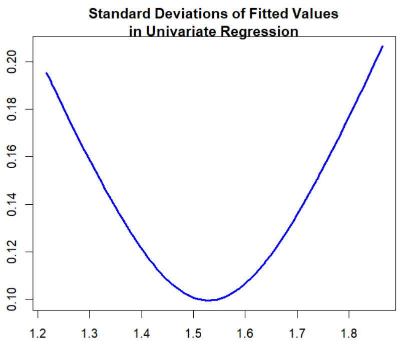
$$\frac{\mathbb{H}\,\mathbb{E}[\hat{\varepsilon}\hat{\varepsilon}^T]\,\mathbb{H}^T}{d_{free}} = \sigma^2_{\varepsilon}\,\mathbb{H} = \sigma^2_{\varepsilon}\,\mathbb{X}(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T$$

The square of the *influence matrix* $\mathbb{H}$ is equal to itself (it's idempotent): $\mathbb{H}\,\mathbb{H}^T = \mathbb{H}$.

The variance of the *fitted values* σ^2_{fit} increases with the distance of the *predictors* from their mean values.

This is because the *fitted values* farther from their mean are more sensitive to the variance of the regression slope.

```
> # Calculate the influence matrix
> influencem <- predictor %*% invpred
> # The influence matrix is idempotent
> all.equal(influencem, influencem %*% influencem)
```



```
> # Calculate covariance and standard deviations of fitted values
> betas <- invpred %*% response
> fittedv <- drop(predictor %*% betas)
> residuals <- drop(response - fittedv)
> degf <- (NROW(predictor) - NCOL(predictor))
> residvar <- sqrt(sum(residuals^2)/degf)
> fitcovar <- residvar*influencem
> fitsd <- sqrt(diag(fitcovar))
> # Plot the standard deviations
> fitsd <- cbind(fitted=fittedv, stddev=fitsd)
> fitsd <- fitsd[order(fittedv), ]
> plot(fitsd, type="l", lwd=3, col="blue",
+      xlab="Fitted Value", ylab="Standard Deviation",
+      main="Standard Deviations of Fitted Values\nin Univariate Reg
```

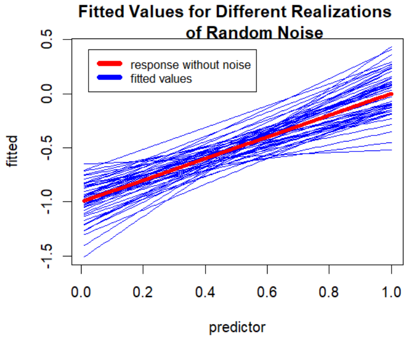
Fitted Values for Different Realizations of Random Noise

The fitted values are more volatile for *predictor* values that are further away from their mean, because those points have higher *leverage*.

The higher *leverage* of points further away from the mean of the *predictor* is due to their greater sensitivity to changes in the slope of the regression.

The fitted values for different realizations of random noise can be calculated using the influence matrix.

```
> # Calculate response without random noise for univariate regressi
> # equal to weighted sum over columns of predictor.
> betas <- c(-1, 1)
> response <- predictor %*% betas
> # Perform loop over different realizations of random noise
> fittedv <- lapply(1:50, function(it) {
+   # Add random noise to response
+   response <- response + rnorm(nrows, sd=1.0)
+   # Calculate fitted values using influence matrix
+   influencem %*% response
+ })  # end lapply
> fittedv <- rutils::do_call(cbind, fittedv)
```



```
> # Plot fitted values
> matplot(x=predictor[,2], y=fittedv,
+ type="l", lty="solid", lwd=1, col="blue",
+ xlab="predictor", ylab="fitted",
+ main="Fitted Values for Different Realizations
+ of Random Noise")
> lines(x=predictor[,2], y=response, col="red", lwd=4)
> legend(x="topleft", # Add legend
+        legend=c("response without noise", "fitted values"),
+        title=NULL, inset=0.05, cex=0.8, lwd=6,
+        lty=1, col=c("red", "blue"))
```

Predictions From *Univariate Regression* Models

The prediction y_{pred} from a regression model is equal to the *response value* corresponding to the *predictor* vector with the new data $\mathbb{X}_{new}$:

$$y_{pred} = \mathbb{X}_{new} \, \beta$$

The variance σ^2_{pred} of the *predicted value* is:

$$\sigma^2_{pred} = \frac{\mathbb{E}[\mathbb{X}_{new} \mathbb{X}_{inv} \hat{\varepsilon} \, (\mathbb{X}_{new} \mathbb{X}_{inv} \hat{\varepsilon})^T]}{d_{free}} =$$

$$\frac{\mathbb{E}[\mathbb{X}_{new} \mathbb{X}_{inv} \hat{\varepsilon} \hat{\varepsilon}^T \mathbb{X}^T_{inv} \mathbb{X}^T_{new}]}{d_{free}} = \sigma^2_{\varepsilon} \mathbb{X}_{new} \mathbb{X}_{inv} \mathbb{X}^T_{inv} \mathbb{X}^T_{new} =$$

$$\sigma^2_{\varepsilon} \, \mathbb{X}_{new} (\mathbb{X}^T \mathbb{X})^{-1} \mathbb{X}^T_{new} = \mathbb{X}_{new} \, \sigma^2_{\beta} \, \mathbb{X}^T_{new}$$

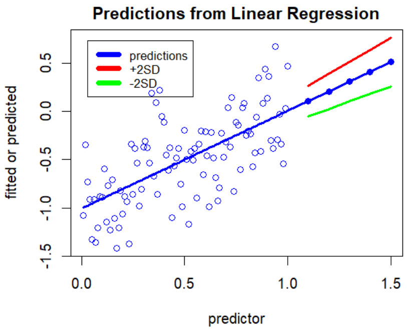
The variance σ^2_{pred} of the *predicted value* is equal to the *predictor* vector multiplied by the *covariance matrix* of the *regression coefficients* σ^2_{β}.

```
> # Inverse of predictor matrix squared
> predictor2 <- MASS::ginv(crossprod(predictor))
> # Define new predictors
> newdata <- (max(predictor[, 2]) + 10*(1:5)/nrows)
> # Calculate the predicted values and standard errors
> predictorn <- cbind(rep(1, NROW(newdata)), newdata)
> predsd <- sqrt(predictorn %*% predictor2 %*% t(predictorn))
> predictv <- cbind(
+   prediction=drop(predictorn %*% betas),
+   stddev=diag(residvar*predsd))
> # Or: Perform loop over predictorn
> predictv <- apply(predictorn, MARGIN=1, function(predictor) {
+   # Calculate predicted values
+   prediction <- predictor %*% betas
+   # Calculate standard deviation
+   predsd <- sqrt(t(predictor) %*% predictor2 %*% predictor)
+   predictsd <- residvar*predsd
+   c(prediction=prediction, stddev=predictsd)
+ })  # end sapply
> predictv <- t(predictv)
```

Confidence Intervals of Regression Predictions

The variables σ_{ε}^2 and σ_y^2 follow the *chi-squared* distribution with $d_{free} = (n - k - 1)$ degrees of freedom, so the *predicted value* y_{pred} follows the *t-distribution*.

```
> # Prepare plot data
> xdata <- c(predictor[,2], newdata)
> xlim <- range(xdata)
> ydata <- c(fittedv, predictv[, 1])
> # Calculate t-quantile
> tquant <- qt(pnorm(2), df=degf)
> predictlow <- predictv[, 1]-tquant*predictv[, 2]
> predicthigh <- predictv[, 1]+tquant*predictv[, 2]
> ylim <- range(c(response, ydata, predictlow, predicthigh))
> # Plot the regression predictions
> plot(x=xdata, y=ydata, xlim=xlim, ylim=ylim,
+      type="l", lwd=3, col="blue",
+      xlab="predictor", ylab="fitted or predicted",
+      main="Predictions from Linear Regression")
> points(x=predictor[,2], y=response, col="blue")
> points(x=newdata, y=predictv[, 1], pch=16, col="blue")
> lines(x=newdata, y=predicthigh, lwd=3, col="red")
> lines(x=newdata, y=predictlow, lwd=3, col="green")
> legend(x="topleft", # Add legend
+        legend=c("predictions", "+2SD", "-2SD"),
+        title=NULL, inset=0.05, cex=0.8, lwd=6,
+        lty=1, col=c("blue", "red", "green"))
```

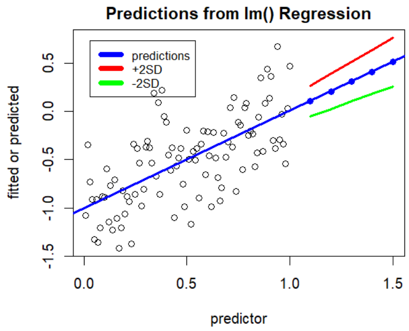


Predictions From *Linear Regression* Using Function `lm()`

The function `predict()` is a *generic function* for forecasting based on a given model.

`predict.lm()` is the predict method for linear models (regressions) produced by the function `lm()`.

```
> # Perform univariate regression
> predictor <- predictor[, 2]
> model <- lm(response ~ predictor)
> # Perform prediction from regression
> newdata <- data.frame(predictor=newdata)
> predictlm <- predict(object=model,
+   newdata=newdata, confl=1-2*(1-pnorm(2)),
+   interval="confidence")
> predictlm <- as.data.frame(predictlm)
> all.equal(predictlm$fit, predictv[, 1])
> all.equal(predictlm$lwr, predictlow)
> all.equal(predictlm$upr, predicthigh)
> plot(response ~ predictor,
+      xlim=range(predictor, newdata),
+      ylim=range(response, predictlm),
+      xlab="predictor", ylab="fitted or predicted",
+      main="Predictions from lm() Regression")
```



```
> abline(model, col="blue", lwd=3)
> with(predictlm, {
+   points(x=newdata$predictor, y=fit, pch=16, col="blue")
+   lines(x=newdata$predictor, y=lwr, lwd=3, col="green")
+   lines(x=newdata$predictor, y=upr, lwd=3, col="red")
+ })  # end with
> legend(x="topleft", # Add legend
+        legend=c("predictions", "+2SD", "-2SD"),
+        title=NULL, inset=0.05, cex=0.8, lwd=6,
+        lty=1, col=c("blue", "red", "green"))
```

Homework Assignment

Required

- Study all the lecture slides in *FRE6871_Lecture_4.pdf*, and run all the code in *FRE6871_Lecture_4.R*