6 Código desarrollado

El aspecto más triste de la vida en este preciso momento es que la ciencia reúne el conocimiento más rápido de lo que la sociedad reúne la sabiduría.

Isaac Asimov

Una vez finalizado todo el análisis matemático en torno a este proyecto, tanto en este capítulo como en los siguientes se va a dejar de lado las demostraciones teóricas correspondientes al escenario, modelo o algoritmo, para estudiar el código desarrollado en este proyecto. Este código representa el principal esfuerzo invertido a lo largo de la realización de este trabajo, por lo que será necesario mostrar las consideraciones principales y los detalles más importantes con el fin de justificar el tiempo invertido en este apartado. Es importante mencionar que uno de los objetivos, a parte de conseguir entender toda la matemática detrás de este proyecto, consistía en trasladar el código realizado originalmente en *MATLAB* a otro tipo de lenguaje más propio del ML y que, además fuera de código abierto como *Python*. Por tanto, se trata igualmente de un capítulo con una alta relevancia dentro de esta memoria.

Es necesario realizar una serie de comentarios antes de entrar en detalle en el código ya que ayudan a entender la estructuración del código:

- El código está dividido en clases en las cuales se agrupan un porcentaje bastante alto del código completo. Se irá estudiando la estructuración y contenido de cada clase de manera individual.
- Un fichero principal orquestará las llamadas a cada una de las clases. Es importante en este punto tener en cuenta los argumentos de entrada y salida de cada uno de los métodos de cada clase. Se reservará también un espacio dentro de este punto para estudiar el algoritmo de programación desarrollado para orquestar el uso de las clases.
- Los resultados y las gráficas obtenidas gracias a este código serán comentadas en un capitulo posterior.

6.1 Presentación de las clases

Aunque estos apartados podrían ser considerados menos interesante, el autor de esta memoria considera muy importante su introducción ya que forman parte del esfuerzo realizado a la hora de llevar a cabo este trabajo. Es por ello que no se va a detallar en gran manera su contenido y únicamente se van a mencionar las variables y métodos principales dentro de cada clase.

6.1.1 Class *Flags*

Se trata de una clase llamada al comienzo del código correspondiente al *script* que orquesta las llamadas a las clases. En la llamada al constructor de esta clase se va a especificar el valor de las banderas que se quiere para cada uno de los supuestos planteados en este escenario. Esto es, el campo recibido en los sensores varía con el tiempo o es constante, o los sensores se encuentran fijos o desplazando su posición, entre otros.

En la Tabla 6.1, se muestran cada uno de los *flags* que forman parte del código.

En relación a la bandera correspondiente al escenario, se puede apreciar que tiene tres valores distintos, los cuales se describen a continuación:

Tabla 6.1 Banderas usadas en el código.

Bandera	Definición
scenario	Escenario usado (1, 2 ó 3).
flag_tx_est	Estimación de la posición del transmisor (1 ó 0).
flagGPstatic	Uso de algorimto GPR para escenario estático (1 ó 0).
flagGPrecursive	Uso de algoritmo GPR para escenario recursivo (1 ó 0).
flagOKD	Uso del algoritmo OKD. Algoritmo detallado en [39].
flagGholami	Uso de algoritmo "Gholami". Algoritmo detallado en [40].
flagWatcharapan	Uso de algoritmo "Watcharapan". Algoritmo detallado en [41].
flagfigs	Representación de las figuras (1 ó 0).
flagdisp	Representación de los hiperparámetros estimados del kernel (1 ó 0).
flagGridFix	Posición fija de los nodos (1 ó 0).
optimiseHyperparameters	Los hiperparámetros se obtienen del Kernel (True) o de los datos (False).
GPRegression	El modelo usado es GP para regresión (1 ó 0).
GPRegression_inducing	El modelo usado es GP para regresión con *inducing inputs* (1 ó 0).

1. Todos los sensores están estáticos.
2. Todos los sensores se están moviendo.
3. Todos los sensores están estáticos pero no están siempre transmitiendo.

Además, se observa que aparece un término nuevo relacionado con el algoritmo de GPR usado el cual recibe el nombre de *inducing inputs*. Se trata de una mejora del algoritmo de GPR que reduce el tiempo de computación y cuyas prestaciones y base teórica se comentarán en un capítulo posterior.

En función del valor de cada bandera, el escenario o el algoritmo usado puede variar obteniendo unos resultados diferentes en cada caso. De esta manera, se pueden simular distintos supuesto únicamente modificando el valor de la bandera y sin la necesidad de realizar grandes cambios en el código.

6.1.2 Class *Variables*

Al igual que ocurría para las banderas, existen una serie de variables o parámetros cuyos valores son constantes a lo largo de la ejecución completa del código. Por tanto, bastará con inicializarlos al comienzo del código. Estos parámetros corresponden a términos dentro del modelo cuyo valor no se estima sino que permanece fijo.

En la Tabla 6.2 se detallan cada una de estos parámetros.

El valor de la mayoría de estas constantes fueron especificados en los capítulos anteriores en los que se explicaba la procedencia e importancia de cada parámetros así como su valor. Se considera de gran importancia especificar correctamente el valor de cada variable a la hora de llamar al constructor de la clase ya que podrían inicializarse de manera errónea estos parámetros dando lugar a resultados no esperados.

6.1.3 Class *setup*

Esta clase da un giro respecto a la dinámica de las anteriores clases ya que no es utilizada para inicializar valores constantes o banderas que serán usadas a lo largo de la ejecución del código, sino que se trata de una clase preparada para inicializar el *setup* necesario para programa.

Cuando se habla de término *setup* se refiere a una serie de vectores en los cuales se van a almacenar los resultados obtenidos. Además, en esta clase se especifica el escenario de aplicación del algoritmo de GPR, esto es, la posición de los nodos donde se quiere calcular el campo recibido, o la posición inicial del transmisor en caso de no requerir de su estimación.

Para ello, cuenta con una serie de funciones que se van a ir llamando progresivamente a lo largo de la ejecución del programa principal, las cuales son descritas a continuación:

- **node_position**: Hace uso de la bandera *flagGridFix* para inicializar la posición de los nodos de manera fija o aleatoria.
- **tx_position**: Inicializa la posición del transmisor. Siempre se establece un valor inicial y en función de la bandera *flag_tx_est* se decide si se estima nuevamente la posición o se mantiene el valor establecido inicialmente. La posición del transmisor se elimina de entre las posibles posiciones de los nodos.

Tabla 6.2 Parámetros constantes a lo largo de la ejecución del código.

Variable	Descripción	Valor por defecto
shadowvarSim [dB]	Varianza del efecto shadowing en el exterior.	0:2:10
Nexp	Número total de experimentos.	10
Ttotal	Instantes de tiempo simulados.	10
P [dBm]	Potencia transmitida.	-10
U	Sensores disponibles entre el total de nodos.	10
alphai	Pérdidas por multitrayecto.	3.5
sigma_u	rho_u en [34].	200
Dcorr	Distancia de correlación.	50
varnoise	Varianza de ruido.	7
forgetting_factor	Factor de olvido. Un valor igual a cero olvida todo lo anterior mientras que con uno igual a uno todas las muestras aportan lo mismo.	0.5
rateoff	Porcentaje de los nodos totales que estarán apagados en cada instante de tiempo.	20
var_movement	Varianza entre instantes de tiempo consecutivos en el que se mueve los nodos.	1
noise_sigmau	Varianza correspondiente a la imprecisión en la posición de los sensores.	0

- **node_distribution**: Emplea la bandera *flag_moving* para determinar la ocupación de las posiciones de los nodos (especificadas en la función *node_position*) a lo largo de los instantes de tiempo.
- **initialize_vector**: Como se comentó previamente, es necesario generar una serie de vectores en los que se va a almacenar los resultados obtenidos a lo largo de la ejecución del código. Con el fin de optimizar al máximo el uso de memoria se hará uso de las banderas *flagGholami*, *flagWatcharapan*, *flagOKD*, *flagGPstatic* y *flagGPrecursive* para incializar los vectores únicamente cuando sea necesario.

Además, en la llamada al constructor de la clase será necesario especificar las características del *grid* donde se van a establecer la posición de los nodos. Por tanto, se especifican los siguientes términos: (1) Las dimensiones del *grid* las cuales son 500×500m en este proyecto, (2) la densidad de nodos dentro del *grid* (4 nodos por cada 100×100m) y (3) la distancia entre nodos, fijada en 100 metros.

6.1.4 Class *aditional_functions*

Tanto esta clase como la que se explica a continuación son las dos más importantes ya que incluyen las funciones más complejas así como la aplicación de la teoría a lenguaje de programación. Específicamente, entre este conjunto de funciones se encuentran aquellas que son utilizadas para la estimación de los parámetros requeridos en el modelo. Algunas de ellas serán tratadas sin entrar excesivamente en detalle en su contenido pero otras en cambio tendrán dedicadas una sección específica debido a su importancia en el proyecto.

Entre las funciones menos importantes se encuentran las listadas a continuación:

- **calculate_real_distance**: Se trata de una función diseñada para calcular la distancia entre las posiciones de los nodos o sensores. Como se puede comprobar en la Ecuacion (4.6) el parámetros distancia es uno de los especificados en el modelo de propagación.
- **calculate_noise_distance**: Es una función desarrollada de manera similar a la anterior pero dedicada exclusivamente a la posición de los sensores que varían su posición inicial de manera aleatoria con el fin de añadir incertidumbre a dicha posición.
- **readjust_node_emition**: El programa está preparado para simular un supuesto en el que los sensores están encendidos o apagados en función del instante de tiempo. Por tanto, será necesario especificar qué sensores están recibiendo señal y qué sensores permanecen apagados en cada instante de tiempo. Está función se encarga de determinar, en base al estado de la bandera *flag_intermitent*, los sensores que están activos en el instante de tiempo actual así como su posición.

En los siguientes apartados se pasará a estudiar de manera más detalla aquellas funciones con mayor relevancia y que además requieren de un mayor esfuerzo para su explicación.

Function *calculate_sample_shadow*

Aunque no se trate de una función encargada de estimar algún parámetro, es conveniente estudiar esta función de manera exhaustiva debido a su relación con la teoría. Es la función encargada de generar muestras que modelan la atenuación debido al fenómeno del *shadowing*, la cual sigue una distribución normal de media cero y varianza *shadowvar*. Estas muestras tienen que ser correladas de acuerdo al parámetro *Dcorr* por lo que son generadas a partir de una variable gaussiana multidimensional de media cero y matriz de covarianza obtenida a partir de la Ecuación (4.3). Para la generación de estas muestras se ha seguido el Código 6.1.

Código 6.1 Gneración de muestras para modelar atenuación debido al efecto de shadowing.

```
1  def calculate_sample_shadow(variables, Nall, Dall, kk):
2
3    # Media 0
4    m=np.zeros((Nall))
5
6    shadowvar=variables.shadowvarSim[kk]
7    shadowvar_nu=np.power(10,shadowvar/10)
8    shadowstd_nu=math.sqrt(shadowvar_nu)
9    shadowstd=10*np.log10(shadowstd_nu)
10   # En cada iteración se tiene que re-calcular este valor
11   # Parámetro: \sigma_v^{2}
12   var_v=shadowstd**2
13
14   # Cálculo de \rho_u
15   rho=np.exp(-Dall/variables.Dcorr)
16
17   # Ecuación (4) en paper
18   C=var_v*rho
19
20   # Generación de muestras por atenuación debido al efecto de shadowing
21   deltaall=np.random.multivariate_normal(m,C)
22
23   # Las muestras se distribuyen entre las salidas para las muestras de
24   # entrenamiento y las muestras de test
25   deltaM=deltaall[0:variables.M*variables.Ttotal]
26   deltaU=deltaall[variables.M*variables.Ttotal:Nall]
27
28   deltaM=np.reshape(deltaM,(variables.M,variables.Ttotal),order='F')
29
30   return deltaM, deltaU, shadowstd, var_v
```

Se puede observar como en la línea 18, la variable C es obtenida de manera equivalente a la matriz de covarianza de la Ecuación (4.3). La media es cero como se aprecia en la línea 4 y, además cabe destacar el cálculo de la variable *var_v* que corresponde con el parámetro σ_v^2 en la Ecuación (4.3). Este valor es calculado en cada iteración ya que los valores de varianza que modelan el efecto *shadowing* varían a lo largo del tiempo con el fin de llevar a cabo diferentes simulaciones para distintos valores de varianza. Esto último se detallará de manera más específica en el apartado de simulaciones realizadas. Un último aspecto a destacar corresponde a las muestras generadas ya que, como se aprecia en las líneas 25 y 26, son distribuidas entre los M nodos (*deltaM*) y los U sensores (*deltaU*). [1]

Function *estimate_tx_pos*

Para el código desarrollado en esta función se ha usado las fórmulas presentadas en el estudio realizado en la Sección 4.2.1. Aunque en esa sección se comentan también aspectos relacionados con el refinamiento llevado a cabo para mejorar aún más la estimación de la posición, no es hasta una función posterior cuando es calculado este refinamiento. Por tanto, en esta función únicamente se estima la posición del transmisor,

[1] La variable U corresponde a los sensores disponibles entre los N sensores totales.

dejando para más adelante la mejora de la estimación y quedando de esta forma un código más simple el cual se muestra en el Código 6.2.

Código 6.2 Estimación de la posición del transmisor.

```
1  def estimate_pos_tx(ZMon, posMxNoise, posMyNoise, posNxNoise, posNyNoise, t):
2    global w_told
3    global postx_est
4    # Pesos. Ecuación (12)
5    wi=10**(ZMon/10)
6    # Algoritmo recursivo para estimar la posición del transmisor
7    if t>0:
8      # Re-calculo de los pesos en cada instante de tiempo
9      # Ecuación (13)
10     w_t=w_told+np.sum(wi)
11     # Uso de los pesos para estimar la posición del transmisor
12     # Ecuación (11)
13     postx_est=np.append((np.sum(wi*posMxNoise)+w_told*postx_est[0])/w_t,
14              (np.sum(wi*posMyNoise)+w_told*postx_est[1])/w_t)
15     w_told=w_t
16   else:
17     postx_est=np.append(np.sum(wi*posMxNoise)/np.sum(wi),
18              np.sum(wi*posMyNoise)/np.sum(wi))
19     w_told=np.sum(wi)
20
21   # Se incluye el error en la localización del transmisor debido
22   # al ruido de la distancia de los senores y los "grid points"
23   dtxNoise=np.sqrt((postx_est[0]-posNxNoise)**2+
24   (postx_est[1]-posNyNoise)**2)
25
26   return dtxNoise, postx_est
```

Tras un análisis del código, se puede apreciar el algoritmo iterativo usado para estimar la posición del transmisor en el que se usa el peso calculado en el instante anterior como ya se explicó en la Sección 4.2.1 y como se observa en la línea 15. Otro aspecto a destacar se encuentra en la relación entre las líneas 5 y 10 y las Ecuaciones (4.13) y (4.14), respectivamente. Por último, la Ecuación (4.12) se desarrolla en las líneas 17 (para el primer instante de tiempo) y 13, dejando de esta forma constancia total entre el código desarrollado y la teoría estudiada. Como se ha comentado anteriormente, las ecuaciones correspondientes al refinamiento en la estimación serán estudiadas en la sección siguiente.

Function *alpha_beta_estimation*

Se trata de la función más importante dentro de este bloque ya que incluye tanto la estimación de los parámetros α y P del modelo de la Ecuación (4.6), como el refinamiento en el cálculo de la estimación del transmisor, cuyo nuevo valor es usado para mejorar de igual forma la estimación inicial de la potencia y del exponente de pérdidas.

Para esta función se ha necesitado de *técnicas* de programación un poco más complejas que hacen que la relación ecuación-código sea más difícil de asimilar, por lo que a diferencia de las funciones anteriores, no se va a mostrar la función completa para, posteriormente realizar un análisis de las líneas más trascendentales, sino que se va a ir más despacio.

Primeramente, se va a proceder al cálculo de la media de las distribuciones correspondientes a las estimaciones del exponente de pérdidas y potencia transmitida. A modo de referencia teórica se va a hacer uso de la Ecuación (4.23) en la que se se lleva a cabo una minimización de una expresión respecto a dos parámetros (μ_α y μ_P). Esta ecuación queda resumida en las líneas presentadas en el Código 6.3.

Código 6.3 Media de la muestras correspondientes a la estimación de α y P.

```
# Variable \hat{q} de la Ecuación (25)
q2use=10*np.log10(dtxNoise[0:variables.Mon])
# Variable \sqrt{\hat{D}}^{-1} en Ecuación (25)
errloc=1./dtxNoise[0:variables.Mon]

# Matriz correspondiente al primer sumando de la Ecuación (25)
Prev=np.append(np.ones(variables.Mon)/errloc, -q2use/errloc)
Prev2=np.reshape(Prev,(variables.Mon,2),order='F')
# Minimización correspondiente a la Ecuación (25)
params=lsq_linear(np.reshape(Prev,(variables.Mon,2),order='F'),
ZMon[0:variables.Mon]/errloc)

meanP=params.x[0]
meanalpha=params.x[1]
```

Las variables que se encuentran en las líneas 2 (*q2use*) y 4 (*errloc*) corresponden respectivamente a los parámetros $\hat{\mathbf{q}}$ y $\sqrt{\hat{\mathbf{D}}}^{-1}$ de la Ecuación (4.23). Por otro lado, y con el fin de calcular los valores que minimizan la función se hace uso del método *lsq_linear* correspondiente a la librería *scipy.optimize*. De acuerdo con la documentación de la librería [42], este método es usado para calcular el mínimo en la siguiente expresión: $\text{mín}_x ||Ax - b||^2$. Donde A es el primer argumento del método y b el segundo. Es por ello que el primer argumento corresponde con la variable *Prev*, la cual se trata de una matriz $Nx2$ donde la primera columna está formada por unos y la segunda por los valores calculados en *q2use*, todo ello dividido por la variable *errloc*. Además, el segundo argumento lo forman los valores de campo recibido en los sensores, los cuales están almacenados en la variable *ZMon* y corresponden al parámetro $\mathbf{z}$, dividido igualmente por la variable *errloc*. De esta forma, los valores mínimos son almacenados en la variable *params* correspondiendo el primer valor a la estimación de la potencia transmitida y el segundo al exponente de pérdidas.

Para el cálculo de la varianza de las distribuciones correspondientes a las estimaciones de α y P se va a proceder de la misma forma pero siguiendo la Ecuación (4.22). En este caso, se ha hecho uso del Código 6.4, el cual se explica a continuación.

Código 6.4 Varianza de la muestras correspondientes a la estimación de α y P.

```
# Variable \hat{\mu}_{z} en PAG 4
meanz=meanP-q2use*meanalpha
# Variable \Sigma_{z}
Sigmaz=(ZMon[0:variables.Mon]-meanz)[:,np.newaxis]@(ZMon[0:variables.Mon]-meanz
    )[np.newaxis,:]
Sigmazalpha=CNoise[0:variables.Mon,0:variables.Mon]+(variables.varnoise+
    variables.noise_varnoise)*np.eye(variables.Mon)+np.diag((variables.sigma_u+
    variables.noise_sigmau)**2/dtxNoise[0:variables.Mon]**2)

# Variable A en PAG 4
AAon=Sigmaz-Sigmazalpha
# Variable \hat{Q} en PAG 4
BBon=q2use[:,np.newaxis]@q2use[np.newaxis,:]

# Se obtienen las diagonales de las dos matrices anteriores
aaon=np.diag(AAon).flatten('F')
bbon=np.diag(BBon).flatten('F')

# Matriz correspondiente al primero sumando de la Ecuación (26)
Prev=np.append(np.ones(np.size(aaon)), bbon)
# Minimización correspondinete a la Ecuación (26)
params=lsq_linear(np.reshape(Prev,(np.size(aaon),2), order='F'),
                  aaon, bounds=(0,np.inf))

```

```
22 varP=params.x[0]
23 varalpha=params.x[1]
```

Primeramente, los resultados almacenados en las líneas 2 y 4 corresponden a los parámetros μ_z y $\Sigma_{z|\alpha,P}$ de las Ecuaciones (4.19) y (4.18) respectivamente. Además, el valor de la variable *Sigmaz* corresponde al resultado de la operación: $\mathbf{\Sigma}_z = (\mathbf{z}-\hat{\boldsymbol{\mu}})(\mathbf{z}-\hat{\boldsymbol{\mu}})^T$. Por tanto, estos valores son usados para calcular las matrices de la Ecuación (4.22), a las cuales se les extrae posteriormente la diagonal almacenando los resultados en las variables *aaon* y *bbon* (líneas 13 y 14). Al igual que se hizo anteriormente, estas dos variables forman parte del primer y segundo argumento del método *lsq_linear*, mostrando especial interés en el primer argumento tratándose este de otra matriz de dimensiones $Nx2$ donde la primera columna está formada por unos y la segunda por el vector *bbon* obtenido a partir de la diagonal de *BBon* (variable que almacena la operación $\hat{\mathbf{q}}\hat{\mathbf{q}}^T$). El segundo argumento está compuesto por la diagonal de la matriz *AAon*, la cual está formada por los resultados de la operación $\mathbf{\Sigma}_z - \mathbf{\Sigma}_{z|\alpha,P}$. Como se demostró anteriormente, los resultados finales se extraen de la variable *params*.

Por último esta función tan completa incluye los aspectos relacionados con el refinamiento de la posición estimada del transmisor, presentado principalmente a partir de la Ecuación (4.15). Para este caso se ha hecho uso de las *funciones lambda* de *Python* en las cuales se definen funciones anónimas que dependen de una o varias variables y se va a proceder a minimizar esa función ya que únicamente depende de un parámetro (la posición del transmisor). El valor mínimo obtenido será la posición re-estimada del transmisor, la cual mejora a la primera estimación. Para eso se usan las líneas del Código 6.5 las cuales se explican más adelante.

Código 6.5 Refinamiento de la posición del transmisor.

```
1  rmean=np.sum(ZMon)/variables.Mon
2  betai=lambda ss, tt: np.log10(np.sqrt((ss-posMxNoise)**2+(tt-posMyNoise)**2))
3  betamean=lambda ss, tt: np.sum(betai(ss,tt))/variables.Mon
4
5  fun=lambda sstt: np.sum((betai(sstt[0],sstt[1])-betamean(sstt[0],sstt[1]))*
6  (ZMon-rmean))/np.sqrt(np.sum((betai(sstt[0],sstt[1])-
7  betamean(sstt[0], sstt[1]))**2)*np.sum((ZMon-rmean)**2))
8
9  x0=postx_est
10 # Minimización correspondiente a la Ecuación (27)
11 postx_est_prev=minimize(fun, x0)
12 postx_est=postx_est_prev.x
```

No merece la pena detenerse en este código ya que se aprecia claramente como en la línea 5 se define la función anónima la cual es minimizada en la línea 11. Los resultados finales de posición estimada del transmisor son extraídos en la línea 12.

6.1.5 Class *algoritmos*

En esta clase se estudian las dos funciones que engloban lo más relevante de este trabajo, ya que en ambas se implementan los algoritmos de GPR para estimación de campo recibido tanto para los casos en los que la información proporcionada por sensores corresponden a campo estático como a campo variante en el tiempo. Para la inicialización del *kernel* y para la optimización de sus hiperparámetros se han usado librerías existentes las cuales han sido descritos en un capítulo anterior. En esta sección únicamente se va a presentar la integración de los conceptos teóricos de GPR en un lenguaje de programación por lo que la base teórica para esta sección se encuentra en las Secciones 4.3.1 y 4.3.2.

La importancia detrás de esta sección está relacionada con el cumplimiento de uno de los objetivos planteados ya que en el capítulo correspondiente a los objetivos se mencionaba que uno de estos consistía en la asimilación y aplicación de un algoritmo de ML en un escenario *real*. Por tanto, y teniendo en cuenta que es necesario comentar también los aspectos restantes del escenario de aplicación, en esta sección confluyen los aspectos más interesantes desde el punto de vista de programación de un algoritmo de ML.

La dinámica seguida en estas funciones es siempre la misma, y se trata de uno de los procedimientos más usados a la hora de aplicar algoritmos de ML en escenarios reales. Normalmente, se parte de librerías las cuales reúnen los aspectos técnicos más complejos con el fin de liberar al programador de la gran carga que

supone desarrollar los algoritmos. Por lo que gracias al uso de los métodos y clases implementadas en estas librerías, el esfuerzo final dedicado se reduce exponencialmente.

Debido a la importancia de estas dos funciones y a la cantidad de código implementado en ellas, se estudiarán por separado como se hizo en la sección anterior.

Function *algoritmo_GPStatic*

Primeramente, para esta sección se ha partido de la base teórica expuesta en la Sección 4.3.1 por lo que las ecuaciones que ahí aparecen serán usadas a lo largo de esta función. Se considera de importancia destacar, antes de entrar a valorar los aspectos más interesantes de esta función, la diferencia entre las muestras de entrenamiento y las de *test*. La variable *Mon* (que corresponde con el número M de nodos del *grid*), la cual es usada varias veces a lo largo de esta función, se utiliza para marcar el límite de las muestras que son usadas para entrenamiento dejando las restantes hasta llegar al total de las proporcionadas por los N sensores para *test*.

La preparación de algunas variables, las cuales son necesarias calcular antes de aplicar el algoritmo, se muestra en las líneas del Código 6.6.

Código 6.6 Preparación de variables aplicadas en GPR.

```
1  Prev=np.append(posNxNoise[0:variables.Mon], posNyNoise[0:variables.Mon])
2  # Entradas correspondientes a las muestras de entrenamiento
3  Xtrain=np.reshape(Prev, (variables.Mon, 2), order='F')
4  N_training=np.size(Xtrain,0)
5  # Salidas correspondientes a las muestras de entrenamiento
6  ytrain=ZMon
7  Prev=np.append(posNxNoise[variables.Mon:variables.N], posNyNoise[variables.Mon:
       variables.N])
8  # Entradas correspondientes a las muestras de test
9  Xtest=np.reshape(Prev, (variables.N-variables.Mon,2), order='F')
10
11 q=10*np.log10(dtxNoise[0:variables.Mon])
12 qd=10*np.log10(dtxNoise[variables.Mon:variables.N])
13 # Variable que trata de emular el kernel LOG y el término bias (\sigma_{P})
14 qq=varalpha*np.append(q,qd)[:,np.newaxis]@np.append(q,qd)[np.newaxis,:]+varP*np
       .ones([variables.N,variables.N])
15 Sigman=(variables.varnoise+variables.noise_varnoise)*np.eye(variables.Mon)+ np.
       diag((variables.sigma_u+variables.noise_sigmau)**2/dtxNoise[0:variables.Mon
       ]**2)
16
17 # Media de GP
18 mtrain=meanP-10*meanalpha*np.log10(dtxNoise[0:variables.Mon])
19 mtest=meanP-meanalpha*qd
```

Como entrada a la *función objetivo* se usa las posiciones imprecisas de los sensores las cuales son definidas a partir de su valor en el eje de abcisas y de ordenada. De ahí que la variable *Xtrain* de la línea 3 sea una matriz de dos columnas y *Mon* filas. Esta variable se usa para almacenar las entradas correspondientes a las muestras de entrenamiento y, de igual forma, modela el parámetro $\hat{\mathbf{X}}$ de las ecuaciones de la Sección 4.3.1. Además, la variable *ytrain* corresponde a las salidas de dicha función objetivo para las muestras de entrenamiento las cuales han sido calculadas anteriormente en otra parte del código y corresponde con el campo recibido en los sensores, almacenado en la variable *ZMon*[2]. Esta variable modela el parámetro $\mathbf{z}(t)$ en las ecuaciones de la ya mencionada sección. Por último, la variable *Xtest* de la línea 9 almacena las entradas pero en este caso correspondientes a las muestras de *test*, esto es, corresponden al parámetro $\mathbf{X}_g$ en las ecuaciones. El subíndice g indica que son posiciones que pertenecen al *grid*.

Por otro lado, las variables de las líneas 11 y 12 están relacionadas con los parámetros $\mathbf{q}$ y $\mathbf{q}_g$ respectivamente. Para el caso en el que no se optimizan los parámetros del *kernel*, y como se verá un poco más adelante, se hace uso de un *kernel* **RBF** más una serie de términos que tratan de modelar el *kernel* propuesto en la Ecuación

[2] El cálculo del campo recibido se lleva a cabo en el *script* principal y será comentado más adelante.

(4.27). Estos términos que se suman a los resultados obtenidos a partir del kernel **RBF** son almacenados en la variable *qq*. Basta con hacer una breve observación para asociar estos términos de la Ecuación (4.27) con la variable *qq*.

La variable *Sigman* de la línea 15 modela las muestras de ruido correspondientes al modelo GP de la Ecuación (4.24) y, como ya explicó, estas muestras siguen una distribución cuya media es cero y cuya matriz de covarianza tiene la siguiente expresión: $\Sigma_n = \sigma_w \mathbf{I}_N + \rho_u^2 \hat{\mathbf{D}}(t)$.

Por último dentro de este primer bloque de líneas, se comenta las variables *mtrain* y *mtest* de las líneas 18 y 19. Estas variables corresponden con los parámetros $\mathbf{m}_{\mathbf{X}_g(t)}$ y $\mathbf{m}_{\hat{\mathbf{X}}(t)}$ de las Ecuaciones (4.30) y (4.25) respectivamente, esto es, con las funciones media de los procesos de GP tanto para las muestras de entrenamiento como para las muestras de *test*.

Por tanto, se dispone en este momento de una serie de variables las cuales han sido inicializadas a partir de los datos de campo recibido en las posiciones de las muestras de entrenamiento, y se quiere estimar la media y la matriz de covarianza de las muestras que modelan el campo estimado en ciertas posiciones, determinadas por las muestras de *test*. Para ello se hará uso de las líneas presentadas en el Código 6.7.

Código 6.7 Estimación de campo recibido en posiciones de *test* I.

```
1  if flags.optimiseHyperparameters==True:
2    # Creación del Kernel
3    bias=varP*np.ones([variables.N,variables.N])
4    NSEKern=GPy.kern.NSE(Xtrain.shape[1])
5    LOGKern=GPy.kern.LOG(Xtrain.shape[1])
6
7    # Suma de los dos kernels. Ecuación (42)
8    kern=NSEKern+LOGKern
9    beta=np.diag(1/((variables.sigma_u+variables.noise_sigmau)**2/dtxNoise**2+ (
         variables.varnoise+variables.noise_varnoise)*np.ones(variables.N)))
10
11   Prev=(ytrain-mtrain)[:,np.newaxis]@np.ones([1,2])
12
13   # Selección del modelo Gaussian Processes for Regression (GPR)
14   mR=GPy.models.GPRegression(Xtrain, Prev, kern)
15   # Optimización de los hiperparámetros
16   mR.optimize()
17
18   # Obtención del kernel una vez optimizado los hiperparámetros
19   kern=mR.kern
20
21   # K_{\hat{X} en Ecuación (37)
22   Ktrain=kern.K(Xtrain,Xtrain)+bias[0:N_training,0:N_training]+beta[0:
         N_training,0:N_training]
23   # K_{X_{g}, \hat{X}} en Ecuación (37)
24   Kx=kern.K(Xtest,Xtrain)+bias[N_training:,0:N_training]
25   # K_{X_{g}} en Ecuación (37)
26   Ktest=kern.K(Xtest,Xtest)+bias[N_training:,N_training:]
27   # Inversa de la Ecuación (41)
28   C_inv=np.linalg.inv(Ktrain+Sigman)
29
30   # La media de la predicción
31   mGP=mtest+Kx@C_inv@(ytrain-mtrain)
32   # La covarianza de la predicción
33   vGP=Ktest-Kx@C_inv@Kx.T
34
35   setup.RSSgpstatic=mGP
36
37   # Error cuadrático medio entre señal estimada y valor teórico
```

```
38      setup.mseGPRstatic[kk,t]=setup.mseGPRstatic[kk,t]+np.sum((setup.RSSgpstatic-
            ZUon)**2)
39    else:
40      # Kernel RBF para el caso en el que no se optimizan los hiperparámetros
41      kern=GPy.kern.RBF(input_dim=2,variance=shadowstd**2,
42      lengthscale=variables.Dcorr, inv_l=True)
43
44      # En este caso se añade la variable qq para tratar de simular lo
45      # máximo posible el kernel de la Ecuación (42)
46      Ktrain=kern.K(Xtrain,Xtrain)+qq[0:N_training,0:N_training]
47      Kx=kern.K(Xtest,Xtrain)+qq[N_training:,0:N_training]
48      Ktest=kern.K(Xtest,Xtest)+qq[N_training:,N_training:]
49      C_inv=np.linalg.inv(Ktrain+Sigman)
50
51      # La media de la predicción
52      mGP=mtest+Kx@C_inv@(ytrain-mtrain)
53      # La covarianza de la predicción
54      vGP=Ktest-Kx@C_inv@Kx.T
55
56      setup.RSSgpstatic=mGP
57
58      # Error cuadrático medio entre señal estimada y valor teórico
59      setup.mseGPstatic[kk,t]=setup.mseGPstatic[kk,t]+np.sum((setup.RSSgpstatic-
            ZUon)**2)
```

Se puede apreciar como el código se divide en dos partes bien diferenciadas, la primera de ella correspondiente a las líneas 1-38 y la segunda a las líneas 40-59. En el primer bloque se lleva a cabo la estimación del campo recibido realizando previamente una **optimización de los hiperparámetros del *kernel***.

Pero primeramente, se realiza la definición del *kernel* en las líneas 3,4 y 5 en las que se inicializa el término *bias* y los *kernel* **NSE** y **LOG** respectivamente. Cada término corresponde con cada sumando de la Ecuación (4.27) y, gracias al uso de la librería *GPy*, la cual se presentó en un capítulo anterior, basta con usar el operando $+$ para definir la función *kernel* por completo.

Posteriormente, destaca las líneas 14 y 16 donde nuevamente la librería *GPy* es usada para definir el modelo, el cual como se ha comentado a lo largo de esta memoria no es otro que un proceso gaussiano para regresión (GPR), y se inicializa con las entradas correspondientes a las muestras de entrenamiento y con las salidas una vez restada la media. Una de las líneas más importantes es la 16, en la que se aprecia como el modelo se optimiza con el fin de obtener aquellos hiperparámetros que proporcionen mejores resultados en la función *kernel*.

Nuevamente, las líneas 21-28 corresponden a la inicialización de variables las cuales son sumamente importante en este proyecto. Con el fin de conseguir un mejor entendimiento de esta parte tan importante, se ha rellenado la Tabla 6.3 en la que se realiza una asociación entre las variables definidas en estas líneas y los parámetros de la Ecuación (4.32).

Tabla 6.3 Asociación parámetro-variable en Ecuación (4.32).

Variable	Parámetro	Argumentos de entrada
Ktrain	$\mathbf{K}_{\hat{\mathbf{X}}}$	Muestras de entrenamiento
Kx	$\mathbf{K}_{\mathbf{X}_g,\hat{\mathbf{X}}}$	Muestras de entrenamiento y test
Ktest	$\mathbf{K}_{\mathbf{X}_g}$	Muestras de test

Para inicializar estas variables se hace uso de la función *kernel* una vez optimizado los hiperparámetros. Los argumentos de entrada de esa función son las posiciones correspondientes a las muestras de entrenamiento o *test* según corresponda. A los resultados proporcionados como salidas de la función se les suma el término *bias* el cual no fue introducido en la optimización del *kernel* y por tanto se añade en este punto y además, a la variable *Ktrain* se le suma la variable *beta* que incluye el ruido de las muestras como se aprecia en la Ecuación (4.32). Adicionalmente, se encuentra la variable *C_inv* que está asociada con el parámetro $\mathbf{C}_{\hat{\mathbf{X}}}^{-1}$ (Ecuación (4.36)).

Todas estas variables definen perfectamente la función densidad de probabilidad conjunta necesaria en todo proceso gaussiano para, una vez condicionada dicha función densidad de probabilidad a las salidas (*ytrain* o $\mathbf{z}(t)$) se pueda obtener la media y la matriz de covarianza de las muestras que modelan el campo estimado. En este punto se aprecia claramente como las líneas 31 y 33 corresponden perfectamente con las Ecuaciones (4.34) y (4.35), calculando de esta forma la media y la matriz de covarianza de las muestras respectivamente.

Por último, en la línea 38 se calcula el error cuadrático medio cuya finalidad será estudiada en el capítulo de simulaciones realizadas.

Para el caso en el que no se optimizan los hiperparámetros el código es mucho más simple ya que únicamente se define un una función *kernel* **RBF** la cual no es optimizada y que además es usada para calcular directamente las variable *Ktrain*, *Kx* y *Ktest* en las líneas 46-49. Adicionalmente, se le suma la variable *qq* con el fin de simular de la manera más fiel posible el *kernek* de la Ecuación (4.27). Igualmente se estima la media y la matriz de covarianza de las muestras en las líneas 52 y 54 a partir de las variables anteriores y de las ecuaciones ya mencionadas.

Function *algoritmo_GPRecursive*

Una vez estudiado el código para el caso en el que campo es estático se va a proceder a presentar el código utilizado para estimación en el supuesto en el que el campo recibido en los sensores sea variante en el tiempo. Las diferencias desde el punto de vista teórico respecto al caso en el que el campo recibido en los sensores es estático han sido estudiadas en un capítulo anterior por lo que aquí únicamente se presentará la aplicación de las fórmulas mostrando especial interés en la introducción del factor de olvido.

Las líneas correspondientes a la estimación en este supuesto se encuentran en el Código 6.8 en el que únicamente se incluyen las líneas correspondientes a la estimación sin optimización de hiperparámetros con el fin de evitar cargar en exceso el código introducido en esta memoria.

Código 6.8 Estimación de campo recibido en posiciones de *test* II.

```
1  global mprior
2  global Cprior
3  Prev=np.append(posNxNoise[0:variables.Mon], posNyNoise[0:variables.Mon])
4  # Entradas correspondientes a las muestras de entrenamiento
5  Xtrain=np.reshape(Prev, (variables.Mon, 2), order='F')
6  N_training=np.size(Xtrain,0)
7  # Salidas correspondientes a las muestras de entrenamiento
8  ytrain=ZMon
9  Prev=np.append(posNxNoise[variables.Mon:variables.N], posNyNoise[variables.Mon:
       variables.N])
10 # Entradas correspondientes a las muestras de entrenamiento
11 Xtest=np.reshape(Prev, (variables.N-variables.Mon,2), order='F')
12
13 q=10*np.log10(dtxNoise[0:variables.Mon])
14 qd=10*np.log10(dtxNoise[variables.Mon:variables.N])
15 # Variable que trata de emular el kernel LOG y el término bias (\sigma_{P})
16 qq=varalpha*np.append(q,qd)[:,np.newaxis]@np.append(q,qd)[np.newaxis,:]+varP*np
       .ones([variables.N,variables.N])
17 Sigman=(variables.varnoise+variables.noise_varnoise)*np.eye(variables.Mon)+ np.
       diag((variables.sigma_u+variables.noise_sigmau)**2/dtxNoise[0:variables.Mon
       ]**2)
18
19 # Media de GP
20 mtrain=meanP-10*meanalpha*np.log10(dtxNoise[0:variables.Mon])
21 mtest=meanP-meanalpha*qd
22
23 # Kernel RBF para el caso en el que no se optimizan los hiperparámetros
24 kern=GPy.kern.RBF(input_dim=2,variance=shadowstd**2,
25 lengthscale=variables.Dcorr, inv_l=True)
26
```

```
27  # En este caso se añade la variable qq para tratar de simular lo
28  # máximo posible el kernel de la Ecuación (42)
29  # K_{\hat{X}} en Ecuación (37)
30  Ktrain=kern.K(Xtrain,Xtrain)+qq[0:N_training,0:N_training]
31  # K_{X_{g}, \hat{X}} en Ecuación (37)
32  Kx=kern.K(Xtest,Xtrain)+qq[N_training:,0:N_training]
33  # K_{X_{g}} en Ecuación (37)
34  Ktest=kern.K(Xtest,Xtest)+qq[N_training:,N_training:]
35  # Inversa de la Ecuación (41)
36  C_inv=np.linalg.inv(Ktrain+Sigman)
37
38  # \mu_{post}. Ecuación (57)
39  mpos=Kx@C_inv@(ytrain-mtrain)
40  # \Sigma_{post}. Ecuación (58)
41  Cpos=Kx@C_inv@Kx.T
42
43  # "prior" in los nodos
44  if t==0 or (flags.flag_moving==0 and flags.flag_intermitent==0):
45    mprior=mpos
46    Cprior=Cpos
47
48  # La media de la predicción
49  mGP=mtest+variables.forgetting_factor*mpos+(1-variables.forgetting_factor)*
        mprior
50  # La covarianza de la predicción
51  vGP=Ktest-variables.forgetting_factor*Cpos-(1-variables.forgetting_factor)*
        Cprior
52
53  setup.RSSgprecursive=mGP
54
55  #\mu_{prior}. Ecuación (59)
56  mprior=variables.forgetting_factor*mprior+(1-variables.forgetting_factor)*mpos
57  #\Sigma_{prior}. Ecuación (60)
58  Cprior=variables.forgetting_factor*Cprior+(1-variables.forgetting_factor)*Cpos
59
60  # Error cuadrático medio entre señal estimada y valor teórico
61  setup.mseGPrecursive[kk,t]=setup.mseGPrecursive[kk,t]+np.sum((setup.
        RSSgprecursive-ZUon)**2)
```

Observando las líneas expuestas, se aprecia como la preparación de variables del Código 6.6 son exactamente iguales que las líneas 3-21. Además, la definición de la función *kernel*, así como el calculo de las variables de la Tabla 6.3 se realiza en las líneas 30-36 de manera similar a como se hacia en el caso estático.

Las novedades en este fragmento de código empiezan a aparecer en la línea 39 y 41, donde se calcula la media y la matriz de covarianza para la función densidad de probabilidad *a posteriori* usando las expresiones de las Ecuaciones (4.39) y (4.40) respectivamente. Por otro lado, para el (1) instante cero, o (2) para el caso en el que los sensores desplacen su posición en cada instante de tiempo, o (3) para el supuesto en el que los sensores permanezcan apagados en algún instante de tiempo, la media y matriz de covarianza *a priori* coincide con los valores *a posteriori* calculados en las líneas mencionadas anteriormente. Esta consideración se muestra en las líneas 44-46.

Una vez inicializadas estas variables, la media y matriz de covarianza de las muestras que modelan el campo estimado se calcula en las líneas 49 y 51 haciendo uso de la base teórica de las Ecuaciones (4.37) y (4.38). En estas líneas, y como ocurría en las ecuaciones, aparece el factor de olvido o *forgetting factor* para dar una mayor relevancia en la estimación a la muestras recientes frente a las más antiguas.

Por último, se calcula la media y matriz de covarianza *a priori* para la siguiente iteración en las líneas 56 y 58 haciendo uso de las Ecuaciones (4.41) y (4.42).

6.1.6 Class *gráficas*

Esta última clase tiene una importancia considerablemente inferior a sus dos inmediatas predecesoras ya que únicamente cuenta con tres métodos los cuales son usados para representar los resultados obtenidos. Los detalles más importantes de estos métodos están relacionados con el uso de la librería *matplotlib*, por lo que se considera innecesario entrar demasiado en detalle.

Los tres métodos que forman esta clase se listan a continuación:

- **representacion_scenario**: Es usada para representar una gráfica muy interesante del escenario de aplicación del algoritmo ya que presenta la posición del transmisor así como de los sensores y nodos donde se pretende estimar el campo recibido.
- **representacion_variacion_potencia**: Se trata de un método usado para representar el campo recibido en función de la distancia. Para los resultados de RSS que se quieren representar se consideran las medidas de los sensores, los valores estimados y los valores teóricos.
- **representacion_final_algoritmo**: Se encarga de representar el error cuadrático medio entre la estimación realizada y el valor teórico real frente a distintos valores de varianza de las muestras que modelan la atenuación por efectos de *shadowing* (σ_v^2).

6.2 Fichero principal

Todas y cada una de las clases presentadas en la anterior sección son inicializadas meticulosamente y los métodos desarrollados en cada una de ellas son llamados de manera orquestada a partir de un fichero principal. En torno al 90 % está condensado en las clases presentadas en la sección previa mientras que el 10 % del código restante, el cual se presenta en esta sección, se encarga de gestionar las llamadas a las clases y métodos de manera ordenada.

Este 10 % correspondiente al fichero principal cuenta con la siguiente estructura: Destaca la presencia de tres bucles que hacen que el tiempo de ejecución del algoritmo sea bastante extenso. En los dos bucles siguientes al primero se itera en torno a los instante de tiempo y los distintos valores de varianza de *shadowing*. Por defecto, se lleva a cabo una simulación de 10 instantes de tiempo, con un segundo por iteración mientras que los valores de varianza que modela las pérdidas por *shadowing* están acotados entre 0.5 y 10, iterando cada dos valores. El primer bucle está implementado con el fin de realizar un promedio de los resultados obtenidos. Más adelante, en el capítulo correspondiente a las simulaciones realizadas, se profundizará más en la importancia de este promediado.

Como se vio en las funciones correspondientes a la Clase *setup*, la posición de los nodos del *grid* se configuraban haciendo uso de funciones definidas en esa clase. En cambio, las posiciones de los sensores dentro de ese *grid* es determinada en el fichero principal. Además, como se comentó en el capítulo correspondiente a la presentación del escenario, es necesario añadir una indeterminación a la posición de los sensores con el fin de dar un mayor realismo al modelo, ya que esta no puede ser perfectamente conocida. Este calculo de error se realiza igualmente en el fichero principal.

Por otro lado, la acción más importante a realizar dentro del fichero principal se recoge en las líneas mostradas en el Código 6.9.

Código 6.9 Calculo de campo recibido en sensores.

```
# Path-loss exponent
alpha=np.ones(variables.N)*variables.alphai

# Shadowing
delta=np.append(deltaM[indexon,t],deltaU)

# Ruido
noise=np.append(np.sqrt(variables.varnoise)*np.random.randn(variables.Mon),np.
    zeros(variables.U))

# RSSI
Z=variables.P-np.diag(10*alpha*np.log10(dtx))@np.ones(variables.N)+delta+noise
```

```
12 ZMon=Z[0:variables.Mon]
13 ZUon=Z[variables.Mon:variables.N]
```

En la línea 8 se observa claramente la implementación en fórmula de la Ecuación (4.6) en la que se aprecia las muestras que modelan la atenuación por *shadowing* así como el ruido gaussiano que modela el ruido propio de las muestras así como la imprecisión en la posición de los sensores. Estas muestras quedan almacenadas en las variables *delta* y *noise* respectivamente. Los resultados obtenidos a partir de la línea 8 tienen principalmente dos funciones:

1. Las primeras *Mon* muestras son tomadas como si fueran información proporcionada por sensores y se almacena en la variable *ZMon*. Esta variable es usada principalmente como salida de la función objetivo en las posiciones de entrenamiento por lo que se asocia al parámetro $\mathbf{z}(t)$ en las Ecuaciones.
2. Las muestras restantes son tomadas como campo recibido de manera teórica en las posiciones de los nodos. Por tanto el valor estimado a partir del algoritmo de GPR será comparado con los valores almacenados en esta variable con el fin de calcular el error cuadrático medio y de esta forma obtener un parámetro que mide la precisión de la estimación realizada.

6.3 Código desarrollado para *sparse* GP

Como ya fue comentado en un capítulo anterior, en este trabajo se hace uso de una técnica que pretende reducir el tiempo de ejecución total de este código. La base teórica detrás de esta técnica se presenta en la Sección 3.4. Por tanto, y siguiendo con la idea de este capítulo, únicamente se mostrarán las líneas de código necesarias para hacer uso de esta técnica.

La librería *GPy*, ya explicada con anterioridad en este documento, permite de manera muy simple determinar el modelo de GP requerido para cada escenario. Como se veía en el Código 6.7 en la línea 12, la elección del modelo usado se lleva a cabo de manera trivial a partir del método *GPRegression*.

Para usar el modelo *sparse* GP con el fin de reducir el tiempo de ejecución total, basta con hacer el cambio mostrado en el Código 6.10.

Código 6.10 Selección del modelo *sparse* GP.

```
1 num_inputs=150
2 mC=GPy.models.SparseGPRegression(Xtrain, Prev, kern, num_inducing=num_inputs)
3 mC.optimize
```

En dicho código se aprecia como únicamente es necesario hacer uso del método *SparseGPRegression* para seleccionar el modelo con reducción de muestras en lugar del modelo de regresión convencional. Además, se observa como en la línea 1 se indica el número de muestras que van a "codificar" la información correspondiente al conjunto total de muestras. Ni que decir tiene que una reducción de este valor implica un decremento en el tiempo total de ejecución pero en cambio, la estimación realizada es menos precisa.