Homework 1

G-Ano08

20 April 2016

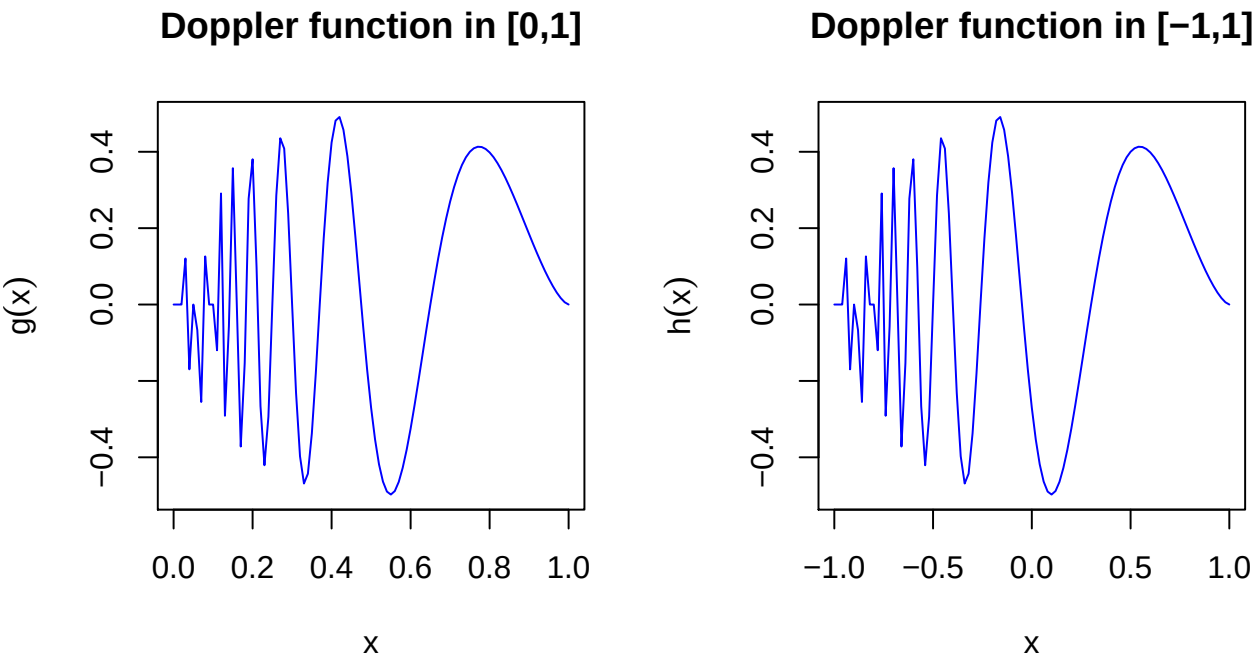
Exercise 1

1.1

The aim of the exercise is to built an approximation of the target function $h(x) = g(0.5(x+1))$, where

$$g(x) = \sqrt{x(1-x)} sin\left(\frac{2.1\pi}{x+0.05}\right)$$

```
# Define the target function

# Doppler function scaled in [0,1]
doppler.fun <-  function(x){
"This function returns the Doppler function in [0,1]:
  x : function argument in [0,1]"
  out = sqrt(x*(1 - x))*sin( (2.1*pi)/(x + 0.05) )
  return(out)
}

# Doppler function scaled in [-1,1]
h_x = function(x){
"This function returns the Doppler function scaled in [-1,1]:
  x : function argument in [-1,1]"
  return(doppler.fun(0.5*(x+1)))
}
```



In order to estimate an approximation of the function we need to be sure that all the requirements are satisfied.

First of all we know that the function space $\mathtt{L}_2([-1,1])$ is defined as

$$\mathtt{L}_2([-1,1]) = \left\{ h : [-1,1] \mapsto \mathbb{R} \quad such \quad that \quad \|h\|_2 = \int_{-1}^{1} |h(x)|^2 \, \mathrm{d}x < \infty \right\}$$

and is known as *Hilbert space*. In this space is defined the *inner product* between two functions $f(\cdot)$ $h(\cdot)$ as follows

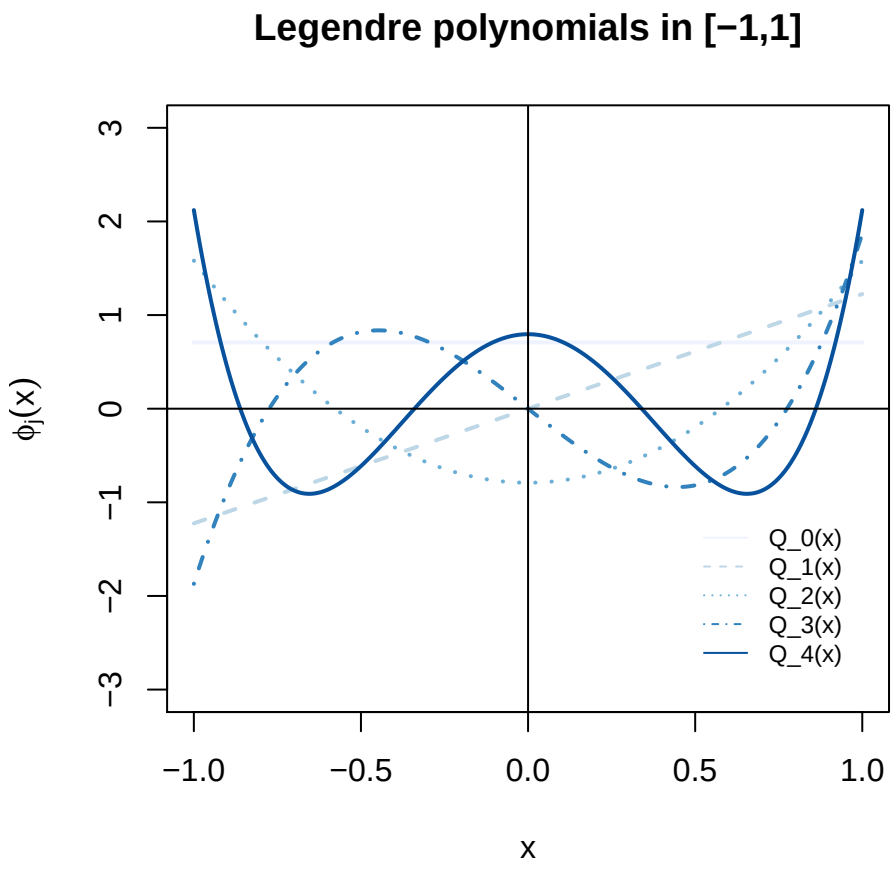
$$\langle f, h \rangle_{\mathtt{L}_2} = \int_{-1}^{1} f(x)h(x)\mathrm{d}x.$$

Furthermore, the *Hilbert space* is defined as *separable*, hence it is possible to define a *countable* orthonormal basis $\{\phi_0(\cdot), \phi_1(\cdot), \dots\}$.

The basis that we consider in this case is the *Legendre*, a typical polynomial extension used for the function defined in [-1,1]. The polynomial $P_j(x)$ of the *Legendre* extension are orthogonal but not orthonormal, so we define a modified version $Q_j(x) = \sqrt{\frac{(2j+1)}{2}} P_j(x)$ which forms an orthonormal basis for $\mathtt{L}_2([-1,1])$.

```
# Set up the function to construct iteratively the Legendre basis
leg.basis = function(x, j) {
    "The function returns the list of the orthogonal and orthonormal basis vectors."
    if (j == 0)
        p = 1 else if (j == 1)
        p = x else {
        # Define the Legendre polynomial p_(j)
        p = x
        # Define the Legendre polynomial p_(j-1)
        p_1 = 1
        for (k in 1:(j - 1)) {
            # Compute the Legendre polynomial p_(j+1)
            p_next = ((2 * k + 1) * x * p - k * p_1)/(k + 1)
            p_1 = p
            p = p_next
        }
    }
    # Orthonormalize the basis vectors
    q = sqrt((2 * j + 1)/2) * p
    return(list(p, q))
}
# Vectorize the function
leg.basis.vec = Vectorize(leg.basis)
```

Once we obtain the modified *Legendre* extension we plot the first five orthonormal basis vectors.



To be sure that the modified basis is orthonormal we build a matrix which shows the result of $\langle \phi_{j1}, \phi_{j2} \rangle_{L_2}$ for all $j_1, j_2 = \{0, 1, \dots\}$.

```
grade=5
# Declare a NA matrix
orth_leg.basis = matrix(nrow=grade, ncol=grade)
# Fill in the matrix : m(j,k) = p_(j)*p_(k)
for (j in 0 : (grade-1)){
  for (k in 0 : (grade-1)){
# Compute the integral in [-1,1] to verify that the basis elements satisfy the requirements
    orth_leg.basis[(j+1),(k+1)] = round( integrate(function(x, j1, j2)
                                  unlist(leg.basis.vec(x, j1)[2,]) *
                                  unlist(leg.basis.vec(x, j2)[2,]),
                                  lower = -1, upper = 1, j1 = k, j2 = j)$value, 5)
    }
  }

orth_leg.basis
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    1    0    0
```

```
## [4,]    0    0    0    1    0
## [5,]    0    0    0    0    1
```

As we expect

$$\begin{cases} \langle \phi_{j_1}, \phi_{j_2} \rangle_{\mathtt{L}_2} = 0 & \forall \quad j_1 \neq j_2 \\ \|\phi_j\|_{\mathtt{L}_2} = 1 & \forall \quad j. \end{cases}$$

1.2 (a)

Once we verify the existence of the orthonormal basis of $\mathtt{L}_2([-1,1])$, we check whether our target function $h(x) \in \mathtt{L}_2([-1,1])$. To prove that we show that

$$\|h\|_2 = \int_{-1}^{1} |h(x)|^2 \, \mathrm{d}x < \infty$$

```
integrate(function(x){return (  (abs(h_x(x)))^2)  }, lower = -1, upper = 1)
```

```
## 0.1717164 with absolute error < 1e-04
```

The value obtained is finite, so our target function $h(x) \in \mathtt{L}_2([-1,1])$.

Thus, follows that

$$h(x) = \sum_{j=0}^{\infty} \beta_j \phi_j(x).$$

The result mentioned above stands if and only if

$$\int_{-1}^{1} h(x)\phi_i(x)\mathrm{d}x = \sum_{j=0}^{\infty} \beta_j \int_{-1}^{1} \phi_j(x)\phi_i(x)\mathrm{d}x$$

(obtained multiplying each side of the equation by $\phi_i(x)$ and applying the inner product definition)

$$\Downarrow$$

$$\langle h, \phi_i \rangle = \sum_{j=0}^{\infty} \beta_j \underbrace{\langle \phi_j, \phi_i \rangle}_{\delta_{ji} = \begin{cases} 0 & j \neq i \\ 1 & j = i \end{cases}} \Leftrightarrow \langle h, \phi_i \rangle = \beta_i$$

In particular the coefficient β_i corresponds to the *Fourier coefficients* that are the ones that return the best approximation.

It can be proved as follows:

Given

- $\{\beta_j\}_j$: the sequence of the Fourier coefficients
- $\{\alpha_j\}_j$: a generic numeric sequence

$$\left\| h - \sum_{j=0}^{J-1} \beta_j \phi_j \right\|_2^2 \leqslant \left\| h - \sum_{j=0}^{J-1} \alpha_j \phi_j \right\|_2^2$$

$$\Downarrow$$

$$\begin{aligned}
\left\| h - \sum_{j=0}^{J-1} \alpha_j \phi_j \right\|_2^2 &= \left\langle h - \sum_{j=0}^{J-1} \alpha_j \phi_j, h - \sum_{j=0}^{J-1} \alpha_j \phi_j \right\rangle \\
&= \langle h, h \rangle - \sum_{j=0}^{J-1} \alpha_j \underbrace{\langle h, \phi_j \rangle}_{\beta_j} - \sum_{j=0}^{J-1} \alpha_j \underbrace{\langle h, \phi_j \rangle}_{\beta_j} + \sum_{j=0}^{J-1} |\alpha_j|^2 \underbrace{\langle \phi_j, \phi_j \rangle}_{1} \\
&= \langle h, h \rangle + \underbrace{\left[\sum_{j=0}^{J-1} |\alpha_j|^2 - 2 \sum_{j=0}^{J-1} \alpha_j \beta_j + \sum_{j=0}^{J-1} |\beta_j|^2 \right]}_{\sum_{j=0}^{J-1} |\alpha_j - \beta_j|^2} - \sum_{j=0}^{J-1} |\beta_j|^2 \\
&= \underbrace{\sum_{j=0}^{J-1} |\alpha_j - \beta_j|^2}_{>0} + \left\| h - \sum_{j=0}^{J-1} \beta_j \phi_j \right\|_2^2
\end{aligned}$$

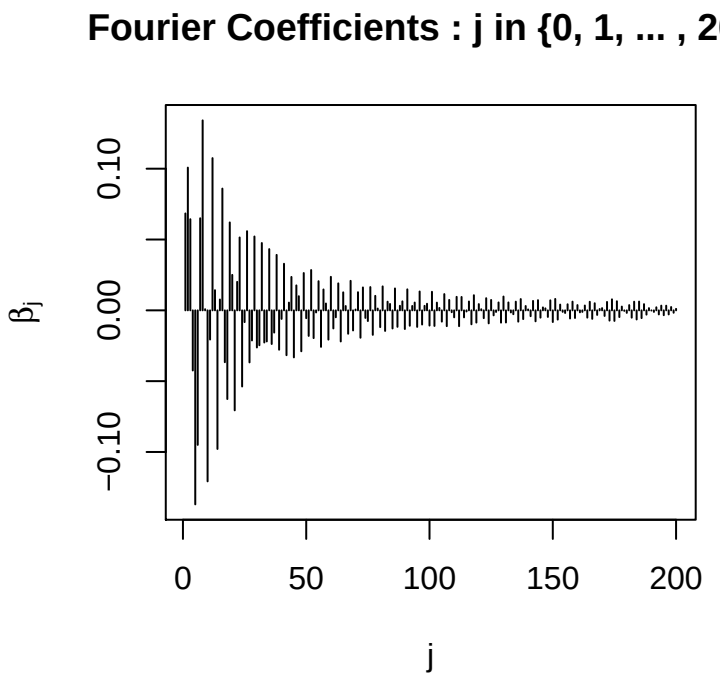
Thus, we define the generalized Fourier coefficient, β_j, in the *Legendre* basis

$$\beta_j = \langle h, \phi_j \rangle_{\mathrm{L_2}} = \int_{-1}^{1} h(x) \phi_j(x) \mathrm{d}x.$$

and evaluate the first 200.

```
# Define the coefficients function
beta_integral = function(j) {
  "This function return the vector of Fourier coefficients. It's lenght depend on:
  - j : length of the vector."
  beta=c()
  # Compute Beta as the integral in [-1,1] of the inner product btw g and phi
  for (i in 0:(j-1)) {
    beta = c(beta, integrate(function(x) h_x(x) * unlist(leg.basis.vec(x,i)[2,]),
                             lower=-1, upper=1)$value)
  }
  return(beta)
```

```
}

# Evaluate the first 200 Fourier coefficients of h in the Legendre basis
beta = beta_integral(200)
```



1.2 (b)

Moreover we can affirm that $h_J(x) = \sum_{j=0}^{J-1} \beta_j \phi_j(x)$ is the element of $\mathtt{L}_2([-1,1])$ that minimizes the $\|h(x) - h_J(x)\|_{L_2}^2$, and it is called the J-*term linear approximation* of $h(x)$.

In particular $h_J(x)$ is the *projection* of $h(x)$ onto the *span* of $\{\phi_0(\cdot), \dots, \phi_{J-1}(\cdot)\}$. We can prove it veryifing that $\langle \pi(h), h - \pi(h) \rangle = 0$, so the *projection* of $h(x)$ onto the *span* of $\{\phi_0(\cdot), \dots, \phi_{J-1}(\cdot)\}$ is orthogonal respect to the differerence between $h(x)$ and the *projection*.

$$\langle \pi(h), h - \pi(h) \rangle = \left\langle \sum_{j=0}^{J-1} \beta_j \phi_j(x), h - \sum_{j=0}^{J-1} \beta_j \phi_j(x) \right\rangle = \left\langle \sum_{j=0}^{J-1} \beta_j \phi_j(x), h \right\rangle - \left\langle \sum_{j=0}^{J-1} \beta_j \phi_j(x), \sum_{j=0}^{J-1} \beta_j \phi_j(x) \right\rangle$$

$$\Downarrow$$

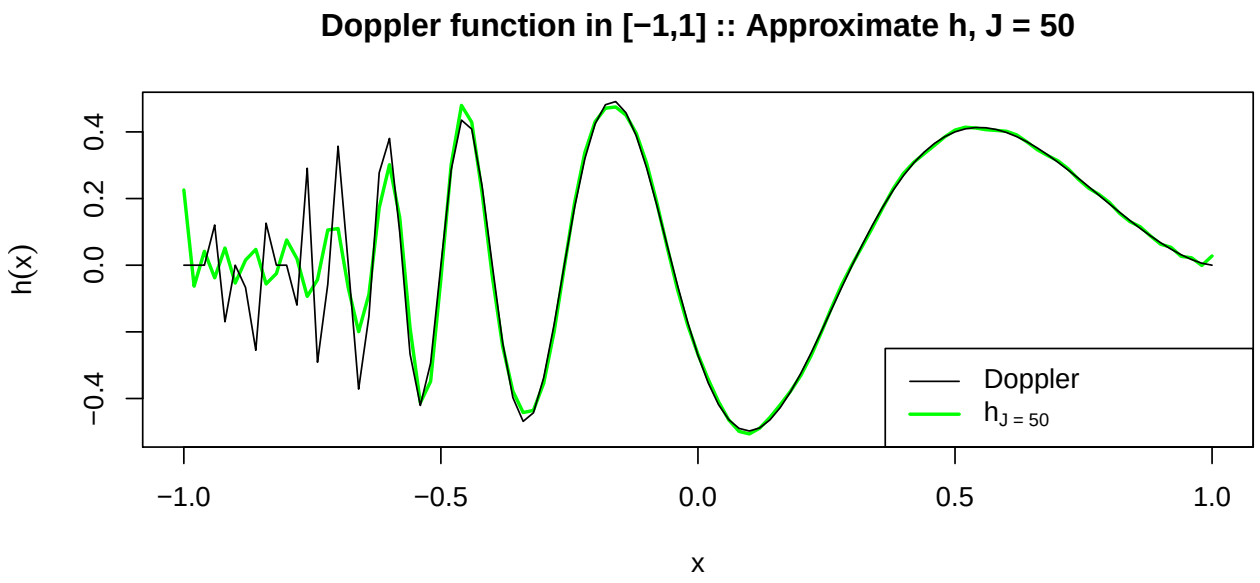
$$\sum_{j=0}^{J-1} \beta_j \underbrace{\int_1^{-1} \phi_j(x) h(x) \mathrm{d}x}_{\beta_j} - \sum_{j=0}^{J-1} |\beta_j|^2 \underbrace{\int_1^{-1} \phi_j^2(x) \mathrm{d}x}_{1} = 0$$

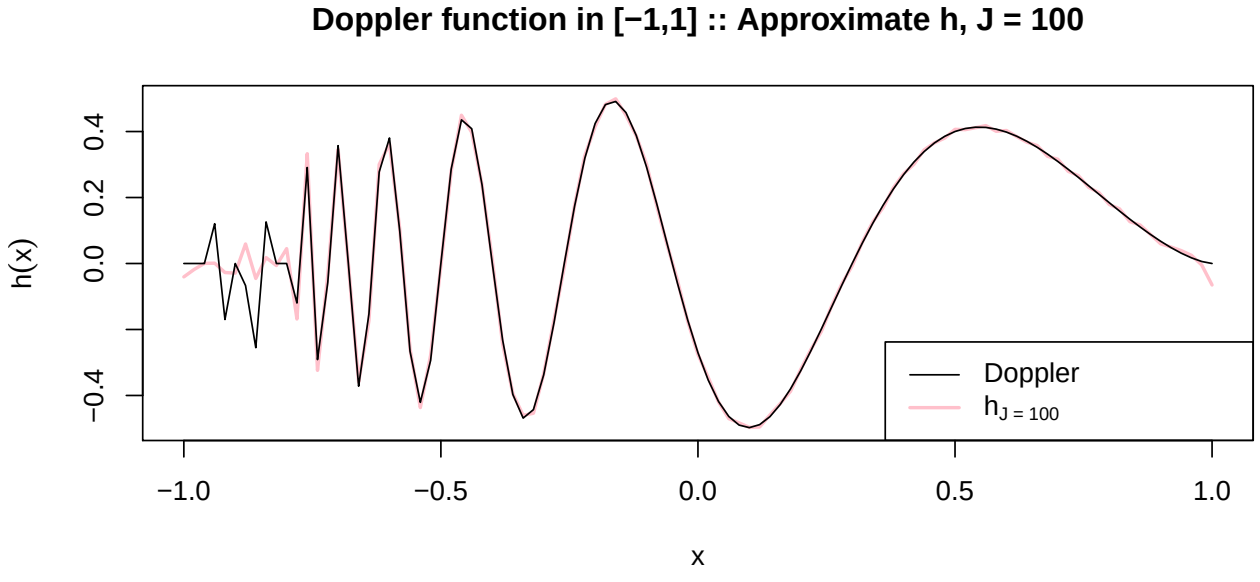
Thus, we are capable to built two *linear approximations*, one for $J = 50$, the other for $J = 100$ and then we can evaluate their squared distance from the target given:

$$\|h(x) - h_J(x)\|_{L_2}^2 = \int_{-1}^{1} (h(x) - h_J(x))^2 \mathrm{d}x$$

```r
# Built two linear approximations for the target function.
# Set up the function to compute the approximation of the target function
h_j = function (x, j, b){
  "This function returns the value of the approximation function given:
  - x : the value in which I compute the function;
  - j : the j-th term for the non-linear approximation;
  - b : the vector of Beta's related to j."
  out = 0
  for (k in 0 : (j-1)){
    out = out + b[k+1] * unlist(leg.basis.vec(x, k)[2,])
  }
  return (out)
}
```

To have an idea on how the story ends..





As we can observe in the figure, the linear approximation obtained cutting $J = 50$ is not extremely different from the one obtained fixing $J = 100$, except for $x \in [-1, -0.5]$ where the latter is remarkably better.

We can verify it also numerically..

```
# One where J = 50
integrate(function(x) (h_x(x) - h_j(x, 50, beta))^2, lower = -1, upper = 1)$value
```

```
## [1] 0.01345591
```

```
# The other for J = 100
integrate(function(x) (h_x(x) - h_j(x, 100, beta))^2, lower = -1, upper = 1)$value
```

```
## [1] 0.003527586
```

.. the squared distances from the target differ for a 10 factor that highly probably depends on the better approximation($J = 100$) in the interval mentioned above.

We can also imagine to increase the value of J, but according to the fact that the values of β_j's for $J > 100$ are small, the improvement of the approximation will not be so significant.

1.2 (c)

Now, define A_j as a non linear space that contains all the functions in the form $\sum_{j=1}^{\infty} a_j \phi_j(x)$ such that at most J of the a_j's are non-zero.

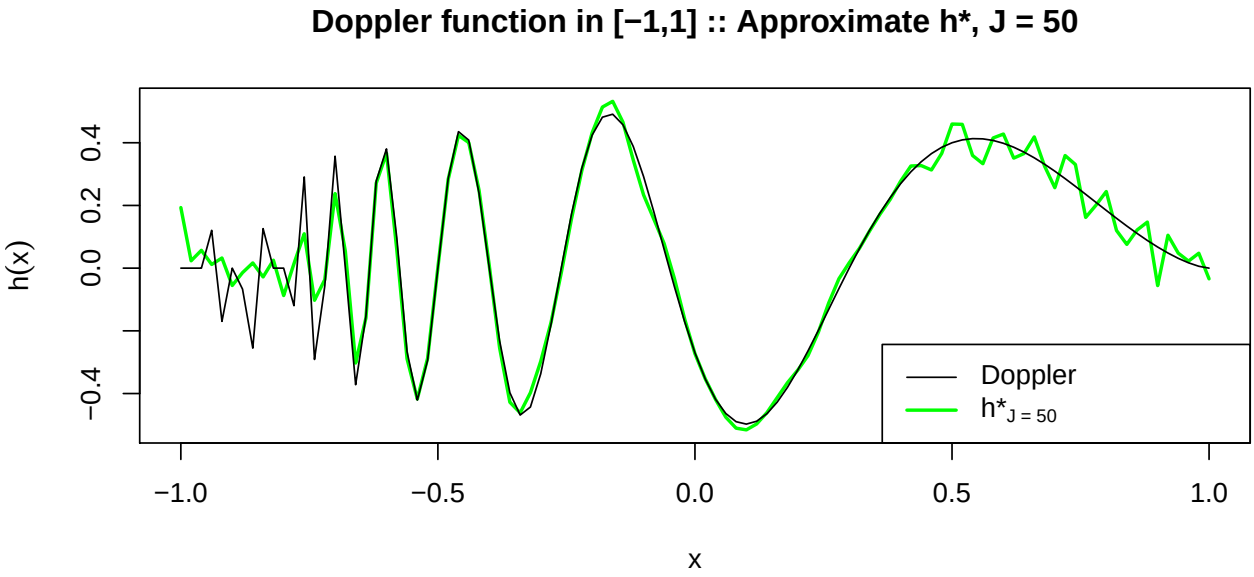
In this space the best approximation to the function $h(x)$ is

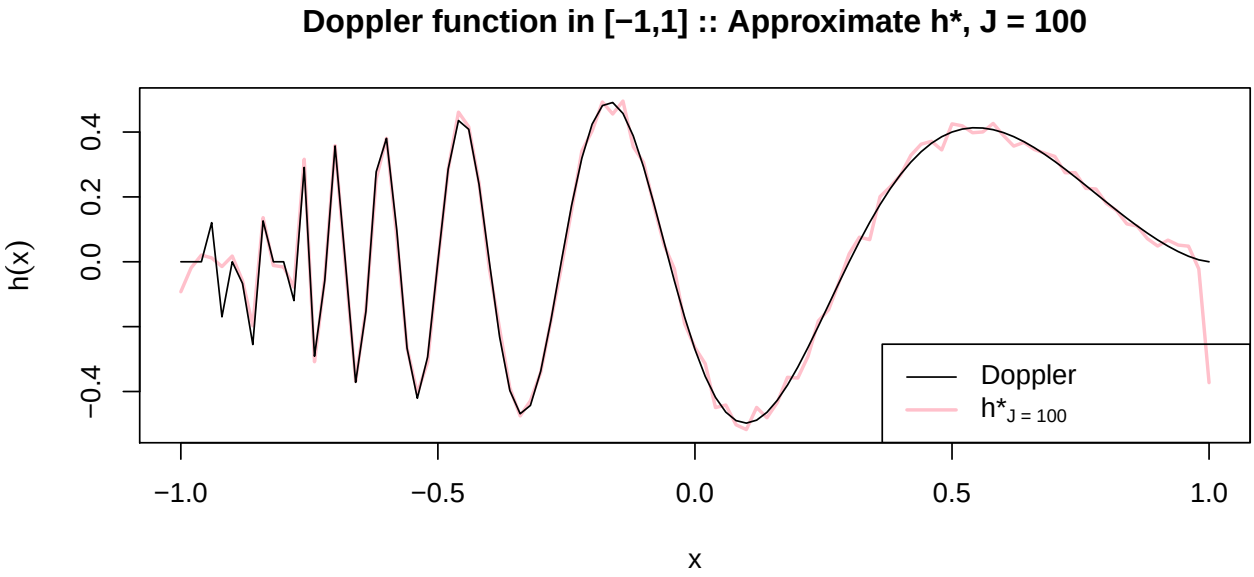
$$h_J^*(x) = \sum_{j \in \Lambda_J} \beta_j \phi_j(x),$$

where Λ_J are the J indices corresponding to the J biggest coefficients $|\beta_j|$'s. That is because β evaluates the covariace in the continuous space and, in particular, it shows how close the function is to the basis elements (big values of β mean that the function is strictly related to the basis).

In particular, for the two cases we present Λ_J is composed respectively by 50 and 100 J's.

```
# In order to compute the greedy approximation define the followinf function
h_j_star = function (x, j, b){
  "This function returns the value of the approximation function given:
  - x : the value in which I compute the function;
  - j : the j-th term for the non-linear approximation;
  - b : the vector of Beta's related to j."
  out = 0
  # Put equal to zero all the beta's smaller than the j-th term.
  b[abs(b) < sort(abs(b), decreasing = TRUE)[j]] = 0
  for (k in 0:(length(b)-1)){
    out = out + b[k+1] * unlist(leg.basis.vec(x,k)[2,])
  }
  return (out)
}
```





Then we evaluate our approximations.

```
# Evaluate the approximation for j = 50
integrate(function(x) (h_x(x) - h_j_star(x, 50, beta))^2, lower = -1, upper = 1)$value
```

```
## [1] 0.009613688
```

```
# Do the same for j = 100
integrate(function(x) (h_x(x) - h_j_star(x, 100, beta))^2, lower = -1, upper = 1)$value
```

```
## [1] 0.00209249
```

As we see from the plots, both the *linear* and the *non-linear* approximations return passing approximation of the target function. We explain this result according to the plot of β's. Infact also the not-ordered β's

assume higher values for the smaller values of J's, hence the coefficients used for the *linear* and the *non-linear* approximations are almost the same.

For example try to analyze beta vector for $J = 50$:

```
# Get the beta's captured by the lineare approximation with J = 50
b_j = beta[1:50]
b_j
```

```
##  [1]  0.0684015246  0.1007691143  0.0643306763 -0.0424656443 -0.1369864121
##  [6] -0.0949871788  0.0650198686  0.1341083357  0.0008080226 -0.1207487365
## [11] -0.0206295623  0.1074667502  0.0142419086 -0.0978618441  0.0076521517
## [16]  0.0859582102 -0.0366239504 -0.0626223324  0.0620811491  0.0250312998
## [21] -0.0706003606  0.0201061035  0.0514592420 -0.0537719799 -0.0083089805
## [26]  0.0558941057 -0.0367534561 -0.0212596823  0.0521325930 -0.0262271003
## [31] -0.0246451533  0.0475311994 -0.0227114109 -0.0219657527  0.0432409956
## [36] -0.0237589388 -0.0157181680  0.0391413241 -0.0278057299 -0.0060426365
## [41]  0.0328884248 -0.0316721034  0.0054680262  0.0235894355 -0.0332117185
## [46]  0.0176173626  0.0099878350 -0.0288993854  0.0263421262 -0.0055437335
```

```
# Pick also the one used by the non-linear estimator
b_j_star = beta[abs(beta) >= sort(abs(beta), decreasing = TRUE)[50]]
b_j_star
```

```
##  [1]  0.06840152  0.10076911  0.06433068 -0.04246564 -0.13698641
##  [6] -0.09498718  0.06501987  0.13410834 -0.12074874 -0.02062956
## [11]  0.10746675 -0.09786184  0.08595821 -0.03662395 -0.06262233
## [16]  0.06208115  0.02503130 -0.07060036  0.02010610  0.05145924
## [21] -0.05377198  0.05589411 -0.03675346 -0.02125968  0.05213259
## [26] -0.02622710 -0.02464515  0.04753120 -0.02271141 -0.02196575
## [31]  0.04324100 -0.02375894  0.03914132 -0.02780573  0.03288842
## [36] -0.03167210  0.02358944 -0.03321172 -0.02889939  0.02634213
## [41]  0.02845409 -0.01960532  0.02059029 -0.02579373 -0.02068432
## [46]  0.02366036  0.01904544 -0.02197300  0.02084104 -0.01933907
```

```
# Count the beta in common for the two approximations
num.common.beta = length(intersect(b_j, b_j_star))
num.common.beta
```

```
## [1] 40
```

Here we see that the two different approximation have in common 40 out of 50 values of β.

```
# Do the same but considering how beta* deviates from beta
beta.ind = tail(which(abs(beta) >= sort(abs(beta), decreasing = TRUE)[50], arr.ind = T),
    1)
over = sum(which(abs(beta) >= sort(abs(beta), decreasing = TRUE)[50], arr.ind = T) >
    50)
over
```

```
## [1] 10
```

Here we observe that the two β's vectors differentiate for 10 coefficients.

```r
# Show the vector of beta's up to index 72 and hold equal to 0 the not used.
b = beta
b[abs(b) < sort(abs(b), decreasing = TRUE)[50]] = 0
b[1:beta.ind]
```

```
##  [1]  0.06840152  0.10076911  0.06433068 -0.04246564 -0.13698641
##  [6] -0.09498718  0.06501987  0.13410834  0.00000000 -0.12074874
## [11] -0.02062956  0.10746675  0.00000000 -0.09786184  0.00000000
## [16]  0.08595821 -0.03662395 -0.06262233  0.06208115  0.02503130
## [21] -0.07060036  0.02010610  0.05145924 -0.05377198  0.00000000
## [26]  0.05589411 -0.03675346 -0.02125968  0.05213259 -0.02622710
## [31] -0.02464515  0.04753120 -0.02271141 -0.02196575  0.04324100
## [36] -0.02375894  0.00000000  0.03914132 -0.02780573  0.00000000
## [41]  0.03288842 -0.03167210  0.00000000  0.02358944 -0.03321172
## [46]  0.00000000  0.00000000 -0.02889939  0.02634213  0.00000000
## [51]  0.00000000  0.02845409 -0.01960532  0.00000000  0.02059029
## [56] -0.02579373  0.00000000  0.00000000 -0.02068432  0.02366036
## [61]  0.00000000  0.00000000  0.01904544 -0.02197300  0.00000000
## [66]  0.00000000  0.00000000  0.02084104  0.00000000  0.00000000
## [71]  0.00000000 -0.01933907
```

Evaluating numerically the loss of the the *linear* and the *non-linear* approximations we see that the former is greater than the latter, it depends on the fact that the *linear* approximation also takes into account some values of β that are small.

Exercise 2

2.1

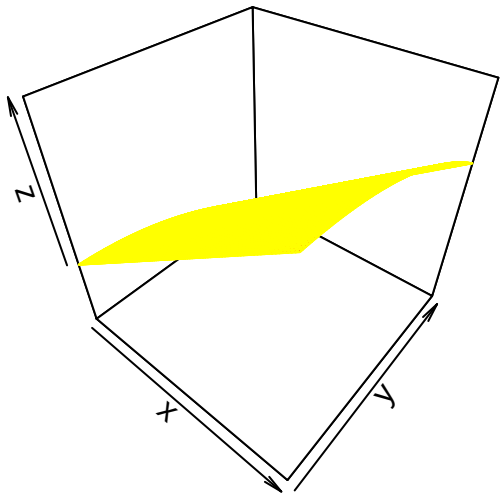
In order to compute a *linear* approximation of the function

$$g(x_1, x_2) = x_1 + cos(x_2)$$

we are going to define some general functions that will be used later for the further analysis.

First of all we define and the target function that corresponds to the one mentioned above.

```
# Define a target function ----------------------------------------
library(plot3D)
library(rgl)
library(plot3D)

# Define the target function that we want to approximate
target.fun = function(x_1, x_2) {
    z = outer(x_1, x_2, FUN = function(x, y) {
        out = x + cos(y)
        return(out)
    })
    return(z)
}
```



You must enable Javascript to view this page properly.

Click **here** to see the 3D plot.

Thus, we verify if the function lies in the *Hilbert space*, $\mathtt{L}_2([0,1] \times [0,1])$. In particular it belongs to the space if

$$\int_0^1 \int_0^1 |g(x_1, x_2)|^2 \mathrm{d}x_1 \mathrm{d}x_2 < \infty$$

```
library(cubature)
adaptIntegrate(function(x) (abs(target.fun(x[1], x[2])))^2, lowerLimit = c(0,
    0), upperLimit = c(1, 1))$integral
```

```
## [1] 1.902129
```

Once we verified the affinity with the *Hilbert space*, we built its orthonormal basis. In particular let $\{\phi_0(\cdot), \phi_1(\cdot), \dots\}$ be the orthonormal basis for the space $\mathtt{L}_2([0,1])$, then the function

$$\{\phi_{j_1 j_2}(x_1, x_2) = \phi_{j_1}(x_1)\phi j_2(x_2) : j_1, j_2 = 0, 1, \dots\}$$

forms an orthonormal basis for $\mathtt{L}_2([0,1] \times [0,1])$, called the *tensor product basis*.

In this case we define the *tensor basis* from the usal *cosine basis*

$$\phi_0(x) = 1 \quad \phi_j(x) = \sqrt{2}cos(j\pi x) \quad j > 1.$$

```
# Define the basis functions ----------------------------------------------

# Cosine-basis
cos.basis = function(x, j) {
    return(1 * (j == 0) + sqrt(2) * cos(pi * j * x) * (j > 0))
}
cos.basis.vec = Vectorize(cos.basis, vectorize.args = "x")

# Since we know that the cosine basis is an orthonormal basis for L2[0,1],
# we affirm that the tensor basis, obtained as : phi(x1,x2) =
# phi(x1)*phi(x_2), is orthonormal too.

# Tensor product basis
tensor.basis = function(x_1, x_2, j1, j2) {
    return(cos.basis(x_1, j1) * cos.basis(x_2, j2))
}
tensor.basis.vec = Vectorize(tensor.basis, vectorize.args = c("j1", "j2"))
```

In order to verify the orthonormality of the *tensor basis*, according to what we said previously, we just need to verify the orthonormality of the *cosin basis*. So we want to obtain that

$$\begin{cases} \langle \phi_{j_1}, \phi_{j_2} \rangle_{\mathtt{L}_2} = 0 & \forall \quad j_1 \neq j_2 \\ \|\phi_j\|_{\mathtt{L}_2} = 1 & \forall \quad j. \end{cases}$$

```
grade=10
# Declare a NA matrix
orth_cos.basis = matrix(nrow=grade, ncol=grade)
for (j in 0 : (grade-1)){
  for (k in 0 : (grade-1)){
# Compute the integral in [0,1] to verify that the basis elements satisfy the requirements
    orth_cos.basis[(k+1),(j+1)] = round( integrate(function(x, j1, j2) tensor.basis(x, x, j1, j2),
lower = 0, upper = 1, j1 = k, j2 = j)$value, 5)
  }
}
orth_cos.basis
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    1    0    0    0    0    0    0    0    0     0
##  [2,]    0    1    0    0    0    0    0    0    0     0
##  [3,]    0    0    1    0    0    0    0    0    0     0
```

```
##  [4,]    0    0    0    1    0    0    0    0    0    0
##  [5,]    0    0    0    0    1    0    0    0    0    0
##  [6,]    0    0    0    0    0    1    0    0    0    0
##  [7,]    0    0    0    0    0    0    1    0    0    0
##  [8,]    0    0    0    0    0    0    0    1    0    0
##  [9,]    0    0    0    0    0    0    0    0    1    0
## [10,]    0    0    0    0    0    0    0    0    0    1
```

Hence, suppose that $\phi_0 = 1$, then any function that belongs to $\mathrm{L}_2([0,1] \times [0,1])$, our *target* function can be expandend in a tensor basis as

$$g(x_1, x_2) = \sum_{j_1=0}^{\infty} \sum_{j_2=0}^{\infty} \beta_{j_1,j_2} \phi_{j_1,j_2}(x_1, x_2) = \sum_{j_1=0}^{\infty} \sum_{j_2=0}^{\infty} \beta_{j_1,j_2} \phi_{j_1)(x_1)} \phi_{j_2}(x_2)$$
$$= \beta_{0,0} + \sum_{j=1}^{\infty} \beta_{j,0} \phi_j(x_1) + \sum_{j=1}^{\infty} \beta_{0,j} \phi_j(x_2) + \sum_{j_1=1}^{\infty} \sum_{j_2=1}^{\infty} \beta_{j_1,j_2} \phi_{j_1)(x_1)} \phi_{j_2}(x_2)$$

where β_{j_1,j_2} denotes the *Fourier* coefficient given by

$$\beta_{j_1,j_2} = \langle g(x_1, x_2), \phi_{j_1,j_2}(x_1, x_2) \rangle_{\mathrm{L}_2} = \int_0^1 \int_0^1 g(x_1, x_2) \phi_{j_1,j_2}(x_1, x_2) \mathrm{d}x_1 \mathrm{d}x_2$$

that are the best to approximate the function. It derives from the extention of the theoretical explanation did in Exercise 1.

```r
# Fourier coefficients ----------------------------------------------------

library(cubature)

# Define the argument that we should integrate to compute the Fourier
# coefficients
integrande.fun = function(x_1, x_2, j_1, j_2) {
    return(tensor.basis(x_1, x_2, j_1, j_2) * target.fun(x_1, x_2))
}
# Then we vectorize the function( for j_1 and j_2) in order to give vectors
# as input
integrande.fun.vec = Vectorize(integrande.fun, vectorize.args = c("j_1", "j_2"))

# We create the Fourier coefficients matrix

# At first we define a function to compute a single Fourier coefficient

fourier.coef = function(j1, j2) {
    out = adaptIntegrate(function(x, j_1, j_2) integrande.fun(x[1], x[2], j_1,
        j_2), lowerLimit = c(0, 0), upperLimit = c(1, 1), j_1 = j1, j_2 = j2,
        maxEval = 50000, absError = 1e-04)$integral
    return(out)
}

# Then we vectorize the function
```

```
fourier.vec = Vectorize(fourier.coef)
# And finally create the Fourier coefficient matrix
fourier.matrix = function(j_1, j_2) {
    return(outer(0:j_1, 0:j_2, FUN = fourier.vec))
}
```

Hence, we are, finally, able to define our *linear* approximation for a pair (J_1, J_2)

$$g_{J_1,J_2}(x_1, x_2) = \sum_{j_1=0}^{J_1-1} \sum_{j_2=0}^{J_2-1} \beta_{j_1,j_2} \phi_{j_1,j_2}(x_1, x_2)$$

```
# Target function approximation ------------------------------------------
library(cubature)
# Define the function that computes the approximation of g given x1, x2 and
# j1 and j2
approximate.fun = function(x.1, x.2, j1, j2) {
    vec_1 = c()
    for (k in 0:(j1 - 1)) {
        vec_2 = c()
        for (i in 0:(j2 - 1)) {
            vec_2[i + 1] = coeff.matrix[k + 1, i + 1] * tensor.basis(x.1, x.2,
                k, i)
        }
        vec_1[k + 1] = sum(vec_2)
    }
    out = sum(vec_1)
    return(out)
}
# Vectorize the approximation function
approximate.vec = Vectorize(approximate.fun)

# Define the matrix of values necessary to plot the approximated function
approximate.matrix = function(x_1, x_2, j_1, j_2) {
    out = outer(x1, x2, FUN = approximate.vec, j1 = j_1, j2 = j_2)
    return(out)
}
```

2.2 and 2.3

Now we choose three different cutoff pairs (J_1, J_2) that allow us to get three different linear approximations for our *target* function.

To pick these pairs we built a matrix of the loss for $J_1, J_2 = 1, \ldots, 10$, where the loss is

$$\|g(x_1, x_2) - g_{J_1,J_2}(x_1, x_2)\|_{L_2}^2 = \int_0^1 \int_0^1 (g(x_1, x_2) - g_{J_1,J_2}(x_1, x_2))^2 \mathrm{d}x_1 \mathrm{d}x_2$$

```
# Evaluate three different approximation ---------------------------------
# Define the functin to compute the loss related to one approximation
loss = function(j_1, j_2) {
    adaptIntegrate(function(x, j1, j2) (target.fun(x[1], x[2]) - approximate.vec(x[1],
        x[2], j1 + 1, j2 + 1))^2, lowerLimit = c(0, 0), upperLimit = c(1, 1),
```

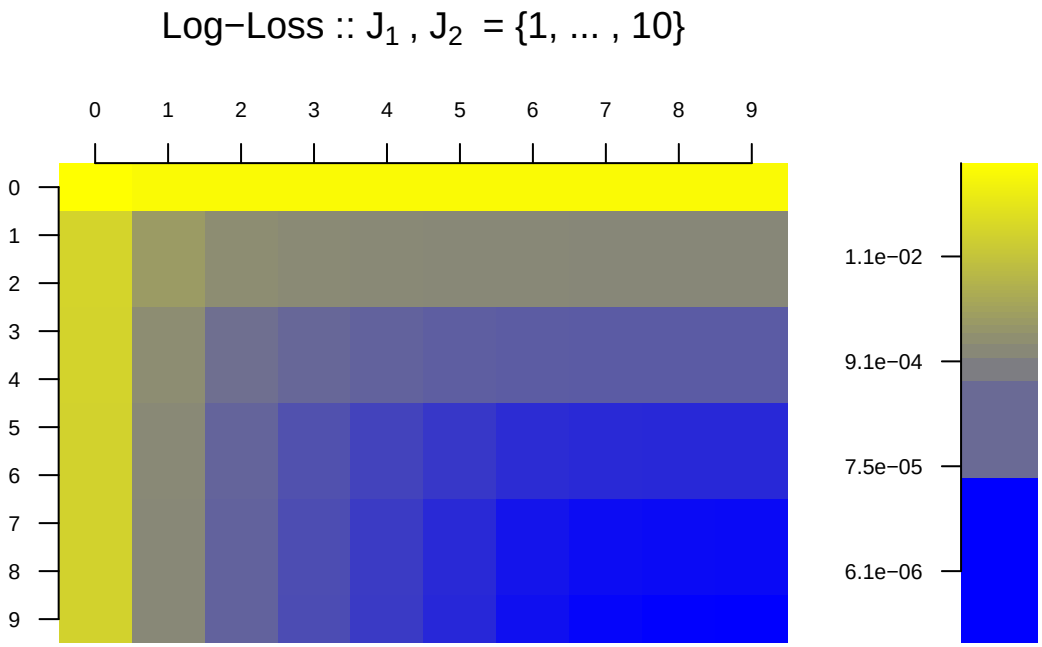
```
        j1 = j_1, j2 = j_2, maxEval = 50000, absError = 1e-04)$integral
}
# Define the matrix of the losses for different combination of beta's
loss.vec = Vectorize(loss)
loss.matrix = function(J1, J2) {
    out = outer(0:(J1 - 1), 0:(J2 - 1), FUN = loss.vec)
    return(out)
}
```

Here the matrix where the columns represent J_2 and the rows J_1.

```
J_1 = J_2 = 10
coeff.matrix = fourier.matrix(J_1, J_2)
err = loss.matrix(J_1, J_2)
err
```

```
##             [,1]        [,2]         [,3]         [,4]         [,5]
##  [1,] 0.10258428 0.084583143 0.0836264879 0.0834635474 8.339737e-02
##  [2,] 0.02045642 0.002455290 0.0015007287 0.0013206361 1.263707e-03
##  [3,] 0.02045642 0.002455290 0.0015007287 0.0013206361 1.263707e-03
##  [4,] 0.01948489 0.001483756 0.0004842083 0.0003641613 2.979856e-04
##  [5,] 0.01948489 0.001483756 0.0004842083 0.0003641608 2.979856e-04
##  [6,] 0.01928363 0.001282494 0.0003258388 0.0001628980 9.672292e-05
##  [7,] 0.01928363 0.001282494 0.0003258388 0.0001628981 9.672289e-05
##  [8,] 0.01925975 0.001258610 0.0003019553 0.0001390144 7.283929e-05
##  [9,] 0.01925975 0.001258610 0.0003019555 0.0001390148 7.284002e-05
## [10,] 0.01925655 0.001255417 0.0002987618 0.0001358212 6.964639e-05
##               [,6]         [,7]         [,8]         [,9]        [,10]
##  [1,] 8.336149e-02 8.334162e-02 8.333737e-02 8.333623e-02 8.333567e-02
##  [2,] 1.235072e-03 1.213761e-03 1.209511e-03 1.208382e-03 1.207814e-03
##  [3,] 1.235072e-03 1.213761e-03 1.209511e-03 1.208382e-03 1.207814e-03
##  [4,] 2.621010e-04 2.422325e-04 2.379832e-04 2.368486e-04 2.362808e-04
##  [5,] 2.621010e-04 2.422324e-04 2.379830e-04 2.368486e-04 2.362808e-04
##  [6,] 6.083833e-05 4.096957e-05 3.672023e-05 3.558574e-05 3.501795e-05
##  [7,] 6.083831e-05 4.096956e-05 3.672024e-05 3.558574e-05 3.501795e-05
##  [8,] 3.695471e-05 1.708607e-05 1.283674e-05 1.170222e-05 1.113442e-05
##  [9,] 3.695536e-05 1.708608e-05 1.283670e-05 1.170218e-05 1.113438e-05
## [10,] 3.376173e-05 1.389246e-05 9.643085e-06 8.508556e-06 7.940756e-06
```

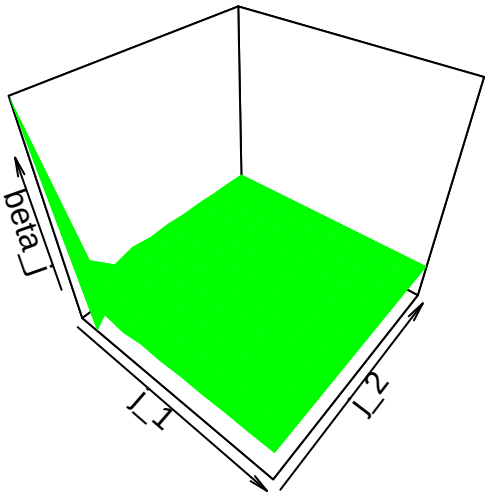
To get a stronger idea of the loss, we would like to plot the matrix.



In particular we gather that the loss is maximum when $J_1 = J_2 = 0$, infact graphically we obtain a plane. Whether we set $J_1 = 0$ and vary J_2, the error commited by the approximation is slightly smaller, from 0.1025843 to 0.0833357. It tells us that if we choose to create our approximation using just the elements of the basis that are proportional only to the cosine (where $J_2 > 0$), we will not improve our approximation so much respect to the case $J_1 = J_2 = 0$. In particulare the inclination is invariate and what change is just the shape of the sinusoide.

On the other hand, when we alter J_1 and hold $J_2 = 0$, the loss slow down remarkably moving J_1 from 0 to 1 and then remain stable; from a graphical point of view the plane change its slope that goes towards the one of the *target* function. That is because in that case the basis elements are not only proportional to a cosine function.

At least but not last we see that when both J_1 and J_2 are grater than 0, the loss go sharply towards zero. It tells us that to have a passing approximation of our target function we need just few basis elements. The latter could be proved by the fact that the values of β become swiftly close to zero. Click **here** to see the 3D plot.



You must enable Javascript to view this page properly.

As we see the *Fourier* coefficients could be considered not significantly greater than zero when J_1, J_2 are both greater than 6.

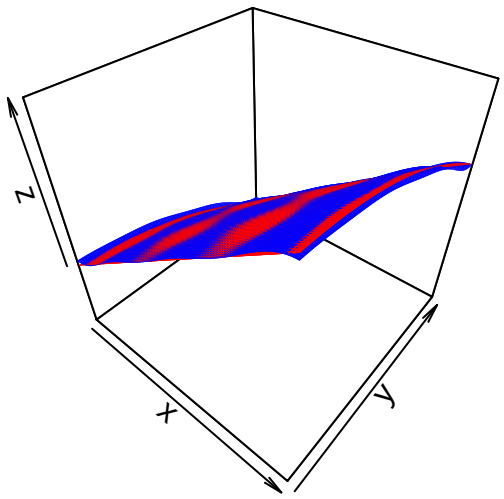
As we can see graphically in the Log-Loss plot, a choose of an homogeneous J_1 and J_2 is better than a non-homogeneous one. In particular we notice how the colour of the cells referring to the diagonal of the matrix get dark faster than the ones outside the diagonal. Furthermore analyzing the matrix *err* considering only the matrix diagonal we see that the first six terms decrease of an order of magnitude one step by step. From this point the loss decreases of a little quantity. That is why, generally, also according to the computational cost, we prefer small values of J. We resume the matrix diagonal below:

$$```rdiag(err)``$$

After this assumptions decide to consider and choose as Js the following three cases:

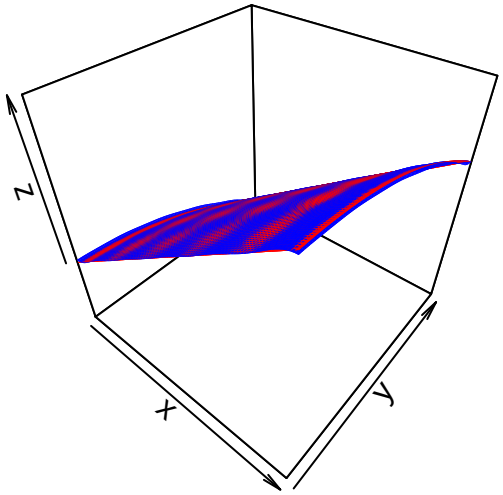
- $J_1 = J_2 = 6$
- $J_1 = J_2 = 10$
- $J_1 = 10, J_2 = 6$

1. $J_1 = J_2 = 6$, whose plot is **here** and the loss is 4.1×10^{-5}



You must enable Javascript to view this page properly.

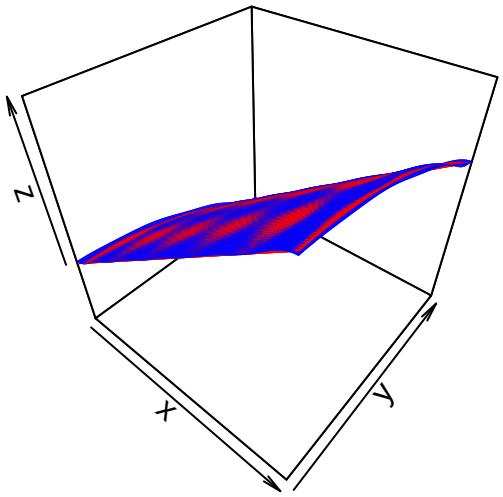
2. $J_1 = J_2 = 10$, whose plot is **here** and the loss is 7.4×10^{-6}



You must enable Javascript to view this page properly.

Of course the loss of the second approximation is better than the first one. But we want to stress that due to the fact the both quantities are extreamely small, we can consider respectable either the first and the second approximation.

3. $J_1 = 10, \quad J_2 = 6$ whose plot is **here** and the loss is 1.4×10^{-5}



You must enable Javascript to view this page properly.

Generally, considering the function $g(x_1, x_2) = x_1 + cosx_2$, we notice that when $\phi_{0,0} = 1$ and $\phi_{j_1,j_2} \propto cos(\cdot)($ where $j_1, j_2 = 0, 1, \ldots)$ we are able to obtain an approximation of the *target* just with a simple linear combination of, the above mentioned, ϕ. That is an alternative explanation for the fact that we choose small values J's to get a reasonable approximation.

Moreover, paying attention on the error matrix we notice that the loss decrease quickly if the J's grow homogeneously. In addition, we can numerically show that if we compare the loss of the J's combination where $J_1 + J_2 = k$, the one with the smallest loss is the one where $J_1 = J_2 = \frac{k}{2}$.

Although the homogeneous J's have smaller losses, we observe also that the differences between the *homogeneous* and the *non-homogeneous* J's decreases increasing the values of J's.

Exercise 3

3.1 and 3.2 Problem setting and covariates mulitcollineariy

In this exercise we have typical chemometrics highly dimensional dataset, with $n = 28$ observation and $r = 268$ covariates, meaning that we have $r >> n$. With this data we want to study the relationship between the overall density of a PET yarn to its NIR spectrum by fitting a linear regression model, however, since the number of covariates is bigger than the observation, we must select lower number of covariates respect to the number of observations. First of all let's load the data in memory and rearrange them.

```
# Step 1: load data into memory

PET = read.csv("PET.txt", sep = " ")
range = 1:21
PET.train = PET[range, -270]
PET.test = PET[-range, -270]
# Define the train set
y.tr = PET.train[, 269]
X.tr = PET.train[, -269]
# Define the test set
y.te = PET.test[, 269]
X.te = PET.test[, -269]

Z.tr = data.frame(X.tr)
Z.te = data.frame(X.te)

apply(Z.tr, 2, mean)
apply(Z.tr, 2, sd)

round(apply(Z.tr, 2, function(x) c(mean(x), sd(x))), 10)
```

By plotting the dataset's NIR spectra we see that the covariates (frequencies) have very similar characteristics, although there are noticeable differences in some curves. This information gives us an insight that probably our covariates set suffers of multicollinearity and fitting a linear regression with the wrong one would lead our model to overfitting.

Those hints suggest us that we need some kind of automated machinery that pick for us the most uncorrelated covariates among them to predict the output. In out case we will use: Stepwise Forward Selection Search.

3.3 Stepwise Forward Selection Search and validation techniques

The *stepwise forward selection search* will select for us the 21 most important covariates with respect to the PET density. This method computes iterativly the correlation between each covariates and the residuals of the output and the active set of covariates.

```
fwd.reg <- function(y, X, k.max = ncol(X), stand = TRUE) {
    # Standardize if asked
    if (stand)
        X <- scale(X)
    # Initialize variable sets & other quantities
    S = NULL
    # active set
    U = 1:ncol(X)
```

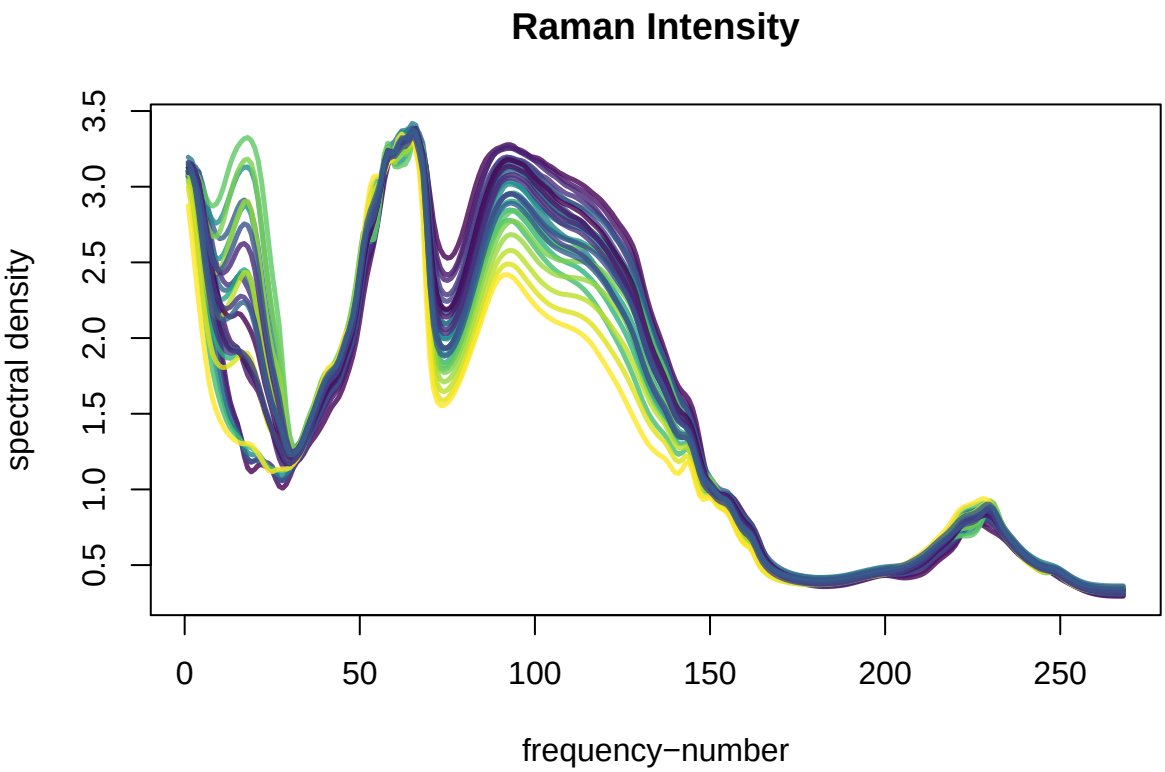


Figure 1: Raman NIR spectra of the polyethylene tereph-thalate (PET) yarns sample. the spectra appear to have very similar characteristics, although there are noticeable differences in some curves

```
    # inactive set
    k.loc = 0
    # current active set dim
    ee = y
    # Loop
    while (k.loc < k.max) {
        ## Update loop-index
        k.loc = k.loc + 1
        ## Step 1: Evaluate correlation with inactive variables
        rr = abs(cor(X[, U], ee))
        ## Step 2: Extract the most correlated variable & update the sets
        J = U[which.max(rr)]
        S = c(S, J)
        U = U[-which.max(rr)]
        ## Step 3: Regress on active var's and get residuals
        ee = resid(lm(y ~ X[, S]))
    }
    # Output
    return(S)
}

S.hat = fwd.reg(y.tr, X.tr)
print(c("Frequency number (covariates) selected by SFWDS:", S.hat), quote = FALSE)
```

```
##  [1] Frequency number (covariates) selected by SFWDS:
```

```
##  [2] 40
##  [3] 241
##  [4] 1
##  [5] 150
##  [6] 90
##  [7] 59
##  [8] 17
##  [9] 77
## [10] 56
## [11] 57
## [12] 65
## [13] 4
## [14] 66
## [15] 61
## [16] 63
## [17] 2
## [18] 54
## [19] 68
## [20] 67
## [21] 60
```

The `S.hat` variable is an array containing the index of the most explicative covariates selected by SFWDS.

This is an empirical technique filtered for us the most correlated covariates with the output, but SFWDS is strongly criticized for the following reason, expecially in multicollinearity settings:

- Whether the input variables are highly correlated the stepwise methods can lead to confusing conclusions;
- It is not sure that the set of variables obtained by the forward selection contains the "best" subset of covariates;
- The single answer returned by a stepwise procedure may not necessary be the only good solution for regression purposes.

This push us checking for which subset of covariates this subset is useful. We need some kind of validation techniques that tell us which of the subset is better than the other, so we define four functions that we will use to computer the optimal k.

We will use four validation techniques

- Train-Test Split
- K-fold cross-validation
- Leave-One-Out cross-validation
- Approximated Leave-One-Out cross-validation

3.3.a - Train-Test Split

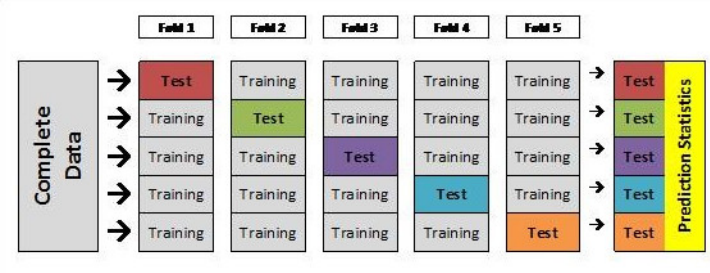
The first validation technique that we try to get an optimal-k is the Train-Test Split validation tecnique. With this technique we simply split our dataset in two part: Train (21 observations) and Test (7 observations) and fit a linear model on the train set. Using this model we compute the MSE between the predicted output and the model fitted over the test set and the real values of the test set.

```
TTS = function(S.hat, mod) {
    # FWD-selected test set - only covariates
    z.te <- Z.te[, S.hat, drop = FALSE]
    # Predict on the FWD-selected test set
    y.hat <- predict(mod, z.te)
    # Evaluate empirical MSE
    return(mean((y.te - y.hat)^2))
}
```

3.3.b - K-Fold Cross-Validation

K-fold cross validation is a method to assess the predictive expressivness of a model. The data set is divided into k subsets, in our specific case 5, by a random permutation of the rows index. At each iteration, one of the k subsets is used as a test-set and the other k-1 subsets are combined and used as a train set. Then the average of the errors in all k tests is calculated by averaging the errors obtained.

The advantage of this method is that how the split is performed is not so influential on the final result, infact, each fold at the end of the execution is used one time as test set, and k-1 times as a train set. The variance of the estimate is reduced increasing the number of k (folds). The disadvantage of this method is that the training algorithm must be executed k times, which means that it takes at least k times to make an evaluation.

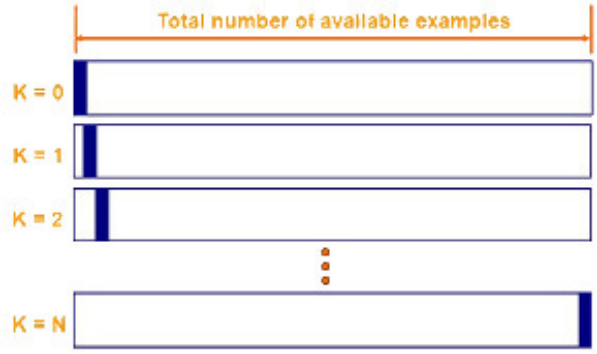



```
KF = function(S.hat, idx) {
    if (idx > 15)
        return(NA)
    set.seed(123)
    nfold = 5
    rarr = sample(1:nrow(PET.train))
    nFold.train = PET.train[S.hat]
    nFold.train["y"] = y.tr

    score = c()
    aux = c(1:4)

    for (k in 1:nfold) {
        if (k == nfold) {
            idxFold = rarr[c(aux, c((aux[4] + 1):length(rarr)))]
        } else {
            idxFold = rarr[aux]
        }
        y.aux1 = y.tr[-idxFold]
        dat <- data.frame(y.aux1, z.tr <- Z.tr[-idxFold, S.hat, drop = FALSE])
        mod <- lm(y.aux1 ~ ., data = dat)
        z.tr <- Z.tr[idxFold, S.hat, drop = FALSE]
```

```
        y.hat <- predict(mod, z.tr)
        y.hat
        score[k] = mean((y.tr[idxFold] - y.hat)^2)
        aux = aux + 4
    }
    return(mean(score))
}
```

3.3.c Leave-One-Out Cross-validation

The Leave-One-Out is a special case of K-fold where the folds are exactly the number of the observation (in our case 21).



As one could imagine from the above scheme the Leave-One-Out in principle would be very expensive to compute but the analytical formula of the error allows us to compute the error in one shot by fitting just one model:

$$\hat{R}_{LOO(S)} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{Y_i - \hat{Y}_i(S)}{1 - \mathbb{H}_{i,i}(S)}\right)^2$$

where $\mathbb{H}_{i,i}(S)$ is the i^{th} diagonal element of the hat matrix

$$\mathbb{H}(S) = \mathbb{X}_S(\mathbb{X}_S^T\mathbb{X}_S)^{-1}\mathbb{X}_S^T$$

where $\mathbb{X}$ is the design matrix.

The main drawback of the LOO is that the resulting MSE is affected by an higher variance than k-fold (the training sets in LOO have more overlap. This makes the estimates from different folds more dependent than in the k-fold, and hence increases the overall variance).

```
LOO1 = function(S.hat, mod) {
    z.tr <- Z.tr[, S.hat, drop = FALSE]
    y.hat = predict(mod, z.tr)

    LOO = data.matrix(Z.tr[S.hat])
    H_hat2 = LOO %*% solve(t(LOO) %*% LOO) %*% t(LOO)
    somma = 0
    for (k in 1:21) {
        somma = somma + ((y.tr[k] - y.hat[k])/(1 - H_hat2[k, k]))^2
    }
```

```
    return(somma/21)
}
```

3.3.d - Approximated Leave-One-Out Cross-Validation

The approximated version of the LOO allows us to avoid the computation of the H-hat matrix which implies the computation of an inverse of a matrix $((\mathbb{X}_S^T \mathbb{X}_S)^{-1}$ which has a cubic cost $O(n^3))$. So we approximate the previous formula with:

$$\hat{R}_{LOO(S)} = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{Y_i - \hat{Y}_i(S)}{1 - \frac{|S|}{n}} \right)^2$$

where we have approximated $\mathbb{H}_{i,i}(S)$ with its average value:

$$\frac{1}{n} \sum_{i=1}^{n} (\mathbb{H}_{i,i}(S)) = \frac{trace(\mathbb{H}(S))}{n} = \frac{|S|}{n}$$

```
LOO2 = function(S.hat, mod) {
    z.tr <- Z.tr[, S.hat, drop = FALSE]
    y.hat = predict(mod, z.tr)

    somma = 0
    for (k in 1:21) {
        somma = somma + ((y.tr[k] - y.hat[k])/(1 - 20/21))^2
    }
    return(somma/21)
}
```

Conclusions

The meaty part! At the beginning of our journey we where looking for using SFWDS and the validation techniques to get the optimal model to predict our output avoiding multicollinearity and hence overfitting. We now have to compute those optimal models! The following chunk of code that computes for us in one shot the optimal k (hence the optimal model) respect to each specific validation technique.

```
scorers = list("Test-Train Split ", "K-fold C-V", "Leave-One-Out C-V", "Approx. LOO C-V")

compute.ktop = function(S.hat) {
    # Initialize the vectors of scores
    modelTTS.score <- rep(NA, length(S.hat))
    modelKF.score <- rep(NA, length(S.hat))
    modelLOO1.score <- rep(NA, length(S.hat))
    modelLOO2.score <- rep(NA, length(S.hat))
    # Loop
    for (idx in 1:length(S.hat)) {
        # Build the data.frame with the FWD-selected variables To understand the
        # option 'drop = FALSE' read the help file: ?'['
        dat <- data.frame(y.tr, z.tr <- Z.tr[, S.hat[1:idx], drop = FALSE])
        # Fit the linear model on FWD-selected training data
        mod <- lm(y.tr ~ ., data = dat)
```

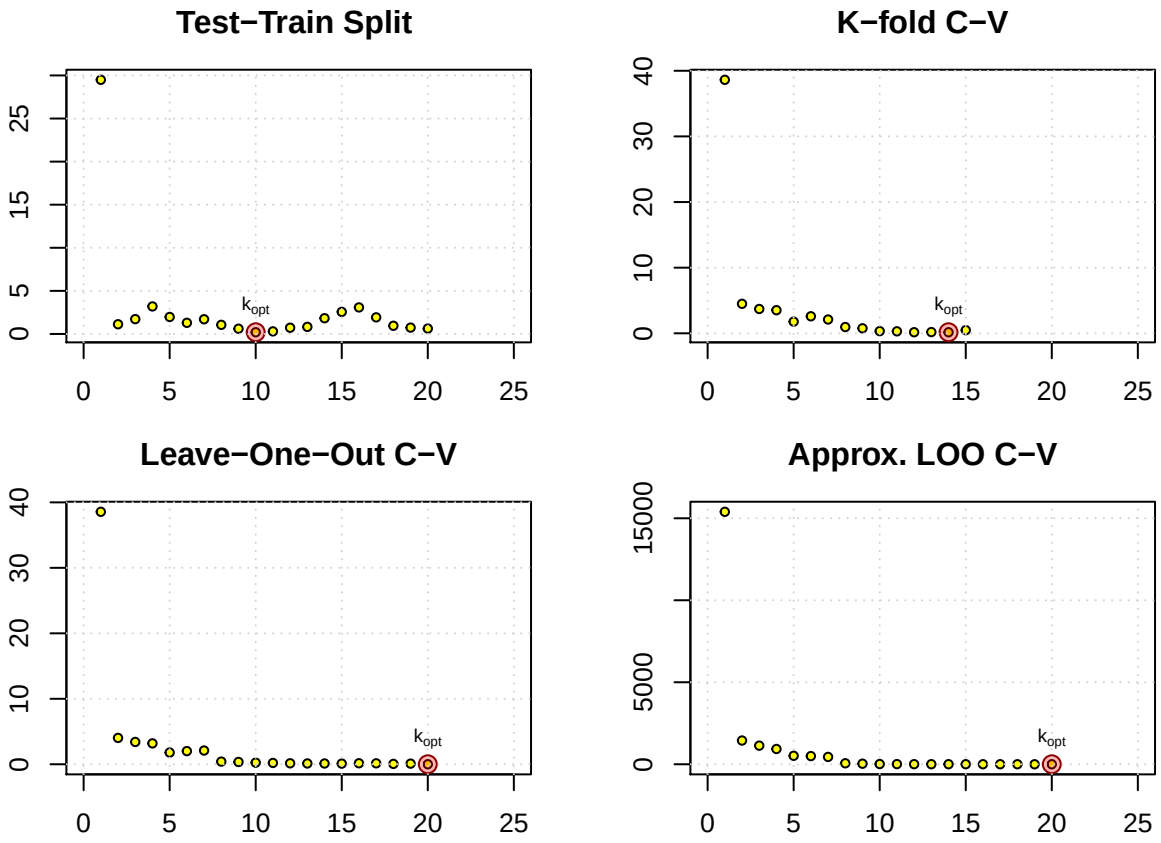
```
        modelTTS.score[idx] = TTS(S.hat[1:idx], mod)
        modelKF.score[idx] = KF(S.hat[1:idx], idx)
        modelLOO1.score[idx] = LOO1(S.hat[1:idx], mod)
        modelLOO2.score[idx] = LOO2(S.hat[1:idx], mod)
    }
    # Optimal number of covariates

    par(mfrow = c(2, 2), plt = c(1, 1, 2, 2), mar = c(2, 2, 3, 3))
    k1 = which.min(modelTTS.score)
    plot.k.opt(modelTTS.score, k1, scorers[[1]])
    # print(modelTTS.score)
    k2 = which.min(modelKF.score)
    plot.k.opt(modelKF.score, k2, scorers[[2]])
    # print(modelKF.score)
    k3 = which.min(modelLOO1.score)
    plot.k.opt(modelLOO1.score, k3, scorers[[3]])
    # print(modelLOO1.score)
    k4 = which.min(modelLOO2.score)
    plot.k.opt(modelLOO2.score, k4, scorers[[4]])
    # print(modelLOO2.score)

    return(c(k1, k2, k3, k4))

}
```

Now that we have a function to compute the optimal k with respect each model, we need a function to plot the results in a way that we are able to visualize them and see how much this optimal k with respect to each one of those is better with respect to the others.

```
plot.k.opt = function(model.score, k.opt, scorer) {
    plot(model.score, pch = 21, bg = "yellow", xlim = c(0, 25), cex = 0.7, main = scorer,
        ylab = "RSS-Test")
    points(k.opt, model.score[k.opt], pch = 21, cex = 1.5, col = "darkred",
        bg = rgb(1, 0, 0, 0.3))
    text(k.opt, model.score[k.opt], expression(k[opt]), cex = 0.7, pos = 3)
    grid()

}
```

Now that we have coded the plot function we are able to compute and display the optimal model with respect to each model we just run it.

```
k.opt = compute.ktop(S.hat)
```




```
k.opt
```

```
## [1] 10 14 20 20
```

From the four plots we can clearly see that the four validation techiniques resulted in three different values of k and hence three different models. The next step is to understand which one of those is better with respect to our test set. Now we simply fit three linear models with k = 10, 14, 20 and compute the MSE with respect to the test set.

```
corMat = list()
k = 1
for (i in k.opt) {
    dat = data.frame(y.tr, z.tr <- Z.tr[, S.hat[1:i], drop = FALSE])
    mod = lm(y.tr ~ ., data = dat)
    mod.sum = summary(mod)
    y.hat = predict(mod, Z.te[, S.hat[1:i], drop = FALSE])
    print(matrix(c(scorers[[k]], i, mod.sum$r.squared, mean(resid(mod)^2), mean((y.te -
        y.hat)^2)), nrow = 1, ncol = 5, dimnames = list(c(), c("validation technique",
        "Optimal K:", "R-squared", "RSS-train", "RSS-test"))))
    corMat[[k]] = data.matrix(Z.tr[, S.hat[1:i]])
    k = k + 1
}
```

```
##      validation technique Optimal K: R-squared
## [1,] "Test-Train Split "  "10"       "0.999967487150546"
##      RSS-train           RSS-test
## [1,] "0.028912118570056" "0.198832403603585"
##      validation technique Optimal K: R-squared
## [1,] "K-fold C-V"         "14"       "0.999992713539842"
##      RSS-train            RSS-test
## [1,] "0.00647949975462287" "1.82735798916159"
##      validation technique Optimal K: R-squared RSS-train
## [1,] "Leave-One-Out C-V"  "20"       "1"       "0"
##      RSS-test
## [1,] "0.625022906600984"
##      validation technique Optimal K: R-squared RSS-train
## [1,] "Approx. LOO C-V"    "20"       "1"       "0"
##      RSS-test
## [1,] "0.625022906600984"
```

The first thing that we clearly see is that the R-squared value is approximately equal to one for each of those models.

Comments

The code below produces a vector *corMat* which elements are the correlation matricies of the different models.

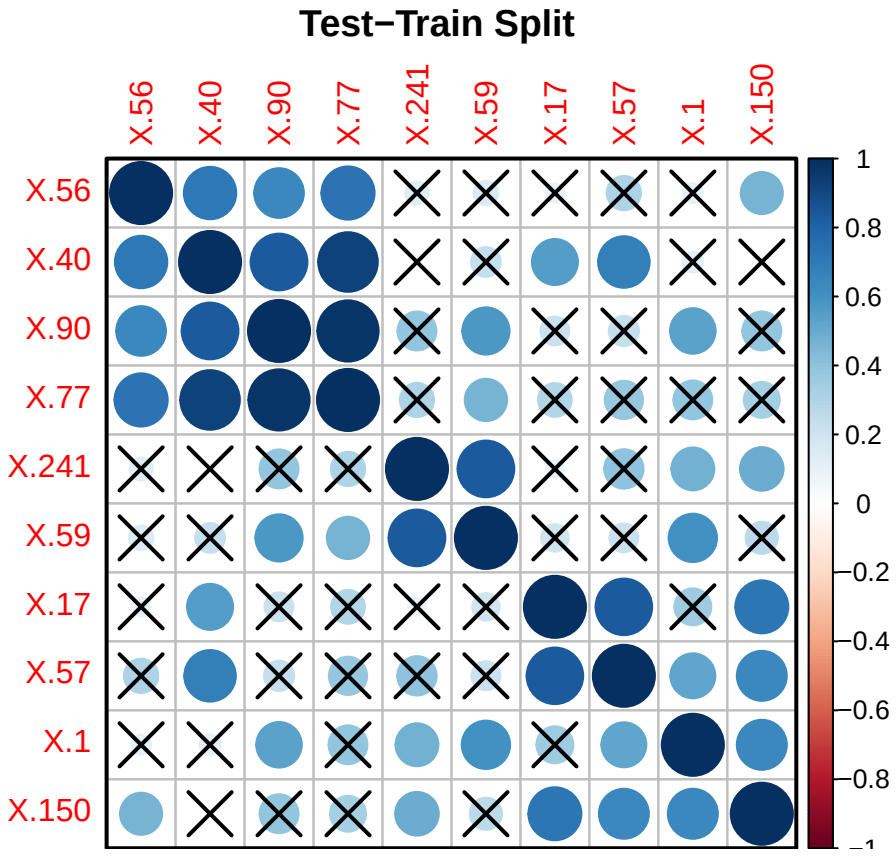
```
cor.mtest <- function(mat, conf.level = 0.95){
  mat <- as.matrix(mat)
    n <- ncol(mat)
    p.mat <- lowCI.mat <- uppCI.mat <- matrix(NA, n, n)
    diag(p.mat) <- 0
    diag(lowCI.mat) <- diag(uppCI.mat) <- 1
    for(i in 1:(n-1)){
        for(j in (i+1):n){
            tmp <- cor.test(mat[,i], mat[,j],
                   alternative = "two.sided",conf.level = conf.level)
            p.mat[i,j] <- p.mat[j,i] <- tmp$p.value
            lowCI.mat[i,j] <- lowCI.mat[j,i] <- tmp$conf.int[1]
            uppCI.mat[i,j] <- uppCI.mat[j,i] <- tmp$conf.int[2]
        }
    }
    return(list(p.mat, lowCI.mat, uppCI.mat))
}
```

To understand how much multicollinearity affect our subset of covariates is useful to use visualization and clustering. With this goal in mind we define a function that performs a double-sided Test(at leve 0.95) for correlation between each pair of covariates in the the design matrix.

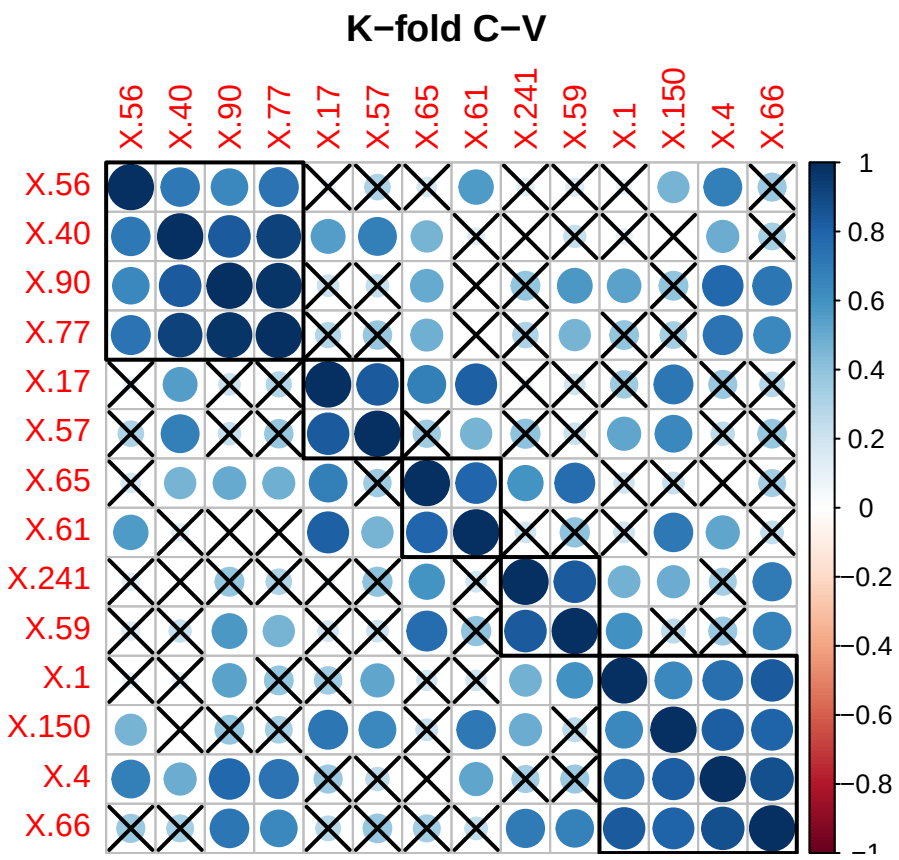
Then for each of the three model we plot the *absolute correlation matrix* ordered using herarcical clustering up to a point where no tick is present within any cluster, in a way that we can consider any vector in the cluster "replaceable" with another in that specific cluster.

Important remark:

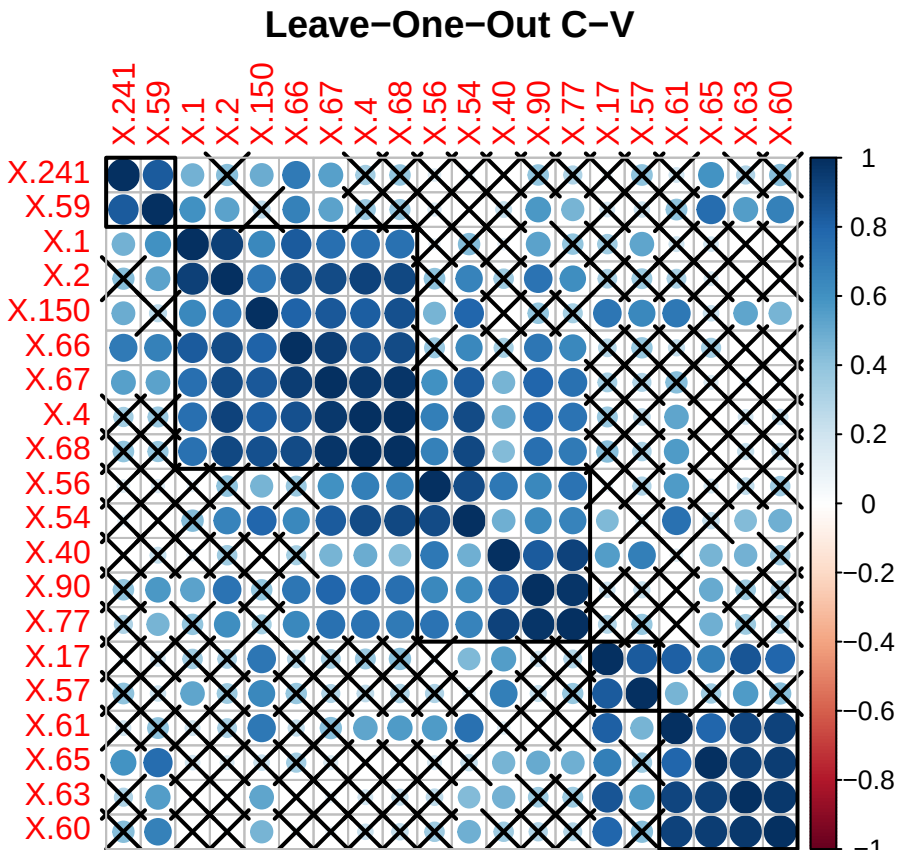
We use the absolute correlation matrix because the multicollinearity affects not only almost correlated covariates, but also almost anticorrelated variable because they span the same subspace, but, since hierarcical clustering is defined only for positive distances, we get the absolute values of the matrix and run it as there was no anticorrelation.



For the first model with k = 10 we clearly see that we are able to create 4 different clusters in which the covariates have passed the correlation test among themselfs. This implies that in order to avoid multicollinearity we should select 4 out of 10 covariates to rappresent the whole subset and drop the other 6.



For the second model with k = 14 we see that we can produce a minimum of 5 clusters without any tick in it. This suggest us that in this case just one out the four new covariates it is really useful to improve our model.



In the last model, which is rapresentative for both the LOO and the approximated LOO, we have k=20. Even in this case the cluster are 5, meaning that none of the new covariates are really useful to improve our model: none of them is spanning a new subset of the space.

With this analysis of the clustered absolute correlation matricies we see that the stepwise foward search is not really able to detect an optimal subset of covariates even if coupled with cross validation. Our K = 10, 14, 20 are clearly affected by multicollinearity inducing our model to be overfitted on our specific dataset, infact we have $R^2 \approx 1$, for each of three models.