UNIVERSITAT JAUME I

GRADO EN MATEMÁTICA COMPUTACIONAL

ESTANCIA EN PRÁCTICAS Y PROYECTO FINAL DE GRADO

Teorema de Ascoli-Arzelà

Autor:
Alberto HERNÁNDEZ GÓMEZ

Tutora académica:
Marina MURILLO ARCILA

Fecha de lectura: 22 de Julio de 2019
Curso académico 2018/2019

Resumen

El presente documento se encuentra dividido en dos partes: la primera se corresponde con la estancia en prácticas realizada en el laboratorio de producción vegetal y microbiología de la Universidad Miguel Hernández de Elche y la segunda parte detalla el trabajo final de grado, centrado en el teorema de Ascoli y Arzelá.

La estancia en prácticas se desarrolló en la Universidad Miguel Hernández, concretamente en el laboratorio de producción vegetal y microbiología del campus de San Juan de dicha universidad. El objetivo de la estancia, dirigida por el doctor Francisco Rodríguez Valera, consistía en mejorar los métodos ya existentes para el descubrimiento, detección y tratamiento de islas genómicas, pues la versión informática más reciente es del 2006 y en ella se siguen utilizando ciertos mecanismos obsoletos en su búsqueda.

Respecto al trabajo final de grado se estudió en profundidad el teorema de Ascoli-Arzelà detallando su prueba así como diversos ejemplos en los que queda patente la utilidad del mencionado teorema. Además se presentan todos los conceptos previos necesarios para su comprensión.

Índice general

Capítulo 1

Memoria de la estancia en prácticas

1.1. Introducción

La estancia en prácticas se realizó durante los meses de julio y agosto en la Universidad Miguel Hernández (UMH) de Elche, concretamente en el campus de San Juan. En dicho campus se halla el edificio Muhammad Al-Shafra donde está el laboratorio de producción vegetal y microbiología, entre otros. El estudio principal fue dirigido por el doctor Francisco Rodríguez Valera.

El objetivo del estudio era implementar un método informático para la detección y tratamiento de las islas genómicas en las diferentes secuencias de muestra que se desease. Aunque ya existían métodos con esta finalidad; estos resultaban muy poco eficientes ya que varios procesos se encontraban obsoletos, tales como el algoritmo empleado para la combinación y ensamblaje de las muestras o el hecho de que las comparaciones finales tuviesen que ser realizadas a mano por el usuario. Por ello, el principal objetivo del estudio llevado a cabo en mi estancia era reducir el tiempo que se necesitaba para analizar las muestras y obtener conclusiones, ya fuese mediante la optimización de los algoritmos existentes o mediante la implementación de nuevas estructuras de programación más veloces que las anteriores.

1.2. La empresa

El departamento donde se desarrolló mi estancia en prácticas se encuentra en el área de conocimiento de producción vegetal y microbiología de la UMH. Sus profesores imparten docencia en diversas titulaciones y participan en diferentes proyectos de investigación. Como el nombre del departamento sugiere, la docencia y las investigaciones se centran tanto en la producción vegetal (investigando proyectos como la protección de cultivos, jardinería o paisajismo) como en la microbiología (realizando estudios acerca de los procesos de la microbiología, biotecnología, enología o industrias agroalimentarias) [1].

1.3. Programas y lenguajes de programación empleados

A continuación se detalla el software informático y lenguaje de programación empleados a lo largo de la estancia en prácticas:

1.3.1. Python

Toda la programación se desarrolló íntegramente en el lenguaje de programación python (1.1), concretamente a partir de la versión 3.6. Python es un lenguaje de programación de propósito general interpretado de alto nivel estudiado en la asignatura MT1022 - Algoritmia. Este lenguaje enfatiza la lectura y visibilidad del código remarcando el uso de los espacios en blanco y saltos de línea.

```
def add5(x):
   return x+5

def dotwrite(ast):
   nodename = getNodename()
   label=symbol.sym_name.get(int(ast[0]),ast[0])
   print '    %s [label="%s' % (nodename, label),
   if isinstance(ast[1], str):
      if ast[1].strip():
         print '= %s"];' % ast[1]
      else:
         print '"]'
   else:
      print '"];'
      children = []
      for n, child in enumerate(ast[1:]):
         children.append(dotwrite(child))
      print '    %s -> {' % nodename,
      for name in children:
         print '%s' % name,
```

FIGURA 1.1: Fragmento de un código en python

1.3.2. PyCharm

Para usar este lenguaje de programación se empleó el software PyCharm (1.2), específicamente diseñado para este lenguaje. PyCharm es un entorno de desarrollo integrado (IDE) que proporciona análisis del código, una herramienta de depuración, una unidad de pruebas integrada y un sistema de restauración a versiones anteriores del código, entre otros.

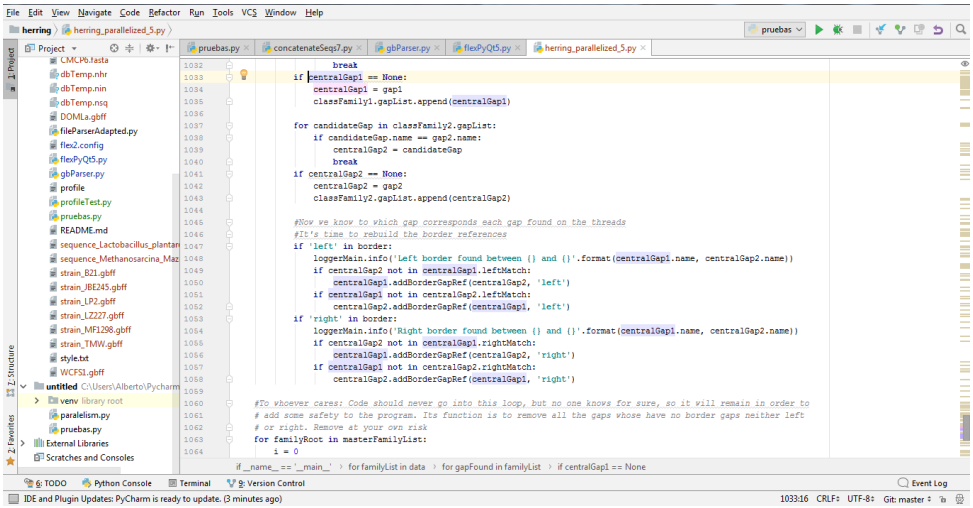

FIGURA 1.2: Captura de pantalla del programa pyCharm

1.3.3. MobaXterm

Por último, para acceder a los servidores del laboratorio y poder ejecutar los programas y realizar pruebas se empleó bash y mobaXterm (1.3). Bash es un procesador de comandos del shell Unix que se ejecuta en una ventana de texto donde el usuario escribe las órdenes que provocan acciones, cuyas consecuencias aparecen inmediatamente después de la orden. También es posible hacer que las órdenes se lean desde un archivo e incluso se puede elegir con qué lenguaje se desea que sean leídas. MobaXterm es un software para la programación remota desde Windows. Proporciona una gran cantidad de herramientas para dichas conexiones además de soportar tanto comandos Unix como bash.

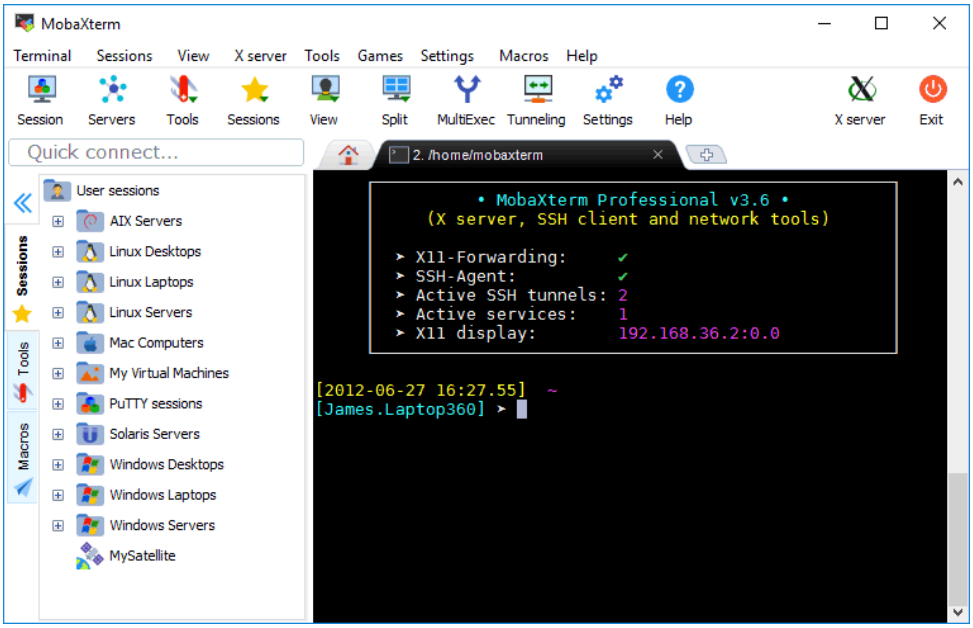

FIGURA 1.3: Captura de pantalla del programa mobaXterm

1.4. Objetivo

El trabajo realizado consistía en mejorar los métodos ya existentes para el descubrimiento, detección y tratamiento de islas genómicas, pues la versión informática más reciente es del 2006 y en ella se siguen haciendo patentes ciertos mecanismos obsoletos en su búsqueda. Sin embargo, ésta es la mejor versión disponible, pues realizar esta búsqueda a mano supone llevar a cabo un gran número de comparaciones sólo entre una muestra y la muestra referente, y dicho número se incrementa notablemente si se desean añadir más muestras a tratar.

Dado que el concepto de las islas genómicas es un estudio que dista de los objetivos del grado intentaré en este documento ofrecer una explicación que profundice sólo en aquellos aspectos relevantes para el trabajo, pero sin olvidar en ningún momento que las mejoras en su búsqueda y detección conforman un tema de suma relevancia dentro del área de la bioinformática.

Las islas genómicas son fragmentos del genoma que presentan variaciones respecto a un genoma de referencia. Dichas variaciones en la secuencia pueden estar codificadas para diferentes funciones y suelen tener un tamaño relativamente grande. Para que el lector pueda hacerse una idea, en la bacteria *lactobacillus plantarum* sus dos secuencias de referencia tienen una longitud de 5 Mb (mega pares de bases) mientras que una isla genómica de dicha bacteria rondaría los 10 kb.

Su correcto descubrimiento y la información que ofrecen resultan importantes para el área de la biología, y éste es el objetivo del trabajo a realizar. Ahora bien, para descubrir qué secciones de las hebras son islas genómicas nos apoyaremos en un mecanismo denominado blast. Este mecanismo comparará las dos secuencias, la muestra y la referencia, para así marcar las secciones con secuencias de pares de bases idénticas o casi idénticas (según el microorganismo a tratar se aceptará un porcentaje de error en el blast superior o inferior); de forma que si múltiples muestras presentan una misma zona en la que sus secuencias no son similares dicha sección es candidata a ser una isla genómica.

1.5. Conceptos previos

A lo largo de esta sección se introducen algunos conceptos del área de biología y que es posible que el lector desconozca.

1.5.1. Genoma

El genoma es el conjunto de genes contenidos en los cromosomas y, puesto que cada cromosoma es una estructura altamente organizada que contiene una gran parte de información genética del ser vivo; el genoma se puede interpretar como la totalidad del material genético que posee un organismo.

Cabe señalar, además, que los organismos diploides tienen dos copias del genoma en sus células debido a la presencia de pares de cromosomas homólogos. Por contra, los organismos haploides sólo tienen una copia. Esto se traducirá más adelante en dos secuencias o una por muestra.

1.5.2. Secuencia

La secuenciación del genoma es un proceso de laboratorio que determina la secuencia completa de ADN del genoma de un organismo. Aunque suene similar a un análisis de ADN no se deben confundir, pues el análisis sólo determina la posibilidad de que el material genético provenga de un individuo o grupo en particular mientras que la secuenciación, al tener en consideración la distribución del material génico, puede hallar relaciones genéticas que determinen, por ejemplo, la susceptibilidad a enfermedades específicas.

La secuenciación no proporciona información clínica inmediata pero su estudio puede llevar a la detección de variantes genéticas específicas y/o enfermedades asociadas. Una forma de realizar este estudio es mediante la detección de islas genómicas.

1.5.3. Isla genómica

Una isla genómica es una parte de un genoma de la que se tienen evidencias de variaciones respecto a un genoma de referencia. Dichas variaciones pueden estar codificadas para diferentes funciones, como simbiosis o patogénesis, y pueden ayudar en la adaptación de un organismo. La misma isla genómica se puede encontrar en especies lejanamente relacionadas como consecuencia de la transferencia horizontal de genes.

Las islas genómicas se caracterizan por su gran tamaño: entre 10 y 500 kb (kilo bases o kilo pares de bases). Muchas están flanqueadas por estructuras repetidas y contienen fragmentos de elementos móviles como fagos y plásmidos.

1.5.4. PCR

PCR es el acrónimo inglés de "polymerase chain reaction" (en español reacción en cadena de la polimerasa). Se trata de una técnica de la biología molecular cuyo objetivo es obtener un gran número de copias de un fragmento de ADN. En teoría, con una única copia del fragmento original (o molde) es posible obtener todas las réplicas que se deseen. De esta forma se amplifica un único fragmento para su posterior estudio y/o análisis de forma más fácil al haber mayor cantidad.

Debido a su bajo coste y gran cantidad de resultados se trata de una técnica muy extendida en prácticamente todas las áreas de la biología. No obstante, hay que tener en cuenta que este proceso puede producir más copias de las deseadas. El molde a replicar se deja en la solución de polimerasa (tras ser debidamente tratado) durante un tiempo proporcional al número de copias: a mayor tiempo en la solución, más duplicados. Si en la solución hay varias secuencias puede que algunas de ellas se repliquen más veces que otras, pudiendo generar más duplicados de esa(s) secuencia(s).

1.5.5. Contig

Los contigs (en español cóntigos) son segmentos de ADN sobrepuestos que juntos denotan la secuencia genética. Son empleados en dos tipos de secuenciación principalmente:

- En la bottom-up sequencing (secuenciación "abajo arriba") se obtienen los contigs en base a una secuencia de referencia por superposición de la información genética, motivo por el que no resulta raro que todos los contigs de una muestra tengan una longitud de pares de bases superior a la muestra original.

- En la top-down sequencing (secuenciación "arriba abajo") se produce el caso contrario: estos clones son superpuestos y ensamblados de modo que forman un mapa completo del genoma empleado para su creación.

Cuando se trabaja con contigs suelen ocurrir ambos tipos de secuenciación indisitintamente, sin embargo en este trabajo sólo ocurre la secuenciación top-down, ya que el objetivo es reagrupar y unir los contigs para que estos formen de nuevo el mapa completo del genoma, es decir, la secuencia muestra.

1.5.6. Blast

BLAST es un acrónimo propio del sector de la bioinformática que significa Basic Local Alignment Search Tool. Se trata de un algoritmo encargado de comparar la información de la secuencia biológica primaria, tales como las secuencias de aminoácidos presentes en proteínas de los nucleótidos de secuencias de ADN y ARN.

Una búsqueda BLAST (o, para abreviar, un blast) permite al investigador comparar una secuencia muestra con una secuencia referente, la cual se conoce o bien porque proviene de una biblioteca propia (ciertas muestras de la biología no se comparten hasta que no son publicadas y permanecen sólo a disposición de la entidad u organismo que los consigue) o de una de las bases de datos de internet de las mismas. Además se puede configurar el blast para que asocie o no las muestras en base a un nivel umbral determinado.

1.6. El trabajo a realizar

Para realizar nuestra tarea se tratará de crear una interfaz que el usuario sea capaz de manejar para así visualizar de una manera clara qué fragmentos hay en común, cuales no, y ser capaz de determinar si alguno de ellos es susceptible de ser una isla genómica. Este proyecto es mucho más ambicioso de lo que parece, especialmente para el área de la biología, y no es posible realizar todo el proyecto en los dos meses de mi estancia. Por ello mi labor se centró en tres partes de programación, descritas en detalle a continuación.

1.6.1. Organización y ensamblaje de secuencias

La primera parte del trabajo consistió en la creación, desarrollo e implementación de un algoritmo para el ensamblaje de contigs en base a una secuencia de referencia. Dicho programa recibe del usuario dos archivos: uno de ellos es el core, es decir, la secuencia de referencia a utilizar; el otro es un fichero de texto con las secuencias de los contigs. Dicho de otra forma, a este programa se le pasará un core (o más bien dos secuencias de referencia) y una muestra a valorar. Esa muestra se compone de una gran cantidad de contigs, es decir, pequeños fragmentos de secuencia con tamaño variable obtenidos tras muestrear una secuencia genética del microorganismo deseado. No obstante, dichos contigs se encuentran totalmente desordenados, estado en el que no son útiles para realizar las comparaciones necesarias.

Esto es lo que trata de solucionar el programa concatenateSeqs.py, presentado en el anexo A. Éste recibe el fichero y realiza blasts entre cada uno de los contigs y las secuencias de referencia. Tras dichos blasts el programa clasifica los contigs:

- En primer lugar, si la secuencia de referencia tiene dos hebras hay que determinar a cuál de las dos referencia el contig. Recordemos que al hacer el blast no se solicita una secuencia idéntica, sino que el porcentaje de acierto sea superior al valor umbral. Por ello se puede dar el caso de que, al realizar el blast, existan zonas con un gran porcentaje de similaridad en las dos hebras y, en estos casos, el programa debe escoger a qué referencia pertenece el contig para el correcto alineamiento y ensamblaje posterior.

- A continuación, se organizan los contigs en función a su posición mediante el blast más representativo. Las posiciones inicial y final de dicho blast en la secuencia de referencia marcarán la posición del contig respecto a otros contigs.

- Por último, se ensamblan los contigs que pertenezcan a la misma secuencia de referencia y cuyas posiciones en dicha secuencia se hallen a menor distancia de un valor umbral que determina el propio usuario y, si éste usa un valor negativo, el programa unirá todos los contigs en orden sin importar su distancia.

Finalmente el programa devuelve un fichero con los contigs reorganizados y unidos en base al umbral especificado. Además, el programa trata una serie de excepciones que, desgraciadamente, son habituales por la forma en la que se obtienen las muestras en el área de la biología y su transcripción a medios informáticos.

Tras realizarse el muestreo no se sabe la orientación de los contigs, y por ello estos pueden tener distinto sentido. Por tanto, si la posición inicial del contig se corresponde (mediante el blast) con la posición mayor de la referencia; dicho contig se debe invertir, lo que supone reescribir los valores inicial y final así como la secuencia. Recordemos que estamos tratando con pares de bases. Al invertir una secuencia no basta con tomar toda la cadena de caracteres y reescribirlos de final a principio, sino que hay que escribir el aminoácido asociado.

Para medir los contigs se realizan una serie de PCRs lo que puede conllevar una duplicación de los contigs de manera parcial o total. En estos casos el programa determina si existen estas copias y, de haberlas, el programa elimina aquellas que aportan menor información; las cuales son por norma general las de menor tamaño.

Ciertas secuencias, sobre todo las de los microorganismos, poseen la información genética circular. A la hora de generar los contigs esta secuencia circular se rompe y después es tratada como si fuese lineal. Por lo tanto es posible que exista un contig cuyo blast comience en el final de la secuencia de referencia y acabe al principio de la misma. En este caso ese contig se debe partir en dos nuevos contigs de forma que cada uno de ellos tenga un blast al inicio y al final de la referencia, pero no en ambos sitios.

A fin de determinar la validez y fiabilidad del algoritmo se realizaron alrededor de 100 pruebas distintas de las cuales se poseen registros que se podrían facilitar en caso de ser necesario.

1.6.2. Comparaciones múltiples entre distintas muestras

El segundo trabajo realizado se basó en la optimización del tiempo requerido por el programa herring.py. Su labor es la de realizar las comparaciones dos a dos entre las distintas muestras con el fin de hallar las islas genómicas. Por supuesto, este programa no las obtiene directamente, sino que al comparar múltiples muestras obtiene las secuencias donde existen blasts comunes, de modo que de este resultado se pueden determinar las secuencias que no son comunes y que, por tanto, son candidatas a ser islas genómicas.

En este caso sí que existía una versión del código, sin embargo dicha versión resultaba tremendamente ineficiente debido a los altos tiempos de espera que debía soportar el usuario. Esto no es de extrañar pues el programa debe realizar comparaciones dos a dos entre las muestras de modo que cubra todas las combinaciones, es decir, con 4 muestras serían necesarias 10 combinaciones, pero 14 muestras elevan la cifra a 105, que suponen alrededor de 30 horas de procesamiento para el programa. Recordemos que siempre hay que tomar la secuencia core o referencia, por lo que siempre hay una muestra adicional.

Para satisfacer las expectativas del laboratorio se reescribió el anterior algoritmo dando lugar a una nueva versión del código (presente en el anexo A) en la cual se eliminan fragmentos innecesarios, se optimizan ciertas búsquedas al emplear algoritmos de búsqueda binaria (tratados en la asignatura MT1022 - Algoritmia), se tratan ciertos problemas que ocasionaban la versión inicial y, sobre todo, se añadió la opción de realizar los procesos de forma concurrente y paralela para un número determinado de núcleos que eran escogidos por el usuario por medio de ThreadPools como se tratan en la asignatura MT1024 - Programación concurrente y paralela pero aplicándolos al lenguaje de programación python.

Se realizaron numerosas comprobaciones para asegurar su fiabilidad y tener cierta garantía de su correcto funcionamiento. Por ejemplo, tratando la bacteria *lactobacillus plantarum* con 14 muestras (105 combinaciones) el programa inicial necesitaba más de 30 horas para producir un resultado; mientras que con la versión paralelizada y usando 8 núcleos el programa finaliza tras 4 horas.

1.6.3. Interfaz gráfica

Por último se realizó otra labor de optimización para reducir el tiempo que necesita la interfaz gráfica para generar el resultado de una forma visible y cómoda para el usuario. En este caso sólo se podían emplear algoritmos más eficientes pues no era factible implementar una estructura concurrente y paralela. Además el código flexPyQt.py, se apoya a su vez en otros dos, gbParser.py y blastParser.py, lo que dificultaba su lectura y tratamiento. Como el último no fue modificado sólo se adjuntan los otros dos en el anexo A.

Además se implementaron diversas búsquedas binarias, se limpió el código de fragmentos redundantes y se redujo la cantidad de variables en memoria. Para conocer el tiempo que empleaba el programa se usó la herramienta profiler. Se adjuntan los resultados de dicha herramienta antes de las modificaciones (Cuadro 1.1) y tras las mismas (Cuadro 1.2) en una prueba con 8 muestras donde se observan las llamadas que más tiempo emplean. Los tiempos pasan a ser de 6 min a 3 min y 20 seg. Se puede encontrar una explicación acerca de la herramienta profiler y el significado de las columnas en el anexo B.

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
50616	79.400	0.002	79.400	0.002	gbParser.py:57(_checkDuplicates)
1	59.314	59.314	354.047	354.047	{built-in method exec_}
25395	54.022	0.002	54.022	0.002	{built-in method sceneRect}
24	49.519	2.063	49.519	2.063	{built-in method _winapi: WaitForSingleObject}
2	16.248	8.124	16.248	8.124	{getOpenFileNames}

CUADRO 1.1: Primeros resultados de profiler con 8 genbanks antes de las modificaciones

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
12	46.213	3.851	46.213	3.851	{built-in method _winapi: WaitForSingleObject}
22053	43.950	0.002	43.950	0.002	{built-in method sceneRect}
1	34.230	34.230	197.212	197.212	{built-in method exec_}
1	5.890	5.890	5.890	5.890	{getOpenFileNames}
50635	4.275	0.000	7.427	0.000	Scanner.py:217(parse_feature)

CUADRO 1.2: Primeros resultados de profiler con 8 genbanks tras las modificaciones

1.7. Planificación temporal de las tareas

1.7.1. Primer informe

El primer día se me enseña el campus de San Juan así como el laboratorio de microbiología y producción vegetal. Soy presentado al resto de compañeros, especialmente a aquellos que también trabajan en el mismo proyecto. A uno de ellos se le encarga informarme del estado actual del proyecto y las áreas en las que se espera que yo trabaje. Además me explica los conceptos de biología necesarios para llevar a cabo el proyecto y el diseño que se espera que logre el programa. Por último recibo las claves y direcciones IP de los servidores a disposición del laboratorio y las versiones más recientes de los códigos que conforman el programa en el que voy a trabajar.

Para la primera quincena se me solicita un objetivo a priori sencillo: paralelizar el programa herring.py y optimizarlo en la medida que sea posible junto a una primera toma de contacto con el programa para comprender qué hace cada parte y su funcionamiento. La paralelización en sí no es un objetivo difícil, pero nunca la había implementado en python (cuando se trabaja en la asignatura MT1024 - Programación concurrente y paralela sólo se programa en java) por lo que el problema en esta parte residía en buscar las estructuras equivalentes en python a las ya conocidas en java y aplicarlas de forma correcta. Respecto a la optimización comencé aplicando búsquedas binarias, pues ya las había tratado con anterioridad (en la asignatura MT1022 - Algoritmia) de modo que me sirvieron para continuar familiarizándome con el código mientras las implementaba.

Este primer objetivo se vió interrumpido de repente cuando uno de los compañeros se tuvo que retirar del proyecto por motivos personales, de modo que se me encargó realizar su parte, pues era prioritaria.

1.7.2. Segundo informe

Debido a la ausencia del compañero se tuvieron que posponer las tareas que se me habían asignado. En ese momento debía trabajar en mejorar la interfaz gráfica (el programa flexPyQt.py y los dos en los que se apoya, gpParser.py y blastParser.py) ya que éste es clave para mostrar los resultados y así poder probar si los demás códigos desarrollados funcionan correctamente. El compañero había dejado una interfaz gráfica que mostraba los parámetros tal y como se exigían pero resultaba muy lenta según el jefe del proyecto, por lo que mi objetivo se tradujo en optimizar el programa de forma que se redujese el tiempo que espera el usuario desde que introduce los datos hasta que el programa muestra los resultados en pantalla.

En primer lugar hallé una forma de medir el tiempo que necesitaba el programa, no sólo por su tiempo de finalización, sino también para conocer qué partes requerían más tiempo. Para ello, buscando por la web, encontré que python dispone del método profiler; el cual muestra, entre otras cosas, esos parámetros deseados.

Para implementar el método profiler es necesario incluir en el código una llamada inicial y una final al método, entre las cuales debe encontrarse el programa que se desea medir. Esto ya supuso un problema, pues la interfaz gráfica en aquel momento no se encontraba completamente automatizada, de modo que el programa no finalizaba si el programador no iba guiando dicha interfaz. Para solucionarlo añadí en la interfaz un comando adicional que se ejecutaba en cuanto el programa había mostrado los resultados. Dicho comando no realizaba ninguna función en la interfaz gráfica, pero en él se encontraba la segunda llamada al método profiler, de modo que, a efectos prácticos, el método consideraba que el programa había finalizado. Tras dicha finalización el método profiler muestra una tabla con los valores de tiempo consumido por cada uno de los métodos del programa ordenados de mayor a menor (se incluye una explicación del método profiler en el apéndice B).

Los datos del método profiler (presentes en este mismo documento en el cuadro 1.1) mostraban que el método checkDuplicates de gbParser.py consumía la mayor cantidad de tiempo. Un análisis de dicho método mostraba que se producían muchas más comparaciones de las necesarias, de modo que modifiqué dicha clase para hacerla más restrictiva a la hora de comparar los datos.

Volviendo a ejecutar el programa y analizando los datos del método profiler el siguiente método que consumía más tiempo era tryFastaFile, también de gbParser.py. Este método se encarga de comprobar si el fichero de las muestras es un genbank o un fasta y, según cuál sea, llamar al método encargado de su lectura. Para dicha lectura se empleaba un método propio de la libreria biopython (librería que contiene una gran cantidad de programas diseñados para procesos de bioinformática en python). Sin embargo, el método utilizado leía una gran cantidad de datos del fichero. Como la gran mayoría de esos datos no eran usados por el programa decidí reescribir la sección de forma que únicamente fuesen leídos los datos necesarios.

Al volver a leer los datos del método profiler se observaba que los métodos que más tiempo empleaban eran aquellos propios de python, es decir, métodos que no habían sido creados por los compañeros de trabajo y no pueden ser modificados. Además el tiempo se había reducido notablemente: se había conseguido una reducción del tiempo de espera cercana al 50 %, el cual aumentaba según las tareas o cantidad de muestras que se introducían. Presenté estos resultados a mi supervisor que los consideró válidos, por lo que pasamos al siguiente objetivo.

1.7.3. Tercer informe

El siguiente objetivo era mejorar el código concatenateSeqs.py, algoritmo encargado de recibir un archivo con contigs de muestra, reordenarlos y unirlos en el orden apropiado y según ciertos parámetros definidos por el usuario. Al igual que en el caso anterior debía mejorar el tiempo que emplea, pero eso era secundario. El mayor problema del código era que su versión actual producía fallos si la muestra que recibía tenía alguna complicación, por mínima que fuese. Además, el programa cometía fallos tales como dar por erróneos contigs con gran información genética, no invertir los contigs con sentido contrario o no descartar los duplicados. Puesto que era difícil conocer de qué parte del código provenían los errores decidimos, tras debatirlo con mi supervisor, elaborar un nuevo concatenateSeqs.py.

Al realizar nuevas versiones del código descubrí importantes errores que debía corregir. Primero, existían problemas a la hora de definir dónde comienzan y acaban las referencias de los contigs en la secuencia principal. Arreglando esto surgieron problemas a la hora de identificar a qué secuencia pertenecía cada contig. En segundo lugar, el código tenía problemas para distinguir qué contigs estaban duplicados y/o cuáles no tenían secuencias idénticas a la secuencia de referencia. En tercer lugar, el código tenía problemas para identificar cuándo había que cortar los contigs, especialmente cuando el contig estaba formado por varios fragmentos a romper (porque dichos fragmentos debían colocarse entre otros contigs). Por último el programa presentaba fallos a la hora de identificar cuándo un contig se encontraba invertido y definir el procedimiento para revertirlo y reescribir la nueva secuencia. Pero el principal problema era cómo identificar qué contigs se tenían que unir, ya que había que definir la posición de cada uno, la distancia entre ellos y, si se unían, formar correctamente el nuevo contig uniendo las secuencias de cada uno en orden correcto. Además había que definir los valores de la posición inicial y final de su referencia a la secuencia principal y, por supuesto, eliminar del fichero de contigs los dos contigs anteriores y escribir el nuevo contig resultante en su lugar.

Por cuestiones de seguridad y precaución creaba nuevas versiones del código cuando me disponía a realizar cambios importantes o pasado un determinado tiempo. La versión presentada en este documento es la octava versión del código. Sin embargo y para asegurar al equipo que el código era lo bastante robusto realicé pruebas con 97 muestras distintas incluyendo familias del mismo organismo y familias distintas, así como muestras procedentes de distintos bancos de genomas o almacenadas en distintos formatos. Cuando todas las pruebas dieron resultados correctos las presenté a mi supervisor y compañeros de trabajo. Tras su visto bueno di por concluida esta parte.

1.7.4. Cuarto informe

A la vista de mis resultados se me pidió regresar al código herring.py para corregir dos situaciones en las que fallaba el programa: cuándo se debe eliminar un segmento de un blast para que su posición inicial o final (según corresponda) se ajuste al resto de blast de las otras secuencias (ver figura 1.4) y cuándo se debe eliminar un blast si tiene un tamaño relativamente pequeño comparado al de la zona candidata a ser isla genómica en la que se encuentra (ver figura 1.5). El objetivo de eliminar esos blasts es ampliar las zonas que pueden ser islas genómicas para su posterior estudio.

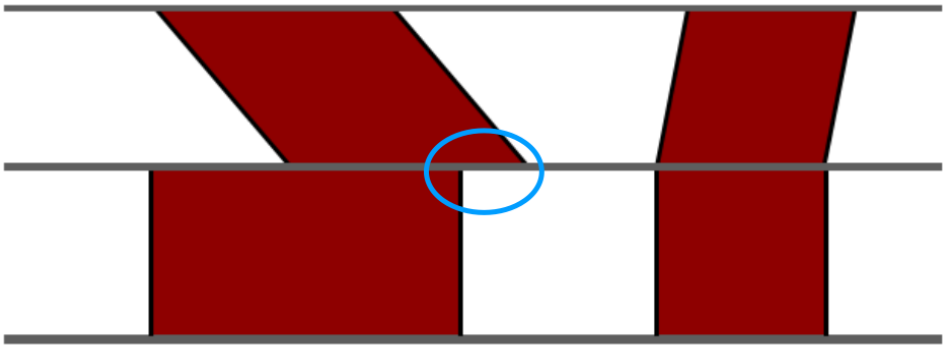

FIGURA 1.4: Error tipo 1: La parte de blast señalada debe ser eliminada

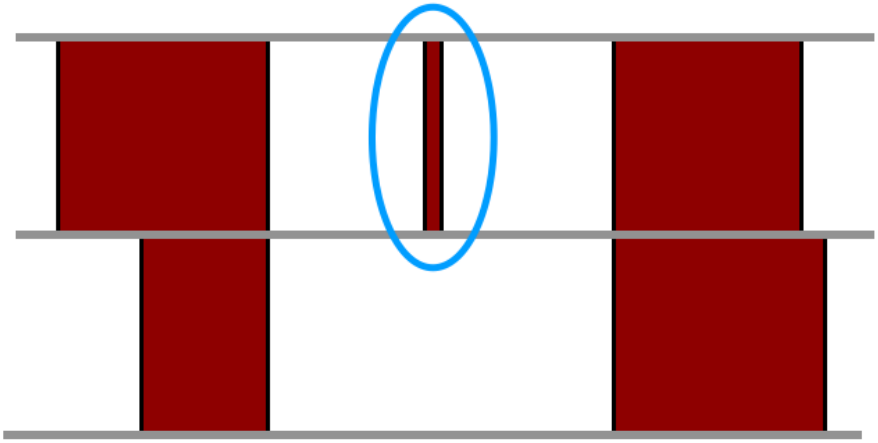

FIGURA 1.5: Error tipo 2: El blast señalado debe ser descartado

El problema radicaba en la detección de los bordes derecho e izquierdo de los blasts: cada contig no tenía problema para detectar sus extremos pero surgían errores para obtener el siguiente borde. Tras la modificación de diversos métodos y depuración de cierto código redundante se logró el resultado esperado, pero con un problema, ya que algunos resultados eran imprimidos por duplicado en el fichero resultante. Corregir este último fallo fue una tarea más tediosa de lo que podría parecer. El problema radicaba en que las comparaciones se realizaban por huecos (las zonas en las que no habían blasts) dos a dos, sin embargo no se verificaba en ningún momento si uno de esos huecos ya había sido valorado, dando lugar a tratar uno de esos tipos de errores por duplicado. Con ese cambio se logró la solución actual.

Los últimos días, para facilitar la labor de mis compañeros, me dediqué a anotar comentarios y explicaciones en los códigos, especialmente en aquellas partes que había modificado y/o creado.

1.8. Ejemplos de ejecución

Los tres programas anteriores trabajan para ofrecer una imagen detallada de las muestras tratadas y la referencia. Como se puede observar en la figura 1.6, el programa siempre intenta ubicar los genomas de referencia en el centro mientras que los contigs procedentes de las muestras se colocan alrededor, de forma que no resulta difícil observar la ordenación que los contigs deben llevar para asemejarse al genoma de referencia. Los paralelogramos rojos representan los blasts, o mejor dicho, representan dos zonas que el blast ha determinado que tienen una secuencia muy similar (y en el mismo orden).

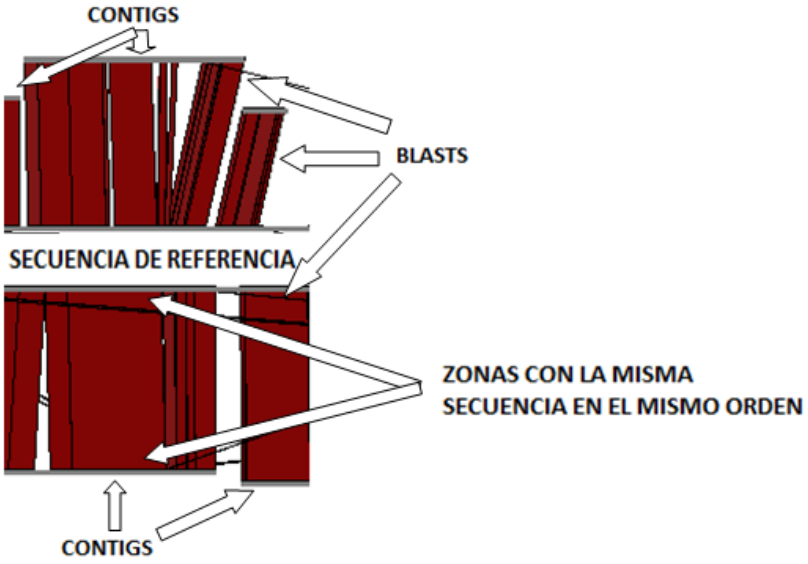

FIGURA 1.6: Fragmento de la salida del programa señalando la ubicación de los distintos elementos de interés para el usuario

También cabe señalar que de un mismo contig pueden encontrarse varias secuencias idénticas a la del contig (este hecho se observa de forma muy clara en el gráfico 1.7). Esto se debe a que es posible que una misma secuencia se repita en varios sectores, pues recordemos que la información genética se compone de 4 aminoácidos distintos. Es tarea del programa determinar qué blast es el más representativo y ubicar el contig en consecuencia.

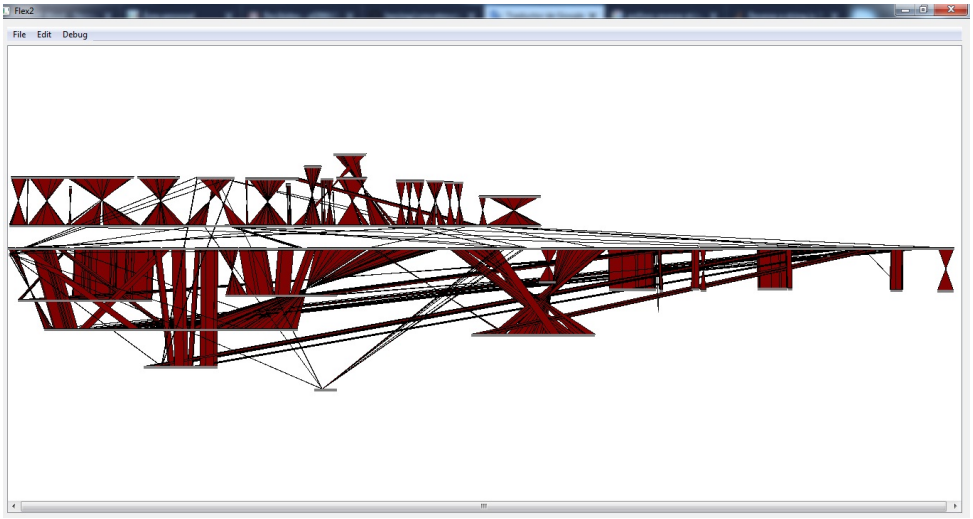

FIGURA 1.7: Ejemplo de fallo en el análisis de la archaea *Vibrio Vulnificus CMCP6*

Para finalizar se ofrece al lector una muestra de la salida gráfica que ofrece el conjunto de programas al analizar dos muestras de microorganimos de distintas clases: bacterias (Figura 1.8) y archaeas (Figura 1.9) las cuales se pueden comparar con la misma muestra de archaea evaluada con la versión inicial de código (Figura 1.7)

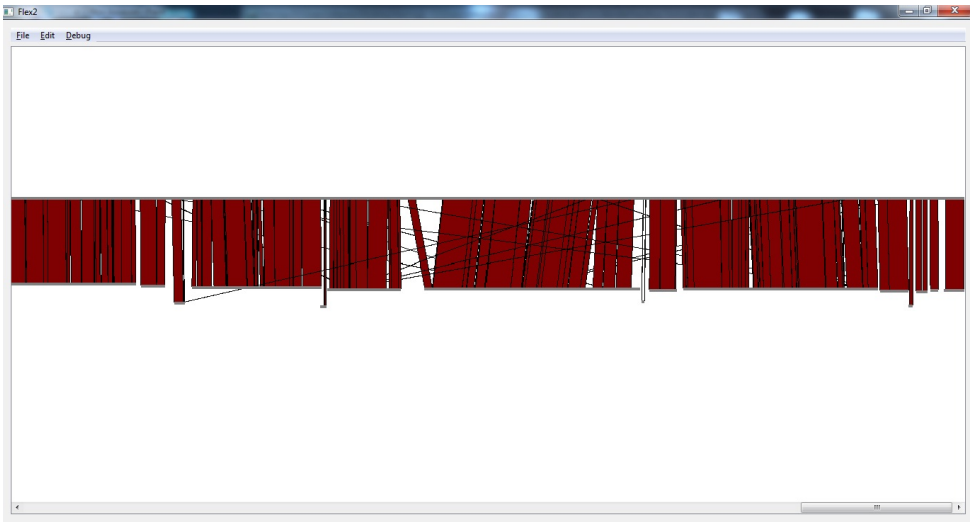

FIGURA 1.8: Análisis de una muestra (línea inferior) y del genoma de referencia (linea superior) de la bacteria *lactobacillus plantarum*

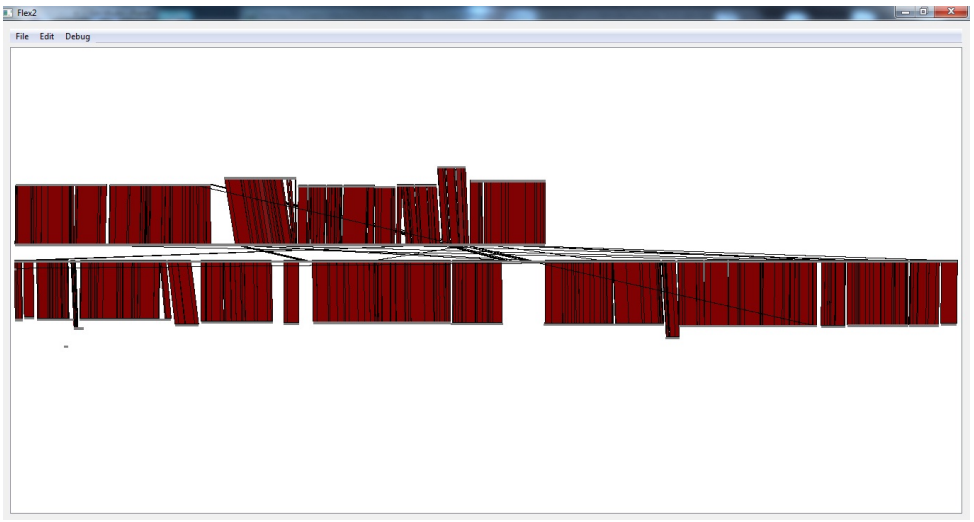

FIGURA 1.9: Análisis de una muestra (líneas exteriores) y del genoma de referencia (lineas centrales) de la archaea *Vibrio Vulnificus CMCP6*

En los casos de una única secuencia, como las bacterias, su representación resulta trivial. En los casos de dos secuencias, como podemos observar con las archaeas, hay que tratar todas las muestras de la mejor forma posible de manera que siga resultando fácilmente visible. Por convenio se sitúan las dos secuencias de referencia en el centro del programa y encima y debajo de éstas los contigs que referencian a cada una.

Como último apunte señalar que el programa ofrece al usuario la posibilidad de personalizar la salida. Por ejemplo se puede modificar el color de los blasts, observable en las figuras 1.10 y 1.11.

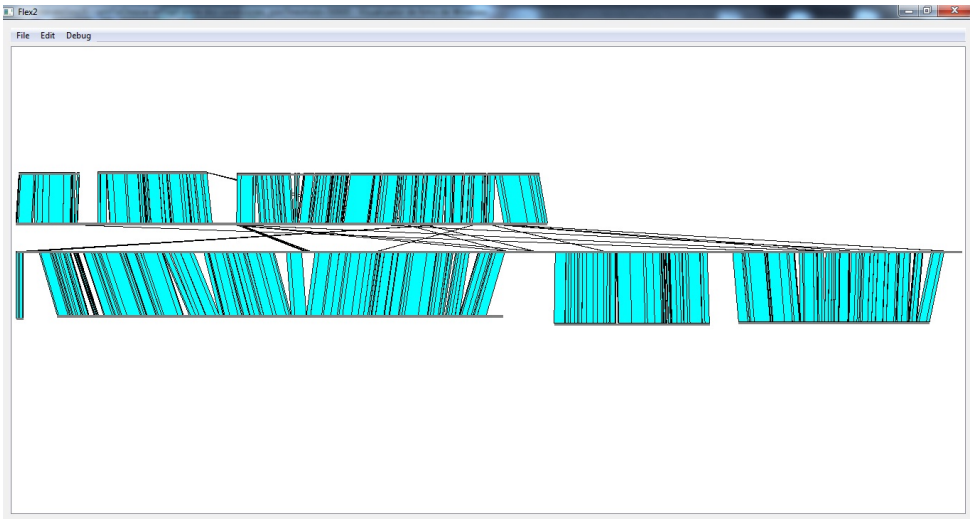

FIGURA 1.10: Blasts coloreados de azul en la muestra de la archaea *Vibrio Vulnificus CMCP6*

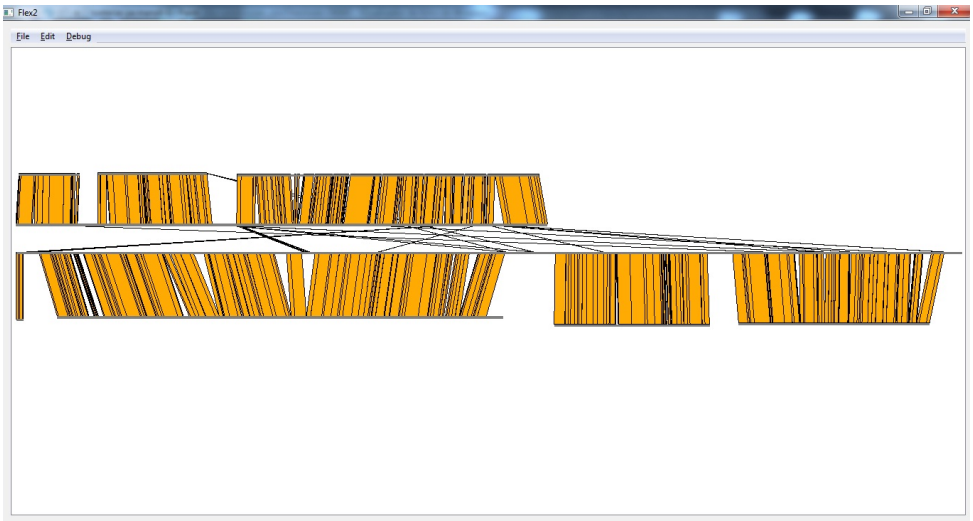

FIGURA 1.11: Blasts coloreados de amarillo en la muestra de la archaea *Vibrio Vulnificus CMCP6*

Capítulo 2

Memoria TFG

2.1. Introducción

El análisis funcional ha sido definido por **J. Dieudonné** como "*... el estudio de los espacios topológicos y de las aplicaciones definidas entre subconjuntos de los mismos, sujetas a distintas condiciones algebraicas y topológicas.*". Dentro de su ambigüedad, la definición de Dieudonné pone de manifiesto alguna de las características más importantes del análisis funcional: la tendencia hacia la algebrización del análisis, el énfasis en los resultados de carácter estructural y la fuerte influencia de la topología. De hecho, como el propio Dieudonné señala, es prácticamente imposible disociar los comienzos de la Topología General y el Análisis Funcional.

Como toda teoría matemática, el análisis funcional surge de la necesidad de encontrar nuevas técnicas para abordar una serie de problemas que los métodos tradicionales no podían resolver.

Probablemente los antecedentes más claros del Análisis Funcional se pueden encontrar en el Cálculo de Variaciones. Con este nombre se conoce una serie de problemas en los que se trata de maximizar o minimizar no ya una función real definida sobre un subconjunto de $\mathbb{R}^n$, sino una expresión del tipo

$$J(\varphi) = \int_a^b F(\varphi(x), \varphi'(x), \ldots)dx$$

siendo F una función regular, y las "variables" φ un adecuado conjunto de curvas regulares parametrizadas en $[a, b]$. Es en este contexto donde aparece primero la idea de *campo funcional*, como conjunto de funciones admisibles, y la de *distancia entre funciones*.

Los esfuerzos de Weierstrass y su escuela consiguieron fundamentar rigurosamente la mayor parte de los resultados y argumentos clásicos del Cálculo de Variaciones, salvo el principio general de existencia de extremo. Este fallo es debido a que los teoremas generales que implican la existencia de extremos de una función real (*continua*), están basados en la noción de *compacidad*. De hecho, la primera demostración rigurosa del principio de Dirichlet fue dada por **D. Hilbert** alrededor de 1900. Para ello, Hilbert tuvo que redescubrir una versión del que puede considerarse uno de los primeros teoremas del Análisis Funcional: el teorema de Ascoli-Arzelà.

Uno de los primeros problemas planteados con la rigorización del Análisis, fue estudiar condiciones bajo las cuales el límite (puntual) de una sucesión de funciones, conserva las buenas propiedades que pudieran tener las funciones de la sucesión (continuidad, derivabilidad, etc). Los primeros intentos en esta dirección, consistieron en imponer condiciones más restrictivas sobre la forma de converger de la sucesión. Así surgió la noción de *convergencia uniforme* (Weierstrass, 1841; Stokes, 1847; Von Seidel, 1848; Cauchy, 1853). La postura de los italianos **U. Dini, G. Ascoli** y **C. Arzelà**, fue radicalmente diferente. En lugar de modificar la noción de convergencia empleada, dieron una condición general sobre el conjunto formado por la sucesión de funciones (la *equicontinuidad*: Ascoli, 1883), de tal modo que el límite puntual es necesariamente continuo. Los trabajos posteriores permitieron demostrar que toda sucesión equicontinua de funciones acotadas sobre un cerrado y acotado de $\mathbb{R}^n$, posee una subsucesión uniformemente convergente, extendiendo así el clásico teorema de Bolzano para conjuntos acotados de $\mathbb{R}$ a conjuntos de funciones. Una versión de este teorema de compacidad en espacios funcionales es la que redescubrió Hilbert para su demostración del Principio de Dirichlet.

En nuestro trabajo final de grado nos centraremos en proporcionar una prueba detallada del teorema clásico de Ascoli-Arzelà.

2.2. Conceptos previos

Para comprender el teorema de Ascoli-Arzelà introduciremos ciertos conceptos previos correspondientes al área de topología y análisis, algunos de los cuales ya fueron tratados en las asignatura MT1027 - Topología, además de otros conceptos nuevos [3] [4] [5].

Definición 2.2.1. *Un conjunto X y una aplicación $\alpha : X \times X \to X$ (donde $\alpha(a,b) = a+b$) definen un grupo abeliano siempre que:*

1. $a + (b + c) = (a + b) + c$ *para todo* a, b, c.
2. $a + b = b + a$ *para todo* a, b.
3. *Existe un elemento* $0 \in X$ *tal que* $a + 0 = a$.
4. *Para cada* a *existe un elemento* a' *el cual verifica que* $a + a' = 0$.

Definición 2.2.2. *Sea $\mathbb{R}$ el conjunto de números reales con las operaciones suma y multiplicación usuales. Un grupo abeliano X junto a una aplicación $m : \mathbb{R} \times X \to X$ (donde $m(\lambda, a) = \lambda a$) es un **espacio vectorial real** siempre que, para todo λ, μ, a, b:*

1. $\lambda(a + b) = \lambda a + \lambda b$.
2. $(\lambda + \mu)a = \lambda a + \mu a$.
3. $(\lambda\mu)a = \lambda(\mu a)$.
4. $1 \times a = a$.

Definición 2.2.3. *Un **espacio métrico** es un par (X, d) formado por un conjunto X y una distancia d sobre X.*

Una distancia (o una métrica) sobre un conjunto X es una aplicación $d : X \times X \to \mathbb{R}$ con las propiedades siguientes:

1. $d(x, y) \geq 0$ *para todo* $x, y \in X$.
2. $d(x, y) = 0$ *si y sólo si* $x = y$.

3. *Propiedad de simetría:* $d(x,y) = d(y,x)$ *para todo* $x, y \in X$.
4. *Desigualdad triangular:* $d(x,z) \leq d(x,y) + d(y,z)$ *para todo* $x, y, z \in X$.

Definición 2.2.4. *Llamamos* ***bola abierta de radio*** δ *centrada en el punto* x_0 *al conjunto:*

$$B_\delta(x_0) = B(x_0, \delta) = \{x \in X : d(x, x_0) < \delta\}.$$

Definición 2.2.5. *Sean* (X,d) *y* (Y,ρ) *dos espacios métricos, y sea* $x_0 \in X$. *La aplicación* $f : (X,d) \to (Y,\rho)$ *es* ***continua*** *en* $x_0 \in X$ *si para cada* $\epsilon > 0$, *existe un* $\delta > 0$ *que satisface* $\rho(f(x), f(x_0)) < \epsilon$ *para todo* $x \in X$ *con* $d(x, x_0) < \delta$.

Definición 2.2.6. *Sea X un conjunto. Una topología en X es una familia* τ *de subconjuntos de X que satisface:*

1. *Cada unión de miembros de* τ *es también miembro de* τ.
2. *Cada intersección finita de miembros de* τ *es también miembro de* τ.
3. $\emptyset$ *y X son miembros de* τ.

Un par (X,τ) *consistente en un conjunto X y una topología* τ *en X se denomina* ***espacio topológico***.

Definición 2.2.7. *Sea* (X,τ) *un espacio topológico y sea* $A \subset X$. *Definimos:*

$$\textit{Clausura de } A\text{: } \bar{A} = \bigcap\{C \subset X : A \subset C \text{ y } C \text{ es cerrado}\}.$$

$$\textit{Interior de } A\text{: } \mathring{A} = \bigcup\{O \subset X : O \subset A \text{ y } O \text{ es abierto}\}.$$

Definición 2.2.8. *Decimos que un espacio topológico* (X,τ) *es* ***compacto*** *si para todo recubrimiento de X formado por conjuntos abiertos,* $X = \bigcup_{i \in I} V_i$, *es siempre posible encontrar un subrecubrimiento finito; esto es, una cantidad finita de abiertos* $V_{j1}, V_{j2}, \ldots, V_{jn}$ *tal que* $X = \bigcup_{k=1}^{n} V_{jk}$.

Definición 2.2.9. *Decimos que una sucesión* $(x_n)_{n=1}^{\infty} \subset (X,d)$ *es* ***de Cauchy*** *si dado* $\epsilon > 0$ *existe* n_0 *tal que para todos* $n, n' \geq n_0$ *se tiene que* $d(x_n, x_{n'}) < \epsilon$.

Definición 2.2.10. *Decimos que un espacio métrico (X, d) es **completo** si toda **sucesión de Cauchy** converge a algún punto de (X, d).*

Denotaremos como $C(X)$ al espacio de funciones continuas, es decir:

$$C(X) = \{f : X \to \mathbb{R} : f \text{ es continua en } X\}$$

Definición 2.2.11. *Sea X un espacio métrico y sea $S \subset C(X)$ un subconjunto de funciones continuas de X en $\mathbb{R}$.*

1. *S será acotada en $x_0 \in X$ si $sup\{|f(x_0)| : f \in S\} < \infty$.*
2. *S será puntualmente acotado si $sup\{|f(x)| : f \in S\} < \infty$ para todo $x \in X$.*
3. *S será uniformemente acotado en $B \subset X$ si $sup\{|f(x)| : f \in S, x \in B\} < \infty$.*
4. *S será totalmemente acotado si para cada $r > 0$ existe un número finito de puntos $x_1, x_2, \ldots, x_n \in X$ tales que $S \subset B(x_1, r) \bigcup \ldots \bigcup B(x_n, r)$.*

Proposición 1. Un espacio topológico (X, τ) es compacto si es completo y totalmente acotado.

Definición 2.2.12. *Dado X un espacio métrico, decimos que un conjunto $A \subset X$ es precompacto en X (o relativamente compacto) cuando $\bar{A}$ es un conjunto compacto. Es decir, un conjunto es precompacto si su clausura es compacta.*

Definición 2.2.13. *Sea X un espacio métrico y consideremos el correspondiente espacio, $C(X)$, de las funciones escalares definidas y continuas en X. Dado $S \subset C(X)$, se tiene que S es equicontinuo en $x \in X$ si para cada $\epsilon > 0$ existe $\delta > 0$ tal que si $d(x, y) < \delta$ entonces $|f(x) - f(y)| < \epsilon$ para cada $f \in S$.*

Definición 2.2.14. *Sea X un espacio vectorial sobre el cuerpo $\mathbb{K}$. Una norma en X es una aplicación de X en $\mathbb{K}$ que denotamos por $\| \| : X \to \mathbb{K}$ tal que para cada $x, y \in X$ y cada $\alpha \in \mathbb{K}$ se verifican las siguientes propiedades:*

1. $\|x\| \geq 0$.

2. $\|x\| = 0$ *si y sólo si* $x = 0$.
3. $\|\alpha x\| = |\alpha| \times \|x\|$.
4. $\|x + y\| \leq \|x\| + \|y\|$.

Un espacio vectorial X dotado de una norma se le denomina ***espacio normado****.*

A continuación mostraremos algunos ejemplos de normas y espacios normados que nos serán útiles para el desarrollo de nuestro trabajo.

Ejemplo 1. Dado $X = \mathbb{K}^n$ y $\overrightarrow{x} = (x_1, x_2, \ldots, x_n) \in X$. Se define la norma-p como:

$$\| \overrightarrow{x} \|_p = (\sum_{i=1}^{n} |x_i|^p)^{1/p} = \sqrt[p]{|x_1|^p + |x_2|^p + \ldots + |x_n|^p}$$

Por ejemplo, si $p = 1$ se obtiene la norma $\| \overrightarrow{x} \|_1 = |x_1| + |x_2| + \ldots + |x_n|$, que define la llamada **distancia Manhattan**, mientras que si $p = 2$ se obtiene $\| \overrightarrow{x} \|_2 = \sqrt{|x_1|^2 + |x_2|^2 + \ldots + |x_n|^2}$ denominada **norma euclídea**.

Se puede definir la llamada **norma infinito** como:

$$\| \overrightarrow{x} \|_\infty = max(|x_1|, |x_2|, \ldots, |x_n|)$$

Ejemplo 2. Sea S un conjunto y sea $B(S) = \{f : S \to \mathbb{K} : f \text{ es acotada } \}$. Tenemos que $B(S)$ es un espacio vectorial con las operaciones usuales: $f + g$ y αf, $(\alpha \in \mathbb{K})$. Definimos $\|f\|_\infty = sup\{|f(t)| : t \in S\}$; es sencillo comprobar que $\| \ \|_\infty$ es una norma en $B(S)$ (llamada del supremo o de la **convergencia uniforme**). Observemos que si la sucesión $(f_n)_{n \in \mathbb{N}}$ converge a f_0 será $lim f_n(t) = f_0(t)$ para cada $t \in S$; es decir, $(f_n)_{n \in \mathbb{N}}$ converge a f_0 puntualmente en S pero además esta convergencia es uniforme en $t \in S$ (dado $\epsilon > 0$ existe $n_0 \in \mathbb{N}$ tal que $|f_n(t) - f_0(t)| < \epsilon$ para cada $t \in S$ si $n \geq n_0$).

Observemos que puede suceder que $(f_n)_{n \in \mathbb{N}}$ converja puntualmente a f_0 en S pero es falso que $lim||f_n - f_0|| = 0$. Por ejemplo, si $S = (-1, 1)$ y para cada $n \in \mathbb{N}$ se define $f_n(t) = t^n$ si $t \in (-1, 1)$ se verifica que (f_n) converge puntualmente a $f_0 = 0$ en S pero no lo hace uniformemente.

En el caso en que $S = \mathbb{N}$ tenemos que $B(\mathbb{N})$ se define exactamente el espacio vectorial de las sucesiones acotadas de escalares y la correspondiente norma es

$$\|(a_n)_{n\in\mathbb{N}}\| = sup\{\|a_n\| : \ n \in \mathbb{N}\}$$

Este espacio normado suele ser denotado por l_∞.

Ejemplo 3. Denotamos como:

$$l_1 = \{(x_n)_{n\in\mathbb{N}} : x_n \in \mathbb{K} \text{ si } n \in \mathbb{N} \text{ y } \sum_{n=1}^{\infty} |x_n| < \infty\}.$$

$$l_2 = \{(x_n)_{n\in\mathbb{N}} : x_n \in \mathbb{K} \text{ si } n \in \mathbb{N} \text{ y } (\sum_{n=1}^{\infty} |x_n|^2)^{1/2} < \infty\}.$$

$$l_\infty = \{(x_n)_{n\in\mathbb{N}} : x_n \in \mathbb{K} \text{ si } n \in \mathbb{N} \text{ y } sup|x_i| < \infty\}.$$

Se tiene que l_1, l_2, l_∞ son espacios vectoriales con las operaciones usuales $(x_n)_{n\in\mathbb{N}} + (y_n)_{n\in\mathbb{N}}$, $\alpha(x_n)_{n\in\mathbb{N}}$ con $\alpha \in \mathbb{K}$.

Se verifica que

$$||(x_n)_{n\in\mathbb{N}}|| = \sum_{n=1}^{\infty} |x_n| \text{ es una norma en } l_1.$$

$$||(x_n)_{n\in\mathbb{N}}|| = (\sum_{n=1}^{\infty} |x_n|^2)^{1/2} \text{ es una norma en } l_2.$$

$$||(x_n)_{n\in\mathbb{N}}|| = sup|x_i| \text{ es una norma en } l_\infty.$$

Ejemplo 4. Sea (X, d) un espacio métrico compacto donde el espacio $C(X)$ denota el espacio de funciones continuas $f : X \to \mathbb{R}$. El espacio $C(X)$ dotado de la norma $||f|| := max\{|f(x)| \ : x \in X\}$ induce la métrica:

$$\sigma(f, g) = ||f - g|| = max\{|f(x) - g(x)| \ : x \in X\}.$$

2.3. Teorema de Ascoli-Arzelà

Teorema 1. Teorema de Ascoli-Arzelà (1883)

Sea X un espacio métrico compacto. $S \subset C(X)$ es puntualmente acotado en X y equicontinuo en cada $x \in X$ si y solo si S es precompacto en $C(X)$.

Demostración. En primer lugar probaremos que si S es puntualmente acotado y equicontinuo entonces es precompacto.

Por hipótesis S es puntualmente acotado, por tanto para cada $x \in X$ se verifica que $\{|f(x)| : f \in S\}$ es un conjunto acotado.

Sea $\epsilon > 0$. Puesto que S es equicontinuo, para cada $x \in X$ existe $\delta_x > 0$ tal que si $y \in B(x, \delta_x)$ se cumple que $|f(x) - f(y)| < \epsilon$ para $f \in S$.

Además, por la compacidad de X existirá un conjunto $\{x_1, x_2, \ldots, x_n\} \subset X$ tal que $X = \bigcup_{i=1}^{n} B(x_i, \delta_i)$.

Por último, dado que S es puntualmente acotado por hipótesis, para cada $i \in \{1, \ldots, n\}$ existe $\alpha_i > 0$ tal que si $f \in S$ se cumple que $|f(x_i)| \leq \alpha_i$.

Sea $\alpha = max\{\alpha_1, \alpha_2, \ldots, \alpha_n\} + \epsilon$. Si $y \in X$ y $f \in S$ existe $j \in \{1, \ldots, n\}$ tal que $y \in B(x_j, \delta_j)$ (recordemos que $X = \bigcup_{i=1}^{n} B(x_i, \delta_i)$). Por tanto $|f(y) - f(x_j)| < \epsilon$ y por tanto $|f(y)| = |f(y) - f(x_j) + f(x_j)| \leq |f(y) - f(x_j)| + |f(x_j)| \leq \epsilon + \alpha_j \leq \epsilon + max\{\alpha_1, \alpha_2, \ldots, \alpha_n\} = \alpha$. Así pues S será uniformemente acotada en X (es decir, S es subconjunto acotado de $C(X)$).

Sea $M = \{\lambda \in \mathbb{R} : |\lambda| \leq \alpha\}$, tenemos que $M^n = M \times M \times \ldots \times M$ es un subconjunto compacto de $\mathbb{R}^n$. En $\mathbb{R}^n$ consideramos la norma $|| \, ||_1$.

Para cada $f \in S$ denotamos por $e(f) = (f(x_1), f(x_2), \ldots, f(x_n)) \in M^n$. Por la compacidad de M^n existe un número finito de bolas abiertas $V_1, V_2, \ldots, V_m$ en $\mathbb{R}^n$ con radio menor que ϵ de modo que $M^n \subset V_1 \bigcup V_2 \bigcup \ldots \bigcup V_m$.

Para cada $j \in \{1, 2, \ldots, m\}$ escogemos $f_j \in S$ tal que $e(f_j) \in V_j$ si es que

$$V_j \bigcap \{e(f) : f \in S\} \neq \emptyset.$$

A continuación probaremos que $S \subset \bigcup_{j=1}^{m} B(f_j, 4\epsilon)$.

Sea $f \in S$ entonces existe $j \in \{1, 2, \ldots, m\}$ tal que $e(f) \in V_j$ y se cumplirá $|f(x_i) - f_j(x_i)| < 2\epsilon$ para cada $i \in \{1, 2, \ldots, n\}$.

Si $x \in X$ tenemos que existe $i \in \{1, 2, \ldots, n\}$ tal que $e(f) \in V_j$ y se verificará $|f(x) - f(x_i)| < \epsilon$ y $|f_j(x) - f_j(x_i)| < \epsilon$.

Por tanto $|f(x) - f_j(x)| \leq |f(x) - f(x_i)| + |f(x_i) - f_j(x_i)| + |f_j(x_i) - f_j(x)| \leq \epsilon + 2\epsilon + \epsilon = 4\epsilon$.

De aquí deducimos que S es precompacto.

Procedamos ahora a probar la otra implicación. Supongamos que S es precompacto, lo que implica que $\bar{S}$ es compacto. Puesto que $\bar{S}$ es compacto, también es totalmente acotado. En particular, S es totalmente acotado y por tanto puntualmente acotado.

Veamos ahora que S es equicontinuo. Para ello fijamos $\epsilon > 0$; existen $f_1, \ldots, f_n \in C(X)$ tales que

$$S \subset B(f_1, \epsilon/3) \bigcup \ldots \bigcup B(f_n, \epsilon/3).$$

Cada f_i es uniformemente continua ya que (X, d) es compacto. Por tanto existe $\delta > 0$ tal que

$|f_i(x) - f_i(y)| < \epsilon/3$ para todos x, y tales que $d(x, y) < \delta$, para todo $1 \leq i \leq N$.

Dada cualquier función $f \in S$ y sabiendo que $S \subset B(f_1, \epsilon/3) \bigcup \ldots \bigcup B(f_n, \epsilon/3)$ existirá $1 \leq j \leq N$ tal que $\sigma(f, f_j) < \epsilon/3$ (donde $\sigma(f, g)$ se corresponde con la métrica del ejemplo 4).

Si $x, y \in X$ satisfacen $d(x, y) < \delta$ entonces

$$|f(x) - f(y)| \leq |f(x) - f_j(x)| + |f_j(x) - f_j(y)| + |f_j y - f(y)|$$

$$\leq \sigma(f, f_j) + |f_j(x) - f_j(y)| + \sigma(f, f_j) < 3\epsilon/3 = \epsilon.$$

Por tanto deducimos que S es equicontinuo. □

Corolario 1. Sea X un espacio métrico compacto y sea $S \subset C(X)$ equicontinuo en cada $x \in X$ y puntualmente acotado. Cada sucesión de S tiene alguna subsucesión que converge uniformemente en X a algún elemento de $C(X)$.

Demostración. Es una sencilla consecuencia del teorema de Ascoli-Arzelà, ya que al cumplirse las hipótesis del teorema se deduce que $\bar{S}$ es compacto en $C(X)$. □

A continuación, mostramos un ejemplo donde aplicamos el teorema de Ascoli-Arzelà para ver que cierto conjunto de funciones continuas es precompacto.

Ejemplo 5. Sea $\mathcal{F}$ el subconjunto de $C([0,1])$ dotado de la norma del supremo y consistente en las funciones f de la forma

$$f(x) = \sum_{n=1}^{\infty} a_n sin(n\pi x) \quad \text{con} \quad \sum_{n=1}^{\infty} n|a_n| \leq 1.$$

Puesto que la serie que define f converge uniformemente f pertenece a $C([0,1])$. Por otra parte, $\mathcal{F}$ es acotado en $C([0,1])$ ya que dada cualquier $f \in \mathcal{F}$ se tiene

$$\|f\|_\infty \leq \sum_{n=1}^{\infty} |a_n| \leq \sum_{n=1}^{\infty} n|a_n| \leq 1.$$

Por el teorema del valor medio, para cualquier $x < y \in \mathbb{R}$ existe un $x < \epsilon < y$ tal que

$$sin(x) - sin(y) = cos(\epsilon)(x - y).$$

Por tanto, para todo $x, y \in \mathbb{R}$ se tiene

$$|sin(x) - sin(y)| \leq |x - y|.$$

Dada cualquier $f \in \mathcal{F}$ se tiene que

$$|f(x) - f(y)| \leq \sum_{n=1}^{\infty} |a_n||sin(n\pi x) - sin(n\pi y)| \leq \sum_{n=1}^{\infty} \pi n|a_n||x - y| \leq \pi|x - y|.$$

Por tanto, dado $\epsilon > 0$ podemos escoger $\delta = \epsilon/\pi$, y entonces $|x - y| < \delta$ implica $|f(x) - f(y)| < \epsilon$ para todo $f \in \mathcal{F}$. Por el teorema de Ascoli-Arzelà, $\mathcal{F}$ es un subconjunto precompacto de $C([0, 1])$.

A continuación mostramos un ejemplo donde se pone de manifiesto que la compacidad del espacio métrico X es condición necesaria para la validez del teorema de Ascoli-Arzelà.

Ejemplo 6. Sea $C_0(\mathbb{R})$ el espacio de Banach de las funciones continuas $f : \mathbb{R} \to \mathbb{R}$ tales que $f(x) \to 0$ cuando $|x| \to \infty$, dotado de la norma del supremo.

Dado $n \in \mathbb{N}$ se define $f_n \in C_0(\mathbb{R})$ como

$$f_n(x) = \begin{cases} 1 & \text{if } |x| \leq n \\ n/|x| & \text{if } |x| > n \end{cases}$$

A continuación, mostraremos que el conjunto $\mathcal{F} = \{f_n : n \in \mathbb{N}\}$ es un subconjunto acotado y equicontinuo, pero (f_n) no tiene ninguna sucesión uniformemente convergente y, por tanto, el teorema de Ascoli-Arzelà no se cumple.

Demostración. Claramente se tiene que $\|f_n\|_\infty = 1$ para cada $n \in \mathbb{N}$ y por tanto, $\mathcal{F}$ es uniformemente acotado. Por otra parte, se cumple que:

1. Si $|x|, |y| \leq n$ entonces

$$|f_n(x) - f_n(y)| = 0.$$

2. Si $|x|, |y| \geq n$ entonces

$$|f_n(x) - f_n(y)| = \frac{n||x|-|y||}{|xy|} \leq \frac{|x-y|}{n}.$$

3. Si $|y| \leq n \leq |x|$ (y de forma similar si $|x| \leq n \leq |y|$), entonces

$$|f_n(x) - f_n(y)| = \frac{|x|-n}{|x|} \leq \frac{|x|-|y|}{|x|} \leq \frac{|x-y|}{n}.$$

Por tanto, se sigue que $|f_n(x) - f_n(y)| \leq |x - y|$ para todo $n \in \mathbb{N}$ y $x, y \in \mathbb{R}$, lo que implica que $\mathcal{F}$ es uniformemente equicontinua (y no sólamente puntualmente equicontinua).

Además, $f_n(x) \to 1$ cuando $n \to \infty$ para cada $x \in \mathbb{R}$. Por tanto, si la sucesión tuviera una subsucesión (f_{n_k}) que converge uniformemente, esta tendría que converger a 1. Sin embargo, $\|f_n - 1\|_\infty = 1$ para cada $n \in \mathbb{N}$, por tanto (f_n) no tiene una subsucesión uniformemente convergente, lo que conlleva que $\mathcal{F}$ no es precompacto. □

De esta forma, se prueba que el ejemplo anterior no contradice el teorema de Ascoli-Arzelà puesto que $\mathbb{R}$ no es compacto. Además, se prueba que la compacidad es condición necesaria para que se cumpla el teorema.

Capítulo 3

Conclusiones

Mi estancia en la universidad Miguel Hernández fue muy satisfactoria. Se me presentó un gran proyecto que se estaba desarrollando en ese mismo momento, con cierta base ya creada y mucho trabajo restante. Mi objetivo consistió en solucionar problemas y optimizar el código informático, además de elaborar nuevas funciones para el programa.

Una gran parte de la programación consiste en repasar, revisar y reestructurar el código ya creado para depurarlo y obtener el resultado deseado. Por supuesto suponía que en el mundo laboral esta parte también existiría de modo que procedí de igual manera que en los trabajos de programación de las asignaturas informáticas de la carrera. Tras cada modificación de los fragmentos del programa obtenía un resultado más cercano al esperado, lo que me animaba a continuar con mi trabajo sin encontrarlo excesivamente tedioso.

Por otra parte, la optimización de los tiempos de ejecución era una parte crítica del trabajo. Por ello necesité toda la teoría al respecto vista durante el grado, especialmente la de las asignaturas MT1022 - Algoritmia y MT1024 - Programación Concurrente y Paralela. Utilicé algunas de las estructuras de programación vistas en dichas asignaturas y tuve que aprender y emplear otras nuevas.

Me siento muy agradecido con el personal con el que tuve el placer de trabajar ya que existía un ambiente muy estimulante y cooperativo entre todos aquellos que trabajabamos en el proyecto. Periódicamente nos reuníamos para pensar y ofrecer distintas soluciones a los problemas planteados e incluso a la hora de rehacer ciertos programas primero realizabamos una lluvia de ideas con el fin de conocer cuál sería el mejor enfoque para el código.

A nivel personal considero que adquirí nuevas habilidades para el tratamiento de programas y repasé técnicas y conceptos ya vistos durante la carrera. Además tuve ocasión de conocer de primera mano el trabajo desde un punto de vista más computacional que puede desarrollar un matemático.

El objetivo del trabajo final de grado fue el estudio del teorema de Ascoli-Arzelà. Este teorema supone una importante herramienta en topología y análisis funcional para determinar si una familia de funciones continuas en un espacio métrico compacto es precompacta.

Este trabajo me ha permitido recordar conceptos vistos en la asignatura MT1027 - Topología y aprender otros nuevos. Por otra parte, también me ha servido para aprender a redactar y estructurar textos matemáticos.

Apéndice A

Programas empleados en las prácticas

En este apéndice se incluyen las versiones finales de los programas desarrollados durante la estancia en prácticas. Obviamente todos ellos se encuentran escritos en el lenguaje de programación python

A.1. flexPyQt.py

```
import sys, os, random, traceback, subprocess, platform
from PyQt5.QtWidgets import QApplication, QVBoxLayout,
    QHBoxLayout, QWidget, QDesktopWidget, QGraphicsScene,
    QGraphicsView, QGraphicsRectItem, QGraphicsPolygonItem, \
     QMainWindow, QMenuBar, QAction, QFileDialog, QTableWidget
         , QTableWidgetItem, QCheckBox, QGraphicsItem, QLabel,
         QColorDialog, QHeaderView, QPushButton, \
     QRadioButton, QButtonGroup, QComboBox, QLineEdit,
         QGridLayout, QTableView, QTabWidget, QInputDialog,
         QSlider
import PyQt5.QtSvg as QtSvg
import PyQt5.QtCore as QtCore
import PyQt5.QtGui as QtGui
import blastParser, gbParser
import xml.etree.ElementTree as ET
import cProfile
```

```
#I/O FUNCTIONS
def parseOldGenomeFile(filename, genomeScene):
    with open(filename) as inputFile:
        for line in inputFile:
            #Ignore comments
            if line[0] == '#':
                pass
            elif line[0:9] == 'sequences':
                seqLine = line.split(':')[1]
                seqLine2 = seqLine.split(';')
                for sequence in seqLine2:
                    seqInfo = sequence.split('=')
                    genomeScene.createChromosome(int(seqInfo
                        [1].rstrip('␣')), seqInfo[0].strip('␣'
                        ), 0, 0)
            elif len(line.split('\t')) > 8 and 'source' not
               in line.split('\t')[7]:


                cdsLine = line.split('\t')
                qualDict = {}
                qualDict['product'] = cdsLine[7]
                chr = genomeScene.findChromosomeByName(
                    cdsLine[0])
                # Length / position / strand / name / type /
                    qualifierDict
                chr.createGene((int(cdsLine[4])-int(cdsLine
                    [3])), int(cdsLine[3]), cdsLine[5],
                    cdsLine[1], cdsLine[2],
                                qualDict)

            elif 'source' in line.split('\t')[7]:
                cdsLine = line.split('\t')
                chr = genomeScene.findChromosomeByName(
                    cdsLine[0])
                chr.sequence = str(cdsLine[8])
                print(chr.sequence)

            else:
                pass
```

```
def parseFastaFile(filename, genomeScene):
    chromList = gbParser.parseFastaFiles(filename)
    for chrom in chromList:
        genomeScene.createChromosome(chrom.length, chrom.name
            , 0, 0)


def parseBlastFile(filename, genomeScene):
    newHits = blastParser.parseBlastFile(filename)
    families = blastParser.groupHits(newHits)
    for family in families:
        #family._equalize()
        newFamily = genomeScene.createBlastFamily(family.
            parents)
        for BlastHit in family:
            newFamily.createPoly(BlastHit)


def saveFlexFile(genomeScene, fileHandle):
    root = ET.Element('flexFile')
    root.set('nOfChrom', str(len(genomeScene.chrList)))
    root.set('nOfBlastFamilies', str(len(genomeScene.
        blastFamilies)))


    configElement = ET.Element('config')
    sceneElement = ET.Element('sceneRectDimensions')
    sceneElement.set('x', str(genomeScene.sceneRect().width()
        ))
    sceneElement.set('y', str(genomeScene.sceneRect().height
        ()))
    configElement.append(sceneElement)
    root.append(configElement)


    # Add Chromosomes from the genomeScene to the file
    for chrom in genomeScene.chrList:
        chromElement = ET.Element('Chromosome')
        chromElement.set('name', chrom.name)
        chromElement.set('length', str(chrom.w))
```

```
        chromElement.set('position', (str(chrom.pos().x()) +
            ',' + str(chrom.pos().y())))
        chromElement.set('sequence', chrom.sequence)
        #Add chromosome features from featureList
        for feature in chrom.geneList:
            featureElement = ET.Element('Feature')
            featureElement.set('name', feature.name)
            featureElement.set('length', str(feature.w))
            featureElement.set('position', str(feature.
                position))
            featureElement.set('strand', feature.strand)
            featureElement.set('type', feature.type)
            #Add qualifiers from the qualifier dict
            qualifierElement = ET.Element('Qualifiers')
            for key in feature.qualifiers.keys():
                qualifierElement.set(key, str(feature.
                    qualifiers[key]))
            featureElement.append(qualifierElement)
            chromElement.append(featureElement)
        root.append(chromElement)

    #Add Blast Families from the genomeScene to the file
    for blastFam in genomeScene.blastFamilies:
        blastFamElement = ET.Element('BlastFamily')
        blastFamElement.set('parent1', str(blastFam.parents
            [0]))
        blastFamElement.set('parent2', str(blastFam.parents
            [1]))
        #Add blastPolys from blastPolyList
        for blastPoly in blastFam.blastPolyList:
            blastPolyElement = ET.Element('BlastPoly')
            blastPolyElement.set('pos1start', str(int(
                blastPoly.pos1start)))
            blastPolyElement.set('pos1end', str(int(blastPoly
                .pos1end)))
            blastPolyElement.set('pos2start', str(int(
                blastPoly.pos2start)))
            blastPolyElement.set('pos2end', str(int(blastPoly
                .pos2end)))
            blastPolyElement.set('identity', str(blastPoly.
                identity))
```

```
            #We'll get the chrom attributes from the
                geneFamily - If something does not work, start
                 from here
            blastFamElement.append(blastPolyElement)
        root.append(blastFamElement)

    #Write the XML Tree to a file
    elementTree = ET.ElementTree(element=root)
    elementTree.write(fileHandle)


def loadFlexFile(filename, genomeScene):
    flexFile = ET.ElementTree(file=filename)
    # .find() finds Elements, .get() finds attributes
    root = flexFile.getroot()

    sceneDim = root.find('config').find('sceneRectDimensions'
        )
    #sceneX = float(sceneDim.get('x'))
    #sceneY = float(sceneDim.get('y'))
    #genomeScene.setSceneRect(0, 0, sceneX, sceneY)


    #Get all chromosomes from XML file
    for chrom in root.findall('Chromosome'):
        name = chrom.get('name')
        length = int(float(chrom.get('length')))
        position = chrom.get('position').split(',')
        sequence = chrom.get('sequence')
        newChrom = genomeScene.createChromosome(length, name,
            float(position[0]), float(position[1]), sequence)
        #Create genes according to feature tags
        for feature in chrom.findall('Feature'):
            name = feature.get('name')
            length = int(feature.get('length'))
            strand = feature.get('strand')
            position = int(feature.get('position'))
            type = feature.get('type')
            qualifier = feature.findall('Qualifiers')
            qualDict = {}
            try:
                for qualItem in qualifier:
```

```
                    for item in qualItem.attrib.items():
                        qualDict[item[0]] = item[1]

            except Exception:
                pass
            newChrom.createGene(length, position, strand,
                name, type, qualDict)

    #Get all blast families from XML file
    for blastFam in root.findall('BlastFamily'):
        parents = (blastFam.get('parent1'), blastFam.get('
            parent2'))
        newFamily = genomeScene.createBlastFamily(parents)
        chrom1 = genomeScene.findChromosomeByName(parents[0])
        chrom2 = genomeScene.findChromosomeByName(parents[1])
        #Create blastPolygons according to blastPoly tags
        for blastPoly in blastFam.findall('BlastPoly'):
            pos1start = float(blastPoly.get('pos1start'))
            pos1end = float(blastPoly.get('pos1end'))
            pos2start = float(blastPoly.get('pos2start'))
            pos2end = float(blastPoly.get('pos2end'))
            identity = float(blastPoly.get('identity'))
            newFamily.createPoly2(chrom1, chrom2, pos1start,
                pos1end, pos2start, pos2end, identity)


def getFastaFile(chromList):
    if os.path.exists('blastSeqs_flex.temp.fasta'):
        os.remove('blastSeqs_flex.temp.fasta')
    with open('blastSeqs_flex.temp.fasta', 'w') as blastFile:
        for chrom in chromList:
            blastFile.write('>' + chrom.name + '\n')
            blastFile.write(str(chrom.sequence) + '\n')


def runBlastOnSeqs(blastPath, blastSettings, genomeScene):
    #Create blast db
    subprocess.call([blastPath + 'makeblastdb', '-in', '
        blastSeqs_flex.temp.fasta', '-out', 'dbTemp', '-dbtype
        ', 'nucl'])
    #Run blast
    if platform.system() == 'Windows':
```

```
        if blastSettings['blastType'] == 'blastn':
            subprocess.call([blastPath + 'blastn.exe', '-
                query', 'blastSeqs_flex.temp.fasta', '-db', '
                dbTemp',
                             '-out' , 'blastSeqs_flex.blast',
                                 '-num_threads', '4', '-
                                 outfmt', '6'])
        else:
            subprocess.call([blastPath + 'tblastx.exe', '-
                matrix', str(blastSettings['blastMatrix']), '-
                query',
                             'blastSeqs_flex.temp.fasta', '-
                                 db', 'dbTemp', '-out', '
                                 blastSeqs_flex.blast',
                             '-num_threads', '4', '-outfmt',
                                 '6'])

    else:
        if blastSettings['blastType'] == 'blastn':
            subprocess.call([blastPath + 'blastn', '-query',
                'blastSeqs_flex.temp.fasta', '-db', 'dbTemp',
                '-out',
                             'blastSeqs_flex.blast', '-
                                 num_threads', '4', '-outfmt',
                                  '6' ])
        else:
            subprocess.call([blastPath + 'tblastx', '-matrix'
                , str(blastSettings['blastMatrix']), '-query',
                             'blastSeqs_flex.temp.fasta', '-
                                 db', 'dbTemp', '-out', '
                                 blastSeqs_flex.blast',
                             '-num_threads', '4', '-outfmt',
                                 '6'])

    filterBlastParameters = {'minAln':[0, 'auto'], 'minIdent'
        :90, 'removeAdj':[0,None]}
    try:
        if blastSettings['minAln'] == 'auto':
            pass
        else:
            filterBlastParameters['minAln'][0] = int(
                blastSettings['minAln'])
```

```
            filterBlastParameters['minAln'][1] = None
    except Exception:
        pass
    try:
        filterBlastParameters['minIdent'] = float(
            blastSettings['minIdent'])
    except Exception:
        pass
    try:
        filterBlastParameters['removeAdj'][0] = bool(
            blastSettings['mergeAdj'][0])
        filterBlastParameters['removeAdj'][1] = int(
            blastSettings['mergeAdj'][1])
        print(filterBlastParameters['removeAdj'])
    except Exception:
        pass

    newBlastHits = blastParser.parseBlastFile('blastSeqs_flex
        .blast', minIdentity=filterBlastParameters['minIdent'
        ],
                                                  minAln=
                                                      filterBlastParameters
                                                      ['minAln'
                                                      ][0])
    print('group hits')
    families = blastParser.groupHits(newBlastHits)
    print('group hits finished')
    for family in families:
        print('equalize')
        family._equalize()

        if filterBlastParameters['minAln'][1] == 'auto':
            family.removeSmallHits()
        family.removeOwnHits()
        family.removeInternalHits()
        if filterBlastParameters['removeAdj'][0] == True:
            family.mergeBlastList(filterBlastParameters['
                removeAdj'][1], 1.50)
        else:
            print('no merge!')
```

```
            newFamily = genomeScene.createBlastFamily(family.
                parents)
            for BlastHit in family:
                newFamily.createPoly(BlastHit)

        if blastSettings['saveFile'] == True:
            try:
                os.remove('blastSequences_flex2_original.blast')
            except Exception:
                pass
            os.rename('blastSeqs_flex.blast', '
                blastSequences_flex2_original.blast')
        else:
            os.remove('blastSeqs_flex.blast')
        #os.remove('blastSeqs_flex.temp.fasta')


def parseStyleFile(filename):
    paintOrderList = []
    with open(filename) as paintFile:
        for line in paintFile:
            newPaintOrder = {}
            orderLine = line.split('\t')
            newPaintOrder['type'] = orderLine[0].rstrip()
            if ':' in orderLine[1]:
                newPaintOrder['class'] = orderLine[1].split('
                    :')[0].rstrip()
                newPaintOrder['class2'] = orderLine[1].split(
                    ':')[1].rstrip()
            else:
                newPaintOrder['class'] = None
                newPaintOrder['class2'] = None

            newPaintOrder['delimiter'] = orderLine[2].rstrip
                ()
            if orderLine[3].rstrip() != 'None':
                newPaintOrder['color'] = orderLine[3].split('
                    /')
                for number in newPaintOrder['color']:
                    number = int(number)
            else:
                newPaintOrder['color'] = [None, None, None]
```

```
            paintOrderList.append(newPaintOrder)
        return paintOrderList



#CLASS OBJECTS
class GenomeScene(QGraphicsScene):
    def __init__(self, settings):
        super().__init__()
        self.setMinimumRenderSize(1.0)
        #Removing the index improves performance
            significantly
        self.setItemIndexMethod(QGraphicsScene.NoIndex)
        self.chrList = []
        self.blastFamilies = []
        self.fosmidSize = int(settings['fosmidSize'])
        self.displayType = settings['displayType']
        #self.setSceneRect(0 , 0, 25000, 25000)

    def createChromosome(self, w, name, x, y, sequence = None
        ):
        goodName = self.checkChromosomeNames(name)
        h = self.fosmidSize
        chr = Chromosome(0, 0, w, h, goodName, sequence)
        chr.setPos(x, y)
        self.chrList.append(chr)
        self.addItem(chr)
        self.views()[0].fitNewObject()
        return chr

    def createBlastFamily(self, parents):
        newFamily = BlastFamily(parents, self)
        self.findChromosomeByName(parents[0]).blastList.
            append(newFamily)
        self.findChromosomeByName(parents[1]).blastList.
            append(newFamily)
        self.blastFamilies.append(newFamily)
        return newFamily

    def findChromosomeByName(self, name):
        target = name
        for chr in self.chrList:
```

```
            if chr.name == target:
                return chr

        return None

    def applyStyle(self, filename):
        paintOrderList = parseStyleFile(filename)
        for chr in self.chrList:
            for cds in chr.geneList:
                cds.applyStyle(paintOrderList)
        self.update()

    def sortChromosomesByHeight(self):
        self.chrList.sort(key=lambda Chromosome: Chromosome.
            pos().y())
        for chr in self.chrList:
            print(chr.name)

    def checkChromosomeNames(self, name):
        print('checking name')
        nameList = []
        for chr in self.chrList:
            nameList.append(chr.name)
        if name in nameList:
            i = 1
            while i > 0:
                newName = '{0}_{1}'.format(name, i)
                if newName not in nameList:
                    return newName
                else:
                    i += 1
        else:
            return name

    def deleteChromosome(self, name):
        deleteBlastList = []
        for blastFamily in self.blastFamilies:
            if name in blastFamily.parents:
                deleteBlastList.append(blastFamily)

        for blastFamily in deleteBlastList:
            blastFamily.deleteFamily()
```

```
        self.findChromosomeByName(name).deleteChromosome()

    def hideChromosome(self, name, bool):
        self.findChromosomeByName(name).hideChromosome(bool)
        for blastFamily in self.blastFamilies:

            if name in self.blastFamily.parents and self.
               findChromosomeByName(blastFamily.parents[0]).
               isVisible() and self.findChromosomeByName(
               blastFamily.parents[1]).isVisible():
                blastFamily.setBlastVisibility(bool)
            else:
                pass

    def fosmidSizeChanged(self, settings):
        self.fosmidSize = int(settings['fosmidSize'])
        for chr in self.chrList:
            chr.fosmidSizeChanged(self.fosmidSize)


class GenomeViewer(QGraphicsView):
    def __init__(self, scene):
        super().__init__(scene)
        self.fitInView(scene.sceneRect(), QtCore.Qt.
           KeepAspectRatio)
        self.panning = False
        self.panPos = None
        self.zoomLvl = 0
        self.changeShapeOnZoom = True
        self.displayType = scene.displayType

        #OpenGL support is a can of worms I'd prefer not to
           open
        #self.setViewport(GLWidget(parent = self, flags=self.
           windowFlags()))

    def wheelEvent(self, QWheelEvent):
        #I copypasted this from stackOverflow, but I should
           recode it so it uses scaleFactor
        self.setTransformationAnchor(QGraphicsView.NoAnchor)
        self.setResizeAnchor(QGraphicsView.NoAnchor)
```

```
        zoomInFactor = 1.25
        zoomOutFactor = 1/zoomInFactor
        oldPos = self.mapToScene(QWheelEvent.pos())

        if QWheelEvent.angleDelta().y() > 0:
            endFactor = zoomInFactor
            self.zoomLvl += 1
        else:
            endFactor = zoomOutFactor
            self.zoomLvl -= 1
        self.scale(endFactor, endFactor)

        newPos = self.mapToScene(QWheelEvent.pos())
        delta = newPos - oldPos
        self.translate(delta.x(), delta.y())

        #Adjust Cds to zoom. first, figure how many screen
            pixels is a qgraphicsscene unit
        viewPortRect = QtCore.QRect(0, 0, self.scene().views
            ()[0].viewport().width(),
                                    self.scene().views()[0].
                                        viewport().height())
        visibleSceneRectWidth = int(self.scene().views()[0].
            mapToScene(viewPortRect).boundingRect().width())
        viewportWidth = int(self.scene().views()[0].viewport
            ().width())
        target = viewportWidth / visibleSceneRectWidth

        if self.changeShapeOnZoom == True:
            for chrom in self.scene().chrList:
                for cds in chrom.geneList:
                    if cds.type == 'CDS' and self.displayType
                        == 'adaptative':
                        cds.checkShape(target)
        #self.scene().views()[0].viewport().update(
            viewPortRect)

    def mousePressEvent(self, QMouseEvent):
        if QMouseEvent.button() == QtCore.Qt.MiddleButton:
            self.panning = True
            self.panPos = (QMouseEvent.screenPos().x(),
                QMouseEvent.screenPos().y())
```

```
        else:
            QGraphicsView.mousePressEvent(self, QMouseEvent)

    def mouseMoveEvent(self, QMouseEvent):
        if self.panning == True:

            viewPortRect = QtCore.QRect(0, 0, self.scene().
                views()[0].viewport().width(),
                                        self.scene().views()
                                            [0].viewport().
                                            height())
            visibleSceneRectWidth = int(self.scene().views()
                [0].mapToScene(viewPortRect).boundingRect().
                width())
            viewportWidth = int(self.scene().views()[0].
                viewport().width())
            target = viewportWidth / visibleSceneRectWidth

            xdiff = (QMouseEvent.screenPos().x() - self.
                panPos[0]) * 1/target
            ydiff = (QMouseEvent.screenPos().y() - self.
                panPos[1]) * 1/target
            self.translate(xdiff, ydiff)
            self.panPos = (QMouseEvent.screenPos().x(),
                QMouseEvent.screenPos().y())

        else:
            QGraphicsView.mouseMoveEvent(self, QMouseEvent)

    def mouseReleaseEvent(self, QMouseEvent):
        if QMouseEvent.button() == QtCore.Qt.MiddleButton:
            self.panning = False

        else:
            QGraphicsView.mouseReleaseEvent(self, QMouseEvent
                )

    def fitNewObject(self):
        self.ensureVisible(self.scene().sceneRect())
```

```
class Chromosome(QGraphicsRectItem):
    def __init__(self, x, y, w, h, name, sequence=None):
        self.h = h
        self.w = w
        super().__init__(x, y, self.w, self.h)
        self.setPos(QtCore.QPoint(x, y))
        self.ItemIsMovable = True
        self.ItemIsSelectable = True
        self.dragged = False
        self.geneList = []
        self.blastList = []
        self.name = name
        self.sequence = sequence
        self.setBrush(QtGui.QBrush(QtCore.Qt.darkGray))
        self.setZValue(2.0)

    def mousePressEvent(self, QGraphicsSceneMouseEvent):
        self.dragged = True

    def mouseReleaseEvent(self, QGraphicsSceneMouseEvent):
        self.dragged = False

    def mouseMoveEvent(self, QGraphicsSceneMouseEvent):
        if self.dragged == True:
            xdiff = QGraphicsSceneMouseEvent.scenePos().x() -
                QGraphicsSceneMouseEvent.lastScenePos().x()
            ydiff = QGraphicsSceneMouseEvent.scenePos().y() -
                QGraphicsSceneMouseEvent.lastScenePos().y()

            chromosomeX = self.pos().x() + xdiff
            chromosomeY = self.pos().y() + ydiff
            self.setPos(QtCore.QPoint(chromosomeX,
               chromosomeY))
            for blastFamily in self.blastList:
                blastFamily.updatePolyPos()

            for cds in self.geneList:
                cds.moveCDS(xdiff, ydiff)

    def createGene(self, w, pos, strand, name, type,
       qualifiers):
```

```
        cds = CDS(self, w, pos, strand, name, type,
            qualifiers)
        self.geneList.append(cds)
        self.scene().addItem(cds)
        self.scene().views()[0].fitInView(self.scene().
            sceneRect(), QtCore.Qt.KeepAspectRatio)
        cds.checkShape()
        return cds

    def deleteChromosome(self):
        for cds in self.geneList:
            self.scene().removeItem(cds)
            del cds
        self.scene().chrList.remove(self)
        self.scene().removeItem(self)
        del self

    def hideChromosome(self, bool):
        for cds in self.geneList:
            cds.setVisible(bool)
        self.setVisible(bool)

    def fosmidSizeChanged(self, h):
        self.h = h
        self.setRect(self.pos().x(), self.pos().y(), self.w,
            self.h)
        self.boundingRect = QtGui.QPolygonF(self.pos().x(),
            self.pos().y(), self.w, self.h)
        for cds in self.geneList:
            cds.fosmidSizeChanged(self.h)


class CDS(QGraphicsPolygonItem):
    def __init__(self, chromosome, w, pos, strand, name, type
        , qualifiers):
        self.h = chromosome.h * 2
        self.w = w
        self.position = pos
        self.parent = chromosome
        self.qualifiers = qualifiers
        self.strand = strand
        self.displayType = self.parent.scene().displayType
```

```
if type == 'CDS' or type == 'gene':
    self.type = 'CDS'
else:
    self.type = type
self.style = None
x = chromosome.pos().x() + int(pos)
y = chromosome.pos().y() - ((self.h - self.parent.h)
   /4)
if self.type == 'repeat_region':
    shapes = self.calculateShapes(self.parent, pos,
       type = 'repeat')
elif self.type == 'CDS':
    shapes = self.calculateShapes(self.parent, pos,
       type='cds')
else:
    shapes = self.calculateShapes(self.parent, pos,
       type='misc')

self.rectPolygon = shapes[0]
self.trianPolygon = shapes[1]
self.arrowPolygon = shapes[2]

super().__init__(self.rectPolygon)
self.setPos(QtCore.QPoint(x, y))
self.name = name
self.setAcceptHoverEvents(True)
if self.type == 'repeat_region':
    self.style = QtGui.QBrush(QtCore.Qt.cyan)
    self.setBrush(self.style)
elif self.type == 'CDS':
    self.style = QtGui.QBrush(QtCore.Qt.darkGreen)
    self.setBrush(self.style)
    pen = QtGui.QPen()
    pen.setWidth(50)
    pen.setCosmetic(False)
    self.setPen(pen)
else:
    self.style = QtGui.QBrush(QtCore.Qt.darkMagenta)
    self.setBrush(self.style)
```

```
        self.setFlag(QGraphicsItem.ItemSendsGeometryChanges,
            True)
        self.setZValue(3.0)


    def moveCDS(self, xdiff, ydiff):
        self.setPos(QtCore.QPoint(self.pos().x() + xdiff,
            self.pos().y() + ydiff))

    def mousePressEvent(self, QGraphicsSceneMouseEvent):
        self.parent.dragged = True

    def mouseReleaseEvent(self, QGraphicsSceneMouseEvent):
        self.parent.dragged = False

    def mouseMoveEvent(self, QGraphicsSceneMouseEvent):
        self.parent.mouseMoveEvent(QGraphicsSceneMouseEvent)

    def hoverEnterEvent(self, QGraphicsSceneHoverEvent):
        self.setBrush(QtGui.QBrush(QtCore.Qt.darkYellow))
        tooltipText = '{}␣on␣{}\nType:␣{}\nLength:␣{},␣on␣{}
            ␣strand'.format(self.name, self.parent.name, self.
            type, self.w, self.strand)
        try:
            tooltipText += '\nProduct:␣{}'.format(self.
                qualifiers['product'])
        except KeyError:
            pass

        self.setToolTip(tooltipText)

    def hoverLeaveEvent(self, QGraphicsSceneHoverEvent):
        self.setBrush(QtGui.QBrush(self.style))

    def mouseDoubleClickEvent(self, QGraphicsSceneMouseEvent)
        :
        self.window = CDSInfoWidget(cds = self)
        self.window.show()

    def checkShape(self, target=None):
```

```
        if self.displayType == 'arrows' and self.arrowPolygon
            is not None:
            self.setPolygon(self.arrowPolygon)
            self.prepareGeometryChange()
            self.update()
        elif self.displayType == 'rectangles':
            self.setPolygon(self.rectPolygon)
            self.prepareGeometryChange()
            self.update()

        else:
            if target is None:
                viewPortRect = QtCore.QRect(0, 0, self.scene
                    ().views()[0].viewport().width(),
                                            self.scene().
                                                views()[0].
                                                viewport().
                                                height())
                visibleSceneRectWidth = int(self.scene().
                    views()[0].mapToScene(viewPortRect).
                    boundingRect().width())
                viewportWidth = int(self.scene().views()[0].
                    viewport().width())
                target = viewportWidth /
                    visibleSceneRectWidth

            if (self.w * target) > 10 and self.polygon() !=
                self.arrowPolygon:
                self.setPolygon(self.arrowPolygon)
                self.prepareGeometryChange()
                self.update()

            elif (self.w * target) < 10 and self.polygon() !=
                self.rectPolygon:
                self.setPolygon(self.rectPolygon)
                self.prepareGeometryChange()
                self.update()

            else:
                pass

    def calculateShapes(self, chromosome, pos, type):
```

```
if type == 'repeat':
    # Get Rectangle Shape
    point1 = QtCore.QPoint(self.w, self.h * -1)
    point2 = QtCore.QPoint(self.w, (self.h / -4))
    point3 = QtCore.QPoint(0, (self.h / -4))
    point4 = QtCore.QPoint(0, self.h * -1)
    rectPolygon = QtGui.QPolygonF((point1, point2,
       point3, point4))

    return [rectPolygon, None, None]

elif type == 'misc':
    # Get Rectangle Shape
    point1 = QtCore.QPoint(self.w, self.h)
    point2 = QtCore.QPoint(self.w, self.h * 2)
    point3 = QtCore.QPoint(0, self.h * 2)
    point4 = QtCore.QPoint(0, self.h)
    rectPolygon = QtGui.QPolygonF((point1, point2,
       point3, point4))

    return [rectPolygon, None, None]


else:
    # Get Rectangle Shape
    point1 = QtCore.QPoint(self.w , self.h / -4)
    point2 = QtCore.QPoint(self.w, self.h)
    point3 = QtCore.QPoint(0, self.h)
    point4 = QtCore.QPoint(0, self.h/-4)
    rectPolygon = QtGui.QPolygonF((point1, point2,
       point3, point4))

    # Get triangle shape
    if self.strand == '+':
        point1 = QtCore.QPoint(self.w, (self.h / 2.5)
           )
        point2 = QtCore.QPoint(0, (self.h / -4))
        point3 = QtCore.QPoint(0, self.h)
        trianPolygon = QtGui.QPolygonF((point1,
           point2, point3))
    else:
```

```
            point1 = QtCore.QPoint(0, (self.h / 2.5))
            point2 = QtCore.QPoint(self.w, (self.h / -4))
            point3 = QtCore.QPoint(self.w, self.h)
            trianPolygon = QtGui.QPolygonF((point1,
                point2, point3))

        # Get Arrow Shape
        if self.strand == '+':
            point1 = QtCore.QPoint(self.w, (self.h / 2.5)
                )
            point2 = QtCore.QPoint((self.w * 0.66), self.
                h)
            point3 = QtCore.QPoint((self.w * 0.66), (self
                .parent.h * 1.5))
            point4 = QtCore.QPoint(0, (self.parent.h *
                1.5))
            point5 = QtCore.QPoint(0,0)
            point6 = QtCore.QPoint((self.w * 0.66), 0)
            point7 = QtCore.QPoint((self.w * 0.66), (self
                .h / -4))
            arrowPolygon = QtGui.QPolygonF((point1,
                point2, point3, point4, point5, point6,
                point7))
        else:
            point1 = QtCore.QPoint(0, (self.h / 2.5))
            point2 = QtCore.QPoint((self.w * 0.33), self.
                h)
            point3 = QtCore.QPoint((self.w * 0.33), (self
                .parent.h * 1.5))
            point4 = QtCore.QPoint(self.w, (self.parent.h
                 * 1.5))
            point5 = QtCore.QPoint(self.w, 0)
            point6 = QtCore.QPoint((self.w * 0.33), 0)
            point7 = QtCore.QPoint((self.w * 0.33), (self
                .h / -4))
            arrowPolygon = QtGui.QPolygonF((point1,
                point2, point3, point4, point5, point6,
                point7))

        return [rectPolygon, trianPolygon, arrowPolygon]
```

```
    def modifyBrush(self, hue = None, saturation = None,
        value = None):
        oldColor = self.style.color()
        newColor = QtGui.QColor(0, 0, 0)
        try:
            newColor = newColor.fromHsv(int(hue), int(
                saturation) * 2.55, int(value) * 2.55, 255)

        except Exception as e:
            self.style.setColor(oldColor)
            pass

        if newColor.getHsv()[0] < 0:
            self.style.setColor(oldColor)
        else:
            self.style.setColor(newColor)
        self.setBrush(self.style)

    def applyStyle(self, paintOrderList):
        newColor = [None, None, None]

        for order in paintOrderList:
            if self.type != order['type']:
                continue
            #Classify by length
            if order['class'] == None:
                newColor = [order['color'][0], order['color'
                    ][1], order['color'][2]]
                self.modifyBrush(newColor[0], newColor[1],
                    newColor[2])
                continue
            elif order['class'] == 'length':
                if order['class2'] == '>':
                    if self.w > int(order['delimiter']):
                        newColor = [order['color'][0], order[
                            'color'][1], order['color'][2]]
                    else:
                        continue
                elif order['class2'] == '<':
                    if self.w < int(order['delimiter']):
                        newColor = [order['color'][0], order[
                            'color'][1], order['color'][2]]
```

```
                    else:
                        continue

            elif order['class'] == 'qualifier':
                try:
                    cdsValue = self.qualifiers[order['class2'
                        ]]

                    if str(order['delimiter'].rstrip()) in
                        str(cdsValue):
                        newColor = [order['color'][0], order[
                            'color'][1], order['color'][2]]
                    else:
                        continue
                except Exception:
                    continue
                self.modifyBrush(newColor[0], newColor[1],
                    newColor[2])

    def fosmidSizeChanged(self, h):
        self.h = h * 2
        if self.type == 'repeat_region':
            shapes = self.calculateShapes(self.parent, self.
                pos, type = 'repeat')
        elif self.type == 'CDS':
            shapes = self.calculateShapes(self.parent, self.
                pos, type='cds')
        else:
            shapes = self.calculateShapes(self.parent, self.
                pos, type='misc')

        self.rectPolygon = shapes[0]
        self.trianPolygon = shapes[1]
        self.arrowPolygon = shapes[2]

        self.checkShape()



class BlastFamily:
    def __init__(self, parents, genomeScene):
        self.parents = parents
```

```
        self.genomeScene = genomeScene
        self.blastPolyList = []

    def createPoly(self, BlastHit):
        blastPoly = BlastPolygon(self.genomeScene.
            findChromosomeByName(BlastHit.parents[0]), self.
            genomeScene.findChromosomeByName(BlastHit.parents
            [1]),
                                 BlastHit.seq1pos[0],
                                    BlastHit.seq1pos[1],
                                    BlastHit.seq2pos[0],
                                    BlastHit.seq2pos[1],
                                    BlastHit.identity,
                                    BlastHit.mismatches)
        self.blastPolyList.append(blastPoly)
        self.genomeScene.addItem(blastPoly)

    def createPoly2(self, chrom1, chrom2, pos1start, pos1end,
        pos2start, pos2end, identity, mismatches):
        blastPoly = BlastPolygon(chrom1, chrom2, pos1start,
            pos1end, pos2start, pos2end, identity, mismatches)
        self.blastPolyList.append(blastPoly)
        self.genomeScene.addItem(blastPoly)

    def setBlastVisibility(self, bool):
        for blastPoly in self.blastPolyList:
            blastPoly.setVisible(bool)

    def updatePolyPos(self):
        for blastPoly in self.blastPolyList:
            blastPoly.calculatePolygon()

    def deleteFamily(self):
        for blastPoly in self.blastPolyList:
            self.genomeScene.removeItem(blastPoly)
            del blastPoly
        self.genomeScene.findChromosomeByName(self.parents
            [0]).blastList.remove(self)
        self.genomeScene.findChromosomeByName(self.parents
            [1]).blastList.remove(self)
        self.genomeScene.blastFamilies.remove(self)
        del self
```

```
    def changeBlastColor(self, color):
        for blast in self.blastPolyList:
            blast.brush = QtGui.QBrush(color)
            blast.setBrush(QtGui.QBrush(color))
            blast.changeSaturation()


class BlastPolygon(QGraphicsPolygonItem):
    def __init__(self, chrom1, chrom2, pos1start, pos1end,
       pos2start, pos2end, identity, mismatches):
        self.pos1start = pos1start
        self.pos1end = pos1end
        self.pos2start = pos2start
        self.pos2end = pos2end
        self.chrom1 = chrom1
        self.chrom2 = chrom2
        self.identity = identity
        self.mismatches = mismatches
        self.brush = QtGui.QBrush(QtCore.Qt.darkRed)

        point1 = QtCore.QPoint(self.chrom1.pos().x() + self.
           pos1end, self.chrom1.pos().y())
        point2 = QtCore.QPoint(self.chrom2.pos().x() + self.
           pos2end, self.chrom2.pos().y())
        point3 = QtCore.QPoint(self.chrom2.pos().x() + self.
           pos2start, self.chrom2.pos().y())
        point4 = QtCore.QPoint(self.chrom1.pos().x() + self.
           pos1start, self.chrom1.pos().y())
        polygon = QtGui.QPolygonF((point1, point2, point3,
           point4))

        super().__init__(polygon)
        self.setBrush(self.brush)
        self.changeSaturation()
        self.setAcceptHoverEvents(True)
        self.setZValue(1.0)
        self.tooltip = self.createTooltip()

        #Fixed the pen issue!
        pen = QtGui.QPen()
        pen.setWidth(0.5)
```

```
        pen.setCosmetic(True)
        self.setPen(pen)

    def calculatePolygon(self):
        point1 = QtCore.QPoint(self.chrom1.pos().x() + self.
            pos1end, self.chrom1.pos().y())
        point2 = QtCore.QPoint(self.chrom2.pos().x() + self.
            pos2end, self.chrom2.pos().y())
        point3 = QtCore.QPoint(self.chrom2.pos().x() + self.
            pos2start, self.chrom2.pos().y())
        point4 = QtCore.QPoint(self.chrom1.pos().x() + self.
            pos1start, self.chrom1.pos().y())
        polygon = QtGui.QPolygonF((point1, point2, point3,
            point4))
        self.setPolygon(polygon)

    def hoverEnterEvent(self, QGraphicsSceneHoverEvent):
        self.setBrush(QtGui.QBrush(QtCore.Qt.darkYellow))
        self.setToolTip(self.tooltip)

    def hoverLeaveEvent(self, QGraphicsSceneHoverEvent):
        self.setBrush(self.brush)

    def createTooltip(self):
        tooltip = (self.chrom1.name+ '\n' + str(int(self.
            pos1start)) + '␣-␣' + str(int(self.pos1end)) + '\n
            \n' +
             self.chrom2.name+'\n' + str(int(self.pos2start))
                + '␣-␣' + str(int(self.pos2end)) + '\n\n' +
              'Identity␣=␣' + str(self.identity) + '\n\n' + '
                 Mismatches␣=␣' + str(self.mismatches))

        return tooltip

    def changeSaturation(self):
        oldColor = self.brush.color().toHsv()
        identityValue = float(self.identity) / 100
        newSat = oldColor.saturation()
        newVal = oldColor.value()
        if newSat > 0 and identityValue < 0.95:
            newSat = int(oldColor.saturation() *
                identityValue * 0.9)
```

```
        elif newSat > 0 and identityValue > 0.95:
            newSat = int(oldColor.saturation() *
                identityValue)
        #if newVal < 255:
            #newVal = int(oldColor.value() * (1/identityValue
                ))
        newColor = QtGui.QColor().fromHsv(oldColor.hue(),
            newSat, newVal, 255)
        newBrush = QtGui.QBrush(newColor)
        self.brush = newBrush
        self.setBrush(newBrush)


class BlastFamilyWidget(QWidget):
    def __init__(self, blastList, chromList):
        super().__init__()
        self.blastList = blastList
        self.chromList = chromList
        #self.setWindowModality(QtCore.Qt.ApplicationModal)

        self.initUI()

    def initUI(self):

        self.setGeometry(0, 0, 550, 400)
        self.setWindowTitle('Canvas Editor')

        # Set blastFamilyTable
        self.blastTable = QTableWidget()
        self.generateBlastTable(self.blastTable)

        #Set Chromtable
        self.chromTable = QTableWidget()
        self.generateChromTable(self.chromTable)

        self.tabs = QTabWidget()
        self.tab1 = QWidget()
        self.tab2 = QWidget()
        self.tabs.addTab(self.tab1, "Sequences")
        self.tabs.addTab(self.tab2, "Blast Families")

        mainLayout = QVBoxLayout()
```

```
        layVBoxTab1 = QVBoxLayout()
        layVBoxTab2 = QVBoxLayout()
        layVBoxTab1.addWidget(self.chromTable)
        layVBoxTab2.addWidget(self.blastTable)
        self.tab1.setLayout(layVBoxTab1)
        self.tab2.setLayout(layVBoxTab2)
        mainLayout.addWidget(self.tabs)
        self.setLayout(mainLayout)
        self.center()
        self.show()

    def center(self):
        qr = self.frameGeometry()
        cp = QDesktopWidget().availableGeometry().center()
        qr.moveCenter(cp)
        self.move(qr.topLeft())

    def generateBlastTable(self, qtable):
        qtable.setRowCount(len(self.blastList))
        qtable.setColumnCount(5)
        qtable.setHorizontalHeaderLabels(['Sequence 1', '
            Sequence 2', 'N of Blasts', 'Visible?',''])
        qtable.horizontalHeader().setSectionResizeMode(
            QHeaderView.ResizeToContents)
        qtable.setSelectionBehavior(QTableView.SelectRows)

        for i in range(0, len(self.blastList)):

            parent1Cell = QTableWidgetItem(self.blastList[i].
                parents[0])
            parent2Cell = QTableWidgetItem(self.blastList[i].
                parents[1])
            nOfBlastsCell = QTableWidgetItem(str(len(self.
                blastList[i].blastPolyList)))
            visibleCell = QCheckBox(self.blastTable)
            visibleCell.clicked.connect(self.hideBlast)
            visibleCell.setTristate(False)
            qtable.setCellWidget(i, 3, visibleCell)
            deleteBlastCell = QPushButton('Delete', self.
                blastTable)
            deleteBlastCell.clicked.connect(self.deleteBlast)
            qtable.setCellWidget(i, 4, deleteBlastCell)
```

```
            # The try/Except block is here so the program
               doesn't crash if it sees a blastFamily with 0
               blasts
            try:
                if self.blastList[i].blastPolyList[0].
                   isVisible() == True:
                    visibleCell.setCheckState(QtCore.Qt.
                       Checked)
                else:
                    visibleCell.setCheckState(QtCore.Qt.
                       Unchecked)
            except IndexError:
                visibleCell.setCheckState(QtCore.Qt.Unchecked
                   )

            qtable.setItem(i, 0, parent1Cell)
            qtable.setItem(i, 1, parent2Cell)
            qtable.setItem(i, 2, nOfBlastsCell)

    def generateChromTable(self, qtable):
        qtable.setRowCount(len(self.chromList))
        qtable.setColumnCount(5)
        qtable.setHorizontalHeaderLabels(['Sequence␣Name', '
           Sequence␣Length', 'Visible?', '', ''])
        qtable.horizontalHeader().setSectionResizeMode(
           QHeaderView.ResizeToContents)
        qtable.setSelectionBehavior(QTableView.SelectRows)

        for i in range(0, len(self.chromList)):
            nameCell = QTableWidgetItem(self.chromList[i].
               name)
            lengthCell = QTableWidgetItem(str(len(self.
               chromList[i].sequence)))
            changeNameCell = QPushButton('Change␣Name', self.
               chromTable)
            changeNameCell.clicked.connect(self.changeName)
            qtable.setCellWidget(i, 3, changeNameCell)
            deleteSeqCell = QPushButton('Delete', self.
               chromTable)
            deleteSeqCell.clicked.connect(self.deleteSequence
               )
            qtable.setCellWidget(i, 4, deleteSeqCell)
```

```
            visibleCell = QCheckBox(self.chromTable)
            visibleCell.clicked.connect(self.hideSequence)
            visibleCell.setTristate(False)
            qtable.setCellWidget(i, 2, visibleCell)

            if self.chromList[i].isVisible() == True:
                visibleCell.setCheckState(QtCore.Qt.Checked)
            else:
                visibleCell.setCheckState(QtCore.Qt.Unchecked
                    )

            qtable.setItem(i, 0, nameCell)
            qtable.setItem(i, 1, lengthCell)

    def hideBlast(self):
        ch = self.sender()
        ix = self.blastTable.indexAt(ch.pos())
        row = ix.row()
        if ch.checkState() == QtCore.Qt.Checked:
            self.blastList[row].setBlastVisibility(True)
        else:
            self.blastList[row].setBlastVisibility(False)

    def deleteBlast(self):
        ch = self.sender()
        ix = self.blastTable.indexAt(ch.pos())
        row = ix.row()
        self.blastList[row].deleteFamily()
        self.generateBlastTable(self.blastTable)

    def deleteSequence(self):
        ch = self.sender()
        ix = self.chromTable.indexAt(ch.pos())
        row = ix.row()
        self.chromList[row].scene().deleteChromosome(self.
            chromList[row].name)
        self.generateChromTable(self.chromTable)
        self.generateBlastTable(self.blastTable)

    def hideSequence(self):
        ch = self.sender()
        ix = self.chromTable.indexAt(ch.pos())
```

```
        row = ix.row()
        if ch.checkState() == QtCore.Qt.Checked:
            self.chromList[row].scene().hideChromosome(self.
                chromList[row].name, True)
        else:
            self.chromList[row].scene().hideChromosome(self.
                chromList[row].name, False)
        self.generateBlastTable(self.blastTable)

    def changeName(self):
        newName, okPressed = QInputDialog.getText(self, '
            Input new name', 'Input new name')
        if okPressed and newName != '':
            ch = self.sender()
            ix = self.chromTable.indexAt(ch.pos())
            row = ix.row()
            self.chromList[row].name = newName
        self.generateChromTable(self.chromTable)


class BlastInfoWidget(QWidget):
    blastInfoTrigger = QtCore.pyqtSignal(list)

    def __init__(self, chrList):
        super().__init__()
        self.setGeometry(0, 0, 600, 350)
        self.chrList = chrList
        self.initUI()
        self.blastSettings = {'blastType': 'blastn', '
            blastMatrix': 'BLOSUM62', 'minIdent': '90.0', '
            minAln': '1000',
                              'mergeAdj': [False, 0], '
                                 saveFile': False, 'blastYpos
                                 ':False}

    def initUI(self):
        self.setWindowTitle('Perform Blast Comparison')

        self.label = QLabel()
        self.label.setText('Sequences in current scene:')

        self.chrTable = QTableWidget()
```

```
self.chrTable.setRowCount(len(self.chrList))
self.chrTable.setColumnCount(3)
self.chrTable.setHorizontalHeaderLabels(['Name', '
    Length', 'Select?'])
self.chrTable.horizontalHeader().setSectionResizeMode
    (0, QHeaderView.Stretch)
self.chrTable.horizontalHeader().setSectionResizeMode
    (1, QHeaderView.ResizeToContents)
self.chrTable.horizontalHeader().setSectionResizeMode
    (2, QHeaderView.ResizeToContents)

for i in range(0, len(self.chrList)):
    idCell = QTableWidgetItem(self.chrList[i].name)
    lengthCell = QTableWidgetItem(str(len(self.
        chrList[i].sequence)))
    parseCell = QCheckBox(self.chrTable)
    parseCell.setTristate(False)
    parseCell.setCheckState(QtCore.Qt.Checked)
    self.chrTable.setItem(i, 0, idCell)
    self.chrTable.setItem(i, 1, lengthCell)
    self.chrTable.setCellWidget(i, 2, parseCell)

self.settingsButton = QPushButton('Blast␣Options...')
self.blastButton = QPushButton('Start␣Blast')
self.selectAllState = True
self.selectAllButton = QPushButton('Deselect␣All')

self.settingsButton.clicked.connect(self.
    getBlastSettings)
self.blastButton.clicked.connect(self.performBlast)
self.selectAllButton.clicked.connect(self.
    toggleSelectState)

layHBox = QHBoxLayout()
layHBox.addWidget(self.settingsButton)
layHBox.addWidget(self.blastButton)
layHBox.addWidget(self.selectAllButton)

layVBox = QVBoxLayout()
layVBox.addWidget(self.label)
layVBox.addWidget(self.chrTable)
layVBox.addLayout(layHBox)
```

```
        self.setLayout(layVBox)
        self.center()

    def getBlastSettings(self):
        self.window = BlastSettingsWidget(self.blastSettings)
        self.window.show()
        self.window.blastSettingsTrigger.connect(self.
            storeBlastSettings)

    def storeBlastSettings(self, dict):
        self.blastSettings = dict

    def performBlast(self):
        finalTable = []
        for i in range(0, self.chrTable.rowCount()):
            if self.chrTable.cellWidget(i, 2).isChecked() ==
                True:
                item = self.chrTable.item(i, 0).text()
                finalTable.append(item)
        self.blastInfoTrigger.emit([finalTable, self.
            blastSettings])
        self.close()

    def center(self):
        qr = self.frameGeometry()
        cp = QDesktopWidget().availableGeometry().center()
        qr.moveCenter(cp)
        self.move(qr.topLeft())

    def toggleSelectState(self):
        if self.selectAllState is True:
            for i in range(0, self.chrTable.rowCount()):
                self.chrTable.cellWidget(i, 2).setCheckState(
                    QtCore.Qt.Unchecked)
            self.selectAllState = False
            self.selectAllButton.setText('Select␣All')
        else:
            for i in range(0, self.chrTable.rowCount()):
                self.chrTable.cellWidget(i, 2).setCheckState(
                    QtCore.Qt.Checked)
            self.selectAllState = True
```

```
            self.selectAllButton.setText('Deselect␣All')


class GBInfoWidget(QWidget):

    gbInfoTrigger = QtCore.pyqtSignal(list)
    searchPathTrigger = QtCore.pyqtSignal(list)

    def __init__(self, gbList):
        super().__init__()
        self.setGeometry(0, 0, 600, 350)
        self.gbList = gbList
        self.initUI()

    def initUI(self):
        self.setWindowTitle('Parse␣Genbank␣Files')
        self.label = QLabel()
        self.label.setText('The␣following␣sequences␣were␣
            found:')


        self.gbTable = QTableWidget()

        self.generateGbTable(self.gbTable)


        self.addButton = QPushButton('Add␣more␣sequences...')
        self.parseButton = QPushButton('Parse!')
        self.selectAllState = True
        self.selectAllButton = QPushButton('Deselect␣All')

        self.selectAllButton.clicked.connect(self.
            toggleSelectState)
        self.addButton.clicked.connect(self.addMoreSequences)
        self.parseButton.clicked.connect(self.clickingParse)


        layHBox = QHBoxLayout()
        layHBox.addWidget(self.selectAllButton)
        layHBox.addWidget(self.addButton)
        layHBox.addWidget(self.parseButton)
```

```
        layVBox = QVBoxLayout()
        layVBox.addWidget(self.label)
        layVBox.addWidget(self.gbTable)
        layVBox.addLayout(layHBox)

        self.setLayout(layVBox)
        self.center()
        self.setWindowState(QtCore.Qt.WindowActive)


    def generateGbTable(self, qtable):
        qtable.setRowCount(len(self.gbList))
        qtable.setColumnCount(5)
        qtable.setHorizontalHeaderLabels(['File', 'Genbank␣
            Locus','Genbank␣Accession' , 'Length', 'Parse?'])

        qtable.horizontalHeader().setSectionResizeMode(0,
            QHeaderView.Stretch)
        qtable.horizontalHeader().setSectionResizeMode(1,
            QHeaderView.ResizeToContents)
        qtable.horizontalHeader().setSectionResizeMode(2,
            QHeaderView.ResizeToContents)
        qtable.horizontalHeader().setSectionResizeMode(3,
            QHeaderView.ResizeToContents)
        qtable.horizontalHeader().setSectionResizeMode(4,
            QHeaderView.ResizeToContents)

        for i in range(0, len(self.gbList)):
            fileCell = QTableWidgetItem(self.gbList[i][0])
            idCell = QTableWidgetItem(self.gbList[i][2])
            locusCell = QTableWidgetItem(self.gbList[i][1])
            lengthCell = QTableWidgetItem(str(self.gbList[i
                ][3]))
            parseCell = QCheckBox(self.gbTable)
            parseCell.setTristate(False)
            parseCell.setCheckState(QtCore.Qt.Checked)
            qtable.setItem(i, 0, fileCell)
            qtable.setItem(i, 1, locusCell)
            qtable.setItem(i, 2, idCell)
            qtable.setItem(i, 3, lengthCell)
            qtable.setCellWidget(i, 4, parseCell)
```

```
    def clickingParse(self):
        self.getSelectedSeqs()

    def toggleSelectState(self):
        if self.selectAllState is True:
            for i in range(0, self.gbTable.rowCount()):
                self.gbTable.cellWidget(i, 4).setCheckState(
                    QtCore.Qt.Unchecked)
            self.selectAllState = False
            self.selectAllButton.setText('Select␣All')
        else:
            for i in range(0, self.gbTable.rowCount()):
                self.gbTable.cellWidget(i, 4).setCheckState(
                    QtCore.Qt.Checked)
            self.selectAllState = True
            self.selectAllButton.setText('Deselect␣All')

    def center(self):
        qr = self.frameGeometry()
        cp = QDesktopWidget().availableGeometry().center()
        qr.moveCenter(cp)
        self.move(qr.topLeft())

    def addMoreSequences(self):
        plotHandle = QFileDialog.getOpenFileNames(self, '
            Select␣Genbank/Fasta␣File', './',
                                                  'Genbank/
                                                     Fasta␣
                                                     Files␣
                                                     (*.
                                                     genbank␣
                                                     *.gb␣*.
                                                     gbff␣*.
                                                     gbk␣*.
                                                     pgbk␣*.
                                                     fasta␣*.
                                                     fa␣*.fna
                                                     )␣;;␣All
                                                     ␣Files␣
                                                     (*.*)')
        if plotHandle[0]:
            newgbList = gbParser.getNewRecords(plotHandle[0])
```

```
            self.searchPathTrigger.emit(plotHandle[0])
            for item in newgbList:
                self.gbList.append(item)
        self.generateGbTable(self.gbTable)



    def storeBlastSettings(self, dict):
        self.blastSettings = dict

    def getSelectedSeqs(self):
        finalTable = []
        for i in range(0, self.gbTable.rowCount()):
            if self.gbTable.cellWidget(i, 4).isChecked() ==
               True:
                items = [self.gbTable.item(i,0).text(), self.
                    gbTable.item(i,1).text(), self.gbTable.
                    item(i,2).text(),
                            self.gbTable.item(i,3).text()]
                finalTable.append(items)
        self.gbInfoTrigger.emit(finalTable)
        self.close()


class BlastSettingsWidget(QWidget):
    blastSettingsTrigger = QtCore.pyqtSignal(dict)
    def __init__(self, blastSettings):
        super().__init__()
        self.setGeometry(0, 0, 650, 200)
        self.blastSettings = blastSettings
        self.initUI()

    def initUI(self):
        self.setWindowTitle('Blast Settings')
        self.labelBlastType = QLabel()
        self.labelBlastType.setText('Blast Type')

        self.blastTypeGroup = QButtonGroup()
        self.buttonBlastn = QRadioButton('blastn')
        self.buttonTblastx = QRadioButton('tblastx')
        if self.blastSettings['blastType'] == 'blastn':
            self.buttonBlastn.setChecked(True)
```

```
else:
    self.buttonTblastx.setChecked(True)
self.blastTypeGroup.addButton(self.buttonBlastn)
#self.blastTypeGroup.addButton(self.buttonBlastn)
self.blastTypeGroup.addButton(self.buttonTblastx)

self.comboMatrix = QComboBox()
self.comboMatrix.addItem('BLOSUM62')
self.comboMatrix.addItem('BLOSUM90')
self.comboMatrix.addItem('BLOSUM80')
self.comboMatrix.addItem('BLOSUM50')
self.comboMatrix.addItem('BLOSUM45')
self.comboMatrix.addItem('PAM70')
self.comboMatrix.addItem('PAM30')

comboIndex = self.comboMatrix.findText(self.
    blastSettings['blastMatrix'])
if comboIndex > 0:
    self.comboMatrix.setCurrentIndex(comboIndex)
else:
    pass

self.labelMinIdent = QLabel()
self.labelMinIdent.setText('Minimum␣Identity␣(0-100)%
    ')
self.linEditMinIdent = QLineEdit()
self.linEditMinIdent.setText(self.blastSettings['
    minIdent'])

self.labelMinAln = QLabel()
self.labelMinAln.setText('Minimum␣Alignment')
self.linEditMinAln = QLineEdit()
self.linEditMinAln.setText(self.blastSettings['minAln
    '])

self.labelMergeBlasts = QLabel()
self.labelMergeBlasts.setText('Merge␣Adjacent␣Blasts?
    ')
self.checkMergeBlasts = QCheckBox()
self.checkMergeBlasts.clicked.connect(self.
    checkAdjBlastsButton)
self.checkMergeBlasts.setTristate(False)
```

```
        self.linEditMergeBlasts = QLineEdit()
        self.linEditMergeBlasts.setReadOnly(True)

        if self.blastSettings['mergeAdj'][0] == True:
            self.checkMergeBlasts.setCheckState(QtCore.Qt.
                Checked)
            self.linEditMergeBlasts.setText(self.
                blastSettings['mergeAdj'][1])

        else:
            self.checkMergeBlasts.setCheckState(QtCore.Qt.
                Unchecked)
            self.linEditMergeBlasts.setReadOnly(True)

        self.labelSaveFiles = QLabel()
        self.labelSaveFiles.setText('Save␣Blast␣file?')
        self.checkSaveFiles = QCheckBox()
        self.checkSaveFiles.setTristate(False)
        if self.blastSettings['saveFile'] == True:
            self.checkSaveFiles.setCheckState(QtCore.Qt.
                Checked)
        else:
            self.checkSaveFiles.setCheckState(QtCore.Qt.
                Unchecked)

        self.labelBlastYPos = QLabel()
        self.labelBlastYPos.setText('Perform␣Blast␣based␣on␣Y
            ␣pos')
        self.checkBlastYPos = QCheckBox()
        self.checkBlastYPos.setTristate(False)
        if self.blastSettings['blastYpos'] == True:
            self.checkBlastYPos.setCheckState(QtCore.Qt.
                Checked)

        self.buttonSave = QPushButton('Save␣and␣exit')
        self.buttonSave.clicked.connect(self.saveSettings)
        self.buttonExit = QPushButton('Cancel')

        layGbox = QGridLayout()
        layGbox.addWidget(self.labelBlastType, 0, 0)
        layGbox.addWidget(self.buttonBlastn, 1, 0)
        layGbox.addWidget(self.buttonTblastx, 1, 1)
```

```
        layGbox.addWidget(self.comboMatrix, 1, 2)
        layGbox.addWidget(self.labelMinIdent, 2, 0)
        layGbox.addWidget(self.linEditMinIdent, 2, 1)
        layGbox.addWidget(self.labelMinAln, 3, 0)
        layGbox.addWidget(self.linEditMinAln, 3, 1)
        layGbox.addWidget(self.labelMergeBlasts, 4, 0)
        layGbox.addWidget(self.checkMergeBlasts, 4, 1)
        layGbox.addWidget(self.linEditMergeBlasts, 4, 2)
        layGbox.addWidget(self.labelSaveFiles, 5, 0)
        layGbox.addWidget(self.checkSaveFiles, 5, 1)
        layGbox.addWidget(self.labelBlastYPos, 6, 0)
        layGbox.addWidget(self.checkBlastYPos, 6, 1)
        layGbox.addWidget(self.buttonExit, 7, 1)
        layGbox.addWidget(self.buttonSave, 7, 2)

        self.setLayout(layGbox)

        self.center()
        self.show()

    def center(self):
        qr = self.frameGeometry()
        cp = QDesktopWidget().availableGeometry().center()
        qr.moveCenter(cp)
        self.move(qr.topLeft())

    def checkAdjBlastsButton(self):
        if self.checkMergeBlasts.isChecked() == True:
            self.linEditMergeBlasts.setReadOnly(False)
            self.linEditMergeBlasts.setText('50')
        else:
            self.linEditMergeBlasts.setReadOnly(True)
            self.linEditMergeBlasts.setText('')

    def saveSettings(self):
        if self.buttonTblastx.isChecked() == True:
            self.blastSettings['blastType'] = 'tblastx'
            self.blastSettings['blastMatrix'] = self.
               comboMatrix.currentText()
        else:
            self.blastSettings['blastType'] = 'blastn'
```

```
        self.blastSettings['minIdent'] = self.linEditMinIdent
            .text()
        self.blastSettings['minAln'] = self.linEditMinAln.
            text()

        if self.checkMergeBlasts.isChecked() == True:
            self.blastSettings['mergeAdj'][0] = True
            self.blastSettings['mergeAdj'][1] = self.
                linEditMergeBlasts.text()
        else:
            self.blastSettings['mergeAdj'][0] = False
            self.blastSettings['mergeAdj'][1] = 50

        if self.checkSaveFiles.isChecked() == True:
            self.blastSettings['saveFile'] = True
        else:
            self.blastSettings['saveFile'] = False

        if self.checkBlastYPos.isChecked() == True:
            self.blastSettings['blastYpos'] = True
        else:
            self.blastSettings['blastYpos'] = False

        self.blastSettingsTrigger.emit(self.blastSettings)
        self.close()


class SizeSliderWidget(QWidget):
    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        self.slider = QSlider(QtCore.Qt.Horizontal, self)
        self.buttonSave = QPushButton('Save and exit')
        self.buttonSave.clicked.connect(self.saveSettings)
        self.setGeometry(0, 0, 650, 200)

        self.show()

    def saveSettings(self):
        pass
```

```
class CDSInfoWidget(QWidget):
    def __init__(self, cds):
        super().__init__()
        self.cds = cds
        self.initUI()

    def initUI(self):
        #self.setGeometry(0, 0, 750, 400)
        self.setWindowTitle('CDS Info')
        self.label = QLabel()
        self.copyNtButton = QPushButton('Copy nt Sequence to
            clipboard')
        self.copyNtButton.clicked.connect(self.copyNtToClip)
        self.copyAaButton = QPushButton('Copy aa Sequence to
            clipboard')
        self.copyAaButton.clicked.connect(self.copyAaToClip)

        pos = (self.cds.position, self.cds.position + self.
            cds.w)
        labelText = '''\

        Feature Name: {}
        Parent Sequence: {}
        Feature Type: {}
        Feature Location: {}
        Feature Length: {}
        Feature Strand: {}

        FEATURE QUALIFIERS:\n\
                    '''.format(self.cds.name, self.cds.parent
                        .name,self.cds.type, (str(pos[0]) + '
                        - ' + str(pos[1])), self.cds.w, self.
                        cds.strand)
        for key in self.cds.qualifiers:
            if key != 'translation':
                labelText += '\n\t\t{}: {}'.format(key, self.
                    cds.qualifiers[key])
        self.label.setText(labelText)
```

```
        self.copyNtButton = QPushButton('Copy␣nt␣Sequence␣to␣
            clipboard')
        self.copyNtButton.clicked.connect(self.copyNtToClip)
        self.copyAaButton = QPushButton('Copy␣aa␣Sequence␣to␣
            clipboard')
        self.copyAaButton.clicked.connect(self.copyAaToClip)

        layHBox = QHBoxLayout()
        layHBox.addWidget(self.copyNtButton)
        if 'translation' in self.cds.qualifiers:
            layHBox.addWidget(self.copyAaButton)
        layVBox = QVBoxLayout()
        layVBox.addWidget(self.label)
        layVBox.addLayout(layHBox)
        self.setLayout(layVBox)

    def copyNtToClip(self):
        cb = QtGui.QGuiApplication.clipboard()
        cb.clear(mode=cb.Clipboard)
        cdsSeq = str(self.cds.parent.sequence[self.cds.
            position:(self.cds.position + self.cds.w)])
        cb.setText(cdsSeq, mode=cb.Clipboard)

    def copyAaToClip(self):
        cb = QtGui.QGuiApplication.clipboard()
        cb.clear(mode=cb.Clipboard)
        cb.setText(self.cds.qualifiers['translation'][0],
            mode = cb.Clipboard)



#Inherit from QWidget
class MainWidget(QWidget):
    def __init__(self):
        #Super calls the parent object, then we use its
            constructor
        super().__init__()
        self.initUI()

    def initUI(self):
        self.setGeometry(0, 0, 300, 200)
```

```
self.searchPath = './'
self.settings = {
    'blastPath': '',
    'displayType': '',
    'fosmidSize': 0
}
self.loadConfigFile()
print(self.settings)

self.scene = GenomeScene(self.settings)
self.view = GenomeViewer(self.scene)
```

```
#Maximize the window (Qt Keywords (like Qt::
    WindoMaximized) are in the PyQt5.QtCore module
self.setWindowState(QtCore.Qt.WindowMaximized)

#Menu stuff
openPlot = QAction('&Load␣Plot␣File', self)
openPlot.triggered.connect(self.showPlotDialog)

openGenBank = QAction('&Add␣Genbank/Fasta␣sequences',
    self)
openGenBank.triggered.connect(self.showGbDialog)

openBlast = QAction('&Load␣blast␣file', self)
openBlast.triggered.connect(self.showBlastDialog)

cleanCanvas = QAction('&Clean␣canvas', self)
cleanCanvas.triggered.connect(self.getNewCanvas)

deleteBlast = QAction('&Delete␣blasts', self)
deleteBlast.triggered.connect(self.deleteBlasts)

performBlast = QAction('Perform␣Blast...', self)
```

```
performBlast.triggered.connect(self.getBlasts)

savePlot = QAction('&Save␣Plot␣File', self)
savePlot.triggered.connect(self.saveFlexFile)

manageCanvas = QAction('&Edit␣Canvas', self)
manageCanvas.triggered.connect(self.manageFamilies)

changeBlastColor = QAction('&Change␣Blast␣Color',
    self)
changeBlastColor.triggered.connect(self.
    _changeBlastColor)

takeScreenshot = QAction('&Save␣canvas␣as␣image',
    self)
takeScreenshot.triggered.connect(self.
    saveScreenshotDialog)

styleLoad = QAction('&Load␣style␣file', self)
styleLoad.triggered.connect(self.loadStyleFile)

scramble = QAction('&Scramble␣Sequences', self)
scramble.triggered.connect(self.scrambleChrms)

debug = QAction('&Debug', self)
debug.triggered.connect(self.changeChrSize)


menuBar = QMenuBar()
fileMenu = menuBar.addMenu('&File')
editMenu = menuBar.addMenu('&Edit')
debugMenu = menuBar.addMenu('&Debug')
fileMenu.addAction(openPlot)
fileMenu.addAction(openBlast)
fileMenu.addAction(savePlot)
fileMenu.addAction(styleLoad)
fileMenu.addAction(takeScreenshot)
fileMenu.addAction(openGenBank)
editMenu.addAction(manageCanvas)
editMenu.addAction(performBlast)
```

```
        editMenu.addAction(deleteBlast)
        editMenu.addAction(cleanCanvas)
        editMenu.addAction(changeBlastColor)
        debugMenu.addAction(scramble)
        debugMenu.addAction(debug)

        layVBox = QVBoxLayout()
        layVBox.addWidget(menuBar)
        layVBox.addWidget(self.view)
        self.setLayout(layVBox)

        self.setWindowTitle('Flex2')
        self.center()
        self.show()

    def loadConfigFile(self):
        try:
            with open('flex2.config', 'r') as configFile:
                for line in configFile:
                    if line[0] == '#':
                        pass
                    if line.split('=')[0] in self.settings.
                       keys():
                        print('Found', line.split('=')[0], ',
                           the value is', line.split('=')
                           [1].rstrip('\n'))
                        self.settings[line.split('=')[0]] =
                           line.split('=')[1].rstrip('\n')
        except Exception:
            pass

    def center(self):
        qr = self.frameGeometry()
        cp = QDesktopWidget().availableGeometry().center()
        qr.moveCenter(cp)
        self.move(qr.topLeft())

    def manageFamilies(self):
        self.window = BlastFamilyWidget(self.scene.
           blastFamilies, self.scene.chrList)
        self.window.show()
        #self.signal1 = self.window
```

```
def showBlastDialog(self):
    blastHandle = QFileDialog.getOpenFileName(self, '
       Select␣Blast␣File', self.searchPath, 'Blast␣Files␣
       (*.blastn␣*.plot.blastn.clean' +
            '*.blastp␣*.plot.blastp.clean)␣;;␣All␣Files␣
               (*.*)')
    if blastHandle[0]:
        self.getDirectoryFromPath(blastHandle[0])
        parseBlastFile(blastHandle[0], self.scene)

def showPlotDialog(self):
    plotHandle = QFileDialog.getOpenFileName(self, '
       Select␣Plot␣File', self.searchPath, 'Flex␣Files␣
       (*.plot␣*.flex)␣;;␣All␣Files␣(*.*)')
    if plotHandle[0]:
        try:
            newScene = GenomeScene()
            self.view.setScene(newScene)
            if plotHandle[0].split('.')[-1] == 'plot':
                self.getDirectoryFromPath(plotHandle[0])
                parseOldGenomeFile(plotHandle[0],
                   newScene)
            elif plotHandle[0].split('.')[-1] == 'flex':
                self.getDirectoryFromPath(plotHandle[0])
                loadFlexFile(plotHandle[0], newScene)

            #newScene.applyStyle('./style.txt')
            self.scene = newScene
            #self.view = newView
            self.view.update()
        except Exception as e:
            traceback.print_exc()
            self.view.setScene(self.scene)

def deleteBlasts(self):
    length = len(self.scene.blastFamilies)
    for i in range(0, length):
        self.scene.blastFamilies[0].deleteFamily()

def getNewCanvas(self):
    newScene = GenomeScene(self.settings)
```

```
        self.scene = newScene
        self.view.setScene(newScene)

    def getBlasts(self):
        self.window = BlastInfoWidget(self.scene.chrList)
        self.window.blastInfoTrigger.connect(self.
            processBlastOrders)
        self.window.show()

    def saveScreenshotDialog(self):
        scPath = QFileDialog.getSaveFileName(self, 'Select␣
            Directory␣to␣save', self.searchPath, 'PNG␣Format␣
            (*.png)␣;;␣SVG␣Format␣(*.svg)␣;;␣All␣Files␣(*.*)')
        if scPath[1] == 'PNG␣Format␣(*.png)':
            self.getDirectoryFromPath(scPath[0])
            self.saveScreenshotPNG(scPath[0])

        else:
            self.getDirectoryFromPath(scPath[0])
            self.saveScreenshotSVG(scPath[0])

    def saveScreenshotPNG(self, path):
        self.scene.clearSelection()
        self.scene.setSceneRect(self.scene.itemsBoundingRect
            ())
        image = QtGui.QImage(self.view.size().width(), self.
            view.size().height(), QtGui.QImage.Format_ARGB32)
        painter = QtGui.QPainter(image)
        painter.fillRect(image.rect(), QtGui.QBrush(QtCore.Qt
            .white))
        self.view.render(painter)
        painter.end()
        image.save(path + '.png')

    def saveScreenshotSVG(self, path):
        self.scene.clearSelection()
        self.scene.setSceneRect(self.scene.itemsBoundingRect
            ())
        svgGen = QtSvg.QSvgGenerator()
        svgGen.setFileName(path + '.svg')
        svgGen.setSize(QtCore.QSize(self.view.size().width(),
            self.view.size().height()))
```

```
        painter = QtGui.QPainter(self.view)
        painter.begin(svgGen)
        painter.fillRect(self.view.rect(), QtGui.QBrush(
            QtCore.Qt.white))
        self.view.render(painter)
        painter.end()

    def saveFlexFile(self):
        fileHandle = QFileDialog.getSaveFileName(self, '
            Select Directory to save', self.searchPath)
        if fileHandle[0]:
            saveFlexFile(self.scene, fileHandle[0])

    def showGbDialog(self):
        plotHandle = QFileDialog.getOpenFileNames(self, '
            Select Genbank File', self.searchPath, 'Genbank/
            Fasta Files (*.genbank *.gb *.gbff *.gbk *.pgbk *.
            fasta *.fa *.fna) ;; All Files (*.*)')
        if plotHandle[0]:
            self.getDirectoryFromPath(plotHandle[0])
            gbList = gbParser.getNewRecords(plotHandle[0])
            self.window = GBInfoWidget(gbList)
            self.window.gbInfoTrigger.connect(self.
                processGenbanks)
            self.window.searchPathTrigger.connect(self.
                getDirectoryFromPath)
            self.window.show()

    def processGenbanks(self, queryList):
        seqList = queryList
        print('processing sequences, sequences to process:',
            len(seqList))
        #create dictionary
        seqDictGb = {}
        seqDictFasta = {}

        for seq in seqList:
            print("valor en seqList ")
            print(seq)
            targetDict = None
            if gbParser.tryNewFastaFile(seq[0]) is True:
                print('True')
```

```
                targetDict = seqDictFasta
            else:
                targetDict = seqDictGb
            if seq[0] not in targetDict.keys():
                targetDict[seq[0]] = [[seq[1], seq[2], seq
                    [3]]]

            else:
                targetDict[seq[0]].append([seq[1], seq[2],
                    seq[3]])


        chromListGb = gbParser.parseGbFiles(seqDictGb.keys(),
            seqDictGb)
        chromListFasta = gbParser.parseFastaFiles(
            seqDictFasta.keys(), seqDictFasta)
        chromList = chromListGb + chromListFasta
        print('Genbanks:', len(seqDictGb), '+ Fastas:', len(
            seqDictFasta), '=', len(chromList))
        for chrom in chromList:
            newChrom = self.scene.createChromosome(chrom.
                length, chrom.name, 0, 0, chrom.seq)
            for feature in chrom.features:
                # Length / position / strand / name / type /
                    qualifierDict
                newChrom.createGene(int(feature.position[1])
                    - int(feature.position[0]), int(feature.
                    position[0]),
                                    feature.position[2], feature.id,
                                        feature.type, feature.
                                        qualifiers)

    def processBlastOrders(self, list):
        blastSettings = list[1]
        seqList = list[0]
        chrList = []
        for seqName in seqList:
            chrList.append(self.scene.findChromosomeByName(
                seqName))
        print(len(chrList))
        if blastSettings['blastYpos']:
            self.blastBasedOnYPos(chrList, blastSettings)
```

```
        else:
            getFastaFile(chrList)
            runBlastOnSeqs(self.settings['blastPath'] ,
                blastSettings, self.scene)

    def blastBasedOnYPos(self, chrList, blastSettings):
        chrList.sort(key=lambda Chromosome: Chromosome.pos().
            y())

        for i in range(0, len(chrList)-1):
            currList = [chrList[i], chrList[i + 1]]
            getFastaFile(currList)
            runBlastOnSeqs(self.settings['blastPath'],
                blastSettings, self.scene)

    def loadStyleFile(self):
        styleHandle = QFileDialog.getOpenFileName(self, '
            Select Style File', './', 'Text Files (*.txt) ;; 
            All Files (*.*)')
        if styleHandle[0]:
            self.scene.applyStyle(styleHandle[0])

    def printWindowSizes(self):
        self.scene.sortChromosomesByHeight()
        viewPortRect = QtCore.QRect(0, 0, self.view.viewport
            ().width(), self.view.viewport().height())
        visibleSceneRect = self.view.mapToScene(viewPortRect)
            .boundingRect()
        print('INIT STATES')
        print('ViewportSize', self.view.viewport().width(), '
            x', self.view.viewport().height())
        print('ScenerectSize', int(self.scene.sceneRect().
            width()), 'x', int(self.scene.sceneRect().height()
            ))
        print(visibleSceneRect.width(), visibleSceneRect.
            height())
        print('Chromosome positions')
        for chr in self.scene.chrList:
            print('pos:', chr.name, chr.pos().x(), 'x', chr.
                pos().y())
            print('scenePos:', chr.name, chr.scenePos().x(),
                'x', chr.scenePos().y())
```

```
def _changeBlastColor(self):
    pr.disable()
    pr.print_stats(sort='time')
    newColor = QColorDialog.getColor()
    for blastFamily in self.scene.blastFamilies:
        blastFamily.changeBlastColor(newColor)

def scrambleChrms(self):
    for chr in self.scene.chrList:
        newX = random.randint(0, (int(self.scene.
            sceneRect().width() - chr.w / 2)))
        newY = random.randint(0, (int(self.scene.
            sceneRect().height() - chr.h / 2)))
        xdiff = newX - chr.pos().x()
        ydiff = newY - chr.pos().y()
        chr.setPos(QtCore.QPoint(newX, newY))

        for blastFamily in chr.blastList:
            blastFamily.updatePolyPos()

        for cds in chr.geneList:
            cds.moveCDS(xdiff, ydiff)

def getDirectoryFromPath(self, path):
    if path is list:
        splitPath = path[-1].split('/')
    else:
        splitPath = path[-1].split('/')
    del splitPath[-1]
    newPath = ''

    for pathPart in splitPath:
        newPath += pathPart + '/'
    self.searchPath = newPath

def changeChrSize(self):
    value = QInputDialog.getText(self, 'Input Dialog', '
        Input fosmid size')
    if value[1] == True:
        self.settings['fosmidSize'] = int(value[0])
        self.scene.fosmidSizeChanged(self.settings)
```

```
app = QApplication(sys.argv)
pr = cProfile.Profile()
pr.enable()

ex = MainWidget()

sys.exit(app.exec_())
```

A.2. gbParser.py

```
from Bio import SeqIO
from itertools import filterfalse
import argparse
import sys
from subprocess import call



fosmidList = []

class Fosmid():
    def __init__(self, name, length, seq):
        self.seq = seq
        self.name = name
        self.length = length
        self.features = []
        self.featureDict = {}

    def addFeature(self, feature):
        #Check the feature type and add it to the appropiate
           list
        found = False
        for key in self.featureDict:

            if key == feature.type:
                self.featureDict[key] += 1
                feature.id = key.lower() + '_' + str(self.
                   featureDict[key])
                found = True
                break
        #If the relevant type is not yet in the list, just
           add it
        if not found:
            self.featureDict[feature.type] = 1
            feature.id = feature.type + '_0'
        #Finally, add the feature to feature list
        self.features.append(feature)

    def purgeGeneList(self):
```

```
        #Some features appear 2 times in GenBank feature list
           : once as themselves and another as a gene. Both
           entries have the same
        # locus_tag qualifiers, so use those to remove the
           gene entries (CDS entries have more info)
        #In case this does not work correctly: db_xref is
           another possible qualifier, check both features'
           position

        locusList = []
        geneList = []
        otherList = []

        for feature in self.features:
            try:
                if feature.type != 'gene' and feature.
                   qualifiers['locus_tag'] != None:
                    locusList.append(feature)
                else:
                    geneList.append(feature)
            except(KeyError):
                otherList.append(feature)

        print(len(locusList), len(geneList), len(otherList))

#add each feature whose doesnt exist in locusList
        locusList = sorted(locusList, key= lambda x:x.
           qualifiers['locus_tag'])
        newGeneList = [feature for feature in self.features
           if self._checkDuplicates(feature,locusList) ==
           False]
        self.features = locusList + newGeneList

    def _checkDuplicates(self, feature, locuslist):
        #new meta
        pos_start = 0
        pos_end = len(locuslist)-1
        found = False
        while pos_start<pos_end and not found:
            try:
                pos_act = pos_start + (pos_end - pos_start)
                   // 2
```

```
                #i += 1
                #print('inicio: {}, medio: {}, final: {},
                    iteracion: {}'.format(pos_start, pos_act,
                    pos_end, i))
                if locuslist[pos_act].qualifiers['locus_tag']
                    == feature.qualifiers['locus_tag']:
                    found = True
                elif locuslist[pos_act].qualifiers['locus_tag
                    '] < feature.qualifiers['locus_tag']:
                    if pos_start == pos_act:
                        break
                    pos_start = pos_act
                elif locuslist[pos_act].qualifiers['locus_tag
                    '] > feature.qualifiers['locus_tag']:
                    pos_end = pos_act
            except(KeyError):
                return False
        return found

    def returnFeatureTypes(self):
        resultDict = {}
        for feature in self.features:
            if feature.type not in resultDict.keys():
                resultDict[feature.type] = 1
            else:
                resultDict[feature.type] += 1
        return resultDict

    def removeSourceFeature(self):
        for feature in self.features:
            if feature.type == 'source':
                self.features.remove(feature)
                break



class Feature():

    def __init__(self, Fosmid,  gbFeatList):
        self.id = None
        self.fosmid = Fosmid
        self.type = gbFeatList.type
```

```
        self.sequence = None
        #Get position
        self.position = [gbFeatList.location.start.position,
            gbFeatList.location.end.position, gbFeatList.
            location.strand]
        if self.position[2] == -1:
            self.position[2] = '-'
        elif self.position[2] == 1:
            self.position[2] = '+'
        self.qualifiers = gbFeatList.qualifiers


    def getFeatureSequence(self, sequence):
        self.sequence = sequence



#Get Filename
gbFiles = []
inputFiles = ['M1627.gbff']


'''
#Parse genbank,
for file in inputFiles:
    print(file)
    inputFile = SeqIO.parse(file, 'genbank')
    for record in inputFile:
        gbFiles.append(record)

print(len(gbFiles), 'records from', len(inputFiles), 'file(s)
    in input\n')

for gbRecord in gbFiles:

    newFosmid = Fosmid(name=gbRecord.id, length=gbRecord.
        features[0].location.end, seq=gbRecord.seq)
    featureList = gbRecord.features
    for rawFeature in featureList:
        newFeature = Feature(newFosmid, rawFeature)
```

```
            newFeature.getFeatureSequence(str(gbRecord.seq[
                rawFeature.location.start.position:rawFeature.
                location.end.position]))
            newFosmid.addFeature(newFeature)

        print(len(newFosmid.features))
        print(str(newFosmid.returnFeatureTypes()))

        newFosmid.purgeGeneList()

        print(str(newFosmid.returnFeatureTypes()))
'''

def getRecords(filenames):
    gbFiles = []
    for file in filenames:
        print(file)

        if tryNewFastaFile(file) is False:
            inputFile = SeqIO.parse(file, 'genbank')
            print('opening as genbank')
        else:
            inputFile = SeqIO.parse(file, 'fasta')
            print('opening as fasta')

        for record in inputFile:
            print(file,record.name, record.id, len(record.seq
                ))
            gbFiles.append((file,record.name, record.id, len(
                record.seq)))
    return gbFiles

#new meta
def getNewRecords(filenames):
    gbFiles = []
    for file in filenames:
        name = None
        accession = None
        size = 0
        file_type = None
        i = 0
        added = False
```

```
with open(file) as f:
    for line in f:
        if file_type == None:
            content = line[:1]
            if content == '>':
                file_type = 'fasta'
                print('opening as fasta')
            elif content == 'L':
                file_type = 'genBank'
                print('opening as genbank')
            else:
                print("file extension not expected")
                break
        if file_type == 'fasta':
            if i == 0:
                content = line.split()[0]
                name = content[1:]
                accession = content[1:]
            else:
                if line[:1] == '>':
                    print(file, name, accession, size
                        )
                    gbFiles.append((file, name,
                        accession, size))
                    content = line.split()[0]
                    name = content[1:]
                    accession = content[1:]
                    size = 0
                    i = 0
                else:
                    # -1 because of the \n statement
                    size += len(line) - 1

        if file_type == 'genBank':
            if i == 0:
                content = line.split()
                name = content[1]
                size = content[2]
            if i == 3:
                content = line.split()
                accession = content[1]
```

```
                    if not added and name != None and
                       accession != None and size != 0:
                        print(file, name, accession, size)
                        gbFiles.append((file, name, accession
                           , size))
                        added = True
                    elif added:
                        if len(line) == 2:
                            content = line.strip()
                            if content[0] == '/' and content
                               [1] == '/':
                                added = False
                                i = -1
                i += 1
        if file_type == 'fasta':
            print(file, name, accession, size)
            gbFiles.append((file, name, accession, size))
    return gbFiles




def parseFastaFiles(filenames, exceptionDict = None):
    print('no of filenames:', len(filenames))
    fastaRecordList = []
    for file in filenames:
        with open(file, 'r') as filehandle:
            print('processing file')
            inputFile = SeqIO.parse(filehandle, 'fasta')
            for record in inputFile:
                print('processing record')
                for query in exceptionDict[file]:
                    if record.name == query[0] and len(record
                       .seq) == int(query[2]):
                        fastaRecordList.append(record)
                    else:
                        print('match not found!')
                        print(record.name, query[0])
                        print(len(record.seq), int(query[2]))
    fosmidList = []
    print(len(fastaRecordList))
```

```
    for fastaRecord in fastaRecordList:
        newFosmid = Fosmid(name=fastaRecord.name, length=len(
            fastaRecord.seq), seq=fastaRecord.seq)
        fosmidList.append(newFosmid)
    return fosmidList


def parseGbFiles(filenames, exceptionDict = None):

    gbRecordList = []
    for file in filenames:
        with open(file, 'r') as filehandle:
            inputFile = SeqIO.parse(filehandle, 'genbank')
            for record in inputFile:
                for query in exceptionDict[file]:
                    if record.name == query[0] and record.id
                        == query[1] and len(record.seq) == int
                        (query[2]):
                        gbRecordList.append(record)
                    else:
                        print('match not found!')
                        print(record.name, query[0])
                        print(record.id, query[1])
                        print(len(record.seq), int(query[2]))
    fosmidList = []
    for gbRecord in gbRecordList:
        newFosmid = Fosmid(name=gbRecord.name, length=len(
            gbRecord.seq), seq=gbRecord.seq)
        featureList = gbRecord.features
        for rawFeature in featureList:

            newFeature = Feature(newFosmid, rawFeature)
            if (newFeature.position[1] - newFeature.position
                [0]) == newFosmid.length:
                continue
            else:
                newFeature.getFeatureSequence(
                    str(gbRecord.seq[rawFeature.location.
                        start.position:rawFeature.location.end
                        .position]))
                newFosmid.addFeature(newFeature)
```

```
            newFosmid.purgeGeneList()
            newFosmid.removeSourceFeature()
            fosmidList.append(newFosmid)

    return fosmidList

def tryNewFastaFile(filename):
    try:
        with open(filename) as f:
            content = f.readline()[:1]
            if content == '>':
                res = True
            else:
                res = False
        return res
    except Exception:
        return False
```

A.3. herring.py

```
#todo - right now the fasta has to have the same name as its
    header to work. Not a critical error, but a pain in the
    ass

import subprocess, fnmatch, os, copy, logging, itertools, sys
    , pickle, time, math
from builtins import print

from Bio import SeqIO
import multiprocessing

#Log setup so we can have more than 1 log at the same time (
    shamelessly stolen from stackOverflow)
def setup_logger(logger_name, log_file, level=logging.DEBUG):
    l = logging.getLogger(logger_name)
    formatter = logging.Formatter('%(message)s')
    fileHandler = logging.FileHandler(log_file, mode='a')
    fileHandler.setFormatter(formatter)
    l.setLevel(level)
    l.addHandler(fileHandler)

setup_logger('herringMain', 'herring.main.log')
loggerMain = logging.getLogger('herringMain')

sys.path.insert(0, '/home/asier/PycharmProjects/flex2')
import blastParser

#Windows stretcher path
#stretcherPath = 'C:\mEMBOSS\stretcher.exe'
stretcherPath = '/usr/bin/stretcher'
#blastPath = '/opt/ncbi-blast-2.6.0+/bin/'
blastPath = '/home/rohit/alberto/ncbi-blast-2.7.1+/bin/'
prodigalPath = '/home/rohit/asier/programs/prodigal/'

###Lactobacillus stuff
#fastaList = ['B21.fasta', 'C410L1.fasta', 'DOMLa.fasta', '
    JBE245.fasta', 'JBE490.fasta',
#             'LP2.fasta', 'LP3.fasta', 'LY78.fasta', 'LZ227.
    fasta', 'MF1298.fasta', 'RI113.fasta',
```

```
#               'SRCM100434.fasta', 'ZJ316.fasta', 'WCFS1.fasta
    ']
#fastaList = ['WCFS1.fasta', 'LZ227.fasta', 'B21.fasta']
#refFasta = 'lactobacillus.coreWCFS1.fasta'

###Methanosarcina stuff
#fastaList = ['C16.fasta', 'Goe1.fasta', 'LYC.fasta', 'S-6.
    fasta', 'SarPi.fasta', 'Tuc01.fasta', 'WWM610.fasta']
fastaList = ['S-6.fasta', 'LYC.fasta', 'WWM610.fasta']
refFasta = 'methanosarcina.core.fasta'

###Clostridioides stuff
#fastaList = ['630.fasta', '630Drm.fasta', '630Dta.fasta', '
    BI9.fasta', 'CD196.fasta', 'CD630DERM.fasta', 'CF5.fasta',
#               'M68.fasta', 'M120.fasta', 'NAP11.fasta', '
    R0104a.fasta', 'R20291.fasta', 'W0003a.fasta',
#               'W0022a.fasta', 'W0023a.fasta']
#refFasta = 'clostridioides.core.fasta'

###Listeria stuff
#fastaList = ['CFSAN023459.fasta', 'EGD.fasta', 'EGDe.fasta',
     'FDA00011238.fasta', 'FORC049.fasta', 'H34.fasta',
#               'J0161.fasta', 'L2624.fasta', 'L2676.fasta', '
    LM11.fasta', 'LM850658.fasta', 'M7.fasta',
#               'NCTC10357.fasta', 'NTSN.fasta', 'WSLC1019.
    fasta']
#refFasta = 'listeria.core.fasta'


fastaListLengths = {}
fileList = []
masterFamilyList = []

#How many cores will be used?
cores = 3
#Hopefully on a near future user will enter this number, but
    today is not that day. Today you must write a number

#Stuff anti-weird numbers
if cores < 1:
    print('Not␣a␣good␣number␣of␣cores.␣Please␣insert␣a␣
        natural␣number')
```

```
    #Natural numbers are those positive ones excluding zero
       and without decimals. Dont change it, looks cool
    exit(-1)
f = math.factorial
max_number_of_combinations = f(cores) // f(2) // f(cores - 2)
#author's note: we use the integer division (//) to avoid
   possible oveflows
if cores > max_number_of_combinations:
    cores = max_number_of_combinations


class GapFamily:
    def __init__(self, gapList):
        self.parents = (gapList[0][0], gapList[0][4])
        self.gapList = []
        for gapItem in gapList:
            newGap = Gap(gapItem, self)
            self.gapList.append(newGap)

    def writeGapInfo(self, gapList, filename):
        #Super duper ultra cool new mode to check and create
           (if needed) the directory for the filename
        # Proudly made in StackOverflow
        os.makedirs(os.path.dirname('./logs/' + filename),
           exist_ok=True)
        with open('./logs/' + filename, "w") as output:
            output.write('NUMBER OF GAPS: {}\n\n'.format(len(
               gapList)))
            for i, gap in enumerate(gapList):
                gapStr = gap.buildInfoStr()
                output.write(gapStr)
                if i < len(gapList) - 1:
                    output.write('
                       ----------------------------------------\n\
                       n')

    def getParentSeq(self, parent=0):
        gapName = ''
        gapSeq = None
        # get sequence from the fasta file
        for filePair in fileList:
            if self.parents[parent] in filePair:
```

```
            gapName = filePair[parent]
        for record in SeqIO.parse(gapName, 'fasta'):
            if record.name == self.parents[1]:
                gapSeq = record.seq
        return gapSeq

    def findAdjacentGaps(self, gap):
        #sort gaps by seq1pos
        self.gapList.sort(key=lambda gap: gap.parentPos[0])
        #get the index for the target gap
        targetIndex = self.gapList.index(gap)
        prevGap = None
        nextGap = None
        loggerMain.info('target index: {}, gapListLen: {}'.
            format(targetIndex, len(self.gapList)))
        if targetIndex != len(self.gapList)-1:
            nextGap = targetIndex + 1
        if targetIndex != 0:
            prevGap = targetIndex - 1
        if prevGap is not None:
            prevGap = self.gapList[prevGap]
        if nextGap is not None:
            nextGap = self.gapList[nextGap]
        return (prevGap, nextGap)

    def equalize(self):
        refName = getRefFastaName(refFasta)
        loggerMain.info('parents[0] = {}, refName = {}'.
            format(self.parents[0], refName))
        if self.parents[0] == refName:
            parents1 = self.parents[0]
            self.parents = (refName, parents1)
            loggerMain.info('fixed: parent[0] (parent) is {}
                and parent[1] (core) is {}'.format(self.
                parents[0],




        for gap in self.gapList:
```

```
            loggerMain.info('gap.parents[0] = {}, family.
                parents[0] is {}'.format(gap.parents[0], self.
                parents[0]))
            if gap.parents[0] != self.parents[0]:

                newParent = copy.copy(gap.corePos)
                newCore = copy.copy(gap.parentPos)

                gap.parents = self.parents
                gap.corePos = newCore
                gap.parentPos = newParent
            loggerMain.info('parents = {} , corePos = {} , 
                parentPos = {}'.format(gap.parents, gap.
                corePos,
```

```
    def getGapsByType(self, type='analyze'):
        targetList = []
        for gap in self.gapList:
            if gap.type == type:
                targetList.append(gap)
        return targetList

    def findOrphans(self, type = ('left', 'right', 'both')):
        orphanList = []
        for gap in self.gapList:
            orphanInfo = gap.isGapOrphan()
            if orphanInfo[0] == True and orphanInfo[1] in
                type:
                orphanList.append(gap)
        return orphanList

class Gap():
    def __init__(self, gapList, family):
        self.family = family
        self.parents = (gapList[0], gapList[4])
        self.corePos = (gapList[5], gapList[6])
```

```
        self.parentPos = (gapList[1], gapList[2])
        self.type = gapList[8]
        self.leftBorder = None
        self.rightBorder = None
        self.leftMatch = []
        self.rightMatch = []
        no = str(len(self.family.gapList))
        self.name = str(str(self.parents[0]) + '-' + str(self
            .parents[1]) + '_' + no)
        self.gapLength = self.corePos[1] - self.corePos[0]

        if self.gapLength < 0:
            self.gapLength *= -1
            self.corePos = (self.corePos[1], self.corePos[0])

    def buildInfoStr(self):
        diagStr = ''
        diagStr += ('Gap Name: {}\n'.format(self.name))
        diagStr += ('PARENT INFO:\n')
        diagStr += ('Parent 1: {}\n\t{} - {}\n'.format(self.
            parents[1], self.corePos[0], self.corePos[1]))
        diagStr += ('Parent 2: {}\n\t{} - {}\n'.format(self.
            parents[0], self.parentPos[0], self.parentPos[1]))
        diagStr += ('EDGES\nLEFT BORDER: {}\n'.format(len(
            self.leftMatch)))
        for match in self.leftMatch:
            diagStr += ('\t{}\n'.format(match.name))
        diagStr += ('\nRIGHT BORDER: {}\n'.format(len(self.
            rightMatch)))
        for match in self.rightMatch:
            diagStr += ('\t{}\n'.format(match.name))
        return diagStr

    def getParentSeq(self, parent=0):
        gapName = ''
        gapSeq = None
        # get sequence from the fasta file
        for filePair in fileList:
            if self.parents[parent] in filePair:
                gapName = filePair[parent]
        for record in SeqIO.parse(gapName + '.fasta', 'fasta'
            ):
```

```
            if record.name == gapName:
                gapSeq = record.seq
        return gapSeq

    def getBorderSequence(self, size):
        loggerMain.info('finding␣border␣sequences␣for␣{}␣with
            ␣size␣{}'.format(self.name, size))
        loggerMain.info('parent␣sequence␣used␣is␣{}'.format(
            self.parents[0]))
        gapSeq = self.getParentSeq()
        #Check if we can pick a sequence of the required size
            without taking a gap by mistake:
        adjGaps = self.family.findAdjacentGaps(self)
        #Let's start with the left border
        if adjGaps[0] is not None:
            leftGapPos = adjGaps[0].parentPos[1]
            if (self.parentPos[0] - leftGapPos) < size:
                loggerMain.info('there␣is␣not␣enough␣space␣
                    for␣a␣full␣left␣border!␣size:␣{},␣space␣
                    between␣gaps:␣{}'
                                ''.format(size, self.
                                    parentPos[0] - leftGapPos
                                    ))
                gapSeqLeft = ((self.parentPos[0] - leftGapPos
                    ), gapSeq[leftGapPos:self.parentPos[0]])
            else:
                gapSeqLeft = (size, gapSeq[(self.parentPos[0]
                    - size):self.parentPos[0]])
        else:
            if (self.parentPos[0] - size) > 0:
                gapSeqLeft = (size, gapSeq[(self.parentPos[0]
                    - size):self.parentPos[0]])
            else:
                gapSeqLeft = (size, gapSeq[:self.parentPos
                    [0]])
        #right border is next
        if adjGaps[1] is not None:
            rightGapPos = adjGaps[1].parentPos[0]
            if (rightGapPos - self.parentPos[1]) < size:
                loggerMain.info('there␣is␣not␣enough␣space␣
                    for␣a␣full␣right␣border!␣size:␣{},␣space␣
                    between␣gaps:␣{}'
```

```
                                ''.format(size,rightGapPos -
                                    self.parentPos[1]))
            gapSeqRight = ((rightGapPos - self.parentPos
                [1]), gapSeq[self.parentPos[1]:rightGapPos
                ])
        else:
            gapSeqRight = (size, gapSeq[self.parentPos
                [1]:self.parentPos[1] + size])
    else:
        if (self.parentPos[1] + size) < len(gapSeq):
            gapSeqRight = (size, gapSeq[self.parentPos
                [1]:self.parentPos[1] + size])
        else:
            gapSeqRight = (size, gapSeq[self.parentPos
                [1]:])
    #then return both borders
    self.leftBorder = gapSeqLeft
    self.rightBorder = gapSeqRight
    loggerMain.info('borders for {} found: left border is
        {}, while right border is {}'.format(self.name,
                                                [(
                                                    self
                                                    .
                                                    parentPos
                                                    [0]
                                                     -

                                                    self
                                                    .
                                                    leftBorder
                                                    [0])
                                                    ,
                                                    self
                                                    .
                                                    parentPos
                                                    [0]],
```

```
                                                    [self
                                                        .
                                                        parentPos
                                                        [1],

                                                        self
                                                        .
                                                        parentPos
                                                        [1]
                                                         +

                                                        self
                                                        .
                                                        rightBorder
                                                        [0]])
                                                        )
        return [gapSeqLeft, gapSeqRight]

    def addBorderGapRef(self, gap, type):
        if type == 'left':
            self.leftMatch.append(gap)
        elif type == 'right':
            self.rightMatch.append(gap)

    def getOrphanList(self):
        orphanList = []
        for leftGap in self.leftMatch:
            if leftGap not in self.rightMatch:
                orphanList.append((leftGap, 'left'))

        for rightGap in self.rightMatch:
            if rightGap not in self.leftMatch:
                orphanList.append((rightGap, 'right'))
        return orphanList

    def isGapOrphan(self):
        if len(self.leftMatch) == len(self.rightMatch) and
           len(self.leftMatch) + len(self.rightMatch) != 0:
            return [True, 'both']
        elif len(self.leftMatch) < len(self.rightMatch):
            return [True, 'right']
        else:
```

```
            return [True, 'left']
            #Notice that there are some gaps that will be
               classified as left even they are not, like
               those with len
            # for both borders equal zero, but we do this on
               purpose because the only call for this
               function is to
            # classify the gaps in diagnostic Good or All.
               Actually we could dessign this function only
               with the two
            # first lines and a return [True, 'left'] at the
               end, user wont notice any difference.
            # I wont swap the function as told just because
               other programmers may modify this section in
               order to expand
            # its use, then maybe they find useful this '
               extended version'


def saveHerringData(filename):
    with open(filename, 'wb') as output:
        pickle.dump(masterFamilyList, output, pickle.
           HIGHEST_PROTOCOL)
        pickle.dump(fileList, output, pickle.HIGHEST_PROTOCOL
           )

def loadHerringData(filename):
    with open(filename, 'rb') as input:
        familylist = pickle.load(input)
        filelist = pickle.load(input)
    return(familylist, filelist)

def alignSeqsByBlast(seq1, seq2, borderSize, name):
    loggerMain.info('Aligning␣2␣Sequences␣by␣Blast')
    outfile = name + 'blastResults.temp.blastn'
    loggerMain.info('Running␣blastn...')
    subprocess.call([blastPath + 'blastn', '-query', seq1, '-
       subject', seq2, '-out', outfile, '-outfmt', '6'])
    statsDict = {'Matches%: 0, 'AlnLen': 0, 'Gaps': 0, '
       Mismatches': 0}
    if os.stat(outfile).st_size > 0:
        loggerMain.info('There␣was␣a␣match!')
```

```
        with open(outfile, 'r') as blastHandle:
            statsDict = {'Matches%:0 , 'AlnLen': 0, 'Gaps':
                0, 'Mismatches' : 0}
            #remember: queryID / subjectID / percentage of
                identical matches / alignment length / n of
                mismatches
            # / n of gaps / start in query / end in query /
                start in subject / end in subject / E-value /
                bitscore
            blastLineInfo = blastHandle.readline().split('\t'
                )
            statsDict['Matches%'] = float(blastLineInfo[2])
            statsDict['AlnLen'] = float(int(blastLineInfo[3])
                / borderSize)
            statsDict['Mismatches'] = int(blastLineInfo[4])
            statsDict['Gaps'] = int(blastLineInfo[5])
    else:
        loggerMain.info('No␣match␣found')
    os.remove(outfile)
    return statsDict

def findAllGaps(blastFamily, thresholdMin, thresholdAnalysis,
    masterFamilyList):
    loggerMain.info('Looking␣for␣gaps...')
    loggerMain.info('ThresholdMin␣is␣{},␣ThresholdAnalysis␣is
        ␣{}'.format(thresholdMin, thresholdAnalysis))
    blastFamily._equalize()
    blastFamily.sortHits()
    loggerMain.info('N␣of␣total␣Blasts:␣{}'.format(len(
        blastFamily.blastList)))
    blastList = blastFamily.blastList
    gapList = []

    for i in range(0, len(blastList) - 1):
        loggerMain.info('Iteration␣n␣{}'.format(i))
        # Get the 2 blasts to compare
        fstBlast = copy.copy(blastList[i])
        scdBlast = copy.copy(blastList[i + 1])
        # Check if the blasts are reversed, and if they are
            fix them
        if fstBlast.seq1pos[1] < fstBlast.seq1pos[0]:
```

```
        fstBlast.seq1pos = (fstBlast.seq1pos[1], fstBlast
            .seq1pos[0])
    if scdBlast.seq1pos[1] < scdBlast.seq1pos[0]:
        scdBlast.seq1pos = (scdBlast.seq1pos[1], scdBlast
            .seq1pos[0])
    # calculate distance between both blasts in both
        sequences
    loggerMain.info('blast1pos: {} - {}'.format(fstBlast.
        seq1pos[0], fstBlast.seq1pos[1]))
    loggerMain.info('blast2pos: {} - {}'.format(scdBlast.
        seq1pos[0], scdBlast.seq1pos[1]))
    seq1dtce = abs(scdBlast.seq1pos[0] - fstBlast.seq1pos
        [1])
    seq2dtce = abs(scdBlast.seq2pos[0] - fstBlast.seq2pos
        [1])
    loggerMain.info('Distances between blasts: seq1 {},
        seq2 {}'.format(seq1dtce, seq2dtce))
    total = 0
    # Check if the distances in both sequences meet the
        required threshold
    if seq1dtce > thresholdMin:
        total += 1
    if seq2dtce > thresholdMin:
        total += 1
    # then assign a category depending on the distances
    # gapList is: parent1, seq1pos1, seq1pos2, seq1dtce,
        parent2, seq2pos1, seq2pos2, seq2dtce, type
    if total == 0:
        loggerMain.info('Total is {}, which means there
            is no gap'.format(total))
        pass
    elif total > 0:
        loggerMain.info('Total is {}, which means there
            is a gap'.format(total))
        type = 'ignore'

        if seq1dtce + seq2dtce > thresholdAnalysis:
            type = 'analyze'

        gapList.append([fstBlast.parents[0], fstBlast.
            seq1pos[1], scdBlast.seq1pos[0], seq1dtce,
```

```
                                fstBlast.parents[1], fstBlast.
                                    seq2pos[1], scdBlast.seq2pos
                                    [0], seq2dtce, type])
            loggerMain.info('Sotring gap as gapList {}'.
                format(
                 [fstBlast.parents[0], fstBlast.seq1pos[1],
                    scdBlast.seq1pos[0], seq1dtce, fstBlast.
                    parents[1],
                  fstBlast.seq2pos[1], scdBlast.seq2pos[0],
                     seq2dtce, type]))

    # Sort the list by seq1pos1
    gapList.sort(key=lambda x: x[1])
    # create gap family, then add it to the master list
    if len(gapList) > 0:
        newGapFamily = GapFamily(gapList)
        masterFamilyList.append(newGapFamily)

def getRefFastaName(refFasta):
    return SeqIO.read(refFasta, 'fasta').name

def createCombinedFasta(fastaList):
    loggerMain.info('Combining fastas')
    with open('combinedFasta.temp', 'w') as output:
        for fasta in fastaList:
            for record in SeqIO.parse(fasta, 'fasta'):
                if (str(record.name), fasta) not in fileList:
                    fileList.append((str(record.name), fasta)
                        )
                if record.name not in fastaListLengths.keys()
                    :
                    fastaListLengths[record.name] = len(
                        record.seq)
                output.write('>' + str(record.name) + '\n')
                output.write(str(record.seq) + '\n')
    loggerMain.info('Fastas combined')

def removeBlastDBFiles():
    loggerMain.info('cleaning Blast DB files...')
    dirFiles = os.listdir(os.curdir)
    removedFileList = []
    removedFileCounter = 0
```

```
    for file in dirFiles:
        if file.split('.')[0] == 'dbTemp':
            removedFileCounter += 1
            removedFileList.append(file)
            os.remove(file)
    loggerMain.info('{} files removed: {}'.format(
       removedFileCounter, removedFileList))

def performBlastAgainstCore(fastaList, refFasta, minThres,
   maxThres, masterFamilyList):
    for fasta in fastaList:
        createCombinedFasta([fasta, refFasta])
        subprocess.call([blastPath + 'makeblastdb', '-in', '
           combinedFasta.temp', '-out', 'dbTemp', '-dbtype',
           'nucl'])
        subprocess.call(
            [blastPath + 'blastn', '-query', 'combinedFasta.
               temp', '-db', 'dbTemp', '-out', 'blastSeqs.
               blastn',
             '-num_threads', '4', '-outfmt', '6'])
        os.remove('combinedFasta.temp')
        removeBlastDBFiles()

        blastHits = blastParser.parseBlastFile('blastSeqs.
           blastn', minIdentity=85, minAln=1500)
        blastFamilies = blastParser.groupHits(blastHits)
        for family in blastFamilies:
            family._equalize()
            family.addParentLengths(fastaListLengths)
            family.removeOwnHits()
            family.removeInternalHits()
            family.removeSamePosHits()
            family.removeStrangeHits()
            findAllGaps(family, thresholdMin=minThres,
               thresholdAnalysis=maxThres, masterFamilyList=
               masterFamilyList)

def equalizeBorderSizes(gap1borders, gap2borders):
    # remember: getBorderSequence returns a tuple: ((
       leftBorderSize, leftBorderSeq), (rightBorderSize,
       rightBorderSeq))
```

```
# borders might have different sizes, so we have to
    equalize the border size
# start with the left border:
if (gap1borders[0][0] + gap2borders[0][0]) > 0:
    if gap1borders[0][0] != gap2borders[0][0]:
        loggerMain.info('left side borders are not of the
            same size! gap1: {}, gap2: {}'.format(
            gap1borders[0][0],




        if gap1borders[0][0] > gap2borders[0][0]:
            #we calculate the difference with the total
                bordar size because it has the same result
            diff = gap1borders[0][0] - gap2borders[0][0]
            gap1borders[0] = (gap1borders[0][0] - diff,
                gap1borders[0][1][diff:])
            loggerMain.info('Fixed gap1 border: now
                gap1size is {} and gap2size is {}'.format(
                gap1borders[0][0],




        elif gap2borders[0][0] > gap1borders[0][0]:
            diff = gap2borders[0][0] - gap1borders[0][0]
            gap2borders[0] = (gap2borders[0][0] - diff,
                gap2borders[0][1][diff:])
            loggerMain.info('Fixed gap2 border: now
                gap1size is {} and gap2size is {}'
                            ''.format(gap1borders[0][0],
                                gap2borders[0][0]))
        loggerMain.info('gap1leftBorder: {},
            gap2leftBorder: {}'.format(len(gap1borders
            [0][1]),
```

```
#right border:
if (gap1borders[1][0] + gap2borders[1][0]) > 0:
    if gap1borders[1][0] != gap2borders[1][0]:
        loggerMain.info('right␣side␣borders␣are␣not␣of␣
            the␣same␣size!␣gap1:␣{},␣gap2:␣{}'.format(
            gap1borders[1][0],



        if gap1borders[1][0] > gap2borders[1][0]:
            diff = gap1borders[1][0] - gap2borders[1][0]
            gap1borders[1] = (gap1borders[1][0] - diff,
                gap1borders[1][1][:len(gap1borders[1][1])
                - diff])
            loggerMain.info(
                'Fixed␣gap1␣border:␣now␣gap1size␣is␣{}␣
                    and␣gap2size␣is␣{}'.format(gap1borders
                    [1][0],



        elif gap2borders[1][0] > gap1borders[1][0]:
            diff = gap2borders[1][0] - gap1borders[1][0]
            gap2borders[1] = (gap2borders[1][0] - diff,
                gap2borders[1][1][:len(gap2borders[1][1])
                - diff])
            loggerMain.info('Fixed␣gap2␣border:␣now␣
                gap1size␣is␣{}␣and␣gap2size␣is␣{}'.format(
                gap1borders[1][0],
```

```
            loggerMain.info('gap1rightBorder: {}, '
                'gap2rightBorder: {}'.format(len(gap2borders
                [1][1]),




        else:
            loggerMain.info('No need to fix borders: gap1size
                is {} and gap2size is {}'.format(gap1borders
                [1][0],



def compareTwoGaps(gap1, gap2, size, actualLog, name,
    storeResults = True):
    # Get the border sequences for both gaps
    gap1borders = gap1.getBorderSequence(size)
    gap2borders = gap2.getBorderSequence(size)
    actualLog.info(gap1borders)
    actualLog.info(gap2borders)
    #equalize borders if required
    if gap1borders[0][1] != None and gap2borders[0][1] !=
        None and gap1borders[0][0] != gap2borders[0][0]:
        equalizeBorderSizes(gap1borders, gap2borders)
    if gap1borders[1][1] != None and gap2borders[1][1] !=
        None and gap1borders[1][0] != gap2borders[1][0]:
        equalizeBorderSizes(gap1borders, gap2borders)
    #call stretcher and get the alignment. first, prepare the
        fasta files
    tempFastaFiles = [name + 'gap1left.temp.fasta', name + '
        gap1right.temp.fasta',
                          name + 'gap2left.temp.fasta', name
                              + 'gap2right.temp.fasta']

    actualLog.info('comparing left border')
```

```
if gap1borders[0][1] != None and gap2borders[0][1] !=
   None:
    with open(tempFastaFiles[0], 'w') as gap1left:
        gap1left.write('>gap1left\n')
        gap1left.write(str(gap1borders[0][1]))
    with open(tempFastaFiles[2], 'w') as gap2left:
        gap2left.write('>gap2left\n')
        gap2left.write(str(gap2borders[0][1]))
    leftalnResults = alignSeqsByBlast(name + 'gap1left.
       temp.fasta', name + 'gap2left.temp.fasta',
                                        gap1borders[0][0],
                                           name)
else:
    actualLog.info('No sequences for the left border:
       border 1 is {} and border 2 is {}'
                   ''.format(len(gap1borders[0][1]), len(
                      gap2borders[0][1])))
    leftalnResults = {'Identity': -1, 'Gaps': 0, 'Matches
       %': 0, 'AlnLen': 0, 'Mismatches': 0}

actualLog.info('comparing right border')
if gap1borders[1][1] != None and gap2borders[1][1] !=
   None:
    with open(tempFastaFiles[1], 'w') as gap1right:
        gap1right.write('>gap1right\n')
        gap1right.write(str(gap1borders[1][1]))
    with open(tempFastaFiles[3], 'w') as gap2right:
        gap2right.write('>gap2right\n')
        gap2right.write(str(gap2borders[1][1]))
    rightalnResults = alignSeqsByBlast(name + 'gap1right.
       temp.fasta', name + 'gap2right.temp.fasta',
                                         gap1borders[1][0],
                                            name)
else:
    actualLog.info('No sequences for the right border')
    rightalnResults = {'Identity': -1, 'Gaps': 0, '
       Matches%': 0, 'AlnLen': 0, 'Mismatches': 0}

#remove all temp fasta files
for file in tempFastaFiles:
    if file in os.listdir(os.curdir):
        os.remove(file)
```

```
    #check if they are similar. If they are, then add borders
    response = [False, False]
    data = []
    actualLog.info('leftalnResults: {}'.format(leftalnResults
        ))
    if leftalnResults['Matches%] > 89 and leftalnResults['
        AlnLen'] > 0.89:
         actualLog.info('left sides match')
         response[0] = True
         if storeResults == True:
             gap1.addBorderGapRef(gap2, 'left')
             gap2.addBorderGapRef(gap1, 'left')
             data.append('left')

    actualLog.info('rightalnResults: {}'.format(
        rightalnResults))
    if rightalnResults['Matches%] > 89 and leftalnResults['
        AlnLen'] > 0.89:
         actualLog.info('right sides match')
         response[1] = True
         if storeResults == True:
             gap1.addBorderGapRef(gap2, 'right')
             gap2.addBorderGapRef(gap1, 'right')
             data.append('right')

    if response[0] != response[1]:
         loggerMain.info('both borders do not match!, left
            border: {}, right border: {}'.format(response[0],




         if closeGapByCorePos(gap1, gap2, size, name):
             data.append('right')
    return (gap1, gap2, data)

def compareGapFamilies(gapFamily1, gapFamily2, size,
    actualLog):
    # we want all combinations, not permutations (
        permutations also include the order of the elements -
        so AB & BA)
```

```
    # combList = itertools.combinations(gapFamily1.gapList +
       gapFamily2.gapList, 2)

    # Actually we don't want all combinations either, we just
       want all combinations between 2 sets (e.g. if list1 =
       25
    # and list2 = 15, there would be 780 combinations but
       only 300 combinations between both sets -> Try list
    # comprehensions instead, and build an iterator if it
       becomes too much of a memory burden (it will)
    currComb = 0
    combList = iter([(x, y) for x in gapFamily1.getGapsByType
       ('analyze') for y in gapFamily2.getGapsByType('analyze
       ')])

    familyName = './' + gapFamily1.parents[0] + ',' +
       gapFamily2.parents[0]
    if not os.path.exists('./temp/'):
        os.mkdir('./temp/')

    result = []
    for gapPair in combList:
        currComb += 1
        if gapPair[0].family != gapPair[1].family:
            actualLog.info('comparing␣gap␣pair␣{}␣-␣{}'.
               format(gapPair[0].name, gapPair[1].name))
            result.append(compareTwoGaps(gapPair[0], gapPair
               [1], size, actualLog, name = familyName))

    return result

def checkForGapClusters(masterFamilyList):
    gapClusterList = []
    gapMasterList = []
    gapCheckedList = []
    #populate gapMasterList
    for gapFamily in masterFamilyList:
        for gap in gapFamily.getGapsByType('analyze'):
            gapMasterList.append(gap)
    #find clusters
    for gap in gapMasterList:
        if gap not in gapCheckedList:
```

```
            if set(gap.leftMatch) == set(gap.rightMatch):
                newCluster = [gap]
                for borderGap in gap.leftMatch:
                    if set(borderGap.leftMatch) == set(
                       borderGap.rightMatch):
                        newList = copy.copy(borderGap.
                           leftMatch)
                        newList.remove(gap)
                        newList.append(borderGap)
                        if set(gap.leftMatch) == set(newList)
                           :
                            newCluster.append(borderGap)
                for gap in newCluster:
                    gapCheckedList.append(gap)
                gapClusterList.append(newCluster)
    return gapClusterList

def dumpClusterData(gapClusterList):
    # get a list of all sequences analyzed:
    analyzedList = []
    for filePair in fileList:
        if filePair[1] != refFasta:
            analyzedList.append(filePair[0])

    #also get a list for all gaps not part of a gapCluster
    unclusteredGaps = []
    # create and populate the fastaList
    fastaList = []
    for file in fileList:
        if file[1] != refFasta:
            fastaList.append(file[1])

    performProdigalTraining(fastaList)
    #If the cluster is a real cluster (it has more than one
       gap), then add it to the unclustered list
    for i, gapCluster in enumerate(gapClusterList):
        #We want to have in the info file which sequences do
           not have the
        if len(gapCluster) > 1:
            clusterGapNames = []
            clusterGapIndexes = []
            orfDict = getORFs(gapCluster, i)
```

```
for gap in gapCluster:
    clusterGapNames.append(gap.parents[0])
    clusterGapIndexes.append(gapCluster.index(gap
        ))
infoFile = open('./other/gapCluster_{}.txt'.
    format(i), 'w')
infoFile.write('NAME\tPOS1\tPOS2\tLEN\tNOOFGENES\
    n')
fastaFile = open('./other/gapCluster_{}.fasta'.
    format(i), 'w')
for name in analyzedList:
    if name in orfDict.keys():
        index = clusterGapNames.index(name)
        gap = gapCluster[index]
        # write fasta file:
        fastaFile.write('>{}-{}\n'.format(gap.
            name, gap.parentPos[1] - gap.parentPos
            [0]))
        fastaFile.write('{}\n'.format(gap.
            getParentSeq()[gap.parentPos[0]:gap.
            parentPos[1]]))
        # write info file:
        infoFile.write('{}\t{}\t{}\t{}\t{}\n'.
            format(gap.parents[0], gap.parentPos
            [0], gap.parentPos[1],
```

```
                                                                    gap
                                                                        .
                                                                        parentPos
                                                                        [1]

                                                                        -

                                                                        gap
                                                                        .
                                                                        parentPos
                                                                        [0],

                                                                        len
                                                                        (
                                                                        orfDict
                                                                        [
                                                                        name
                                                                        ])
                                                                        )
                                                                        )

                    else:
                        # write info file:
                        infoFile.write('{}\t{}\t{}\t{}\n'.format(
                            name, 'NULL', 'NULL', 'NULL'))

                infoFile.close()
                fastaFile.close()
            else:
                unclusteredGaps.append(gapCluster[0])

def closeGapByCorePos(gap1, gap2, size, name, storeResults =
    True):
    if (gap1.corePos[1] - gap2.corePos[1]) > 0:
        longGap = gap1
        targetGap = gap2
        coreDiff = gap1.corePos[1] - gap2.corePos[1]
    else:
        longGap = gap2
        targetGap = gap1
        coreDiff = gap2.corePos[1] - gap1.corePos[1]
```

```
longBorder = longGap.getBorderSequence(size + coreDiff)
    [1]
targetBorder = targetGap.getBorderSequence(size +
    coreDiff)[1]
if longBorder[0] < coreDiff:
    loggerMain.info('\tNot␣enough␣space␣for␣the␣check!␣
        long␣border␣size:␣{},␣coreDiff:␣{}'.format(
        longBorder[0],




    return False
else:
    targetBorder = [targetBorder[0] - coreDiff,
        targetBorder[1][coreDiff:]]
    #longBorder = [longBorder[0] - coreDiff, longBorder
        [1][coreDiff:]]
    eqBorders = equalizeBorders(longBorder, targetBorder)
    tempFastaFiles = [name + 'border1Reg.temp.fasta',
        name + 'border2Reg.temp.fasta']

    loggerMain.info('comparing␣right␣border')
    if eqBorders[0][1] != None and eqBorders[1][1] !=
        None:
        with open(tempFastaFiles[0], 'w') as border1fasta
            :
            border1fasta.write('>border1\n')
            border1fasta.write(str(eqBorders[0][1]))
        with open(tempFastaFiles[1], 'w') as border2fasta
            :
            border2fasta.write('>border2\n')
            border2fasta.write(str(eqBorders[1][1]))
        blastResults = alignSeqsByBlast(tempFastaFiles
            [0], tempFastaFiles[1], eqBorders[0][0], name)
    else:
        loggerMain.info('No␣sequences␣for␣the␣left␣border
            :␣border␣1␣is␣{}␣and␣border␣2␣is␣{}'.format(
            len(eqBorders[0][1]), len(eqBorders[0][1])))
        blastResults = {'Identity': -1, 'Gaps': 0, '
            Matches%': 0, 'AlnLen': 0, 'Mismatches': 0}
```

```
            loggerMain.info('\t{}'.format(blastResults))
            if blastResults['Matches%] > 80 and blastResults['
               AlnLen'] > 0.90:
                if storeResults == True:
                    gap1.addBorderGapRef(gap2, 'right')
                    gap2.addBorderGapRef(gap1, 'right')
                    return True
        return False

def equalizeBorders(border1, border2, typeOfBorder='right'):
        loggerMain.info('running equalizeBorders')
        if (border1[0] + border2[0]) > 0 and border1[0] ==
           border2[0]:
            loggerMain.info('No need to equalize, both sequences
               are the same size. border 1 {}, border 2 {}'
                            ''.format(border1[0], border2[0]))
            return (border1, border2)
        elif (border1[0] + border2[0]) > 0 and typeOfBorder == '
           right':
            loggerMain.info('sequences are not the same size,
               treating them as RIGHT sequences. border 1 {},
               border 2 {}'
                            ''.format(border1[0], border2[0]))

            if border1[0] > border2[0]:
                diff = border1[0] - border2[0]
                border1 = (border1[0] - diff, border1[1][:len(
                   border1[1]) - diff])
                loggerMain.info('Fixed border1: now border1 is {}
                   and selfSize is {}'.format(border1[0],border2
                   [0]))
                return (border1, border2)

            elif border2[0] > border1[0]:
                diff = border2[0] - border1[0]
                border2 = (border2[0] - diff, border2[1][:len(
                   border2[1]) - diff])
                loggerMain.info('Fixed border2: now spacerSize is
                   {} and selfSize is {}'.format(border1[0],
                   border2[0]))
                return (border1, border2)
        else:
```

```
        loggerMain.info('Both␣sequences␣have␣no␣length')

def performProdigalTraining(list):
    createCombinedFasta(list)
    subprocess.call([prodigalPath+'prodigal', '-p', 'train',
       '-i', 'combinedFasta.temp', '-t', 'training.tmp.trn'])
    os.remove('combinedFasta.temp')
    return 'training.tmp.trn'

def getORFs(gapCluster, index):
    seqDict = {}
    for gap in gapCluster:
        name = 'gapCluster_{}_{}'.format(index, gap.parents
           [0])
        seq = gap.getParentSeq()[gap.parentPos[0]:gap.
           parentPos[1]]
        seqDict[name] = seq
    with open('combinedGapFastas.temp.fasta', 'w') as output:
        for key in seqDict:
            output.write('>{}\n'.format(key))
            output.write('{}\n'.format(seqDict[key]))

    subprocess.call([prodigalPath+'prodigal', '-i', '
       combinedGapFastas.temp.fasta', '-p', 'single', '-t',
                     'training.tmp.trn', '-f', 'sco', '-o', '
                        outputGaps_{}.sco'.format(index)])
    orfDict = parseScoFile('outputGaps_{}.sco'.format(index))
    os.remove('outputGaps_{}.sco'.format(index))
    return orfDict

def parseScoFile(scoFile):
    scoDict = {}
    with open(scoFile, 'r') as scoInput:
        newKey = None
        while True:
            line = scoInput.readline()
            if not line:
                break
            if line[0:3] == '#␣S':
                newKey = line.split(';')[-1].split('"')[-2].
                   split('_')[-1]
                scoDict[newKey] = []
```

```
            elif line[0] == '>':
                splitLine = line.split('_')
                geneInfo = (splitLine[1], splitLine[2],
                    splitLine[3].rstrip('\n'))
                scoDict[newKey].append(geneInfo)
    return scoDict

def checkForProblemType2(masterFamilyList, actualLogger):
    size = 5000
    toRemoveGaps = []
    newGaps = []
    for family in masterFamilyList:
        gapsToRemoveList = []
        newGapsFamilyList = []
        for gap in family.gapList:
            actualLogger.info('comparing␣left␣border')
            orphanList = [x[0] for x in gap.getOrphanList()
                if x[1] == 'left']
            closedGapList = []
            for leftGap in orphanList:
                if (gap.gapLength) > (leftGap.gapLength):
                    longGap = gap
                    longDtce = gap.gapLength
                    targetGap = leftGap
                    targetDtce = leftGap.gapLength
                else:
                    longGap = leftGap
                    longDtce = leftGap.gapLength
                    targetGap = gap
                    targetDtce = gap.gapLength
                familyName = './' + gap.family.parents[0] + '
                    ,' + leftGap.family.parents[0]
                results = iterativeCloseGaps(targetGap,
                    longGap, targetDtce, longDtce, size,
                    actualLogger,
                                             name = familyName)
                if results is not None:
                    closedGapList.append(results)
            for res in closedGapList:
                gapsToRemoveList += res[0]
                newGapsFamilyList.append(res[1])
```

```
        toRemoveGaps += gapsToRemoveList
        for gap1 in newGapsFamilyList:
            equal = False
            for gap2 in newGaps:
                if set(gap1.parentPos) == set(gap2.parentPos)
                    and set(gap1.corePos) == set(gap2.corePos
                   ) and \
                        set(gap1.family) == set(gap2.family):
                    equal = True
                    break
            if equal == False:
                newGaps.append(gap1)

    toRemoveGaps.sort(key=lambda x: x.name)
    cleanGapsToRemove = []
    for i in range(0, len(toRemoveGaps) - 1):
        if i < len(toRemoveGaps) - 2:
            results = toRemoveGaps[i].name == toRemoveGaps[i
               + 1].name
            if results is False:
                cleanGapsToRemove.append(toRemoveGaps[i])
        else:
            if toRemoveGaps[i] not in cleanGapsToRemove:
                cleanGapsToRemove.append(toRemoveGaps[i])
            if toRemoveGaps[i + 1] not in cleanGapsToRemove:
                cleanGapsToRemove.append(toRemoveGaps[i + 1])

    newGaps.sort(key=lambda x: x.name)
    actualLogger.info('purging␣gaps:␣gaps␣in␣
       cleanGapsToRemove:␣{}'.format(len(cleanGapsToRemove)))
    for family in masterFamilyList:
        actualLogger.info('checking␣family␣{}'.format(family.
           parents))
        removeList = []
        for i, gap in enumerate(family.gapList):
            if gap in cleanGapsToRemove:
                removeList.append(gap)
                actualLogger.info('\t{}\tgap␣{}␣marked␣for␣
                   removal'.format(i, gap.name))
            else:
```

```
                actualLogger.info('\t{}\tgap not in
                    removalList, check matchLists for mentions
                    '.format(i))
                removalLeft = []
                for matchGap in gap.leftMatch:
                    if matchGap in cleanGapsToRemove:
                        removalLeft.append(matchGap)
                for markedGap in removalLeft:
                    gap.leftMatch.remove(markedGap)

                removalRight = []
                for matchGap in gap.rightMatch:
                    if matchGap in cleanGapsToRemove:
                        removalRight.append(matchGap)

                for markedGap in removalRight:
                    gap.rightMatch.remove(markedGap)
        for markedGap in removeList:
            family.gapList.remove(markedGap)

gapsByName = []
for gap in newGaps:
    gap.family.gapList.append(gap)
    gapsByName.append(gap.name)

# Compare the new gaps to get new relations
actualLogger.info('\n\nASSINGING BORDERS TO THE NEW GAPS\
    n\n')
dangerousCombinations = []
for family in masterFamilyList:
    combList = iter([(x, y) for x in newGaps for y in
        family.getGapsByType('analyze')])
    if not os.path.exists('./temp/'):
        os.mkdir('./temp/')

    for gapPair in combList:
        #There is the case in which two gaps are written
            from different families if both of them are in
            newSeqs
        # In order to prevent duplicity is the second and
            third condition. We'll let only one of those
            combinations
```

```
            # happen and the others will be discarted.
            if gapPair[0].family != gapPair[1].family and (
               gapPair[0].name, gapPair[1].name) not in \
                     dangerousCombinations and (gapPair[1].
                        name, gapPair[0].name) not in
                        dangerousCombinations:
                if gapPair[0].name in gapsByName and gapPair
                   [1].name in gapsByName:
                    dangerousCombinations.append((gapPair[0].
                       name, gapPair[1].name))
                actualLogger.info('comparing␣gap␣pair␣{}␣-␣{}
                   '.format(gapPair[0].name, gapPair[1].name)
                   )
                familyName = './' + gapPair[0].family.parents
                   [0] + ',' + gapPair[1].family.parents[0]
                compareTwoGaps(gapPair[0], gapPair[1], size,
                   actualLogger, name=familyName)

def iterativeCloseGaps(targetGap, longGap, targetDtce,
   longDtce, size, actualLogger, name):
    targetGapList = [targetGap]
    iterativeTargetGap = targetGap
    iterativeDtce = targetDtce
    gapThreshold = 1000
    longDtce += gapThreshold

    while True:
        actualLogger.info('\tstarting␣iteration:␣
           iterativeDtce␣is␣{},␣longDtce␣is␣{},␣gapThreshold␣
           is␣{}'.format(
```

```
        adjIterativeTargetGap = iterativeTargetGap.family.
           findAdjacentGaps(iterativeTargetGap)[1]
        if adjIterativeTargetGap is None:
```

```
            actualLogger.info('\t No more gaps to the right, 
                breaking loop')
            break
        actualLogger.info('\tadjIterativeTargetGap is {}, 
            with parentPos of {}'.format(adjIterativeTargetGap
            .name,
                                                                adj

        actualLogger.info('\titerative dtce is {} + {}, 
            longDtce + gapThreshold is ({} + {})'.format(
            iterativeDtce,
                                          adjIterativeTargetGap
                                              .gapLength,
                                          longDtce -
                                          gapThreshold,
                                          gapThreshold))
        iterativeDtce += adjIterativeTargetGap.gapLength

        if iterativeDtce > longDtce:
            actualLogger.info('\t iterative dtce > longDtce, 
                try merging method')
            regResult = regularCloseGaps(targetGap, longGap, 
                targetDtce, longDtce, size, actualLogger, name
                )
            if regResult is not None:
                return regResult
            else:
                actualLogger.info('regular method did not 
                    provide any results')
                return None

        # Get borders
        longBorder = longGap.getBorderSequence(size)[1]
        adjBorder = adjIterativeTargetGap.getBorderSequence(
            size)[1]

        eqBorders = equalizeBorders(longBorder, adjBorder)
```

```
actualLogger.info('\teqBorders len: {}, {}'.format(
    len(eqBorders[0][1]), len(eqBorders[1][1])))

# then run blast:
tempFastaFiles = [name + 'border1Adj.temp.fasta',
    name + 'border2Adj.temp.fasta']

actualLogger.info('comparing right border')
if eqBorders[0][1] != None and eqBorders[1][1] !=
    None:
     with open(tempFastaFiles[0], 'w') as border1fasta
         :
          border1fasta.write('>border1\n')
          border1fasta.write(str(eqBorders[0][1]))
     with open(tempFastaFiles[1], 'w') as border2fasta
         :
          border2fasta.write('>gap2left\n')
          border2fasta.write(str(eqBorders[1][1]))
     blastResults = alignSeqsByBlast(tempFastaFiles
         [0], tempFastaFiles[1], eqBorders[0][0], name)
else:
     actualLogger.info('No sequences for the left
         border: border 1 is {} and border 2 is {}'.
         format(
          len(eqBorders[0][1]), len(eqBorders[0][1])))
     blastResults = {'Identity': -1, 'Gaps': 0, '
         Matches%': 0, 'AlnLen': 0, 'Mismatches': 0}

# analyzeBlastResults
if blastResults['Matches%'] > 80 and blastResults['
    AlnLen'] > 0.90:
     # create a new gap object and return it + the
         gaps used to create it
     newGap = Gap([targetGap.parents[0], targetGap.
         parentPos[0], adjIterativeTargetGap.parentPos
         [1], 'None',
           targetGap.parents[1], targetGap.corePos[0],
               adjIterativeTargetGap.corePos[1], 'None',
           'analyze'], targetGap.family)

     newNo = int(round((float(targetGap.name.split('_'
         )[-1]) +
```

```
                                    float(adjIterativeTargetGap.
                                        name.split('_')[-1])) / 2,
                                        0))
                newGap.name = str(str(newGap.parents[0]) + '-' +
                    str(newGap.parents[1]) + '_' + str(newNo))
                targetGapList.append(adjIterativeTargetGap)
                return (targetGapList, newGap)

            else:
                targetGapList.append(adjIterativeTargetGap)
                iterativeTargetGap = adjIterativeTargetGap
                actualLogger.info('\t new iterativeGap is {}'.
                    format(adjIterativeTargetGap.name))

def regularCloseGaps(targetGap, longGap, targetDtce, longDtce
    , size, actualLogger, name):
    actualLogger.info('\tAttempting to close gap via merging'
        )
    longBorder = longGap.getBorderSequence(size)[1]
    actualLogger.info('\tlong dtce:{}, targetDtce:{}, size:{}
        '.format(longDtce, targetDtce, size))
    targetBorder = targetGap.getBorderSequence(size + (
        longDtce - targetDtce))[1]
    actualLogger.info('\tlong border: {}\n\ttargetBorder: {},
        diffDtce: {}'.format(longBorder, targetBorder,
```

```
    if targetBorder[0] < (longDtce - targetDtce):
        actualLogger.info('\tNot enough space for the check')
        return None

    else:
```

```
targetBorder = [targetBorder[0] - (longDtce -
    targetDtce), targetBorder[1][(longDtce -
    targetDtce):]]
eqBorders = equalizeBorders(longBorder, targetBorder)
# then run blast:
tempFastaFiles = [name + 'border1Reg.temp.fasta',
    name + 'border2Reg.temp.fasta']

actualLogger.info('comparing right border')
if eqBorders[0][1] != None and eqBorders[1][1] !=
    None:
     with open(tempFastaFiles[0], 'w') as border1fasta
         :
          border1fasta.write('>border1\n')
          border1fasta.write(str(eqBorders[0][1]))
     with open(tempFastaFiles[1], 'w') as border2fasta
         :
          border2fasta.write('>gap2left\n')
          border2fasta.write(str(eqBorders[1][1]))
     blastResults = alignSeqsByBlast(tempFastaFiles
         [0], tempFastaFiles[1], eqBorders[0][0], name)
else:
     actualLogger.info('No sequences for the left
         border: border 1 is {} and border 2 is {}'.
         format(
          len(eqBorders[0][1]), len(eqBorders[0][1])))
     blastResults = {'Identity': -1, 'Gaps': 0, '
         Matches%': 0, 'AlnLen': 0, 'Mismatches': 0}

# analyzeBlastResults
if blastResults['Matches%'] > 80 and blastResults['
    AlnLen'] > 0.90:
     newGap = Gap([targetGap.parents[0], targetGap.
         parentPos[0], targetGap.parentPos[1] + (
         longDtce -
               targetDtce), 'None', targetGap.parents
                  [1], targetGap.corePos[0], targetGap.
                  corePos[1] + (longDtce
              - targetDtce), 'None', 'analyze'],
                  targetGap.family)
     newGap.name = newGap.name.split('_')[0] + '_' +
         targetGap.name.split('_')[-1] + 'M'
```

```
            return ([targetGap], newGap)
        else:
            return None

def worker(tuple):
    family1 = tuple[0]
    family2 = tuple[1]
    size = 5000
    #Create one log for each combination and pass the log's
       reference at the function's call
    setup_logger('herring_process', 'herring.{}.{}.log'.
       format(family1.parents[0], family2.parents[0]))
    actualLog = logging.getLogger('herring_process')
    actualLog.info("Now working on families {} and {}".format
       (family1.parents[0], family2.parents[0]))
    data = compareGapFamilies(family1, family2, size,
       actualLog)
    #data is a list of tuples, all combinations of gaps from
       the two families of the thread
    return data


if __name__ == '__main__':

    timeStart = time.time()
    #Magic number powered by last author
    thresholdAnalysis = 2000
    blastFamilies = performBlastAgainstCore(fastaList,
       refFasta, 500, thresholdAnalysis, masterFamilyList)
    os.remove(os.curdir + '/blastSeqs.blastn')
    for family in masterFamilyList:
        family.equalize()

    #go back to old log here
    setup_logger('herringMain', 'herring.main.log')
    loggerMain = logging.getLogger('herringMain')
    #Save the gap file, just in case something breaks so we
       don't have to do the analysis all over again

    #Perform comparisons
    familyCombList = itertools.combinations(masterFamilyList,
        2)
```

```
#initialize the thread pool
loggerMain.info('Now␣launching␣the␣threads.␣Number␣of␣
    threads␣desired␣by␣user:␣{}'.format(cores))
p=multiprocessing.Pool(cores)
data = p.map(worker, familyCombList)
#Wait until the threads have finished their work
p.close()
p.join()
loggerMain.info('Threads␣have␣already␣finished␣(all␣of␣
    them,␣program␣is␣supposed␣to␣wait␣for␣them)')

loggerMain.info('Now␣assigning␣each␣tuple␣element␣from␣
    the␣threads␣to␣corresponding␣family')
for familyList in data:
    #Each element is a list of tuples which represents
        all the gaps between the two families
    for gapFound in familyList:
        #Each tuple is the data for the gaps. Its
            structure is (gap1, gap2, border)
        #Border is a list of strings. It can be empty,
            figure 'left', 'right' or both (but not the
            word 'both')
        gap1 = gapFound[0]
        gap2 = gapFound[1]
        border = gapFound[2]
        family1Name = gap1.family.parents
        family2Name = gap2.family.parents
        classFamily1 = None
        classFamily2 = None

        #Look for the gap's family
        for familyRoot in masterFamilyList:
            if classFamily1 == None and familyRoot.
                parents == family1Name:
                 classFamily1 = familyRoot
            if classFamily2 == None and familyRoot.
                parents == family2Name:
                 classFamily2 = familyRoot
            if classFamily1 != None and classFamily2 !=
                None:
                 break
```

```
#Now look for the gap whose name is the same as
   the gap's thread
centralGap1 = None
centralGap2 = None
for candidateGap in classFamily1.gapList:
    if candidateGap.name == gap1.name:
        centralGap1 = candidateGap
        break
if centralGap1 == None:
    centralGap1 = gap1
    classFamily1.gapList.append(centralGap1)

for candidateGap in classFamily2.gapList:
    if candidateGap.name == gap2.name:
        centralGap2 = candidateGap
        break
if centralGap2 == None:
    centralGap2 = gap2
    classFamily2.gapList.append(centralGap2)

#Now we know to which gap corresponds each gap
   found on the threads
#It's time to rebuild the border references
if 'left' in border:
    loggerMain.info('Left␣border␣found␣between␣{}
       ␣and␣{}'.format(centralGap1.name,
       centralGap2.name))
    if centralGap2 not in centralGap1.leftMatch:
        centralGap1.addBorderGapRef(centralGap2,
            'left')
    if centralGap1 not in centralGap2.leftMatch:
        centralGap2.addBorderGapRef(centralGap1,
            'left')
if 'right' in border:
    loggerMain.info('Right␣border␣found␣between␣
       {}␣and␣{}'.format(centralGap1.name,
       centralGap2.name))
    if centralGap2 not in centralGap1.rightMatch:
        centralGap1.addBorderGapRef(centralGap2,
            'right')
    if centralGap1 not in centralGap2.rightMatch:
```

```
                    centralGap2.addBorderGapRef(centralGap1,
                        'right')

#To whoever cares: Code should never go into this loop,
    but no one knows for sure, so it will remain in order
    to
# add some safety to the program. Its function is to
    remove all the gaps whose have no border gaps neither
    left
# or right. Remove at your own risk
for familyRoot in masterFamilyList:
    i = 0
    while i < len(familyRoot.gapList):
        gap  = familyRoot.gapList[i]
        if len(gap.leftMatch) + len(gap.rightMatch) == 0:
            familyRoot.gapList.remove(gap)
        else:
            i += 1

saveHerringData('masterListTest.pk1')
#loadedStuff = loadHerringData('masterListTest.pk1')
#masterFamilyList = loadedStuff[0]
#fileList = loadedStuff[1]

checkForProblemType2(masterFamilyList, loggerMain)

loggerMain.info('Final␣steps.␣Cleaning␣files␣created␣by␣
    the␣program')
tempFastaFiles = ['*gap1left.temp.fasta', '*gap1right.
    temp.fasta', '*gap2left.temp.fasta', '*gap2right.temp.
    fasta',
            '*border1Reg.temp.fasta', '*border2Reg.temp.
                fasta', '*border1Adj.temp.fasta', '*
                border2Adj.temp.fasta']
for file in os.listdir('.'):
    for pattern in tempFastaFiles:
        if fnmatch.fnmatch(file, pattern):
            os.remove(file)

for file in os.listdir('./temp'):
    for pattern in tempFastaFiles:
        if fnmatch.fnmatch(file, pattern):
```

```
                os.remove(file)

    loggerMain.info('Storing all the info into two tipes of
        files')
    for family in masterFamilyList:
        family.writeGapInfo(family.findOrphans(type = ('left'
            , 'right')), 'diagnostic-All' + '-' +
                            str(family.parents[0]) + '.txt')
        family.writeGapInfo([item for item in family.gapList
            if item in family.findOrphans(type=('both'))],
                            'diagnostic-Good' + '-' + str(
                                family.parents[0]) + '.txt')

    #gapClusterList = checkForGapClusters(masterFamilyList)
    #dumpClusterData(gapClusterList)

    timeFinish = time.time()
    duration = timeFinish - timeStart
    if duration < 60:
        print('This program needed {} seconds'.format(
            duration))
    if duration > 60:
        print('This program needed {} hour(s) and {} minute(s
            ) aprox'.format(duration//3600, round((duration
            %3600)/60)))
    loggerMain.info('This program needed {} hour(s), {}
        minute(s) and {} second(s)'.format(duration//3600,
                                                    (
                                                        duration
                                                        %3600)
                                                        //60,

                                                        round
                                                        (((
                                                        duration
                                                        %3600)
                                                        %60)
                                                        /60)
                                                        )
                                                        )
```

A.4. concatenateSeqs.py

```
from Bio import SeqIO, SeqRecord, Alphabet
from subprocess import call
from collections import Counter
import os, sys, argparse, logging

#Argparser stuff
argParser = argparse.ArgumentParser(description='
    concatenateSeqs␣joins␣a␣collection␣of␣contigs␣together␣
    using␣a␣'
                                    'complete␣sequence␣as␣
                                       reference.␣Requires␣
                                       blast␣2.5+␣and␣the␣
                                       Biopython␣library.\n'
                                    'WARNING:␣The␣algorithm␣
                                       used␣for␣the␣assembly␣
                                       is␣not␣perfect␣-␣if␣
                                       the␣output␣looks␣'
                                            'wrong,␣it\'s
                                               ␣probably␣
                                               the␣
                                               assembler
                                               \'s␣fault.
                                               ')
argParser.add_argument('-i', '--input', nargs = 1, required =
     True, type=str,
                       help='fasta␣file␣containing␣the␣
                          contigs␣to␣be␣merged')
argParser.add_argument('-r', '--reference', nargs= 1,
    required = True, type=str,
                       help='fasta␣file␣containing␣the␣
                          reference␣genome')
argParser.add_argument('-o', '--output', nargs=1, required =
    True, type=str,
                       help='prefix␣for␣the␣output␣files')
argParser.add_argument('--join_threshold', nargs='?', default
    =5000, type=int,
                       help='maximum␣distance␣between␣contigs
                          ␣to␣be␣joined␣together␣(default:␣
                          5000).␣A␣negative␣value␣'
```

```
                                'will␣join␣all␣contigs␣in␣order␣(
                                  same␣result␣for␣all␣negative␣
                                  values)')
argParser.add_argument('--cut_threshold', nargs='?', default
    =0.99, type=float,
                        help= 'The␣script␣will␣attempt␣to␣cut␣
                            contigs␣in␣order␣to␣align␣them␣to␣
                            the␣reference.␣Value␣'
                                'given␣will␣mark␣the␣sequence␣
                                  percentage␣that␣a␣gap␣needs␣
                                  in␣order␣to␣be␣cut.␣If␣value␣
                                  is'
                                'zero␣program␣wont␣cut.␣Values␣
                                  between␣0-1␣(Default:␣0.99)')
argParser.add_argument('--outfmt', nargs='?', default='fasta'
    , type=str,
                        help='output␣format,␣either␣\'genbank
                            \'␣or␣\'fasta\'␣(Default:␣fasta)')


#Initialize variables
#blastpath = '/opt/ncbi-blast-2.6.0+/bin/'
blastpath = '/home/rohit/alberto/ncbi-blast-2.7.1+/bin/'
ref_fasta = argParser.parse_args(sys.argv[1:]).reference[0]
contig_fasta = argParser.parse_args(sys.argv[1:]).input[0]
output_prefix = argParser.parse_args(sys.argv[1:]).output[0]
join_threshold = argParser.parse_args(sys.argv[1:]).
    join_threshold
cut_threshold = argParser.parse_args(sys.argv[1:]).
    cut_threshold
output_type = argParser.parse_args(sys.argv[1:]).outfmt

#Initalize log
def setup_logger(logger_name, log_file, level=logging.DEBUG):
    l = logging.getLogger(logger_name)
    formatter = logging.Formatter('%(message)s')
    file_handler = logging.FileHandler(log_file, mode='a')
    file_handler.setFormatter(formatter)
    l.setLevel(level)
    l.addHandler(file_handler)
```

```
setup_logger('concatenate', 'Concatenate_7_{}.log'.format(
    output_prefix))
logger = logging.getLogger('concatenate')

logger.info('Parameters␣passed␣to␣script:␣ref_fasta:␣{},␣
    contig_fasta:␣{},␣output_prefix:␣{},␣join_threshold:␣{},␣'
            'cut_threshold:␣{},␣output_type:␣{}'.format(
                ref_fasta, contig_fasta, output_prefix,
                join_threshold,
                                                    cut_threshold
                                                        ,
                                                    output_type
                                                    ))


contig_list = {}
removed_contig_list = []
ref_list = []
similarity_threshold = None


def assemble_contigs(contig_list, reference, join_threshold):
    sol = []
    raw_list = []
    for contig in contig_list:
        contig.set_borders(reference)
        if contig.ref_start is None:
            sol.append(contig)
        else:
            raw_list.append(contig)
    contigs_by_start = sorted(raw_list, key=lambda contig:
       contig.ref_start)
    contigs_by_end = sorted(raw_list, key=lambda contig:
       contig.ref_end)
    if join_threshold < 0:
        logger.info('\tnegative␣join_threshold.␣Must␣join␣all
           ␣the␣contigs')
    while len(contigs_by_end) > 0:
        actual_contig = contigs_by_end[0]
        name = actual_contig.id
        logger.info('\tLooking␣for␣contigs␣near␣{}'.format(
           name))
        sequence = actual_contig.seq
```

```
start = actual_contig.ref_start
end = actual_contig.ref_end
dict = actual_contig.blasts_dict
contigs_by_start.remove(actual_contig)
used_contigs = []
j=1
for contig in contigs_by_start:
    if actual_contig.ref_seq == contig.ref_seq and (
       join_threshold < 0 or
                                                    end
                                                        +
                                                        join_threshold
                                                        >
                                                        contig
                                                        .
                                                        ref_start
                                                        )
                                                        :

        name += contig.id
        sequence += contig.seq
        start = min(start, contig.ref_start)
        end = max(end, contig.ref_end)
        dict[reference] += contig.blasts_dict[
           reference]
        used_contigs.append(contig)
        if join_threshold >= 0:
            logger.info('contig␣{}␣is␣near␣enough:␣
               end␣+␣join_threshold␣>␣contig.
               ref_start␣-->␣{}␣+␣{}␣=␣{}␣>␣'
                        '{}'.format(contig.id, end,
                           join_threshold, end +
                           join_threshold, contig.
                           ref_start))
        j+=1
    else:
        break
for elem in used_contigs:
```

```
                contigs_by_start.remove(elem)
                contigs_by_end.remove(elem)
            new_contig = Sequence(SeqRecord.SeqRecord(sequence,
                id=name), ref_seq=actual_contig.ref_seq)
            new_contig.ref_start = start
            new_contig.ref_end = end
            new_contig.blasts_dict = dict
            sol.append(new_contig)
            contigs_by_end.remove(actual_contig)
            if j == 1:
                logger.info('No contigs near enough to join')
            else:
                logger.info('Search finished. {} contigs have
                    been joined\n'.format(j))
        logger.info('Finish assembling contigs with reference {}\
            n'.format(reference))
        return sol


class Sequence:
    def __init__(self, record, ref_seq = None):
        self.ref_seq = ref_seq
        self.id = record.id
        self.seq = record.seq
        self.seq.description = 'None'
        self.seq.alphabet = Alphabet.generic_dna
        self.blasts_dict = {}
        self.ref_start = None
        self.ref_end = None
        self.strand = None
        self.length = len(self.seq)
        self.removed_reason = None

    def add_blast_hit(self, blast_hit, ref_seq):
        if ref_seq.id not in self.blasts_dict.keys():
            self.blasts_dict[ref_seq.id] = [blast_hit]
        else:
            self.blasts_dict[ref_seq.id].append(blast_hit)

    def choose_ref_seq(self):
        logger.info('starting choose_ref_seq for contig {}'.
            format(self.id))
```

```
        #If there is only blast for one ref sequence, then
           there is no need for processing
        if len(self.blasts_dict.keys()) < 1:
            logger.info('{} got no references, so it will be
               discarded'.format(self.id))
            self.removed_reason = 'No blast_hits assigned'
            self.ref_seq = 'None'
            removed_contig_list.append(self)
        elif len(self.blasts_dict.keys()) == 1:
            for key in self.blasts_dict.keys():
                logger.info('only one reference, assigning {}
                   as ref_seq'.format(key))
                self.ref_seq = key

    def cut_sequence(self, ref_seq):
        logger.info('cutting sequence {}'.format(self.id))
        raw_blast_list = self.blasts_dict[ref_seq.id]
        blast_list = []
        for blast in raw_blast_list:
            if blast.similarity > similarity_threshold:
                blast_list.append(blast)
        blast_list.sort(key = lambda match:match.ref_start)
        max_diff = ref_seq.length * 0.01
        max_diff_cand = None
        for i in range(0, len(blast_list)-1):
            actual_blast = blast_list[i]
            next_blast = blast_list[i + 1]
            start_next_blast = next_blast.ref_start
            end_actual_blast = actual_blast.ref_end
            diff = abs(start_next_blast - end_actual_blast)
            if diff > max_diff + ref_seq.length * 0.01:
                max_diff = diff
                max_diff_cand = i
        if max_diff_cand is None:
            return None
        #A gap has been found
        new_dict = {}
        new_dict[ref_seq.id] = []
        cut_dtce = 0
        for j in range(0, max_diff_cand+1):
            if cut_dtce < blast_list[j].query_end < self.
               length:
```

```
                cut_dtce = blast_list[j].query_end
            new_dict[ref_seq.id].append(blast_list[j])
        if cut_dtce == 0:
            return [self, None]
        new_seq_1 = Sequence(SeqRecord.SeqRecord(self.seq[:
            cut_dtce], id=self.id + '_1'), ref_seq=self.
            ref_seq)
        new_seq_2 = Sequence(SeqRecord.SeqRecord(self.seq[
            cut_dtce:], id=self.id + '_2'), ref_seq=self.
            ref_seq)
        new_seq_1.ref_start = blast_list[0].ref_start
        new_seq_1.ref_end = blast_list[max_diff_cand].ref_end
        new_seq_1.blasts_dict = new_dict

        other_dict = {}
        other_dict[ref_seq.id] = []
        new_seq_2.ref_start = blast_list[max_diff_cand + 1].
            ref_start
        for j in range(max_diff_cand + 1, len(blast_list)):
            other_dict[ref_seq.id].append(blast_list[j])
        new_seq_2.ref_end = blast_list[-1].ref_end
        new_seq_2.blasts_dict = other_dict
        logger.info('2 new sequences: {} ({} - {}) w/ length
            {} and {} ({} - {}) w/ length {}'.format(
            new_seq_1.id, new_seq_1.ref_start, new_seq_1.
                ref_end, new_seq_1.length,
            new_seq_2.id, new_seq_2.ref_start, new_seq_2.
                ref_end, new_seq_2.length))
        return [new_seq_1, new_seq_2]

    def is_duplicated(self, final_dict):
        for key in final_dict:
            for contig in final_dict[key]:
                if self.seq in contig.seq:
                    logger.info('{} seq is included in {} seq
                        , so will be discarded'.format(self.id
                        , contig.id))
                    self.removed_reason = 'Duplicated contig'
                    removed_contig_list.append(self)
                    return True
        logger.info('Contig {} seems fine. No need to delete'
            .format(self.id))
```

```
        return False

    def place_contig_on_reference(self, ref_seq):
        min_start = self.ref_start
        max_end = self.ref_end
        if min_start is not None and max_end is not None:
            if max_end - min_start > ref_seq.length *
               cut_threshold:
                logger.warning('{} is considered for cutting:
                   max_end - min_start > ref_seq.length *
                   cut_threshold -->'
                               '{} - {} = {} > {} = {} * {}'.
                                  format(self.id, max_end,
                                  min_start, max_end -
                                  min_start,
                                                  ref_seq.
                                                     length
                                                      *
                                                     cut_threshold
                                                     ,
                                                     ref_seq
                                                     .
                                                     length
                                                     ,
                                                     cut_threshold
                                                     ))
                return 'CUT'
            else:
                self.ref_start = min_start
                self.ref_end = max_end
                logger.info('{} has found a good match in {}:
                   both max_end and min_start are not None
                   but max_end - '
                            'min_start < ref_seq.length *
                               cut_threshold --> {} - {} = {}
                               < {} = {} * {}'.format(self.
                               id,
```

```
                                                                ref_seq
                                                                    .
                                                                    id
                                                                    ,
                                                                    max_end
                                                                    ,
                                                                    min_start
                                                                    ,
                                                                    max_end
                                                                     -

                                                                    min_start
                                                                    ,
                                                                ref_seq
                                                                    .
                                                                    length
                                                                     *

                                                                    cut_threshold
                                                                    ,
                                                                    ref_seq
                                                                    .
                                                                    length
                                                                    ,
                                                                    cut_threshold
                                                                    ))
                    return 'OK'
            else:
                logger.warning('{} has no operating size, so it
                    will be discarted (max_end and min_start are
                    None)'.format(self.id))
                return 'NO'

    def reverse(self, key):
        blast_list = sorted(self.blasts_dict[key], key=lambda
            match: match.length, reverse=True)
        if blast_list[0].strand == 1:
            self.seq = self.seq.reverse_complement()
            logger.info('contig {} has been reversed'.format(
                self.id))

    def set_borders(self, key):
```

```
        #This function is an extension from
           get_operating_size
        min_start = None
        max_end = None
        for match in self.blasts_dict[key]:
            if len(self.blasts_dict[key]) == 1:
                min_start = match.ref_start
                max_end = match.ref_end
            elif match.similarity > similarity_threshold and
               match.length > 500:
                min_candidate = match.ref_start
                max_candidate = match.ref_end
                if min_start is None or min_start >
                   min_candidate:
                    min_start = min_candidate
                if max_end is None or max_end < max_candidate
                   :
                    max_end = max_candidate
            elif join_threshold < 0:
                min_candidate = match.ref_start
                max_candidate = match.ref_end
                if min_start is None or min_start >
                   min_candidate:
                    min_start = min_candidate
                if max_end is None or max_end < max_candidate
                   :
                    max_end = max_candidate
        self.ref_start = min_start
        self.ref_end = max_end

    def split_to_choose(self):
        logger.info('\tsplit_to_choose:␣now␣evaluating␣contig
           ␣{}'.format(self.id))
        sorted_full_list = []
        for key in self.blasts_dict.keys():
            sorted_list = sorted(self.blasts_dict[key], key=
               lambda match: match.query_start)
            for match in sorted_list:
                if match.similarity > similarity_threshold
                   and match.length > 500:
                    sorted_full_list.append(match)
```

```
# Cases where there is 0 blasts need to go to
    choose_ref_seq() so them will be discarted
if len(sorted_full_list) == 0:
    logger.info('{} got no useful blasts, so it will
        be discarded --> No blasts with similarity
        above {} and '
                'length above 500 bp\n'.format(self.
                    id, similarity_threshold))
    self.removed_reason = 'No blast_hits assigned'
    self.ref_seq = 'None'
    removed_contig_list.append(self)
    return []
i = 0
j = 0
sol = []
while i < len(sorted_full_list):
    actual_blast = sorted_full_list[i]
    key = actual_blast.parent_seq
    if i == 0:
        query_starting_pos = 0
    else:
        query_starting_pos = actual_blast.query_start
    if i == len(sorted_full_list)-1:
        query_ending_pos = self.length
    else:
        query_ending_pos = sorted_full_list[i+1].
            query_start
    reference_starting_pos = actual_blast.ref_start
    reference_ending_pos = actual_blast.ref_end
    dict = {}
    dict[key] = [actual_blast]
    new_seq = Sequence(SeqRecord.SeqRecord(self.seq[
        query_starting_pos:query_ending_pos],
                                           id=self.id
                                               + '_{}
                                               '.
                                               format(
                                               j)),
                                               ref_seq
                                               =self.
                                               ref_seq
                                               )
```

```
            j += 1
            new_seq.ref_start = reference_starting_pos
            new_seq.ref_end = reference_ending_pos
            new_seq.blasts_dict = dict
            i += 1
            if new_seq.length == 0:
                new_seq.removed_reason = 'No real contig (
                    sequence length = 0)'
                new_seq.ref_seq = 'None'
                removed_contig_list.append(new_seq)
                logger.info('contig {} removed because its
                    sequence length was 0'.format(new_seq.id))
            else:
                new_seq.choose_ref_seq()
                sol.append(new_seq)
        logger.info('contig {} has been split into {} new
            contigs\n'.format(self.id, j))
        return sol

    def write_fasta(self):
        with open('ref_seq_temp.fasta', 'w') as fasta_file:
            fasta_file.write('>' + self.id + '\n')
            fasta_file.write(str(self.seq))


class BlastHit:
    def __init__(self, parent, length, similarity,
        ref_start_pos, ref_end_pos, query_start_pos,
        query_end_pos,
                 bit_score):
        self.parent_seq = parent
        self.length = int(length)
        self.similarity = float(similarity)
        self.ref_start = int(ref_start_pos)
        self.ref_end = int(ref_end_pos)
        self.query_start = int(query_start_pos)
        self.query_end = int(query_end_pos)
        self.strand = 0
        self.bit_score = bit_score

        #check the strand of the fragment, fix the start/end
            positions if it is from the complementary strand
```

```
        if self.ref_start > self.ref_end:
            self.strand = 1
            self.ref_start = int(ref_end_pos)
            self.ref_end = int(ref_start_pos)

        #Fix bitscores in scientific notation
        if 'e' in self.bit_score:
            bit_score_split = self.bit_score.split('e')
            if bit_score_split[1][0] == '+':
                self.bit_score = float(bit_score_split[0]) *
                   pow(10, int(bit_score_split[-1]))

            elif bit_score_split[1][0] == '-':
                self.bit_score = float(bit_score_split[0]) *
                   pow(10, (int(bit_score_split[-1])* -1))

        else:
            self.bit_score = float(bit_score)

# Main body. Program starts here

#First step: Load up references and contig sequences
logger.info('loading ref file')
with open(ref_fasta, 'r') as reference_file:
    parsed_references = SeqIO.parse(reference_file, 'fasta')
    for record in parsed_references:
        ref_list.append(Sequence(record))
        logger.debug('\tref sequence {} loaded, with length
           {}'.format(record.id, len(record.seq)))

logger.info('loading fasta file')
with open(contig_fasta, 'r') as contig_file:
    parsed_contigs = SeqIO.parse(contig_file, 'fasta')
    for record in parsed_contigs:
        contig_list[record.id] = (Sequence(record))

#Perform blasts
logger.info('doing blasts')
good_contigs = {}
similarity_list = []
for ref_seq in ref_list:
    good_contigs[ref_seq.id] = []
```

```
    ref_seq.write_fasta()
    call([blastpath + 'makeblastdb', '-in', 'ref_seq_temp.
       fasta', '-out', 'dbTest', '-dbtype', 'nucl'])
    call([blastpath + 'blastn', '-query', contig_fasta, '-db'
       , 'dbTest', '-out', 'blast_results.blastn', '-
       num_threads',
          '4', '-outfmt', '6'])
    with open('blast_results.blastn', 'r') as blast_file:
        for line in blast_file:
            split_line = line.split('\t')
            if split_line[0] in contig_list.keys() and
               split_line[1] == ref_seq.id:
                similarity_list.append(float(split_line[2]))
                contig_list[split_line[0]].add_blast_hit(
                   BlastHit(split_line[1], split_line[3],
                   split_line[2],
                                split_line[8], split_line[9],
                                    split_line[6], split_line
                                   [7], split_line[11]),
                                   ref_seq)
    os.remove('blast_results.blastn')
    os.remove('ref_seq_temp.fasta')
os.remove('dbTest.nsq')
os.remove('dbTest.nin')
os.remove('dbTest.nhr')
res = Counter(similarity_list).most_common(5)
#Look for the first four non-100.0 positions. Notice that
   only one of those can be 100.0 and it should be always at
   top
#five positions, but it wont be always the first one
first = res[1][0]
second = res[2][0]
third = res[3][0]
fourth = res[4][0]
if first == 100.0:
    first = res[0][0]
elif second == 100.0:
    second = res[0][0]
elif third == 100.0:
    third = res[0][0]
elif fourth == 100.0:
    fourth = res[0][0]
```

```
similarity_threshold = (first + second + third + fourth)/4
print('\nStep 1 done: References and contigs loaded. Blasts
    performed\n')

#Step 2: in order to get better matches split the contigs
    into smaller ones based on its blasts. Then assign each
    contig
#to a reference based on some values
logger.info('number of reference sequences: {}'.format(len(
    ref_list)))
logger.info('number of contig sequences: {}'.format(len(
    contig_list.keys())))
logger.info('choosing ref_seqs for each contig')
contig_split_dict = {}
for contig in contig_list:
    list = contig_list[contig].split_to_choose()
    for elem in list:
        #Each elem is a new contig
        contig_split_dict[elem.id] = elem
print('Step 2 done: Contigs split and reference chosen\n')

logger.info('\n\n')
logger.info('choosing ref_seq results:')
#Get some logger data
count_contig_dict = {}
ref_seq = None
for key in contig_split_dict:
    try:
        ref_seq = contig_split_dict[key].ref_seq
    except AttributeError:
        ref_seq = 'None'
    if ref_seq not in count_contig_dict.keys():
        count_contig_dict[ref_seq] = 1
    else:
        count_contig_dict[ref_seq] += 1
for key in count_contig_dict.keys():
    logger.info('{}: {}'.format(key, count_contig_dict[key]))

#Keep only non-removed contigs (obviously)
clean_contig_list = []
for key in contig_split_dict:
    if contig_split_dict[key] not in removed_contig_list:
```

```
        clean_contig_list.append(contig_split_dict[key])

logger.info('\n\n')
logger.info('starting␣place␣contig␣on␣reference')
#Step 3: evaluate each contig into its reference: check whose
    are good, bad and those who need to be cut
#Notice that contigs were cut becore. This time are cut only
   the contigs which start and end positions are on each side
#of the reference
for i, ref in enumerate(ref_list):
    logger.info('starting␣to␣place␣contig␣on␣reference␣{}␣for
       ␣contigs␣assigned␣to␣it'.format(ref.id))
    count = 0
    totalOKs = 0
    for contig in clean_contig_list:
        if contig.ref_seq == ref.id:
            logger.info('\tprocessing␣contig␣{}␣assigned␣to␣
               {}'.format(contig.id, contig.ref_seq))
            count += 1
            check = contig.place_contig_on_reference(ref)
            if check == 'OK':
                totalOKs += 1
                good_contigs[ref.id].append(contig)
            elif check == 'NO':
                contig.removed_reason = 'No␣relevant␣blast␣
                   matches'
                removed_contig_list.append(contig)
            elif check == 'CUT':
                new_seqs = contig.cut_sequence(ref)
                if new_seqs is not None and cut_threshold >
                   0:
                    totalOKs += 1
                    good_contigs[ref.id].append(new_seqs[0])
                    if new_seqs[1] is not None:
                        totalOKs += 1
                        good_contigs[ref.id].append(new_seqs
                           [1])
                        logger.info('Seq1␣{}␣and␣seq2␣{}␣with
                           ␣ref_seq␣{}␣added'.format(new_seqs
                           [0].id, new_seqs[1].id,
```

```
                    else:
                        logger.info('Seq1 {} with ref_seq {}
                            added'.format(new_seqs[0].id, ref.
                            id))
                else:
                    if new_seqs is None:
                        logger.info('{} cannot be cut, so it
                            will be discarded (new seq is None
                            )'.format(contig.id))
                    else:
                        logger.info('cut_threshold is 0 or
                            lower, cant cut {}'.format(contig.
                            id))
                    contig.removed_reason = 'Not a good match
                        - Tried to cut'
                    removed_contig_list.append(contig)

        logger.info('results for {}: {}/{} placed. Total Contigs:
            {}'.format(ref.id, totalOKs, count,
```

le

```
        total_length = 0

        for contig in good_contigs[ref.id]:
            total_length += contig.length
print('Step 3 done: Contigs evaluated for its reference\n')
logger.info('\n\n')

#Step 4: Check the length of the contigs and the length of
    the blasts. If the total blast surface is way smaller than
# the contig lenght discard it. Reverse the good ones
```

```
not_good_enough = []
for ref in ref_list:
    logger.info('Checking␣contigs␣assigned␣to␣reference␣{}'.
       format(ref.id))
    for contig in good_contigs[ref.id]:
        min_pos = None
        max_pos = None
        for blast in contig.blasts_dict[ref.id]:
            if min_pos is None or blast.query_start < min_pos
               :
                min_pos = blast.query_start
            if max_pos is None or blast.query_end > max_pos:
                max_pos = blast.query_end
        blast_length = max_pos - min_pos
        if blast_length/contig.length < 0.1:
            logger.info('{}␣has␣nearly␣no␣blasts␣for␣its␣size
               ,␣so␣it␣will␣be␣discarted:␣blast_length/contig
               .length␣=␣'
                        '{}␣/␣{}␣=␣{}␣<␣0.1'.format(contig.id
                           , blast_length, contig.length,
                           blast_length/contig.length))
            not_good_enough.append((contig, ref.id))
        else:
            logger.info('{}␣is␣a␣good␣contig.␣Check␣strand␣
               for␣reverse␣(if␣needed)'.format(contig.id))
            contig.reverse(ref.id)

for contig, key in not_good_enough:
    contig.removed_reason = 'No␣relevant␣information␣contig'
    removed_contig_list.append(contig)
    good_contigs[key].remove(contig)
print('Step␣4␣done:␣Contig␣sequences␣reversed␣if␣needed\n')
logger.info('\n\n')

logger.info('Safety␣measure:␣checking␣if␣some␣contigs␣are␣
   duplicated')
#Safety measure: check for duplicates in contigs. Usually the
    program wont have duplicates, but one never knows for
   sure
final_dict = {}
for key in good_contigs:
    final_dict[key] = []
```

```
    contig_list = sorted(good_contigs[key], key=lambda contig
        :contig.length, reverse=True)
    for contig in contig_list:
        if not contig.is_duplicated(final_dict):
            final_dict[key].append(contig)

logger.info('\n\n')
#Step 5: Almost finish. Try to assemble contigs based on the
    threshold given to the program
assembled_contigs = {}
for ref in ref_list:
    logger.info('start␣assembling␣contigs␣for␣{},␣minimum␣
        distance␣to␣join:␣{}␣bp'.format(ref.id, join_threshold
        ))
    assembled_contigs[ref.id] = assemble_contigs(final_dict[
        ref.id], ref.id, join_threshold)

for ref in ref_list:
    if ref.id not in assembled_contigs.keys():
        assembled_contigs[ref.id] = []
print('Step␣5␣done:␣Contigs␣assembled␣for␣given␣parameters\n'
    )

#Last step: Final files creation
logger.info('writing␣output␣Files')
for ref in assembled_contigs:
    good_list = []
    for i, contig in enumerate(assembled_contigs[ref]):
        good_list.append(SeqRecord.SeqRecord(contig.seq, id='
            contig_'+ str(i)))
    with open(output_prefix + ref + '.good_contigs.' +
        output_type, 'w') as output_good:
        SeqIO.write(good_list, output_good, output_type)


bad_list = []
for contig in removed_contig_list:
    bad_list.append(SeqRecord.SeqRecord(contig.seq, contig.id
        ))
with open(output_prefix + '.badContigs.' + output_type, 'w')
    as output_bad:
    SeqIO.write(bad_list, output_bad, output_type)
```

```
print('Program finished\n')
```

Apéndice B

El método Profiler

En este apéndice se incluye una explicación del método profiler así como el significado de cada una de sus columnas

B.1. Introducción a profiler

Profiler proporciona un perfil determinístico de los programas python. Este perfil es un conjunto de estadísticas que describen cuántas veces son ejecutadas cada una de las partes de un programa y durante cuánto tiempo. [2]

B.2. Significado de las cabeceras

Profile devuelve, entre otros, una tabla con 6 columnas de información:

- ncalls

  Número de llamadas realizadas a esa función.

- tottime

  Tiempo total empleado en dicha función. Se excluye el tiempo que gastan las llamadas a subfunciones.

- percall

Cociente entre tottime y ncalls, es decir, tiempo que emplea cada función por uso.

- cumtime

Tiempo acumulado que se emplea en esta función y en todas las subfunciones, desde que se llaman hasta que finalizan.

- percall

Similar al anterior percall, cociente entre cumtime y las llamadas primitivas.

- filename:lineno(function)

Proporciona la información relevante de la función: qué función es, a qué algoritmo pertenece y en qué línea está.

Bibliografía

[1] *https : //www.umh.es/contenido/pdi/ : uor*_106/*datos_es.html*

[2] *https : //docs.python.org/*2/*library/profile.html*

[3] Dugundi, J., Topology, Allyn and Bacon, 1970

[4] Runde, V., A taste of Topology, Springer, 2005

[5] Simmons, G., Topology and Modern Analysis, McGraw-Hill, 1963