Diseño de una herramienta para el filtrado y anotación de variantes genómicas

Didac Barroso Bergadà
Máster universitario en Bioinformática y Bioestadística UOC-UB
Bioinformática translacional, análisis de datos y genómica del cáncer

Elisabeth Castellanos Pérez
Carles Ventura Royo

05/06/2018

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Diseño de una herramienta para el filtrado y anotación de variantes genómicas*
Nombre del autor:	*Didac Barroso Bergadà*
Nombre del consultor/a:	*Elisabeth Castellanos Pérez*
Nombre del PRA:	*Carles Ventura Royo*
Fecha de entrega (mm/aaaa):	*06/2018*
Titulación::	*Máster universitario en Bioinformática y Bioestadística UOC-UB*
Área del Trabajo Final:	*Bioinformática translacional, análisis de datos y genómica del cáncer*
Idioma del trabajo:	*Castellano*
Palabras clave	*Variante Genómica* *Variant Call Format (VCF)* *Next Generation Sequencing (NGS)*

Resumen del Trabajo (máximo 250 palabras):

La causa de las enfermedades hereditarias está ligada a variaciones en el genoma. Una de las formas más extendidas de identificar estas variaciones es la secuenciación mediante *Next Generation Sequencing* (NGS). Dichas técnicas de NGS generan una gran cantidad de datos genómicos. Para gestionar estos datos es necesario un tratamiento bioinformático robusto capaz de obtener resultados científicamente interpretables. Los pipelines de análisis de los datos de secuenciación permiten identificar las variantes genómicas comparando la información de secuenciación con un genoma de referencia. Las variantes identificadas se almacenan en unos ficheros en un formato llamado *Variant Call Format* (VCF). Los archivos VCF tienen una estructura compleja y pueden contener millones de variantes. Estos hechos dificultan en gran manera su gestión.

En este trabajo se ha desarrollado una herramienta que permite gestionar ficheros VCF de gran tamaño de forma eficiente y aportar al usuario la información necesaria para que este seleccione que variantes quiere obtener mediante un robusto sistema de filtrado. Además el usuario puede incluir nueva información mediante diferentes sistemas de anotación. Después de la gestión del fichero VCF, se obtiene una tabla con las variantes resultantes en un formato leíble. Para facilitar la interacción entre el usuario y la herramienta se ha desarrollado una aplicación web. Mediante esta aplicación, el usuario define todos los parámetros necesarios para la importación, filtrado y anotación.

Para comprobar el correcto funcionamiento de la herramienta se han llevado a cabo diferentes procesos de filtrado y anotación paralelos para validar que los resultados obtenidos son correctos.

Abstract (in English, 250 words or less):

The cause of the inherited diseases is linked to variations in the genome. One of the most widespread ways to identify these variations is sequencing through Next Generation Sequencing (NGS). The NGS techniques generate a large amount of genomic data. To manage these data, a robust bioinformatic treatment capable of obtaining scientifically interpretable results is necessary. Pipelines for analysis of sequencing data allow to identify genomic variants by comparing sequencing information with a reference genome. The identified variants are stored in files in a format called Variant Call Format (VCF). VCF files have a complex structure and may contain millions of variants. These facts make their management very hard.

In this work we have developed a tool that allows us to manage large VCF files efficiently and provide the users with the necessary information so that they can select which variants they want to obtain through a robust filtering system. In addition, the users can include new information through different annotation systems. After the management of the VCF file, a table with the resulting variants in a readable format is obtained. To facilitate interaction between the users and the tool, a web application has been developed. Through this application, the users define all the necessary parameters for import, filtering and annotation.

To verify the correct functioning of the tool, parallel different filtering and annotation processes have been carried out to validate that the results obtained are correct.

Contenido

Lista de Figuras

1. Introducción

1.1 Contexto y justificación del trabajo

1.1.1 Contexto

Variación genética

Cada copia del genoma humano es única, y difiere en secuencia de cualquier otra copia en la población en aproximadamente 1 en cada 1.250 nucleótidos. Esta varianza en la secuencia de ADN influye en características individuales tales como la apariencia física, la susceptibilidad a las enfermedades y la respuesta a los tratamientos médicos(1).

El término variante puede usarse para describir una alteración que puede ser benigna, patógena o de significado desconocido(2). El término variante se usa cada vez más en lugar del término mutación.

Existen diferentes tipos de variantes genómicas. Estas pueden ser: *single nucleotide variant* (SNV), inserciones o supresiones. Los polimorfismos de un solo nucleótido o SNV son cambios de un nucleótido por otro, las inserciones son la inclusión de uno o varios nucleótido en la secuencia genómica y las supresiones son la eliminación de uno o varios nucleótido en la secuencia (3). La causa principal de las variaciones son los errores en la replicación del ADN. La maquinaria de replicación del ADN tiene una gran fiabilidad, la probabilidad de que se produzca una mutación en una base durante la replicación es de entre 10^{-9} y 10^{-10}, esta fiabilidad se consigue gracias a la combinación de varios procesos. Las polimerasas restringen la tasa de mutación a través de una combinación de selección de nucleótidos, corrección exonucleolítica de copias erróneas y reparación de desemparejamiento post-replicativo (MMR)(4).

Cuando las variaciones genéticas corresponden a una secuencia de ADN de gran tamaño podemos hablar de variaciones estructurales. Este tipo de variaciones pueden ser ocasionadas por la reordenación cromosómica.

La alta capacidad de la célula para controlar la estabilidad genética no impide que se puedan dar variantes genómicas que resultan ser patogénicas. En enfermedades de carácter genético como la enfermedad de Huntington o la hemofilia A, se ha identificado la presencia de SNV en la secuencia de algunos genes relacionados con la estas enfermedades (5,6). También se hipotetiza que defectos en el mecanismo de control de la estabilidad genética pueden alimentar el proceso de aparición de variantes genéticas, contribuyendo a la aparición de enfermedades genéticas. Por lo tanto, el

estudio de las variantes genéticas puede ser un mecanismo efectivo de diagnóstico de enfermedades genéticas y ayudar a comprender las causas por las cuales se desarrollan dichas enfermedades.

Secuenciación

Para localizar e identificar las variantes genéticas de un sujeto de estudio una de las herramientas más extendidas es la secuenciación. Después del descubrimiento de la estructura tridimensional del ADN empezó el lento desarrollo de técnicas que permitían el estudio de la secuencia de ADN. Dado el gran tamaño de las moléculas de ADN, las técnicas iniciales solo obtenían la composición de dichas moléculas, sin aportar información sobre su orden. En 1972 Walter Fier obtuvo por primera vez la primera secuencia genética codificante, en este caso procedente de un bacteriófago, usando las técnicas de detección de fragmentos de DNA marcados con radiación propuesta por Fred Sanger. Cuatro años más tarde Fier obtuvo el genoma completo de dicho bacteriófago(7). En los años siguientes la técnica propuesta por Sanger fue mejorando. Se incluyeron los dideoxido nucleótidos para delimitar el final de la polimerización, se cambió el sistema de marcado introduciendo la detección fluoro-métrica y se mejoró la detección con el uso de la electroforesis basada en capilares(8). Así nacía la primera generación de métodos de secuenciación.

La segunda generación de métodos de secuenciación se empezó a desarrollar gracias al cambio de la detección fluoro-métrica por la detección usando la reacción del pirofosfato, lo que eliminaba la necesidad de usar nucleótidos marcados con fluoróforos(9). Este cambio sumado a la automatización de las reacciones de PCR para la secuenciación permitió secuenciar por primera vez el genoma de un ser humano en el año 2001(10). Hoy en día las técnicas de secuenciación han seguido mejorando hasta alcanzar la tercera generación o *Next Generation Sequencing* (NGS). Existen diferentes secuenciadores comerciales que permiten llevar a cabo la NGS: como *Ilumina*, *Solexa e Ion proton*(11). Los secuenciadores ofrecen un rendimiento tan alto que se puede secuenciar el genoma entero de un ser humano en un periodo corto de tiempo y a un precio asequible, hasta el punto en que la secuenciación de todo el genoma ya se usa como método para diagnosticar enfermedades genéticas (12). La gran cantidad de

datos de secuenciación que se obtiene gracias a esta secuenciación de nueva generación, así como la alta sensibilidad requerida para realizar ciertos análisis requiere cada vez más unos procesos bioinformáticos de análisis de datos más robustos y eficientes. Al disponer de gran cantidad de datos es necesario un proceso de tratado que pueda hacer converger toda la información obtenida en respuestas concretas a las cuestiones planteadas antes de la secuenciación(13).

Para realizar un proceso de secuenciación es necesario purificar una cantidad se ADN suficiente de una muestra. Normalmente son necesarios como mínimo unos 200ng de ADN purificado. Este ADN se fragmenta en pequeños fragmentos de entre 100 y 500 pares de bases. A los fragmentos se les añade un *barcode*, una secuencia corta de nucleótidos que permite la identificación de la muestra a la cual pertenece cada fragmento. Después de añadir el *barcode* se amplifican los fragmentos para disponer de más copias para poder secuenciar. Este conjunto de fragmentos amplificados se llama librería. Las librerías se introducen en un secuenciador de *next generation sequencing* (NGS)(14). El secuenciador separa cada fragmento por muestras en función de su *barcode* y realiza el proceso de secuenciación explicado anteriormente.

Pipeline de análisis de datos de secuenciación

FastQ format

Al final del proceso automatizado de secuenciación se obtienen unos ficheros llamados *FastQ.* Estos ficheros contienen una entrada por cada pequeña secuencia de entre 30 y 100 pares de bases obtenida durante la secuenciación. Cada pequeña secuencia corresponde a la secuenciación de una parte de un fragmento de DNA de la librería.

Una entrada de fichero *FASTQ* dispone de tres líneas. En la primera se identifica cada pequeña secuencia obtenida de la secuenciación de los fragmentos de ADN. La segunda contiene el orden de los nucleótidos del fragmento de ADN de formato *FASTA.* Este formato es el más extendido para transcribir secuencias de nucleótidos, usando las letras A, T, C y G para simbolizar cada nucleótido. La tercera línea es un valor de calidad para cada nucleótido que describe la fiabilidad de la lectura basándose en los parámetros experimentales mediante los cuales se ha obtenido. Dependiendo del tipo

de secuenciador usado y del número de muestras secuenciadas, al final del proceso de secuenciación se pueden llegar a obtener unos 10 millones de secuencias(15).

Alineamiento de secuencias: BAM format

Para poder integrar todas estas pequeñas secuencias o *reads* en una sola secuencia a partir de la cual se pueda otorgar significado biológico es necesario alienar toda la información usando un genoma de referencia. Varios algoritmos son capaces de realizar esta tarea. Los más comunes son BWT (*Burrows-Wheeler transform*), *space seed* y *hashing.* Existen varias aplicaciones que implementan el uso de los algoritmos para integrarlos en un pipeline de análisis(16).

Una vez realizado el alineamiento sobre el genoma de referencia se obtiene un fichero llamado *sequence alingment map* (SAM). Este tipo de ficheros normalmente se usa en forma binaria para facilitar su computación. La forma binaria de los ficheros SAM es el *binary alingment map* (BAM). Los ficheros BAM contienen la posición genómica de cada *read* permitiendo facilitar la detección de las diferencias entre las secuencias obtenidas y un genoma de referencia(17). Al alinear los *reads* normalmente quedan solapados entre ellos. Por lo tanto cada nucleótido del genoma de referencia puede formar parte de la secuencia de varios *reads*. El número de *reads* que contienen cada nucleótido se denomina cobertura o *coverage*. Como más *coverage* presente un nucleótido, más seguro se puede estar que el tipo de nucleótido leído en la secuenciación es correcto. Al realizar un alineamiento puede ser que existan divergencias entre el nucleótido descrito por los *reads* y el genoma de referencia. En este caso se trata de una variante genómica.

Variant Calling Format (VCF)

Variant Calling

Uno de los tipos de información de relevancia biológica que se puede obtener a partir de los datos de secuenciación es el listado de las variantes genómicas encontradas. Para poder obtener este listado se realiza un proceso llamado *variant calling*. Existen varias herramientas para realizar el *variant calling* como GATK o STRELKA. Estas herramientas parten del fichero de alineamiento BAM y realizan un tratamiento estadístico de los

diferentes *reads* para identificar si existe una variante. Los parámetros de estas herramientas se pueden modificar y normalmente cada variante debe tener un mínimo *coverage* para ser identificada o "llamada". Se puede determinar la cigosidad de las variantes en función del número de reads en los que se encuentran. Si el número de reads con la variante es de cerca del 100%, dicha variante será homocigota, es la variante se encontrará en las dos dotaciones cromosómicas. Si se encuentra en alrededor del 50% de los *reads* la variantes es heterocigota, solo se encuentra en una dotación cromosómica. Las variantes con unos *reads* inferiores al 50% pueden corresponderse al mosaicismo. Es decir, que solo algunas células de todas las analizadas tienen la variante(18). Dependiendo del sistema de *variant calling,* el porcentaje de reads para clasificar las variantes puede variar. Dependiendo del sesgo en la amplificación y del proceso de preparación de las librerías, se ha establecido que el porcentaje de *reads* para atribuir heterocigosidad puede variar entre el 30% y el 70%.

A partir del *variant calling* se identifican tres tipos de variantes: *single nucleotide variants* (SNV), inserciones y deleciones(19). Los SNV son un cambio concreto de un nucleótido en los *reads* en comparación con el genoma de referencia. Las deleciones son nucleótidos del genoma de referencia que no se encuentran en los *reads*. Y las inserciones son nucleótidos presentes en los *reads* que no se encuentran en el genoma de referencia. Todas las variantes encontradas son almacenadas en unos ficheros con un formato llamado *variant call format* (VCF).

Archivos VCF

Los archivos VCF consisten en un encabezamiento de un número arbitrario de líneas, que describe el proceso por el cual se han identificado las variantes así como su anotación. Después del encabezamiento se encuentra el listado de variantes. Cada línea representa una variante. Al principio de la línea se describe la variante con su posición genómica (cromosoma, posición, dirección de la cadena), el nucleótido del genoma de referencia y el nucleótido alternativo encontrado en los *reads*. Después se encuentra la información sobre la calidad otorgada a la detección de la variante y su filtrado. Siguiendo la línea se encuentra información sobre el genotipo de la variante, como por ejemplo la frecuencia alélica. Por último se encuentra un campo donde se puede añadir

la anotación deseada(20). Existen aplicaciones como *Annovar* que permiten añadir dicha anotación de forma automatizada.

Los archivos VCF aportan una gran flexibilidad al tratado de las variantes y su almacenamiento, no obstante, no tienen una estructura que permite que sean leídos por el usuario. Además pueden llegar a almacenar millones de variantes haciendo imposible la lectura por parte de la computadora.

1.1.2 Justificación

El análisis de datos de *next generation sequencing* (NGS) devuelve una gran cantidad de variantes genéticas. Esas variantes son procesadas por un pipeline de análisis establecido en el grupo de investigación, generando archivos *Varian Call Format* (VCF) de gran tamaño. Los archivos VCF contienen las variantes genéticas encontradas al comparar una secuencia de ADN obtenida con un ADN de referencia. Estos archivos contienen la lista de variantes genómicas, el control de calidad y la anotación para cada variante. La anotación se obtiene de diferentes bases de datos públicas como *RefGene, ExAc* o 1000 Genomas, entre otros(21). Sin embargo, esta gran cantidad de información se vuelve difícil de usar debido al gran tamaño y la estructura interna de los archivos VCF. Los investigadores necesitan una herramienta bioinformática que pueda procesar los archivos VCF y ayudarlos a responder la consulta que puedan tener sobre ellos.

1.2 Objetivos del trabajo

Diseñar scripts de R capaces de leer, filtrar, comparar y extraer datos de archivos VCF

Diseñar los scripts necesarios para la extracción de datos y filtrado y documentarlos adecuadamente. Diseñar todas la funciones necesarias para filtrar y anotar usando bases de datos.

Integrar los scripts en una herramienta que pueda filtrar la información requerida de los archivos VCF.

Diseñar una aplicación web basada en el paquete shiny de R para integrar todas las funciones de importación, filtrado, anotación y exportación. La aplicación ha de permitir al usuario una interacción sencilla e intuitiva con las herramientas de filtrado. Diseñar

también las funciones que permitan la exportación de los resultados en un formato legible para el usuario.

1.3 Metodología y enfoque

1.3.1 Concepción de la herramienta

Para el desarrollo de la herramienta se han descrito los conceptos alrededor de los cuales ha de pivotar el proceso de diseño:

Implementación: El objetivo de este diseño es aportar a los usuarios una herramienta que permita filtrar y anotar las variantes presentes en un fichero VCF.

Versatilidad: La herramienta no ha de ceñirse a una estructura concreta de fichero VCF. Ha de ser capaz de adaptar los procesos a cada tipo de VCF. Además la estructuración de los filtros ha de permitir al usuario obtener las respuestas que necesita, permitiendo tantas opciones de filtrado como sea posible.

Gestión de la memoria: Los archivos VCF pueden contener una gran cantidad de información. El uso de esta gran cantidad de información puede requerir una gran potencia computacional y un gran uso de la memoria de la computadora que lleva a cargo el proceso de análisis. La herramienta ha de ser capaz de realizar los procesos de filtrado usando el mínimo de memoria imprescindible. Además ha de realizar los procesos en un tiempo de ejecución razonable. La herramienta además ha de ser capaz de gestionar los casos en que no se disponga de suficiente memoria fraccionando la ejecución.

Interacción: En la medida de lo posible la herramienta ha de poder ser usada de forma sencilla por el usuario, sin requerir de grandes conocimientos de programación.

1.3.2 Herramientas bioinformáticas

Las herramientas bioinformáticas usadas para el desarrollo han sido:

Sistema operativo

El sistema operativo usado para el desarrollo de la herramienta es una distribución estable de *Debian*. *Debian* es un sistema operativo abierto y gratuito diseñado por una

amplia comunidad de desarrolladores. *Debian* usa el *kernel* o núcleo del sistema operativo del sistema *Linux*(22). Algunas funciones de la aplicación usan la interacción con el sistema operativo para llevar a cabo diversas tareas como prefiltrar o buscar en bases de datos.

Lenguaje R

La herramienta ha sido diseñada usando el entorno R. R es un conjunto integrado de diferentes tipos de software para manipulación de datos, cálculo y visualización gráfica. R está distribuido por el proyecto CRAN y tiene un acceso gratuito. El proyecto CRAN además mantiene R y lo actualiza periódicamente(23). El lenguaje usado por R permite acciones típicas de cualquier lenguaje de programación como condicionales, *loops* y funciones recursivas. Además R está diseñado para poder llevar a cabo manipulación sencilla de los datos por medio de un sistema de objetos donde se puede almacenar la estructura en múltiples formatos, des de vectores numéricos o matrices a objetos complejos que pueden almacenar coordenadas genómicas o archivos VCF. Esta versatilidad permite poder gestionar grandes cantidades de datos de forma eficiente, reduciendo las necesidades computacionales del diseño.

La otra ventaja que aporta R para gestionar grandes cantidades de datos es la existencia de miles de funciones diseñadas por otros usuarios. Estas funciones se pueden obtener de forma sencilla en forma de paquetes y su uso permite la implementación sencilla de código escrito por otros usuarios. En el diseño de la herramienta se han usado cuatro paquetes de R:

memoise: Guarda en caché los resultados de una función para que cuando la vuelvas a llamar con los mismos argumentos, devuelva el valor pre calculado(24).

yaml: Importa y exporta ficheros con extensión *yaml*. Los ficheros *yaml* permiten la estructuración sencilla de los datos en forma de lista(25).

tools: Herramientas para el desarrollo, administración y documentación de paquetes(26).

shiny: Este paquete proporciona herramientas para diseñar una aplicación web de forma sencilla e intuitiva. Se ha usado para el diseño de la interfaz gráfica de la aplicación(27).

Bioconductor

Bioconductor es un proyecto de software de desarrollo y código abierto que proporciona herramientas para el análisis y la comprensión de datos genómicos de alto rendimiento. Se basa principalmente en el lenguaje de programación R. *Bioconductor* ofrece paquetes para la manipulación de todo tipo de datos biológicos como *microarrays* de DNA, secuencias o SNV. Para el diseño de la herramienta se han usado tres paquetes de *Bioconductor*(28):

VariantAnnotation: Este paquete permite la importación y la manipulación de ficheros VCF. *VariantAnnotation* permite cargar solo los campos necesarios de un fichero VCF reduciendo de gran manera el uso de la memoria por parte de la computadora. En el diseño de la herramienta este paquete ha sido usado para la importación de datos desde el fichero VCF hasta el entorno R(29).

regionR: Ofrece un marco estadístico basado en pruebas de permutación personalizables para evaluar la asociación entre conjuntos de regiones genómicas y otras características genómicas(30).

BamSignals: Este paquete permite obtener de manera eficiente vectores de cobertura de archivos BAM indexados. Cuenta el número de lecturas en rangos genómicos dados y computa perfiles de lectura y perfiles de cobertura(31).

Bases de datos

Bioconductor ofrece paquetes que permiten interactuar con diferentes bases de datos realizando una consulta sobre un campo de la base datos y obteniendo los genes relacionados. También permite aportar nombres de genes y obtener toda la información de un campo concreto relacionada con esos genes. Este proceso se lleva a cabo usando la función *select*. Los paquetes de bases de datos usados siguiendo este sistema son:

AnnotationHub: Permite obtener los nombres de los genes en códigos de diferentes bases de datos(32).

GO.db: Es una base de datos que mantiene un modelo computacional actualizado y completo de sistemas biológicos, desde el nivel molecular hasta vías más grandes, sistemas celulares y de organismos(33).

reactome.db: Es una base de datos de vías metabólicas revisada manualmente(34).

MsigDB: Es una base de datos de firmas moleculares de diferentes grupos de genes anotados(35).

KEGGREST: *KEGG* es una base de datos usada para comprender las funciones y características de elementos de alto nivel del sistema biológico, como la célula, el organismo y el ecosistema, a partir de información de nivel molecular, especialmente conjuntos de datos moleculares a gran escala generados por secuenciación del genoma(36).

No todas las bases de datos tienen un sistema de interacción basado en paquetes de *Bioconductor*. En el caso de las bases de datos *Uniprot* y *Cosmic* se ha descargado la base de datos completa y se ha almacenado en el directorio de la aplicación(37,38). Para llevar a cabo las consultas en estas bases de datos se ha usado el mecanismo de interacción con el sistema de R.

1.3.3 Metodología

La base del proceso de desarrollo ha de ser la interacción con los usuarios futuros de la herramienta, los investigadores. La comunicación y la interpretación de las necesidades de los investigadores es básico para poder crear una herramienta útil para la investigación.

El desarrollo bioinformático de la herramienta está basado en el diseño de funciones cortas que responden a cada necesidad concreta del proceso. Estas funciones han de ser usadas por funciones de ejecución que centralicen cada proceso de forma no específica. Este diseño ha de permitir aportar flexibilidad al desarrollo pudiendo ampliar las capacidades de la herramienta diseñando una función corta, que pueda ser usada por las funciones de ejecución

1.4 Planificación

1.4.1 Recursos

La herramienta se desarrolla en el entorno del grupo de investigación de cáncer hereditario del instituto Germans Trias y Pujol. Este hecho permite la interacción directa con los investigadores, futuros usuarios de la herramienta, posibilitando un *feedback* constante durante el proceso.

Durante el desarrollo de la herramienta en el instituto se dispondrá de un ordenador de sobremesa. Este ordenador dispone del sistema operativo *Debian* y un entorno de uso del lenguaje R, *Rstudio*. Este ordenador está conectado en red con un servidor central de más capacidad computacional para realizar los procesos de la herramienta.

1.4.2 Planificación temporal

En el anexo 1 se adjuntan los diagramas de Gantt con la planificación temporal llevada a cabo para la realización del proyecto. Se adjuntan tres diagramas de Gantt. El primero corresponde al diseño de funciones descrito en el primer objetivo. El segundo corresponde al cumplimiento del segundo objetivo y el tercero a la redacción de documentación.

1.5 Producto obtenido

El producto obtenido es una herramienta en forma de aplicación web que permite al usuario la lectura, filtrado y comparación de bases de datos de forma intuitiva. Para la ejecución de dicha aplicación se ha diseñado diversas funciones agrupadas en los siguientes ficheros:

Input.R: Documento con las funciones en R para la obtención de la estructura I la importación de datos de los ficheros VCF.

Filter.R: Documento con las funciones en R para el filtrado.

Utils.R*:* Documento con varias funciones de apoyo en R.

Database.R: Documento con las funciones de interacción con bases de datos en R.

Batchworking.R: Documento con la función para trabajo en *batch* en R.

Compare.R: Documento con las funciones para comparación de ficheros en R

Execute.R: Documento con la función para la ejecución del filtrado con todas las funciones integradas

ExonCoverage.R: Documento con la función para la comprobación de la cobertura de las ROIs

app.R: Documento con el diseño de la aplicación web.

También se ha escrito una guía para que el usuario pueda usar la herramienta:

Readme.txt: Guía de usuario de la herramienta

1.6 Breve descripción de los siguientes capítulos

Diseño de la herramienta: En este apartado se describe cada paso de ejecución de la herramienta y su interacción gráfica.

Ejemplo de Validación: En este apartado se describe un ejemplo del proceso de validación de la herramienta para comprobar que no existieran errores en el diseño.

2. Diseño de la herramienta

Como se ha detallado anteriormente, el objetivo de este proceso de desarrollo es obtener una herramienta que permita la importación, filtrado y anotación de ficheros VCF de forma fácil para el usuario. Además también se quiere que la herramienta devuelva los resultados de forma legible, en un formato de tabla de datos en que la información de cada variante ocupe una fila y cada campo del fichero VCF una columna. Este capítulo detalla el diseño y funcionamiento de esta herramienta mostrando las partes de la interfaz que permiten llevar a cabo cada parte del proceso de ejecución de la herramienta.

Para que el usuario pueda interaccionar con las funciones de la herramienta se ha desarrollado una aplicación en formato web usando el paquete *shiny* de R, descrito en la introducción. El código correspondiente a la configuración visual y el servidor de ejecución se encuentra en el documento *app.R* del anexo 2.

2.1 Escaneado e importación de ficheros VCF: *Scan VCF header*

La primera parte de la aplicación permite al usuario seleccionar el directorio donde se encuentran los ficheros VCF para analizar y el directorio donde quiere que se almacene el resultado del filtrado. En la figura 1 se puede apreciar el sistema de selección

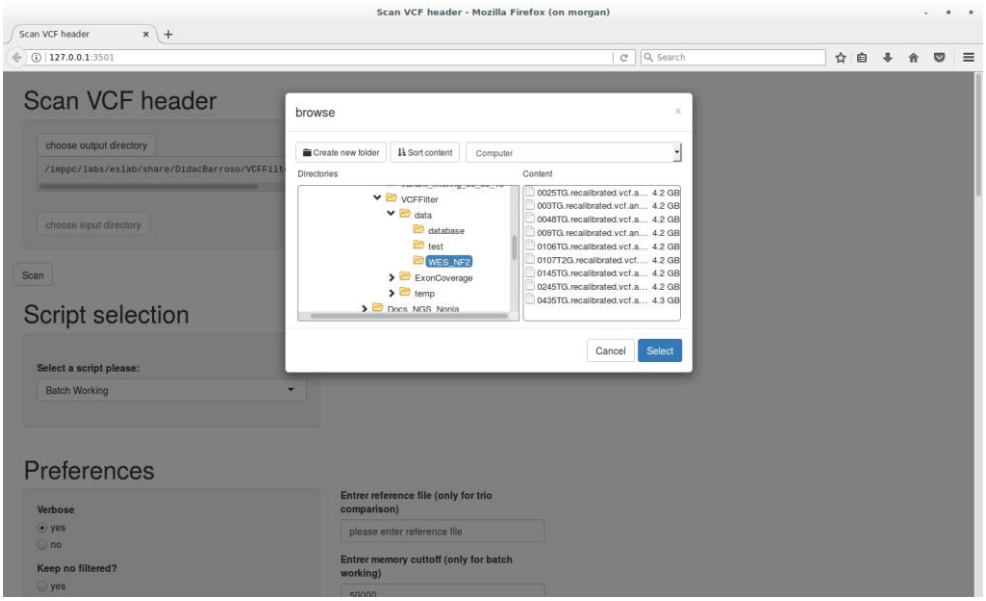


Figura 1: Selección de ficheros

Una vez seleccionados estos parámetros el usuario hace clic en el botón *Scan.* Al pulsar este botón se llama a la función *prepareHeaderOutput* aportándole la información sobre la localización de los directorios. Esta función usa el paquete *VariantAnnotation* para extraer la información del *header* de los ficheros VCF. Luego comprueba que todos los ficheros del directorio compartan el mismo *header*. En caso que existan diferencias en el *header* de los ficheros la función devuelve un error. Este hecho es causado por la necesidad que todos los ficheros a filtrar compartan la misma estructura. Este error, así como todos los errores que pueden suceder durante la ejecución de la función, son mostrados en un cuadro que aparece debajo el botón de escaneado. Una vez comprobada la estructura, la función devuelve una tabla con todos los campos presentes en el fichero VCF como se ve en la figura 2. En esta tabla se describe la información contenida en cada campo para que el usuario pueda elegir el tipo de información por la que quiere filtrar. También se describe si el tipo de información que contiene cada campo es numérica o texto.

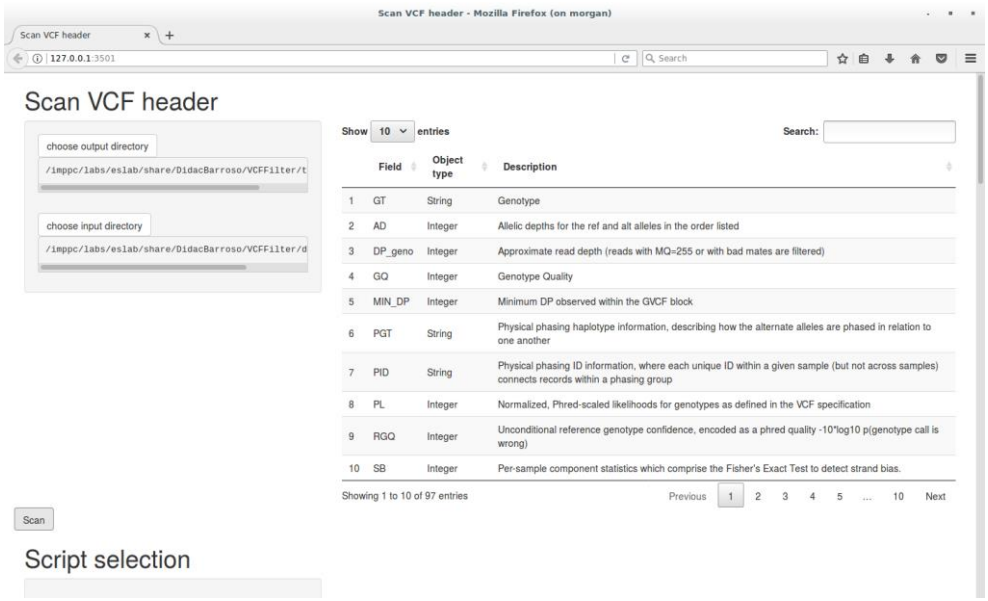


Figura 2: Escaneado

2.2 Tipos de ejecución: *Script selection*

Una vez el usuario conoce la información contenida en los ficheros VCF puede empezar a seleccionar los parámetros para el filtrado. El primer parámetro a seleccionar es el tipo de ejecución de filtrado que quiere llevar a cabo. Cada tipo de ejecución llama a una función concreta que realiza un proceso de filtrado diferente (Figura 3). Existen tres funciones de ejecución diferenciadas:

***Batch working**:* Al llamar esta función se computan todos los ficheros VCF como elementos individuales, a cada fichero se le aplican los filtros definidos, se añade la anotación requerida y se genera un fichero de salida en la carpeta de *output* seleccionada por el usuario.

Compare trio: Para la ejecución de esta función es necesario que en la carpeta de *input* solo se encuentren tres ficheros VCF. También es necesario que en la sección de *Preferences* se introduzca el nombre de uno de los tres ficheros que se tomará como fichero de referencia. Al ejecutarse esta función se aplican los filtros definidos sobre las variantes del fichero VCF de referencia y se anota. Después se comprueba si las variantes que han pasado los filtros están presentes en los otros dos ficheros y se añade una columna en el output con esta información. Esta función está diseñada

especialmente para comprobar si un hijo ha heredado las variantes genéticas de alguno de sus dos progenitores.

Compare files: Al ejecutar esta función se aplican los filtros definidos a cada uno de los ficheros del directorio *input*. Las variantes que pasan los filtros son almacenadas en una lista. Cada entrada de la lista tiene como identificador la definición de la variante y contiene como información los ficheros donde se encuentra dicha variante. Después de computar todos los ficheros se devuelve un solo documento con todas las variantes de la lista, anotando en que ficheros se encuentra la variante. Cabe destacar que las variantes de cada fichero tienen una información sobre el genotipo diferente. Por lo tanto, al ejecutar la función se devolverá la información sobre el genotipo de uno de los ficheros en los que se encuentra.

Las tres funciones se pueden consultar en los archivos *compare_files.R* y *batch_working.R* del anexo. Se ha decidido optar por tres funciones de ejecución diferentes para poder hacer la herramienta más flexible y responder a las diferentes necesidades de filtrado que se puedan dar.

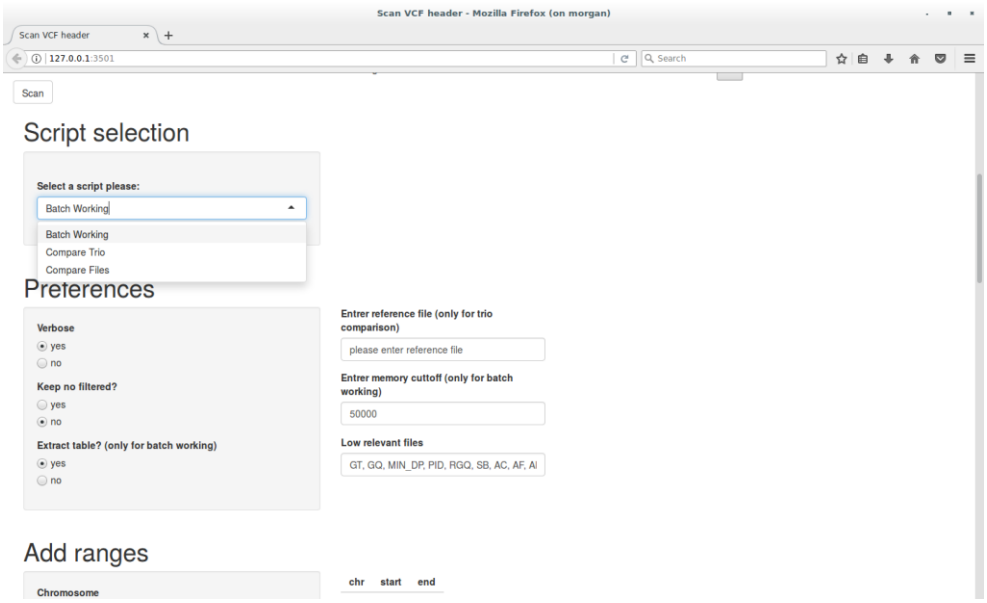


Figura 3: Selección de ejecución

2.3 Preferencias de ejecución: *Preferences*

Después de elegir el tipo de ejecución el usuario puede elegir diferentes preferencias de ejecución (Figura 3). Estas preferencias permiten modular el tipo de ejecución de la herramienta para adaptarla a las necesidades del usuario. Las diferentes preferencias a modular son:

Verbose: Durante la ejecución de la herramienta se han preparado una serie de mensajes para que el usuario pueda comprobar su desarrollo. Esta opción permite al usuario decidir si quiere que se muestren o no estos mensajes en el terminal de ejecución.

Keep no filtered*:* Durante la preparación del archivo de salida se extraen toda la información de las variantes del archivo VCF usando el paquete *VariantAnnotation*. La función de este paquete encargada de la extracción utiliza las coordenadas genómicas concretas de la variante para cargar sus datos. Puede ser que dos variantes con las mismas coordenadas sean cargadas pero solo una haya pasado los filtros. En este caso la herramienta permite elegir si se quiere mantener estas variantes que no han pasado los filtros en el archivo de salida o mantenerlas. En el caso que se quieran mantener se añade anotación a la variante describiendo si ha pasado o no los filtros.

Extract table: Según el tipo de filtrado se puede devolver una gran cantidad de variantes. En este caso no es posible carga en la memoria esa gran cantidad de variantes y transformar-las a un formato legible. Por lo tanto, en el caso que se haya

elegido el tipo de ejecución *batch working* se puede elegir si se quiere que el archivo de salida tenga formato VCF o de tabla legible. Esto permite guardar las variantes otra vez en formato VCF sin tener problemas de memoria para posteriores análisis aunque no puedan ser consultadas por el usuario. En el caso de devolver las variantes en formato VCF no es posible añadir anotación mediante la herramienta. Cuando se opta por las ejecuciones *compare trio* y *compare multiple file* no es posible elegir esta opción. Como se ha descrito en el apartado anterior estos dos tipos de ejecución requieren añadir anotación para su correcto funcionamiento, por lo tanto solo se puede obtener el archivo de salida en formato legible.

Reference file: En el caso que se opte por el modo de ejecución *compare trio,* en este campo se ha de escribir el fichero para ser usado de referencia, como se ha detallado en el apartado anterior.

Memory cuttoff: En el caso que se disponga de una computadora con poca memoria es posible que al ejecutar la herramienta no se exista suficiente memoria para cargar la información necesaria. En ese caso la herramienta ofrece la opción de dividir las variantes por sus coordenadas genómicas para fragmentar la ejecución del filtrado. Sin embargo las variantes no están distribuidas de forma uniforme por el genoma y, por lo tanto, no se puede hacer un cálculo general de la memoria usada por todo el fichero VCF para poder fragmentar-lo, dado que algunas regiones con muchas variantes podrían igualmente colapsar la memoria. Por lo tanto se calcula la memoria que ocupan las variantes situadas en una región genómica de un millón de pares de bases. El usuario puede delimitar cuanto espacio espera que ocupen estas variantes en la memoria (en megabytes). Si el valor obtenido por la herramienta es mayor al aportado por el usuario, se dividen las variantes por su posición genómica en el mínimo necesario para no sobrepasar ese límite.

Low relevant files: Los archivos VCF pueden contener gran cantidad de información para cada variante. A veces el usuario no necesita consultar algunos campos que no son necesarios para su estudio. La herramienta ofrece la opción de reordenar el archivo de salida legible, de forma que los campos que el usuario considere poco relevantes se ubiquen en las columnas del final del documento. En este apartado se pueden escribir aquellos campos que una vez realizado el escaneo no se consideren útiles. Cada nombre de campo debe estar separado por una coma

2.4 Zonas genómicas: Ranges

La herramienta ofrece la posibilidad de limitar la zona genómica de las variantes a filtrar. Para llevar a cabo esta limitación de regiones se usa la opción del paquete *VariantAnnotation* que permite delimitar las variantes cargadas en la memoria. De esta forma, si el usuario delimita unas zonas genómicas de interés se reducirá de forma importante el número de variantes haciendo el proceso mucho más eficiente computacionalmente. Las regiones genómicas se pueden seleccionar en la sección *Add ranges* de la aplicación (Figura 4). En esta sección el usuario introduce el nombre del cromosoma, el nucleótido de inicio y el de final de la región genómica. El usuario puede introducir tantas regiones genómicas como quiera.

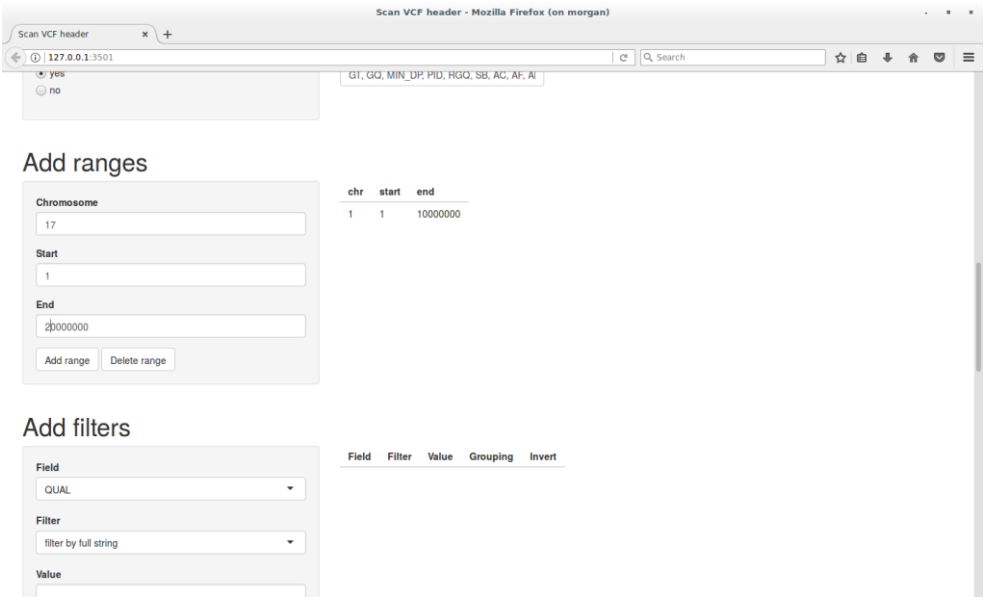


Figura 4: Ranges

2.5 Filtrado

Las variantes almacenadas en un fichero VCF pueden llegar a superar el millón. Esta gran cantidad de variantes hace que el usuario no pueda manejar tal cantidad de datos y además, muchas veces, no es computacionalmente posible cargar toda la información de las variantes en memoria. En consecuencia la herramienta ofrece al usuario la capacidad de filtrar las variantes para obtener solo las que necesita. Para realizado el filtrado se ha desarrollado una función llamada *computeMultipleQuery*. Esta función

recibe tres inputs básicamente, la localización del fichero que se quiere filtrar, una matriz de datos con los diferentes filtros y, opcionalmente, unas coordenados genómicas para limitar el filtrado a una región en concreto. La función se basa en iterar por las diferentes demandas de filtrado. Cada demanda de filtrado consta de cinco secciones: El campo del VCF, el filtro, el valor de filtrado, el grupo en el que se encuentra el filtro y si se quiere invertir el resultado del filtro.

A partir de la primera sección, el campo del VCF, la función usa el paquete *VariantAnotation* para extraer del fichero VCF únicamente el campo requerido. Esto reduce en gran manera el uso de memoria, dado que no es necesario cargar todo el fichero para obtener datos de sus campos. A cada valor del campo extraído se le asigna un identificador único de la variante a la cual corresponde. Una vez extraído el campo se le aplica el filtro. Cada filtro es una pequeña función independiente que es usada por la función *computeMultipleQuery.* El filtro usa el campo extraído y el valor de filtrado para devolver los identificadores de todas la variantes que han pasado dicho filtro.
Por ejemplo si el usuario quiere obtener todas las variantes anotadas como parte del gen NF1, la función primero extraerá el nombre del gen con el que se ha anotado cada variante, luego aplicará el filtro para encontrar textos coincidentes con el valor de filtrado NF1 y obtendrá los identificadores de las variantes que tienen el valor NF1 en el campo nombre del gen.
Se han diseñado nueve filtros diferentes:

filterByFullString: busca si un texto dado coincide con el texto en el campo seleccionado de cada variante. Se debe dar un valor de texto.
filterByContainingString: busca si un texto dado está presente en el campo seleccionado de cada variante. Se debe dar un valor de texto.
filterByMin: busca si el valor numérico del campo seleccionado está sobre un valor dado. Se debe dar un valor numérico. Los decimales deben estar determinados por un punto.
filterByMax: busca si el valor numérico del campo seleccionado está por debajo de un valor determinado. Se debe dar un valor numérico. Los decimales deben estar determinados por un punto

filterByInterval: determina si el valor numérico del campo seleccionado se encuentra entre un valor de intervalo dado. El intervalo debe definirse siguiendo la siguiente estructura: "number-number", ejemplo: 1-10. Los decimales deben estar determinados por un punto

filterByMultipleString: busca si alguna de las secuencias de texto del conjunto dado coincide con la cadena en el texto seleccionado de cada variante. El conjunto de secuencias de texto debe estar separado por el símbolo "/".

filterByContainingMultipleString: busca si alguna de las secuencias de texto del conjunto dado está presente en el texto en el campo seleccionado de cada variante. El conjunto de secuencias de texto debe estar separado por el símbolo "/".

filterByADInterval: este filtro solo se puede usar cuando se selecciona el campo AD (*allelic depth*). El filtro determina si la profundidad alélica está entre un intervalo dado. El valor del intervalo AD debe seguir la siguiente estructura: 1-10 / 10-20.

filterByADFreq: este filtro solo se puede usar cuando se selecciona el campo AD (*allelic depth*). El filtro obtiene la frecuencia alélica de la AD (*allelic depth*) y luego verifica si está por encima de un mínimo. Se debe dar un valor numérico. Los decimales deben estar determinados por un punto

Si se ha elegido invertir el resultado del filtro se devolverán todos los identificadores de las variantes que no hayan pasado el filtro. En el caso del ejemplo anterior, se devolverían todos los identificadores de las variantes anotadas con un gen diferente a NF1.

Los identificadores obtenidos después de computar cada demanda de filtrado se almacenan. Al acabar la iteración por todas las demandas de filtrado se comparan los identificadores de todas aquellas demandas agrupadas juntas y se eliminan los identificadores de aquellas variantes que no hayan superado todas las demandas de filtrado del mismo grupo.

Por ejemplo si una demanda de filtrado requiere las variantes del gen NF1, otra demanda requiere las variantes con una frecuencia poblacional menor a 0,1 y las dos demandas se encuentran en el grupo de filtrado 1, se eliminaran los identificadores de las variantes que no estén anotadas con el gen NF1 y no tengan una frecuencia

poblacional menor a 0,1. En cambio, si cada demanda de filtrado está en un grupo diferente se devolverán todas las variantes anotadas con el gen NF1 y todas las variantes con una frecuencia poblacional menor a 0,1.

Al final la función *computeMultipleQuery* devuelve los identificadores de las variantes que han pasado el sistema de filtrado. Este sistema de filtrado permite una gran versatilidad ya que el usuario puede combinar los filtros de forma que pueda conseguir exactamente las variantes que responden a las preguntas que tenga. Además la estructura de la función permite que se puedan añadir más filtros de forma sencilla dado que las funciones cortas de filtrado son independientes entre ellas.

Las demandas de filtrado se introducen en la herramienta mediante la sección *Add filters* de la aplicación. En esta sección el campo a filtrar aparece como un desplegable en el que se puede seleccionar cualquier campo del fichero VCF. La información sobre los campos se obtiene mediante el escaneado previamente descrito. Los filtros y el grupo del filtro también se seleccionan mediante un desplegable. El valor aplicable al filtro se aporta mediante una entrada de texto (Figura 5).

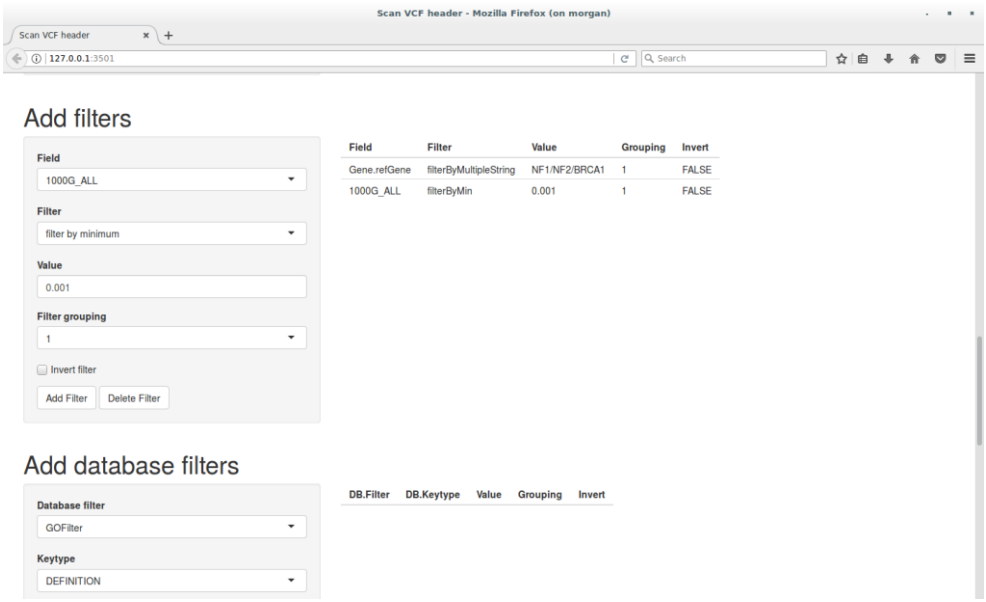


Figura 5: Filtrado

Cabe destacar que existe un proceso de control de valor del filtro antes de la ejecución para comprobar que se corresponda con el filtro a usar, es decir, si se usa un filtro de intervalo se comprueba que efectivamente el valor entrado es un intervalo separado por un guion. Una vez definida la demanda de filtrado si el usuario hace clic en el botón de *Add filter* esta se añade a la tabla donde se almacenan todas las demandas de filtrado. Mediante el botón *Delete filters* se elimina la última demanda de filtrado.

2.6 Bases de datos

A veces la información contenida en un fichero VCF no es suficiente para que el usuario pueda obtener las respuestas que busca. Para responder a esta necesidad la herramienta permite al usuario interactuar con las diferentes bases de datos, descritas en el apartado herramientas bioinformáticas de la introducción. La herramienta puede interactuar con las bases de datos de dos maneras:

2.6.1 Filtrado usando bases de datos

Para que el usuario pueda usar la información contenida en las bases de datos para filtrar se ha desarrollado una función general de interacción con las bases de datos. Esta función utiliza en función del requerimiento del usuario funciones más pequeñas cada una de las cuales ha sido diseñada especialmente para interactuar con una base de datos concreta.

Por razones de memoria y por el tipo de interacción con las bases de datos, solo se pueden usar para filtrar las bases de datos que estructuran la información por genes, como se ha descrito en la introducción. Este tipo de bases de datos estructuran la información que contienen en diferentes campos o *keytypes*. Dentro de cada *keytype* se almacenan las diferentes tipologías o *keys*. Cada *key* almacena a su vez el nombre de los genes que están relacionado con ella. La herramienta buscará las *keys* de cada *keytype* que contengan la palabra clave o valor que el usuario haya introducido y devolverá el nombre de los genes que contenga.

Por ejemplo, la base de datos *Gene Ontology* (GO) tiene una *keytype* llamada *TERM*. En esta *keytype* se almacenan todos los términos de GO. Si el usuario ha buscado en esta *keytype* la palabra cáncer, se devolverá el nombre de todos los genes que estén dentro de una *key* que contenga la palabra cáncer.

El nombre de los genes obtenidos se añade como una entrada más del proceso de filtrado descrito en el apartado anterior. En el campo nombre del gen se aplica el filtro *filterByMultipleString* de forma que cada elemento separado por una barra es el nombre de un gen. Como el filtrado por bases de datos tiene una relación directa con el sistema de filtrado también se puede seleccionar en que grupo de filtrado se quiere añadir cada filtrado por bases de datos, haciendo posible el uso conjunto de filtros y filtros por base de datos. Cabe destacar que es necesario que las variantes estén anotadas con el nombre del gen para poder usar el filtro por bases de datos.

Las demandas de filtrado por bases de datos se introducen mediante la sección *Add database filters* de la aplicación. En este apartado se puede seleccionar la base de datos y la *keytype* a utilizar por medio de un desplegable e introducir el texto o valor a buscar. Después se ha de seleccionar el grupo de filtros en el que se quiere añadir y clicar en el botón *Add database filter* (Figura 6).

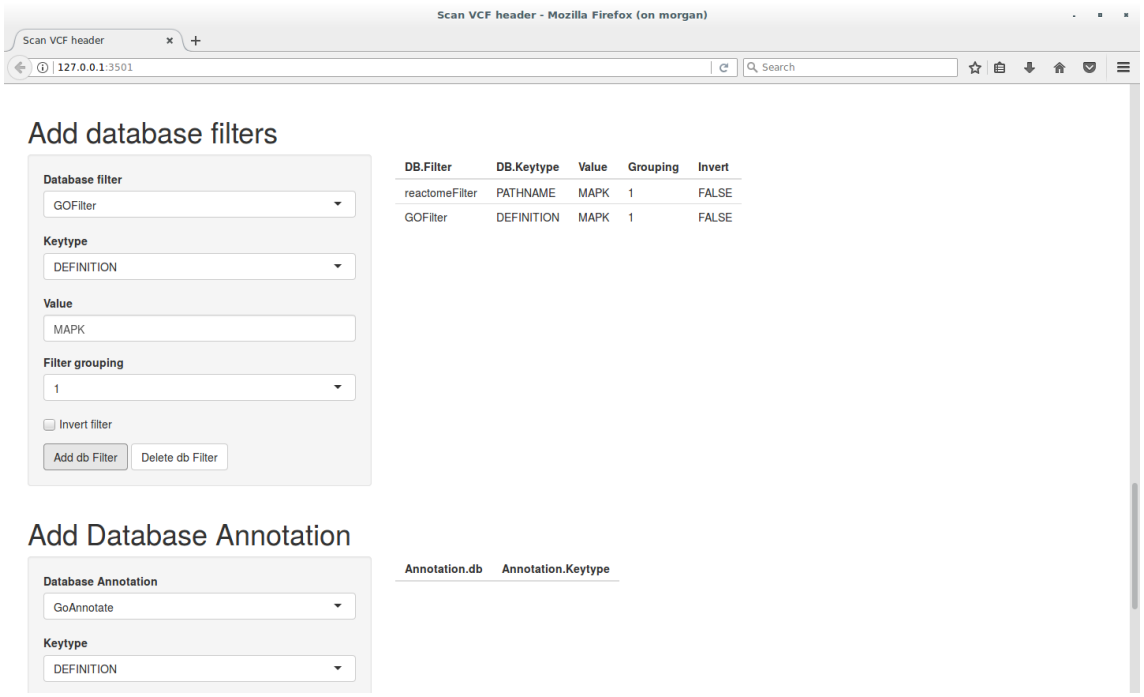


Figura 6: Filtrado con base de datos

2.6.2 Anotación usando bases datos

A parte de filtrar usando bases de datos al usuario también le puede interesar añadir información de estas bases de datos. Igual que en el aparatado anterior, se ha diseñado una función general de anotación que utiliza funciones concretas para cada base de datos. La herramienta permite al usuario anotar de dos maneras diferentes, en función del nombre del gen con el que está anotada la variante, o en función del cambio de aminoácido que origina la variante. En los dos casos la demanda de anotación se incluye como una columna en el archivo de salida de la herramienta, una vez se ha realizado el filtrado.

En el caso de la anotación por nombre de gen se usa un sistema parecido al descrito en el ejemplo del apartado anterior, pero en vez de buscar una palabra clave en las diferentes *keys* se utiliza el nombre del gen para buscar aquellas *keys* que tienen relación con el nombre del gen. Todas la *keys* encontradas por cada gen son incluidas en una columna extra en el documento de salida, separadas por una barra.

En el caso de la anotación por cambio de aminoácido se han descargado las bases de datos de *Uniprot* y *Cosmic* para poder realizar consultas de forma local. El método de interacción de las bases de datos utiliza el mecanismo de interacción de R con el sistema operativo para buscar dentro de estas bases de datos, utilizando el comando

grep. Primero se buscan coincidencias entre el nombre de gen anotado en la base de datos con el nombre del gen con el que se ha anotado la variante. Después se busca si en la posición de cambio de aminoácido anotado para la variante existe alguna región de interés, como por ejemplo dominios proteicos, descrita en la base de datos. Se construye una columna de anotación con los resultados encontrados que se añade al archivo de salida.

La selección de la anotación que se quiere añadir al archivo de salida se hace mediante la sección *Add annotation* de la aplicación (Figura 7). En esta sección se puede seleccionar que base de datos se quiere usar para anotar mediante un desplegable. Las bases de datos disponibles han sido descritas en el apartado de herramientas bioinformáticas. Una vez seleccionada la base de datos se puede seleccionar el *keytype* que se quiere usar para anotar.

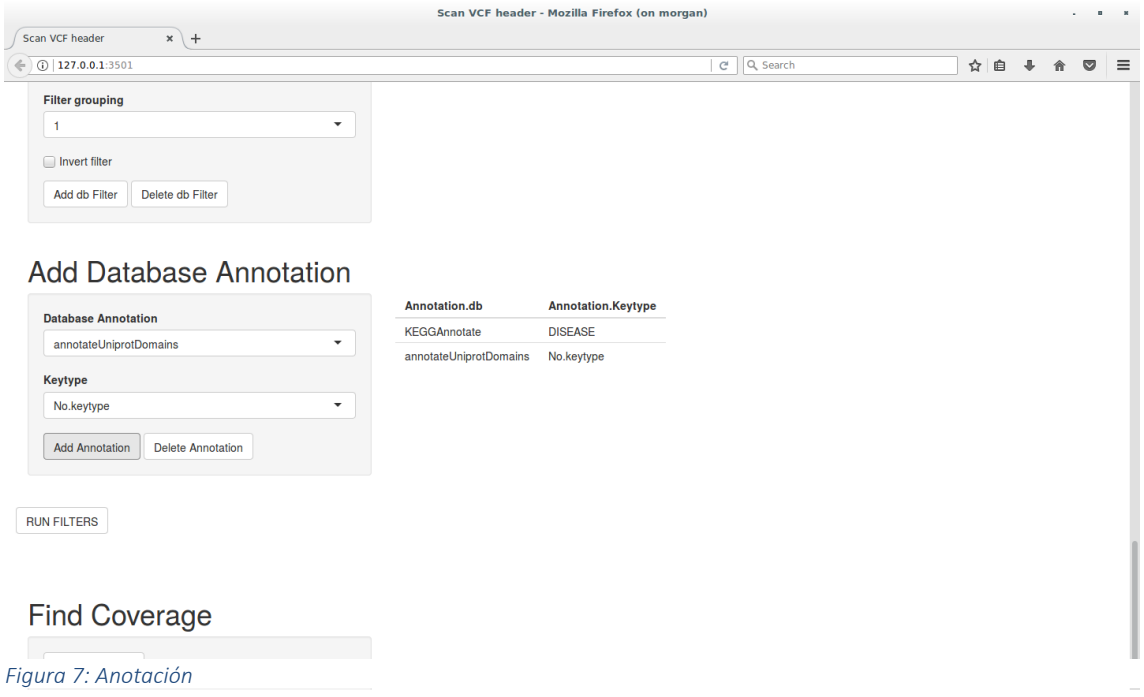


Figura 7: Anotación

A parte de la anotación usando bases de datos también es posible añadir una columna de anotación con la frecuencia alélica. Esta frecuencia se calcula a partir del campo *allelic depth* de la parte de información genotípica del fichero VCF. Esta anotación ayuda al usuario a saber con qué frecuencia se ha encontrado una variante en el proceso de secuenciado, facilitando la interpretación de los datos.

2.7 Ejecución y Exportación

Una vez el usuario ha seleccionado el tipo de ejecución, las preferencias, los filtros y las anotaciones que ha de usar la herramienta, clica en el botón *run* para empezar la ejecución (Figura 7). El proceso de ejecución empieza por la creación de un directorio temporal. En caso de varias ejecuciones en paralelo se crea un directorio individual para cada ejecución. En este directorio temporal se almacena toda la información que el usuario ha introducido en la aplicación mediante un fichero en formato *yaml*. Una vez almacenado el fichero *yaml* se usa una función general de ejecución que llevará a cabo la interpretación del fichero *yaml*. La función de ejecución comprueba que todos los requerimientos realizados por el usuario son adecuados para el normal funcionamiento del filtrado y la anotación y genera una matriz de datos con los requerimientos de filtrado y anotación. En el caso que haya requerimientos de filtrado por bases de datos, se realiza la consulta a las bases de datos y se añade el nombre de los genes a la matriz de datos de filtrado, como se ha explicado en el apartado de filtrado usando bases de datos. Este procesamiento previo ayuda a detectar posibles incongruencias en la ejecución y aporta al usuario mensajes de error explicando donde que parte de la información introducida por este contiene errores.

Cuando acaba todo el procesamiento de la información introducida por el usuario, se procede a usar una de las tres funciones de ejecución descritas en el apartado de tipos de ejecución. Como ya se ha explicado, estas funciones realizan el filtrado y la anotación de las variantes de formas diferentes, no obstante, comparten algunos elementos. El proceso de *variant calling,* descrito en la introducción, puede devolver variantes que no están presentes en el fichero BAM usado, pero si en otros ficheros BAM usados para generar el algoritmo para identificar las variantes. En este caso las variantes se describen con un genotipo 0/0. Todas las formas de ejecución realizan un prefiltrado para eliminar estas variantes del fichero. Esto hace que el total de variantes a gestionar sea menor, reduciendo el uso de memoria. Una vez realizado el prefiltrado se procede a indexar el fichero VCF para poder acceder a las variantes en función de sus coordenadas genómicas de forma sencilla.

El archivo VCF, prefiltrado e indexado, es usado para realizar el filtrado de variantes genómicas. Como se ha descrito en el apartado de filtrado, al acabar dicho proceso se obtiene un identificador único por cada variante que ha pasado los filtros. Este

identificador se usa entonces para generar el archivo de salida. Se carga en la memoria toda la información de las variantes usando el paquete *VariantAnnotation*. Se da formato a la información de cada variante agrupándola en celdas de matriz, donde cada celda contiene la información correspondiente a un campo del fichero VCF. Después de este proceso se obtiene una matriz de datos donde cada fila corresponde a la información de una variante y cada columna a un campo del fichero VCF. Este proceso puede llegar a ser largo dado que, a diferencia del filtrado, la cantidad de información que se carga y procesa para darle formato puede llegar a ser muy grande.

Una vez obtenida la matriz con los datos de todas las variantes que han pasado los filtros se procede a anotar las variantes. Por cada anotación requerida por el usuario se añade una columna al final de la matriz de datos con la información requerida de la base de datos.

Las matrices de datos obtenidas al final de todo el proceso realizado por la herramienta, se almacenan en el directorio de salida proporcionado por el usuario, en un formato de datos separados por tabulación. Este formato de datos puede ser consultado de manera sencilla usando cualquier programa de hoja de cálculo.

2.8 Zonas de cobertura de interés

El hecho que el pipeline de análisis descrito en la introducción no encuentre una variante genómica no quiere decir que esta no exista. Se puede dar el caso que se hayan realizado demasiadas pocas lecturas durante el proceso de secuenciación. En este cado el proceso bioinformático de análisis no podrá encontrar las variantes.

A parte del filtrado y anotación de ficheros VCF la herramienta también permite comprobar la cobertura de secuenciación de zonas de interés. Para realizar este proceso se ha diseñado una función que utiliza el paquete *BamSignals*, de *Bioconductor*. Este paquete permite la comparación de coordenadas genómicas con las posiciones genómicas de los *reads* de un fichero BAM. En el directorio de la aplicación están almacenadas las coordenadas genómicas de todos los exones, anotados de forma canónica por la universidad de California Santa Clara. Las coordenadas génomicas de los exones y las de los *reads* se comparan y se devuelve una tabla describiendo que nucleótidos de cada exón están cubiertos por 20 *reads* o menos. Esta tabla permite al

usuario saber que regiones de su interés están o no cubiertas por el proceso de secuenciación.

Para llevar a cabo este proceso el usuario ha de introducir en la aplicación la dirección al fichero BAM a partir del cual ha obtenido las variantes del fichero VCF y un directorio donde almacenar la tabla de salida. Entonces puede clicar en el botón *Get coverage* y obtendrá una tabla con las zonas de interés poco cubiertas. Esta tabla se mostrará en la aplicación y se almacenar en el directorio de salida (Figura 8).

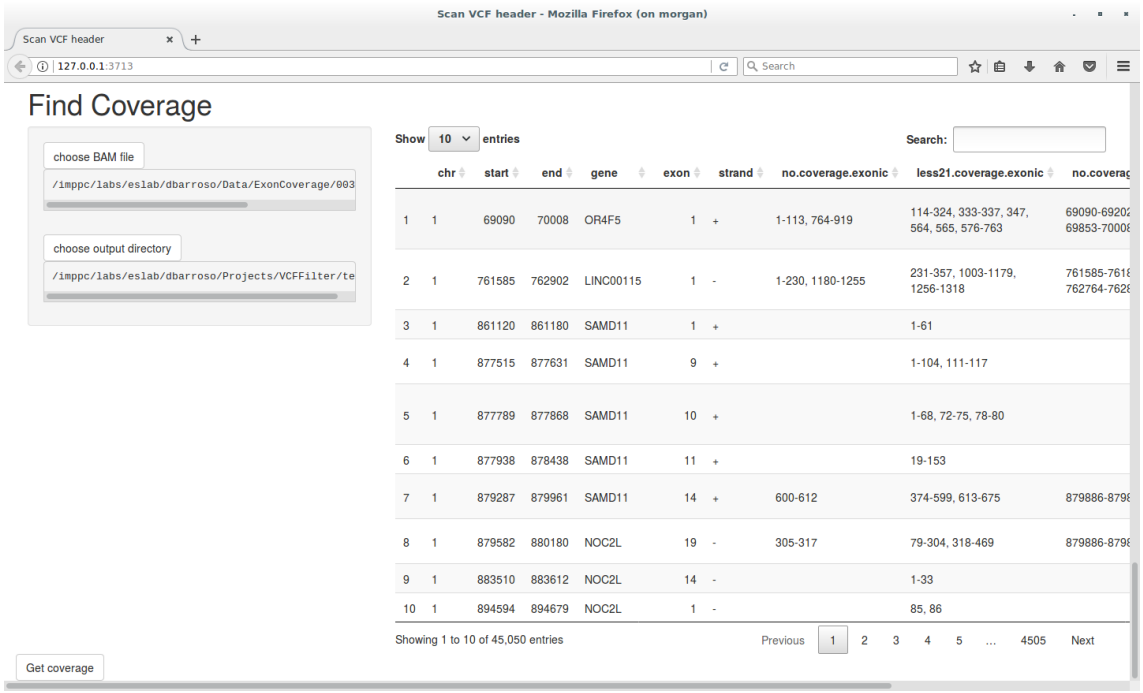


Figura 8: Cobertura de zonas de interés

3. Ejemplo de validación

3.1 Dataset

Para testar el funcionamiento de la herramienta se han usado 9 ficheros VCF. Estos ficheros proceden de la secuenciación mediante *illumina* de 9 muestras de *schwannomas*, tumores que se desarrollan en las células de *Schwan*. Los archivos procedentes de secuenciación, se han alineado con el genoma de referencia hg37 mediante el algoritmo BWT para obtener los ficheros BAM. A partir de los ficheros BAM se ha realizado el *variant calling* utilizando STRELKA y se han anotado usando *ANNOVAR*. Los ficheros utilizados tienen estos nombres: *003TG.annotated.vcf, 009TG.annotated.vcf, 0025TG.annotated.vcf, 0048TG.annotated.vcf,*

0106TG.annotated.vcf, 0107T2G.annotated.vcf, 0145TG.annotated.vcf, 0245TG.annotated.vcf, 0435TG.annotated.vcf.

3.2 Ejecución

Para validar la herramienta se han realizado varios filtrajes en los que se espera obtener el mismo resultado. Se selecciona aleatoriamente un grupo de variantes y se aplican diferentes filtros que han de devolver dichas variantes. Un ejemplo de este mecanismo de validación es el siguiente:

Selección aleatoria de variantes:

Ficheros	Chr	Start	End	Referencia	Nucleótido Alternativo	Gene	Aminoacid Change
0025TG.annotated.vcf, 009TG.annotated.vcf, 0106TG.annotated.vcf, 0107T2G.annotated.vc, 0145TG.annotated.vcf, 0245TG.annotated.vcf	11	7818383	7818383	G	GATATGGTTACCAGGTAGATGC	OR5P2	S36delinsCIYLVTIS
009TG.annotated.vcf	17	29679361	29679361	G	A	NF1	S2494N
003TG.annotated.vcf	22	30069405	30069405	C	T	NF2	R424C

Filtrajes definidos:

1: Búsqueda por nombre de gen

Ejecución: *Batch Working*

Demandas de filtrado

Campo	Filtro	Valor	Grupo	Inversión
Gene.refGene	Filter by full string	OR5P2	1	No
Gene.refGene	Filter by full string	NF1	2	No
Gene.refGene	Filter by full string	NF2	3	No

2: Búsqueda por profundidad alélica

Ejecución*: Compare Files*

Demandas de filtrado

Campo	Filtro	Valor	Grupo	Inversión
AAChange.refGene	Filter by containing string	S36delinsCIYLVTIS	1	No
AAChange.refGene	Filter by AD interval	S2494N	2	No
AAChange.refGene	Filter by AD interval	R424C	3	No

3: Búsqueda por localización genómica

Ejecución: *Compare Files*

Genomic Ranges

Chr	Start	End
11	7800000	7900000
17	29600000	29700000
22	30000000	30100000

En los tres casos se usan las preferencias de ejecución predeterminadas y no se añade anotación.

3.3 Resultado

Después de los tres procesos de filtrado se obtienen las variantes filtradas. Se observa que las variantes elegidas para realizar el control están presentes en los tres casos (Figuras 9-11). Para las variantes filtradas usando el método de ejecución *batch working* se obtiene un fichero por cada muestra. En el caso de usar el método *compare files* se obtiene un solo fichero por todas las variantes.

Este proceso se ha repetido combinando las diferentes opciones que ofrece la herramienta para poder validar el funcionamiento de cada una de ellas.

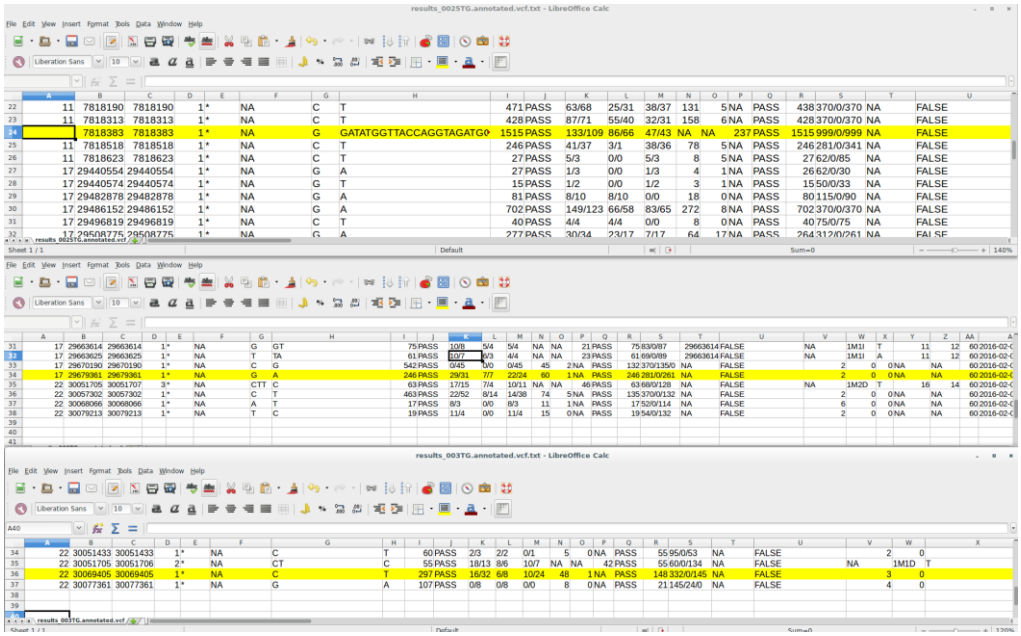


Figura 9: Variantes filtradas por nombre de gen

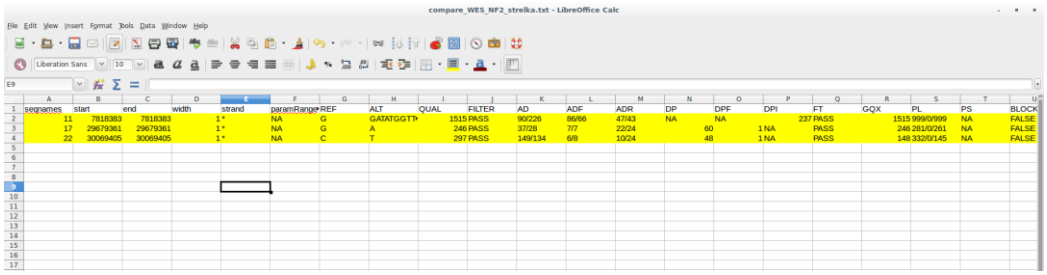


Figura 10: Variantes filtradas por cambio de aminoácido

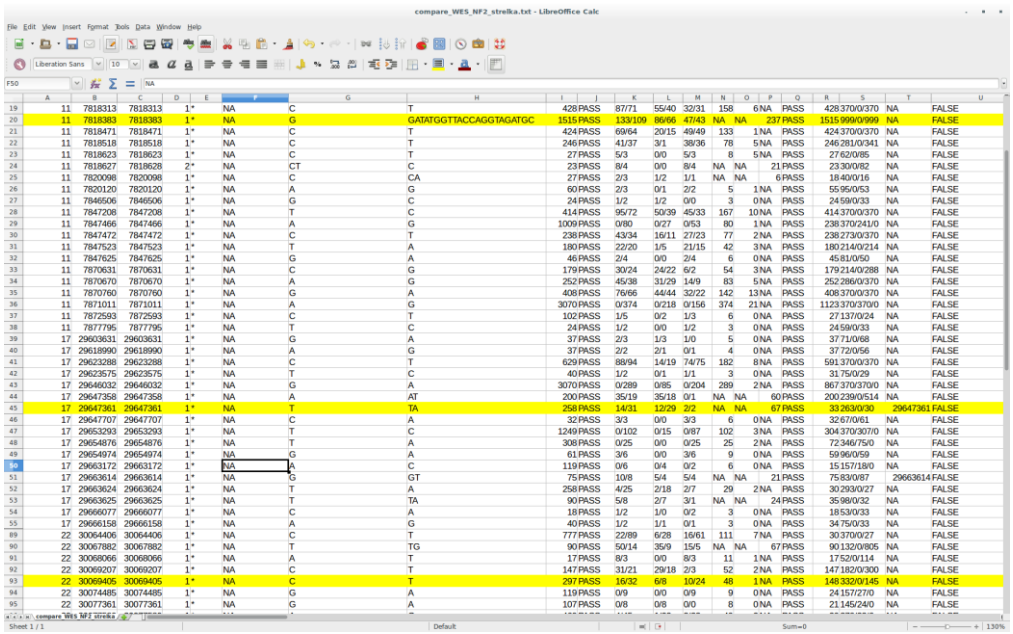


Figura 11: Filtrado por localización genómica

4. Conclusiones

Al final de todo el proceso de desarrollo se ha obtenido una herramienta completamente funcional, con una interfaz con el usuario sencilla e intuitiva. La herramienta permite al usuario escanear, filtrar, filtrar y anotar usando bases de datos y crear archivos de salida leíbles a partir de ficheros VCF. La herramienta se adapta eficazmente a cualquier estructura de archivo VCF con el que interaccione. El sistema de filtrado implementado es versátil y permite estructurar los requerimientos de filtrado de manera que se obtengan las variantes que respondan a las preguntas del usuario. La interacción con las bases de datos se realiza de forma efectiva permitiendo disponer de la información que estas ofrecen. La herramienta dispone de un formato de salida útil para el usuario. Además se ha podido incorporar un sistema de comprobación de cobertura de zonas de interés para complementar la herramienta.

El proceso de desarrollo de la herramienta se ha llevado a cabo siguiendo la planificación temporal propuesta. No ha existido ninguna desviación temporal significante. El proceso de desarrollo de funciones para la interacción con bases de datos ha resultado ser el más difícil. Cada base de datos presenta una organización diferente y ha resultado complicado desarrollar un mecanismo de interacción generalizado. No obstante, se ha podido añadir a la planificación del desarrollo la funcionalidad de cobertura de zonas de interés dado que la temporización era la correcta.

El *feedback* con los investigadores durante el desarrollo ha sido muy positivo y ha permitido adaptar la herramienta a las necesidades que se iban planteando. Esto ha permitido tener un producto final realmente útil para el proceso de investigación. Además la estructuración de las funciones propuesta en la metodología ha permitido añadir de forma sencilla nuevas funcionalidades a partir de las necesidades que han ido surgiendo durante el desarrollo.

Los futuros añadidos que se le podrían hacer a la herramienta son:

- Añadir mensajes de ayuda en la aplicación para facilitar su uso por parte del usuario. La herramienta presenta muchos campos y funcionalidades diferentes, por lo tanto, su uso puede resultar un poco confuso al principio.

- Mejorar el seguimiento de errores. Cuando sucede un error en la aplicación algunos mensajes de error no son lo suficientemente explicativos.
- Dar una estructura diferente a las bases de dados descargadas para que el proceso de consulta sea más eficiente. Para consultar las bases de datos *Uniprot* y *Cosmic* tienen una estructura lineal que dificulta las búsquedas. Realizar un proceso de pre-formateado podría ayudar a reducir los tiempos de ejecución de la anotación

5. Glosario

VCF: Variant Call Format

NGS: Next Generation Sequencing

ADN: Ácido Desoxidorribonucleico

PCR: Polimerase Chain Reaction

SNV: Single Nucleotide Variant

SAM: Sequence Alingment Map

BAM: Binary Alingment Map

GO: Gene Ontology

KEGG: Kyoto Encyclopedia of Genes and Genomes

MMR: Mismatch Repair

BWT: Burrows-Wheeler transform

6. Bibliografía

1. Reich DE, Schaffner SF, Daly MJ, McVean G, Mullikin JC, Higgins JM, et al. Human genome sequence variation and the influence of gene history, mutation and recombination. Nat Genet. 2002;32(1):135–42.
2. Definition of genetic variant - NCI Dictionary of Genetics Terms - National Cancer Institute [Internet]. [cited 2018 May 28]. Available from: https://www.cancer.gov/publications/dictionaries/genetics-dictionary/def/genetic-variant
3. Types of genetic variation | EMBL-EBI Train online [Internet]. [cited 2018 May 28]. Available from: https://www.ebi.ac.uk/training/online/course/human-genetic-variation-i-introduction/what-genetic-variation/types-genetic-variation
4. Herr AJ, Williams LN, Preston BD. NIH Public Access. 2013;46(6):548–70.
5. Lee J-M, Gillis T, Mysore JS, Ramos EM, Myers RH, Hayden MR, et al. Common SNP-Based Haplotype Analysis of the 4p16.3 Huntington Disease Gene Region. 2012 [cited 2018 Jun 2];
6. Astermark J, Oldenburg J, Carlson J, Pavlova A, Kavakli K, Berntorp E, et al. Polymorphisms in the TNFA gene and the risk of inhibitor development in patients with hemophilia A. [cited 2018 Jun 2];
7. Heather JM, Chain B. The sequence of sequencers: The history of sequencing DNA. Genomics [Internet]. 2016;107(1):1–8.
8. Sanger F, Nicklen S, Coulson a R. DNA sequencing with chain-terminating inhibitor. Proc Natl Acad Sci USA. 1977;74(12):5463–7.
9. Shendure J, Ji H. Next-generation DNA sequencing. Nat Biotechnol. 2008;26(10):1135–45.
10. Venter JC, Adams MD, Myers EW, Li PW, Mural RJ, Sutton GG, et al. The sequence of the human genome. Science (80-) [Internet]. 2001;291(5507):1304–51.
11. Meyer M, Kircher M. Illumina sequencing library preparation for highly multiplexed target capture and sequencing. Cold Spring Harb Protoc. 2010;5(6).
12. Voelkerding K V., Dames SA, Durtschi JD. Next-generation sequencing:from basic research to diagnostics. Clin Chem. 2009;55(4):641–58.
13. Mardis ER. A decade's perspective on DNA sequencing technology. Nature

[Internet]. 2011;470(7333):198–203.

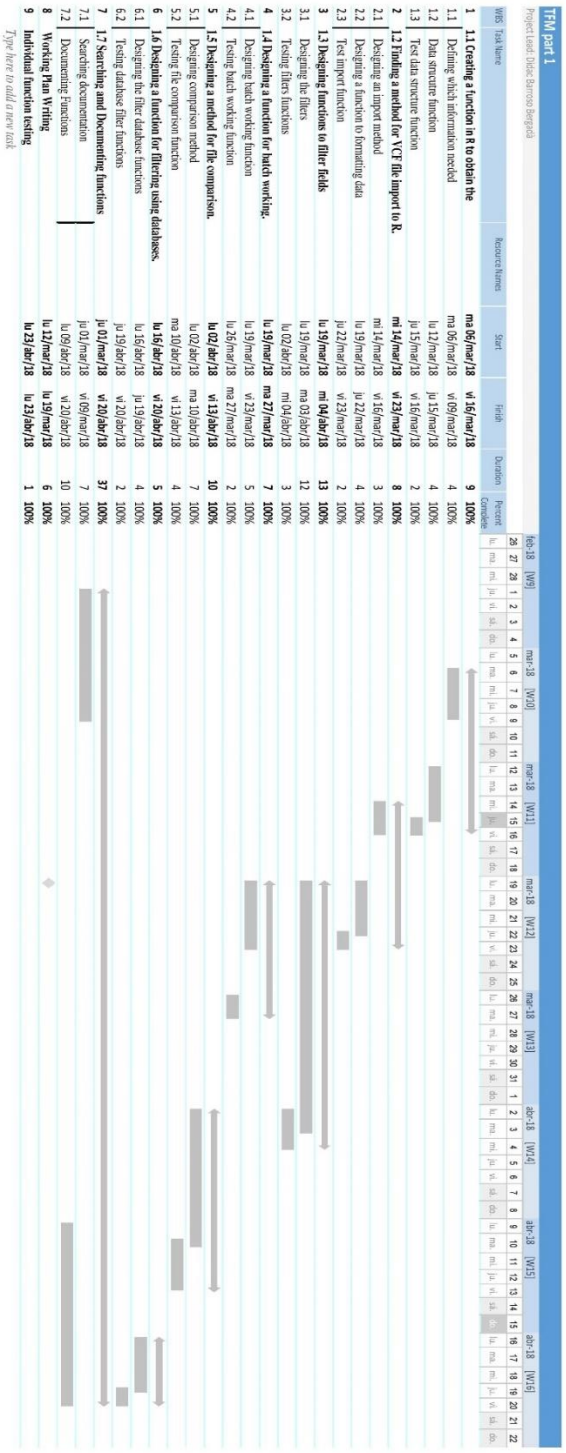
14. Wong KH, Jin Y, Moqtaderi Z. Multiplex Illumina Sequencing Using DNA Barcoding. In: Current Protocols in Molecular Biology [Internet]. Hoboken, NJ, USA: John Wiley & Sons, Inc.; 2013 [cited 2018 Jun 2].
15. Cock PJA, Fields CJ, Goto N, Heuer ML, Rice PM. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. Nucleic Acids Res. 2009;38(6):1767–71.
16. Lam TW, Li R, Tam A, Wong S, Wu E, Yiu SM. High throughput short read alignment via bi-directional BWT. 2009 IEEE Int Conf Bioinforma Biomed BIBM 2009. 2009;31–6.
17. Li H, Handsaker B, Wysoker A, Fennell T, Ruan J, Homer N, et al. The Sequence Alignment/Map format and SAMtools. Bioinformatics. 2009;25(16):2078–9.
18. Saunders CT, Wong WSW, Swamy S, Becq J, Murray LJ, Cheetham RK. Strelka: Accurate somatic small-variant calling from sequenced tumor-normal sample pairs. Bioinformatics. 2012;28(14):1811–7.
19. Xu C. A review of somatic single nucleotide variant calling algorithms for next-generation sequencing data. Comput Struct Biotechnol J [Internet]. 2018;16:15–24.
20. Danecek P, Auton A, Abecasis G, Albers CA, Banks E, DePristo MA, et al. The variant call format and VCFtools. Bioinformatics. 2011;27(15):2156–8.
21. Home of variant tools | RefGene [Internet]. [cited 2018 Jun 2]. Available from: http://varianttools.sourceforge.net/Annotation/RefGene
22. Debian -- El sistema operativo universal [Internet]. [cited 2018 May 28]. Available from: https://www.debian.org/
23. R: The R Project for Statistical Computing [Internet]. [cited 2018 May 28]. Available from: https://www.r-project.org/
24. Wickham H, Hester J, Müller K, Cook D, Maintainer]. Title Memoisation of Functions. 2017 [cited 2018 May 28]; Available from: https://github.com/hadley/memoise
25. CRAN - Package yaml [Internet]. [cited 2018 May 28]. Available from: https://cran.r-project.org/web/packages/yaml/index.html
26. tools package | R Documentation [Internet]. [cited 2018 May 28]. Available from:

https://www.rdocumentation.org/packages/tools/versions/3.5.0

27. Shiny [Internet]. [cited 2018 May 28]. Available from: https://shiny.rstudio.com/
28. Gentleman RC, Carey VJ, Bates DM, Bolstad B, Dettling M, Dudoit S, et al. Bioconductor: open software development for computational biology and bioinformatics. Genome Biol [Internet]. 2004 Sep 15 [cited 2018 May 28];5(10):R80.
29. Obenchain V, Lawrence M, Carey V, Gogarten S, Shannon P, Morgan M. VariantAnnotation: a Bioconductor package for exploration and annotation of genetic variants. Bioinformatics [Internet]. 2014 Jul 15 [cited 2018 May 28];30(14):2076–8.
30. Gel B, Díez-Villanueva A, Serra E, Buschbeck M, Peinado MA, Malinverni R. regioneR: an R/Bioconductor package for the association analysis of genomic regions based on permutation tests. Bioinformatics [Internet]. 2015 Sep 30 [cited 2018 May 28];32(2):btv562.
31. Mammana A HJ. bamsignals: Extract read count signals from bam files. R package version 1.12.0. 2018.
32. Morgan M. AnnotationHub: Client to access AnnotationHub resources. R package version 2.12.0. 2018.
33. Carlson M. GO.db: A set of annotation maps describing the entire Gene Ontology. R package version 3.6.0. 2018.
34. Ligtenberg W. reactome.db: A set of annotation maps for reactome. R package version 1.64.0. 2018.
35. Liberzon A, Subramanian A, Pinchback R, Thorvaldsdottir H, Tamayo P, Mesirov JP. Molecular signatures database (MSigDB) 3.0. Bioinformatics [Internet]. 2011 Jun 15 [cited 2018 May 28];27(12):1739–40.
36. Tenenbaum D. KEGGREST: Client-side REST access to KEGG. R package version 1.20.0. 2018.
37. UniProt [Internet]. [cited 2018 May 28]. Available from: http://www.uniprot.org/
38. COSMIC | Catalogue of Somatic Mutations in Cancer [Internet]. [cited 2018 May 28]. Available from: https://cancer.sanger.ac.uk/cosmic

7. Anexos

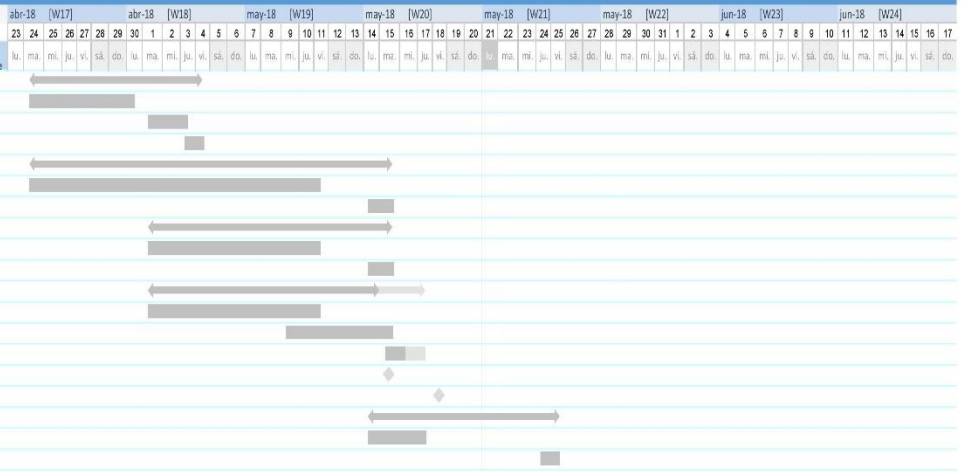
ANNEXO 1: Diagramas de Gantt



TFM part 2

Project Lead: Didac Barroso

WBS	Task Name	Resource Names	Start	Finish	Duration	Percent Complete
1	**2.1 Designing interaction function**		**ma 24/abr/18**	**vi 04/may/18**	**9**	**100%**
1.1	Designing a function for interaction		ma 24/abr/18	lu 30/abr/18	5	100%
1.2	Integrating the function in tool		ma 01/may/18	ju 03/may/18	3	100%
1.3	Testing the interaction function		ju 03/may/18	vi 04/may/18	2	100%
2	**2.2 Integrate the functions**		**ma 24/abr/18**	**ma 15/may/18**	**16**	**100%**
2.1	Function integration		ma 24/abr/18	vi 11/may/18	14	100%
2.2	Testing function integration		lu 14/may/18	ma 15/may/18	2	100%
3	**2.3 Design a return function**		**ma 01/may/18**	**ma 15/may/18**	**11**	**100%**
3.1	Design output function		ma 01/may/18	vi 11/may/18	9	100%
3.2	Test output function		lu 14/may/18	ma 15/may/18	2	100%
4	**2.4 Documenting the tool**		**ma 01/may/18**	**ju 17/may/18**	**13**	**83%**
4.1	Integrating function documentations		ma 01/may/18	vi 11/may/18	9	100%
4.2	Writing a user guide		mi 09/may/18	ma 15/may/18	5	100%
4.3	Writing using examples		ma 15/may/18	ju 17/may/18	3	50%
5	Functional tool		ma 15/may/18	ma 15/may/18	1	100%
6	Tool documentation		vi 18/may/18	vi 18/may/18	1	100%
7	**Regions of Interest tool Design**		**lu 14/may/18**	**vi 25/may/18**	**10**	**100%**
7.1	Tool Design		lu 14/may/18	ju 17/may/18	4	100%
7.2	Tool testing		ju 24/may/18	vi 25/may/18	2	100%



Project Writting

Project Lead: Didac Barroso

WBS	Task Name	Resource Names	Start	Finish	Duration	Percent Complete
1	**3.1 Working Plan Writing**		**ma 06/mar/18**	**vi 16/mar/18**	**9**	**100%**
1.1	Defining objectives		ma 06/mar/18	vi 09/mar/18	4	100%
1.2	Defining tasks		lu 12/mar/18	ma 13/mar/18	2	100%
1.3	Scheduling tasks		mi 14/mar/18	mi 14/mar/18	1	100%
1.4	Writing document		ju 15/mar/18	vi 16/mar/18	2	100%
2	**3.2 Memory Writing**		**ma 01/may/18**	**vi 01/jun/18**	**24**	**17%**
2.1	Writing introduction		ma 01/may/18	vi 11/may/18	9	0%
2.2	Writing tool description		lu 14/may/18	mi 23/may/18	8	50%
2.3	Writing results and conclusions		mi 23/may/18	vi 01/jun/18	8	0%
3	**3.3 Virtual Presentation Design**		**mi 06/jun/18**	**lu 25/jun/18**	**14**	**0%**
3.1	Designing presentation		mi 06/jun/18	vi 08/jun/18	3	0%
3.2	Preparing presentation		lu 11/jun/18	mi 13/jun/18	3	0%
3.3	Performing Presentation		ju 14/jun/18	lu 25/jun/18	8	0%
4	Work Plan		lu 19/mar/18	lu 19/mar/18	1	100%
5	Memory writing		ma 05/jun/18	ma 05/jun/18	1	0%
6	Virtual presentation		lu 25/jun/18	lu 25/jun/18	1	0%

ANNEXO 2: Guía de Usuario

This document is a guide for using the VCF filter app.

Interface

A web app for user interaction with the VCF filter tool has been designed. In order to launch the app paste this code in a Linux terminal:

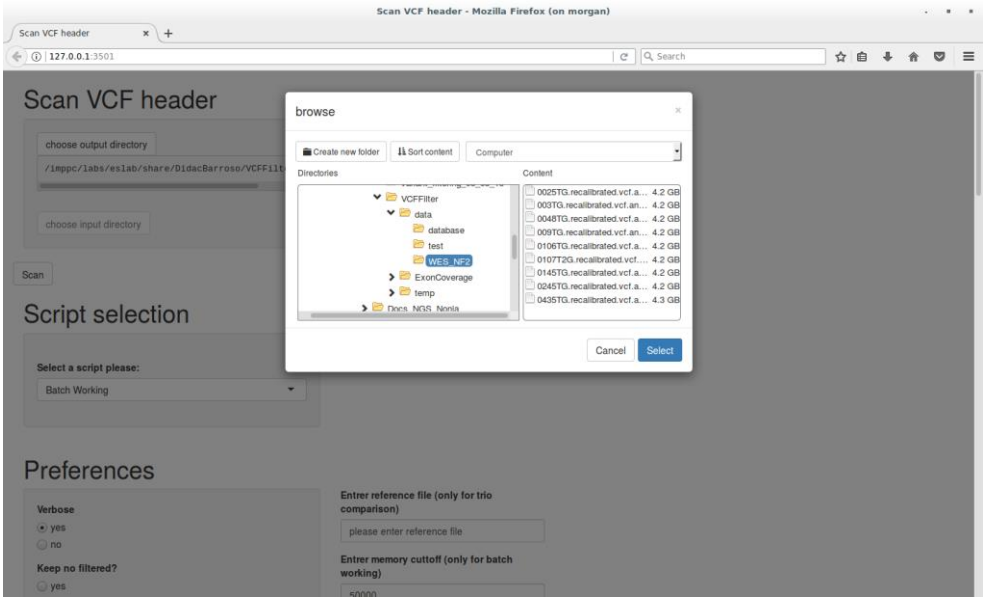
Rscript -e 'library(methods); shiny::runApp("/imppc/labs/eslab/share/DidacBarroso/VCFFilter/app.R", launch.browser=TRUE)'

The code opens a web browser page with the app. The app requires having the next R libraries installed:

shiny, shinyFiles, rhandsontable, DT, VariantAnnotation, regioneR, memoise, yaml, tools, AnnotationHub, GO.db, reactome.db, MSigDB, KEGGREST, plyr

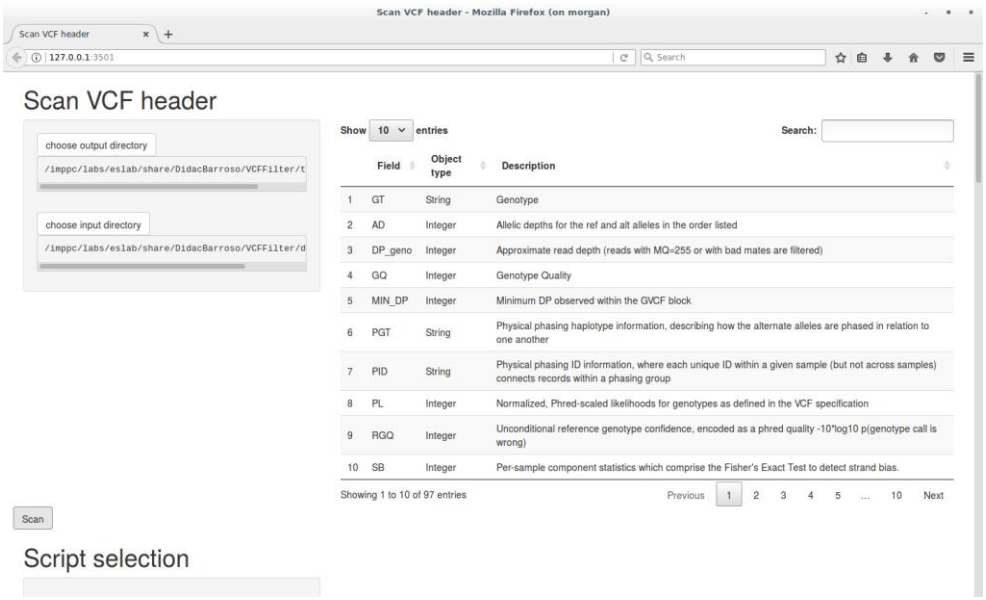
Folder selection

The first part of the app usage consists in the input and output folder selection. The input folder must contain at least one VCF file for filtering. All files in the input folder must have the same VCF format. The output folder is the place where the files answering the query are stored.



Scan

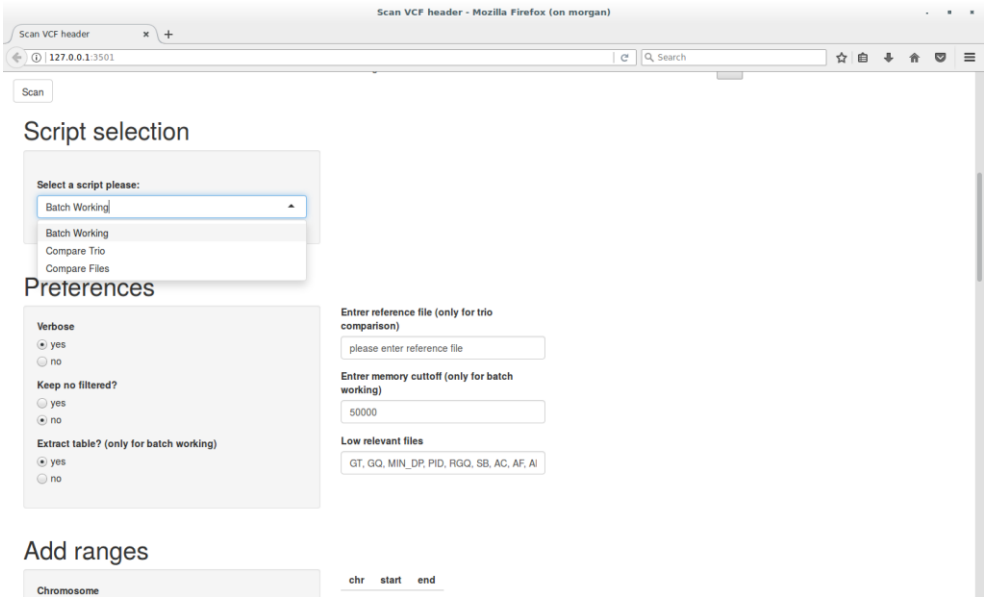
After folder selection user must click on the scan button. Pressing this button returns a table with all the fields present in the input folder VCF files. The user can see the description of each field in an interactive table for choosing properly the filter parameters. This table is also stored in the output folder.



Script selection

Once VCF files are scanned user can proceed to select the filtering parameters. The first choice user can make is the script used for filtering. Three scripts are available:

- batchWorkingFilter: filters and annotates each file separately.
- compareTrio: filters and annotates a reference file, checking if the existing variants are found in two progenitors files.
- compareMultipleFiles: filters and annotates all the variants present in multiple files. All variants are returned in a single file, annotating in which file is found every variant. The genotype data returned for each variant corresponds to the genotype of the first file where variant is found.



Preferences

Some parameters can be choose by the user in order to modulate the filtering process, however the predetermined values let work the script properly. Read documentation carefully before changing them:

Verbose

When YES option is selected, the messages explaining the process are appearing in the terminal.

Keep no filtered

Keep the variants which have not passed the filters but are in the same location as any variants which passed the filter. Default is No.

Extract table (only for batch working)

When YES option is selected, the variants are returned in a human readable table if NO is selected, variants are returned in a VCF file. This option is only available for batch working script, other scripts always are returning a human readable table.

Enter reference file (only for trio comparison)

When comparing trios a reference file is needed for filtering and comparison with progenitors

Enter memory cutoff (only for batch working)

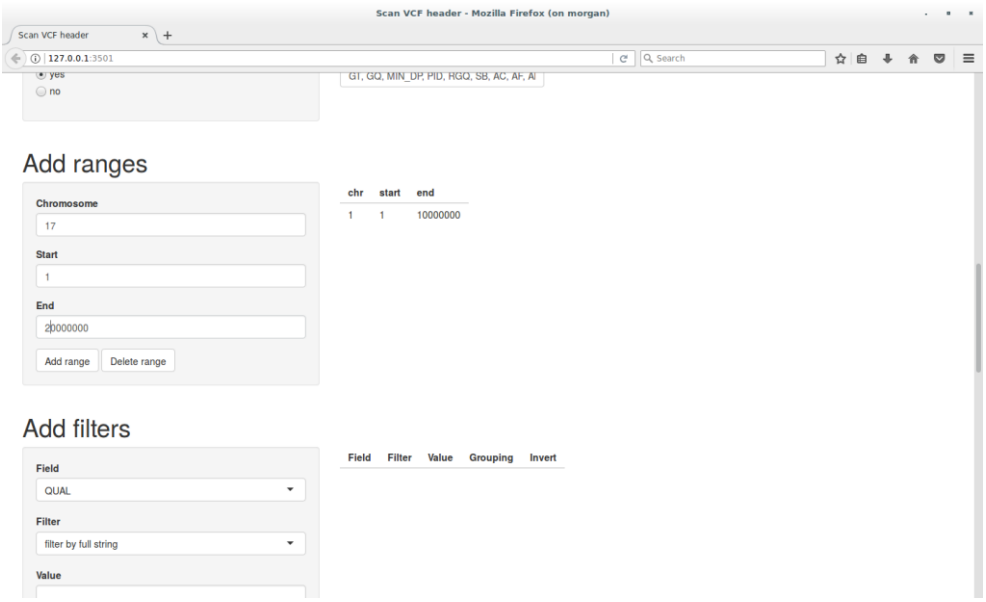
VCF files can have a huge amount of variants. Some computers may not be able to load the required data for filtering. Lowing the memory cutoff reduces the memory usage of the tool segmenting the data by gnomic regions. However the execution time is going to be bigger.

Low relevant fields

User can list here the fields in the VCF not relevant for his/her purposes. This fields are gonna be move to the right-most part of the output.

Limit Ranges

User can restrict the filtering to delimited genomic regions. In order to limit the regions filtered user have to write the chromosome name and the start and end genomic position and click add range. Added ranges can be deleted pressing the delete range button. It is possible to select all the genomic ranges the user need. Only variants present in the required genomic positions are returned. If no ranges are given tool returns the whole genome filtered variants.



Filters:

User can filter the variants of a VCF file using this section of the app. User can choose as many filter entries as desired.

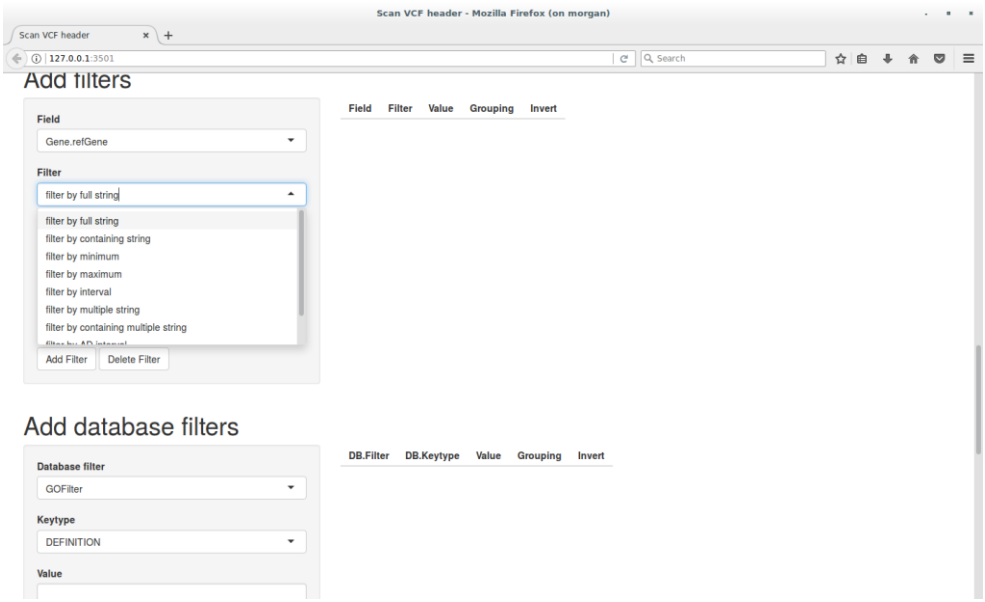
Field

After scanning, a scroll selection is available with all the fields present in the VCF file.

Filter

User can select a filter to apply to the selected field. 9 filters are available:

- filterByFullString: Finds if a given string matches the string in the selected field of each variant. A string value must be given.
- filterByContainingString: Finds if a given string is present in the selected field of each variant. A string value must be given.
- filterByMin: Finds if the numeric value of the selected field is over a given value. A numeric value must be given. The decimals must be determined by a dot.
- filterByMax: Finds if the numeric value of the selected field is below a given value. A numeric value must be given. The decimals must be determined by a dot
- filterByInterval: Finds if the numeric value of the selected field is between a given interval value. Interval must be defined following the next structure: "number-number", example: 1-10. The decimals must be determined by a dot
- filterByMultipleString: Finds if any of the given set string matches the string in the selected field of each variant. The set of strings must follow the next structure: "string/string/string", example: NF1/NF2/BRCA1. There is no limit on the number of given strings.
- filterByContainingMultipleString: Finds if any of the given set string are present in the string in the selected field of each variant. The set of strings must follow the next structure: "string/string/string", example: NF1/NF2/BRCA1. There is no limit on the number of given strings.
- filterByADInterval: This filter can only be used when AD (allelic depth) field is selected. The filter finds if the allelic depth is between a given interval. The AD interval value must follow the next structure: "number-number/number-number", example: 1-10/10-20.
- filterByADFreq: This filter can only be used when AD (allelic depth) field is selected. The filter obtains the allelic frequency from the AD (allelic depth) then checks if it is over a minimum. A numeric value must be given. The decimals must be determined by a dot



Value

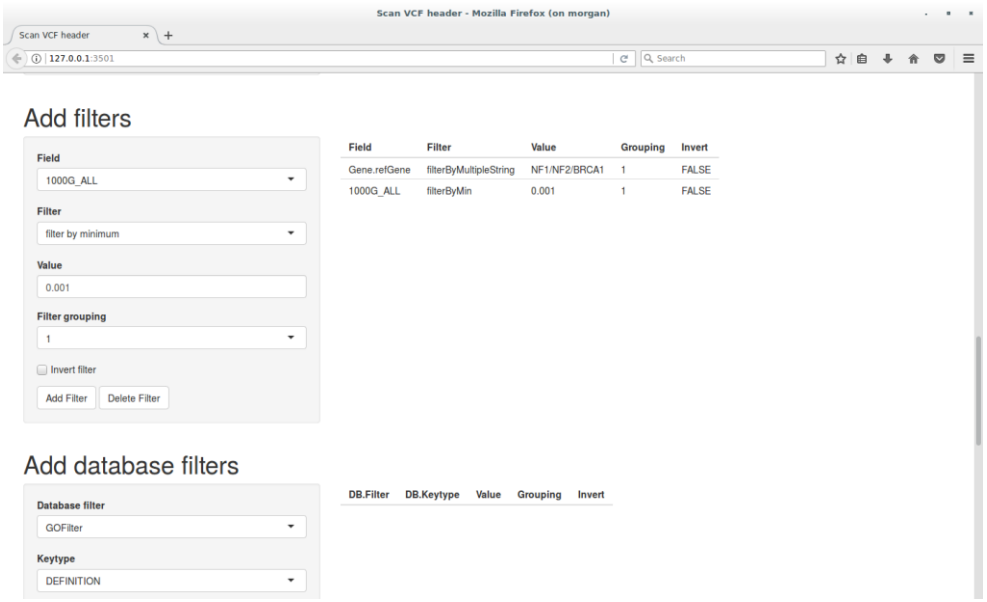
The string, number or interval to be applied to the filters. It must follow the structure defined for the chosen filter.

Filter Grouping

The groups of filters are defined by a number selected by scrolling. All the filter entries which share the same number are grouped together. Grouped filters are excluding between them, only variants which match all the filter entries grouped will be returned. Different filter groups are including, variants only need to pass one filter group to be returned.

Invert filter

The filter entry is inverted. The variants which don't fulfill the filter entry will be returned.



Database filtering

User can also filter variants using data present in different databases.

Database filter

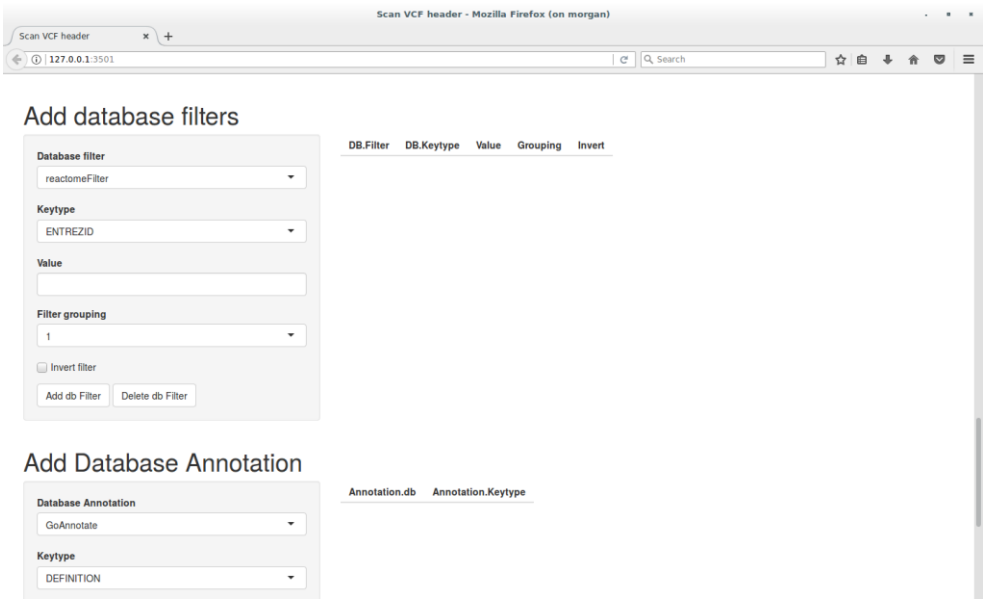
The method for database filtering requires a scroll selecting of the the database to be used. Four databases are avaiable: GO, reactome, MSigDB and KEGG.

Keytype

Keytypes are the different fields where data is stored in a database. User must select, if possible, one of the keytypes for filtering, the next keytypes are available:

- GOFilter: keytypes(DEFINITION, GOID, ONTOLOGY, TERM)
- reactomeFilter: keytypes(ENTREZID, GO, PATHID, PATHNAME, REACTOMEID)
- MSigDBFilter: keytypes(HALLMARK, C1_POSITIONAL, C2_CURATED, C3_MOTIF, C4_COMPUTATIONAL, C5_GENE_ONTOLOGY, C6_ONCOGENIC_SIGNATURES, C7_IMMUNOLOGIC_SIGNATURES)

- KEGGFilter: No keytypes

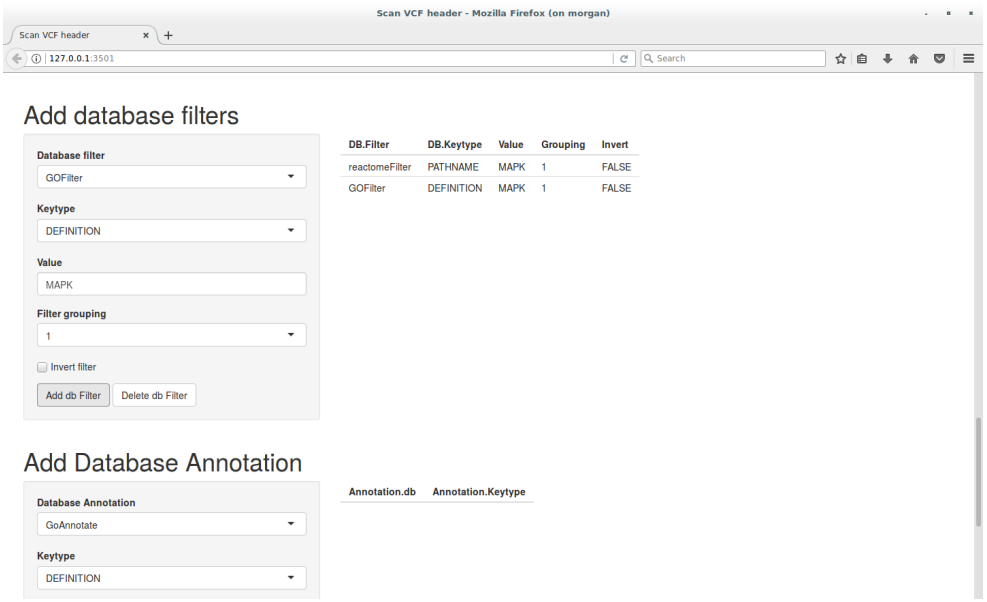


Value

The value to be searched in the selected keytype. The value only have to be present in the keytype, doesn't need to match a value of the keytype. In general entries have to be a string with keywords or database identifiers. The variants which are in genes related with that keytype value are returned.

Filter Grouping

Works together with the filter grouping explained before.



Database annotation

Variants which passed the filters can be annotated. Annotation can be made using the gene name of each variant or using the amino-acid change that each variant produces.

The following functions can be scrolling selected to annotate using the gene name: GOAnnotate, reactomeAnnotate, MSigDBAnnotate, KEGGAnnotate. The functions to annotate using the amino-acid changes are: annotateUniprotDomains, annotateUniprot, annotateCosmic.

Once function is selected an scroll for selection can be used to select the keytype to be used for annotation. The following keytypes can be used:

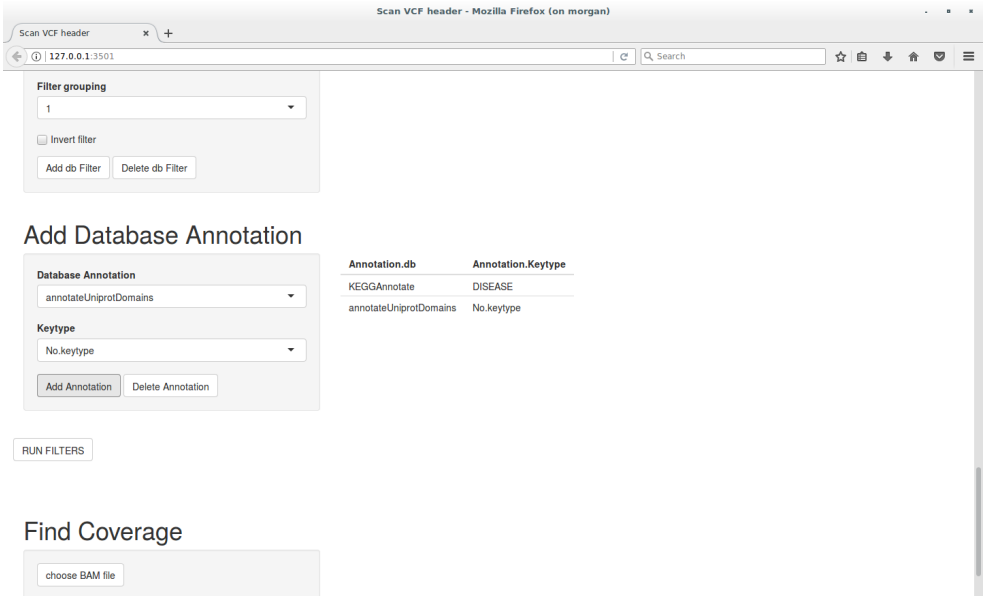
- GoAnnotate: keytypes(DEFINITION, GOID, ONTOLOGY, TERM)
- reactomeAnnotate: keytypes(ENTREZID, GO, PATHID, PATHNAME, REACTOMEID)

- MSigDBAnnotate: keytypes(HALLMARK, C1_POSITIONAL, C2_CURATED, C3_MOTIF, C4_COMPUTATIONAL, C5_GENE_ONTOLOGY, C6_ONCOGENIC_SIGNATURES, C7_IMMUNOLOGIC_SIGNATURES)
- KEGGAnnotate: PATHWAY, DISEASE, BRITE
- annotateUniprotDomains: No keytypes
- annotateUniprotFt: No keytypes
- annotateCosmic: keytypes(Gene name, Accession Number, Gene CDS length, HGNC ID, Sample name, ID_sample, ID_tumour, Primary site, Site subtype 1, Site subtype 2, Site subtype 3, Primary histology, Histology subtype 1, Histology subtype 2, Histology subtype 3, Genome-wide screen, Mutation ID, Mutation CDS, Mutation AA, Mutation Description, Mutation zygosity, LOH, GRCh, Mutation genome position, Mutation strand, SNP, Resistance Mutation, FATHMM prediction, FATHMM score, Mutation somatic status, Pubmed_PMID, ID_STUDY, Sample source, Tumour origin, Age)

One column per annotation request will be added to the output with the annotation data.

Run Filters

After all parameters, filters and annotations are selected user have to click on the Run Filter button. Filtering are processed and a tab separated output is generated. There is one row for each variant and a column for each field in the VCF. The annotation columns are added too at the most-right columns. The generated files are stored in the output directory. When all processing is done a message appears below the button. If any error occurs, data about error is printed below the Run Filter button.



Find Coverage

All variants which passed the filtering are returned after clicking on Run Filters. However, sequencing not always cover the whole genome. In order to check the sequencing coverage a script is included in the tool. User have to select the BAM file used for variant calling to obtain the VCF file. Also have to select an output folder. After clicking in the Get Coverage button a table shows the exonic coverage of all files. The output is a tab delimited file where every row is an exon and the columns describe the parts of the exons which are not covered and the parts which are low covered (up to 20 reads).

ANNEXO 3: Scripts de R

batch_working.R

#'

#'batchWorkingFilter

#'

#'@description

#'

#'The function applies filters to a set of VCF files and prepares an annotated output for each one.

#'

#'@details

#'

#'The function iterates a set of given VCF files. For each VCF files the function prefilters, indexes and obtains the fields of the files.

#'Then checks if the memory usage is too high using the function splitGenome and uses the function compute multiple query to filter and prepare the output for each file.

#'

#'@usage

#'

#'batchWorkingFilter(data.files, input.directory, temporal.directory, query.dataframe, low.relevant.fields, output.directory, memory.cutoff, keep.nofiltered, table.extract, db.dataframe = NULL, limit.ranges = NULL, verbose = FALSE)

#'

#'@param data.files (character vector) the names of all the VCF files to be filtered

#'@param input.directory (character) the path to the directory where the input VCFs fiels are stored

#'@param temporal.directory (character) a temporal directory where store the prefiltered file

#'@param query.dataframe (dataframe) containing a checked relation of queries

#'@param low.relevant.files (character vector) a vector containing the name of the low relevant fields to be moved to the end of the output row

#'@param output.directory (character) the path where store the output file

#'@param memory.cutoff (numeric) a maximum value of the variant size contained in the first 10000bp. In Mb

```
#'@param keep.nofiltered (logical) if TRUE the variants in the same position as the filtered variants are keeped and a column is added to identify them
#'@param table.extract (logical) if TRUE the output is a dataframe. If false the output is a VCF file.
#'@param db.dataframe (dataframe) containing the annotation request
#'@param limit.ranges (Granges) the ranges to limit the query to some genetic regions
#'@param verbose(logical) in case of TRUE messages are shown
#'
#'@return
#'
#'Stores a set of VCFs or dataframes with the variants which passed the filters.
#'
#'@export batchWorkingFilter
#'

batchWorkingFilter<- function(data.files, input.directory, temporal.directory, query.dataframe, low.relevant.fields, output.directory, memory.cutoff, keep.nofiltered, table.extract, db.dataframe = NULL, limit.ranges = NULL, verbose = FALSE, ref.file = NULL){
 for (i in seq_len(length(data.files))){

   #Obtain file names
   raw.data.vcf<- file.path(input.directory, data.files[i])
   basic.name<- file_path_sans_ext(basename(raw.data.vcf))
   output.file<- file.path(output.directory, paste0("results_", basic.name))

   #Prefilter
   data.vcf<- vcfPrefilter(raw.data.vcf, basic.name, temporal.directory, verbose)

   #Get index
   tabix.dataVCF<- indexVCF(data.vcf, verbose)
```

```
   #Get field list
   field.list<- obtainVcfFields(data.vcf)

   if (is.null(limit.ranges)){
   #Check memory usage and tile genome is needed
    tiled.genome<- splitGenome(tabix.dataVCF, field.list, memory.cutoff, verbose = FALSE)
   } else {
    tiled.genome<- limit.ranges
   }
   #Execute query
   answerQuery(query.dataframe, field.list, tabix.dataVCF, output.file, keep.nofiltered, low.relevant.fields, tiled.genome, table.extract = table.extract, limit.ranges, db.dataframe, verbose = verbose)
  }
}
```

compare_files

```
#'
#'variantListPrepare
#'
#'@description
#'
#'If there is no query dataframe the function extract the ALT field of a VCF file and prepares a vector with the identifiers of the variants.
#'If there is a query dataframe uses the function compute multiple query to filter and obtain the identifier vector.
#'
#'@details
#'
```

```
#'The function first check if there is a query dataframe. If it did not exists the function extracts the ALT field of a VCF file and prepares
#'variant identifiers using the makeTagVector function. If there is a query dataframe the function call the compute multiple query function to obtain
#'the identifier vector
#'
#'@usage
#'
#'variantListPrepare(query.dataframe, tabix.dataVCF, field.list, keep.nofiltered, memory.cutoff, low.relevant.files, output.file, table.extract = FALSE, limit.ranges = NULL, verbose = FALSE)
#'
#'
#'@param query.dataframe (dataframe) containing a checked relation of queries
#'@param tabix.dataVCF (tabix environtment) an environtement to acces to an indexed VCF file
#'@param field.list (character vector) containing all the avaiable fields in the VCF file
#'@param limit.ranges (Granges) the ranges to limit the query to some genetic regions
#'@param verbose(logical) in case of TRUE messages are shown
#'
#'@return
#'
#'A character vector with the identifiers of all the variants which passed the filters
#'
#'@export variantListPrepare
#'

variantListPrepare<- function(query.dataframe, tabix.dataVCF, field.list, limit.ranges = NULL, verbose = FALSE){

  if (is.null(query.dataframe)){
    if (is.null(limit.ranges)){
```

```
    svp <- ScanVcfParam(fixed = c("ALT"), geno = NA, info = NA)
   } else {
    svp <- ScanVcfParam(fixed = c("ALT"), geno = NA, info = NA, which = limit.ranges)
   }
   readed.vcf<- readVcf(tabix.dataVCF, "hg19", svp)
   expanded.vcf<- expand(readed.vcf)

   #Generate tag vector
   tag.vector<- makeTagVector(expanded.vcf)

  } else {
   tag.vector<- computeMultipleQuery(query.dataframe, field.list, tabix.dataVCF,
limit.ranges, verbose = verbose)
  }
  return(tag.vector)
}
########################################################################
##################### Find inherited ############################
########################################################################

#'
#'findHereditary
#'
#'@description
#'
#'The function compares a list of variant identificators from different files and checks
which identificators are present in the reference file.
#'
#'@details
#'
#'The function first identifies the two files which are not the reference one. Then
compares this two files with the referecne and construct three logical vectors.
```

```
#'Eac one indicating if the variant is present in a not reference file or in both. Using this
logical vector the function contructs a vector output where for each variant
#'are returned the name not reference fiels where is found.
#'
#'@usage
#'
#'findHereditary(variant.list, ref.file, verbose = FALSE)
#'
#'
#'@param variant.list(list) A list with the variant identifiers of each file
#'@param ref.file(character) The file to be used as reference
#'@param verbose(logical) in case of TRUE messages are shown
#'
#'@return
#'
#'A character vector with the files where each variant of the reference is present
#'
#'@export findHereditary
#'



findHereditary<- function(variant.list, ref.file, verbose = FALSE){
  #identify parents
  list.names<- names(variant.list)
  parent1<- list.names[-which(list.names == ref.file)][1]
  parent2<- list.names[-which(list.names == ref.file)][2]

  #Compare parents to ref
  p1<- variant.list[[ref.file]] %in% variant.list[[parent1]]
  p2<- variant.list[[ref.file]] %in% variant.list[[parent2]]
  homoz<- p1 & p2
```

```
  hetero1<- p1 != homoz
  hetero2<- p2 != homoz

  #Create matrix with comparison
  inherited<- matrix(ncol=1, nrow = length(variant.list[[ref.file]]))
  inherited[which(homoz)]<- paste(parent1, parent2, sep = ", ")
  inherited[which(hetero1)]<- parent1
  inherited[which(hetero2)]<- parent2
  inherited[which(is.na(inherited))]<- "No inherited"

  return(inherited)
}
########################################################################
############################## Compare trio function ##############
########################################################################
#'
#'compareTrio
#'
#'@description
#'
#'Function filters and prepares a table output of a reference VCF file adding a column showing the inheretance relation with the other given files.
#'
#'@details
#'
#'Function firsts indexes and prefilters a set of given VCF files. Then it obtains an identifier for each variant which passes the filters, if there is no filters
#'all variants are returned. The function compares the identifiers of the reference VCF file with the others using the function find hereditary. At last function prepares
#'a table output using the function output prepare and the variant identifiers. A column with the inheritance is added to the table
#'
```

```
#'@usage
#'
#'compareTrio(data.files, input.directory, temporal.directory, query.dataframe, low.relevant.fields, output.directory, memory.cutoff, keep.nofiltered, table.extract, db.dataframe = NULL, limit.ranges = NULL, verbose = FALSE)
#'
#'@param data.files (character vector) the names of all the VCF files to be filtered
#'@param input.directory (character) the path to the directory where the input VCFs fiels are stored
#'@param temporal.directory (character) a temporal directory where store the prefiltered file
#'@param query.dataframe (dataframe) containing a checked relation of queries
#'@param low.relevant.files (character vector) a vector containing the name of the low relevant fields to be moved to the end of the output row
#'@param output.directory (character) the path where store the output file
#'@param memory.cutoff (numeric) a maximum value of the variant size contained in the first 10000bp. In Mb
#'@param keep.nofiltered (logical) if TRUE the variants in the same position as the filtered variants are keeped and a column is added to identify them
#'@param table.extract (logical) if TRUE the output is a dataframe. If false the output is a VCF file.
#'@param db.dataframe (dataframe) containing the annotation request
#'@param limit.ranges (Granges) the ranges to limit the query to some genetic regions
#'@param verbose(logical) in case of TRUE messages are shown
#'
#'@return
#'
#'Stores a dataframe with the variants of the reference file which passed the filters with a column showing the inheritance.
#'
#'@export compareTrio
#'
```

```
compareTrio<- function(data.files, input.directory, temporal.directory, query.dataframe, low.relevant.fields, output.directory, memory.cutoff, keep.nofiltered, table.extract, db.dataframe = NULL, limit.ranges = NULL, verbose = FALSE, ref.file = NULL){
  variant.list<- list()
  for (i in seq_len(length(data.files))){

    #Obtain file names
    raw.data.vcf<- file.path(input.directory, data.files[i])
    basic.name<- file_path_sans_ext(basename(raw.data.vcf))

    #Prefilter
    data.vcf<- vcfPrefilter(raw.data.vcf, basic.name, temporal.directory, verbose)

    #Get index
    tabix.dataVCF<- indexVCF(data.vcf, verbose)

    #Get field list
    field.list<- obtainVcfFields(data.vcf)



    #Finding variants
    tag.vector<- variantListPrepare(query.dataframe, tabix.dataVCF, field.list, limit.ranges, verbose)

    variant.list[[basic.name]]<- tag.vector
  }
  ref.file.basic<- file_path_sans_ext(basename(ref.file))
  #Find hereditary
  if (is.null(variant.list[[ref.file.basic]])){
    stop("No variants in reference file passed the filters")
```

```
  } else {
    inherited<- findHereditary(variant.list, ref.file.basic, verbose = FALSE)
    filtered.variants<- variant.list[[ref.file.basic]]

    variant.ranges<- generateRanges(variant.list[[ref.file]])
    output.file<- file.path(output.directory, paste0("inherited_", ref.file.basic))

    #Create output
    tabix.dataVCF<- TabixFile(file.path(temporal.directory, paste0("prefiltered_", ref.file.basic, ".vcf.bgz")))
    outputCreate(tabix.dataVCF, variant.ranges, filtered.variants, output.file, keep.nofiltered, low.relevant.fields, db.dataframe, table.extract, append = FALSE, inherited, verbose)
  }
}

#############################################################################
####################
############################ Compare multiple files ###################################
#############################################################################
###################
#'
#'compareMultipleFiles
#'
#'@description
#'
#'Function filters variants present in a set of VCF file and prepares a single output table with all the filtered variants adding a column containing the presence of each variant in the files
#'
#'@details
```

#'

#'Function firsts indexes and prefilters a set of given VCF files. Then it obtains an identifier for each variant which passes the filters, if there is no filters

#'all variants are returned. With the filtered variants it prepares a list of variants where all the files containing the variants are stored. Then prepares a table with all

#'the variants of the list adding a column with the presence of the variant in the files.

#'

#'@usage

#'

#'compareMultipleFiles(data.files, input.directory, temporal.directory, query.dataframe, low.relevant.fields, output.directory, memory.cutoff, keep.nofiltered, table.extract, db.dataframe = NULL, limit.ranges = NULL, verbose = FALSE)

#'

#'@param data.files (character vector) the names of all the VCF files to be filtered

#'@param input.directory (character) the path to the directory where the input VCFs fiels are stored

#'@param temporal.directory (character) a temporal directory where store the prefiltered file

#'@param query.dataframe (dataframe) containing a checked relation of queries

#'@param low.relevant.files (character vector) a vector containing the name of the low relevant fields to be moved to the end of the output row

#'@param output.directory (character) the path where store the output file

#'@param memory.cutoff (numeric) a maximum value of the variant size contained in the first 10000bp. In Mb

#'@param keep.nofiltered (logical) if TRUE the variants in the same position as the filtered variants are keeped and a column is added to identify them

#'@param table.extract (logical) if TRUE the output is a dataframe. If false the output is a VCF file.

#'@param db.dataframe (dataframe) containing the annotation request

#'@param limit.ranges (Granges) the ranges to limit the query to some genetic regions

#'@param verbose(logical) in case of TRUE messages are shown

#'

```
#'@return
#'
#'Stores a dataframe with all the variants present in all files adding a column listing in which files is present each variant
#'
#'@export compareMultipleFiles
#'

compareMultipleFiles<- function(data.files, input.directory, temporal.directory, query.dataframe, low.relevant.fields, output.directory, memory.cutoff, keep.nofiltered, table.extract, db.dataframe = NULL, limit.ranges = NULL, verbose = FALSE, ref.file = NULL){
 variant.list<- list()
 whole.output<- VCF()

 for (i in seq_len(length(data.files))){

  #Obtain file names
  raw.data.vcf<- file.path(input.directory, data.files[i])
  basic.name<- file_path_sans_ext(basename(raw.data.vcf))

  #Prefilter
  data.vcf<- vcfPrefilter(raw.data.vcf, basic.name, temporal.directory, verbose)

  #Get index
  tabix.dataVCF<- indexVCF(data.vcf, verbose)

  #Get field list
  field.list<- obtainVcfFields(data.vcf)



  #Finding variants
```

```
   filtered.variants<- variantListPrepare(query.dataframe, tabix.dataVCF, field.list, limit.ranges, verbose)

   if (is.null(filtered.variants)){
    msg("No variants passed the filters in file",  basic.name, verbose = verbose)
   } else {
    for (j in seq_len(length(filtered.variants))){
     if(is.null(variant.list[[filtered.variants[j]]])) {
      variant.list[[filtered.variants[j]]]<- basic.name
     } else { #If variant is already in the list
      variant.list[[filtered.variants[j]]]<- paste(variant.list[[filtered.variants[j]]], basic.name, sep = ", ")
     }
    }
    variant.ranges<- generateRanges(filtered.variants)
    #Create output for each file
    tabix.dataVCF<- TabixFile(file.path(temporal.directory, paste0("prefiltered_", basic.name, ".vcf.bgz")))
    readed.output<- generateCompareOutput(tabix.dataVCF, variant.ranges, verbose)

    #Append each file output to general output
    if (i == 1){
     whole.output<- readed.output
     colnames(whole.output)<- NULL

    } else  {
     colnames(readed.output)<- NULL
     whole.output<- rbind(whole.output, readed.output)
     whole.output<- unique(whole.output)
    }
   }
  }
```

```
 if(length(whole.output) != 0) {

   #Expand and check if variants have past the filter
   whole.output<- expand(whole.output)
   right.variants<- checkReadedVariants(whole.output, names(variant.list))
   whole.output<- whole.output[right.variants]



   #Extract dataframe
   whole.dataframe<- vcfToDataframe(whole.output, keep.nofiltered)
   whole.dataframe<- reorderOutput(whole.dataframe, low.relevant.fields)

   whole.dataframe<- addFilenameColumn(variant.list, whole.dataframe, verbose)
   save(variant.list, file = "temp/varlist.Rdata")
   if (is.null(db.dataframe) == FALSE){
     whole.dataframe<- annotateOutput(whole.dataframe, db.dataframe)
   }
   output.file<-                file.path(output.directory,                paste0("compare_",
basename(input.directory)))
   write.table(whole.dataframe,  file  =  paste0(output.file,  '.txt'),  sep  =  "\t",  quote  =
FALSE, row.names = FALSE)
 } else {
   msg("No variants passed the filters", verbose = verbose)
 }
}
```

database.R

```
##############################################
################## GO ########################
##############################################
```

```
#################### GO filter ########################
#'GOFilter
#'
#'@description
#'
#'Function search the genes related to a given keytype and value in the Gene Onthology Database
#'
#'@details
#'
#'Function extracts the keytype from the database and searches the elements wich contain the given value. Then uses the elements to search in the GO database for the
#'GOIDs. The GOID are transformed to the symbol gene name using the AnnotationHub database.
#'
#'@usage
#'
#'GOFilter(db.keytype, db.value)
#'
#'@param db.keytype (character) name of the GO keytype
#'@param db.value (character) value to be searched in the keytype
#'
#'@return
#'
#'A character vector with gene names
#'
#'@export GOFilter
#'
GOFilter<- function(db.keytype, db.value){
  #Load annotation hub
  ah<- AnnotationHub()
```

```
  orgs <- subset(ah, ah$rdataclass == "OrgDb")
  HumanDb<- query(orgs, "Homo Sapiens")[[1]]
  #Obtain GO keys and grep query
  GO.keys<- keys(GO.db, db.keytype, allowEscapes = TRUE)
  query.GO<- GO.keys[grep(db.value, GO.keys)]
  #Check query.GO for wrong scape characters
  query.GO<- sub("\\\"", "", query.GO)
  query.GO<- sub("\\\" ", " ", query.GO)
  query.GO<- sub("\\\".", "", query.GO)
  query.GO<- sub("\\\"", "", query.GO)
  #Pass keytyoe to GOID
  found.GO<- select(GO.db, keys = query.GO, keytype = db.keytype, columns =
"GOID",allowEscapes = TRUE)$GOID
  names.found<- select(HumanDb, keys = found.GO, keytype = "GO", columns =
"SYMBOL")$SYMBOL
  names.genes<- as.character(na.omit(names.found))
  #Disconnect from annotation hub
  dbFileDisconnect(HumanDb$conn)

  return(names.genes)
}

############## Go Annotate #################
#'GoAnnotate
#'
#'@description
#'
#'Function search the given keytype annotation in the Gene Onthology database for the
requested genes
#'
#'@details
#'
```

```
#'Function transforms the genes to GO notation using AnnotationHub. Then searches in the datbase for the keys related with the genes only for the given keytype.
#'the keys are stored in a list with a single entry for gene.
#'
#'@usage
#'
#'GoAnnotate(gene.vector, db.keytype)
#'
#'@param gene.vector(character vector) names of the genes to be annotated (symbol)
#'@param db.keytype (character) name of the GO keytype
#'
#'@return
#'
#'A named list with annotation. Gene names are the names of the list
#'
#'@export GoAnnotate
#'
GoAnnotate<- function(gene.vector, db.keytype){
  #Load annotation hub
  ah<- AnnotationHub()
  orgs <- subset(ah, ah$rdataclass == "OrgDb")
  HumanDb<- query(orgs, "Homo Sapiens")[[1]]

  #Obtain GO ID from symbol
  gene.list<- list()
  go.genes<- select(HumanDb, keys = gene.vector, keytype = "SYMBOL", columns = "GO")
  #If there are symbols look for the requested column
  if (all(is.na(go.genes)) == FALSE){
    found.GO<- cbind(go.genes, select(GO.db, keys = go.genes$GO, keytype = "GOID", columns = db.keytype)[,2])
    colnames(found.GO)[5]<- "found"
```

```
  }
  gene.found<- unique(found.GO$SYMBOL)
  #Make a list with the obtained columns and gene names
  for (i in seq_len(length(gene.vector))){
    gene.list[[gene.found[i]]]<-  as.vector(found.GO$found[which(found.GO$SYMBOL  ==
gene.found[i])])

  }
  #Disconnect from annotation hub
  dbFileDisconnect(HumanDb$conn)
  return(gene.list)
}
################################################################################
###############
############################## Reactome
##################################################
################################################################################
###############

############################### Reactome filter
########################################
#'reactomeFilter
#'
#'@description
#'
#'Function search the genes related to a given keytype and value in the Reactome
Database
#'
#'@details
#'
```

```
#'Function extracts the keytype from the database and searches the elements wich contain the given value. Then uses the elements to search in the reactome database for the
#'ENTREZIDs. The ENTREZIDs are transformed to the symbol gene name using the AnnotationHub database.
#'
#'@usage
#'
#'reactomeFilter(db.keytype, db.value)
#'
#'@param db.keytype (character) name of the GO keytype
#'@param db.value (character) value to be searched in the keytype
#'
#'@return
#'
#'A character vector with gene names
#'
#'@export reactomeFilter
#'
reactomeFilter<- function(db.keytype, db.value){
  #Load annotation hub
  ah<- AnnotationHub()
  orgs <- subset(ah, ah$rdataclass == "OrgDb")
  HumanDb<- query(orgs, "Homo Sapiens")[[1]]
  #Obtain keys from requested keytype
  reac.keys<- keys(reactome.db, db.keytype)
  query.reac<- reac.keys[grep(db.value, reac.keys)]
  #Use keys to obtain the gene ENTREZID and pass it to symbol
  id.genes<- select(reactome.db, keys = query.reac, keytype = db.keytype, columns = "ENTREZID")$ENTREZID
  names.genes<- select(HumanDb, keys = id.genes, keytype = "ENTREZID", columns = "SYMBOL")$SYMBOL
```

```
  names.genes<- as.character(na.omit(names.genes))
  #Disconnect from annotation hub
  dbFileDisconnect(HumanDb$conn)
  return(names.genes)
}

############################ Reactome output ###################################
reactomeAnnotate<- function(gene.vector, db.keytype){
  #Load annotation hub
  ah<- AnnotationHub()
  orgs <- subset(ah, ah$rdataclass == "OrgDb")
  HumanDb<- query(orgs, "Homo Sapiens")[[1]]
  #Pass gens from symbol to ENTREZID
  gene.list<- list()
  reac.genes<- select(HumanDb, keys = gene.vector, keytype = "SYMBOL", columns = "ENTREZID")
  #If there are genes obtain the keytype requested
  if (all(is.na(reac.genes)) == FALSE){
    found.reac<- select(reactome.db, keys = reac.genes$ENTREZID, keytype = "ENTREZID", columns = db.keytype)
  }
  #Make a list with the keys obtained for each gene
  for (i in seq_len(length(gene.vector))){
    gene.list[[reac.genes$SYMBOL[i]]]<- found.reac[,2][which(found.reac[,1] == reac.genes$ENTREZID[i])]

  }
  #Disconnect from annotation hub
  dbFileDisconnect(HumanDb$conn)
  return(gene.list)
}
```

```
##############################################################################
#######
################################ MSigDB
########################################
##############################################################################
#######

############################## MSigDB Filter
##################################
#'MSigDBFilter
#'
#'@description
#'
#'Function search the genes related to a given keytype and value in the MSig Database
#'
#'@details
#'
#'Function extracts the keytype from the database and searches the elements wich
contain the given value. Then uses the elements to search in the MSig database for the
#'symbol gene name
#'
#'@usage
#'
#'MSigDBFilter(db.keytype, db.value)
#'
#'@param db.keytype (character) name of the GO keytype
#'@param db.value (character) value to be searched in the keytype
#'
#'@return
#'
#'A character vector with gene names
```

```
#'
#'@export MSigDBFilter
#'
MSigDBFilter<- function(db.keytype, db.value){
  #Obtain the keytype position
  keytype.position<- grep(db.keytype, names(MSigDB))
  #Obtain gene names from the keytype
  key.position<- grep(db.value, names(MSigDB[[keytype.position]]))
  names.genes<- MSigDB[[keytype.position]][key.position]



  return(names.genes)
}
############################## MSigDB Output
#######################################
MSigDBAnnotate<- function(gene.vector, db.keytype){
  #Obtain the keytype
  keynumber<- grep(db.keytype, names(MSigDB))
  #Obatain the gene names from the keytype and make a list with an entry for each
gene
  gene.list<- list()
  for (i in seq_len(length(gene.vector))){
    gene.found<- lapply(MSigDB[[keynumber]], function(x){gene.vector[i] %in% x})
    gene.list[[gene.vector[i]]]<- names(which(unlist(gene.found)))
  }



  return(gene.list)
}

#########################################################
################## KEGG ################################
```

```
#########################################################

################ KEGG filter ########################
#'KEGGFilter
#'
#'@description
#'
#'Function search the genes related to a given keytype and value in the Kegg Database
#'
#'@details
#'
#'Function extracts the keytype from the database and searches the elements wich contain the given value. Then uses the elements to search in the Kegg database for the
#'ENTREZIDs. The ENTREZIDs are transformed to the symbol gene name using the AnnotationHub database.
#'
#'@usage
#'
#'KEGGFilter(db.keytype, db.value)
#'
#'@param db.keytype (character) name of the GO keytype
#'@param db.value (character) value to be searched in the keytype
#'
#'@return
#'
#'A character vector with gene names
#'
#'@export KEGGFilter
#'
KEGGFilter<- function(db.keytype = NULL,db.value){
  #Load annotation hub
  ah<- AnnotationHub()
```

```
  orgs <- subset(ah, ah$rdataclass == "OrgDb")
  HumanDb<- query(orgs, "Homo Sapiens")[[1]]
  #Search in kegg db
  kegg.genes<- keggFind("T01001", db.value)
  #Pass KEGG id to uniprot ID
  uni.genes<- keggConv("ncbi-geneid", names(kegg.genes))
  #Delete prefix and spaces
  uni.genes<- unlist(strsplit(uni.genes, split="ncbi-geneid:"))
  uni.genes<- uni.genes[which(uni.genes != "")]
  #Transform names from uniprot ID to SYMBOL
  names.genes<- select(HumanDb, keys = uni.genes, keytype = "ENTREZID", columns =
"SYMBOL")$SYMBOL
  names.genes<- unique(names.genes)
  #Disconnect from annotation hub
  dbFileDisconnect(HumanDb$conn)
return(names.genes)
}

###################### KEGG output ##########################
KEGGAnnotate<- function(gene.vector, db.keytype){
  #Load annotation hub
  ah<- AnnotationHub()
  orgs <- subset(ah, ah$rdataclass == "OrgDb")
  HumanDb<- query(orgs, "Homo Sapiens")[[1]]

  #Transfom genes from Symbol to entrez id
  entrez.genes<- select(HumanDb, keys = gene.vector, keytype = "SYMBOL", columns =
"ENTREZID")$ENTREZID
  entrez.genes<-      unlist(lapply(entrez.genes,      function(gene){paste0("ncbi-geneid:",
gene)}))

  gene.list<- list()
```

```
  #Search each gene
  for (i in seq_len(length(gene.vector))){
    #If uni gene is na skip annotation
    if (entrez.genes[i] == "ncbi-geneid:NA"){
      gene.list[[gene.vector[i]]]<- ""
    } else {
    #Transform names from uniprot to KEGG ID
    kegg.genes<- keggConv("T01001", entrez.genes[i])
    #Make query keeping keytype
    kegg.annotation<- tryCatch(keggGet(kegg.genes)[[1]][[db.keytype]], error =
function(e) NA)

    gene.list[[gene.vector[i]]]<- paste(kegg.annotation, collapse = " / ")
    }
  }
  #Disconnect from annotation hub
  dbFileDisconnect(HumanDb$conn)
  return(gene.list)
}

#########################################################################
################################## Cosmic ################################
#########################################################################

################ Cosmic annotate ##############################
getKeytypeCosmic<- function(db.keytype, gene.vector){
  grep.columns<- "head -1
data/database/CosmicCompleteTargetedScreensMutantExport.tsv"
  cosmic.columns<- system(grep.columns, intern = TRUE)
  cosmic.columns<- unlist(strsplit(cosmic.columns, split = "\t"))
  keytype.position<- which(cosmic.columns %in% db.keytype)
  cosmic.list<- list()
```

```
  for (i in seq_len(length(gene.vector))){
    message(i)
    grep.command<- paste0("if grep '^", gene.vector[i], "\t' data/database/CosmicCompleteTargetedScreensMutantExport.tsv; then : ; else echo 'NULL'; fi;" )
    entries<- system(grep.command, intern = TRUE)

    cosmic.presence<- is.null(entries) == FALSE
    splited.entries<- strsplit(entries, split= "\t")
    aa.mutation<- unlist(lapply(splited.entries, function(entry){entry[19]}))
    aa.greg<- gregexpr(text =  aa.mutation, pattern = "(?<=p\\.[A-z])[0-9]+(?=[A-z])", perl = TRUE)
    aa.positions<- as.numeric(regmatches(x = aa.mutation, m = aa.greg))
    na.positions<- which(is.na(aa.positions))


    cosmic.keytype<- unlist(lapply(splited.entries, function(entry){entry[keytype.position]}))

    cosmic.list[[gene.vector[i]]]$presence<-  cosmic.presence
    cosmic.list[[gene.vector[i]]]$position<- aa.positions[-na.positions]
    cosmic.list[[gene.vector[i]]]$keytype<- cosmic.keytype[-na.positions]
  }
  return(cosmic.list)
}

annotateCosmic<- function(output.dataframe, db.keytype){
  #Obtain vector of gens
  gene.names<- lapply(output.dataframe$Gene.refGene, function(var){unlist(strsplit(var, split = ", "))})
  gene.vector<- unique(unlist(gene.names))
```

```
  #Obtain domains from given genes
  cosmic.list<- getKeytypeCosmic(db.keytype, gene.vector)

  #Obtain aminnoacid change position for each variant
  aa.greg<- gregexpr(text = output.dataframe$AAChange.refGene, pattern =
"(?<=p\\.[A-z])[0-9]+(?=[A-z])", perl = TRUE)
  aa.positions<- regmatches(x = output.dataframe$AAChange.refGene, m = aa.greg)
  names(aa.positions)<- output.dataframe$Gene.refGene
  cosmic.found<- character()
  cosmic.column<- matrix(ncol = 2, nrow = length(aa.positions))

  #Iterate on variants
  for (i in seq_len(length(aa.positions))){
    if (length(aa.positions[[i]]) >= 1){
      #Obtain gene position for each variant on cosmic list
      cosmic.list.position<- which(names(cosmic.list) %in% gene.names[[i]])
      cosmic.found<- character()
      #Iterate on each cosmic list gene present on the variant
      for (j in seq_len(length(cosmic.list.position))){
        #Add presence logical if don't exists
        if (is.na(cosmic.column[i,2])){
        cosmic.column[i,2]<- cosmic.list[[cosmic.list.position[j]]]$presence
        #if exists paste new logical
        } else {
          cosmic.column[i,2]<-                                    paste(cosmic.column[i,2],
cosmic.list[[cosmic.list.position[j]]]$presence, sep = " / ")
        }
        if(all(is.na(cosmic.list[[cosmic.list.position[j]]]$position)) == FALSE){
          #Iterate on different aminoacid position for each varian
          for(k in seq_len(length(aa.positions[[i]]))){
            #Check if position is in the cosmic position
```

```
        cosmic.locations<- which((cosmic.list[[cosmic.list.position[j]]]$position == as.numeric(aa.positions[[i]][k])))
        #Store coincidences
        cosmic.found<- c(cosmic.found, cosmic.list[[cosmic.list.position[j]]]$keytype[cosmic.locations])
       }
      }
     }
   }
   #If there are coincidences include them in the output column
   if (length(cosmic.found) >= 1){
    cosmic.column[i,1]<- paste(cosmic.found, collapse = " / ")
   } else {
    cosmic.column[i,1]<- NA
   }


 }
 colnames(cosmic.column)<- c(paste("Cosmic", db.keytype, sep = "_"), "Cosmic_presence")
 return(cosmic.column)
}

#####################################################################
################## Uniprot#####################################
#####################################################################

#################### Uniprot output ##############################################
getUniprotDomains<- function(gene.vector){

 domain.list<- list()
```

```
  #Iterate on vector of genes
  for (i in seq_len(length(gene.vector))){
    #Grep de gene entry and the domain using bash system conection
    grep.command<- paste0("sed -n '/Name=", gene.vector[i], "/,/SQ SEQUENCE/p'
data/database/uniprot_sprot_human.dat |
                          if grep 'FT   DOMAIN'; then : ; else echo 'NULL'; fi;")

    domains<- system(grep.command, intern = TRUE)
    #Create a list with all domain features
    if (is.null(domains)== FALSE){
      #Separate domain features
      domains<- strsplit(domains, split = "  ")
      #Eliminate spaces
      domains<- lapply(domains, function(gn){gn[which(gn != "")]})
      #Store domain start and end in the list
      domain.list[[gene.vector[i]]]$Start<- unlist(lapply(domains,
function(dom){as.numeric(dom[3])}))
      domain.list[[gene.vector[i]]]$End<- unlist(lapply(domains,
function(dom){as.numeric(dom[4])}))
      #Store description until dot
      domain.list[[gene.vector[i]]]$Description<- unlist(lapply(domains, function(dom){
                                    as.character(unlist(strsplit(x = dom[5], split =
"\\."))[1]}))
    }
  }
  return(domain.list)
}

getUniprotAnnotation<- function(gene.vector){
  #Iterate on vector of genes
  uniprot.list<- list()
  for (i in seq_len(length(gene.vector))){
```

```
   #Grep de gene entry and eliminate domain using bash system conection
   grep.command<- paste0("sed -n '/Name=", gene.vector[i], "/,/SQ SEQUENCE/p'
data/database/uniprot_sprot_human.dat | grep 'FT [[:upper:]]' |
                    if grep -v 'FT DOMAIN'; then : ; else echo 'NULL'; fi;")
   ft.uniprot<- system(grep.command, intern = TRUE)
   #Create a list with all features
   if (is.null(ft.uniprot)== FALSE){
    #Separate every feature
    ft.uniprot<- strsplit(ft.uniprot, split = "  ")
    #Eliminate spaces
    ft.uniprot<- lapply(ft.uniprot, function(gn){gn[which(gn != "")]})
    #Store domain start and end in the list
    uniprot.list[[gene.vector[i]]]$Start<- unlist(lapply(ft.uniprot,
function(ft){as.numeric(ft[3])}))
    uniprot.list[[gene.vector[i]]]$End<- unlist(lapply(ft.uniprot,
function(ft){as.numeric(ft[4])}))
    #Store together feature type and description
    uniprot.list[[gene.vector[i]]]$Description<- unlist(lapply(ft.uniprot,
function(ft){paste(ft[2], ft[5], sep = " : ")}))
   }
  }
  return(uniprot.list)
}

######################## Annotate Uniprot domains ####################
annotateUniprotDomains<- function(output.dataframe, db.keytype){
  db.keytype<-NULL
  #Obtain vector of gens
  gene.names<- lapply(output.dataframe$Gene.refGene, function(var){unlist(strsplit(var,
split = ", "))})
  gene.vector<- unique(unlist(gene.names))
```

```
 #Obtain domains from given genes
 domain.list<- getUniprotDomains(gene.vector)

 #Obtain aminnoacid change position for each variant
 aa.greg<- gregexpr(text = output.dataframe$AAChange.refGene, pattern =
"(?<=p\\.[A-z])[0-9]+(?=[A-z])", perl = TRUE)
 aa.positions<- regmatches(x = output.dataframe$AAChange.refGene, m = aa.greg)
 names(aa.positions)<- output.dataframe$Gene.refGene
 domains.found<- character()
 domain.column<- matrix(ncol = 1, nrow = length(aa.positions))

 #Iterate on variants
 for (i in seq_len(length(aa.positions))){
  if (length(aa.positions[[i]]) >= 1){
   #Obtain gene position for each variant on domain list
   domain.list.position<- which(names(domain.list) %in% gene.names[[i]])
   domains.found<- character()
   #Iterate on each domain list gene present on the variant
   for (j in seq_len(length(domain.list.position))){
    if(all(is.na(domain.list[[domain.list.position[j]]]$Start)) == FALSE){
     #Iterate on different aminoacid position for each varian
     for(k in seq_len(length(aa.positions[[i]]))){
      #Check if position is in the domain interval
      domain.locations<- which((domain.list[[domain.list.position[j]]]$Start <=
as.numeric(aa.positions[[i]][k]) & domain.list[[domain.list.position[j]]]$End >=
as.numeric(aa.positions[[i]][k])))
      #Store coincidences
      domains.found<- c(domains.found,
domain.list[[domain.list.position[j]]]$Description[domain.locations])
     }
    }
```

```
    }
   }
   #If there are coincidences include them in the output column
   if (length(domains.found) >= 1){
    domain.column[i,]<- paste(domains.found, collapse = " / ")
   } else {
    domain.column[i,]<- NA
   }
  }
  colnames(domain.column)<- "Uniprot.domains"
  return(domain.column)
}


######################## Annotate uniprot FT ####################
annotateUniprotFt<- function(output.dataframe, db.keytype){
  db.keytype<-NULL
  #Obtain vector of gens
  gene.names<- lapply(output.dataframe$Gene.refGene, function(var){unlist(strsplit(var,
split = ", "))})
  gene.vector<- unique(unlist(gene.names))
  #Obtain feature list
  ft.list<- getUniprotAnnotation(gene.vector)

  #Obtain aminnoacid change position for each variant
  aa.greg<- gregexpr(text = output.dataframe$AAChange.refGene, pattern =
"(?<=p\\.[A-z])[0-9]+(?=[A-z])", perl = TRUE)
  aa.positions<- regmatches(x = output.dataframe$AAChange.refGene, m = aa.greg)
  names(aa.positions)<- output.dataframe$Gene.refGene
  ft.found<- character()
  ft.column<- matrix(ncol = 1, nrow = length(aa.positions))
```

```
  #Iterate on each variant
  for (i in seq_len(length(aa.positions))){
    if (length(aa.positions[[i]]) >= 1){
      #Find position of variant genes in domain list
      ft.list.position<- which(names(ft.list) %in% gene.names[[i]])
      ft.found<- character()
      #Iterate on variant genes of domain list
      for (j in seq_len(length(ft.list.position))){
        if(all(is.na(ft.list[[ft.list.position[j]]]$Start)) == FALSE){
          #Iterate on every aminoacid change postion
          for(k in seq_len(length(aa.positions[[i]]))){
            #Find if which position are within the domain interval and store them
            ft.locations<- which((ft.list[[ft.list.position[j]]]$Start <=
as.numeric(aa.positions[[i]][k]) & ft.list[[ft.list.position[j]]]$End >=
as.numeric(aa.positions[[i]][k])))
            ft.found<- c(ft.found, ft.list[[ft.list.position[j]]]$Description[ft.locations])
          }
        }
      }
    }
    #If there are coincidences include them in the output column
    if (length(ft.found) >= 1){
      ft.column[i,]<- paste(ft.found, collapse = " / ")
    } else {
      ft.column[i,]<- NA
    }
  }
  colnames(ft.column)<- "Uniprot.ft"
  return(ft.column)
}
```

```
##############################################################################
#####
######################### Database Filter ###################################
##############################################################################
#####
dataBaseFilter<- function(input.database, query.dataframe, verbose= FALSE){


 #Check db dataframe
 db.dataframe<- dbQueryCheck(input.database, verbose)



 #If is null skip
 if (is.null(db.dataframe) == FALSE){
  for (i in seq_len(nrow(db.dataframe))){
    #Obtain row values
    database<- db.dataframe$database[i]
    db.keytype<- db.dataframe$db.keytype[i]
    db.value<- db.dataframe$db.value[i]
    db.group<- db.dataframe$Group[i]
    db.invert<- db.dataframe$inclusive[i]

    names.genes<- unlist(match.fun(database)(db.keytype, db.value))
    total.genes<- names.genes

    #Eliminate duplicates and NAs
    total.genes<- unique(unlist(total.genes))
    total.genes<- na.omit(total.genes)
    total.genes<- paste(total.genes, collapse = "/")
    db.row<-          c("Gene.refGene","filterByMultipleString",          total.genes,
as.character(db.group), as.character(db.invert))
```

```
    #Check if query dataframe exists if not create it
    if (is.null(query.dataframe)){
      query.dataframe<- data.frame(t(data.frame(db.row)), stringsAsFactors = F)

    } else {
      query.dataframe<- rbind(query.dataframe, db.row)
    }
   }

 }

 #If there is not query dataframe
 if (nrow(query.dataframe) == 0){
   query.dataframe<- NULL
 #If exists give colnames
 } else {
   names(query.dataframe)<- c("field", "filter", "value", "adding", "inclusive")
 }

 return(query.dataframe)
}

##############################################################################
########
############################ Anotate Output
#######################################
##############################################################################
#######
annotateOutput<- function(output.dataframe, db.dataframe){
```

```
  #Check if annotate dataframe exists
  if (is.null(db.dataframe) == FALSE){
    #Prepare gene vector
    gene.names<- lapply(output.dataframe$Gene.refGene, function(var){strsplit(var, split
= ", ")})
    gene.vector<- unique(unlist(gene.names))
    #Iterate on each annotation request
    for (i in seq_len(nrow(db.dataframe))){

      database<- db.dataframe$database[i]
      db.keytype<- as.character(db.dataframe$keytype[i])
      #If the request is for uniprot database use direct function
      if (grepl(pattern = "Uniprot", database) | grepl(pattern = "Cosmic" , database) |
grepl(pattern = "addAF", database)){
        annotation.column<- match.fun(database)(output.dataframe, db.keytype)
        annotation.column<- data.frame(annotation.column)
        colnames(annotation.column)<- database
      #If not prepare annotation values
      } else {

        #Use annotation function
        gene.list<- match.fun(database)(gene.vector, db.keytype )

        #Prepare annotation column from list
        annotation.column<- matrix(ncol = 1, nrow = length(gene.names))
        colnames(annotation.column)<- db.keytype
        for (i in seq_len(length(gene.names))){
          annotation.column[i,1]<-      paste(unlist(gene.list[which(names(gene.list)      %in%
unlist(gene.names[i]))]), collapse = " // ")

        }
      }
```

```
    if (grepl(pattern = "addAF", database)){
      ad.index<- grep("^AD$", names(output.dataframe))
      output.dataframe<- data.frame(cbind(output.dataframe[, 1:ad.index],
annotation.column, output.dataframe[,
((ad.index+1):length(names(output.dataframe)))]))
    } else {
    output.dataframe<- cbind(output.dataframe, annotation.column)
    }
   }
  }
  return(output.dataframe)
}
```

execute.R

```
### Libraries ####

library(VariantAnnotation)
library(regioneR)
library(memoise)
library(yaml)
library(tools)


#Anotation libraries
library(AnnotationHub)
library(GO.db)
library(reactome.db)
library(MSigDB)
library(KEGGREST)
library(plyr)
```

```
source("input.R")
source("output.R")
source("utils.R")
source("filters.R")
source("compare_files.R")
source("batch_working.R")
source("database.R")

msg ("Libraries loaded")

#########################################################################
#################### Read YAML file ##############################
#########################################################################
execute<- function(user.directory, file.yaml){
temporal.directory<- user.directory
#input<- input.list
input<- yaml.load_file(file.yaml)
#Extract variables from yaml
script<- input$script

memory.cutoff<- as.numeric(input$memory_cutoff)
keep.nofiltered<- as.logical(input$keep_nofiltered)
table.extract<- as.logical(input$table_extract)
verbose<- as.logical(input$verbose)
include<- as.logical(input$include_genes)

#Extract file paths
input.directory<- input$files$input.directory
header.output.path<- input$files$output.header
output.directory<- input$files$output.directory

#Extract query
```

```
query<- input$query_list
ref.file<- input$ref_file
list.ranges<- input$limit_ranges

input.database<-input$database_list
input.annotate<- input$annotate_list
#Extract low relevant files
low.relevant.fields<- as.character(unlist(input$low_relevant_fields))

########################################################################
################### Check Data From Yaml ###########################
########################################################################



#Process input
data.files<- list.files(input.directory)

#Check or Create output directory
if (dir.exists(output.directory) == FALSE){
  dir.create(output.directory)
} else {
  checkOutputDirectory(data.files, output.directory, script, input.directory, ref.file)
}

#Check query
query.dataframe<- queryRead(query, file.path(input.directory, data.files[1]), verbose =
verbose)

#Check annotation request

db.dataframe<- checkAnnotate(input.annotate)
```

```
#Add database filter to query dataframe
query.dataframe<- dataBaseFilter(input.database, query.dataframe, verbose)

#Check ranges
limit.ranges<- rangesRead(list.ranges)

################################################################################
#
############################ Execute ###################################
################################################################################

match.fun(script)(data.files, input.directory, temporal.directory, query.dataframe, low.relevant.fields, output.directory, memory.cutoff, keep.nofiltered, table.extract, db.dataframe, limit.ranges, verbose, ref.file)
msg("DONE", verbose = verbose)
}
```

ExonCoverage.R

```
library(regioneR)
library(bamsignals)
library(R.utils)
library(tools)

#Code for obtaining the gene names of exons from Agilent V5 (to get targets)

#listed<- strsplit(covered$V4, split = ",")
#ref<- unlist(lapply(listed, function(x){x[1]}))
#ref<- ref[grep("ref", ref)]
#genes<-strsplit(ref, split = "\\|")
#genes<- unlist(lapply(genes, function(x){x[2]}))
```

```
#genes<- unique(genes)
#write.table(genes, file = "agilent_genes.txt", sep = "\t", quote = F, row.names = F, col.names = F)


#Execute on terminal to change chr notation
#sed -e 's/chr//' data/exons.bed > data/exons_ensemble.bed


# Code for changing the exons of minus strand
#readed.exons<- read.table("data/exons_ensemble.bed", stringsAsFactors = F, sep = "\t", header = F, quote = "")
#readed.exons$V5<- readed.exons$V5 + 1
#write.table(readed.exons, "data/exons_ensemble.bed", sep = "\t", quote = F, row.names = F)
#Execute on terminal
#awk -v OFS='\t' 'NR==FNR{a[$4]=$5; next} $6=="-"{$5=a[$4]-$5+1}1' data/exons_ensemble.bed{,} | column -t > data/exons_strand.bed

#Set bam file
bam.file<- input.bam
basic.name<- file_path_sans_ext(basename(bam.file))
#bam.file<- commandArgs(trailingOnly = TRUE)
#Load exons
readed.exons<- read.table("ExonCoverage/data/exons_strand.bed", stringsAsFactors = F, sep = "", header = T, quote = "")
names(readed.exons)<- c("chr", "start", "end", "gene", "exon", "strand")
#delete extra chrs
extra.chr<- grep("_", readed.exons$chr)
filtered.exons<- readed.exons[-extra.chr,]
#Obtain Granges
exon.ranges<- toGRanges(filtered.exons)
```

```
# compute coverage per base
coverage<- bamCoverage(bampath = bam.file, gr = exon.ranges)
#Search no covered positions
no.coverage.exonic<-unlist(lapply(coverage, function(exon){
  positions<- which(exon == 0)
  seqToHumanReadable(positions)
}))
#Search low covered positions
less21.coverage.exonic<- unlist(lapply(coverage, function(exon){
  positions<- which(exon >= 1 & exon <= 20)
  seqToHumanReadable(positions)
}))
#Search no covered positions in genomic coordinates
no.coverage.genomic<-unlist(lapply(seq_along(coverage), function(i){
  genomic.position<- filtered.exons$start[i]
  positions<- which(coverage[i] == 0) + (genomic.position - 1)
  seqToHumanReadable(positions)
}))
#Search low positions in genomic coordinates
less21.coverage.genomic<-unlist(lapply(seq_along(coverage), function(i){
  genomic.position<- filtered.exons$start[i]
  positions<- which(coverage[i] >= 1 & coverage[i] <= 20) + (genomic.position - 1)
  seqToHumanReadable(positions)
}))
#Paste all coverages
all.coverage<- unlist(lapply(coverage, function(x){paste(x, collapse = ", ")}))
message("keep doin")
coverage.columns<-                data.frame(all.coverage,                no.coverage.exonic,
less21.coverage.exonic,           no.coverage.genomic,          less21.coverage.genomic,
stringsAsFactors = FALSE)
covered.exons<- cbind(filtered.exons, coverage.columns)
```

```
#Order
chr.column<- as.character(covered.exons[,1])
chr.column[which(chr.column == "X")]<- 24
chr.column[which(chr.column == "Y")]<- 25
chr.column<- as.numeric(chr.column)
covered.exons<- covered.exons[order(chr.column, covered.exons[,2]),]

less21.positions<- which(covered.exons$no.coverage.exonic == "" & covered.exons$less21.coverage.exonic == "")

less21.coverage<- covered.exons[-less21.positions, -7]



#Select only low coverage or no coverage

write.table(covered.exons, file = filePath(cov.dir, paste0("coverage_", basic.name, ".txt")), sep = "\t", quote = F, row.names = F)
write.table(less21.coverage, file = filePath(cov.dir, paste0("low_coverage_", basic.name, ".txt")), sep = "\t", quote = F, row.names = F)


message("done")
```

filters.R

```
#####################################################################
############################Filter functions ########################
#####################################################################

################ Filter by character ####################
```

```
#' filterByFullString
#'
#'@description
#'
#'Function filters a vector finding if a given string is found in a character vector.
#'
#'@details
#'
#'Function checks if a given extracted field is equal to a given string for each variant. Then returns a logical vetor with the answers
#'
#'@usage
#'
#'filterByFullString(query.value, extracted.field)
#'
#'@param query.value (character) a string with the value to find
#'@param extracted.field (vector) extracted field of a VCF file
#'
#'@return
#'
#'Returns a logical vector for each variant. If true variants passed the filter
#'
#'@export filterByFullString

filterByFullString<- function(query.value, extracted.field){
 filtered.variants<- unlist(lapply(extracted.field, function(x){return(any(x %in% query.value))}))
 return(filtered.variants)
}

#'filterByContainingString
#'
```

```
#'@description
#'
#'Function filters a vector finding if a value is found in a character vector
#'
#'@details
#'
#'Function checks if a given extracted field elements contain a given string. Then returns
a logical vetor with the answers
#'
#'@usage
#'
#'filterByContainingString(query.value, extracted.field)
#'
#'@param query.value (character) a string with the value to find
#'@param extracted.field (vector) extracted field of a VCF file
#'
#'@return
#'
#'Returns a logical vector for each variant. If true variants passed the filter
#'
#'filterByContainingString

filterByContainingString <- function(query.value, extracted.field){
  return(grepl(pattern =  query.value, x = extracted.field))
}

#'filterByMultipleString
#'
#'@description
#'
#'Function filters a vector finding if one of the values given is in a character vector
#'
```

```
#'@details
#'
#'Function checks if a given extracted field elements is one of the given strings. Then returns a logical vetor with the answers
#'
#'@usage
#'
#'filterByMultipleString(query.value, extracted.field)
#'
#'@param query.value (character) a string with the value to find
#'@param extracted.field (vector) extracted field of a VCF file
#'
#'@return
#'
#'Returns a logical vector for each variant. If true variants passed the filter
#'
#'@export filterByMultiplegString
#'
filterByMultipleString<- function(query.value, extracted.field){
 query.value<- as.character(unlist(strsplit(query.value, split="/")))
 filtered.variants<- unlist(lapply(extracted.field, function(x){return(any(x %in% query.value))}))
 return(filtered.variants)
}

#'filterByContainingMultipleString
#'
#'@description
#'
#'Function filters a vector finding if one of the values given is contained in a character vector
#'
```

```
#'@details
#'
#'Function checks if a given extracted field elements contains one of the given strings.
Then returns a logical vetor with the answers
#'
#'@usage
#'
#'filterByContainingMultipleString(query.value, extracted.field)
#'
#'@param query.value (character) a string with the value to find
#'@param extracted.field (vector) extracted field of a VCF file
#'
#'@return
#'
#'Returns a logical vector for each variant. If true variants passed the filter
#'
#'@export filterByContainingMultiplegString
#'
filterByContainingMultipleString<- function(query.value, extracted.field){
  query.value<- as.character(unlist(strsplit(query.value, split="/")))
  total.logical<- logical(length = length(extracted.field))
  for (i in seq_len(length(query.value))){
    logical.vector<- grepl(pattern =  query.value[i], x = extracted.field)

    total.logical<- total.logical | logical.vector
  }
  return(total.logical)
}
#########Filter by numeric ############################
#'
#'filterByMin
#'
```

```
#'@description
#'
#'Function filters a vector finding if any element is bigger than a cutoff.
#'
#'@details
#'
#'Function checks if a given extracted field is bigger than a cutoff. Then returns a logical
vetor with the answers.
#'
#'@usage
#'
#'filterByMin(query.value, extracted.field)
#'
#'@param query.value (numeric) the minimum cutoff
#'@param extracted.field (vector) extracted field of a VCF file
#'
#'@return
#'
#'Returns a logical vector for each variant. If true variants passed the filter.
#'
#'@export filterByMin
#'

filterByMin<- function(query.value, extracted.field){
  if (length(unlist(extracted.field)) == length(extracted.field)){
    extracted.field<- as.numeric(sub("^\\.$", 0, extracted.field))
    if (is.numeric(unlist(extracted.field))){
      return(extracted.field >= as.numeric(query.value))
    } else {
      stop("Not all values in field are numeric")
    }
  } else {
```

```
    stop("There is more than one value for a variant")
  }
}

#'
#'filterByMax
#'
#'@description
#'
#'Function filters a vector finding if any element is smaller than a cutoff.
#'
#'@details
#'
#'Function checks if a given extracted field is smaller than a cutoff. Then returns a logical vetor with the answers.
#'
#'@usage
#'
#'filterByMax(query.value, extracted.field)
#'
#'@param query.value (numeric) the maximum cutoff
#'@param extracted.field (vector) extracted field of a VCF file
#'
#'@return
#'
#'Returns a logical vector for each variant. If true variants passed the filter.
#'
#'@export filterByMax
#'

filterByMax<- function(query.value, extracted.field){
  if (length(unlist(extracted.field)) == length(extracted.field)){
```

```
    extracted.field<- as.numeric(sub("^\\.$", 0, extracted.field))
    if (is.numeric(unlist(extracted.field))){
      return(extracted.field <= as.numeric(query.value))
    } else {
      stop("Not all values in field are numeric")
    }
  } else {
    stop("There is more than one value for a variant")
  }
}

#'
#'filterByInterval
#'
#'@description
#'
#'Function filters a vector finding if any element is between an interval.
#'
#'@details
#'
#'Function checks if a given extracted field is between an interval. Then returns a logical
vetor with the answers.
#'
#'@usage
#'
#'filterByInterval(query.value, extracted.field)
#'
#'@param query.value (characer) the minimum and maximum of the interval separated
by a "-"
#'@param extracted.field (vector) extracted field of a VCF file
#'
#'@return
```

```
#'
#'Returns a logical vector for each variant. If true variants passed the filter.
#'
#'@export filterByInterval
#'

filterByInterval<- function(query.value, extracted.field){
  #Separate interval and find variants within interval
  query.value<- as.numeric(unlist(strsplit(query.value, split="-")))
  if (length(unlist(extracted.field)) == length(extracted.field)){
    extracted.field<- as.numeric(sub("^\\.$", 0, extracted.field))
    if (is.numeric(unlist(extracted.field))){
      return(extracted.field >= query.value[1] & extracted.field <= query.value[2])
    } else {
      stop("Not all values in field are numeric")
    }
  } else {
    stop("There is more than one value for a variant")
  }
}


###################### Filter By AD interval ######################3
#'
#'filterByADInterval
#'
#'@description
#'
#'Function filters a vector finding if AD element is between an interval.
#'
#'@details
#'
```

```
#'Function checks if the values of AD are bigger than a given thershold
#'
#'@usage
#'
#'filterByADInterval(query.value, extracted.field)
#'
#'@param query.value (characer) Two intervals separated by "/". The minimum and maximum of the interval have to be separated by a "-"
#'@param extracted.field (vector) extracted AD field of a VCF file, with the two elements of AD in different columns.
#'
#'@return
#'
#'Returns a logical vector for each variant. If true variants passed the filter.
#'
#'@export filterByADInterval
#'
filterByADInterval<- function(query.value, extracted.field){

  query.value<- (unlist(strsplit(query.value, split="/")))
  logical1<- filterByInterval(query.value[1], extracted.field[,1])
  logical2<- filterByInterval(query.value[2], extracted.field[,2])
  return(logical1 & logical2)
}
######################## Filter By AD frequency ########################
#'
#'filterByADFreq
#'
#'@description
#'
#'Function filters a vector finding if AD frequency is bigger than a given thershold.
#'
```

```
#'@details
#'
#'Function calculates the AD frequency from the two values of AD. Then checks if this value is bigger than a given threshold
#'
#'@usage
#'
#'filterByADFreq(query.value, extracted.field)
#'
#'@param query.value (numeric) The minimum thresholf for an AD frequency
#'@param extracted.field (vector) extracted AD field of a VCF file, with the two elements of AD in different columns.
#'
#'@return
#'
#'Returns a logical vector for each variant. If true variants passed the filter.
#'
#'@export filterByADFreq
#'
filterByADFreq<- function(query.value, extracted.field){

  extracted.field<- (extracted.field[,2]/(extracted.field[,1] + extracted.field[,2]))
  logical<- filterByMin(query.value, extracted.field)
  return(logical)
}
#################### Call filter functions #################
#'
#'filterExtractedField
#'
#'@description
#'
#'Function calls the requested filter
```

```
#'
#'@details
#'
#'Function match the function to be used as filter and calls it using the query value and
the extracted field
#'
#'@usage
#'
#'filterByExtractedField(extracted.field, query filter, query.value)
#'
#'@param extracted.field (vector) extracted field of a VCF file structured as a vector
#'@param query.filter (character) the name of the filter to be used.
#'@param query.value (numeric) The value to be used for filtering
#'
#'
#'@return
#'
#'Returns a logical vector for each variant. If true variants passed the filter.
#'
#'@export filterByExtractedField
#'
filterExtractedField<- function(extracted.field, query.filter, query.value){
  #Call the function matched with query type
  return(match.fun(query.filter)(query.value, extracted.field))
}



##############################################################################
##################################Multiple Query Function#################
##############################################################################
#'
```

```
#'computeMultipleQuery
#'
#'@description
#'
#'function applies filters to all the variants of an indexed VCF file. Returns a vector containing the identifiers for the variants that passed the filters
#'
#'@details
#'
#'The fucntion iterates due the rows of a dataframe containing all the queries. For each filter query the requested filter function is called returning a logical vector of all
#'variants where true means the variant passed the filter. After each filter a total logical vector is updated keeping as true only the variants which passed all the runned filters.
#'When iteration is finished the logical vector is used to obtain the identifiers of the variants which passed all the filters. This identifiers are returned. If there is not a
#'query dataframe function returns all the identifiers of vcf.
#'
#'@usage
#'
#'computeMultipleQuery(query.dataframe, field.list, tabix.dataVCF, limit.ranges=NULL, verbose = FALSE)
#'
#'@param query.dataframe (dataframe) containing a checked relation of queries
#'@param field.list (character vector) containing all the avaiable fields in the VCF file
#'@param tabix.dataVCF (tabix environtment) an environtement to acces to an indexed VCF file
#'@param limit.ranges (Granges) the ranges to limit the query to some genetic regions
#'@param verbose(logical) in case of TRUE messages are shown
#'
#'@return
#'
#'A character vector with the identifiers of all the variants which passed the filters
```

```
#'
#'@export computeMultipleQuery
#'

computeMultipleQuery<- function(query.dataframe, field.list, tabix.dataVCF, limit.ranges=NULL, verbose = FALSE){
 #If there is not query dataframe return all vcf variants
 query.dataframe<- as.data.frame(query.dataframe)
 if (nrow(query.dataframe) == 0){
  #Read VCF
  svp <- ScanVcfParam(fixed = "ALT", geno = NA, info = NA)
  readed.vcf<- readVcf(tabix.dataVCF, "hg19", svp)
  # If there are readed variants
  if (length(readed.vcf) > 0){
   expanded.vcf<- expand(readed.vcf)
   #Generate tag vector and extract field
   total.variants<- makeTagVector(expanded.vcf)
  }
 #Filter if there is query dataframe
 } else {
  total.variants<- character()
  for (j in seq_len(max(as.numeric(query.dataframe[,4])))){
   row.locations<- which(query.dataframe[,4] == j)
   new.filter.group<- TRUE
   for (i in row.locations){
    #Extract single query
    query.field<- query.dataframe[i,1]
    query.filter<- query.dataframe[i,2]
    query.value<- query.dataframe[i,3]
    query.invert<- query.dataframe[i,5]
    #Extract field
    extracted.field<- extractFieldList(query.field, field.list, tabix.dataVCF, limit.ranges)
```

```
      #If there is a extracted field
      if (is.null(extracted.field) == FALSE){

        #Filter extracted field
        if (new.filter.group){
          msg("filtering by ", query.field, verbose = verbose)
          total.logical<- filterExtractedField(extracted.field, query.filter, query.value)
          #If inclusive FALSE invert logical
          if (query.invert){
            total.logical<- !total.logical
          }
          #Print number of filtered variants
          filtered.count<- length(which(total.logical == FALSE))
          msg(filtered.count, " variants of", length(total.logical),  " filtered out", verbose =
verbose)
          new.filter.group<- FALSE
        } else {
          msg("filtering by ", query.field, verbose = verbose)
          found.logical<- filterExtractedField(extracted.field, query.filter, query.value)
          #If inclusive FALSE invert logical
          if (query.invert){
            found.logical<- !found.logical
          }
          #number of filtered out variants
          filtered.prev<- length(which(total.logical == FALSE))
          #variants wihch fulfill both filters
          total.logical<- total.logical & found.logical
          #Print number of filtered variants
          filtered.count<- length(which(total.logical == FALSE)) - filtered.prev
          msg(filtered.count, " variants of ", length(total.logical) - filtered.prev,  " filtered
out", verbose = verbose)
```

```
      }
    } else {
      total.logical<- 0
    }
  }

  if (length(which(total.logical==TRUE)) != 0){
    #Keep only identificators
    if (query.field == "AD"){
      filtered.variants<- row.names(extracted.field)
      filtered.variants<- filtered.variants[total.logical]
    } else {
      filtered.variants<- extracted.field[total.logical]
      filtered.variants<- names(filtered.variants)
    }
    #Save variants which passed the filters
    total.variants<- c(total.variants, filtered.variants)

    msg(length(filtered.variants), " passed the filters ", j, verbose = verbose)
  } else {
    msg(paste0("0 variants passed the filters ", j), verbose = verbose)
  }
 }
}
if (length(total.variants) == 0){
 total.variants<- NULL
}
if (is.null(total.variants) == FALSE){
 #Delete repeated variants and sort
 total.variants<- unique(total.variants)
 total.variants<- sort(total.variants)
}
```

```
  return(total.variants)
}

################################################################################
#
#####################  Prefilter function ################################
################################################################################
##
#'
#'vcfPrefilter
#'
#'@description
#'
#'Function prefilters a VCF file deleting all the variants with a 0/0 or 0 GT.
#'
#'@details
#'
#'Function uses the system interaction to call a shell grep which select all variants with a GT different to 0/0 or 0 of a VCF file.
#'Then stores the file in a temporal directory
#'
#'@usage
#'
#'vcfPrefilter(raw.data.vcf, basic.name, temporal.directory, verbose = FALSE)
#'
#'@param raw.data.vcf (VCF) file containing the variants to be prefiltered
#'@param basic.name (character) the name to be used to names the field.
#'@param temporal.directory (character) a temporal directory where store the prefiltered file
#'@param verbose(logical) in case of TRUE messages are shown
#'
#'
```

```
#'@return
#'
#'Returns the full path to a prefiltered VCF file.
#'
#'@export vcfPrefilter
#'

vcfPrefilter<- function(raw.data.vcf, basic.name, temporal.directory, verbose = FALSE){

 #Name prefilter file
 prefiltered.file<-  file.path(temporal.directory,  paste0("prefiltered_",  basic.name,
".vcf"))

 if (file.exists(prefiltered.file)){
  msg("Prefilter already done", verbose = verbose)
 } else {
  msg("Starting prefilter", verbose = verbose)
  if (grepl(".vcf.gz", raw.data.vcf)){
   prefilter<- paste0("zcat ", raw.data.vcf, " | grep -v '\t\\./.:' | ", "grep -v '\t.:' ", " | ",
"grep -v '\t0/0:' ",   "> ", prefiltered.file)
  } else {
   #Prefilter deleting GT = =0/0 or = 0
   prefilter<- paste0("grep -v '\t\\./.:' ", raw.data.vcf, " | ", "grep -v '\t.:' ", " | ", "grep -v
'\t0/0:' ",   "> ", prefiltered.file)
  }
  system(prefilter)
  msg("Prefiltered done", verbose = verbose)
 }

 return(prefiltered.file)
}
```

```
################################################################################
#########
####################################Answer
query##############################
################################################################################
##########

#'
#'answerQuery
#'
#'@description
#'
#'The function applies filters to a VCF and prepares an annotated output.
#'
#'@details
#'
#'The function applies the filters to a VCF file using the computeMultipleQuery funtion.
Then transforms the variant identifiers to ranges.
#'This ranges are used to create an output using the outputCreate function. If a tiled
genome is given the output is created for each part of the genome appending it
#'to the anterior part
#'
#'@usage
#'
#'answerQuery(query.dataframe, field.list, tabix.dataVCF, output.file, keep.nofiltered,
low.relevant.files, tiled.genome = NULL, table.extract = FALSE, limit.ranges = NULL,
db.dataframe = NULL, verbose = FALSE)
#'
#'@param query.dataframe (dataframe) containing a checked relation of queries
#'@param field.list (character vector) containing all the avaiable fields in the VCF file
#'@param tabix.dataVCF (tabix environtment) an environtement to acces to an indexed
VCF file
```

```
#'@param output.file (character) the path where store the output file
#'@param keep.nofiltered (logical) if TRUE the variants in the same position as the filtered variants are keeped and a column is added to identify them
#'@param low.relevant.files (character vector) a vector containing the name of the low relevant fields to be moved to the end of the output row
#'@param tiled.genome (Granges) a set of ranges containing the whole genome. Used to iterate trough the genome if its needed
#'@param table.extract (logical) if TRUE the output is a dataframe. If false the output is a VCF file.
#'@param limit.ranges (Granges) the ranges to limit the query to some genetic regions
#'@param db.dataframe (dataframe) containing the annotation request
#'@param verbose(logical) in case of TRUE messages are shown
#'
#'@return
#'
#'Stores a VCF or a dataframe with the variants which passed the filters.
#'
#'@export answerQuery
#'

answerQuery<- function(query.dataframe, field.list, tabix.dataVCF, output.file, keep.nofiltered, low.relevant.files, tiled.genome = NULL, table.extract = FALSE, limit.ranges = NULL, db.dataframe = NULL, verbose = FALSE){


 if (is.null(tiled.genome)){
   #Filter
   filtered.variants<- computeMultipleQuery(query.dataframe, field.list, tabix.dataVCF, limit.ranges, verbose = verbose)
   #Create output
   if (is.null(filtered.variants) == FALSE){
     variant.ranges<- generateRanges(filtered.variants)
```

```
    outputCreate(tabix.dataVCF, variant.ranges, filtered.variants, output.file,
keep.nofiltered, low.relevant.files, db.dataframe, table.extract = table.extract, append =
FALSE, verbose = verbose)
   } else {
     msg("No variants found", verbose = verbose)
   }
  } else {
   append<- TRUE
   for(i in seq_len(length(tiled.genome))){
     #Filter
     msg("\n part ", i, verbose = verbose)
     limit.ranges<- tiled.genome[i]
     filtered.variants<- computeMultipleQuery(query.dataframe, field.list, tabix.dataVCF,
limit.ranges, verbose = verbose)
     #Create Output
     if (is.null(filtered.variants) == FALSE){
       variant.ranges<- generateRanges(filtered.variants)
       outputCreate(tabix.dataVCF, variant.ranges, filtered.variants, output.file,
keep.nofiltered, low.relevant.files, db.dataframe, table.extract = table.extract, append =
append, verbose = verbose)
     } else {
       msg("No variants found", verbose = verbose)
     }
   }
  }
}
```

input.R

```
#########################################################################
####################Scan VCF header#####################################
```

```
#############################################################################
#' ObtainVcfFields
#'
#'@description
#'
#' Function to obtain the fields present in a VCF file
#'
#'@details
#'
#' This function uses the scanVCFHeader function of Annotated Variants package to extract all the information from the VCF header. Field names are then obtained and classified.
#'
#'@usage
#'
#' obtainVcfFields(data.vcf)
#'
#'@param data.vcf (character vector) the path to the VCF field
#'
#'@return
#'
#'Returns a list of the fields present in the VCF file
#'
#'@export ObtainVcfFields



obtainVcfFields<- function(data.vcf){

  #Scan header
  fields.vcf<- scanVcfHeader(data.vcf)
```

```
  #Extract field names
  info.fields <- rownames(info(fields.vcf))
  geno.fields<- rownames(geno(fields.vcf))
  fixed.fields<- c(names(fixed(fields.vcf)), "QUAL")
  repeated<- character()
  if (any(geno.fields %in% info.fields)){
    repeated<- geno.fields[which(geno.fields %in% info.fields)]
    geno.fields<- geno.fields[-which(geno.fields %in% repeated)]
    info.fields<- info.fields[-which(info.fields %in% repeated)]
    repeated.fields<- unlist(lapply(repeated, function(el){ c(paste(el, "geno", sep = "_"),
paste(el, "info", sep = "_")) }))
    #Create list
    field.list<- list(fixed.fields, geno.fields, info.fields, repeated.fields)
    names(field.list)<- c("exFixed", "exGeno", "exInfo", "Repeated")
  } else {

  #Create list
  field.list<- list(fixed.fields, geno.fields, info.fields)
  names(field.list)<- c("exFixed", "exGeno", "exInfo")
  }
  return(field.list)
}



#########################################################################
#####
#################################Check
query##################################
#########################################################################
#####
```

```
#' queryRead
#'
#'@description
#'
#'Function checks the queries and returns it in a dataframe
#'
#'@details
#'
#'This function obtains a query list and checks if all the parts of the query fulfill the requeriments for its posterior usage in the VCF filtering. Every single query
#'must have four elements. The first element has to be a field of the VCF file. The second one has to be the name of the filter to be applied. The third element
#'has to be a character or numeric vector to be used as factor for filtering. the fourth element must be a number grouping the different filters
#'
#'@usage
#'
#'queryRead(query ,data.vcf, verbose = FALSE)
#'
#'@param query (list) with the queries to the VCF filter
#'@param data.vcf (character vector) with the path to the VCF fiel
#'@param verbose (boolean) if TRUE messages are returned
#'
#'@return
#'
#'Returns a dataframe with the queries
#'
#'@export queryRead


queryRead<- function(query, data.vcf, verbose = FALSE){
```

```
  if (is.null(unlist(query))){
    query.dataframe<- NULL
  } else {
    if ((length(unlist(unname(query))) %% 5) != 0){
      stop("missed an element of query")
    }

    #Read query table
    query.dataframe<- as.data.frame(matrix(unlist(unname(query)), nrow = 5),
stringsAsFactors = FALSE)
    query.dataframe<- as.data.frame(t(query.dataframe), stringsAsFactors = FALSE)

    names(query.dataframe)<- c("field", "filter", "value", "adding", "inclusive")

    #Obtain vcf fields
    vcf.fields<- unlist(obtainVcfFields(data.vcf))

    #Check query
    for (i in seq_len(nrow(query.dataframe))){

      #Query field in field list
      if (!(query.dataframe$field[i] %in% vcf.fields )){
        stop(query.dataframe$field[i], "is not found in VCF header")
      }

      #Query typer in query type list
      if (!(existsFunction(query.dataframe$filter[i]))){
        stop(query.dataframe$filter[i], " is not an avaiable filter method")
      }

      #Check query factor when is interval
      if (grepl("Interval", query.dataframe$filter[i])){
```

```
    #Check if is AD interval
    if (grepl("AD", query.dataframe$filter[i])){
      query.value<- unlist(strsplit(query.dataframe$value[i], split="/"))
      query.value<- as.numeric(unlist(strsplit(query.value, split="-")))
      if (length(query.value)!= 4){
        stop("AD Interval must have four numbers")
      }
    } else {

      query.value<- as.numeric(unlist(strsplit(query.dataframe$value[i], split="-")))
      if (length(query.value)!= 2){
        stop("Interval must have two numbers")
      }
    }
   #Check query factor when is a string
  } else if (grepl("String", query.dataframe$filter[i])){

    if (!(is.character(query.dataframe$value[i]))){
      stop(query.dataframe$value[i], "must be class character")
    }

  } else {  #Check query factor when is numeric

    query.value<- as.numeric(query.dataframe$value[i])
    if (!(is.numeric(query.value))){
      stop("Factor must be class numeric")
    }
  }
 }
 query.dataframe$adding<- as.numeric(query.dataframe$adding)
}
#return cheked query dataframe
```

```
  return(query.dataframe)
}



#######################################################################
######################Create Index #################################
#######################################################################
#' IndexVCF
#'
#'@description
#'
#'Function indexes vcf files
#'
#'@details
#'
#'The function compress and indexes a VCF file. then creates a Tabix environtment
#'
#'@usage
#'
#'queryRead(data.vcf, verbose = FALSE)
#'
#'@param data.vcf (character vector) the path to the VCF file
#'@param verbose (boolean) if TRUE messages are returned
#'
#'@return
#'
#'Returns a Tabix environtment for the VCF given
#'
#'@export indexVCF
```

```
#Check if VCF has index, if not generate it
indexVCF<- function(data.vcf, verbose = FALSE){


  now.msg("Creating index", verbose = verbose)
  compressVcf<- bgzip(data.vcf)
  idx<- indexTabix(compressVcf, "vcf")
  tabix.dataVCF<- TabixFile(compressVcf, idx)
  msg("Index created", verbose = verbose)

  return(tabix.dataVCF)
}



################################################################################
#######
################# Extract field functions
########################################
################################################################################
#######
#' exFixed
#'
#'@description
#'
#'Function extracts an asked fixed field, naming it with an identifier for each variant
#'
#'@details
#'
#'The functions uses the readVCF function from VariantAnnotation package to read the
ALT field and another fixed field of a VCF.
#'Then uses the ALT field together with the variant description to make a tag using the
makeTagVector funtion.
```

```
#'The extracted fixed field is returned as a vector named with the tag.
#'
#'
#'@usage
#'
#'exFixed(query.field, tabix.dataVCF, limit.ranges=NULL)
#'
#'@param query.field (character vector) the name of the field to be extracted
#'@param tabix.dataVCF (tabix environment) a tabix environment with the direction to an indexed VCF file
#'@param limit.ranges (Granges) a granges to define the part of genome to be extracted. If not given all genome is extracted
#'
#'@return
#'
#'Returns a vector containing an extracted fixed field named with the description of each variant
#'
#'@export exfixed
#'
exFixed <- function(query.field, tabix.dataVCF, limit.ranges=NULL){
  #If is fixed field
  #Check if there are limit ranges
  if (is.null(limit.ranges)){
    svp <- ScanVcfParam(fixed = c("ALT", query.field), geno = NA, info = NA)
  } else { #If not
    svp <- ScanVcfParam(which = limit.ranges ,fixed = c("ALT", query.field), geno = NA, info = NA)
  }

  #read vcf with setted params
  readed.vcf<- readVcf(tabix.dataVCF, "hg19", svp)
```

```
  # If there are readed variants
  if (length(readed.vcf) > 0){
    expanded.vcf<- expand(readed.vcf)
    #Generate tag vector and extract field
    tag.vector<- makeTagVector(expanded.vcf)
    extracted.field<- fixed(expanded.vcf)[[3]]
    #Name extracted field with tags
    names(extracted.field)<- tag.vector
  }
  return(extracted.field)
}
#' exGeno
#'
#'@description
#'
#'Function extracts an asked geno field, naming it with an identifier for each variant
#'
#'@details
#'
#'The functions uses the readVCF function from VariantAnnotation package to read the
ALT field and a geno field of a VCF.
#'Then uses the ALT field together with the variant description to make a tag using the
makeTagVector funtion.
#'The extracted geno field is returned as a vector named with the tag.
#'
#'
#'@usage
#'
#'exGeno(query.field, tabix.dataVCF, limit.ranges=NULL)
#'
#'@param query.field (character vector) the name of the field to be extracted
```

```
#'@param tabix.dataVCF (tabix environment) a tabix environment with the direction to an indexed VCF file
#'@param limit.ranges (Granges) a granges to define the part of genome to be extracted. If not given all genome is extracted
#'
#'@return
#'
#'Returns a vector containing an extracted geno field named with the description of each variant
#'
#'@export exGeno
#'
exGeno<- function(query.field, tabix.dataVCF, limit.ranges=NULL){

 #Check if there are limit ranges
 if (is.null(limit.ranges)){
   svp <- ScanVcfParam(fixed = "ALT", geno = query.field, info = NA)
 } else {
   svp <- ScanVcfParam(which = limit.ranges ,fixed = "ALT", geno = query.field, info = NA)
 }
 #make query
 readed.vcf<- readVcf(tabix.dataVCF, "hg19", svp)
 # If there are readed variants
 if (length(readed.vcf) > 0){
   expanded.vcf<- expand(readed.vcf)
   #Generate tag vector and extract field
   tag.vector<- makeTagVector(expanded.vcf)
   if (query.field == "AD"){
     extracted.field<- geno(expanded.vcf)$AD[,,]
     row.names(extracted.field)<- tag.vector
   } else {
     extracted.field<- geno(expanded.vcf)[[1]]
```

```
    #Name extracted field with tags
    names(extracted.field)<- tag.vector
   }

 }
 return(extracted.field)
}

#' exinfo
#'
#'@description
#'
#'Function extracts an asked info field, naming it with an identifier for each variant
#'
#'@details
#'
#'The functions uses the readVCF function from VariantAnnotation package to read the ALT field and a info field of a VCF.
#'Then uses the ALT field together with the variant description to make a tag using the makeTagVector funtion.
#'The extracted info field is returned as a vector named with the tag.
#'
#'
#'@usage
#'
#'exInfo(query.field, tabix.dataVCF, limit.ranges=NULL)
#'
#'@param query.field (character vector) the name of the field to be extracted
#'@param tabix.dataVCF (tabix environment) a tabix environment with the direction to an indexed VCF file
#'@param limit.ranges (Granges) a granges to define the part of infome to be extracted. If not given all infome is extracted
```

```
#'
#'@return
#'
#'Returns a vector containing an extracted info field named with the description of each
variant
#'
#'@export exinfo
#'

exInfo<- function(query.field, tabix.dataVCF, limit.ranges=NULL){
  #Check if there are limit ranges
  if (is.null(limit.ranges)){
    svp <- ScanVcfParam(fixed = "ALT", geno = NA, info = query.field)
  } else {
    svp <- ScanVcfParam(which = limit.ranges ,fixed = "ALT", geno = NA, info = query.field)
  }
  #make query
  readed.vcf<- readVcf(tabix.dataVCF, "hg19", svp)

  # If there are readed variants
  if (length(readed.vcf) > 0){
    expanded.vcf<- expand(readed.vcf)
    #Generate tag vector and extract field
    tag.vector<- makeTagVector(expanded.vcf)
    extracted.field<- info(expanded.vcf)[[1]]
    #Name extracted field with tags
    names(extracted.field)<- tag.vector
  }
  return(extracted.field)
}
```

```
#' extractFieldList
#'
#'@description
#'
#'Function extracts the ALT field and another choosen field, toghether with the variant description from a VCF.
#'
#'@details
#'
#'The function finds where the requested field is located and chooses which function use to extract it. It is possible to specify if all the genome has to be extracted or only a given part.
#'Then choosen functionis used to extract the field from an indexed VCF file. ALT field is also extracted. The variant description and the ALT field are used to create a TAG.
#'Extracted field is returned as a vector named with the tags created. In case field is repeated, function extraxts the field from where the users has required.
#'
#'@usage
#'
#'extractFieldList(query.field, field.list, tabix.dataVCF, limit.ranges=NULL)
#'
#'@param query.field (character vector) the name of the field to be extracted
#'@param field.list (list) list of fields present in the VCF file
#'@param tabix.dataVCF (tabix environment) a tabix environment with the direction to an indexed VCF file
#'@param limit.ranges (Granges) a granges to define the part of genome to be extracted. If not given all genome is extracted
#'
#'@return
#'
#'Returns a vector with an extracted field named with the description of each variant
#'
```

```
#'@export extractFieldList
#'
extractFieldList<- function(query.field, field.list, tabix.dataVCF, limit.ranges=NULL){

 which.field.list<-   names(field.list)[sapply(seq_along(field.list),function(el){   query.field
%in% field.list[[el]]})]

 if (which.field.list == "Repeated"){
   extract.function<- unlist(strsplit(query.field, split = "_"))[2]
   query.field<- unlist(strsplit(query.field, split = "_"))[1]
   if (extract.function == "info"){
     which.field.list<- "exInfo"
   }
   if (extract.function == "geno"){
     which.field.list<- "exGeno"
   }
 }

 extracted.field<- match.fun(which.field.list)(query.field, tabix.dataVCF, limit.ranges)
 if (query.field != "AD"){
   extracted.field<- as.list(extracted.field)
 }
 return(extracted.field)
}
##########################################################################
########################
#####################################      Create      ranges      from      query
#################################
##########################################################################
########################
#' rangesRead
#'
```

```
#'@description
#'
#'Function checks a list of ranges and transforms it to a Granges object
#'
#'@details
#'
#'The function checks if exists a given list of ranges, transforms it in a dataframe and
then in a GRanges object using the toGranges function from regioneR package.
#'
#'@usage
#'
#'rangesRead(list.ranges)
#'
#'@param list.ranges(list) a list containing the ranges to be transformed
#'
#'@return
#'
#'Returns a Granges object of the given ranges
#'
#'@export rangesRead


rangesRead<-function(list.ranges){
 #If there is a list of ranges pass to dataframe and the to Granges
 if (is.null(unlist(list.ranges, recursive = TRUE))){
   limit.ranges<- NULL
 } else {
   ranges.dataframe<- data.frame(matrix(ncol = 3, nrow = length(list.ranges)))
   for(i in seq_len(length(list.ranges))){
     row<- unlist(unname(list.ranges[[i]]))
     ranges.chr<- as.character(row[1])
     ranges.start<- as.numeric(row[2])
```

```
    ranges.end<- as.numeric(row[3])
    ranges.dataframe[i,1]<- ranges.chr
    ranges.dataframe[i,2]<- ranges.start
    ranges.dataframe[i,3]<- ranges.end
  }
  colnames(ranges.dataframe)<- c("chr", "start", "end")
  limit.ranges<- toGRanges(ranges.dataframe)
  }
  return(limit.ranges)
}

################################################################################
############################
################################## Database Query process
########################################
################################################################################
############################
#' dbQueryCheck
#'
#'@description
#'
#'Function checks a list of queries to use as filters using databases and transforms it to a
dataframe
#'
#'@details
#'
#'The function checks if the query has the needed structure and if the databases
required for filtering are avaiable.
#'The transforms the given list in a dataframe
#'
#'@usage
#'
```

```
#'dbQueryCheck(input.database, verbose= FALSE)
#'
#'@param input.database(list) a list containing the queries to be checked
#'@param verbose(logical) in case of TRUE messages are shown
#'
#'@return
#'
#'Returns a checked dataframe with the queries for filtering using databases
#'
#'@export dbQueryCheck
#'
dbQueryCheck<- function(input.database, verbose= FALSE){

 if (is.null(unlist(input.database))){
  db.dataframe<- NULL
 } else {
  if ((length(unlist(unname(input.database))) %% 5) != 0){
   stop("missed an element of database query")
  }
  #Read query table
  db.dataframe<-  as.data.frame(matrix(unlist(unname(input.database)),  nrow  =  5),
stringsAsFactors = FALSE)
  db.dataframe<- as.data.frame(t(db.dataframe), stringsAsFactors = FALSE)
  names(db.dataframe)<- c("database", "db.keytype", "db.value", "Group", "inclusive")

  for (i in seq_len(nrow(db.dataframe))){
   if (existsFunction(db.dataframe$database[i]) == FALSE){
    stop(db.dataframe$database[i], " is not an avaiable database")
   }
  }
 }
```

```
  return(db.dataframe)
}

#################################### Databasse Annotation Input Check ####################################3333
#' checkAnnotate
#'
#'@description
#'
#'Function checks a list of annotation requests
#'
#'@details
#'
#'The function checks if the annotation request has the needed structure and if the databases required for annotation are avaiable.
#'The transforms the given list in a dataframe
#'
#'@usage
#'
#'checkAnnotate(input.annotate, verbose= FALSE)
#'
#'@param input.annotate(list) a list containing the annotation request to be checked
#'@param verbose(logical) in case of TRUE messages are shown
#'
#'@return
#'
#'Returns a checked dataframe with the annotation requests
#'
#'@export checkAnnotate
#'
checkAnnotate<- function(input.annotate){
```

```
  if (is.null(unlist(input.annotate, recursive = TRUE))){
    db.dataframe<- NULL
  } else {
    if ((length(unlist(unname(input.annotate))) %% 2) != 0){
      stop("missed an element of database query")
    }
    #Read query table
    db.dataframe<-   as.data.frame(matrix(unlist(unname(input.annotate)),   nrow   =   2),
stringsAsFactors = FALSE)
    db.dataframe<- as.data.frame(t(db.dataframe), stringsAsFactors = FALSE)
    names(db.dataframe)<- c("database", "keytype")

    for (i in seq_len(nrow(db.dataframe))){
      if (existsFunction(db.dataframe$database[i]) == FALSE){
        stop(db.dataframe$database[i], " is not an avaiable database")
      }
    }
  }
  return(db.dataframe)
}

#'
#'checkOutputDirectory
#'
#'@description
#'
#'The function deletes files with the same name as future output files
#'
#'@details
#'
#'The function generates the names of the output files of filtering and checks if they
already exists in  the output directory. In that case it deletes them.
```

```
#'
#'@usage
#'
#'checkOutputDirectory<- function(data.files, output.directory, script, input.directory =
NULL, ref.file = NULL)
#'
#'@param data.files (character vector) the names of all the VCF files to be filtered
#'@param output.directory (character) the path where store the output file
#'@param script (character) name of the script to be used for filtering
#'@param input.directory (character) the path  of directory where input is found
#'@param ref.file (character) the reference file in trio comparison
#'@return
#'
#'Deletes files with the same name as future output files
#'
#'@export checkOutputDirectory
#'

checkOutputDirectory<-  function(data.files,  output.directory,  script,  input.directory =
NULL, ref.file = NULL){
 #Obtain basic names input
 basic.names.input<- file_path_sans_ext(basename(data.files))
 #Obtain files in output directory
 dir.files<- list.files(output.directory)
 #Create the future output file name
 if (script == "batchWorkingFilter"){
  output.file<- paste0("results_", basic.names.input)
 }
 if (script == "compareTrio"){
  output.file<- file.path(output.directory, paste0("inherited_", basename(ref.file)))
 }
 if (script ==  "compareMultipleFiles"){
```

```
   output.file<-                file.path(output.directory,                paste0("compare_",
basename(input.directory)))
  }
  #Check if the future files already exists and remove them
  repeated.files<- dir.files[which(file_path_sans_ext(dir.files) %in% output.file)]
  file.remove(file.path(output.directory, repeated.files))
}
```

output.R

```
######################Header output#######################################
#' prepareHeaderOutput
#'
#'@description
#'
#'Function checks if all files have the same header and returns a table with all the fields
of a VCF file
#'
#'@details
#'
#'This function uses the scanVCFHeader function of Annotated Variants package to
extract all the information from the VCF header. Compares the header of all files and if
any one is different returnas an error.
#'If ther is no error field names are obtained and returned as a txt field.
#'
#'@usage
#'
#'prepareHeaderOutput(input.dir, header.output.path)
#'
#'@param input.dir (character vector) the path to the input directory
#'@param header.output.path (character vector) the path where fiel has to be stored
```

```
#'
#'@return
#'
#'Returns a txt file with all fields
#'
#'@export prepareHeaderOutput



prepareHeaderOutput<- function(input.dir, header.output.path){
 files<- list.files(input.dir)
 head.list<- list()
 for (i in seq_len(length(files))){

  data.vcf<- file.path(input.dir, files[i])
  #Scan VCF and extract fields
  fields.vcf<- scanVcfHeader(data.vcf)
  info.names<-rownames(info(fields.vcf))
  geno.names<-rownames(geno(fields.vcf))

  if (any(geno.names %in% info.names)){
    repeated<- geno.names[which(geno.names %in% info.names)]
    geno.names[which(geno.names %in% repeated)]<- paste(repeated, "geno", sep =
"_")
    info.names[which(info.names %in% repeated)]<- paste(repeated, "info", sep = "_")
  }

  info.fields<- cbind(info.names, as.data.frame.DataTable(info(fields.vcf))[2:3])
  geno.fields<- cbind(geno.names, as.data.frame.DataTable(geno(fields.vcf))[2:3])
  colnames(info.fields)[1]<- "fields"
  colnames(geno.fields)[1]<- "fields"
  head.table<- rbind(geno.fields, info.fields)
```

```
  head.list[[i]]<- head.table[,1]
 }
 #Check if all files are the same
 if (length(unique(head.list)) != 1){
  stop("Not all the VCF have the same structure")
 } else {
 #output a table
 write.table(head.table, header.output.path, quote = FALSE, append = FALSE, row.names = FALSE, col.names = FALSE, sep = "\t")
 }
}


#####################################################################
############################## Output ################################
#####################################################################

####################### Function to create dataframe from readed VCF ##########################
#'VCFToDataframe
#'
#'@description
#'
#'Function transform a VCF readed using VariantAnnotation package in a human readable dataframe.
#'
#'@details
#'
#'Function extracts the row ranges, geno fields and info fields from a readed VCF. The element of each field for each variant is pasted in order to show it as an cell
#'of a dataframe. Function gives the possibility of kepping the variants which are in the same genomic location as the ones that passed the filters. In that case a column
```

```
#'indicating if each variant passed the filters is added
#'
#'@usage
#'
#'vcfToDataframe(readed.output, keep.nofiltered, rigth.variants)
#'
#'@param readed.output (expanded VCF object) containing the variant data to be transformed into a dataframe
#'@param keep.nofiltered (logical) if TRUE the variants in the same position as the filtered variants are keeped and a column is added to identify them
#'@param right.variants (logical vector) of lenth equal as readed.output incating which variants passed the filters.
#'
#'@return
#'
#'A dataframe with a single row per variant with all the variant data
#'
#'@export vcfToDataframe


vcfToDataframe<- function(readed.output, keep.nofiltered, rigth.variants){

  #Create dataframe with id data
  out.rowranges<- as.data.frame.DataTable(rowRanges(readed.output))
  #Extract geno data
  geno.list<- geno(readed.output)
  for (i in seq_len(length(names(geno.list)))){
    if (class(geno.list[[i]][1])=="list"){
      pasted.geno<- unlist(lapply(geno.list[[i]], function(rw){ paste(unlist(rw), collapse = "/")}))
    } else {
      field.geno<- as.data.frame(geno.list[[i]])
```

```
    pasted.geno<- unlist(apply(field.geno, 1, function(rw){ paste(rw, collapse = "/") } ))
   }
   if (i == 1){
    out.geno<- pasted.geno
   } else {
    out.geno<- cbind(out.geno, pasted.geno)
   }
  }
  out.geno<- as.data.frame.DataTable(out.geno)
  names(out.geno)<- names(geno.list)

  #Extract info data
  info.list<- info(readed.output)
  for (i in seq_len(length(names(info.list)))){
   info.list[[i]]<- unlist(lapply(info.list[[i]], function(lst){ paste(unlist(lst), collapse = ", ")}))
  }
  out.info<- as.data.frame.DataTable(info.list)

  #Create dataframe
  output.dataframe<-cbind(out.rowranges, out.geno, out.info)

  #Add nonfiltered column
  if (keep.nofiltered == TRUE){
   column.names<- c(colnames(output.dataframe), "Passed_filters")
   output.dataframe<- cbind(output.dataframe, right.variants)
   colnames(output.dataframe)<- column.names
  }
  return(output.dataframe)
}

##############################Function to check readed output########################################
```

```
#'checkReadedVariants
#'
#'@description
#'
#'Function checks if a set of readed variants passed the given filters
#'
#'@details
#'
#'Function uses the makeTagVector function to create an identifier of each variant present in a readed VCF. Then compares the identifiers whit the identifiers of the variants
#'which passed the filters pointing which variants passed the filters and which don't
#'
#'@usage
#'
#'checkReadedVariants(readed.output, filtered.variants)
#'
#'@param readed.output (expanded VCF object) containing the variant data to be transformed into a dataframe
#'@param filtered.variants (character vector) with the identifiers of variants which passed the filters
#'
#'@return
#'
#'A logical vector of length equal to readed output indicating which filtered variants are in the readed output
#'
#'@export checkReadedVariants
#'
checkReadedVariants<- function(readed.output, filtered.variants) {
  #Check if there are wrong variants and eliminate them
  output.tags<- makeTagVector(readed.output)
```

```
  right.variants<- output.tags %in% filtered.variants

  return(right.variants)
}


##################################################Read output and prepare#######################################
#'
#'outputCreate
#'
#'@description
#'
#'Function extracts the data of the variants described by a variant identifier vector
#'
#'@details
#'
#'Function firsts reads the VCF file and creates an expanded VCF object. Then checks the readed variants using the CheckReadedVariants function. If table extract is TRUE
#'fucntion transforms the VCF file into a dataframe using the vcftoDataframe function. If an annotation dataframe is given, the annotateOutput function is used to add
#'annotation columns to the output dataframe. The resulting dataframe is finally stored
#'
#'@usage
#'
#'outputCreate <- function(tabix.dataVCF, variant.ranges, filtered.variants, output.file, keep.nofiltered, low.relevant.fields, db.dataframe, table.extract = FALSE, append = FALSE, inherited = NULL, verbose = FALSE)
#'
#'@param tabix.dataVCF (tabix environtment) an environtement to acces to an indexed VCF file
#'@param variant.ranges (Granges object) with the genomic location of filtered variants
```

```
#'@param filtered.variants (character vector) with the identifiers of variants which passed the filters
#'@param output.file (character) the path where store the output file
#'@param low.relevant.files (character vector) a vector containing the name of the low relevant fields to be moved to the end of the output row
#'@param keep.nofiltered (logical) if TRUE the variants in the same position as the filtered variants are keeped and a column is added to identify them
#'@param db.dataframe (dataframe) containing the annotation request
#'@param table.extract (logical) if TRUE the output is a dataframe. If false the output is a VCF file.
#'@param append (logical) if TRUE append resulting dataframe to the existing one
#'@param inherited (character vector) describing in which files each variant is found
#'@param verbose(logical) in case of TRUE messages are shown
#'
#'@return
#'
#'Stores a dataframe with all the filtered variants
#'
#'@export outputCreate

outputCreate <- function(tabix.dataVCF, variant.ranges, filtered.variants, output.file, keep.nofiltered, low.relevant.fields, db.dataframe, table.extract = FALSE, append = FALSE, inherited = NULL, verbose = FALSE){

  #Create output parameters
  output.svp<- ScanVcfParam(which = variant.ranges)

  #Read variants
  readed.output<- readVcf(tabix.dataVCF, "hg19", output.svp)
  readed.output<- unique(readed.output)
  readed.output<- expand(readed.output)
```

```
 #Check if variants have past the filter
 right.variants<- checkReadedVariants(readed.output, filtered.variants)

 if (keep.nofiltered == FALSE){
  readed.output<- readed.output[right.variants]
 }

 if (table.extract == TRUE){
  #VCF to Data frame

  output.dataframe<- vcfToDataframe(readed.output, keep.nofiltered)

  #Reorder variants
  output.dataframe<- reorderOutput(output.dataframe, low.relevant.fields)

  if (is.null(inherited) == FALSE){
   #Add hereditary column
   output.dataframe<- cbind(output.dataframe, inherited)
  }
  if (is.null(db.dataframe) == FALSE){
   output.dataframe<- annotateOutput(output.dataframe, db.dataframe)
  }
  if (file.exists(paste0(output.file, '.txt'))){
   write.table(output.dataframe, file = paste0(output.file, '.txt'), sep = "\t", quote =
FALSE, append = append, row.names = FALSE, col.names = FALSE)
  } else {
   write.table(output.dataframe, file = paste0(output.file, '.txt'), sep = "\t", quote =
FALSE, append = append, row.names = FALSE, col.names = TRUE)
   }
 } else {
  writeVcf(readed.output, filename = paste0(output.file, '.vcf'))
 }
```

```
}
######################## Reorder Output ###################################
#'reorderOutput
#'
#'@description
#'
#'Function reorders columns of output dataframe by field relevancy and rows by genomic position
#'
#'@details
#'
#'Function moves the columns defined in the low.relevant.fields vector to rightmost part of the dataframe. Then orders the variants due the genomic position
#'giving the X and Y chromosomes the 24th and 25th positions.
#'
#'@usage
#'
#'reorderOutput(output.dataframe, low.relevant.fields)
#'
#'@param output.dataframe (dataframe) containing the variant data, one row per variant
#'@param low.relevant.files (character vector) a vector containing the name of the low relevant fields to be moved to the end of the output row
#'
#'@return
#'
#'A dataframe with the variants ordered by genomic position
#'
#'@export reorderOutput
```

```
reorderOutput<- function(output.dataframe, low.relevant.fields){

 #Find low relevant positions and reorder columns
 if (length(low.relevant.fields) > 0){
 low.relevant.fields.position<- which(colnames(output.dataframe) %in%
low.relevant.fields)
 output.dataframe<- cbind(output.dataframe[,-low.relevant.fields.position],
output.dataframe[,low.relevant.fields.position])
 }
 #Reorder rows by genomic position
 chr.column<- as.character(output.dataframe[,1])
 chr.column[which(chr.column == "X")]<- 24
 chr.column[which(chr.column == "Y")]<- 25
 chr.column<- as.numeric(chr.column)
 output.dataframe<- output.dataframe[order(chr.column, output.dataframe[,2]),]

 return(output.dataframe)
}

############################################################ Compared
Output ############################################################
#'generateCompareOutput
#'
#'@description
#'
#'Function reads a VCF using given variant ranges
#'
#'@details
#'
#'Function uses the readVCF function from VariantAnnotation package to obtain a VCF
object from a VCF file
#'
```

```
#'@usage
#'
#'generateCompareOutput(tabix.dataVCF, variant.ranges, verbose = FALSE)
#'
#'@param tabix.dataVCF (tabix environtment) an environtement to acces to an indexed
VCF file
#'@param variant.ranges (Granges object) with the genomic location of filtered variants
#'@param verbose(logical) in case of TRUE messages are shown
#'
#'@return
#'
#'A VCF object with vatinats in given ranges
#'
#'@export generateCompareOutput
#'
generateCompareOutput<- function(tabix.dataVCF, variant.ranges, verbose = FALSE){

#Create output parameters
output.svp<- ScanVcfParam(which = variant.ranges)

#Read variants
readed.output<- readVcf(tabix.dataVCF, "hg19", output.svp)
readed.output<- unique(readed.output)

return(readed.output)
}

################################ Add column with file names
#################################################################
#'addFilenameColumn
#'
#'@description
```

```
#'
#'Function adds a column to a dataframe pointing the presence of each variant in different files
#'
#'@details
#'
#'Function obtains the identifier of each vector using the makeTagVector function. Then finds the variant identifier in the variant list and adds the variant list information to a
#'new dataframe column
#'
#'@usage
#'
#'addFilenameColumn(variant.list, whole.dataframe, verbose = FALSE)
#'
#'@param whole.dataframe (dataframe) containing the variant data, one row per variant
#'@param variant.list (list) containing one entry per variant with all the fields where this variant is present
#'@param verbose(logical) in case of TRUE messages are shown
#'
#'@return
#'
#'A dataframe with an extra column with the file presence
#'
#'@export addFilenameColumn

addFilenameColumn<- function(variant.list, whole.dataframe, verbose = FALSE){

  variant.tag<- makeTagVector(whole.dataframe)
  files<- matrix(ncol = 2, nrow = nrow(whole.dataframe))
  #Iterate on tags
  for (i in seq_len(length(variant.tag))){
```

```
   #Obtain file name from list
   files[i,1]<- variant.list[[variant.tag[i]]]
   files[i,2]<- length(unlist(strsplit(variant.list[[variant.tag[i]]], split = ",")))
   colnames(files)<- c("files", "Number_files")
  }
 return(cbind(whole.dataframe, files))
}
```

utils.R

```
#########################################################################
############################ Verbose functions############################
#########################################################################
#' now
#'
#'@description
#'
#'now function returns the time when something happens
#'
#'@details
#'
#'If verbose is true function returns the System time
#'
#'@usage
#'
#'now(verbose=TRUE)
#'
#'@param verbose (logical) if TRUE system time is returned
#'
#'
#'@return
```

```
#'
#'A message with the system time
#'
#'@export now

now <- function(verbose=TRUE) {
  if(verbose) {
    return(as.character(Sys.time()))
  }
}
#' msg
#'
#'@description
#'
#'function returns a message if verbose = TRUE
#'
#'@details
#'
#'If verbose is true function returns the given message
#'
#'@usage
#'
#'now(..., verbose=TRUE)
#'
#'@param ... (character) message to be returned
#'@param verbose (logical) if TRUE system time is returned
#'
#'
#'@return
#'
#'A message
#'
```

```
#'@export msg
#


msg <- function(..., verbose=TRUE) {
  if(verbose) {
    mm <- paste0(...)
    message(mm)
  }
}
#' now.msg
#'
#'@description
#'
#'function return a message and the time that something happens when verbose = TRUE
#'
#'@details
#'
#'If verbose is true function returns the given message together with the systme time
#'
#'@usage
#'
#'now.msg(..., verbose=TRUE)
#'
#'@param ... (character) message to be returned
#'@param verbose (logical) if TRUE system time is returned
#'
#'
#'@return
#'
#'A message and the system time
#'
```

```
#'@export now.msg
#

now.msg <- function(...,verbose=TRUE) {
  msg(..., verbose = verbose)
  now(verbose=verbose)
}


#################### Make variant identifier###################################
#' makeTagVector
#'
#'@description
#'
#'Function creates a variant identifier from the rowranges of a VCF field or from a dataframe.
#'
#'@details
#'
#'When given an expanded vcf file, with one variant per line, function takes the chr, start, end, ref and alt of every variant.
#'The fields taken are used to generate a vector of identifiers, pasting them. If a dataframe is given function takes the chr, start, end, ref and alt of every variant
#'and makes the identifiers
#'
#'@usage
#'
#'makeTagVector(expanded.vcf, verbose = FALSE)
#'
#'@param variants (character vector, dataframe) a vcf file with at least the ALT field loaded or a dataframe obtained from a VCF
```

```
#'@param verbose (boolean) if TRUE messages are returned
#'
#'@return
#'
#'Returns a character vector with a variant identifier for every variant
#'
#'@export makeTagVector

makeTagVector<- memoise(function(variants){
  if (is.data.frame(variants)){
    rdf<- variants
    #Add symbol of ALT in case of deletion
    alt<- rdf$ALT
    alt[which(alt == "")]<- "-"

    #Paste values in tag
    tag.vector<- paste(rdf[,1], rdf[,2], rdf[,3], rdf[,7], alt, sep = "_")

  } else {
    #Pass ranges to data frame
    rdf<- toDataframe(rowRanges(variants))
    #Add symbol of ALT in case of deletion
    alt<- rdf$ALT
    alt[which(alt == "")]<- "-"
    #Paste values in tag
    tag.vector<- paste(rdf[,1], rdf[,2], rdf[,3], rdf[,5], alt, sep = "_")
  }
  return(tag.vector)
})
```

```
##############################################################################
######################
################################ Prepare row ranges
##########################################
##############################################################################
######################
#' generateRanges
#'
#'@description
#'
#'Function transforms a set of variant identifiers in a set of Granges
#'
#'@details
#'
#'Function splits the elements of the identifier and creates a dataframe with them. Then
uses the toGranges object to creates the set Granges
#'
#'@usage
#'
#'generateRanges(filtered.variants)
#'
#'@param filtered.variants (character vector) A set of variat identifiers with the
structure: chr_start_end_ref_alt
#'
#'@return
#'
#'Returns a set of Granges
#'
#'@export makeTagVector
#'
generateRanges<- function(filtered.variants){
```

```
  #Create dataframe
  variants.dataframe<- data.frame(matrix(nrow = length(filtered.variants), ncol = 5))

  #Fill dataframe with tags
  for (i in seq_len(length(filtered.variants))){
    variants.dataframe[i,]<- unlist(strsplit(filtered.variants[i], split = "_"))
  }

  #Coerce to numeric
  variants.dataframe$X2<- as.numeric(variants.dataframe$X2)
  variants.dataframe$X3<- as.numeric(variants.dataframe$X3)

  #Add space in deletions
  variants.dataframe$X5[which(variants.dataframe$X6 == "-")]<- ""
  names(variants.dataframe)<- c("chr", "start", "end", "REF", "ALT")

  #Generate Granges
  variant.ranges<-  toGRanges(variants.dataframe[1:5])
  return(variant.ranges)
}


################################################################################
############################## Memory Check ####################################
################################################################################
#'
#'memoryCheck
#'
#'@description
#'
#'The function checks the memory usage of loading a VCF and returns the minimum number of parts to split the genome without passing the memory cutoff.
```

```
#'
#'@details
#'
#'The function generates a Granges object with the firs 10000bp of a VCF file. Then uses the Granges object to extract 2 fields and checks if the memory used to load them
#'is over a cutoff. In case the memory is over the cutoff the function calculate the minimum number of parts to split the genome without being over the cutoff.
#'
#'@usage
#'
#'memoryCheck(tabix.dataVCF, field.list, memory.cutoff, verbose = FALSE)
#'
#'@param field.list (character vector) containing all the avaiable fields in the VCF file
#'@param tabix.dataVCF (tabix environtment) an environtement to acces to an indexed VCF file
#'@param memory.cutoff (numeric) a maximum value of the variant size contained in the first 10000bp. In Mb
#'@param verbose(logical) in case of TRUE messages are shown
#'
#'@return
#'
#'A numeric with the number of parts to tile the genome
#'
#'@export memoryCheck
#'
memoryCheck<- function(tabix.dataVCF, field.list, memory.cutoff, verbose = FALSE){

 #Define query parameters
 first.chr<- seqnamesTabix(tabix.dataVCF)[1]
 test.ranges<- toGRanges(data.frame(first.chr, 10000, 20000))
 svp <- ScanVcfParam(which=test.ranges, fixed = c("ALT", "QUAL"), info = NA, geno = NA)
```

```
  #make query
  readed.vcf<- readVcf(tabix.dataVCF, "hg19", svp)
  readed.size<- (as.numeric(object.size(readed.vcf)))/(1024^2)
  msg(readed.size, verbose = verbose)

  #Define number of tile parts
  if (readed.size > memory.cutoff){
    tile.parts<- readed.size / memory.cutoff
  } else {
    tile.parts <- 1
  }

  return(tile.parts)
}


##############################################################################
######################### Split genome ##############################
##############################################################################
#'
#'splitGenome
#'
#'@description
#'
#'The function creates a set of ranges that cover the whole genome if the memory usage to load the first 10000bp is too high
#'
#'@details
#'
```

```
#'The function uses the memoryCheck function to obtain the number of parts to split the genome. In case the parts are more than one, the function loads the human genome and
#'uses the tileGenome function over the on it, setting as number of tile the memoryCheck function result and returns the parts in a Granges object
#'@usage
#'
#'memoryCheck(tabix.dataVCF, field.list, memory.cutoff, verbose = FALSE)
#'
#'@param field.list (character vector) containing all the avaiable fields in the VCF file
#'@param tabix.dataVCF (tabix environtment) an environtement to acces to an indexed VCF file
#'@param memory.cutoff (numeric) a maximum value of the variant size contained in the first 10000bp. In Mb
#'@param verbose(logical) in case of TRUE messages are shown
#'
#'@return
#'
#'A set of Granges in which the genome is splited
#'
#'@export memoryCheck
splitGenome<- function(tabix.dataVCF, field.list, memory.cutoff, verbose = FALSE){

 tile.parts<- memoryCheck(tabix.dataVCF, field.list, memory.cutoff, verbose = verbose)

 if (tile.parts > 1){
  msg("List bigger than memory cutoff", verbose = verbose)
  #Load human genome
  human.genome <- getGenomeAndMask("hg19", mask=NA)$genome
  seqlevelsStyle(human.genome) <- "Ensembl"

  #Check present chromosomes
```

```
    chr.present<- seqnamesTabix(tabix.dataVCF)

    #Tile only cromosomes present in vcf file
    hg.filtered<- filterChromosomes(human.genome, organism = "hg", keep.chr = chr.present)
    seqlengths(hg.filtered)<- width(ranges(hg.filtered))

    tiled.genome<- unlist(tileGenome(seqlengths(hg.filtered), ntile = tile.parts))
    msg("genome divided in ", length(tiled.genome), " parts", verbose = verbose)
  } else {
    tiled.genome<- NULL
  }
  return(tiled.genome)
}


addAF<- function(output.dataframe, db.keytype){
  db.keytype<- NULL
  AD<- as.character(output.dataframe$AD)
  AD<- strsplit(AD, split = "/")
  AD1<- as.numeric(unlist(AD)[c(TRUE,FALSE)])
  AD2<- as.numeric(unlist(AD)[c(FALSE,TRUE)])
  #Put AF next to AD
  AF.computed<- AD2/(AD1+AD2)

  return(AF.computed)
}
```

app.R

```
### Libraries ####
#Shiny libraries
library(shiny)
library(shinyFiles)
library(rhandsontable)
library(DT)

#Filter libraries
library(VariantAnnotation)
library(regioneR)
library(memoise)
library(yaml)
library(tools)


#Anotation libraries
library(AnnotationHub)
library(GO.db)
library(reactome.db)
library(MSigDB)
library(KEGGREST)
library(plyr)

#Sources
source("execute.R")
source("input.R")
source("output.R")
source("utils.R")
source("filters.R")
source("compare_files.R")
source("batch_working.R")
```

```
source("database.R")

#Empty head table
head.table<- matrix(ncol = 3, nrow = 1)
colnames(head.table)<- c("Field", "Object type", "Description")

#Scripts
names.scripts<- c("Batch Working", "Compare Trio", "Compare Files")
scripts<- list("batchWorkingFilter", "compareTrio", "compareMultipleFiles")
names(scripts)<- names.scripts


#Binary
binary.choose<- c(TRUE, FALSE)
names.binary<- c("yes", "no")
names(binary.choose)<- names.binary



#Filters
names.filters<- c("filter by full string", "filter by containing string", "filter by minimum",
"filter by maximum", "filter by interval", "filter by multiple string", "filter by containing
multiple string", "filter by AD interval", "filter by AD frequency")
filters<- c("filterByFullString", "filterByContainingString", "filterByMin", "filterByMax",
"filterByInterval", "filterByMultipleString", "filterByContainingMultipleString",
"filterByADInterval", "filterByADFreq")
names(filters)<- names.filters

#Database filters
db.filters<- c("GOFilter", "reactomeFilter", "MSigDBFilter", "KEGGFilter")
GOFilter.keytypes<- c("DEFINITION", "GOID", "ONTOLOGY", "TERM")
```

```
reactomeFilter.keytypes<- c("ENTREZID", "GO", "PATHID", "PATHNAME", "REACTOMEID")
MSigDBFilter.keytypes<- c("HALLMARK", "C1_POSITIONAL", "C2_CURATED", "C3_MOTIF", "C4_COMPUTATIONAL", "C5_GENE_ONTOLOGY", "C6_ONCOGENIC_SIGNATURES", "C7_IMMUNOLOGIC_SIGNATURES")
KEGGFilter.keytypes<- "NO.keytypes"
db.filter.list<- list(GOFilter.keytypes, reactomeFilter.keytypes, MSigDBFilter.keytypes, KEGGFilter.keytypes)
names(db.filter.list)<- db.filters

#Database annotate
db.annotate<- c("GoAnnotate", "reactomeAnnotate", "MSigDBAnnotate", "KEGGAnnotate", "annotateUniprotDomains", "annotateUniprotFt", "annotateCosmic", "addAF")
GO.ann<- c("DEFINITION", "GOID", "ONTOLOGY", "TERM")
reactome.ann<- c("ENTREZID", "GO", "PATHID", "PATHNAME", "REACTOMEID")
MSigDB.ann<- c("HALLMARK", "C1_POSITIONAL", "C2_CURATED", "C3_MOTIF", "C4_COMPUTATIONAL", "C5_GENE_ONTOLOGY", "C6_ONCOGENIC_SIGNATURES", "C7_IMMUNOLOGIC_SIGNATURES")
KEGG.an<- c("PATHWAY", "DISEASE", "BRITE")
uniprotdomains.ann<- "No.keytype"
uniprotfeatures.ann<- "No.keytype"
cosmic.ann<- c("Gene name", "Accession Number", "Gene CDS length", "HGNC ID", "Sample name", "ID_sample", "ID_tumour", "Primary site", "Site subtype 1", "Site subtype 2", "Site subtype 3", "Primary histology", "Histology subtype 1", "Histology subtype 2", "Histology subtype 3", "Genome-wide screen", "Mutation ID", "Mutation CDS", "Mutation AA", "Mutation Description", "Mutation zygosity", "LOH", "GRCh", "Mutation genome position", "Mutation strand", "SNP", "Resistance Mutation", "FATHMM prediction", "FATHMM score", "Mutation somatic status", "Pubmed_PMID", "ID_STUDY", "Sample source", "Tumour origin", "Age")
Allele.Frequency<- "AF"
```

```
ann.filter.list<- list(GO.ann, reactome.ann, MSigDB.ann, KEGG.an, uniprotdomains.ann,
uniprotfeatures.ann, cosmic.ann, Allele.Frequency)
names(ann.filter.list)<- db.annotate

low.relevant.fields<- paste(sep = ", ", "GT", "GQ", "MIN_DP", "PID", "RGQ", "SB", "AC",
"AF", "AN", "BaseQRankSum", "ClippingRankSum","DP_info", "DS", "END", "FS",
"HaplotypeScore", "InbreedingCoeff", "MLEAC", "MLEAF", "MQRankSum",
"NEGATIVE_TRAIN_SITE", "POSITIVE_TRAIN_SITE", "ReadPosRankSum", "SOR",
"culprit", "cytoBand", "Passed_filters")
#####################################################################
######################### UI ##########################################
#####################################################################
# Define UI for application
ui <- fluidPage(
############## HEADER ##########################################
  #Scan header
 pageWithSidebar(headerPanel("Scan VCF header"),
 #Select directories
   sidebarPanel(
     shinyDirButton("dirout",  "choose output directory", "browse"),
     br(),
     verbatimTextOutput("diroutout"),
     br(),
     shinyDirButton("dirin", "choose input directory", "browse"),
     br(),
     verbatimTextOutput("dirinout")),
 mainPanel(dataTableOutput("hout"))),
 #Scan
 actionButton("scanhead", "Scan"),
 br(),
 verbatimTextOutput("scan"),
 br(),
```

```
 verbatimTextOutput("herror"),
 br(),
#################### PARAMETERS ##########################################
 headerPanel("Script selection"),
 sidebarPanel(
  br(),
  #Select scrpit
  selectInput("script",label = "Select a script please:", choices = scripts)),
 br(),
 pageWithSidebar(headerPanel("Preferences"),

  #Select parameters
  #Logical
  sidebarPanel( radioButtons(inputId = "verbose", label = "Verbose", choices = binary.choose),
          radioButtons(inputId = "keep", label = "Keep no filtered?", choices = binary.choose, selected = FALSE),
          radioButtons(inputId = "table", label = "Extract table? (only for batch working)", choices = binary.choose)
          ),
  mainPanel (
  #Reference file
  textInput("reffile", "Entrer reference file (only for trio comparison)", "please enter reference file"),
  #memory cutoff
  textInput("cutoff", "Entrer memory cuttoff (only for batch working)", "50000"),
  #low relevant files
  textInput("lowr", "Low relevant fields", value = low.relevant.fields))),

######################### Add and Delete ranges ##########################################
```

```
  pageWithSidebar(headerPanel("Add ranges"),
            sidebarPanel(textInput("chr", "Chromosome"),
                         textInput("start", "Start"),
                         textInput("end", "End"),
                         actionButton("updater", "Add range"),
                         actionButton("deleter", "Delete range")),
            mainPanel(tableOutput("table1"))),

########################## Add and Delete filters
##########################################
  pageWithSidebar(headerPanel("Add filters"),
            sidebarPanel(selectInput("field", "Field", choices = "Scan header please"),
                         selectInput("filter", "Filter", choices = filters),
                         textInput("fvalue", "Value"),
                         selectInput("groupf", "Filter grouping", choices = c(1:10)),
                         checkboxInput(inputId = "incf", label = "Invert filter", value = FALSE),
                         actionButton("updatef", "Add Filter"),
                         actionButton("deletef", "Delete Filter")),
            mainPanel(tableOutput("table2"))),

########################## Add and Delete database filters
##########################################
  pageWithSidebar(headerPanel("Add database filters"),
            sidebarPanel(selectInput("dbfilter", "Database filter", choices = db.filters),
                         uiOutput("secondSelectiondb"),
                         textInput("dvalue", "Value"),
                         selectInput("groupd", "Filter grouping", choices = c(1:10)),
                         checkboxInput(inputId = "incd", label = "Invert filter", value = FALSE),
                         actionButton("updated", "Add db Filter"),
                         actionButton("deleted", "Delete db Filter")),
            mainPanel(tableOutput("table3"))),
```

```
######################## Add and Delete Annotation
########################################
  pageWithSidebar(headerPanel("Add Database Annotation"),
            sidebarPanel(selectInput("dbann", "Database Annotation", choices =
db.annotate),
                         uiOutput("secondSelectiona"),
                         actionButton("updatea", "Add Annotation"),
                         actionButton("deletea", "Delete Annotation")),
            mainPanel(tableOutput("table4"))),

  br(),

#################### RUN
#################################################################
  #Run script
  actionButton("yaml", "RUN FILTERS"),
  br(),
  textOutput("msg"),
  br(),
  verbatimTextOutput("error"),
  br(),
############## COVERAGE ######################################
#Scan header
  br(),
  pageWithSidebar(headerPanel("Find Coverage"),
            #Select directories
            sidebarPanel(
              shinyFilesButton("dirfile",  "choose BAM file", "browse", multiple = FALSE,
class = NULL),
              br(),
              verbatimTextOutput("dirfileout"),
              br(),
```

```
            shinyDirButton("dircov", "choose output directory", "browse"),
            br(),
            verbatimTextOutput("dircovout")),
          mainPanel(dataTableOutput("covt"))),
#Scan
 actionButton("scancov", "Get coverage"),
 br()
)
#####################################################################
###################### SERVER ######################################
#####################################################################
# Define server logic required
server <- function(input, output, session) {

######################### Scan HEADER and DIRECTORIES
#####################################

 volumes<- getVolumes()
 #Select input and output directory
 shinyDirChoose(input, "dirin", roots = volumes)
 output$dirinout<- renderText(parseDirPath(roots = volumes, selection = input$dirin))
 shinyDirChoose(input, "dirout", roots = volumes)
 output$diroutout<- renderText(parseDirPath(roots = volumes, selection =
input$dirout))


 #When click scan obtain vcf header
 observeEvent(input$scanhead,{
  #Parese paths
  input.dir<- parseDirPath(roots = volumes, selection = input$dirin)
  header.dir<- parseDirPath(roots = volumes, selection = input$dirout)
```

```
  #Scan and save header
  tryCatch(prepareHeaderOutput(input.dir, file.path(header.dir, "header.txt")),
           error = function(e){output$herror<- renderText(paste("Ups! an error has happened:", e))
           },
           finally = {output$scan<- renderText("SCAN DONE!")})


  #Prepare table for output
  head.table<-read.table(file.path(header.dir, "header.txt"), sep = "\t", quote = "", stringsAsFactors = FALSE)
  colnames(head.table)<- c("Field", "Object type", "Description")
  output$hout<- renderDataTable(datatable(head.table))#options = list(autoWidth = TRUE, scrollY= TRUE)

  #Update filter selection
  updateSelectInput(session, "field", choices = c("FILTER", "QUAL", head.table[1]))
 })
########################## Add and Delete ranges ##########################################
 valuesr <- reactiveValues()
 valuesr$df1 <- data.frame(chr = character(), start = character(), end = character(), stringsAsFactors = FALSE)
 newEntry <- observe({
  if(input$updater > 0) {
   isolate(valuesr$df1[nrow(valuesr$df1) + 1,]<- c(as.character(input$chr), as.character(input$start), as.character(input$end)))
  }
 })
 deleteEntry <- observe({
  if(input$deleter > 0) {
   isolate(valuesr$df1 <- valuesr$df1[-nrow(valuesr$df1),])
```

```
  }
 })
 output$table1 <- renderTable({valuesr$df1})

########################## Add and Delete filters ##########################################
 valuesf <- reactiveValues()
 valuesf$df2 <- data.frame(Field = character(), Filter = character(), Value = character(), Grouping = numeric(), Invert = logical(), stringsAsFactors = FALSE)
 newEntry <- observe({
  if(input$updatef > 0) {
   isolate(valuesf$df2[nrow(valuesf$df2) + 1,]<- c(as.character(input$field), as.character(input$filter), as.character(input$fvalue), as.numeric(input$groupf), as.logical(input$incf)))
  }
 })
 deleteEntry <- observe({
  if(input$deletef > 0) {
   isolate(valuesf$df2 <- valuesf$df2[-nrow(valuesf$df2),])
  }
 })
 output$table2 <- renderTable({valuesf$df2})

########################## Add and Delete database filters ##########################################
 valuesd <- reactiveValues()
 valuesd$df3 <- data.frame(DB.Filter = character(), DB.Keytype = character(), Value = character(), Grouping = numeric(), Invert = logical(), stringsAsFactors = FALSE)
 newEntry <- observe({
  if(input$updated > 0) {
```

```
    isolate(valuesd$df3[nrow(valuesd$df3) + 1,]<- c(as.character(input$dbfilter), as.character(input$keytyped), as.character(input$dvalue), as.numeric(input$groupd), as.logical(input$incd)))
   }
 })
 deleteEntry <- observe({
  if(input$deleted > 0) {
    isolate(valuesd$df3 <- valuesd$df3[-nrow(valuesd$df3),])
  }
 })
 output$table3 <- renderTable({valuesd$df3})

 #Second selection
 output$secondSelectiondb <- renderUI({
  db.choices<- db.filter.list[[grep(input$dbfilter, names(db.filter.list))]]
  selectInput("keytyped", "Keytype", choices = db.choices)
 })

 ######################### Add and Delete Annotation ##########################################
 valuesa <- reactiveValues()
 valuesa$df4 <- data.frame(Annotation.db = character(), Annotation.Keytype = character(), stringsAsFactors = FALSE)
 newEntry <- observe({
  if(input$updatea > 0) {
    isolate(valuesa$df4[nrow(valuesa$df4) + 1,]<- c(as.character(input$dbann), as.character(input$keytypea)))
  }
 })
 deleteEntry <- observe({
  if(input$deletea > 0) {
    isolate(valuesa$df4 <- valuesa$df4[-nrow(valuesa$df4),])
```

```
  }
 })
 output$table4 <- renderTable({valuesa$df4})

 #Second selection
 output$secondSelectiona <- renderUI({
  ann.choices<- ann.filter.list[[grep(input$dbann, names(ann.filter.list))]]
  selectInput("keytypea", "Keytype", choices = ann.choices)
 })



########################                                                    RUN
################################################


 #When click run
 observeEvent(input$yaml,{
  output$msg<- renderText(" ")
  #Parse and list  directory names
  input.dir<- parseDirPath(roots = volumes, selection = input$dirin)
  output.dir<- parseDirPath(roots = volumes, selection = input$dirout)
  dir.paths<- list(input.directory = input.dir, output.directory = output.dir)
  #ref file path
  ref.file<- file.path(input.dir, input$reffile)

  #Prepare filters lists
  if (nrow(valuesr$df1) == 0){
   limit.ranges<- NULL
  } else {
   limit.ranges<- split(valuesr$df1, seq_len(nrow(valuesr$df1)))
  }
```

```
  if (nrow(valuesf$df2) == 0){
    filter.list<- NULL
  } else {
    filter.list<- split(valuesf$df2, seq_len(nrow(valuesf$df2)))
  }

  if (nrow(valuesd$df3) == 0){
    db.list<- NULL
  } else {
    db.list<- split(valuesd$df3, seq_len(nrow(valuesd$df3)))
  }

  if (nrow(valuesa$df4) == 0){
    ann.list<- NULL
  } else {
  ann.list<- split(valuesa$df4, seq_len(nrow(valuesa$df4)))
  }

  #split low relevant files
  lw.relevant<- unlist(strsplit(input$lowr, split = ", "))
  #Prepare yaml list
  input.list<- list(script = input$script, verbose = input$verbose, keep_nofiltered =
input$keep, ref_file = ref.file, table_extract = input$table,
                    memory_cutoff = input$cutoff, include_genes = input$include, files =
dir.paths, limit_ranges = limit.ranges, query_list = filter.list,
                    database_list = db.list, annotate_list = ann.list, low_relevant_fields =
lw.relevant)

  temporal.directory<- tempdir()
  user.directory<- file.path(temporal.directory, basename(input.dir))
  unlink(user.directory, recursive = TRUE)
```

```
  dir.create(user.directory)

  yaml<- file.path(user.directory, "query.yaml")
  write_yaml(x = input.list, file = yaml)

  tryCatch(execute(user.directory, yaml),
           error = function(e){output$error<- renderText(paste("Ups! an error has happened:", e))
                               unlink(user.directory)},
           finally = {output$msg<- renderText("FILTERING DONE!")})
  unlink(user.directory, recursive = TRUE)
 })

######################### Coverage ######################################

 volumes<- getVolumes()
 #Select input and output directory
 shinyFileChoose(input, "dirfile", roots = volumes, session = session)
 shinyDirChoose(input, "dircov", roots = volumes)

 output$dirfileout<- renderText(paste(unlist(input$dirfile[[1]]), collapse = "/"))
 output$dircovout<- renderText(parseDirPath(roots = volumes, selection = input$dircov))

 #When click get coverage
 observeEvent(input$scancov,{
 #Parse paths
 input.bam<- paste(unlist(input$dirfile[[1]]), collapse = "/")
 cov.dir<- parseDirPath(roots = volumes, selection = input$dircovout)

 #Find coverage
 source("ExonCoverage.R", local = TRUE)
```

```
  #Prepare table for output
  cov.table<-read.table(filePath(cov.dir,   paste0("low_coverage_",   basic.name,   ".txt")),
sep = "\t", quote = "", stringsAsFactors = FALSE, header = T)
  output$covt<- renderDataTable(datatable(cov.table))

})

}

# Run the application
shinyApp(ui = ui, server = server)
```