6 Annexes.

6.1 Sequence realign.

In this annex the codes used to realign the sequences taken from [1]. These sequences are aligned following the KABAT scheme and have to be realigned to follow the AHo one.

```
from Bio.Seq import Seq
from Bio import SeqIO
from Bio.SeqRecord import SeqRecord
import subprocess as sub

seqs_list = list(SeqIO.parse(open('/home/david/Desktop/TFM/DatosAstati/ModAsti/allYES_uniqueAA_c90-140_IGHV1-2_IGHJ2_SEQMA45_3SE9GG_reH2_allSEQMA0.afasta'),'fasta'))
# seqs_list = list(SeqIO.parse(open('/home/david/Desktop/TFM/DatosAstati/ModAsti/S2_cluVJ_reH2_allSEQMA0.afasta'),'fasta'))

def quita_gaps(seq):
  cand = ''
  for amin in seq:
    if amin != '-':
      cand += amin
  return cand

def Ident(seqs_list):
  ident = []
  for i in range(len(seqs_list)):
    ident.append(seqs_list[i].id)
  return ident

def escribe_fasta(Chains, Ident, f_out):
  for N_C in range(len(Chains)):
    record = SeqRecord(Seq(str(Chains[N_C])),id=Ident[N_C], description='Asti seq no align')
    SeqIO.write(record,f_out,'fasta')

def desalinea(seqs_list):
  d_out = 'No_align.fasta'
  f_out = open(d_out,'w')
  seqs = []
  ids = Ident(seqs_list)
  for j in range(len(seqs_list)):
    seqs.append(quita_gaps(seqs_list[j].seq))
  escribe_fasta(seqs, ids, f_out)
  return d_out

def alinea(f_in):
  output = 'Alineados.txt'
  sub.run(['ANARCI', '-i', f_in, '-o', output, '-r', 'heavy', '-s', 'a', '--use_species', 'human',
    '--ncpu', '12'], capture_output = True)
  sub.run(['rm', f_in])
  return(output)

def ANARCI_fasta(align_in):
  with open(align_in, 'r') as file:
    Seqs = []
```

```
    Iden = []
    imp_iden = []
    seq = ''
    for line in file:
      splited_line = line.split()
      if len(splited_line) == 6:
        prov_iden = splited_line[1]
      if splited_line[0] == 'H':
        seq += splited_line[2]
      if splited_line[0] == '//':
        if len(seq) == 0 : imp_iden.append(prov_iden)
        if len(seq) <= 149 and len(seq) > 0:
          seq = seq + (149 - len(seq))*'-'
          Seqs.append(seq)
          Iden.append(prov_iden)
        seq =''
  sub.run(['rm', align_in])
  f_out = open('Asti_Aligned.fasta','w')
  print(len(Iden))
  print(len(imp_iden))
  escribe_fasta(Seqs, Iden, f_out)
  return imp_iden

def extrae_data(ident_imp, ref_data):
  ident_tot = []
  final_items = []

  for item in ref_data:
    ident_tot.append(item.id)

  for id_c in ident_imp:
    ind = ident_tot.index(id_c)
    final_items.append(ref_data[ind])
  return final_items

# seqs_list = list(SeqIO.parse(open('./germ_clust.fasta'),'fasta'))
# seqs_list = list(SeqIO.parse(open('./hiper_mut_clust.fasta'),'fasta'))
seqs_list = list(SeqIO.parse(open('../Extrae_cluster_raro/filtradas_all_yes.fasta')
    ,'fasta'))

print('Seqs leidas:', len(seqs_list))

f_in = desalinea(seqs_list)
print('Seqs desalineadas')

txt_align = alinea(f_in)
print('Seqs realineadas')
imp_ident = ANARCI_fasta(txt_align)

imp_final = extrae_data(imp_ident, seqs_list)
print('Impossible to align: ', len(imp_final))

# file_imp = 'imp_to_align.fasta'
# with open('hiper_mut_clust.fasta', 'w') as file:
#   SeqIO.write(imp_final, file, 'fasta')
```

6.2 Sequence generation.

In this annex the codes used to generate the different types of data bases will be presented.

6.2.1 Gaussian data bases.

To generate the data sets of Gaussian sequences there are two possible options depending on how we want to generate them. If you want to generate a large number of sequences at once or if you want to generate small groups of sequences. If you are in the first case, you should use the python code and if you are in the second case, the julia code. This is because the time it takes Julia to generate sequences grows linearly with the number of sequences, but in the case of python it is not like that, it takes much more time than Julia to generate a small number of sequences, but as the time it takes does not scale linearly with the number of sequences, it ends up being much faster generating large blocks of sequences.
The Python script:

```
import numpy as np
import sys
from scipy.stats import multivariate_normal
from Bio.Seq import Seq
from Bio import SeqIO
from Bio.SeqRecord import SeqRecord

sys.path.append("/home/david/Desktop/TFM/seqsTFM-base/seqsTFM/")
from transform import *

arg = len(sys.argv)
if len(sys.argv) == 2:
  for i in range(len(sys.argv)):
    Ncad = int(sys.argv[1])
else:
  print('Num de args incorrecto.')
  print('Usage: ')
  print('GenSem.py Ncad')
  exit()

print('Secuencias a generar: ', Ncad)


###### Funciones ######

def gen_semilla(cov, mean, Nsem, Limite, Lcad):
  '''
  A partir de la matriz inversa de la covarianza y de la media, genera una
  una distribucion gaussiana multivariada con la cual genera las semillas.
  '''
  cadbin = np.random.multivariate_normal(mean, cov, Nsem)
  cadascii = []
  ident = []
  auxascii = np.zeros((Nsem, Lcad), dtype = np.uint8)
  for j in range(Nsem):
    auxascii[j] = binary_to_uint8(floattoint(cadbin[j], Limite))
    cadascii.append("".join([(chr(a)) for a in auxascii[j]]))
```

```
    ident.append(str(j))
  return cadascii, ident

##### Parametros ######

Limite = 0.367
Lcad = 298

##### Main #####

 ref  = np.loadtxt('./Modelo/MedBin.txt')
 ref  = np.loadtxt('./Modelo/CovBin.txt')
print('Modelo cargado')

cadascii, ident = gen_semilla(cov =  ref , mean =  ref , Nsem = Ncad, Limite =
    Limite, Lcad = Lcad)
print('Semilla generada')

with open('Gauss_seqs_' + str(Ncad) + '.fasta', 'w') as file:
  escribe_fasta(cadascii, ident, file)
```

The Julia script:

```
using LinearAlgebra
using Random
using Distributions

function gen_obj_gauss( ref ,  ref)

  cov = ( ref +  ref')/2
  Dist = MvNormal( ref , cov)
  return (Dist)
end

function gen_delt_sem(delt_N, LcadB, Dist)

  CadenaP = Array{Float64}(undef, LcadB)
  Cadena = Matrix{Float64}(undef, delt_N, LcadB)
  for i in 1:delt_N
    rand!(Dist,CadenaP)
    Cadena[i, :] = CadenaP
  end
  return Cadena
end
```

6.2.2 IGoR data bases.

To generate the IGoR databases, as already mentioned in the Section 2.4 section, there are several steps and therefore several programs are needed. The first of them is the main script which is supported by the two scripts that will be presented later. In this main script are the necessary functions to not only generate the nucleotide sequences with IGoR, but also align them with ANARCI.

```
import subprocess as sub
import numpy as np
import csv
```

```
from Bio import SeqIO

def genera_seqs():
  sub.run(['igor','-species','human','-chain','heavy_naive'], capture_output = True
    )
  sub.run(['igor','-generate',str(len_bloque),'-set_wd','./'], capture_output =
    True )

def comprueba():
  with open('./generated/generated_seqs_werr.csv', 'r') as file:
    reader = csv.reader(file)
    a = int(len(list(reader)))
    if a > len_bloque:
      return (True)
    else:
      return (False)

def csv_to_fasta(i):
  output = 'Igor_S' + str(i) + '.fasta'
  sub.run(['python3', 'Igor_Trl.py','./generated/generated_seqs_werr.csv', output],
     capture_output = True)
  return(output)

def alinea(f_in, i):
  output = 'Igor_Aling' + str(i) + '.txt'
  sub.run(['ANARCI', '-i', f_in, '-o', output, '-r', 'heavy', '-s', 'a',
    '--ncpu', '12', '--use_species', 'human'],capture_output = True)
  sub.run(['rm', f_in])
  return(output)

def ANARCI_fasta(align_in, i):
  output =  'Igor_Aling' + str(i) + '.fasta'
  sub.run(['python3', 'ANARCI_txt.py',
    align_in, output], capture_output = True)
  sub.run(['rm', align_in])


long_base = 2000000
len_bloque = 10000
n_files = range(0,int(long_base/len_bloque) + 1)
print('Archivos a generar: ', int(long_base/len_bloque) + 1)


for i in (n_files):
  while True:
    genera_seqs()
    if comprueba() == True:
      break
  f_out = csv_to_fasta(i)
  align_out = alinea(f_out, i)
  ANARCI_fasta(align_out, i)
  print('Avance: ', np.round((i/len(n_files))*100), '%', end = '\r')

with open("/home/david/Desktop/TFM/Igor_Gen_Data/Igor_50k.fasta", "a") as final:
  for j in n_files:
    try:
      file_in = 'Igor_Aling' + str(j) + '.fasta'
      record = SeqIO.parse(file_in,'fasta')
      SeqIO.write(record,final,'fasta')
```

```
      sub.run(['rm', file_in])
    except:
      print('File ', j, 'not found.')
```

In the other two scripts, the sequences are translated and changes are made to their format so one program can correctly receive the output of another. Specifically, the first program receives as input the csv that has IGoR as output, translates the sequences and converts them into fasta format, which is the output format. The second program only makes a format change: from the txt type that has ANARCI as output to fasta format to store the sequences correctly.

```
#A partir de un archivo CSV salido de Igor construye un archivo fasta con las
    secuencias ya traducidas.

from Bio.Seq import Seq
from Bio import SeqIO
from Bio.SeqRecord import SeqRecord
import pandas as pd
import numpy as np
import csv
import sys

N_Arg = len(sys.argv)
if N_Arg == 3:
  File_Ini = sys.argv[1]
  File_Output = sys.argv[2]
else:
  print('Igor_Trl.py Seq_File Input Output')
  print('Seq_File: file which contains the sequences.')
  print('Output File.')
  exit()

def Escribe_Fasta(Chains,Ident,File):
  ''' A partir de dos listas, una con las cadenas y otra con sus identificadores,
  crea un archivo fasta'''
  for N_C in range(len(Chains)):
    record = SeqRecord(Seq(str(Chains[N_C])),id=Ident[N_C],
      description="Generated by Igor")
    #Une las listas de cadenas e identificadores para pasarlas al formato de dato
    #que usa biopython y poder usar con ellas sus funciones.
    SeqIO.write(record,File,'fasta')


def CSV_to_Fasta(File_Ini):
  '''A partir de un archivo CSV con las secuencias crea uno fasta'''
  Chains_A = []
  Ident_A = []
  #Listas donde almacenamos las secuencias de ARN y sus identificadores.
  with open(File_Ini, 'r') as file:
    #F.csv es un archivo CSV con las cadenas de Igor.
    reader = csv.reader(file)
    next(reader, None)
    #Saltamos la cabecera que le pone Igor a los archivos de secuencias.
    for row in reader:
      row = row[0].split(";")
      Ident_A.append(Genera_Iden_ARN(row[0],row[1]))
      #La cabecera ser  un contador y la longitud de la secuencia.
      Chains_A.append(row[1])
```

```
  return(Chains_A,Ident_A)

def Genera_Iden_ARN(Cont,Chain):
  '''
  Will create every chain identification.
  '''
  A_Len = len(Chain)
  #Longitud de la cadena.
  if A_Len % 3 == 0:
    Prod = 'P'
  else:
    Prod = 'NP'
  Iden = Cont+'|'+str(A_Len)+'|'+str(Prod)
  return(Iden)
  #La cabecera ser  un contador y la longitud de la secuencia.

def Translate_ARN(Chains_A,Ident_A):
  '''A partir de las cadenas ARN produce un archivo fasta con las cadenas
  de aminoacidos'''
  Chains_P = []
  Ident_P = []
  #Listas donde almacenamos las secuencias de proteinas y sus identificadores.
  Cont = 0
  #Util para las cabeceras de las secuencias.
  for Chain in Chains_A:
    P_len = len(Chain)
    #Longitud de la cadena.
    Chains_P.append(Seq(Chain).translate(to_stop=True))
    #La cabecera ser  un contador y la longitud de la secuencia.
    Ident_P.append(str(Cont)+'|'+str(P_len))
    Cont += 1
  Prot_F = open(File_Output,'w')
  Escribe_Fasta(Chains_P,Ident_P,Prot_F)

Chains,Ident = CSV_to_Fasta(File_Ini)
Translate_ARN(Chains,Ident)
```

```
from Bio.Seq import Seq
from Bio import SeqIO
from Bio.SeqRecord import SeqRecord
import pandas as pd
import numpy as np
import csv
import sys


N_Arg = len(sys.argv)
if N_Arg == 3:
  File_Ini = sys.argv[1]
  Output = sys.argv[2]
else:
  print('Trans_Def.py Seq_File \t Chain_File_Type \t Output_File')
  print('Seq_File: file which contains the sequences.')
  exit()

def Escribe_Fasta(Chains,Ident,Output):
  File_O = open(Output,'w')
  for N_C in range(len(Chains)):
    record = SeqRecord(Seq(str(Chains[N_C])),id=Ident[N_C],
```

```
      description="Generated by Igor")
    #Do not change the description if then Igor_Trl.py is going to be launched.
    SeqIO.write(record,File_O,'fasta')

with open(File_Ini, 'r') as file:
  Seqs = []
  Iden = []
  Cont = 0
  Esc = 0
  P = 0
  for line in file:
    split_line = line.split()
    if len(split_line)>2 and split_line[2] == 'Generated':
      Marca = Cont + 5
      P_Iden = split_line[0] + split_line[1]
      P_Chain = ''
    if Cont == Marca and line.split('|')[1] == 'human':
      Esc = 1
      P += 1
    if split_line[0] == 'H':
      P_Chain += split_line[2]
    if split_line[0] == '//':
      if Esc == 1 and len(P_Chain) <= 149:
        Seqs.append(P_Chain + (149 - len(P_Chain))*'-')
        Iden.append(P_Iden)
        Esc = 0
    Cont += 1
print(len(Seqs),'\t', P)
Escribe_Fasta(Seqs,Iden,Output)
```

6.3 Null columns calculation.

In this annex is presented a Python script with the following functionalities:

- Columns which contains all the possible amino acids calculation.
- Null columns calculation.
- Comparative between the null columns of a reference data set and another data set.
- Variability column calculation.

```
import numpy as np
import numba as nb
import sys
from Bio import SeqIO
import matplotlib.pyplot as plt

sys.path.append("../seqsTFM-base/seqsTFM/")
from transform import ascii_to_binary_vect, ref_seqs_mat

amin_dict_8 = np.asarray([65, 67, 68, 69, 70, 71, 72, 73, 75, 76, 77, 78, 80, 81,
    82, 83, 84, 86, 87, 89])

########## Functions ##########

```

```
def all_amin_ascii(asti_mat):
  '''
  Returns the number of columns containing all amino acids.
  '''
  ncad, lcad = asti_mat
  asti_mat_T = asti_mat.T
  recuento = np.zeros(lcad)
  for pos in range(lcad):
    if check_all_amin(asti_mat_T[pos]) == len(amin_dict_8):
      recuento[pos] = 1
  return recuento

def check_all_amin(asti_mat_T_i):
  '''
  Check if a column contains all amino acids.
  '''
  count = 0
  for amin in amin_dict_8:
    if amin not in asti_mat_T_i:
      break
    count += 1
  return count

def all_col_0(seqs_mat_B):
  '''
  Returns an array containig the null columns.
  '''
  nseq, lcad_B = seqs_mat_B.shape
  seqs_mat_B = seqs_mat_B.T
  filas_0 = []
  for fila in range(lcad_B):
    if check_col(seqs_mat_B[fila], nseq) == nseq:
      filas_0.append(fila)
  return np.asarray(filas_0)

@nb.njit(parallel = False, fastmath = True)
def check_col(col, nseq):
  '''
  Counts the number of 0's in a column.
  '''
  count = 0
  for i in range(nseq):
    if col[i] == 0:
      count += 1
  return count

def no_cont(filas_ref, filas_p):
  '''
  Returns a vector with the rows that are null in rows_p, but not null in rows_ref.
  '''
  no_cont = []
  for i in range(len(filas_p)):
    cont = False
    for j in range(len(filas_ref)):
      if int(filas_p[i]) == int(filas_ref[j]):
        cont = True
        break
    if cont == False:
      no_cont.append(filas_p[i])
```

```
  return np.asarray(no_cont)

def comp_varia(ref_mat_B, col):
  '''
  Returns the variability of the columns contained in a vector.
  '''
  nseq, lcad_B = ref_mat_B.shape
  ref_mat_B = ref_mat_B.T
  cons = np.zeros(len(col))
  for i in range(len(col)):
    cons[i] = check_col(ref_mat_B[col[i]], nseq)/nseq
  return cons

def plot_var(no_comun, cons, nseqs_ref):
  plt.scatter(no_comun, 1 - cons)
  plt.axhline(y = 1/nseqs_ref, color='r', linestyle=':')
  plt.xlabel('No coincident columns')
  plt.ylabel('Variability in ref. seqs.')
  plt.show()

def save_data(no_comun, cons, data_file):
  data = np.zeros((2, len(no_comun)))
  data[0] = no_comun
  data[1] = cons
  data_out = data_file.split('.')[0] + '.txt'
  with open('./Data/Var/Var_' + data_out, 'w') as out:
    np.savetxt(out, data)

########## Main code ############

#### Data uploading ####
data_file = 'Igor_1M_798k.fasta'
seqs = list(SeqIO.parse(open('../DataBases/Igor/' + data_file),'fasta'))
seqs_mat_B = ascii_to_binary_vect(ref_seqs_mat(seqs))
print('Seq leidas:', len(seqs))

ref_seqs = list(SeqIO.parse(open('../DataBases/Bases_Asti/S1yS2_realign/
    Asti_Aligned_S1.fasta'),'fasta'))
ref_mat_B = ascii_to_binary_vect(ref_seqs_mat(ref_seqs))
print('Seq ref:', len(ref_seqs))

#### Calculation of null rows  ####
filas_0_seqs = all_col_0(seqs_mat_B)
print('Filas nulas en seqs:', len(filas_0_seqs))

filas_0_ref = all_col_0(ref_mat_B)
print('Filas nulas en ref:', len(filas_0_ref))

#### Common columns ####
no_comun = no_cont(filas_ref = filas_0_ref, filas_p = filas_0_seqs)
print('Filas nulas no comunes:', len(no_comun))

cons = comp_varia(ref_mat_B, no_comun)
# plot_var(no_comun, cons, len(ref_seqs))
save_data(no_comun, cons, data_file)
```

6.4 Large data bases covariance and mean calculation.

In this annex is presented a Python script that contains a class object that is able to calculate the mean and the covariance of a large data set. It can be done also considering a prior distribution for the sequences.

```
import numpy as np
import numba as nb
import sys

sys.path.append("./")
from stats import posterior

####### Classes ########
class System:
  '''
  In this class we will store both the covariance and the mean of a data set.
  of a data set. The key is that it will allow us to update both as we add
    sequences.
  This way the size of the data set is not limited by RAM memory.
  '''
  def __init__(self, Lcad_B):
    '''
    Initial parameters: mean and cov, both initialized to 0 and the binary length
    of the sequences we are going to use.
    '''
    self.cov = np.zeros((Lcad_B, Lcad_B))
    self.mean_t = np.zeros(Lcad_B)
    self.Lcad_B = Lcad_B

  def update_no_prior(self, seqs_N, n_seq_i,  N):
    '''
    seqs_N: sequences to be added to the data set.
    n_seq_i: number of sequences considered in the data set so far.
      N: number of sequences to add to the data set in this step.
    From these 3 things it updates the value of the mean and cov assuming
    a value of    = 0.
    '''
    old_weight,  N_weight = calc_weight(n_seq_i,  N)
    mean_N, cov_intra_N = self.cov_and_mean_N(seqs_N)
    self.update_cov_and_mean(old_weight,  N_weight, cov_intra_N, mean_N)

  def update(self, seqs_N, n_seq_i,  N, old , oldU,  ):
    '''
    seqs_N: sequences to be added to the data set.
    n_seq_i: number of sequences considered in the data set so far.
      N: number of sequences to add to the data set in this step.
    old , oldU: a priori distributions for the mean and cov, respectively.
      : fraction of prior to include.
    From these 5 things it updates the value of the mean and cov.
    '''
    old_weight,  N_weight = calc_weight(n_seq_i,  N)
    mean_N, cov_intra_N = self.cov_and_mean_N(seqs_N)
    Post_mean_N, Post_cov_intra_N = posterior(mean_N, cov_intra_N,  ,
    old , oldU, self.Lcad_B)
    self.update_cov_and_mean(old_weight,  N_weight, Post_cov_intra_N,
    Post_mean_N)

```

```
  def update_cov_and_mean(self, weight_old, weight_N, cov_intra_N, mean_N):
    '''
    Update the mean and covariance from the weights and the intra-group mean and
    covariance of the sequences to be added.
    '''
    self.mean_t = self.mean_t * weight_old + mean_N*weight_N
    dif = (mean_N - self.mean_t).reshape(self.Lcad_B, 1)
    cov_inter_N = dif * dif.T
    self.cov = weight_old * self.cov + weight_N * (cov_intra_N + cov_inter_N)

  def cov_and_mean_N(self, seqs_N):
    '''
    Calculate the mean and intra-group covariance of the sequences to be added to
    the data-set.
    '''
    seqs_T = seqs_N.T
    mean_N = np.zeros(self.Lcad_B)
    for i in range(self.Lcad_B):
      mean_N[i] = np.mean(seqs_T[i])
    cov_N = np.cov(seqs_T, bias = True)
    return(mean_N, cov_N)

@nb.njit(parallel = False, fastmath = True)
def calc_weight(n_seq_i,  N):
  '''
  Calculate the weights based on the number of sequences considered and the number
  of sequences to be added in this step.
  '''
  old_Nseq = n_seq_i -  N
  old_weight = old_Nseq/n_seq_i
   N_weight =  N/n_seq_i
  return(old_weight,  N_weight)
```

6.5 Sequence binary and int8 transformation.

In this annex is presented a Python script that contains all the functions needed to transform the data base format and the sequences structure.

```
import numpy as np
import numba as nb
from Bio.Seq import Seq
from Bio import SeqIO
from Bio.SeqRecord import SeqRecord


aminosdict = ['A','C','D','E','F','G','H','I','K','L','M','N','P','Q',
        'R','S','T','V','W','Y','-']
aminosdictnum = [65, 67, 68, 69, 70, 71, 72, 73, 75, 76, 77, 78, 80, 81,
         82, 83, 84, 86, 87, 89, 45]
aminosdict_np = np.asarray(aminosdict)
aminosdictnum_np = np.asarray(aminosdictnum)

Namin = len(aminosdict) - 1

@nb.njit(parallel = True, fastmath = True)
def floattoint_vect(Cadenas, Limite):
  '''
  Going from numbers between 0 and 1 to 0 or 1 in an array of sequences.
```

```
  '''
  ncad, lcad = Cadenas.shape
  for j in nb.prange(ncad):
    for i in nb.prange(lcad):
      if (Cadenas[j,i] > Limite):
        Cadenas[j,i] = 1
      else:
        Cadenas[j,i] = 0
  return Cadenas

@nb.njit(parallel = False, fastmath = True)
def floattoint(Cadena, Limite):
  '''
  Going from numbers between 0 and 1 to 0 or 1 in a single sequence.
  '''
  lcad = len(Cadena)
  for i in range(lcad):
    if (Cadena[i] > Limite):
      Cadena[i] = 1
    else:
      Cadena[i] = 0
  return Cadena


def escribe_fasta(Chains, Ident, file):
  '''
  From a list of sequences and another list with the identifiers, generate a
  fasta file.
  '''
  for N_C in range(len(Chains)):
    record = SeqRecord(Seq(str(Chains[N_C])),id=Ident[N_C],
      description="Semilla")
    SeqIO.write(record, file, 'fasta')

def ref_seqs_mat(RefSeqsList):
  '''
  Creates an array of reference sequences from a list of reference
  sequences generated by biopython.
  '''
  nrefseqs = len(RefSeqsList)
  lrefseqs = len(RefSeqsList[0].seq)
  Ref_Seqs = np.chararray((nrefseqs,lrefseqs),unicode = True)
  for nseq,ref in enumerate(RefSeqsList):
    Ref_Seqs[nseq] = ref.seq
  return(Ref_Seqs)

@nb.njit(parallel = False, fastmath = True)
def ascii_to_binary(seq):
  '''
  Receives a single sequence in ascii and returns that sequence
  translated to binary
  '''
  bseq = np.zeros(len(seq) * Namin, dtype = np.int8)
  for pos,amin in enumerate(seq):
    if amin != '-':
      for i in range(Namin):
        if amin == aminosdict_np[i]:
          aminc = i
          bseq[pos*Namin + aminc] = 1
```

```
          break
  return(bseq)

@nb.njit(parallel = False, fastmath = True)
def ascii_to_binary_vect(seq):
  '''
  Receives a matrix sequence in ascii and returns that matrix
  translated to binary
  '''
  dim = seq.shape
  bseq = np.zeros((dim[0],dim[1] * Namin), dtype = np.int8)
  for s in range(dim[0]):
    for pos,amin in enumerate(seq[s]):
      if amin != '-':
        for i in range(Namin):
          if amin == aminosdict_np[i]:
            aminc = i
            bseq[s, pos*Namin + aminc] = 1
            break
  return(bseq)

@nb.njit(parallel = False, fastmath = True)
def uint8_to_binary(seq):
  '''
  Recibe una unica secuencia en uint8 y devuelve esa secuencia
  traducidas a binario
  '''
  bseq = np.zeros(len(seq) * Namin, dtype = np.int8)
  for pos,num in enumerate(seq):
    if num != 45:
      for i in range(Namin):
        if num == aminosdictnum_np[i]:
          aminc = i
          bseq[pos*Namin + aminc] = 1
          break
  return(bseq)

@nb.njit(parallel = False, fastmath = True)
def uint8_to_binary_vect(seq):
  '''
  Receives an array in which each row is a sequence in
  uint8 and returns those sequences translated to binary
  '''
  dim = seq.shape
  bseq = np.zeros((dim[0],dim[1] * Namin), dtype = np.int8)
  for s in range(dim[0]):
    for pos,num in enumerate(seq[s]):
      if num != 45:
        for i in range(lbloque + 1):
          if num == aminosdictnum_np[i]:
            aminc = i
            bseq[s, pos*lbloque + aminc] = 1
            break
  return(bseq)


def ascii_to_unit8_vect(seqs):
  '''
  Recibe una matriz en la que cada fila es una secuencia en
```

```
  ascii y devuelve esas secuencias traducidas a uint8
  '''
  dim = seqs.shape
  mat = np.zeros((dim[0],dim[1]))
  for i in range(dim[0]):
    mat[i] = np.asarray(seqs[i], dtype = '|S1').view(np.uint8).astype(int)
  return(mat)

def ascii_to_unit8(seq):
  '''
  Recibe una  secuencia en ascii y devuelve esa secuencia
  traducida a uint8
  '''
  mat = np.asarray(seq, dtype = '|S1').view(np.uint8).astype(int)
  return(mat)


@nb.njit(parallel = False, fastmath = True)
def binary_to_uint8_vect(bseq, Lcad):
  '''
  Recibe una matriz en la que cada fila es una secuencia en
  binario y devuelve esas secuencias traducidas a uint8
  '''
  dim = bseq.shape
  seqs_uint = np.zeros((dim[0],Lcad), dtype = np.uint8)
  for s in range(dim[0]):
    for bloque in range(Lcad):
      one = -1
      for i in range(bloque*Namin, ((bloque + 1)*Namin)):
        if bseq[s, i] == 1:
          one = i - bloque*Namin
          break
      if one == -1:
        seqs_uint[s, bloque] = int(aminosdictnum_np[20])
      else:
        seqs_uint[s, bloque] = int(aminosdictnum_np[one])
  return(seqs_uint)

@nb.njit(parallel = False, fastmath = True)
def binary_to_uint8(bseq):
  '''
  Recibe una secuencia en binario y devuelve esa secuencia
  traducida a uint8
  '''
  Lcad = int(len(bseq)/Namin)
  seq_uint = np.zeros(Lcad, dtype = np.uint8)
  for bloque in range(Lcad):
      one = -1
      for i in range(bloque*Namin, ((bloque + 1)*Namin)):
        if bseq[i] == 1:
          one = i - bloque*Namin
          break
      if one == -1:
        seq_uint[bloque] = int(aminosdictnum_np[20])
      else:
        seq_uint[bloque] = int(aminosdictnum_np[one])
  return(seq_uint)

def binary_to_ascii(seq_b):
```

```
198   '''
199   Recibe una secuencia en binario y devuelve esa secuencia
200   traducida a ascii
201   '''
202   lcad_b = len(seq_b)
203   lcad = int(lcad_b/Namin)
204   seq = np.chararray(lcad, unicode = True)
205   for bloque in range(lcad):
206     ind = np.where(seq_b[bloque*Namin:((bloque + 1)*Namin)] == 1)[0]
207     if ind.size > 0:
208       seq[bloque] = aminosdict[int(ind)]
209     else:
210       seq[bloque] = aminosdict[Namin]
211   return seq
212 
213 def binary_to_ascii_vect(seq_b):
214   '''
215   Recibe una matriz en la que cada fila es una secuencia en
216   binario y devuelve esas secuencias traducidas a ascii
217   '''
218   nseq, lcad_b = seq_b.shape
219   lcad = int(lcad_b/Namin)
220   seq = np.chararray((nseq, lcad), unicode = True)
221   for seq_i in range(nseq):
222     for bloque in range(lcad):
223       ind = np.where(seq_b[seq_i, bloque*Namin:((bloque + 1)*Namin)] == 1)[0]
224       if ind.size > 0:
225         seq[seq_i, bloque] = aminosdict[int(ind)]
226       else:
227         seq[seq_i, bloque] = aminosdict[Namin]
228   return seq
229 
230 def unit8_to_ascii(seq):
231   '''
232   Recibe una unica secuencia en uint8 y devuelve esa secuencia
233   traducidas a ascii
234   '''
235   lcad = len(seq)
236   mat = np.chararray(lcad, unicode = True)
237   for i in range(lcad):
238     mat[i] = chr(seq[i])
239   return(mat)
240 
241 def unit8_to_ascii_vect(seq):
242   '''
243   Recibe una matriz en la que cada fila es una secuencia en
244   ascii y devuelve esas secuencias traducidas a uint8
245   '''
246   nseq, lcad = seq.shape
247   mat = np.chararray((nseq,lcad), unicode = True)
248   for j in range(nseq):
249     for i in range(lcad):
250       mat[j, i] = chr(seq[j, i])
251   return(mat)
```

6.6 d_{ant}, d_{inv} and covariance norm calculation.

In this annex is presented a Python script that calculates d_{ant}, d_{inv} and covariance norm calculation while adding sequences to the data set as is described in Section 2.5.

```
import numpy as np
import numba as nb
from Bio import SeqIO
import matplotlib.pyplot as plt
import sys

sys.path.append("/home/david/Desktop/TFM/seqsTFM-base/seqsTFM/")
from large_cov import *
from transform import ref_seqs_mat, ascii_to_binary_vect, floattoint_vect
from stats import prior,

############## Functions #############

def carga_seq(n_seq_i):
  '''
  Add sequences from the sequences list.
  '''
  if Igor:
    if n_seq_i == Nseq_min:
       N_seqs = ascii_to_binary_vect(ref_seqs_mat(seqs_list[:n_seq_i]))
    else:
       N_seqs = ascii_to_binary_vect(ref_seqs_mat(seqs_list[(n_seq_i -  N ):
    n_seq_i]))
  if Julia_C:
    if n_seq_i == Nseq_min:
       N_seqs = Main.gen_delt_sem(n_seq_i, Lcad_B, gauss_obj)
    else:
       N_seqs = Main.gen_delt_sem( N , Lcad_B, gauss_obj)
    if Discreto:
       N_seqs = floattoint_vect( N_seqs , Limite)

  return( N_seqs)

def cov_ref_fija(data, cov_i, ind):
  '''
  Chech if the covariance matrix is invertible.
  '''
  try:
    np.linalg.inv(cov_i)
  except:
    return None, None, True
  else:
    return cov_i, cov_i, False

@nb.njit(parallel = False, fastmath = True)
def calcula_dists(cov_i, cov_ref, cov_ant):
  '''
  Calculates d1_ant, d2_ant, d1_inv, d2_inv and the covariance norm.
  '''
  norm_ant = np.linalg.norm(cov_ant)
  d_ant = np.linalg.norm(cov_i - cov_ant)/norm_ant
  norm_ref = np.linalg.norm(cov_ref)
  d_ref = np.linalg.norm(cov_i - cov_ref)/norm_ref
  d_t = np.linalg.det((cov_i + cov_i.T)/2)
```

```
  d_i = np.linalg.norm(cov_i)
  return d_ref, d_ant, d_i, d_t

def itera(  , old , oldU):
  data = np.zeros((5, N_pasos + 1))
  cov_ref = None
  check = True
  cov_t = System(Lcad_B = Lcad_B)

  for n_seq_i in range(Nseq_min, Nseq +  N ,  N ):

    print('Trabajando con: ', n_seq_i, 'seqs.', end = '\r')
    ind = int((n_seq_i-Nseq_min)/ N )
    data[0, ind] = n_seq_i

     N_seqs = carga_seq(n_seq_i)
    cov_t.update( N_seqs , n_seq_i,  N , old , oldU,  )
    if check:
      cov_ref, cov_ant, check = cov_ref_fija(data, cov_t.cov, ind)
    else:
      data[1, ind], data[2, ind], data[3, ind], data[4, ind] = calcula_dists(cov_t.
   cov, cov_ref, cov_ant)
      cov_ant = cov_t.cov
  return data

########### Parametros ############

Igor = False
if not Igor:
  Julia_C = True
else:
  Julia_C = False

Discreto = True
Limite = 0.367

 N  = 10000
Nseq_min = 10000

   = 0.01

########### Main code ############
if Julia_C:
  Lcad_B = 2980
  Lcad = 149
  Nseq = 1300000
  print('Max seq: ', Nseq)

  from julia.api import Julia
  jl = Julia(compiled_modules = False)
  from julia import Main

  jl = Main.include('../Opt_lambda/GenSem.jl')

   ref  = np.loadtxt('../DataBases/Modelo/Ref/MedBin.txt')
   ref  =  ref[:2980]
   ref  = np.loadtxt('../DataBases/Modelo/Ref/CovBin.txt')
   ref  =  ref[:2980, :2980]
  print( ref.shape,  ref.shape)
```

```
   ref_norm = np.linalg.norm( ref)
   ref_norm = np.linalg.norm( ref)
  print('Modelo cargado')

  gauss_obj = Main.gen_obj_gauss( ref ,  ref)
  print('Obj. gauss generado')

if Igor:
  seqs_list = list(SeqIO.parse(open('../DataBases/Igor/1M_354k.fasta'),'fasta'))
  Lcad = len(seqs_list[0].seq)
  Lcad_B = Lcad*20
  Nseq = len(seqs_list)
  print('Max seq: ', Nseq)

old , oldU = prior(Lcad, Lcad_B)
print('Prior calculada')

N_pasos = int((Nseq - Nseq_min)/ N ) + 1
print('Num. pasos: ', N_pasos)

data = itera(  , old , oldU)
h = data.shape[1] - 1

plt.plot(data[0, 1:h], data[1, 1:h], label = 'Norm_ref')
plt.plot(data[0, 1:h], data[2, 1:h], label = 'Norm_ant')
plt.xlabel('Nseq')
plt.legend()
plt.show()
plt.plot(data[0, 1:h], data[3, 1:h])
plt.plot(data[0, 1:h], data[4, 1:h])

with open('Data.txt', 'w') as file:
  np.savetxt(file, data)

plt.xlabel('Nseq')
plt.ylabel('Cov Norm')
plt.show()
```

6.7 Nucleotide sequence extraction

In this annex is presented two Python scripts. The first one is able to extract the nucleotide sequences from the data obtained from [1]. The second one reproduce the Figure 7 of [1] and divide the sequences into three clusters. This three clusters are the high density zones presented in Section 3.3.

```
from Bio.Seq import Seq
from Bio import SeqIO
from Bio.SeqRecord import SeqRecord
import numpy as np


def escribe_fasta(Chains, Ident, file):
  '''
  From a list of sequences and another list with the identifiers, generate a
  fasta file.
  '''
  for N_C in range(len(Chains)):
```

```
    record = SeqRecord(Seq(Chains[N_C]), id = Ident[N_C], description = "Asti")
    SeqIO.write(record, file, 'fasta')

def cabecera(line_sp, ident, ref_dict):
  '''
  Comprueba si la linea es una cabecera del tipo 'Query=....' y si
  esta contenida en la referencia.
  '''
  if line_sp[0] == 'Query=':
    ident = line_sp[1]
    if ident.split(':')[0] in ref_dict:
      ident = line_sp[1]
    else:
      ident = None
    return ident
  else: return ident

def lee_file(file_name, ref_dict):
  with open(file_name, 'r') as file:
    check = False
    ident = None
    cab = 0
    seq = ''
    IGHV1_2_ident = []
    seq_list = []
    ident_list = []
    for line in file:
      line_sp = line.split()
      if len(line_sp) > 0:
        ident = cabecera(line_sp, ident, ref_dict)
        if line_sp[0].split('*')[0] == 'IGHV1-2' and line_sp[2].split('*')[0] == '
    IGHJ2' and line_sp[6] == 'Yes':
          if ident != None:
            check = True

        if check:
          if len(line_sp) > 4:
            if line_sp[0] == 'Total':
              # print(line_sp)
              IGHV1_2_ident.append(line_sp[7])
            if line_sp[0].split('_')[0] == 'Query':
              seq += line_sp[2]

            if line_sp[0] == 'Effective':
              check = False
              seq_list.append(seq)
              ident_list.append(ident)
              seq = ''

    file.close()
    print('secuencias = ', len(seq_list))
    print('ident = ', len(ident_list))
    print(len(IGHV1_2_ident))
  return seq_list, ident_list, IGHV1_2_ident

def cambia_cab(ini_list):
  '''
  Modifica las cabeceras para que no aparezca la multiplicidad.
  '''
```

```
  cab_new = []
  for item in ini_list:
    cab = item.id.split(':')[0]
    cab_new.append(cab)
  return cab_new

def filtro_buenas(ref_list):
  '''
  Secuencias del archivo de Asti buenas.
  '''
  i = 0
  for item in ref_list:
    if item.id.split('.')[0] == 'SRR277211':
      i += 1
  print('Fitradas:',i)
  return i

def comprueba_mult(ref_list):
  '''
  Checkea cuantas secuencias tienen multiplicidad 1 y cuanto
  suma la multiplicidad total del data base.
  '''
  i = 0
  j = 0
  ref_cab_mult = []
  ref_cab_no_mult = []
  for item in ref_list:
    item_s = item.id.split(':')
    if item.id.split('.')[0] == 'SRR277211':
      i += int(item_s[1])
      ref_cab_mult.append(item.id)
      ref_cab_no_mult.append(item_s[0])
      if int(item_s[1]) == 1:
        j += 1
  print('Con multiplicidad 1:', j)
  print('Mult. total:',i)
  return ref_cab_mult, ref_cab_no_mult

def sustituye_cab(ref_cab, ig_cab, no_mult_cab):
  cab = []
  for item in ig_cab:
    temp = item.split(':')[0]
    if temp in no_mult_cab:
      for ide in ref_cab:
        if ide.split(':')[0] == temp:
          cab.append(ide)
  return cab

def genera_base_not_unique(ident_list, seq_list, IGHV1_2_ident):
  ident_list_fin = []
  seq_list_fin = []
  for j,item in enumerate(ident_list):
    rep = int(item.split(':')[1])
    for i in range(rep):
      ident_list_fin.append(ident_list[j] + '_' + IGHV1_2_ident[j])
      seq_list_fin.append(seq_list[j])
  print('n_cad_final = ',len(ident_list_fin))
  return ident_list_fin, seq_list_fin

```

```
def a ade_eq(ident_list, eq_list):
  for i in range(len(ident_list)):
    ident_list[i] += '_' + eq_list[i]
  return ident_list

########################################################################
################### Main Code ##########################################
########################################################################

ref_file = '../DatosAstati/ModAsti/allYES_uniqueAA_c90-140_IGHV1-2
    _IGHJ2_SEQMA45_3SE9GG_reH2_allSEQMA0.afasta'
ref_dict = SeqIO.to_dict(SeqIO.parse(open(ref_file),'fasta'))
print('File Asti:', len(ref_dict))
ref_list = list(SeqIO.parse(open(ref_file),'fasta'))

new_id = cambia_cab(ref_list)

buenas = filtro_buenas(ref_list)

ref_cab_mult, ref_cab_no_mult = comprueba_mult(ref_list)

file_name = '../DatosAstati/DataBruscolini3/unique_INVCOMPL_NOBAR_all.igblast'
# file_name = 'Igblast_cor.txt'
seq_list, ident_list, IGHV1_2_ident = lee_file(file_name, new_id)
print(len(ident_list))
print(len(IGHV1_2_ident))


ident_list_f = a ade_eq(ident_list, IGHV1_2_ident)
file_out = 'Unique_Fig_7.fasta'
with open(file_out, 'w') as file:
  escribe_fasta(seq_list, ident_list_f, file)

final_cab = sustituye_cab(ref_cab_mult, ident_list, ref_cab_no_mult)
print(len(final_cab))
ident_list_mult, seq_list_fin = genera_base_not_unique(final_cab, seq_list,
    IGHV1_2_ident)
file_out = 'Not_Unique_Fig_7.fasta'
with open(file_out, 'w') as file:
  escribe_fasta(seq_list_fin, ident_list_mult, file)
```

```
from Bio import SeqIO
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import seaborn as sn
from Bio.SeqRecord import SeqRecord
from Bio.Seq import Seq

class Sequence:

  def __init__(self, idt, seq):
    self.id = idt
    self.seq = seq
    self.IGHV1_2 = 0
    self.VRC_PG04 = 0

  def add_sim_IGHV1_2(self, sim_IGHV1_2):
    self.IGHV1_2 = sim_IGHV1_2
```

```
19
20    def add_sim_VRC_PG04(self, sim_VRC_PG04):
21      self.VRC_PG04 = sim_VRC_PG04
22
23  def minus_to_mayus(ref_seq):
24    mayus = ''
25    for i in ref_seq:
26      if i == 'a': mayus += 'A'
27      elif i == 'c': mayus += 'C'
28      elif i == 'g': mayus += 'G'
29      elif i == 't': mayus += 'T'
30      else: print('Caracter extra o:', i)
31    return mayus
32
33  def extrae_V_sim(seq_list):
34    sim = np.zeros(len(seq_list))
35    for j, item in enumerate(seq_list):
36      sim[j] = float((item.id.split('_')[1]).split('%')[0])/100
37    return sim
38
39  def list_data(seq_list, IGHV1_2_ident, VRC_PG04_ident):
40    items = []
41    for i in range(len(seq_list)):
42      item = Sequence((seq_list[i].id).split('_')[0], seq_list[i].seq)
43      item.add_sim_IGHV1_2(IGHV1_2_ident[i])
44      item.add_sim_VRC_PG04(float(VRC_PG04_ident[i]))
45      items.append(item)
46    return items
47
48  def Heat_map(IGHV1_2, VRC_PG04):
49    """
50    Calcula el mapa de calor.
51    """
52    mat = np.zeros((2, len(IGHV1_2)))
53    mat[0] = IGHV1_2
54    mat[1] = np.asarray(VRC_PG04)
55    sn.heatmap(mat, cmap='Blues')
56
57  def ah_shit_here_we_go_again(file_name):
58    ident_list = []
59    ident = None
60    nuc_ident = []
61    sim = np.zeros(2)
62    with open(file_name, 'r') as file:
63      for line in file:
64        line_sp = line.split()
65        if len(line_sp) > 0:
66
67          if line_sp[0] == 'Query=':
68            ident_list.append(line_sp[1])
69
70          if len(line_sp) > 6 and line_sp[0] == 'Identities':
71            if ident == ident_list[-1]:
72              div = line_sp[2].split('/')
73              sim[0] += int(div[0])
74              sim[1] += int(div[1])
75            else:
76              div = line_sp[2].split('/')
77              sim[0] = int(div[0])
```

```
            sim[1] = int(div[1])
            ident = ident_list[-1]

        if line_sp[0] == 'Effective':
          nuc_ident.append(sim[0]/sim[1])

  # print(len(ident_list))
  # print(len(nuc_ident))
  return ident_list, nuc_ident

def Plotea_3D(IGHV1_2_ident, VRC_PG04_ident):

  fig = plt.figure()          #create a canvas, tell matplotlib it's 3d
  ax = fig.add_subplot(111, projection='3d')

  hist, xedges, yedges = np.histogram2d(IGHV1_2_ident, VRC_PG04_ident, bins=(20,20)
    , range = [[0.6, 1], [0.6, 1]])
  xpos, ypos = np.meshgrid(xedges[:-1]+xedges[1:], yedges[:-1]+yedges[1:])

  xpos = xpos.flatten()/2.
  ypos = ypos.flatten()/2.
  zpos = np.zeros_like (xpos)

  dx = xedges [1] - xedges [0]
  dy = yedges [1] - yedges [0]
  dz = hist.flatten()

  cmap = cm.get_cmap('jet') # Get desired colormap - you can change this!
  max_height = np.max(dz)   # get range of colorbars so we can normalize
  min_height = np.min(dz)
  # scale each z to [0,1], and get their rgb values
  rgba = [cmap((k-min_height)/max_height) for k in dz]

  ax.bar3d(xpos, ypos, zpos, dx, dy, dz, color=rgba, zsort='average')
  ax.invert_xaxis()
  plt.xlabel("Nucleotides identity with IGHV1-2*02")
  plt.ylabel("Nucleotides identity with VRC_PG04")
  plt.savefig("Fig_7_Asti.pdf")
  plt.show()

def decide_clust_nuc(item):
  dist_hiper = np.sqrt((0.75 - item.IGHV1_2)**2 + (0.95 - item.VRC_PG04)**2)
  dist_germ = np.sqrt((0.95 - item.IGHV1_2)**2 + (0.725 - item.VRC_PG04)**2)
  dist_inter = np.sqrt((0.75 - item.IGHV1_2)**2 + (0.75 - item.VRC_PG04)**2)
  return ('hiper', 'germ', 'inter')[np.argmin((dist_hiper, dist_germ, dist_inter))]

def div_clust_nuc(data):
  hiper = []
  germ = []
  inter = []
  print('len data:', len(data))
  for ind, item in enumerate(data):
    temp_ident = item.id
    clust = decide_clust_nuc(item)
    if clust == 'hiper':
      if not hiper or temp_ident != hiper[-1].id:
        hiper.append(item)
    elif clust == 'germ':
      if not germ or temp_ident != germ[-1].id:
```

```
        germ.append(item)
    elif clust == 'inter':
      if not inter or temp_ident != inter[-1].id:
        inter.append(item)
  print(len(hiper))
  print(len(germ))
  print(len(inter))
  print()
  return hiper, germ, inter


def extrae_data(items, ref_data, ref_list_cab):
  ident_clus = []
  ident_tot = []
  final_items = []

  for item in items:
    ident_clus.append(item.id)
  for item in ref_list_cab:
    ident_tot.append(item)

  for id_c in ident_clus:
    ind = ident_tot.index(id_c.split(':')[0])
    final_items.append(ref_data[ind])
  return final_items

def escribe_fasta(data, file):
  Chains = []
  Ident = []
  for i in range(len(data)):
    Chains.append(data[i].seq)
    Ident.append(data[i].id)
  print(len(Chains))
  for N_C in range(len(Chains)):
    record = SeqRecord(Seq(str(Chains[N_C])),id = Ident[N_C])
    SeqIO.write(record, file, 'fasta')
def quita_ident(ref_data):
  ident = []
  for item in ref_data:
    ident.append(item.id.split('_')[0].split(':')[0])
  return ident

ref_file = './filtradas_all_yes.fasta'
ref_list = list(SeqIO.parse(open(ref_file),'fasta'))
ref_list_cab = quita_ident(ref_list)

seq_file = './Unique_Fin.fasta'
seq_list = list(SeqIO.parse(open(seq_file),'fasta'))

# ref_mayus = minus_to_mayus(ref_list[1].seq)
# print(ref_mayus)
# exit()
file_name = 'VRC_PG04_align.txt'
ident_list, VRC_PG04_ident = ah_shit_here_we_go_again(file_name)

file_name = 'IGHV1_2_02_align.txt'
ident_list, IGHV1_2_ident = ah_shit_here_we_go_again(file_name)

IGHV1_2_ident_s = extrae_V_sim(seq_list)
```

```
195
196 IGHV1_2_ident = np.asarray(IGHV1_2_ident)
197 VRC_PG04_ident = np.asarray(VRC_PG04_ident)
198
199 data = list_data(seq_list, IGHV1_2_ident, VRC_PG04_ident)
200
201 Plotea_3D(IGHV1_2_ident, VRC_PG04_ident)
202 # exit()
203
204 hiper, germ, inter = div_clust_nuc(data)
205 print(len(hiper + germ + inter))
206 hiper_prot = extrae_data(hiper, ref_list, ref_list_cab)
207 print(len(hiper_prot))
208
209 s = 0
210 i = 0
211 for item in hiper:
212   s += item.IGHV1_2
213   i += item.VRC_PG04
214 print(s/len(hiper))
215 print(i/len(hiper))
216
217 germ_prot = extrae_data(germ, ref_list, ref_list_cab)
218 print(len(germ_prot))
219
220 s = 0
221 i = 0
222 for item in germ:
223   s += item.IGHV1_2
224   i += item.VRC_PG04
225 print(s/len(germ))
226 print(i/len(germ))
227
228 inter_prot = extrae_data(inter, ref_list, ref_list_cab)
229 print(len(inter_prot))
230
231 s = 0
232 i = 0
233 for item in inter:
234   s += item.IGHV1_2
235   i += item.VRC_PG04
236 print(s/len(inter))
237 print(i/len(inter))
238
239
240 file_hiper = 'hiper_mut_clust.fasta'
241 # escribe_fasta(hiper_prot, file_hiper)
242 with open('hiper_mut_clust.fasta', 'w') as file:
243   SeqIO.write(hiper_prot, file, 'fasta')
244
245 file_germ = 'germ_clust.fasta'
246 # escribe_fasta(germ_prot, file_germ)
247 with open('germ_clust.fasta', 'w') as file:
248   SeqIO.write(germ_prot, file, 'fasta')
249
250 file_inter = 'inter_clust.fasta'
251 # escribe_fasta(inter_prot, file_inter)
252 with open('inter_clust.fasta', 'w') as file:
253   SeqIO.write(inter_prot, file, 'fasta')
```

```
254
255 # plt.hist(a, bins = 100, color = 'red')
256 # plt.show()
257 # sim = extrae_V_sim(seq_list)
258 # plt.hist(IGHV1_2_ident, bins = 100, color = 'red')
259 # plt.xlabel('Nucleotides identity with IGHV1-2*02')
260 # plt.show()
261 # plt.hist(VRC_PG04_ident, bins = 100, color = 'red')
262 # plt.xlabel('Nucleotides identity with VRC_PG04')
263 # plt.show()
```

6.8 Kullback-Leibler divergence calculation.

In this annex is presented a Python script that calculates the Kullback-Leibler divergence from one data set to another reference data set.

```
1  from Bio import SeqIO
2  import numpy as np
3  import numba as nb
4  import sys
5  import matplotlib.pyplot as plt
6
7  sys.path.append("/home/david/Desktop/TFM/seqsTFM-base/seqsTFM/")
8  from transform import ascii_to_unit8_vect, ref_seqs_mat
9
10 aminosdictnum = [65, 67, 68, 69, 70, 71, 72, 73, 75, 76, 77, 78, 80, 81,
11          82, 83, 84, 86, 87, 89, 45]
12
13 amin_dict_u8 = np.asarray(aminosdictnum)
14
15 class_input = [('seqs_mat', nb.float64[:, :]), ('n_seq', nb.int64), ('l_seq', nb.
       int64)]
16 @nb.experimental.jitclass(class_input)
17 class seqs_mat:
18
19   def __init__(self, seqs_mat):
20     self.seqs_mat = seqs_mat
21     self.n_seq = seqs_mat.shape[0]
22     self.l_seq = seqs_mat.shape[1]
23
24 @nb.njit(parallel = False, fastmath = True)
25 def rep_amin(col_vect, amin_ref, n_seq):
26   n_rep = 0
27   for amin in col_vect:
28     if amin == amin_ref:
29       n_rep += 1
30   return n_rep/n_seq
31
32 @nb.njit(parallel = True, fastmath = True)
33 def probabilities(seqs_mat_u8):
34   seqs_mat_u8_t = seqs_mat_u8.seqs_mat.T
35   prob_mat = np.zeros((len(amin_dict_u8), seqs_mat_u8.l_seq))
36   for col in range(seqs_mat_u8.l_seq):
37     for ind, amin in enumerate(amin_dict_u8):
38       prob_mat[ind, col] = rep_amin(seqs_mat_u8_t[col], amin, seqs_mat_u8.n_seq)
39   return prob_mat
40
41 @nb.njit(parallel = False, fastmath = True)
```

```
def curva_KL(q, p, l_seq, fallo):
  KL = np.zeros(l_seq)
  h = 0 #Cuenta las columas en las que q_i(aa)=0 pero p_i(aa) no se anula.
  rep_null = np.zeros(l_seq)
  prob_null = np.zeros(l_seq)
  for pos in range(l_seq):
    for amin in range(len(amin_dict_u8)):
      if q[amin, pos] == 0:
        if p[amin, pos] == 0:
          KL[pos] += 0
        else:
          h += 1
          rep_null[pos] += 1
          KL[pos] += p[amin, pos]*np.log(p[amin, pos]/fallo) #si q_i(aa)=0 lo pongo como 1/n_g, siendo n_g el numero de secuencias de la
                                    # germline, para evitar que explote.
          prob_null[pos] += p[amin, pos]
          if p[amin, pos] > 0.25:
            print('null')
            print(p[amin, pos])
            print(pos, amin_dict_u8[amin])
            print()
      else:
        if p[amin, pos] == 0:
          KL[pos] += 0
        else:
          KL[pos] += p[amin, pos]*np.log(p[amin, pos]/q[amin, pos])
  print('fallos:', h)
  return KL, rep_null, prob_null

germ_list = list(SeqIO.parse('../HammingDist/Plots_def/Comp/C2_1.fasta', 'fasta'))
# print(len(germ_list))
germ_seqs_mat = seqs_mat(ascii_to_unit8_vect(ref_seqs_mat(germ_list)))
germ_prob = probabilities(germ_seqs_mat)

inter_list = list(SeqIO.parse('../HammingDist/Plots_def/Comp/C2_2.fasta', 'fasta'))
# print(len(inter_list))
inter_seqs_mat = seqs_mat(ascii_to_unit8_vect(ref_seqs_mat(inter_list)))
inter_prob  = probabilities(inter_seqs_mat)

hiper_list = list(SeqIO.parse('../HammingDist/Plots_def/Comp/C1.fasta', 'fasta'))
# print(len(hiper_list))
hiper_seqs_mat = seqs_mat(ascii_to_unit8_vect(ref_seqs_mat(hiper_list)))
hiper_prob  = probabilities(hiper_seqs_mat)

print('Intermedio:')
KL_1, rep_null_1, prob_null_1 = curva_KL(q  = germ_prob, p = inter_prob, l_seq = germ_seqs_mat.l_seq, fallo = 1/len(germ_list))
print('10 mejores posiciones inter:')
print(np.argsort(KL_1*-1)[:10])

print('Hipermutado:')
KL_2, rep_null_2, prob_null_2 = curva_KL(q  = germ_prob, p = hiper_prob, l_seq = germ_seqs_mat.l_seq, fallo = 1/len(germ_list))
print('10 mejores posiciones hiper:')
print(np.argsort(KL_2*-1)[:10])

x = range(len(KL_1))

```

```
figure_config()
plt.plot(x, KL_1, label = 'C2_2 cluster')
plt.plot(x, KL_2, label = 'C1 cluster')
plt.legend(loc = 'upper right')
plt.xlabel('Sequence possition')
plt.ylabel('Kullback Leibler divergence')
plt.savefig('KL_curve.pdf')
plt.show()

figure_config()
plt.vlines(x=63, ymin=0, ymax=8, color = 'black', label = 'Seq. pos. = 63')
plt.vlines(x=73, ymin=0, ymax=8, color = 'purple', label = 'Seq. pos. = 73')
plt.plot(x[50:80], KL_1[50:80], label = 'C2_2 cluster')
plt.plot(x[50:80], KL_2[50:80], label = 'C1 cluster')
plt.legend(loc = 'upper left')
plt.xlabel('Sequence possition')
plt.ylabel('Kullback Leibler divergence')
plt.savefig('KL_curve_amp.pdf')
plt.show()
```

6.9 Minimize α

In this annex is presented a Python script that was used to calculate the relaxation matrix of the OU process, as it is explained in the Section 3.4.

```
import numba as nb
import numpy as np
import sys
from math import pi as
from math import lgamma
from scipy.optimize import minimize
from scipy.linalg import expm
from Bio import SeqIO

sys.path.append("../seqsTFM-base/seqsTFM")
from transform import ref_seqs_mat, ascii_to_binary_vect
from stats import prior,
from large_cov import *
from plot import plot

###### Classes ######

class Alt_System(System):
  '''
  Heredamos la clase System y a adimos el numero de sequencias
  '''
  def __init__(self, Lcad_B, n_seqs):
    System.__init__(self, Lcad_B)
    self.n_seqs = n_seqs

###### Functions ######

def E_diag(alpha, t):
  D = -alpha*t
  for l in range(Lcad):
    D[l*Q:(l + 1)*Q, l*Q:(l + 1)*Q] = np.exp(D[l*Q:(l + 1)*Q, l*Q:(l + 1)*Q])
  return D

```

```
def ineq_constraint(x):
    """constrain all elements of x to be >= 0"""
    return x

def eq_constraint(x):
    """constrain the sum of all rows to be equal to 1"""
    return x - 1

def from_m_to_v_cajas_sim(m, L, Q):
  triang = int(Q*(Q + 1)/2)
  v = np.zeros(L*triang, dtype = np.float64)
  for i in range(L):
    caja = m[i*Q:(i+1)*Q, i*Q:(i+1)*Q]
    v[triang*i:triang*(i+1)] = caja[np.triu_indices(Q)]
  return v

def from_v_to_m_cajas_sim(v, L, Q):
  triang = int(Q*(Q + 1)/2)
  m = np.zeros((L*Q, L*Q))
  caja = np.zeros((Q, Q))
  for i in range(L):
    caja[np.triu_indices(Q)] = v[i*triang:(i + 1)*triang]
    caja[np.tril_indices(Q, -1)] = caja.T[np.tril_indices(Q, -1)]
    m[i*Q:(i + 1)*Q, i*Q:(i + 1)*Q] = caja
  return m

def carga_seq(n_seq_i, seqs_list,  N , Nseq_min):
  '''
  Carga parte de las secuencias, en lugar de todas, de la lista para evitar
    overfloats.
  '''
  if n_seq_i == Nseq_min:
     N_seqs = ascii_to_binary_vect(ref_seqs_mat(seqs_list[:n_seq_i]))
  elif n_seq_i > len(seqs_list):
     N_seqs = ascii_to_binary_vect(ref_seqs_mat(seqs_list[(n_seq_i -  N ):]))
  else:
     N_seqs = ascii_to_binary_vect(ref_seqs_mat(seqs_list[(n_seq_i -  N ):n_seq_i
    ]))
  return( N_seqs)

def cov_and_mean_with_prior(seqs, n_seq, old , oldU,   ,  N , Nseq_min):
  '''
  Calcula la media y la covarianza de una lista de secuencias, a adiendo una prior
    .
  '''
  cov_and_mean = Alt_System(Lcad_B, n_seq)
  for n_seq_i in range(Nseq_min, n_seq +  N ,  N ):
     N_seqs = carga_seq(n_seq_i, seqs,  N , Nseq_min)
    cov_and_mean.update( N_seqs , n_seq_i,  N , old , oldU,   )
  return cov_and_mean

def cov_and_mean_no_prior(seqs, n_seq,  N , Nseq_min):
  '''
  Calcula la media y la covarianza de una lista de secuencias.
  '''
  cov_and_mean = Alt_System(Lcad_B, n_seq)
  for n_seq_i in range(Nseq_min, n_seq +  N ,  N ):
     N_seqs = carga_seq(n_seq_i, seqs,  N , Nseq_min)
    cov_and_mean.update_no_prior( N_seqs , n_seq_i,  N )
```

```
  return cov_and_mean

def omega_theta_0(IGoR_system, Germ_system):
  init_point = Alt_System(Lcad_B, IGoR_system.n_seqs)
   _0 = IGoR_system.n_seqs
  n_0 = Germ_system.n_seqs
  init_point.mean_t = (n_0/(n_0 +  _0))*Germ_system.mean_t + ( _0/(n_0 +  _0))*
   IGoR_system.mean_t

  d_mean = (Germ_system.mean_t - IGoR_system.mean_t).reshape(Lcad_B, 1)
  init_point.cov = (IGoR_system.cov + n_0*Germ_system.cov + ((n_0* _0)/(n_0 +  _0
   ))*(d_mean*d_mean.T))/( _0 + G + 2)

  return init_point

def extrae_ _t(var_opt):
  result = var_opt.x
  t = result[:2]
   _v = result[2:]
     = from_v_to_m_cajas_sim( _v , Lcad, Q)
  return t,

def save_ _t(t,   ,   ):
  name_t = 't_' + str(int(  )) + '.txt'
  with open(name_t, 'w') as file:
    np.savetxt(file, t)

  name_t = '  _ ' + str(int(  )) + '.txt'
  with open(name_t, 'w') as file:
    np.savetxt(file,   )

def scores_klus_k(  ,   , nseq_k):
   _prima =    + nseq_k
   _prima =    + nseq_k

  ln_num = (G/2)*np.log( _prima) + (G*2)*( _prima + G + 2)*np.log( _prima + G +
   2) - (G/2)*( _prima + G + 2)
  ln_den = (G/2)*np.log(2*  ) + (G* _prima/2)*np.log(2) + lgamma( _prima/2) -
   lgamma(G)
  return(ln_num - ln_den)

def prob_ac(t,   ,   ,   , System_1, System_2, System_3, pre_med, pre_cov,  _ref ,
    _ref):
  s_1 = scores_klus_k(  ,   , System_1.n_seqs)
  s_2 = scores_klus_k(  ,   , System_2.n_seqs)
  s_3 = scores_klus_k(  ,   , System_3.n_seqs)

  F = (G + 2)/2
  print(s_1 + s_2 + s_3)

  ldet_1 = log_det_ _func(  , t[0], pre_cov, pre_med,   , System_1,  _ref ,  _ref
   )
  ldet_2 = log_det_ _func(  , t[1], pre_cov, pre_med,   , System_2,  _ref ,  _ref
   )
  ldet_3 = log_det_ _func(  , 1, pre_cov, pre_med,   , System_3,  _ref ,  _ref)

  ldet_t = F*(ldet_1 + ldet_2 + ldet_3)
  print(ldet_t)
  return (ldet_t, s_1 + s_1 + s_1 - ldet_t)
```

```
142
143 def log_det_ _func(  ,  t , pre_cov, pre_med,   , System_k,  _ref ,  _ref):
144
145   E = expm(-  *  t )
146      =  _ref  - np.matmul(E, np.matmul(pre_cov, E.T))
147
148   pre_med = (pre_med).reshape(Lcad_B, 1)
149      = ( _ref  - np.matmul(E, pre_med).reshape(Lcad_B,))
150
151   inc = (System_k.mean_t -   ).reshape(Lcad_B, 1)
152    _prima =    + System_k.n_seqs*System_k.cov + ((System_k.n_seqs*  )/(System_k.
       n_seqs +   ))*(np.matmul(inc, inc.T))
153
154   return (np.linalg.slogdet( _prima)[1])
155
156 def min_ (variables,   ,  _ref ,  _ref , pre_cov, pre_med, k1_System, k2_System,
       k3_System, germ_System):
157   '''
158   Calcula el score siguiendo la ecuacion 34 y los maximiza buscando maximizar la
       ecuacion 30
159   '''
160   t_1 = variables[0]
161   t_2 = variables[1]
162
163   B = from_v_to_m_cajas_sim(variables[2:], Lcad, Q)
164      = np.matmul(B.T, B)
165
166
167   F = (G + 2)/2
168
169   ldet_1 = log_det_ _func(  , t_1, pre_cov, pre_med,   , k1_System,  _ref ,  _ref
       )
170   ldet_2 = log_det_ _func(  , t_2, pre_cov, pre_med,   , k2_System,  _ref ,  _ref
       )
171   ldet_3 = log_det_ _func(  , 1, pre_cov, pre_med,   , k3_System,  _ref ,  _ref)
172   ldet_4 = log_det_ _func(  , 0, pre_cov, pre_med,   , germ_System,  _ref ,  _ref
       )
173
174   # ldet_t = F*(ldet_1 + ldet_2 + ldet_3)
175   ldet_t = (ldet_1 + ldet_2 + ldet_3 + ldet_4)
176   return  ldet_t
177
178 def itera_ ( _ref ,  _ref , Init_point, k1_System, k2_System, k3_System,
       germ_System):
179   '''
180   Calcula el alpha optimo para cada valor de lambda.
181   '''
182   variables = np.ones(int((Q*(Q + 1)/2)*Lcad + 2))*0.1
183   # variables = np.ones(200 + 2)
184
185   pre_med =  _ref  - Init_point.mean_t
186   pre_cov =  _ref  - Init_point.cov*(G + 2 + Init_point.n_seqs)
187
188   variables[2:] = from_m_to_v_cajas_sim(np.copy( _ref), Lcad, Q) #Inicializamos
       alpha plano.
189
190   bounds_1 = ((1e-10, None), (1e-10, None))
191   bounds_2 = ((None, None), (None, None))*int((Q*(Q + 1)/2)*Lcad/2)
192   # bounds_2 = ((None, None), (None, None))*int(100)
```

```
  bounds = bounds_1 + bounds_2

  # cons = ({'type': 'eq', 'fun': lambda x:  x[5] - 1})

    _v  = np.asarray([1123])

  for     in   _v:

       =     + G - 1

    log_det_ini, score_ini = prob_ac(np.ones(3), np.ones((200,200)),    ,    ,
    k1_System, k2_System, k3_System, pre_med, pre_cov,  _ref ,  _ref)
    print('score_ini =', score_ini, 'log_det_ini = ', log_det_ini)

    var_op = minimize(min_ , (variables), args = (  ,  _ref ,  _ref , pre_cov,
    pre_med, k1_System, k2_System, k3_System, germ_System),  bounds =  bounds,
              options = {"disp":True,'maxfun': 35000, 'maxiter': 5})

    print(var_op.success)
    print(var_op.x[:2])
    t_op, B_op  = extrae_ _ t(var_op)
     _op  = np.matmul(B_op.T, B_op)
    save_ _ t(t_op,  _op ,   )
    log_det_fin, score_fin = prob_ac(t_op,  _op ,    ,    , k1_System, k2_System,
    k3_System, pre_med, pre_cov,  _ref ,  _ref)
    print('times:', t_op)
    print('score_fin =',score_fin)
    print('log_det_fin =',log_det_fin)

####### Parameters #######
G = 200
Q = 20

Lcad = 10
Lcad_B = Lcad*Q

 _prior = 0.01

 N_Igor = 10000
Nseq_min_Igor = 10000

 N_Asti = 100
Nseq_min_Asti = 100

###### Main Code #######

## Carga de datos ##

#Datos Igor#
Igor_list = list(SeqIO.parse(open('../HammingDist/Plots_def/outkluster_75_cort.
    fasta'),'fasta'))

#Germline (cluster CC2_1)#
germ_list = list(SeqIO.parse(open('../HammingDist/Plots_def/Cortadas/CC2_1.fasta'),
    'fasta'))

#Resto de clusters (CC1, CC2_2_1, CC_2_2)#
k1_list = list(SeqIO.parse(open('../HammingDist/Plots_def/Cortadas/CC2_2_1.fasta'),
    'fasta'))
```

```
246 k2_list = list(SeqIO.parse(open('../HammingDist/Plots_def/Cortadas/CC2_2_2.fasta'),
        'fasta'))
247 k3_list = list(SeqIO.parse(open('../HammingDist/Plots_def/Cortadas/CC1.fasta'),'
        fasta'))
248
249 print('Datos cargados.')
250 #Prior plana para calcular la cov de IGoR#
251 old , oldU = prior(Lcad, Lcad_B)
252 print('Prior calculada')
253
254 #Calculo de matrices de covarianza de IGoR y germline#
255 Igor_System = cov_and_mean_with_prior(Igor_list, len(Igor_list), old , oldU,
        _prior , N_Igor , Nseq_min_Igor)
256 Germ_System = cov_and_mean_no_prior(germ_list, len(germ_list), N_Asti ,
        Nseq_min_Asti)
257
258 #Calculo de omega_0 y theta_0#
259 Init_point = omega_theta_0(Igor_System, Germ_System)
260
261 #Calculo de matrices de covarianza resto de clusters (CC1, CC2_2_1, CC_2_2)#
262 k1_System = cov_and_mean_no_prior(k1_list, len(k1_list), N_Asti , Nseq_min_Asti)
263 k2_System = cov_and_mean_no_prior(k2_list, len(k2_list), N_Asti , Nseq_min_Asti)
264 k3_System = cov_and_mean_no_prior(k3_list, len(k3_list), N_Asti , Nseq_min_Asti)
265
266 print('Covarianzas halladas.')
267 itera_ (oldU, old , Init_point, k1_System, k2_System, k3_System, Germ_System)
268 print('Scores optimos hallados.')
```