FRE6871 R in Finance
Lecture#4, Spring 2025

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

April 14, 2025

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.

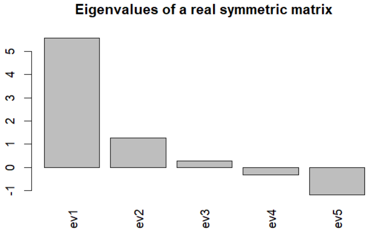
Real symmetric matrices have real *eigenvalues*, and their *eigenvectors* are orthogonal to each other.

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).

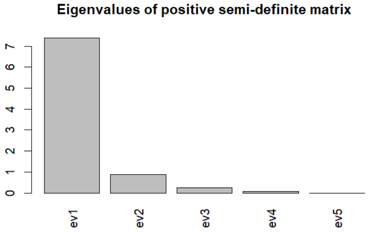
An example of *positive definite* matrices are the covariance matrices of linearly independent variables.

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:

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

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
> 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:

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

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 the left matrix
> leftmat <- matrix(runif(nrows^2), nc=nrows)
> eigend <- eigen(crossprod(leftmat))
> leftmat <- eigend$vectors[, 1:ncols]
> # Calculate the right matrix and singular values
> rightmat <- matrix(runif(ncols^2), nc=ncols)
> eigend <- eigen(crossprod(rightmat))
> rightmat <- eigend$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* (Σ) and its *eigenvectors* ($\mathbb{O}$):

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

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$vectors
> # Calculate the inverse from eigen decomposition
> inveigen <- eigenvec %*% (t(eigenvec) / eigend$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* (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: nrows > ncols
> nrows <- 6 ; ncols <- 4
> matv <- matrix(runif(nrows*ncols), nc=ncols)
> # Calculate the 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 the 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 the generalized inverse from SVD
> invsvd <- svdec$v %*% (t(svdec$u) / svdec$d)
> all.equal(invsvd, invmat)
> # Calculate the 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:

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

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 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 the 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 the regularized inverse using MASS::ginv()
> invmat <- MASS::ginv(matv)
> all.equal(invsvd, invmat)
> # Calculate the 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:

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

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 the solution using inverse of matv
> solutionv <- invmat %*% vecv
> matv %*% solutionv
> # Calculate the 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()`.

```
// 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
```

```
> # 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/lecture_slides/scripts/
> # Calculate the matrix inverse using C++
> invmat <- calc_invmat(matv)
> all.equal(invmat, invmatr)
> all.equal(invmat, MASS::ginv(matv))
> # 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)]
```

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 the 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 the 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 the correlated returns by applying Cholesky
> retscorr <- retp %*% cholmat
> # Calculate the covariance matrix
> covmat2 <- cov(retscorr)
> all.equal(covmat, covmat2)
```

Eigenvalues of Singular Covariance Matrices

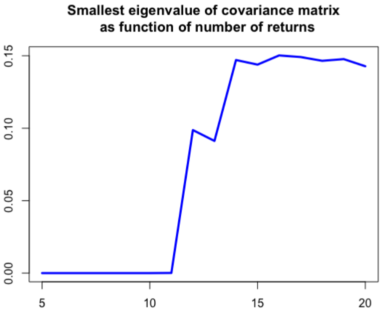
If $\mathbb{R}$ is a matrix of returns (with zero mean) for a portfolio of `k` stocks (columns), over `n` time periods (rows), then the sample covariance matrix is equal to:

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

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
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> retp <- matrix(rnorm(nassets*nrows), nc=nassets)
> # Calculate the centered (de-meaned) returns matrix
> retp <- t(t(retp) - colMeans(retp))
> # Or
> retp <- apply(retp, MARGIN=2, function(x) (x-mean(x)))
> # Calculate the 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*:

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

Where Σ_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 the covariance matrix
> covmat <- cov(matv)
> # Calculate the inverse of covmat - error
> invmat <- solve(covmat)
> # Calculate the 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 the 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 the 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, "Mersenne-Twister", sample.kind="Rejection")
> matv <- matrix(rnorm(5e2), nc=5)
> covmat <- cov(matv)
> cormat <- cor(matv)
> stdev <- sqrt(diag(covmat))
> # Calculate the 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 the shrinkage covariance matrix
> alphac <- 0.5
> covshrink <- (1-alphac)*covmat + alphac*targetmat
> # Calculate the 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*:

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

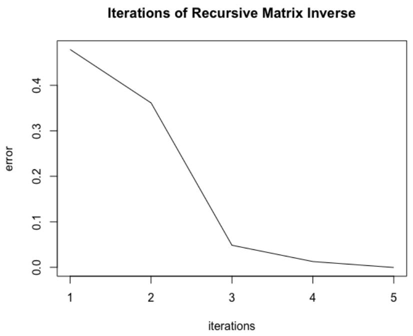
The *Schulz formula* requires a good initial value for the inverse matrix $\mathbb{A}_1^{-1}$ or else the recursion diverges.

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")
```

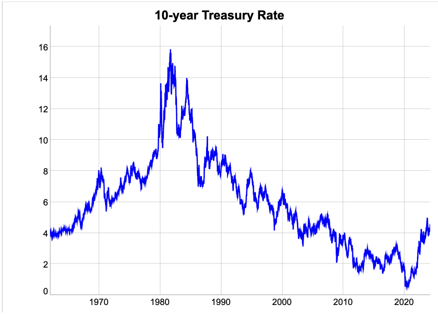
Downloading Treasury Bond Rates from *FRED*

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*.



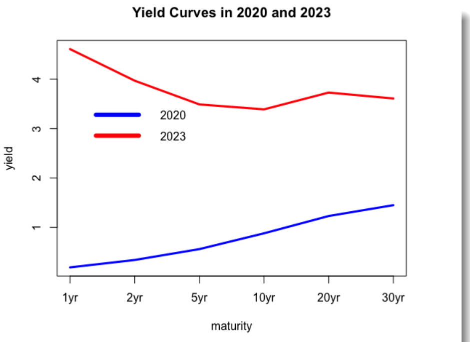

```
> # Symbols for constant maturity Treasury rates
> symbolv <- c("DGS1", "DGS2", "DGS5", "DGS10", "DGS20", "DGS30")
> # Create new environment for time series
> ratesenv <- new.env()
> # Download time series for symbolv into ratesenv
> quantmod::getSymbols(symbolv, env=ratesenv, src="FRED")
> # Remove NA values in ratesenv
> sapply(ratesenv, function(x) sum(is.na(x)))
> sapply(ls(ratesenv), function(namev) {
+   assign(x=namev, value=na.omit(get(namev, ratesenv)),
+    envir=ratesenv)
+ }) # end sapply
> sapply(ratesenv, function(x) sum(is.na(x)))
> # Get class of all objects in ratesenv
> sapply(ratesenv, class)
> # Get class of all objects in R workspace
> sapply(ls(), function(namev) class(get(namev)))
> # Save the time series environment into a binary .RData file
> save(ratesenv, file="/Users/jerzy/Develop/lecture_slides/data/rates_data.RData")
```

```
> # 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.



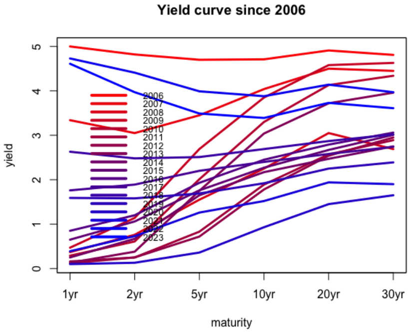

```
> # Load constant maturity Treasury rates
> load(file="/Users/jerzy/Develop/lecture_slides/data/rates_data.RDa
> # Get most recent yield curve
> ycnow <- eapply(ratesenv, xts::last)
> class(ycnow)
> ycnow <- do.call(cbind, ycnow)
> # Check if 2020-03-25 is not a holiday
> date2020 <- as.Date("2020-03-25")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

```
> # Plot using matplot()
> colorv <- c("blue", "red")
> matplot(ycurves, main="Yield Curves in 2020 and 2023", xaxt="n",
+   type="l", xlab="maturity", ylab="yield", col=colorv)
> # Add x-axis
> axis(1, seq_along(rownames(ycurves)), rownames(ycurves))
> # Add legend
> legend("topleft", legend=colnames(ycurves), y.intersp=0.1,
+  bty="n", col=colorv, 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
> datev <- xts::endpoints(ratesenv$DGS1["2006/"], on="years")
> datev <- zoo::index(ratesenv$DGS1["2006/"][datev])
> # Create time series of end-of-year rates
> ycurves <- eapply(ratesenv, function(ratev) ratev[datev])
> ycurves <- rutils::do_call(cbind, ycurves)
> # Rename columns and rows, sort columns, and transpose into matri
> colnames(ycurves) <- substr(colnames(ycurves), start=4, stop=11)
> ycurves <- ycurves[, order(as.numeric(colnames(ycurves)))]
> colnames(ycurves) <- paste0(colnames(ycurves), "yr")
> ycurves <- t(ycurves)
> colnames(ycurves) <- substr(colnames(ycurves), start=1, stop=4)
> # Plot matrix using plot.zoo()
> colorv <- colorRampPalette(c("red", "blue"))(NCOL(ycurves))
> plot.zoo(ycurves, main="Yield Curve Since 2006", lwd=3, xaxt="n"
+    plot.type="single", xlab="maturity", ylab="yield", col=colorv)
> # Add x-axis
> axis(1, seq_along(rownames(ycurves)), rownames(ycurves))
> # Add legend
> legend("topleft", legend=colnames(ycurves), y.intersp=0.1,
+  bty="n", col=colorv, lty=1, lwd=4, inset=0.05, cex=0.8)
```



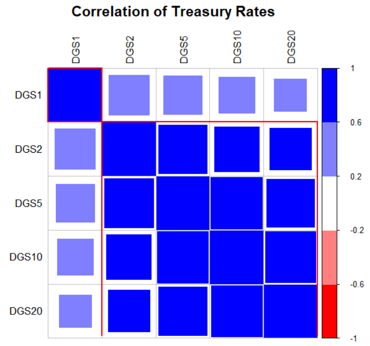

```
> # Alternative plot using matplot()
> matplot(ycurves, main="Yield curve since 2006", xaxt="n", lwd=3, l
+   type="l", xlab="maturity", ylab="yield", col=colorv)
> # Add x-axis
> axis(1, seq_along(rownames(ycurves)), rownames(ycurves))
> # Add legend
> legend("topleft", legend=colnames(ycurves), y.intersp=0.1,
+  bty="n", col=colorv, 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")
> ratem <- mget(symbolv, envir=ratesenv)
> ratem <- rutils::do_call(cbind, ratem)
> ratem <- zoo::na.locf(ratem, na.rm=FALSE)
> ratem <- zoo::na.locf(ratem, fromLast=TRUE)
> # Calculate daily percentage rates changes
> retp <- rutils::diffit(log(ratem))
> # Center (de-mean) the returns
> retp <- lapply(retp, function(x) {x - mean(x)})
> retp <- rutils::do_call(cbind, retp)
> sapply(retp, mean)
> # Covariance and Correlation matrices of Treasury rates
> covmat <- cov(retp)
> cormat <- cor(retp)
> # 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)
> colorv <- colorRampPalette(c("red", "white", "blue"))
> corrplot(cormat, title=NA, tl.col="black",
+     method="square", col=colorv(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 $\mathbf{r}_i$:

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

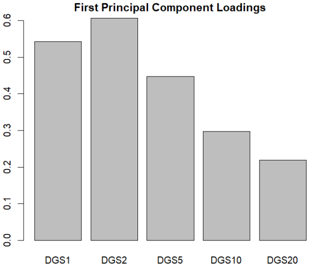
Where $\mathbf{w}_j$ is a vector of weights (loadings) of the *principal component* j, with $\mathbf{w}_j^T \mathbf{w}_j = 1$.

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




```
> # Find weights with maximum variance
> optiml <- optim(par=weightv,
+   fn=objfun,
+   retp=retp,
+   method="L-BFGS-B",
+   upper=rep(5.0, nweights),
+   lower=rep(-5.0, nweights))
> # Optimal weights and maximum variance
> weights1 <- optiml$par
> objfun(weights1, retp)
> # 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")
```

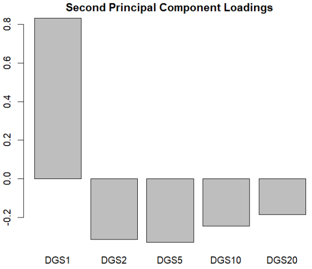
Higher Order Principal Components

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




```
> # pc2 weights and returns
> weights2 <- optiml$par
> pc2 <- drop(retp %*% 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*.

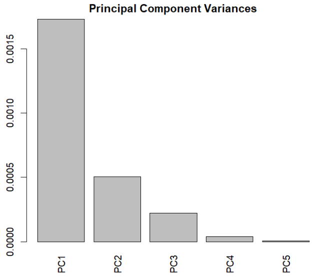
The maximum variance portfolio weights can be found by differentiating $\mathcal{L}$ with respect to $\mathbf{w}$ and setting it to zero:

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

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:

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




```r
> 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(retp, 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(retp, 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

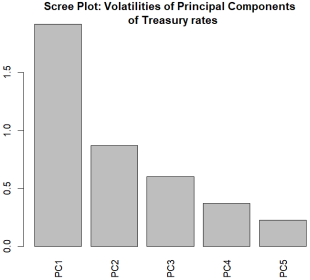
Principal Component Analysis (*PCA*) is a *dimension reduction* technique, that explains the returns of a large number of correlated time series as linear combinations of a smaller number of principal component time series.

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(retp, 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.

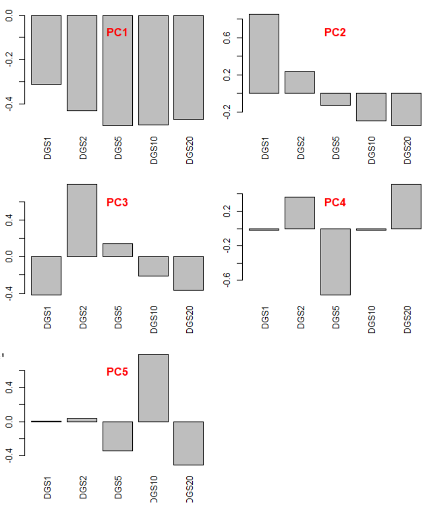
The *principal component* portfolios represent the different orthogonal modes of the data variance.

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
```



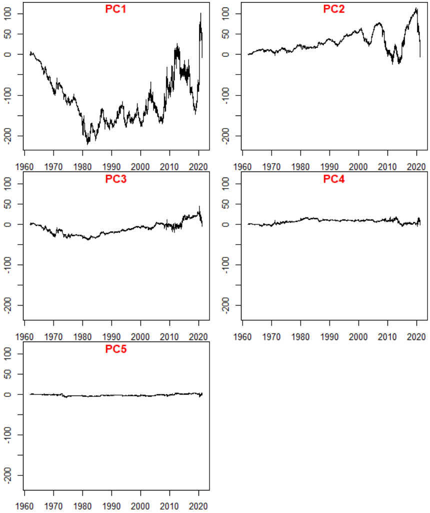
Yield Curve Principal Component Time Series

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 (center and scale) the returns
> retp <- lapply(retp, function(x) {(x - mean(x))/sd(x)})
> retp <- rutils::do_call(cbind, retp)
> sapply(retp, mean)
> sapply(retp, sd)
> # Calculate principal component time series
> retpcac <- retp %*% pcad$rotation
> all.equal(pcad$x, retpcac, check.attributes=FALSE)
> # Calculate products of principal component time series
> round(t(retpcac) %*% retpcac, 2)
> # Coerce to xts time series
> retpcac <- xts(retpcac, order.by=zoo::index(retp))
> retpcac <- cumsum(retpcac)
> # 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(retpcac)
> for (ordern in 1:NCOL(retpcac)) {
+   plot.zoo(retpcac[, 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
> retpca <- retp %*% pcad$rotation
> solved <- retpca %*% solve(pcad$rotation)
> all.equal(coredata(retp), solved)
```

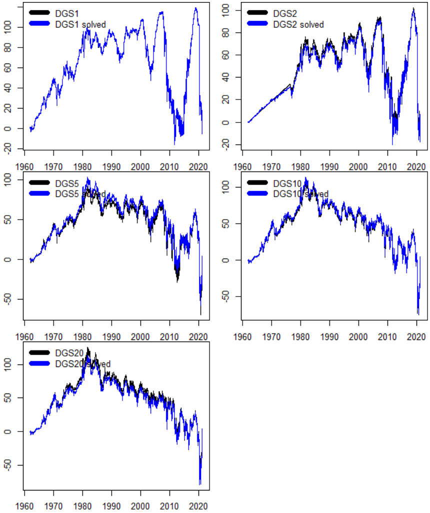
Dimension Reduction Using Principal Component Analysis

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 <- retpca[, 1:3] %*% solve(pcad$rotation)[1:3, ]
> solved <- xts::xts(solved, zoo::index(retp))
> solved <- cumsum(solved)
> retc <- cumsum(retp)
> # 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", y.intersp=0.1,
+    legend=paste0(symboln, 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
> curvep <- 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
> pricev <- 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
> datev <- seq(0, 10, 0.25)
> # Specify the evaluation (as of) date
> setEvaluationDate(as.Date("2018-01-17"))
> # Calculate the zero rates
> ratev <- DiscountCurve(params=curvep, tsQuotes=pricev, times=datev
> # Plot the zero rates
> x11()
> plot(x=ratev$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:

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

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*.

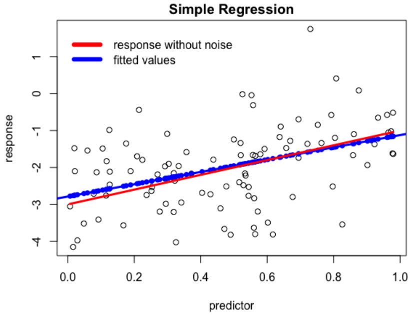
ε_i are the *residuals*, which are usually assumed to be standard normally distributed $\phi(0, \sigma_\varepsilon)$, independent, and stationary.

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 (predm) variable
> nrows <- 100
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> predm <- runif(nrows)
> noisev <- rnorm(nrows)
> # Response equals linear form plus random noise
> respv <- (-3 + 2*predm + noisev)
```

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 centered (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 the regression beta
> betac <- cov(predm, respv)/var(predm)
> # Calculate the regression alpha
> alphac <- mean(respv) - betac*mean(predm)
```

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

$class
[1] "lm"
> eval(regmod$call$formula)  # Regression formula
respv ~ predm
> regmod$coeff  # Regression coefficients
(Intercept)       predm
      -2.79        1.67
> all.equal(coef(regmod), c(alphac, betac),
+       check.attributes=FALSE)
[1] TRUE
```

The *Fitted Values* of Linear Regression

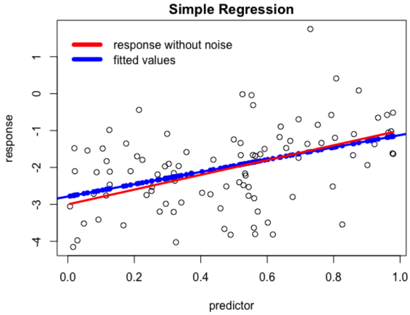
The *fitted values* y_{fit} are the estimates of the *response vector* obtained from the regression model:

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

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.

```
> fitv <- (alphac + betac*predm)
> all.equal(fitv, regmod$fitted.values, check.attributes=FALSE)
> # Plot scatterplot using formula
> plot(formulav, xlab="predictor", ylab="response")
> title(main="Simple Regression", line=0.5)
> # Add regression line
> abline(regmod, lwd=3, col="blue")
> # Plot fitted (forecast) response values
> points(x=predm, y=regmod$fitted.values, pch=16, col="blue")
```



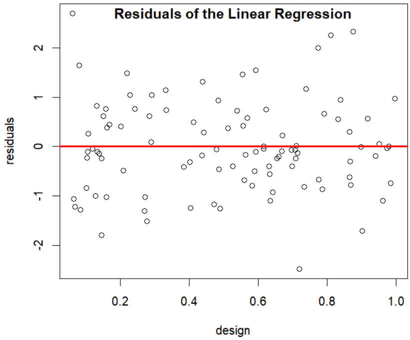

```
> # Plot response without noise
> lines(x=predm, y=(respv-noisev), col="red", lwd=3)
> legend(x="topleft", # Add legend
+        legend=c("response without noise", "fitted values"),
+        title=NULL, inset=0.0, cex=1.0, y.intersp=0.3,
+        bty="n", lwd=6, lty=1, col=c("red", "blue"))
```

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
> fitv <- (alphac + betac*predm)
> resids <- (respv - fitv)
> all.equal(resids, regmod$residuals, check.attributes=FALSE)
[1] TRUE
> # Residuals are orthogonal to the predictor
> all.equal(sum(resids*predm), target=0)
[1] TRUE
> # Residuals are orthogonal to the fitted values
> all.equal(sum(resids*fitv), target=0)
[1] TRUE
> # Sum of residuals is equal to zero
> all.equal(mean(resids), 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(predm, regmod$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:

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

```
> # Calculate the centered (de-meaned) predictor and response vecto
> predc <- predm - mean(predm)
> respc <- respv - mean(respv)
> # Degrees of freedom of residuals
> degf <- regmod$df.residual
> # Standard deviation of residuals
> residsd <- sqrt(sum(resids^2)/degf)
> # Standard error of beta
> betasd <- residsd/sqrt(sum(predc^2))
> # Standard error of alpha
> alphasd <- residsd*sqrt(1/nrows + mean(predm)^2/sum(predc^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.

```
> regsum <- summary(regmod)  # Copy regression summary
> regsum  # 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 ***
predm          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(regsum)$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.

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

Response: respv
          Df Sum Sq Mean Sq F value  Pr(>F)
predm      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
```

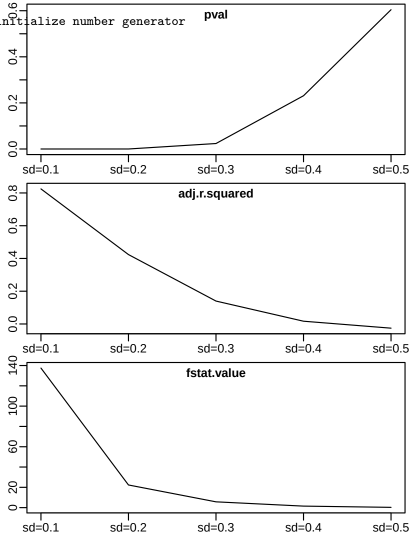
Weak Regression

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

```
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> # High noise compared to coefficient
> respv <- (-3 + 2*predm + rnorm(nrows, sd=8))
> regmod <- lm(formulav)  # Perform regression
> # Values of regression coefficients are not
> # Statistically significant
> summary(regmod)

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
predm          -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
```

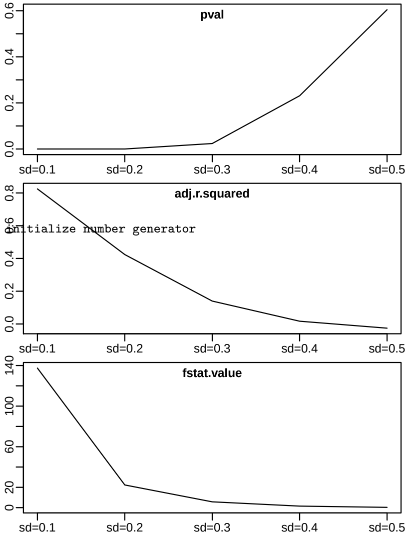
Influence of Noise on Regression

```
> regstats <- function(stdev) {  # Noisy regression
+   set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")  # initialize number generator
+ # Define explanatory (predm) and response variables
+   predm <- rnorm(100, mean=2)
+   respv <- (1 + 0.2*predm + rnorm(nrows, sd=stdev))
+ # Specify regression formula
+   formulav <- respv ~ predm
+ # Perform regression and get summary
+   regsum <- summary(lm(formulav))
+ # Extract regression statistics
+   with(regsum, c(pval=coefficients[2, 4],
+    adj_rsquared=adj.r.squared,
+    fstat=fstatistic[1]))
+ }  # end regstats
> # Apply regstats() to vector of stdev dev values
> vecsd <- seq(from=0.1, to=0.5, by=0.1)
> names(vecsd) <- paste0("sd=", vecsd)
> statsmat <- t(sapply(vecsd, regstats))
> # 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

```
> regstats <- function(datav) {  # get regression
+ # Perform regression and get summary
+   colv <- colnames(datav)
+   formulav <- paste(colv[2], colv[1], sep="~")
+   regsum <- summary(lm(formulav, data=datav))
+ # Extract regression statistics
+   with(regsum, c(pval=coefficients[2, 4],
+    adj_rsquared=adj.r.squared,
+    fstat=fstatistic[1]))
+ }  # end regstats
> # Apply regstats() to vector of stdev 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, "Mersenne-Twister", sample.kind="Rejection")  # initialize number generator
+ # Define explanatory (predm) and response variables
+     predm <- rnorm(100, mean=2)
+     respv <- (1 + 0.2*predm + rnorm(nrows, sd=stdev))
+     regstats(data.frame(predm, respv))
+     }))
> # 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 forecast responses.

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

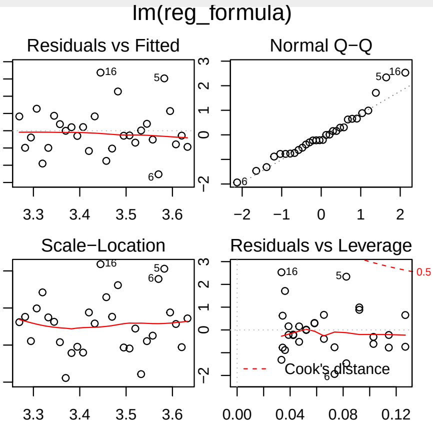
"Residuals vs Leverage" is a scatterplot of the residuals vs. their leverage.

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 forecasts made with all observations and forecasts 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,

```r
> par(mfrow=c(2, 2))  # Plot 2x2 panels
> plot(regmod)  # Plot diagnostic scatterplots
> plot(regmod, 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(regmod)

 Durbin-Watson test

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

Univariate Regression in Homogeneous Form

The *linear regression* can be written in *homogeneous form* by defining a *predictor matrix* $\mathbb{X} = (\mathbb{1}, x)$ with two columns, with the unit column representing the intercept:

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

The two *regression coefficients* are combined into a vector: $\beta = (\alpha, \beta)$.

The solution for the regression coefficients β is given by:

$$\beta = (\hat{\mathbb{X}}^T \hat{\mathbb{X}})^{-1} \hat{\mathbb{X}}^T y = \hat{\mathbb{X}}^{inv} y$$

The matrix $\hat{\mathbb{X}}^{inv} = (\hat{\mathbb{X}}^T \hat{\mathbb{X}})^{-1} \hat{\mathbb{X}}^T$ is the generalized inverse of the *predictor matrix* $\hat{\mathbb{X}}$.

```
> # Define linear regression data
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> nrows <- 100
> # Define predictor matrix
> predm <- runif(nrows)
> # Define response with noise
> noisev <- rnorm(nrows)
> respv <- (-3 + 2*predm + noisev)
> # Add unit column to predictor
> predm <- cbind(rep(1, nrows), predm)
> colnames(predm)[1] <- "intercept"
> # Solve the regression using lm()
> formulav <- respv ~ predm[, 2]
> regmod <- lm(formulav)  # Perform regression
> betalm <- regmod$coeff  # Regression coefficients
> # Solve the regression using the generalized inverse
> predinv <- MASS::ginv(predm)
> betac <- drop(predinv %*% respv)
> all.equal(betalm, betac, check.attributes=FALSE)
[1] TRUE
```

The *Influence Matrix* of Univariate Regression

The fitted values y_{fit} are equal to the response y multiplied by the *influence matrix* H:

$$y_{fit} = \mathbb{X}\beta = \mathbb{X}(\hat{\mathbb{X}}^T \hat{\mathbb{X}})^{-1} \hat{\mathbb{X}}^T y = \mathbb{H} y$$

Where $\mathbb{H} = \mathbb{X}(\hat{\mathbb{X}}^T \hat{\mathbb{X}})^{-1} \hat{\mathbb{X}}^T$ is the *influence matrix*.

The *influence matrix* projects the response vector y onto the regression line, to obtain the fitted values y_{fit}.

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

For univariate regression, 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 is due to the influence of the regression intercept α, and the second term is due to the influence of the regression slope β.

```
> # Calculate the influence matrix
> infmat <- predm %*% predinv
> # The influence matrix is idempotent
> all.equal(infmat, infmat %*% infmat)
> # Calculate the fitted values using influence matrix
> fitv <- drop(infmat %*% respv)
> all.equal(fitv, regmod$fitted.values, check.attributes=FALSE)
> # Calculate the fitted values from regression coefficients
> fitv <- drop(predm %*% betac)
> all.equal(fitv, regmod$fitted.values, check.attributes=FALSE)
```

Covariance Matrix of Fitted Values in Univariate Regression

The response values y can be considered to be *random variables* $\hat{y}$. Then the fitted values y_{fit} are also *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:

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

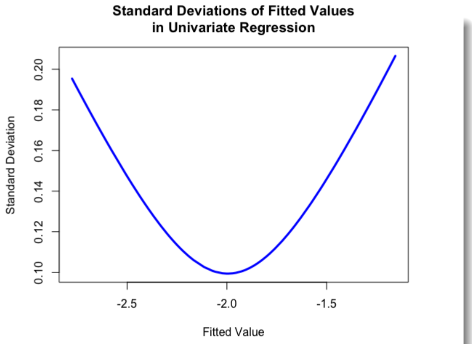
The variance of the fitted values σ^2_{fit} increases with the distance of the *predictors* from their mean values.

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

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

The leverage is the amount by which the fitted values would change if the response values were shifted by a small amount.

The response values farther away from their mean have more *leverage*, that is, more influence on the fitted values, than response values close to the mean.




```
> # Calculate the covariance and standard deviations of fitted value
> resids <- drop(respv - fitv)
> degf <- (NROW(predm) - NCOL(predm))
> residsd <- sqrt(sum(resids^2)/degf)
> fitcovar <- residsd*infmat
> fitsd <- sqrt(diag(fitcovar))
> # Plot the standard deviations
> fitdata <- cbind(fitted=fitv, stdev=fitsd)
> fitdata <- fitdata[order(fitv), ]
> plot(fitdata, type="l", lwd=3, col="blue",
+      xlab="Fitted Value", ylab="Standard Deviation",
+      main="Standard Deviations of Fitted Values\nin Univariate Reg
```

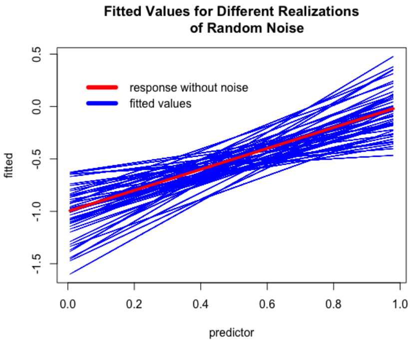
Fitted Values for Different Realizations of Random Noise

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




```
> # Plot fitted values
> matplot(x=predm[, 2], y=fitm,
+ type="l", lty="solid", lwd=1, col="blue",
+ xlab="predictor", ylab="fitted",
+ main="Fitted Values for Different Realizations
+ of Random Noise")
> lines(x=predm[, 2], y=respn, col="red", lwd=4)
> legend(x="topleft", # Add legend
+        legend=c("response without noise", "fitted values"),
+        title=NULL, inset=0.05, cex=1.0, lwd=6, y.intersp=0.4,
+        bty="n", lty=1, col=c("red", "blue"))
```

Forecasts From *Univariate Regression* Models

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

$$y_f = \mathbb{X}_{new} \, \beta$$

The variance σ_f^2 of the *forecast value* is equal to the *predictor* vector multiplied by the *covariance matrix* of the *regression coefficients* σ_β^2:

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

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

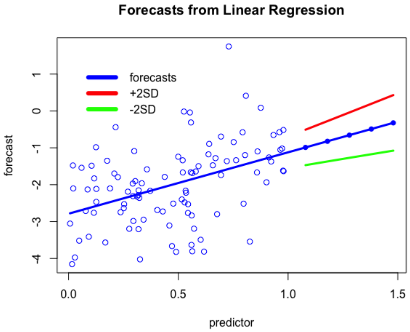
$$\sigma_\varepsilon^2 \, \mathbb{X}_{new} (\mathbb{X}^T \mathbb{X})^{-1} \mathbb{X}_{new}^T = \mathbb{X}_{new} \, \sigma_\beta^2 \, \mathbb{X}_{new}^T$$

```
> # Define new predictor
> newdata <- (max(predm[, 2]) + 10*(1:5)/nrows)
> predn <- cbind(rep(1, NROW(newdata)), newdata)
> # Calculate the forecast values
> fcast <- drop(predn %*% betac)
> # Calculate the inverse of the squared predictor matrix
> pred2 <- MASS::ginv(crossprod(predm))
> # Calculate the standard errors
> predsd <- residsd*sqrt(predn %*% pred2 %*% t(predn))
> # Combine the forecast values and standard errors
> fcast <- cbind(forecast=fcast, stdev=diag(predsd))
```

Confidence Intervals of Regression Forecasts

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 *forecast value* y_f follows the *t-distribution*.

```
> # Prepare plot data
> xdata <- c(predm[, 2], newdata)
> ydata <- c(fitv, fcast[, 1])
> # Calculate the t-quantile
> tquant <- qt(pnorm(2), df=degf)
> fcastl <- fcast[, 1] - tquant*fcast[, 2]
> fcasth <- fcast[, 1] + tquant*fcast[, 2]
> # Plot the regression forecasts
> xlim <- range(xdata)
> ylim <- range(c(respv, ydata, fcastl, fcasth))
> plot(x=xdata, y=ydata, xlim=xlim, ylim=ylim,
+      type="l", lwd=3, col="blue",
+      xlab="predictor", ylab="forecast",
+      main="Forecasts from Linear Regression")
> points(x=predm[, 2], y=respv, col="blue")
> points(x=newdata, y=fcast[, 1], pch=16, col="blue")
> lines(x=newdata, y=fcasth, lwd=3, col="red")
> lines(x=newdata, y=fcastl, lwd=3, col="green")
> legend(x="topleft", # Add legend
+        legend=c("forecasts", "+2SD", "-2SD"),
+        title=NULL, inset=0.05, cex=1.0, lwd=6, y.intersp=0.4,
+        bty="n", lty=1, col=c("blue", "red", "green"))
```

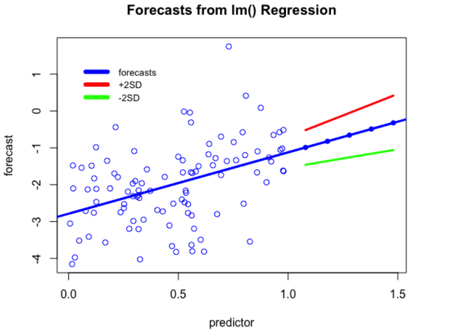


Forecasts of *Linear Regression* Using `predict.lm()`

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

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

```
> # Perform univariate regression
> dframe <- data.frame(resp=respv, pred=predm[, 2])
> regmod <- lm(resp ~ pred, data=dframe)
> # Calculate the forecasts from regression
> newdf <- data.frame(pred=predn[, 2]) # Same column name
> fcastlm <- predict.lm(object=regmod,
+   newdata=newdf, confl=1-2*(1-pnorm(2)),
+   interval="confidence")
> rownames(fcastlm) <- NULL
> all.equal(fcastlm[, "fit"], fcast[, 1])
> all.equal(fcastlm[, "lwr"], fcastl)
> all.equal(fcastlm[, "upr"], fcasth)
> plot(x=xdata, y=ydata, xlim=xlim, ylim=ylim,
+      type="l", lwd=3, col="blue",
+      xlab="predictor", ylab="forecast",
+      main="Forecasts from lm() Regression")
> points(x=predm[, 2], y=respv, col="blue")
```




```
> abline(regmod, col="blue", lwd=3)
> points(x=newdata, y=fcastlm[, "fit"], pch=16, col="blue")
> lines(x=newdata, y=fcastlm[, "lwr"], lwd=3, col="green")
> lines(x=newdata, y=fcastlm[, "upr"], lwd=3, col="red")
> legend(x="topleft", # Add legend
+        legend=c("forecasts", "+2SD", "-2SD"),
+        title=NULL, inset=0.05, cex=0.8, lwd=6, y.intersp=0.4,
+        bty="n", lty=1, col=c("blue", "red", "green"))
```

Spurious Time Series Regression

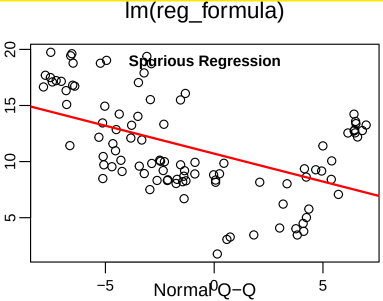
Regression of non-stationary time series creates *spurious* regressions.

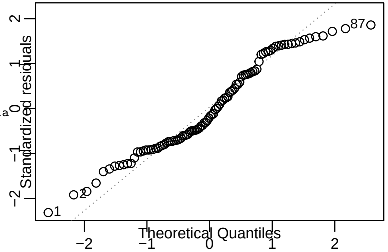
The *t*-statistics, *p*-values, and *R*-squared all indicate a statistically significant regression.

But the Durbin-Watson test shows residuals are autocorrelated, which invalidates the other tests.

The Q-Q plot also shows that residuals are *not* normally distributed.

```
> predm <- cumsum(rnorm(100))  # Unit root time series
> respv <- cumsum(rnorm(100))
> formulav <- respv ~ predm
> regmod <- lm(formulav)  # Perform regression
> # Summary indicates statistically significant regression
> regsum <- summary(regmod)
> regsum$coeff
> regsum$r.squared
> # Durbin-Watson test shows residuals are autocorrelated
> dwtest <- lmtest::dwtest(regmod)
> c(dwtest$statistic[[1]], dwtest$p.value)

> plot(formulav, xlab="", ylab="")  # Plot scatterplot using formula
> title(main="Spurious Regression", line=-1)
> # Add regression line
> abline(regmod, lwd=2, col="red")
> plot(regmod, which=2, ask=FALSE)  # Plot just Q-Q
```





Multivariate Linear Regression

A *multivariate* linear regression model with k *predictors* x_j, is defined by the formula:

$$y_i = \alpha + \sum_{j=1}^{k} \beta_j x_{i,j} + \varepsilon_i$$

α and β are the unknown regression coefficients, with α a scalar and β a vector of length k.

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

The data consists of n observations, with each observation containing k *predictors* and one *response* value.

The *response vector* y, the *predictor* vectors x_j, and the *residuals* ε are vectors of length n.

The k *predictors* x_j form the columns of the (n, k)-dimensional *predictor matrix* $\mathbb{X}$.

The *multivariate regression* model can be written in vector notation as:

$$y = \alpha + \mathbb{X}\beta + \varepsilon = y_{fit} + \varepsilon$$
$$y_{fit} = \alpha + \mathbb{X}\beta$$

Where y_{fit} are the fitted values of the model.

```
> # Define predictor matrix
> nrows <- 100
> ncols <- 5
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> predm <- matrix(runif(nrows*ncols), ncol=ncols)
> # Add column names
> colnames(predm) <- paste0("pred", 1:ncols)
> # Define the predictor weights
> weightv <- runif(3:(ncols+2), min=(-1), max=1)
> # Response equals weighted predictor plus random noise
> noisev <- rnorm(nrows, sd=2)
> respv <- (1 + predm %*% weightv + noisev)
```

Solution of Multivariate Regression

The *Residual Sum of Squares* (*RSS*) is defined as the sum of the squared *residuals*:

$$RSS = \varepsilon^T \varepsilon = (y - y_{fit})^T (y - y_{fit}) =$$
$$(y - \alpha + \mathbb{X}\beta)^T (y - \alpha + \mathbb{X}\beta)$$

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

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

The solutions for α and β are given by:

$$\alpha = \bar{y} - \bar{\mathbb{X}}\beta$$
$$RSS_\beta = -2(\hat{y} - \hat{\mathbb{X}}\beta)^T \hat{\mathbb{X}} = 0$$
$$\hat{\mathbb{X}}^T \hat{y} - \hat{\mathbb{X}}^T \hat{\mathbb{X}}\beta = 0$$
$$\beta = (\hat{\mathbb{X}}^T \hat{\mathbb{X}})^{-1} \hat{\mathbb{X}}^T \hat{y} = \hat{\mathbb{X}}^{inv} \hat{y}$$

Where $\bar{y}$ and $\bar{\mathbb{X}}$ are the column means, and $\hat{\mathbb{X}} = \mathbb{X} - \bar{\mathbb{X}}$ and $\hat{y} = y - \bar{y} = \hat{\mathbb{X}}\beta + \varepsilon$ are the centered (de-meaned) variables.

The matrix $\hat{\mathbb{X}}^{inv}$ is the generalized inverse of the centered (de-meaned) *predictor matrix* $\hat{\mathbb{X}}$.

The matrix $\mathbb{C} = \hat{\mathbb{X}}^T \hat{\mathbb{X}} / (n-1)$ is the *covariance matrix* of the matrix $\mathbb{X}$, and it's invertible only if the columns of $\mathbb{X}$ are linearly independent.

```
> # Perform multivariate regression using lm()
> regmod <- lm(respv ~ predm)
> # Solve multivariate regression using matrix algebra
> # Calculate the centered (de-meaned) predictor matrix and response
> # predc <- t(t(predm) - colMeans(predm))
> predc <- apply(predm, 2, function(x) (x-mean(x)))
> respc <- respv - mean(respv)
> # Calculate the regression coefficients
> betac <- drop(MASS::ginv(predc) %*% respc)
> # Calculate the regression alpha
> alphac <- mean(respv) - sum(colSums(predm)*betac)/nrows
> # Compare with coefficients from lm()
> all.equal(coef(regmod), c(alphac, betac), check.attributes=FALSE)
[1] TRUE
> # Compare with actual coefficients
> all.equal(c(1, weightv), c(alphac, betac), check.attributes=FALSE)
[1] "Mean relative difference: 0.963"
```

Multivariate Regression in Homogeneous Form

If an extra unit column is added to the *predictor matrix* $\mathbb{X} = (\mathbb{1}, \mathbb{X})$ for the intercept term, then the *linear regression* can be written in *homogeneous form*:

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

Where the *regression coefficients* β now contain the intercept α: $\beta = (\alpha, \beta_1, \ldots, \beta_k)$, and the *predictor matrix* $\mathbb{X}$ has $k + 1$ columns and n rows.

The *OLS* solution for the β coefficients is found by equating the *RSS* derivative to zero:

$$RSS_\beta = -2(y - \mathbb{X}\beta)^T\mathbb{X} = 0$$
$$\mathbb{X}^T y - \mathbb{X}^T\mathbb{X}\beta = 0$$
$$\beta = (\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T y = \mathbb{X}_{inv} y$$

The matrix $\mathbb{X}_{inv} = (\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T$ is the generalized inverse of the *predictor matrix* $\mathbb{X}$.

The coefficients β can be interpreted as the projections of the *response vector* y onto the columns of the *predictor matrix* $\mathbb{X}$.

The *predictor matrix* $\mathbb{X}$ maps the *regression coefficients* β into the *response vector* y.

The generalized inverse of the *predictor matrix* $\mathbb{X}_{inv}$ maps the *response vector* y into the *regression coefficients* β.

```
> # Add intercept column to predictor matrix
> predm <- cbind(rep(1, nrows), predm)
> ncols <- NCOL(predm)
> # Add column name
> colnames(predm)[1] <- "intercept"
> # Calculate the generalized inverse of the predictor matrix
> predinv <- MASS::ginv(predm)
> # Calculate the regression coefficients
> betac <- predinv %*% respv
> # Perform multivariate regression without intercept term
> regmod <- lm(respv ~ predm - 1)
> all.equal(drop(betac), coef(regmod), check.attributes=FALSE)
[1] TRUE
```

The *Residuals* of Multivariate Regression

The *multivariate regression* model can be written in vector notation as:

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

Where y_{fit} are the fitted values of the model.

The *residuals* are equal to the *response vector* minus the fitted values: $\varepsilon = y - y_{fit}$.

The *residuals* ε are orthogonal to the columns of the *predictor matrix* $\mathbb{X}$ (the *predictors*):

$$\varepsilon^T \mathbb{X} = (y - \mathbb{X}(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T y)^T \mathbb{X} =$$
$$y^T\mathbb{X} - y^T\mathbb{X}(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T\mathbb{X} = y^T\mathbb{X} - y^T\mathbb{X} = 0$$

Therefore the *residuals* are also orthogonal to the fitted values: $\varepsilon^T y_{fit} = \varepsilon^T \mathbb{X}\beta = 0$.

Since the first column of the *predictor matrix* $\mathbb{X}$ is a unit vector, the *residuals* ε have zero mean: $\varepsilon^T \mathbb{1} = 0$.

```
> # Calculate the fitted values from regression coefficients
> fitv <- drop(predm %*% betac)
> all.equal(fitv, regmod$fitted.values, check.attributes=FALSE)
[1] TRUE
> # Calculate the residuals
> resids <- drop(respv - fitv)
> all.equal(resids, regmod$residuals, check.attributes=FALSE)
[1] TRUE
> # Residuals are orthogonal to predictor columns (predms)
> sapply(resids %*% predm, all.equal, target=0)
[1] TRUE TRUE TRUE TRUE TRUE TRUE
> # Residuals are orthogonal to the fitted values
> all.equal(sum(resids*fitv), target=0)
[1] TRUE
> # Sum of residuals is equal to zero
> all.equal(sum(resids), target=0)
[1] TRUE
```

The *Influence Matrix* of Multivariate Regression

The vector $y_{fit} = \mathbb{X}\beta$ are the fitted values corresponding to the *response vector* y:

$$y_{fit} = \mathbb{X}\beta = \mathbb{X}(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T y = \mathbb{X}\mathbb{X}_{inv} y = \mathbb{H} y$$

Where $\mathbb{H} = \mathbb{X}\mathbb{X}_{inv} = \mathbb{X}(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T$ is the *influence matrix* (or hat matrix), which maps the *response vector* y into the fitted values y_{fit}.

The *influence matrix* $\mathbb{H}$ is a projection matrix, and it measures the changes in the fitted values y_{fit} due to changes in the *response vector* y.

$$\mathbb{H}_{ij} = \frac{\partial y_i^{fit}}{\partial y_j}$$

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

```
> # Calculate the influence matrix
> infmat <- predm %*% predinv
> # The influence matrix is idempotent
> all.equal(infmat, infmat %*% infmat)
[1] TRUE
> # Calculate the fitted values using influence matrix
> fitv <- drop(infmat %*% respv)
> all.equal(fitv, regmod$fitted.values, check.attributes=FALSE)
[1] TRUE
> # Calculate the fitted values from regression coefficients
> fitv <- drop(predm %*% betac)
> all.equal(fitv, regmod$fitted.values, check.attributes=FALSE)
[1] TRUE
```

Multivariate Regression With Centered Variables

The *multivariate regression* model can be written in vector notation as:

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

The intercept α can be substituted with its solution: $\alpha = \bar{y} - \bar{\mathbb{X}}\beta$ to obtain the regression model with centered (de-meaned) response and predictor matrix:

$$y = \bar{y} - \bar{\mathbb{X}}\beta + \mathbb{X}\beta$$

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

The regression model with a centered (de-meaned) *predictor matrix* produces the same fitted values (only shifted by their mean) and *residuals* as the original regression model, so it's equivalent to it.

But the centered regression model has a different *influence matrix*, which maps the centered *response vector* $\hat{y}$ into the centered fitted values $\hat{y}_{fit}$.

```
> # Calculate the centered (de-meaned) fitted values
> predc <- t(t(predm) - colMeans(predm))
> fittedc <- drop(predc %*% betac)
> all.equal(fittedc, regmod$fitted.values - mean(respv),
+   check.attributes=FALSE)
[1] TRUE
> # Calculate the residuals
> respc <- respv - mean(respv)
> resids <- drop(respc - fittedc)
> all.equal(resids, regmod$residuals, check.attributes=FALSE)
[1] TRUE
> # Calculate the influence matrix
> infmatc <- predc %*% MASS::ginv(predc)
> # Compare the fitted values
> all.equal(fittedc, drop(infmatc %*% respc), check.attributes=FALSE
[1] TRUE
```

Multivariate Regression for Orthogonal Predictors

The generalized inverse can be written as:

$$\mathbb{X}_{inv} = (\mathbb{X}^T \mathbb{X})^{-1} \mathbb{X}^T = \mathbb{C}^{-1} \mathbb{X}^T$$

Where $\mathbb{C} = \mathbb{X}^T \mathbb{X}$ is the matrix of inner products of the predictors $\mathbb{X}$.

If the predictors are orthogonal ($x_i \cdot x_j = 0$ for $i \neq j$, and $x_i \cdot x_i = \sigma_i^2$) then the squared predictor matrix $\mathbb{C}$ is diagonal:

$$\mathbb{C} = \begin{pmatrix} \sigma_1^2 & 0 & \cdots & 0 \\ 0 & \sigma_2^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_n^2 \end{pmatrix}$$

And the inverse of the squared predictor matrix $\mathbb{C}^{-1}$ is also diagonal, so the *regression coefficients* can then be written simply as:

$$\beta_i = \frac{x_i \cdot y}{\sigma_i^2}$$

Where $x_i \cdot y$ are the inner products of the predictors x_i times the *response vector* y.

Conversely, if the predictors are *collinear* then their squared predictor matrix is *singular* and the regression is also singular. Predictors are *collinear* if there's a linear combination that is constant.

```
> # Perform PCA of the predictors
> pcad <- prcomp(predm, center=FALSE, scale=FALSE)
> # Calculate the PCA predictors
> predpca <- predm %*% pcad$rotation
> # Principal components are orthogonal to each other
> round(t(predpca) %*% predpca, 2)
> # Calculate the PCA regression coefficients using lm()
> regmod <- lm(respv ~ predpca - 1)
> summary(regmod)
> regmod$coefficients
> # Calculate the PCA regression coefficients directly
> colSums(predpca*drop(respv))/colSums(predpca^2)
> # Create almost collinear predictors
> predcol <- predm
> predcol[, 1] <- (predcol[, 1]/1e3 + predcol[, 2])
> # Calculate the PCA predictors
> pcad <- prcomp(predcol, center=FALSE, scale=FALSE)
> predpca <- predcol %*% pcad$rotation
> round(t(predpca) %*% predpca, 6)
> # Calculate the PCA regression coefficients
> drop(MASS::ginv(predpca) %*% respv)
> # Calculate the PCA regression coefficients directly
> colSums(predpca*drop(respv))/colSums(predpca^2)
```

Regression Coefficients as *Random Variables*

The *residuals* $\hat{\varepsilon}$ can be considered to be *random variables*, with expected value equal to zero $\mathbb{E}[\hat{\varepsilon}] = 0$, and variance equal to σ_ε^2.

The variance of the *residuals* is equal to the expected value of the squared *residuals* divided by the number of *degrees of freedom*:

$$\sigma_\varepsilon^2 = \frac{\mathbb{E}[\varepsilon^T \varepsilon]}{d_{free}}$$

Where $d_{free} = (n - k)$ is the number of *degrees of freedom* of the *residuals*, equal to the number of observations n, minus the number of *predictors* k (including the intercept term).

The *response vector* y can also be considered to be a *random variable* $\hat{y}$, equal to the sum of the deterministic fitted values y_{fit} plus the random *residuals* $\hat{\varepsilon}$:

$$\hat{y} = \mathbb{X}\beta + \hat{\varepsilon} = y_{fit} + \hat{\varepsilon}$$

The *regression coefficients* β can also be considered to be *random variables* $\hat{\beta}$:

$$\hat{\beta} = \mathbb{X}_{inv}\hat{y} = \mathbb{X}_{inv}(y_{fit} + \hat{\varepsilon}) =$$

$$(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T(\mathbb{X}\beta + \hat{\varepsilon}) = \beta + \mathbb{X}_{inv}\hat{\varepsilon}$$

Where β is equal to the expected value of $\hat{\beta}$:
$\beta = \mathbb{E}[\hat{\beta}] = \mathbb{X}_{inv}y_{fit} = \mathbb{X}_{inv}y$.

```
> # Regression model summary
> regsum <- summary(regmod)
> # Degrees of freedom of residuals
> nrows <- NROW(predm)
> ncols <- NCOL(predm)
> degf <- (nrows - ncols)
> all.equal(degf, regsum$df[2])
[1] TRUE
> # Variance of residuals
> residsd <- sum(resids^2)/degf
```

Covariance Matrix of the Regression Coefficients

The *covariance matrix* of the *regression coefficients* $\hat{\beta}$ is given by:

$$\sigma_{\beta}^2 = \frac{\mathbb{E}[(\hat{\beta} - \beta)(\hat{\beta} - \beta)^T]}{d_{free}} =$$

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

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

$$(\mathbb{X}^T \mathbb{X})^{-1} \mathbb{X}^T \sigma_{\varepsilon}^2 \mathbb{1} \mathbb{X} (\mathbb{X}^T \mathbb{X})^{-1} = \sigma_{\varepsilon}^2 (\mathbb{X}^T \mathbb{X})^{-1}$$

Where the expected values of the squared residuals are proportional to the diagonal unit matrix $\mathbb{1}$:

$\frac{\mathbb{E}[\hat{\varepsilon} \hat{\varepsilon}^T]}{d_{free}} = \sigma_{\varepsilon}^2 \mathbb{1}$

If the predictors are close to being *collinear*, then the squared predictor matrix becomes singular, and the covariance of their regression coefficients becomes very large.

The matrix $\mathbb{X}_{inv} = (\mathbb{X}^T \mathbb{X})^{-1} \mathbb{X}^T$ is the generalized inverse of the *predictor matrix* $\mathbb{X}$.

```
> # Inverse of the squared predictor matrix
> pred2 <- MASS::ginv(crossprod(predm))
> # pred2 <- t(predm) %*% predm
> # Variance of residuals
> residsd <- sum(resids^2)/degf
> # Calculate the covariance matrix of betas
> covm <- residsd*pred2
> # round(covm, 3)
> betasd <- sqrt(diag(covm))
> all.equal(betasd, regsum$coeff[, 2], check.attributes=FALSE)
[1] TRUE
> # Calculate the t-values of betas
> betatvals <- drop(betac)/betasd
> all.equal(betatvals, regsum$coeff[, 3], check.attributes=FALSE)
[1] TRUE
> # Calculate the two-sided p-values of betas
> betapvals <- 2*pt(-abs(betatvals), df=degf)
> all.equal(betapvals, regsum$coeff[, 4], check.attributes=FALSE)
[1] TRUE
> # The square of the generalized inverse is equal
> # to the inverse of the square
> all.equal(MASS::ginv(crossprod(predm)), predinv %*% t(predinv))
[1] TRUE
```

Covariance Matrix of the Fitted Values

The fitted values y_{fit} can also be considered to be *random variables* $\hat{y}_{fit}$, because the *regression coefficients* $\hat{\beta}$ are *random variables*:
$\hat{y}_{fit} = \mathbb{X}\hat{\beta} = \mathbb{X}(\beta + \mathbb{X}_{inv}\hat{\varepsilon}) = y_{fit} + \mathbb{X}\mathbb{X}_{inv}\hat{\varepsilon}$.

The *covariance matrix* of the fitted values σ^2_{fit} is:

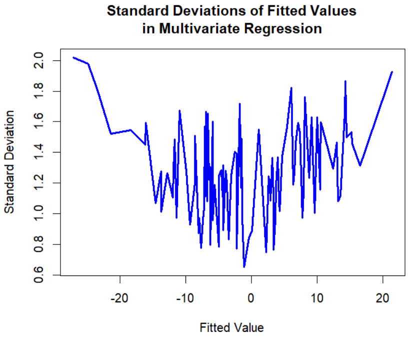
$$\sigma^2_{fit} = \frac{\mathbb{E}[\mathbb{X}\mathbb{X}_{inv}\hat{\varepsilon}\,(\mathbb{X}\mathbb{X}_{inv}\hat{\varepsilon})^T]}{d_{free}} = \frac{\mathbb{E}[\mathbb{H}\,\hat{\varepsilon}\hat{\varepsilon}^T\,\mathbb{H}^T]}{d_{free}} =$$
$$\frac{\mathbb{H}\,\mathbb{E}[\hat{\varepsilon}\hat{\varepsilon}^T]\,\mathbb{H}^T}{d_{free}}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

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
> infmat <- predm %*% predinv
> # The influence matrix is idempotent
> all.equal(infmat, infmat %*% infmat)
```




```
> # Calculate the covariance and standard deviations of fitted value
> fitcovar <- residsd*infmat
> fitsd <- sqrt(diag(fitcovar))
> # Sort the standard deviations
> fitsd <- cbind(fitted=fitv, stdev=fitsd)
> fitsd <- fitsd[order(fitv), ]
> # Plot the standard deviations
> plot(fitsd, type="l", lwd=3, col="blue",
+      xlab="Fitted Value", ylab="Standard Deviation",
+      main="Standard Deviations of Fitted Values\nin Multivariate R
```

Standard Errors of Time Series Regression

Bootstrapping the regression of stock returns shows that the actual standard errors can be much larger as the theoretical standard errors reported by the function `lm()`.

This is because the function `lm()` assumes that the data is normally distributed, while in reality stock returns have very large skewness and kurtosis.

```
> # Load time series of ETF percentage returns
> retp <- rutils::etfenv$returns[, c("XLF", "XLE")]
> retp <- na.omit(retp)
> nrows <- NROW(retp)
> head(retp)
> # Define regression formula
> formulav <- paste(colnames(retp)[1],
+   paste(colnames(retp)[-1], collapse="+"),
+   sep=" ~ ")
> # Standard regression
> regmod <- lm(formulav, data=retp)
> regsum <- summary(regmod)
> cdata <- coredata(retp)
> plot(cdata[, 1], cdata[, 2],
+   xlab="XLE", ylab="XLF", main="Stock Returns")
> abline(regmod, lwd=3, col="red")
> # Bootstrap of regression
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> bootd <- sapply(1:100, function(x) {
+   samplev <- sample.int(nrows, replace=TRUE)
+   regmod <- lm(formulav, data=retp[samplev, ])
+   regmod$coefficients
+ })  # end sapply
> # Means and standard errors from regression
> regsum$coefficients
> # Means and standard errors from bootstrap
> t(apply(bootd, MARGIN=1, function(x)
+   c(mean=mean(x), stderror=sd(x))))
```

Forecasts From Multivariate Regression Models

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

$$y_f = \mathbb{X}_{new} \, \beta$$

The forecast is a *random variable* $\hat{y}_f$, because the *regression coefficients* $\hat{\beta}$ are *random variables*:

$$\hat{y}_f = \mathbb{X}_{new} \hat{\beta} = \mathbb{X}_{new} (\beta + \mathbb{X}_{inv} \hat{\varepsilon}) = y_f + \mathbb{X}_{new} \mathbb{X}_{inv} \hat{\varepsilon}$$

The variance σ_f^2 of the *forecast value* is:

$$\sigma_f^2 = \frac{\mathbb{E}[\mathbb{X}_{new} \mathbb{X}_{inv} \hat{\varepsilon} \, (\mathbb{X}_{new} \mathbb{X}_{inv} \hat{\varepsilon})^T]}{d_{free}} = \frac{\mathbb{E}[\mathbb{X}_{new} \mathbb{X}_{inv} \hat{\varepsilon} \hat{\varepsilon}^T \mathbb{X}_{inv}^T \mathbb{X}_{new}^T]}{d_{free}} = \sigma_\varepsilon^2 \mathbb{X}_{new} \mathbb{X}_{inv} \mathbb{X}_{inv}^T \mathbb{X}_{new}^T = \sigma_\varepsilon^2 \, \mathbb{X}_{new} (\mathbb{X}^T \mathbb{X})^{-1} \mathbb{X}_{new}^T = \mathbb{X}_{new} \, \sigma_\beta^2 \, \mathbb{X}_{new}^T$$

The variance σ_f^2 of the *forecast value* is equal to the *predictor* vector multiplied by the *covariance matrix* of the *regression coefficients* σ_β^2.

```
> # New data predictor is a data frame or row vector
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> newdata <- data.frame(matrix(c(1, rnorm(5)), nr=1))
> colv <- colnames(predm)
> colnames(newdata) <- colv
> newdata <- as.matrix(newdata)
> fcast <- drop(newdata %*% betac)
> predsd <- drop(sqrt(newdata %*% covm %*% t(newdata)))
```

Forecasts From Multivariate Regression Using lm()

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

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

In order for predict.lm() to work properly, the multivariate regression must be specified using a formula.

```
> # Create formula from text string
> formulav <- paste0("respv ~ ",
+   paste(colnames(predm), collapse=" + "), " - 1")
> # Specify multivariate regression using formula
> regmod <- lm(formulav, data=data.frame(cbind(respv, predm)))
> regsum <- summary(regmod)
> # Predict from lm object
> fcastlm <- predict.lm(object=model, newdata=newdata,
+    interval="confidence", confl=1-2*(1-pnorm(2)))
> # Calculate the t-quantile
> tquant <- qt(pnorm(2), df=degf)
> fcasth <- (fcast + tquant*predsd)
> fcastl <- (fcast - tquant*predsd)
> # Compare with matrix calculations
> all.equal(fcastlm[1, "fit"], fcast)
> all.equal(fcastlm[1, "lwr"], fcastl)
> all.equal(fcastlm[1, "upr"], fcasth)
```

Total Sum of Squares and *Explained Sum of Squares*

The *Total Sum of Squares* (*TSS*), the *Explained Sum of Squares* (*ESS*), and the *Residual Sum of Squares* (*RSS*) are defined as:

$$TSS = (y - \bar{y})^T (y - \bar{y})$$
$$ESS = (y_{fit} - \bar{y})^T (y_{fit} - \bar{y})$$
$$RSS = (y - y_{fit})^T (y - y_{fit})$$

Since the *residuals* $\varepsilon = y - y_{fit}$ are orthogonal to the fitted values y_{fit}, they are also orthogonal to the *fitted* excess values $(y_{fit} - \bar{y})$:

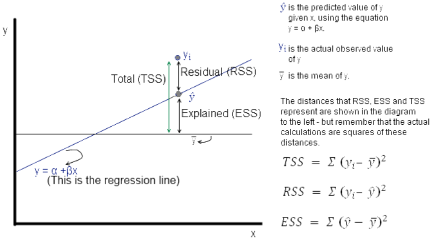
$$(y - y_{fit})^T (y_{fit} - \bar{y}) = 0$$

Therefore the *TSS* can be expressed as the sum of the *ESS* plus the *RSS*:

$$TSS = ESS + RSS$$

It also follows that the *RSS* and the *ESS* follow independent *chi-squared* distributions with $(n - k)$ and $(k - 1)$ degrees of freedom.

The degrees of freedom of the *Total Sum of Squares* is equal to the sum of the *RSS* plus the *ESS*:
$d^{TSS}_{free} = (n - k) + (k - 1) = n - 1$.




```
> # TSS = ESS + RSS
> tss <- sum((respv-mean(respv))^2)
> ess <- sum((fitv-mean(fitv))^2)
> rss <- sum(resids^2)
> all.equal(tss, ess + rss)
[1] TRUE
```

R-squared of Multivariate Regression

The *R-squared* is the fraction of the *Explained Sum of Squares* (*ESS*) divided by the *Total Sum of Squares* (*TSS*):

$$R^2 = \frac{ESS}{TSS} = 1 - \frac{RSS}{TSS}$$

The *R-squared* is a measure of the model *goodness of fit*, with *R-squared* close to 1 for models fitting the data very well, and *R-squared* close to 0 for poorly fitting models.

The *R-squared* is equal to the squared correlation between the response and the fitted values:

$$\rho_{yy_{fit}} = \frac{(y_{fit} - \bar{y})^T (y - \bar{y})}{\sqrt{TSS \cdot ESS}} =$$

$$\frac{(y_{fit} - \bar{y})^T (y_{fit} - \bar{y})}{\sqrt{TSS \cdot ESS}} = \sqrt{\frac{ESS}{TSS}}$$

```
> # Set regression attribute for intercept
> attributes(regmod$terms)$intercept <- 1
> # Regression summary
> regsum <- summary(regmod)
> # Regression R-squared
> rsquared <- ess/tss
> all.equal(rsquared, regsum$r.squared)
[1] TRUE
> # Correlation between response and fitted values
> corfit <- drop(cor(respv, fitv))
> # Squared correlation between response and fitted values
> all.equal(corfit^2, rsquared)
[1] TRUE
```

Adjusted R-squared of Multivariate Regression

The weakness of *R-squared* is that it increases with the number of predictors (even for predictors which are purely random), so it may provide an inflated measure of the quality of a model with many predictors.

This is remedied by using the *residual variance* ($\sigma^2_\varepsilon = \frac{RSS}{d_{free}}$) instead of the *RSS*, and the *response variance* ($\sigma^2_y = \frac{TSS}{n-1}$) instead of the *TSS*.

The *adjusted R-squared* is equal to 1 minus the fraction of the *residual variance* divided by the *response variance*:

$$R^2_{adj} = 1 - \frac{\sigma^2_\varepsilon}{\sigma^2_y} = 1 - \frac{RSS/d_{free}}{TSS/(n-1)}$$

Where $d_{free} = (n - k)$ is the number of *degrees of freedom* of the *residuals*.

The *adjusted R-squared* is always smaller than the *R-squared*.

The performance of two different models can be compared by comparing their *adjusted R-squared*, since the model with the larger *adjusted R-squared* has a smaller *residual variance*, so it's better able to explain the *response*.

```
> nrows <- NROW(predm)
> ncols <- NCOL(predm)
> # Degrees of freedom of residuals
> degf <- (nrows - ncols)
> # Adjusted R-squared
> rsqadj <- (1-sum(resids^2)/degf/var(respv))
> # Compare adjusted R-squared from lm()
> all.equal(drop(rsqadj), regsum$adj.r.squared)
[1] TRUE
```

Fisher's *F-distribution*

Let χ^2_m and χ^2_n be independent random variables following *chi-squared* distributions with m and n degrees of freedom.

Then the random variable:

$$F = \frac{\chi^2_m / m}{\chi^2_n / n}$$

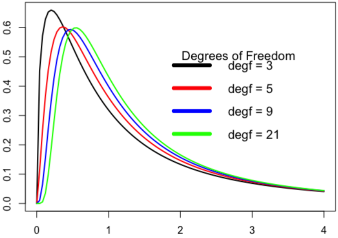
Follows the *F-distribution* with m and n degrees of freedom, with the probability density function:

$$f(F) = \frac{\Gamma((m+n)/2) m^{m/2} n^{n/2}}{\Gamma(m/2)\Gamma(n/2)} \frac{F^{m/2-1}}{(n+mF)^{(m+n)/2}}$$

The *F-distribution* depends on the ratio F and also on the degrees of freedom, m and n.

The function df() calculates the probability density of the *F-distribution*.

```
> # Plot four curves in loop
> degf <- c(3, 5, 9, 21)  # Degrees of freedom
> colorv <- c("black", "red", "blue", "green")
> for (indeks in 1:NROW(degf)) {
+   curve(expr=df(x, df1=degf[indeks], df2=3),
+     xlim=c(0, 4), xlab="", ylab="", lwd=2,
+     col=colorv[indeks], add=as.logical(indeks-1))
+ }  # end for
```




```
> # Add title
> title(main="F-Distributions", line=0.5)
> # Add legend
> labelv <- paste("degf", degf, sep=" = ")
> legend("topright", title="Degrees of Freedom", inset=0.0, bty="n",
+        y.intersp=0.4, labelv, cex=1.2, lwd=6, lty=1, col=colorv)
```

The *F-test* For the Variance Ratio

Let x and y be independent standard *Normal* variables, and let $\sigma_x^2 = \frac{1}{m-1} \sum_{i=1}^{m} (x_i - \bar{x})^2$ and $\sigma_y^2 = \frac{1}{n-1} \sum_{i=1}^{n} (y_i - \bar{y})^2$ be their sample variances.

The ratio $F = \sigma_x^2 / \sigma_y^2$ of the sample variances follows the *F-distribution* with $m-1$ and $n-1$ degrees of freedom.

The *null hypothesis* of the *F-test* test is that the *F-statistic* F is not significantly greater than 1 (the variance σ_x^2 is not significantly greater than σ_y^2).

A large value of the *F-statistic* F indicates that the variances are unlikely to be equal.

The function `pf(q)` returns the cumulative probability of the *F-distribution*, i.e. the cumulative probability that the *F-statistic* F is less than the quantile q.

This *F-test* is very sensitive to the assumption of the normality of the variables.

```
> sigmax <- var(rnorm(nrows))
> sigmay <- var(rnorm(nrows))
> fratio <- sigmax/sigmay
> # Cumulative probability for q = fratio
> pf(fratio, nrows-1, nrows-1)
[1] 0.0642
> # p-value for fratios
> 1-pf((10:20)/10, nrows-1, nrows-1)
 [1] 0.500000 0.318150 0.182964 0.096784 0.047876 0.022467 0.010123
 [9] 0.001888 0.000793 0.000329
```

The *F-statistic* for Linear Regression

The performance of two different regression models can be compared by directly comparing their *Residual Sum of Squares* (*RSS*), since the model with a smaller *RSS* is better able to explain the *response* data.

Let the *restricted* model have p_1 parameters with $df_1 = n - p_1$ degrees of freedom, and the *unrestricted* model have p_2 parameters with $df_2 = n - p_2$ degrees of freedom, with $p_1 < p_2$.

Then their *Residual Sum of Squares* RSS_1 and RSS_2 are independent *chi-squared* random variables with df_1 and df_2 degrees of freedom.

And their difference $(RSS_1 - RSS_2)$ follows a *chi-squared* distribution with $(df_1 - df_2)$ degrees of freedom.

So the F-statistic F:

$$F = \frac{(RSS_1 - RSS_2)/(df_1 - df_2)}{RSS_2/df_2}$$

Follows the *F-distribution* with $(df_1 - df_2)$ and df_2 degrees of freedom (assuming that the *residuals* are normally distributed).

If the *restricted* model has only one parameter (the constant intercept term), then $df_1 = n - 1$, and its fitted values are equal to the average of the *response*: $y_i^{fit} = \bar{y}$, so RSS_1 is equal to the TSS: $RSS_1 = TSS = (y - \bar{y})^2$, so its *Explained Sum of Squares* is equal to zero: $ESS_1 = TSS - RSS_1 = 0$.

Let the *unrestricted* multivariate regression model be defined as:

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

Where y is the *response*, $\mathbb{X}$ is the *predictor matrix* (with k *predictors*, including the intercept term), and β are the k *regression coefficients*.

So the *unrestricted* model has k parameters $(p_2 = k)$, and $RSS_2 = RSS$ and $ESS_2 = ESS$, and then the F-statistic can be written as:

$$F = \frac{ESS/(k-1)}{RSS/(n-k)}$$

The *F-test* for Linear Regression

The sum of the *Explained Sum of Squares* (*ESS*) and the *Residual Sum of Squares* (*RSS*) is equal to the *Total Sum of Squares* (*TSS*):

$$TSS = ESS + RSS$$

A regression model that better explains the *response* data will have a larger *ESS* and a smaller *RSS*.

The *RSS* and the *ESS* follow independent *chi-squared* distributions with $(n - k)$ and $(k - 1)$ degrees of freedom. Where k is the number of explanatory variables (including the intercept term).

Then the *F*-statistic, equal to the ratio of the *ESS* divided by *RSS*:

$$F = \frac{ESS/(k-1)}{RSS/(n-k)}$$

Follows the *F-distribution* with $(k - 1)$ and $(n - k)$ degrees of freedom (assuming that the *residuals* are normally distributed).

```
> # F-statistic from lm()
> regsum$fstatistic
value numdf dendf
 3.37  5.00 94.00
> # Degrees of freedom of residuals
> degf <- (nrows - ncols)
> # F-statistic from ESS and RSS
> fstat <- (ess/(ncols-1))/(rss/degf)
> all.equal(fstat, regsum$fstatistic[1], check.attributes=FALSE)
[1] TRUE
> # p-value of F-statistic
> 1-pf(q=fstat, df1=(ncols-1), df2=(nrows-ncols))
[1] 0.00757
```

The *null hypothesis* of the *F-test* test is that the *F-statistic F* is not significantly greater than 1 (the variance of *ESS* is not significantly greater than of *RSS*).

A large value of the *F-statistic F* indicates that the *ESS* is significantly greater than the *RSS*, and that the regression is able to explain the *response* data well.

A regression model that better explains the *response* data will have a larger *ESS* and a smaller *RSS*, so the *F-statistic F* will be significantly greater than 1.

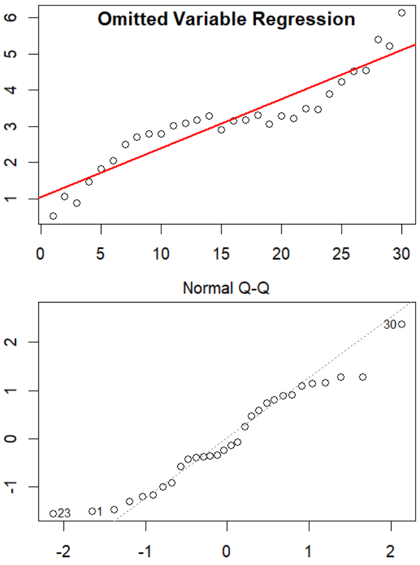
Omitted Variable Bias

Omitted Variable Bias occurs in a regression model that omits important predictors.

The parameter estimates are biased, even though the t-statistics, p-values, and R-squared all indicate a statistically significant regression.

But the Durbin-Watson test shows that the residuals are autocorrelated, which means that the regression coefficients may not be statistically significant (different from zero).

```
> library(lmtest)  # Load lmtest
> # Define predictor matrix
> predm <- 1:30
> omitv <- sin(0.2*1:30)
> # Response depends on both predictors
> respv <- 0.2*predm + omitv + 0.2*rnorm(30)
> # Mis-specified regression only one predictor
> modovb <- lm(respv ~ predm)
> regsum <- summary(modovb)
> regsum$coeff
> regsum$r.squared
> # Durbin-Watson test shows residuals are autocorrelated
> lmtest::dwtest(modovb)
> # Plot the regression diagnostic plots
> x11(width=5, height=7)
> par(mfrow=c(2,1))  # Set plot panels
> par(mar=c(3, 2, 1, 1), oma=c(1, 0, 0, 0))
> plot(respv ~ predm)
> abline(modovb, lwd=2, col="red")
> title(main="Omitted Variable Regression", line=-1)
> plot(modovb, which=2, ask=FALSE)  # Plot just Q-Q
```



Homework Assignment

Required

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