UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA AGRONÒMICA I DEL MEDI NATURAL

ALGORITMO DE SIMULACIÓN PARA DATOS MULTIÓMICOS

TRABAJO FIN DE GRADO EN BIOTECNOLOGÍA

ALUMNO: CARLOS MARTÍNEZ MIRA
TUTORA: SONIA TARAZONA CAMPOS
COTUTORA EXTERNA: ANA CONESA CEGARRA

Curso Académico: 2014-2015

VALENCIA, SEPTIEMBRE DE 2015

Resumen

Algoritmo de simulación para datos multiómicos

Autor — D. Carlos Martínez Mira

Tutor académico — Prof. Dña. Sonia Tarazona Campos

Cotutor — Dña. Ana Conesa Cegarra

Fecha — Valencia, septiembre de 2015

Licencia — Creative Commons "Reconocimiento no Comercial - Sin Obra Derivada"

Resumen — Los avances en tecnologías de secuenciación masiva permiten la obtención de datos de expresión génica así como de sus elementos reguladores y productos a nivel genómico. La integración de estas medidas multiómicas en un modelo estadístico que explique los mecanismos de regulación en la célula es actualmente un reto importante en bioinformática y, por tanto, es importante disponer de herramientas para la validación de dichos modelos.

El laboratorio de Genómica de la Expresión Génica del Centro de Investigación Príncipe Felipe participa en el proyecto europeo STATegra, en el que se han generado datos, entre otros, de RNA-seq, miRNA-seq, methyl-seq, DNase-seq y ChIP-seq, y cuya misión es el desarrollo de modelos estadísticos para la integración de datos multiómicos.

Este trabajo ha consistido en la implementación de un algoritmo de simulación de datos multiómicos que replique experimentos reales de secuenciación para facilitar la validación (*in silico*) de los modelos estadísticos de integración. El algoritmo se ha programado en R para ser incorporado en la librería STATegRa de Bioconductor, lo que ha conllevado una apropiada documentación de las funciones desarrolladas. El algoritmo se ha basado en el estudio de los datos experimentales del proyecto STATegra y permite generar datos de expresión de genes y miRNAs, de metilación, de regulación de factores de transcripción y de accesibilidad de la cromatina para distintas condiciones experimentales y en distintos instantes de tiempo.

Palabras clave — secuenciación de nueva generación (NGS), integración de datos multiómicos, regulación de la expresión génica, bioinformática, simulación de datos ómicos.

Abstract

Abstract — Advances in massive sequencing technologies allow obtaining gene expression data and their regulators and products at the genome level. The integration of these multiomics measures in a statistical model that explains the regulatory mechanisms in the cell is now a major challenge in bioinformatics and, therefore, it is important to have tools for validating such models.

The Genomics of Gene Expression Lab at the Centro de Investigación Príncipe Felipe participates in the European STATegra project, in which they have generated, among others, RNA-seq, miRNA-seq, methyl-seq, DNase-seq and ChIP-seq data, and whose mission is to develop statistical models for the integration of multiomic data.

This work consisted in the implementation of an algorithm to simulate multiomic data that mimic real sequencing experiments to facilitate the validation (*in silico*) of statistical integration models. This algorithm was programmed in R to be incorporated into the STATegRa Bioconductor library, which meant to properly document the developed functions. The algorithm was based on the study of experimental data from the STATegra project and can generate gene and miRNA expression data, methylation data, transcription factor regulation data and chromatin accessibility data at different experimental conditions and times points.

Key words — next generation sequencing (NGS), multiomic data integration, gene expression regulation, bioinformatics, omic data simulation.

Resum

Resum — Els avanços en tecnologies de seqüenciació massiva permeten l'obtenció de dades d'expressió gènica així com dels seus elements reguladors i productes a nivell genòmic. La integració d'aquestes mesures multiòmiques en un model estadístic que explique els mecanismes de regulació en la cèl·lula és actualment un repte important en bioinformàtica i, per tant, és important disposar d'eines per a la validació d'aquests models.

El laboratori de Genòmica de l'Expressió Gènica del Centro de Investigación Príncipe Felipe participa en el projecte europeu STATegra, en el qual s'han generat dades, entre altres, de RNA-seq, miRNA-seq, methyl-seq, DNase-seq i ChIP-seq, i la seua missió és el desenvolupament de models estadístics per a la integració de dades multiòmiques.

Aquest treball ha consistit en la implementació d'un algoritme de simulació de dades multiòmiques que replique experiments reals de seqüenciació per a facilitar la validació (*in silico*) dels models estadístics d'integració. L'algoritme s'ha programat en R per a ser incorporat en la llibreria STATegRa de Bioconductor, la qual cosa ha comportat una apropiada documentació de les funcions desenvolupades. L'algoritme s'ha basat en l'estudi de les dades experimentals del projecte STATegra i permet generar dades d'expressió de gens i miRNAs, de metil·lació, de regulació de factors de transcripció i d'accessibilitat de la cromatina per a diferents condicions experimentals i en diferents instants de temps.

Paraules clau — seqüenciació de nova generació (NGS), integració de dades multiòmiques, regulació de l'expressió gènica, bioinformàtica, simulació de dades òmiques.

Agradecimientos

Este proyecto supone el final de una etapa que empezó hace 4 años y en la que pude embarcarme gracias a la generosidad de mi abuela, a la que aunque ya no esté entre nosotros quiero empezar agradeciéndole todo lo que hizo por mí.

Termino el grado dejando atrás todo tipo de momentos, pero sin duda me voy con dos grandes amigos, Andrea y Jordi, sin los que aún estaría dando vueltas buscando el aula de los exámenes de primer curso.

Los últimos meses fueron duros no solo por la recta final del último año, sino por el nacimiento de mi hijo y sus escasas ganas de dormir. Afortunadamente he contado con ayuda de personas que no han dudado en hacerse cargo cuanto hiciese falta para dejarme tiempo para la universidad, así que doy las gracias a J. Ricardo y Pilar.

También he tenido la gran suerte de realizar el proyecto en un centro de investigación con personas estupendas. Desde aquí, agradezco a todo el departamento de Genómica de la Expresión Génica del Centro de Investigación Príncipe Felipe el trato que han tenido conmigo y la comprensión de mi situación actual con mis nuevas responsabilidades. Quiero agradecer sobre todo a su directora, Ana Conesa, que me permitiera trabajar con ellos, y más que a nadie a mi tutora, Sonia Tarazona, por su gran dedicación y empujar este carro cuesta arriba incluso durante sus vacaciones y fines de semanas para llegar a tiempo, sin ella no habría sido posible.

Dejo para el final las personas más importantes en mi vida: mi familia. Agradezco a mis hermanas el apoyo que me han brindado desde el exilio, a mis padres que se sacrifiquen día a día para darnos lo mejor, a mi hijo Carlos que me alegre las mañanas (y noches) con sus sonrisas y balbuceos, y a su madre, Ana, por ser mi mayor punto de apoyo y soportarme en mis peores momentos haciendo todo lo posible por animarme.

Glosario

– **"Single responsibility principle"** Principio de diseño que establece que una clase debe tener únicamente una función y ésta debe estar contenida exclusivamente en ella.

– **Dependencias** Requerimiento de disponibilidad de librerías externas para usar la funcionalidad que proporcionan.

– **GTF** Formato de archivo utilizado para almacenar información sobre la estructura de genes.

– **Herencia** Mecanismo de reutilización y extensibilidad en programación orientada a objetos, permitiendo la creación de nuevas clases (abstracciones realizadas en programación orientada a objetos para representar una entidad o concepto) partiendo de una anterior, denominada clase padre.

– **Sobrescritura de métodos** En programación orientada a objetos, dícese de la redefinición de algún método existente para proveer una implementación diferente respecto a la clase padre.

Acrónimos

- **ADN** ácido desoxirribonucleico.
- **ARN** ácido ribonucleico.
- **C** citosina.
- **cADN** ADN complementario.
- **FC** "fold-change".
- **GTF** "gene transfer format".
- **NGS** secuenciadores de nueva generación.
- **POO** programación orientada a objetos.
- **RRBS** "reduced representation bisulfite sequencing".
- **T** timina.
- **WGBS** secuenciación genómica por bisulfito.

Índice general

Índice de tablas

Índice de figuras

1 Introducción

1.1. La expresión génica y su regulación

El estudio de la expresión de los genes constituye una de las principales vías de comprensión de la maquinaria celular, permitiendo la identificación y observación de los patrones subyacentes a la fisiología de la célula: qué genes se activan o desactivan durante el crecimiento celular, división, diferenciación o frente a diversos tratamientos con hormonas o toxinas.

En sus inicios los genes sometidos a análisis estaban cuidadosamente seleccionados debido al coste, esfuerzo y limitaciones de la tecnología existente. Desde entonces se ha experimentado un considerable avance representado principalmente por dos grandes sistemas, inicialmente "microarrays" y más tarde los secuenciadores de nueva generación (NGS).

La aparición de estas tecnologías permitió un cambio de enfoque pasando de centrarse en los patrones de un determinado grupo de genes a considerar perfiles de expresión completos, generando un gran impacto en poco tiempo en algunas áreas del conocimiento, como la genética de las enfermedades humanas (Koboldt y col., 2013). Al estudio de un determinado campo en su totalidad, de forma masiva, se le aplica el sufijo *ómica*, así, al estudio de todos los genes de un organismo, o genoma, se le denomina genómica.

Entre las distintas opciones disponibles en genómica destaca el uso de la tecnología NGS, imponiéndose para determinadas aplicaciones frente a otras alternativas, permitiendo a los investigadores generar datos más precisos y en mayor cantidad a un coste cada vez más reducido.

1.1.1. NGS

En la secuenciación del ADN, o determinación de la secuencia que lo compone, se empleó durante muchos años el sistema desarrollado por Sanger (Sanger y col., 1977), constituyendo un método muy robusto y fiable pero con diversas limitaciones para analizar genomas enteros que terminaron dando paso a la aparición de la tecnología NGS aproximadamente en el año 2002.

La posterior maduración de esta nueva tecnología permitió empezar a sustituir los antiguos secuenciadores, y la popularización de los nuevos instrumentos en los últimos años ha conllevado una bajada de costes que se ha traducido en un incremento de la cantidad de genomas secuenciados así como de los estudios a gran escala.

Una de las ventajas de los NGS es el gran volumen de información que son capaces de producir de forma económica, y que expande su utilidad más allá de determinar simplemente el orden de

la secuencia.

Actualmente existen en el mercado equipos de diferentes compañías con sistemas muy distintos, diferenciados por mecanismo de secuenciación, longitud de las lecturas resultante, tiempo en completar el proceso y otros factores (Liu y col., 2012). A pesar de contar con protocolos específicos sus pasos pueden englobarse en un procedimiento básico generalizable en los siguientes puntos, representado por la figura 1.1:

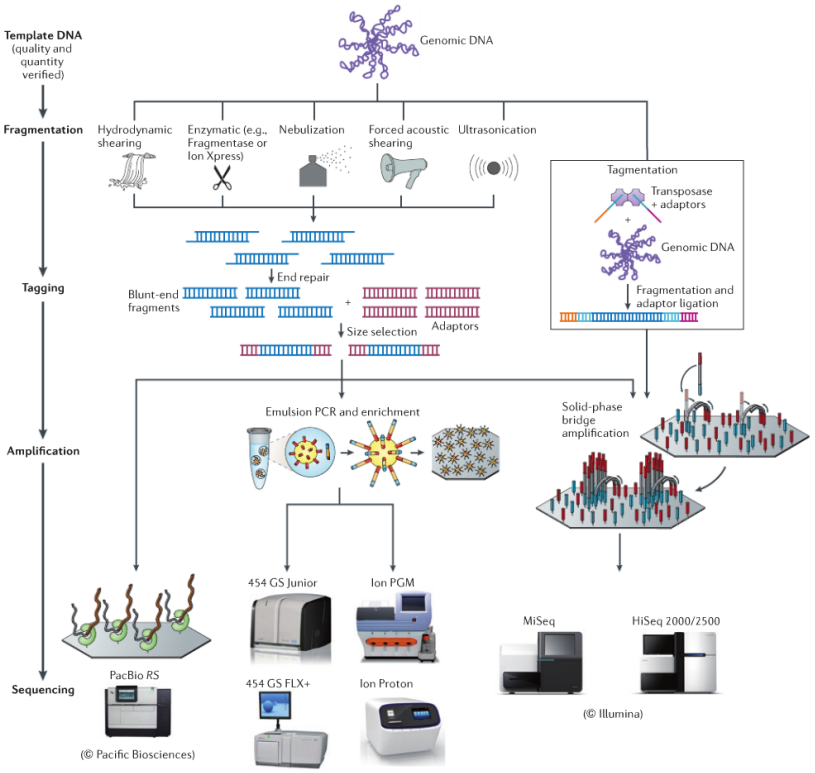


Figura 1.1: Esquema general de secuenciación masiva.

- Preparación del ADN molde
  - Fragmentación: división de la muestra inicial de ADN en fragmentos más pequeños por nebulización, digestión enzimática, etc.
  - Etiquetado: ligación de primer universal.
  - Amplificación: amplificación por PCR en emulsión o puente tras fijación a bolas o placa de cristal, respectivamente (Shendure y Ji, 2008).
- Secuenciación y captación de imágenes: sucesión de reacciones enzimáticas dependientes del mecanismo de secuenciación, como incorporación secuencial de nucleótidos bloqueados de forma reversible o uso de sondas de hibridación y ADN ligasa, seguidas de la captación de imágenes para detectar, generalmente, eventos de fluorescencia (Metzker, 2010).
- Tratamiento informático: alineamiento de las lecturas a un genoma de referencia o ensamblación *de novo* acompañado de proceso de normalización para reducir diferencias técnicas entre muestras y factores que distorsionen la distribución de datos (Aleksic y col., 2014).

1.1.2. Tecnologías derivadas de NGS

La combinación de la tecnología de secuenciación masiva junto a diversos tratamientos previos permite obtener información más allá de la secuencia de ADN, como pueden ser la transcriptómica y epigenómica, dando lugar a diferentes tecnologías que amplían aún más la información obtenible, esencial para interpretar los elementos funcionales del genoma.

RNA-seq

RNA-seq es una tecnología que emplea la secuenciación masiva para permitir identificar y cuantificar los transcritos presentes en la célula en un momento dado, es decir, la expresión de los genes. Se solventan así ciertas limitaciones de métodos anteriores ("microarrays", por ejemplo) como la necesidad de una información del genoma previa en hibridación, junto a la posibilidad de gran ruido de fondo debido a la hibridación cruzada, unido al limitado rango dinámico de detección debido al propio ruido y a la saturación (Z. Wang y col., 2009).

El protocolo de RNA-seq (figura 1.2) consiste generalmente en los siguientes pasos:

1. Selección de ARN mensajero y conversión a librería de fragmentos de cADN.
2. Secuenciación de cada molécula obteniendo secuencias cortas de 30-400 pb.
3. Análisis informático.

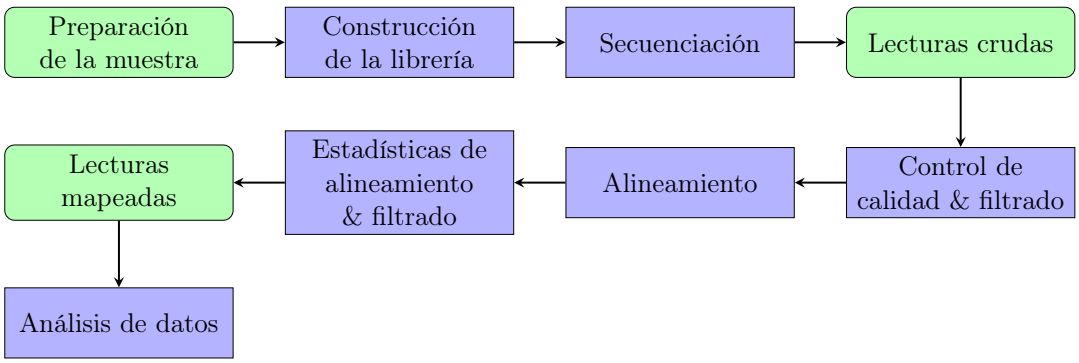


Figura 1.2: Protocolo RNA-seq.

Methyl-seq

La metilación de las bases de citosina en el ADN supone una capa de control epigenético en muchos eucariotas con importantes implicaciones en la expresión génica. La adopción de la tecnología NGS al estudio de la misma ha permitido caracterizar el metiloma con resolución de un único par de bases.

Existen distintas técnicas para determinar el perfil de metilación del ADN (Laird, 2010) entre las que destacan:

– **Digestión por endonucleasasas** basada en la actividad de una enzima digestiva dependiente del estado de metilación, detectando diferencias mediante la comparación de patrones de restricción.

– **Enriquecimiento por afinidad** obtención de regiones metiladas utilizando anticuerpos específicos para citosina metilada sobre ADN desnaturalizado.

– **Conversión por bisulfito** tratamiento del ADN desnaturalizado con bisulfito sódico, provocando la desaminación de las citosinas no metiladas mucho más rápidamente que las metiladas, transformándolas en uracilos.

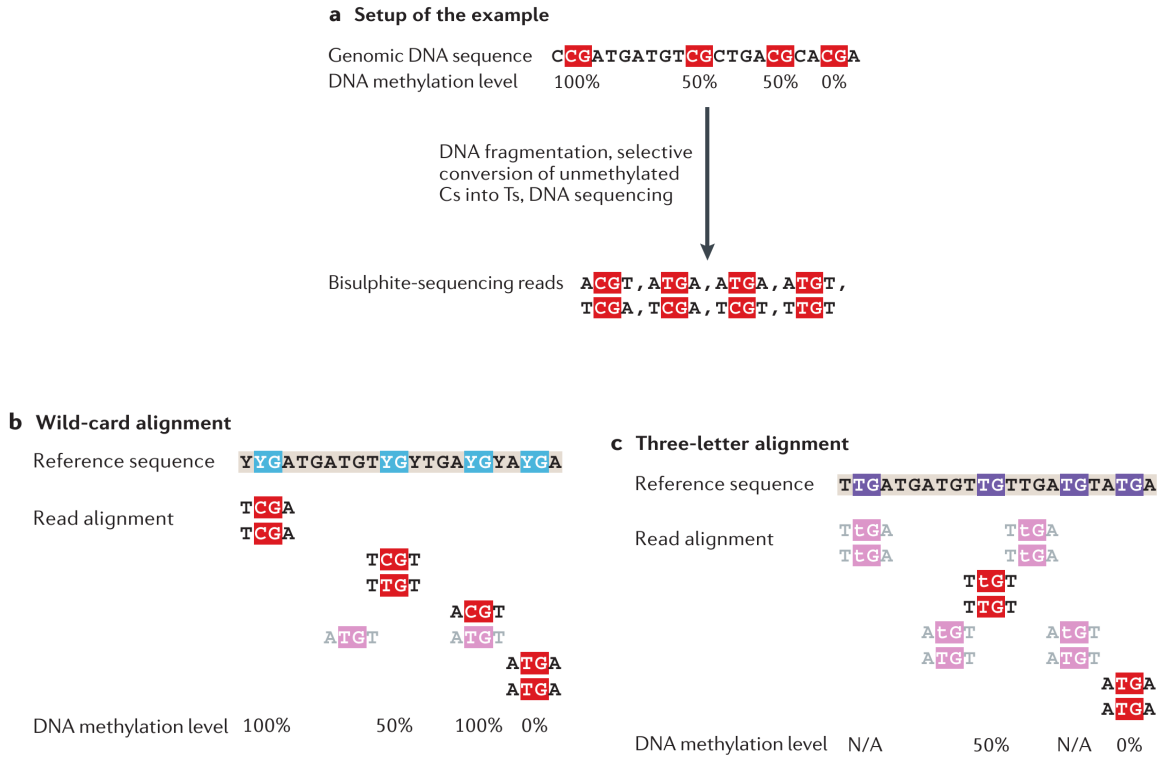


Figura 1.3: Esquema general de Methyl-seq usando secuenciación por bisulfito.

La empleada habitualmente en secuenciación es la tercera, dando lugar al proceso conocido como secuenciación genómica por bisulfito (WGBS) donde el ADN de entrada es tratado con bisulfito sódico y secuenciado.

En el proceso de alineamiento, que la citosina de una lectura solape con otra en el genoma de referencia, es indicativo de que dicha citosina está metilada en al menos una molécula de la muestra ya que no ha sido desaminada. Existen dos tipos de alineamiento, mostrados en la figura 1.3: considerando el ejemplo de secuencias de 4 bases, en el primero se reemplaza toda citosina (C) por Y en el genoma de referencia, que puede solapar tanto con C como con T, descartándose aquellas lecturas con más de un alineamiento perfecto; en el método de tres letras se reduce la complejidad del alineamiento sustituyendo todas las C por timina (T) mayúsculas en la secuencia de referencia, y por t minúsculas en las lecturas, alineando entre ellas igualmente y teniendo como resultado un incremento en el número de sitios que alinean las lecturas, siendo descartadas cuando es más de uno (Bock, 2012).

ChIP-seq

Las interacciones proteína-ADN resultan de gran importancia para entender la regulación transcripcional. Los estados de la cromatina pueden influenciar la transcripción mediante el empaquetamiento del ADN, impidiendo o facilitando el acceso de proteínas de unión a ADN o alterando la superficie del nucleosoma, con efecto activador o inhibitorio en el reclutamiento de complejos de proteínas efectores.

El estudio de todas estas interacciones se basa principalmente en la inmunoprecipitación de cromatina (ChIP), en la que se emplean anticuerpos específicos para determinadas proteínas o nucleosomas; la posterior purificación deriva en un enriquecimiento de los fragmentos de ADN unidos a estos anticuerpos. Según el siguiente paso se puede hablar de diferentes tecnologías, cuando la ChIP va seguida de una secuenciación se denomina ChIP-seq, empleando NGS para estudios que abarquen todo el genoma (P. J. Park, 2009).

El ADN purificado puede ser secuenciado en cualquier plataforma, añadiendo unos adaptadores comunes previos a la amplificación, si la hay.

DNase-seq

La mayor parte de ADN genómico está enrollado alrededor de los nucleosomas afectando así a la transcripción de los genes. Diversas modificaciones locales dan lugar a regiones que desplazan a los nucleosomas facilitando, entre otros, la acción digestiva de la enzima ADNasa I. Estos sitios hipersensibles a la ADNasa I (HS I) coinciden con diversos tipos de marcadores (activadores, promotores, etc.) por lo que la identificación de los mismos a lo largo de todo el genoma contribuye a entender la regulación génica.

En primer lugar, como puede observarse en la figura 1.5, la cromatina es digerida con una pequeña cantidad de ADNasa I que cortará preferentemente los HS I; tras esto se añade un ligando biotinilado a los extremos del resultado de la anterior digestión, usándolo para purificar los fragmentos y secuenciarlos utilizando NGS, dando lugar al proceso DNase-seq (Boyle y col., 2008).

miRNA-seq o small RNA-seq

Existen diversos tipos de ARN pequeños no codificantes en la célula que cumplen una función regulatoria en la célula; de estos los más abundantes son los microARNs, con un tamaño entre 20-23 pares de bases (Hafner y col., 2008).

La técnica miRNA-seq puede ser considerada una variación de RNA-seq, donde la creación de la librería ha sido sometida a un enriquecimiento de selección de los ARNs de menor tamaño y propiedades determinadas que más tarde son secuenciados.

La preparación de la librería, sin embargo, constituye un proceso laborioso que implica la necesidad de usar geles de poliacrilamida para el fraccionamiento y selección por tamaño, que al igual que en RNA-seq será convertido a una librería de cADN.

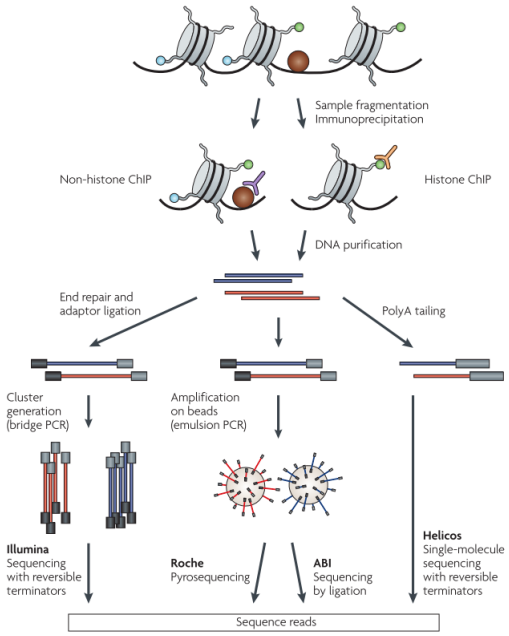


Figura 1.4: Esquema general de ChIP-seq.

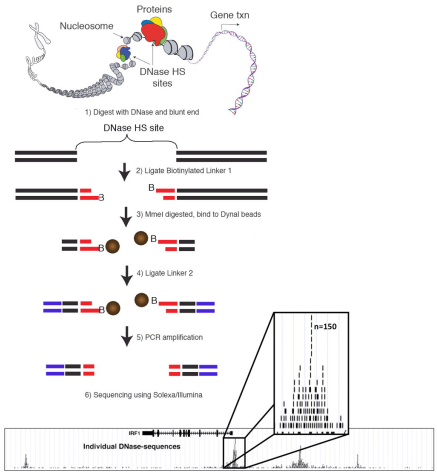


Figura 1.5: Esquema general de DNase-seq.

1.1.3. Reguladores de la expresión génica

De las distintas ómicas anteriormente expuestas podría considerarse RNA-seq como indicativo directo de la expresión génica en sí, mientras que el resto ejercería una influencia sobre las medidas de ésta, con efectos distintos.

Aunque el efecto de un regulador sobre la expresión génica depende del sistema biológico, variando con el contexto, se puede considerar la simplificación de que, generalmente, un regulador puede tener acción inhibitoria o activadora sobre la misma.

La metilación de ADN tiene funciones contrapuestas según la posición de la misma dentro de la unidad transcripcional, así, si los alrededores del sitio de inicio de la transcripción se hallan metilados la expresión se ve bloqueada, mientras que si se da en el cuerpo del gen puede llegar a tener efecto estimulatorio por elongamiento de la transcripción. En algunas zonas especiales como el centrómero actúa como supresor de la expresión de elementos transponibles. Actualmente se desconoce el efecto de la metilación en muchos de los contextos, sin embargo sí hay ciertas características que parecen ser ciertas en mamíferos (Jones, 2012):

- Muchas de las islas CpG localizadas en sitios de inicio de la transcripción no están metiladas.
- La metilación de las islas en dichos sitios de inicio está asociada con silenciamiento a largo plazo (inactivación del cromosona X, por ejemplo).
- La metilación bloquea el inicio de la transcripción pero no la elongación de la misma, aunque en algunos organismos puede ser diferente.
- Las islas CpG en el cuerpo del gen en ocasiones están metiladas, dependiendo del tejido.

- La metilación fuera de las islas CpG es más dinámica y específica a nivel de tejido.
- En elementos transponibles la metilación provoca su silenciamiento pero permite la elongación de la transcripción.

La inmunoprecipitación de cromatina aplicada al análisis de factores de transcripción permite estudiar su función regulatoria y efecto en la expresión. Aunque tradicionalmente se asume que la unión de un factor de transcripción siempre resulta en la sobre-expresión de un gen, un mismo factor puede contribuir a la activación de algunos de sus genes objetivos, y a la represión de algunos otros debido, posiblemente, a diferencias en la unión con cofactores o el contexto de la cromatina (Ouyang y col., 2009).

La compactación del ADN impide el acceso de diversas proteínas reguladoras, por lo que la remodelación de ciertas zonas con desplazamiento de nucleosomas, los sitios hipersensibles a DNasa I de los que DNase-seq proporciona información, son *a priori* más susceptibles a la unión de complejos iniciadores de la transcripción que favorezcan la expresión de los genes. En general, los genes más expresados suelen estar asociados a sitios de inicio de la transcripción que solapan con alguna zona de un sitio hipersensible a la ADNasa I, es decir, un sitio en el que la cromatina está accesible.

Existen dos mecanismos post-transcripcionales bajo los cuales los microARN pueden ejercer una acción de regulación negativa en la expresión génica: escisión de ARN mensajeros con degradación de la molécula, o represión traduccional, bien tras su iniciación ralentizándola o deteniéndola, o tras ella, degradando el producto (Bartel, 2004).

1.1.4. La expresión diferencial

Los análisis transcriptómicos permiten caracterizar y entender la variación fenotípica, siendo uno de los usos más frecuentes la comparación de perfiles mediante la búsqueda de genes diferencialmente expresados, es decir, aquellos que presentan diferencias estadísticamente significativas en su nivel de expresión entre condiciones distintas.

La detección de dichos genes requiere el empleo de herramientas bioinformáticas especializadas, basadas en la adaptación de métodos estadísticos clásicos a este contexto (Smyth, 2005; Anders y Huber, 2012b; Nueda y col., 2014) o en la implementación de nuevas estrategias (Tarazona y col., 2015a). La elección del método dependerá, entre otras cosas, del diseño experimental (comparaciones entre dos o varias poblaciones, series temporales, etc.) o de la naturaleza de los datos (medidas continuas para "microarrays" y discretas para NGS).

La utilización de los métodos de expresión diferencial puede extenderse a otras ómicas para detectar, por ejemplo, cambios en la accesibilidad de la cromatina con DNase-seq, aunque también se han desarrollado o adaptado algoritmos específicos en algunos casos como es el estudio de la metilación diferencial (Akalin y col., 2012).

La gran mayoría de estas herramientas bioinformáticas están implementadas en el lenguaje de programación R (R Core Team, 2014a).

1.2. El lenguaje de programación R

En estadística y especialmente en bioinformática, cada vez está cobrando más relevancia la utilización de R. R puede ser definido como un sistema para computación estadística y gráficos que provee un lenguaje de programación, gráficos de alto nivel, interfaz para otros lenguajes y facilidades de depuración (R Core Team, 2014b). Entre algunas de las cualidades que lo hacen especialmente indicado para el análisis de datos se encuentra la disponibilidad de herramientas estadísticas estándar incluidas por defecto que permiten cargar varios formatos, modelos estadísticos (regresión, ANOVA, GLM, modelos de árbol, etc.) y otras opciones sin necesidad de buscar implementaciones externas.

La comunidad existente entorno a este sistema es numerosa, lo que facilita encontrar gran diversidad de librerías especializadas y soporte en diversos portales. Existen repositorios públicos de librerías, siendo CRAN el más común. CRAN incluye librerías de R muy generales y de distintas disciplinas; sin embargo, las librerías que se utilizan en bioinformática suelen estar disponibles en el repositorio Bioconductor (Gentleman y col., 2004). Bioconductor se define como un proyecto de software libre destinado a proporcionar herramientas para el análisis y comprensión de datos genómicos. Debido a su naturaleza cualquier persona puede registrarse como desarrollador y enviar su programa estructurado en forma de paquete de R siempre que cumpla unos requisitos mínimos, facilitando así la descarga por parte de los usuarios y manejando automáticamente las dependencias.

1.3. La integración de datos ómicos

Como ya ha quedado establecido, en los últimos años se ha pasado a disponer de cada vez más tipos distintos de datos ómicos y en mayor cantidad. El enfoque tradicional, sin embargo, constituía principalmente el estudio individual de cada una de estas ómicas, complementando con otras fuentes en los casos que fuera necesario.

Actualmente se está cambiando el modo de trabajar y se están desarrollando nuevos métodos y estrategias de integración de datos ómicos. De este modo, cuando en un determinado sistema existan diferentes fuentes de información disponibles, se podrán estudiar conjuntamente para entender mejor los mecanismos celulares que regulan la expresión génica (Gomez-Cabrero y col., 2014).

De hecho, existen en la actualidad distintos consorcios internacionales que tienen como objetivo el estudio integrativo del genoma, utilizando para ello información de distintas ómicas (The ENCODE Project Consortium, 2004; National Cancer Institute y National Human Genome Research Institute, 2015**.**) Este trabajo se ha desarrollado en el marco de uno de estos proyectos: STATegra.

1.3.1. El proyecto STATegra

El proyecto europeo STATegra (FP7) tiene como objetivo el desarrollo de métodos estadísticos y software para la integración de datos ómicos de secuenciación masiva, proveyendo a la comunidad de recursos y herramientas sencillas para integrar y entender experimentos que involucren cierta variedad de ómicas.

El consorcio STATegra lo forman laboratorios de toda Europa y EEUU que componen un equipo multidisciplinar abarcando, entre otras disciplinas, la biología, la informática y la estadística. El grupo de Genómica de la Expresión Génica del Centro de Investigación Príncipe Felipe, donde se ha realizado este trabajo, participa en este proyecto y la Dra. Ana Conesa es la coordinadora del mismo.

Entre el software desarrollado por el consorcio STATegra se encuentra el paquete de R STATegRa, disponible en el repositorio Bioconductor para facilitar su acceso a la comunidad científica y que incluye en su versión inicial, entre otros, la definición de métodos y clases para llamar, procesar y generar gráficos de diferentes tipos de ómicas, además de contener funciones para visualización integrada de conjuntos de datos ómicos (The STATegra Consortium, 2007). Futuras versiones de este paquete contendrán los distintos métodos de análisis desarrollados dentro del proyecto para la integración de datos.

1.4. La simulación de datos ómicos

Uno de los pasos de más relevancia al desarrollar software de análisis diferencial es la comprobación de que los resultados generados son correctos, es decir, si las predicciones proporcionadas por un método coinciden con los valores reales.

La validación de los resultados de los métodos estadísticos se puede realizar empleando datos experimentales; sin embargo, en este caso se desconocen normalmente los valores reales o verdaderos. Por ejemplo, en un análisis de expresión diferencial entre un grupo de pacientes enfermos y un grupo control, desconocemos *a priori* cuáles son los genes que verdaderamente están diferencialmente expresados, por lo que difícilmente podemos saber si nuestro método se ha equivocado al predecirlos. Una alternativa es validar las predicciones utilizando otros métodos experimentales alternativos, como por ejemplo, la técnica qPCR, que permite amplificar los genes detectados como diferencialmente expresados junto a otros aleatoriamente seleccionados y comprobar la dirección (expresión al alza o a la baja) así como la dirección del cambio (Rajkumar y col., 2015). A pesar de la mayor idoneidad de probar un sistema sobre muestras reales, la limitación económica y el esfuerzo en tiempo y personal que supone no siempre resulta viable en todos los casos.

Es por ello que la simulación de datos ómicos (análisis *in silico*) es una alternativa muy adecuada que permite, con coste cercano a cero, reproducir conjuntos de datos similares a los experimentales en los que se conoce con total certeza los cambios producidos entre diferentes condiciones, sin posibilidad de confundirlos con ruido asociado y facilitando así la validación de los métodos estadísticos y su comparación en distintos escenarios biológicos.

Existen algoritmos de simulación para algunas ómicas, mayoritariamente para RNA-seq (Soneson, 2014b); sin embargo, hasta el momento no se ha desarrollado ningún software que permita simular conjuntamente distintas ómicas y las relaciones entre ellas, con el fin de validar o comparar los métodos estadísticos generados para la integración de datos multi-ómicos. Este era uno de los objetivos del proyecto STATegra, que ha sido abordado en el presente trabajo.

2

Objetivos

El objetivo del proyecto consiste en el desarrollo de un algoritmo de simulación de datos multiómicos como parte del paquete de R STATegra en Bioconductor. Dicho algoritmo deberá tener las siguientes características:

1. Simular datos de distintas ómicas medidos mediante tecnologías de secuenciación de última generación (NGS): RNA-seq, miRNA-seq, ChIP-seq, DNase-seq y RRBS-seq. Los valores generados serán los conteos que se obtienen en dichas ómicas tras procesar las lecturas de secuenciación. En ningún caso se simulan datos crudos.

2. Simular los mecanismos que regulan la expresión génica (RNA-seq) en función de las medidas obtenidas para sus reguladores (microARNs con miRNA-seq, factores de transcripción con ChIP-seq, accesibilidad de la cromatina con DNase-seq, y metilación con RRBS-seq), a través de programas regulatorios definidos a priori que puedan ser modificados por el usuario.

3. Proporcionar flexibilidad en el diseño experimental: distintas condiciones experimentales, distinto número de réplicas para cada condición experimental y series temporales con distinto número de instantes de tiempo.

Además, dado que el algoritmo formará parte de una librería de Bioconductor, debe cumplir una serie de requisitos:

1. Emplear programación orientada a objetos (POO) con fácil extensibilidad de los reguladores disponibles.

2. Documentación completa de las clases y funciones que estarán disponibles para los usuarios, incluyendo un manual de usuario.

3

Materiales y métodos

3.1. Datos ómicos del proyecto STATegra

Como parte del proyecto STATegra se dispone de datos ómicos procedentes de un diseño experimental que estudia la diferenciación de células B en ratón. Las células B son un tipo de linfocitos que provienen de células pre-B, generadas por células madre en la médula ósea.

Se estudió el sistema B3, donde la célula pasa desde el estadío pre-BI a pre-BII empleando para ello el factor de transcripción Ikaros. Usando una línea celular con una versión inducible de Ikaros se establecieron dos condiciones (control e Ikaros) diferenciadas por la activación de dicho factor de transcripción mediante la aplicación de tamoxifeno. Las muestras "control", sin Ikaros activado, no se diferenciaron a células pre-BII, a diferencia de las muestras "Ikaros".

Para cada condición, se tomaron muestras en 6 instantes de tiempo: 0, 2, 6, 12, 18 y 24 horas. Se obtuvieron tres réplicas biológicas por condición y tiempo, por lo que el número total de muestras analizadas fue 36. A partir de dichas muestras se generaron los datos de RNA-seq, miRNA-seq, DNase-seq y RRBS-seq que se utilizaron en este trabajo. También se obtuvieron datos de ChIP-seq para el estudio de los sitios de unión del factor de transcripción Ikaros. En este caso, sólo se analizaron dos condiciones experimentales equivalentes a control e Ikaros a 24 horas.

Además de los datos crudos generados por los secuenciadores, también se tuvo acceso a los conteos y a datos normalizados, resultado del trabajo de los miembros del consorcio STATegra. Así mismo, se disponía de los ficheros de asociación entre los genes y sus posibles reguladores. En el caso de los miARNs estas asociaciones habían sido obtenidas de bases de datos públicas. Para el resto de ómicas, se utilizó un algoritmo que asoció cada región genómica al gen más cercano.

Estos datos se estudiaron en profundidad para entender la variabilidad entre e intra condiciones experimentales para cada ómica, así como las relaciones entre los genes y sus reguladores.

3.2. Plataforma

El algoritmo de simulación ha sido programado y probado en un equipo con la distribución GNU/Linux Archlinux como sistema operativo, con CPU Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz y 7737712 kB útiles de memoria RAM.

La versión de R 3.2.2 con soporte mkl (Math Kernel Library) que aumenta la eficiencia

de algunas funciones matemáticas ejecutadas en procesadores Intel. Como entorno gráfico de programación se utilizó exclusivamente RStudio en su versión Version 0.99.441.

3.3. Librerías de Bioconductor o CRAN

Las dependendencias o librerías requeridas por el simulador, proporcionadas por la instalación estándar de R, descargadas de Bioconductor o del repositorio oficial de R (CRAN) fueron:

- **gtools** funciones básicas de diversa índole, entre las que se incluyen transformaciones logísticas inversas, mantenimiento de paquetes en R, además de pequeñas funciones de comprobación (par, impar, etc.) que facilitan el desarrollo.
- **GenomicRanges** clases contenedoras de intervalos genómicos. Tras proporcionar identificadores y posiciones de inicio fin de cada uno, permite acceder a la información de manera eficiente así como realizar operaciones rutinarias sin necesidad de volver a programarlas.
- **rtracklayer** manipulación de varios formatos de anotación, como "general feature format" (GFF) y sus derivados, empleados para contener información sobre secuencias asociadas a otros datos de interés (identificadores, localización, etc.)
- **GenomicFeatures** herramientas y métodos para manipular datos con transcritos. Además de posibilitar la descarga de información desde diversas bases de datos, permite trabajar con archivos locales y extraer información sobre las relaciones entre genes, transcritos, exones, etc.
- **matrixStats** operaciones optimizadas sobre columnas y filas de matrices, como medianas de columnas o filas, desviaciones típicas y otras.
- **logging** funciones para generación de archivos de registro que permitan llevar un control de la ejecución del programa y la posterior revisión de los mismo.
- **zoo** dependencia del simulador de metilación, con soporte para la generación de series de tiempo irregulares.
- **HiddenMarkov** dependencia del simulador de metilación para poder emplear modelos ocultos de Márkov que utiliza a nivel interno para definir localizaciones de forma aleatoria dependientes de la distancia, así como asignarles un determinado estado.
- **plyr** funciones para dividir en bloques conjuntos de datos, facilita la selección de subconjuntos.
- **reshape2** reestructuración y agregación de datos, permitiendo variar el formato entre corto (matrices ordinarias) o largo (conteniendo información redundante pero necesaria para ciertos sistemas de generación de gráficos).
- **ggplot2** sistema alternativo de generación de gráficos en R que basa su funcionamiento en capas, resultando en un código más legible.
- **grid** sistema de plantilla para gráficos, facilitando la generación de múltiples gráficos ordenados por página.
- **gridExtra** sistema de ordenación para librería grid, extendiendo las capacidades de la misma.

3.4. Simulación de datos de expresión

Para simular los datos de expresión se tomó como punto de partida el trabajo previo realizado por el grupo de Genómica de la Expresión Génica del Centro de Investigación Príncipe Felipe, donde se llevó a cabo el presente proyecto. El grupo había desarrollado, por una parte, un algoritmo de simulación de datos de expresión génica medida mediante "microarrays" para diseños experimentales de series temporales (Tarazona y col., 2012) y, por otra parte, un algoritmo de simulación de datos de RNA-seq en el que se generaban únicamente dos condiciones experimentales (Tarazona y col., 2015b), ambos programados en R.

Por tanto, en este proyecto se ha trabajado para unir y adaptar los dos algoritmos para generar datos de series temporales de RNA-seq, así como de otras ómicas como miRNA-seq, ChIP-seq o DNase-seq. A continuación se describen brevemente los algoritmos iniciales.

3.4.1. Simulación de datos de series temporales para "microarrays"

El diseño experimental en este caso contempla dos posibles factores: el tiempo y otro adicional (por ejemplo, el tratamiento). Para describir un amplio abanico de escenarios biológicos, el usuario puede modificar una serie de parámetros de entrada del algoritmo como número genes, porcentaje de genes cuya expresión cambia a lo largo del tiempo o entre tratamientos, etc. (ver figura 3.1). Así, se contemplan también distintos patrones de expresión a lo largo del tiempo: inducción continua o transitoria, represión continua o transitoria, perfil plano, etc. (ver figura 3.1). Se considera que los genes actúan en grupos llamados "clases", de forma que todos los genes de la misma clase siguen el mismo patrón temporal y bajo el mismo tratamiento o tratamientos, tal como se representa en la tabla de la parte derecha de la figura 3.1.

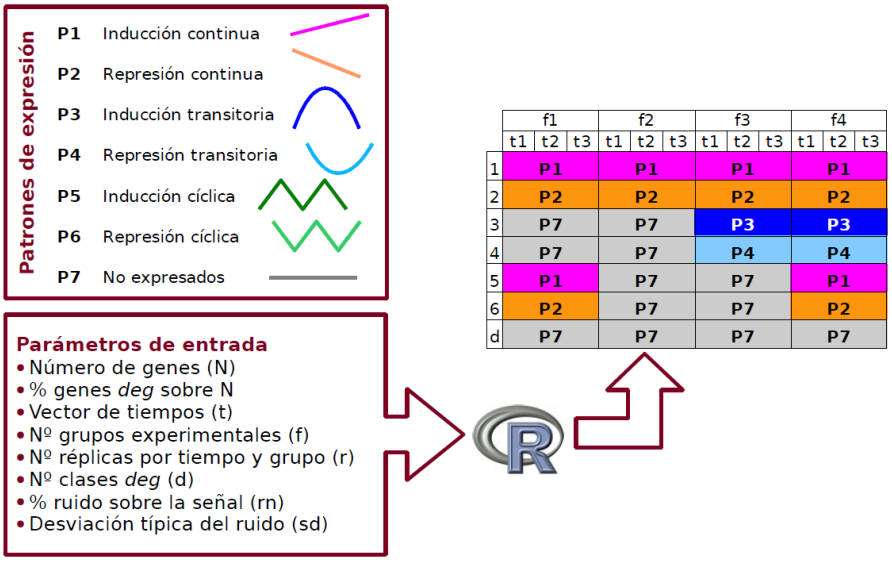


Figura 3.1: Esquema del algoritmo de simulación de datos de series temporales para "microarrays". (Tarazona y col., 2012)

La simulación de los datos de expresión para las distintas réplicas de una misma condición experimental se basa en la distribución de probabilidad normal, que es la distribución de referencia para datos de "microarrays". Los datos generados a partir de dicha distribución se corrigen para incluir distintos tipos de ruido, con tal de hacerlos más similares a los datos reales.

3.4.2. Simulación de datos de RNA-seq para dos condiciones experimentales

A diferencia de los datos de "microarrays", donde las medidas se consideran continuas y que siguen una distribución normal, los conteos procedentes de RNA-seq son enteros no negativos, es decir, valores discretos. El modelo general más usado en este caso es el de la binomial negativa. Por tanto, este algoritmo simula los datos a partir de una distribución binomial negativa cuya media coincide con los conteos iniciales proporcionados por el usuario o generados aleatoriamente, y cuya dispersión se estima a partir de estudios realizados sobre datos reales de distintos organismos (ver figura 3.2). El usuario puede elegir, entre otros, el número de réplicas por condición experimental, la proporción de genes diferencialmente expresados entre las dos condiciones simuladas, y el nivel de ruido de los datos.

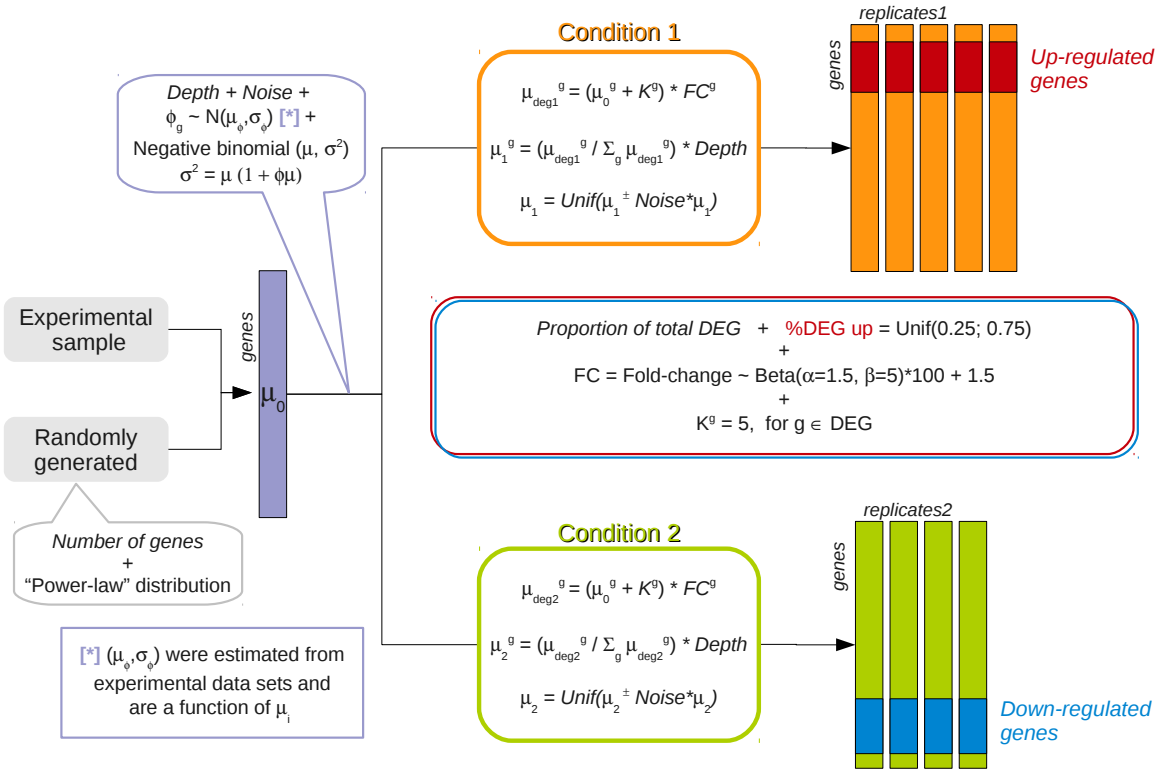


Figura 3.2: Esquema del algoritmo de simulación de datos de RNA-seq para dos condiciones iniciales.

(Tarazona y col., 2015b)

3.5. Simulación de datos de metilación

El algoritmo de simulación para datos de metilación se basó en la implementación libre del algoritmo de simulación WGBSSuite `http://www.wgbssuite.org.uk/` en su versión 0.3.

WGBSsuite es un paquete R con herramientas para análisis de datos de metilación,

comparando diversos métodos de análisis disponibles para permitir la mejor elección, así como de proporcionar un simulador de lecturas de metilación para todo el genoma. El simulador genera datos de metilación para dos condiciones experimentales y tiene en consideración la codependencia espacial de los sitios CpG, estados de metilación múltiple y número de réplicas por condición. También puede generar aleatoriamente los sitios CpG.

La metilación en ADN va asociada, generalmente, a las denominadas regiones CpG, zonas de ADN conteniendo citosinas seguidas de guaninas. La distribución de éstas suele producirse en densas agrupaciones conocidas como islas CpG, flanqueadas por zonas menos densas en contenido CpG: las orillas CpG. El estado de metilación de estas regiones tiene una codependencia espacial: a menor distancia entre dos CpG más probable es que compartan estado que si se encuentran alejadas.

En primer lugar, el algoritmo modela las localizaciones del número de sitios CpG indicados considerando tres estados: isla CpG, desierto CpG y orilla CpG. La probabilidad de cambiar entre estos tres estados viene dada por defecto en el algoritmo, pero el usuario puede modificarla.

El algoritmo considera cuatro estados distintos de metilación en cada sitio CpG: metilado, no metilado y dos estados de transición (metilado $\rightarrow$ transición 1 $\rightarrow$ no metilado y no metilado $\rightarrow$ transición 2 $\rightarrow$ metilado). La probabilidad de cambio entre los distintos estados depende de la distancia entre sitios CpG, pudiendo el usuario configurar la importancia que se le da a la misma. Posteriormente la cobertura se simula mediante una distribución de Poisson con media λ distinta según el estado de metilación.

Finalmente las lecturas se pueden modelar usando distintos tipos de distribuciones, por defecto se utiliza una distribución binomial con n_t intentos y cuyo parámetro de probabilidad de éxito p_s es distinto según estado y se ve modificado por un parámetro de ruido, $d_t \sim N(0, s_0)$, donde la varianza s_0 controla el nivel.

El simulador también incluye soporte para modelar diferencias entre dos condiciones. Los sitios CpG en los que dos condiciones comparten estado de metilación se consideran "fases"; para establecer diferencias entre ellos se aplica una "diferencia de fase" a una probabilidad de éxito p_s empleada para generar las lecturas en el paso anterior, y que se aplica a determinados sitios basándose en los siguientes criterios:

- Se selecciona aleatoriamente uno de los estados con probabilidad proporcional a la longitud de cada uno.
- Del estado seleccionado, se escoge un sitio con probabilidad uniforme (sin reemplazo).
- Se repiten los dos pasos anteriores hasta que la longitud de los sitios seleccionados en la diferencia de fase sean como máximo del 5 % de la longitud del estado simulado, garantizando así que la diferencia de fase pueda ocurrir aleatoriamente y sea solo una pequeña fracción de la región simulada.

4

Resultados

4.1. Implementación general

El algoritmo de simulación, como parte del paquete STATegRa en Bioconductor, quedó sujeto a las normas y recomendaciones aplicables al software de este repositorio. En las mismas se establece la necesidad de emplear un paradigma de programación orientada a objetos (POO) y más concretamente el sistema de clases S4 que ofrece el lenguaje R.

Entre las capacidades disponibles para el sistema S4 se aprovecharon los conceptos de herencia y sobrescritura de métodos para establecer la jerarquía de ficheros incluida en el anexo A.1. El proceso general puede ser dividido en dos bloques diferenciados, permitiendo emplear parcialmente el principio de diseño de software de "single responsibility principle" con un compromiso adecuado entre mantenimiento y complejidad, dadas las limitaciones del lenguaje de programación escogido en cuanto a aplicación de patrones de diseño:

- **Simuladores** entendidos como distintos programas independientes entre ellos, capaces de generar sus propios datos simulados con características propias a partir de la información proporcionada como opciones externas. Es decir, habría un simulador por cada ómica (RNA-seq, ChIP-seq, etc.).
- **Simulación** encargada de procesar las opciones globales del proceso tales como distribución de genes en clases o comprobación de los parámetros pasados, así como finalmente de integrar los datos generados por los componentes individuales (simuladores).

Cada ejecución del algoritmo va asociada a una serie de opciones. En *Simulación* aquellas correspondientes al diseño experimental tales como número de réplicas, condiciones experimentales o tiempos. Los parámetros específicos de cada tecnología irían en cada uno de los simuladores. Existe la posibilidad de ejecutar el algoritmo de forma sencilla sin necesidad de introducir todas las opciones disponibles. Para ello, cada uno de los parámetros configurables cuenta con valores por defecto, siempre que esto es posible.

En un entorno de ejecución con el paquete STATegRa previamente cargado, con la implementación del algoritmo actual, un ejemplo de la inicialización mínima incluyendo todas las ómicas para un experimento básico sería el siguiente:

```
opciones_simuladores <- list(
        # El parámetro "depth" también es opcional, se incluye para mostrar la forma de
        ↪  indicar opciones.
        "SimRNAseq" =  list("depth"=25),
        "SimMethylseq" = list(),
        "SimMiRNAseq" = list(),
        "SimChIPseq" = list(),
        "SimDNaseseq" = list()
)

stSim <- STATegRaSimulation(
        totalGenes = 13000,
        simulators = opciones_simuladores,
        nGeneClasses = 6,
        nRegPrograms = 9
)
```

El manual del uso del mismo puede ser consultado en Bioconductor y algunos ejemplos de su uso en el apartado 4.4.

Una vez con los parámetros de configuración globales obtenidos se procede a simular los datos de todos los simuladores cargados. El esquema general seguido por todos se muestra en la figura 4.1 y puede ser estructurado en varios pasos:

1. Inicio de la simulación.
2. Comprobación de muestra inicial: la muestra inicial puede definirse como la proporción de conteos asignados a cada fila de la matriz de datos (transcrito, región...). Si en las opciones del simulador figuran los datos iniciales procedentes de un experimento real, estos serán los que se empleen como base para el resto de operaciones; si por el contrario no han sido proporcionados se cargarán unos por defecto, existiendo soporte para generación completamente aleatoria, pero desactivada en la versión actual del algoritmo.
3. Establecer diferencias entre condiciones: si el diseño experimental contiene más de una condición la muestra inicial se copia para cada una de estas, estableciéndose posteriormente diferencias entre las mismas que varían según la ómica.
4. Modificar muestra inicial: en el caso de los reguladores, los valores de la muestra inicial se modifican para hacerlos coherentes con respecto a los de expresión, como será explicado en el apartado correspondiente.
5. Generar réplicas con ruido: por cada condición se genera el número de réplicas indicado en las opciones de simulación, considerando la variabilidad inherente a cada ómica y por tanto con parámetros distintos para cada simulador; en este proceso también se tiene en cuenta el ruido configurado según las opciones.
6. Generar serie temporal: si el diseño experimental indica la medida de una sucesión de instantes de tiempo, estos se modelan multiplicando cada valor de la muestra inicial en cada réplica por los coeficientes correspondientes al patrón temporal contemplado en cada grupo de genes y condiciones (consultar sección 4.2.6).

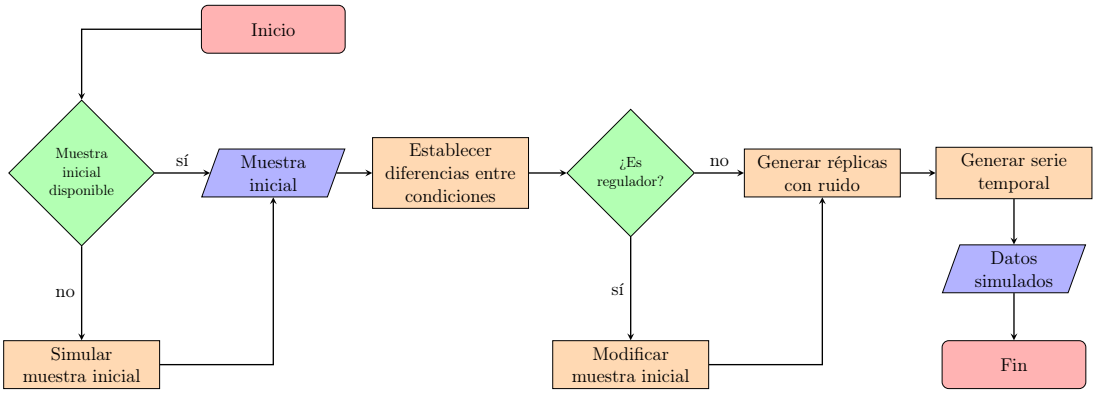


Figura 4.1: Esquema de proceso general de simulación.

Por tanto cada simulador sigue un proceso similar que puede diferir en algunos aspectos. Por ello, para facilitar la comprensión, se explica en primer lugar el simulador de RNA-seq con detalle (sección 4.2) dejando el resto de simuladores para una sección posterior (sección 4.3), donde se mencionarán las diferencias.

4.1.1. Simulación de réplicas

La simulación de réplicas de una misma condición está basada en el trabajo descrito en (Tarazona y col., 2015b), adaptándolo a las necesidades del algoritmo. Este simulador previo, como ya ha sido expuesto, basa la modelización en el uso de una distribución binomial negativa ya que la bibliografía indica que, considerando réplicas biológicas, el número de lecturas de un determinado gen puede considerarse una distribución de Poisson con sobredispersión, cuya modelización es equivalente a la binomial negativa (Soneson y Delorenzi, 2013a; Robles y col., 2012). Con el fin de facilitar la reutilización del proceso en una mayor cantidad de ómicas, la parametrización usada para modelar dicha distribución fue modificada respecto al trabajo previo.

La parametrización de una distribución binomial negativa en R puede ser establecida de dos formas:

- Utilizando la probabilidad de éxito y el número de éxitos que se desea alcanzar.
- Utilizando la media y la varianza.

Dado que el parámetro media es conocido y equivalente a los conteos de un determinado gen o región (μ), se escogió el segundo modelo de parametrización. El parámetro de dispersión "size" de la función en R, cumple que $\sigma^2 = \mu + \frac{\mu^2}{size}$, pudiéndose expresar dicha dispersión en términos de la media y la varianza.

El siguiente paso fue estudiar la variabilidad existente entre réplicas estableciendo la relación entre la expresión media y la desviación típica. Esto se hizo para RNA-seq, ChIP-seq, miRNA-seq y DNase-seq, ya que el simulador de metilación es diferente, como se verá más adelante. En primer lugar, se obtuvieron diagramas de dispersión con la media y desviación típica de cada gen o región en cada condición experimental de los datos de STATegra (ver algunos ejemplos en la figura 4.2). En estos diagramas se pudo observar que la distribución de la desviación típica es distinta para cada valor de la media, y que en general tiende a aumentar. Por ello

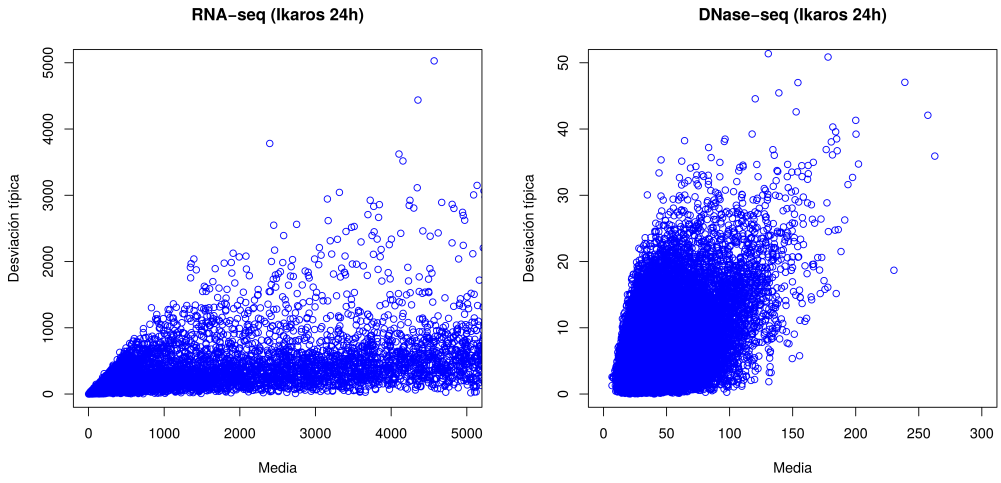


Figura 4.2: Gráfico de dispersión para la media y desviación típica. Muestras de ejemplo de RNA-seq y DNase-seq.

se decidió agrupar las medias en intervalos y estudiar la distribución de la desviación típica en cada uno de los mismos (líneas azules en el ejemplo para DNase-seq en figura 4.3), ajustando una distribución Gamma a estas distribuciones empíricas (líneas rojas en figura 4.3), que por su flexibilidad describe bastante bien el comportamiento de las distribuciones empíricas. Para cada una de las ómicas se construyeron tablas con los parámetros necesarios de la distribución Gamma en cada intervalo en el que se halle la media, que servirá posteriormente para generar aleatoriamente la desviación típica a partir de la distribución Gamma correspondiente.

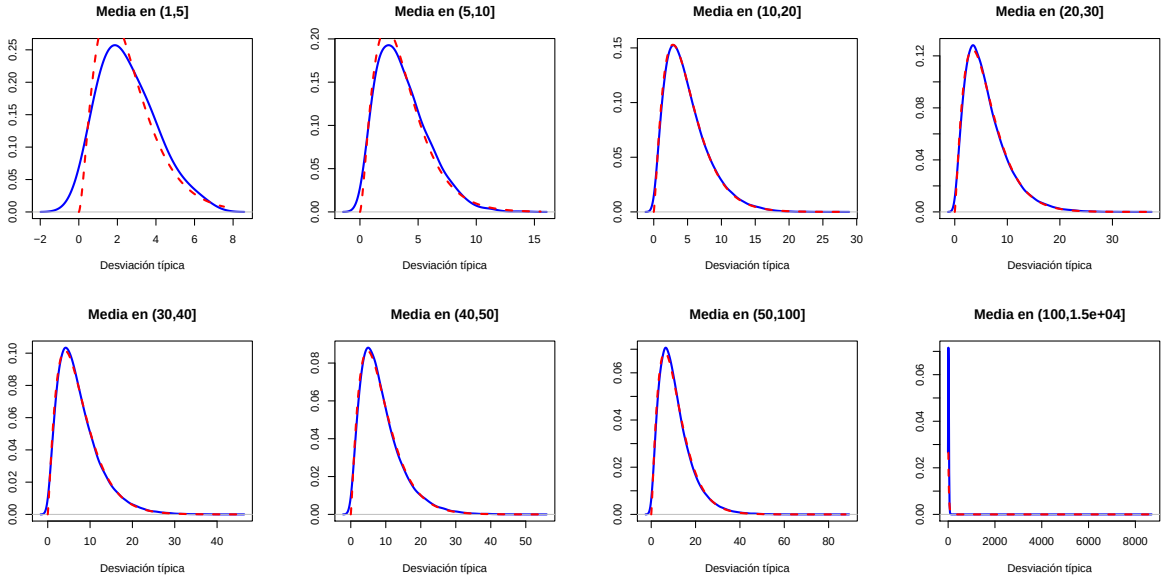


Figura 4.3: Distribución de la desviación típica en cada intervalo de la media. Ejemplo para DNase-seq.

En resumen, para modelizar los datos de RNA-seq, ChIP-seq, miRNA-seq o DNase-seq se emplea una distribución binomial negativa con la parametrización indicada anteriormente, equivalente a los conteos del gen o región (μ), y un parámetro dispersión expresado en términos

de media y desviación típica (σ), siendo modelizable esta última por una distribución Gamma cuyos parámetros varían según la media.

4.2. Simulador de RNA-seq

El análisis de datos experimentales de expresión génica permite observar grupos de genes con comportamientos de expresión similares dadas unas condiciones determinadas. Dentro de estos grupos, además, puede comprobarse que cada regulador puede tener una influencia distinta en la expresión dependiendo del gen. Estos hallazgos constituyen los principios sobre los que se basa la lógica seguida por el simulador, explicada en las secciones siguientes.

Por otra parte, con respecto al diseño experimental, se contemplan las siguientes opciones:

- **Número de condiciones experimentales** número de tratamientos, tejidos, variedades...
- **Serie temporal** instantes de tiempo considerados en el análisis.
- **Número de réplicas por condición experimental y tiempo**

Dada la finalidad de simular datos con diferencias de expresión es necesario introducir ciertas restricciones para asegurar el correcto funcionamiento. Actualmente existe soporte para simular dos condiciones experimentales con una serie de tiempos, pero el usuario podría introducir un solo instante de tiempo, o una sola condición. En estos casos se aplican consideraciones: si solo se desea simular una condición es necesario que el número de instantes de tiempo sea superior a uno para poder modelar diferencias entre ellos; por contra, si se simulan dos condiciones no será obligatorio especificar una serie temporal, pues entre las dos condiciones ya se pueden modelar las diferencias. Es decir, se previene simular una sola condición para un solo instante de tiempo.

La flexibilidad inicialmente prevista del algoritmo respecto a posibilidades de configuración implica que las definiciones de los parámetros utilizados para representar conceptos biológicos sean generales y extrapolables a varios tipos de situaciones. Considerando lo anterior, es recomendable establecer el significado de varios términos previos.

4.2.1. Definiciones previas

El término **escenario** se aplica a cada una de las combinaciones posibles entre condiciones experimentales y tiempos.

Dentro del nivel de clasificación más básico se distinguen tres tipos de genes:

- **Genes expresados** genes con igual expresión en todos los escenarios (genes planos).
- **Genes señal o diferencialmente expresados** genes cuya expresión cambia en algún instante de tiempo o condición experimental (escenario).
- **Genes no expresados** genes con conteos igual a 0 en todos los escenarios.

En aquellos diseños experimentales compuestos por series temporales, se define **patrón temporal** como el comportamiento que presenta la expresión de un gen a lo largo del tiempo, ajustable de forma aproximada a una expresión matemática, representados de forma esquemática en la tabla 4.1 (Tarazona y col., 2012). Para simplificar el algoritmo, se han descartado otros tipos de patrones.

Tabla 4.1: Patrones de expresión.

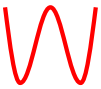
P1: inducción continua		P5: inducción cíclica	
P2: represión continua		P6: represión cíclica	
P3: inducción transitoria		P7: expresión plana	
P4: represión transitoria		No expresado	

- **Inducción continua** Aumento lineal de la actividad del gen con el transcurso del tiempo. Expresión: $a_1 + b_1 \times t[b_1 > 0]$
- **Represión continua** Descenso lineal de la actividad del gen con el transcurso del tiempo. Expresión: $a_1 + b_1 \times t[b_1 < 0]$
- **Inducción transitoria** Gen inactivo en instante inicial, aumento progresivo de su actividad con posterior descenso. Expresión $a_2 + b_2 \times t + c_2 \times t^2[b_2, c_2 > 0]$
- **Represión transitoria** Gen activo en el instante inicial, descenso progresivo de su actividad con posterior incremento. Expresión $a_2 + b_2 \times t + c_2 \times t^2[b_2, c_2 < 0]$
- **Inducción cíclica** El gen, inicialmente inactivo, se activa y reprime múltiples veces a lo largo del tiempo. Expresión: (*No definida en actual implementación*)
- **Represión cíclica** El gen, inicialmente activo, se reprime y reactiva múltiples veces a lo largo del tiempo. Expresión: (*No definida en actual implementación*)
- **Expresión plana** la actividad del gen no se ve alterada con el transcurso del tiempo.

Se define como una **clase de genes** a un grupo de genes con un perfil de expresión similar. Según el diseño del experimento, una clase englobará genes con la misma tendencia de expresión si se considera una sola condición experimental y una serie temporal, o en caso de múltiples condiciones y presencia de tiempos, aquellos con similar **patrón temporal** bajo las mismas condiciones experimentales, como muestra el ejemplo básico de la tabla 4.2.

Tabla 4.2: Ejemplos de clases de genes.

(a) Una condición experimental.

Clase	Patrón
1	
1	
3	
4	
5	
6	

(b) Varias condiciones experimentales.

Clase	Condición 1	Condición 2
1		
2		
3		
4		
5		
6		

4.2.2. Inicialización

Previamente a la generación de la configuración para simular los datos se comprueban las opciones establecidas por el usuario y corrigen, si es necesario, según ciertas restricciones:

- Si se introduce en términos absolutos en lugar de porcentaje, el número de genes expresados ha de ser igual o inferior a los genes totales.
- Si el número de condiciones es inferior a dos, los instantes de tiempo han de ser como mínimo dos.
- El número de clases de genes a simular ha de ser par (como se explica más adelante) e inferior al número posible de combinaciones entre patrones y condiciones.
- La cantidad de programas regulatorios ha de ser igual o superior al número de clases.

Todas las operaciones de selección aleatoria dependen del estado del generador de números aleatorios de R, que puede ser modificado a un determinado valor configurable en opciones para asegurar la reproducibilidad de resultados, si fuera requerido.

El software dispone de dos vías posibles para proveer el número, identificadores y posición de los genes a analizar: como primera opción revisa si se ha indicado algún fichero "gene transfer format" (GTF), del que se extraen por una parte los identificadores y por otra un objeto contenedor (GRanges) conteniendo la posición de inicio y fin de cada uno de los exones; en caso contrario, según la especificación, se generan aleatoriamente tanto los identificadores como

las posiciones, salvo que hayan sido indicadas directamente. En la actual versión del algoritmo, sin embargo, estos pasos han sido temporalmente desactivados a favor de la utilización de conjuntos de datos predefinidos o proporcionados directamente por el usuario.

Con la información disponible de los genes se procede a distribuirlos en clases y crear una variable conteniendo toda la información relativa a la simulación, posibilitando su almacenamiento y posterior carga, permitiendo así una modificación más exhaustiva si fuera necesario. La figura 4.4 muestra el primer paso de clasificación básica que se realiza.

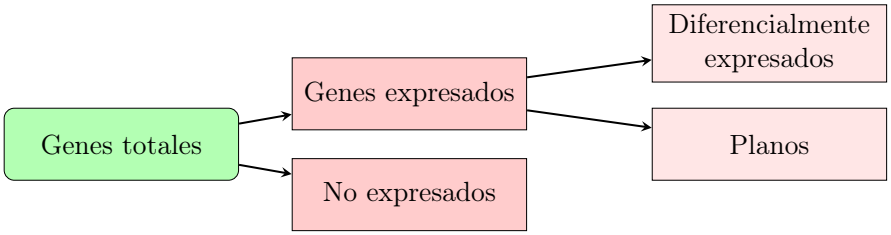


Figura 4.4: Esquema de distribución de genes.

Posteriormente se repite un ciclo de operaciones para los bloques de genes diferencialmente expresados. La primera de ellas consiste en dividirlos entre el número de clases de genes especificado. Sea D el número de genes diferencialmente expresados y C la cantidad de clases de genes a considerar; el reparto se realiza asignando a cada clase un valor $\sim D/C$ de forma que no sea exacto y oscile según el resultado de la división para hacer los resultados más realistas.

Cada clase de genes se caracteriza por un perfil de expresión único asociado siempre a un patrón temporal; en caso de considerar un solo instante de tiempo el patrón asociado será siempre plano por la implementación informática sin que ello influya en los resultados. La asignación de estos perfiles imita el comportamiento biológico y se ha fijado a las siguientes premisas:

- Cada patrón ha de aparecer al menos una vez entre las clases.
- Por cada patrón de inducción asignado a una clase, se asigna el contrario a otra clase con comportamiento de represión. Es por ello que el número de clases debe ser par.
- En caso de considerar varias condiciones experimentales, para genes diferencialmente expresados:
  - Cada clase de genes ha de expresarse al menos bajo una condición.
  - En caso de expresarse bajo varias, el patrón temporal de los genes de dicha clase ha de ser el mismo bajo todos ellos o una combinación con el patrón plano.
  - Para expresión equivalente en todas las condiciones, se modificará la magnitud de cambio en la expresión posteriormente en alguno de ellos.
- Los genes no diferencialmente expresados lo hacen bajo el patrón plano en todos los escenarios.

Las combinaciones posibles para genes diferencialmente expresados son aquellas generadas por combinatoria entre las opciones “expresado con patrón”, “no expresado” y “expresado con patrón plano”.

El proceso comienza asignando a cada clase de genes un patrón temporal distinto hasta conseguir una representación completa de cada uno, insertando en las posiciones impares aquellos

con comportamiento de inducción, y en las pares inmediatas sus opuestos de represión. Una vez completado el reparto inicial y siempre que el diseño experimental lo permita, se reasignan de forma aleatoria hasta completar el número de clases, repitiendo patrón pero bajo combinaciones de condiciones distintas, como muestra el ejemplo de la tabla 4.3.

Tabla 4.3: Asignación de patrones a clases.

(a) Una condición experimental.

Clase	Patrón
1	
2	
3	
4	

(b) Varias condiciones experimentales.

	Clase	Condición 1	Condición 2	
→	1			Por orden
→	...	...	...	
→	3			
→	...	...	...	
	5	.....		Aleatorio
	6	.....		

(a) Con una condición, se asigna directamente una clase a un patrón, pues su número coincide.

(b) Cuando el número de clases posible supera al de patrones, primero se reparten todos los patrones en orden hasta tener una representación completa de los mismos y posteriormente y de forma aleatoria se vuelven a redistribuir hasta completar el número de clases especificado.

En el ejemplo, el patrón de inducción continua queda repetido en las clases 1 y 5, pero difiere en la expresión entre condiciones. Las flechas indican clases en los que el patrón es uniforme a lo largo de los condiciones y en las que se variará la magnitud de cambio en la expresión.

Entre las combinaciones disponibles para los genes diferencialmente expresados es posible obtener el mismo patrón en todas las condiciones, por lo que resulta necesario recurrir a un paso adicional para modelar las diferencias de expresión en estos casos: la modificación del "fold-change" (FC), o magnitud de cambio en la expresión génica, en una o varias de las condiciones. En el paso actual de inicialización se modela una matriz indicando bajo qué condiciones y en qué sentido se realizarán estos cambios, información que posteriormente será empleada al generar los datos del simulador RNA-seq. Se indica un ejemplo en la tabla 4.4.

Tabla 4.4: Ejemplo de modificación de FC.

	Clase	Condición 1	Condición 2
→	1	1,0	1,5
→	2	1,0	1,5
→	3	-1,5	1,0
→	4	1,5	1,0
	5	1,0	1,0
	6	1,0	1,0

Siguiendo el ejemplo mostrado por la tabla 4.3b, las flechas indican las clases en los que el patrón de expresión era uniforme entre condiciones. Para establecer diferencias se modifica el FC al alza o baja de determinados genes en alguna de las condiciones.

4.2.3. Datos iniciales

Actualmente la muestra inicial no es generada aleatoriamente sino que es tomada, en caso de no ser proporcionada por el usuario, de datos reales (una de las muestras del proyecto STATegra). El simulador usado como base en RNA-seq, sin embargo, sí dispone de sistema para generarlas, quedando explicado aquí a efecto únicamente ilustrativo ya que en futuras versiones del algoritmo será activado.

En RNA-seq, la generación aleatoria de n genes sigue una distribución de ley potencial (Furusawa y Kaneko, 2003), $f(x) \propto x^{-\lambda}$, donde x es el nivel de expresión $0 \leq x \leq profundidad/1000$ y $\lambda = 0,5$.

Esta muestra inicial puede, opcionalmente, ser sujeta a un ajuste por longitud del gen, contando con dos funciones programadas por defecto: cuadrática (expresión 4.1) y lineal (expresión 4.2), siendo N el número de genes simulados.

$$\begin{aligned} &\forall i \in [1, \ldots, N] \\ &x = \{longitud_1, \cdots, longitud_i\} \\ &f(i) = conteos_i + conteos_i \times \left(\frac{longitud_i - \bar{x}}{\bar{x}}\right) - conteos_i \times \left(\frac{longitud_i - \bar{x}}{\bar{x}}\right)^2 \end{aligned} \tag{4.1}$$

$$\begin{aligned} &\forall i \in [1, \ldots, N] \\ &x = \{longitud_1, \cdots, longitud_i\} \\ &f(i) = conteos_i + conteos_i \times \left(\frac{longitud_i - \bar{x}}{\bar{x}}\right) \end{aligned} \tag{4.2}$$

Actualmente al emplear datos reales este ajuste no es necesario, habiendo sido programado para cuando la generación aleatoria de conteos iniciales sea reactivada, o para determinados datos reales en los que no se observe una clara presencia de este sesgo y se desee forzar la dependencia con la longitud.

4.2.4. Diferencia de expresión entre condiciones

La muestra inicial (μ_0) es copiada para cada una de las condiciones, y en pasos posteriores cuando sea transformada a varios instantes de tiempo (si se consideran) o se añadan réplicas junto a ruido, los valores serán diferentes de una condición a otra según la configuración generada y descrita en el apartado de implementación general; sin embargo, hay casos donde determinados genes comparten patrón de expresión temporal bajo los mismos escenarios, en ese caso, utilizando una matriz similar a la tabla 4.4, se modelan diferencias entre las condiciones con los cambios indicados por esa tabla de configuración. Así, si el valor leído en la tabla es 1,5 (positivo) se incrementará el nivel de expresión para los genes comprendidos en esa clase y condición, mientras que si es -1,5 se reducirá.

Este cambio de FC se genera siguiendo el procedimiento del simulador base, pero los genes a los que se aplica vienen determinados por la configuración de la simulación. Sea i cada condición en el que se produce un cambio de FC, por cada gen g dentro de la clase asociada, la

expresión media será $\mu_i^g = (\mu_0^g + K_g) \times FC^g$, donde FC es generado aleatoriamente siguiendo una distribución beta $\frac{FC^g - 1,5}{100} \sim Beta(\alpha, \beta)$ con parámetros $\alpha = 1,5$ predefinido y $\beta = 6$ configurable por el usuario. Para posibilitar la modelización de un cambio en FC cuando el valor inicial (μ_0^g) de un gen es igual a 0 se requiere sumar la constante K_g, por defecto a un valor mínimo igual a $K_g = 5$. Para finalizar todas las lecturas son ajustadas a la profundidad (*depth*) configurada en el simulador.

4.2.5. Generación de réplicas y ruido

Debido a que los valores para cada condición provienen de la misma muestra inicial (μ_0), se les aplica en primer lugar cierto nivel de ruido calculado mediante la distribución uniforme $U(\mu_i - ruido \times \mu_i, \mu_i + ruido \times \mu_i)$, aplicando una restricción de valor mínimo 0,1 para permitir que los genes inicialmente sin lecturas tengan la oportunidad de aparecer, y donde *ruido* es un valor definido en el simulador, quedando establecida así la μ_i^g final.

Una vez disponible dicho valor de expresión final para cada gen (μ_i) se halla la desviación típica (σ_i) con una distribución Gamma ($\sigma_i \sim \Gamma(k_i, \theta_i)$) con parametrización basada en forma (k) y escala (θ). Donde k_i y θ_i son los valores asociados al intervalo que comprende μ_i en las tablas generadas en el análisis de la variabilidad explicado previamente.

Los dos valores anteriores constituyen los parámetros utilizados finalmente por la distribución binomial negativa para simular las réplicas necesarias.

4.2.6. Generación de serie temporal

Una vez generados los valores para cada condición y réplica estos son transformados en una serie temporal siempre que la cantidad de instantes de tiempo proporcionados sea superior a uno; para ello cada valor es multiplicado por los coeficientes correspondientes, que coinciden en número con los tiempos considerados.

El cálculo de los coeficientes depende de cada patrón temporal y la expresión matemática que modela su comportamiento. Como ejemplo, sea P_1 el patrón con fórmula asociada $a_1 + b_1 \times t$ donde $b_1 > 0$, los coeficientes calculados para n instantes de tiempo serán:

$$\begin{bmatrix} a_1 & b_1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 \\ 0 & 1 & \cdots & n-1 & n \\ 0 & 1^2 & \cdots & (n-1)^2 & n^2 \end{bmatrix}$$

En el caso de patrones de represión, considerando que en NGS no hay conteos negativos, para emular el comportamiento de expresiones con pendiente negativa la matriz que multiplica a los coeficientes de la fórmula se invierte. Sea P_2 el patrón con fórmula asociada $a_1 + b_1 \times t$ donde $b_1 < 0$, los coeficientes calculados para n instantes de tiempo serán:

$$\begin{bmatrix} |a_1| & |b_1| & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 \\ n & n-1 & \cdots & 1 & 0 \\ n^2 & (n-1)^2 & \cdots & 1^2 & 0 \end{bmatrix}$$

El resultado final de la multiplicación de matrices es un vector con longitud igual al número

de columnas de la segunda matriz, correspondiente a los tiempos, que multiplica los conteos de los genes transformándolos en una serie temporal.

4.3. Otros simuladores

Una de las principales características del simulador es su capacidad de integrar los datos generados para mútiples ómicas. Si en las opciones de la simulación se especifican varias de ellas, los datos de las mismas se ajustarán de forma que exista una relación con los de expresión (RNA-seq).

En la definición de cada uno de los simuladores se indica el efecto de cada regulador sobre la expresión, pudiendo ser:

- **Activador** cuando aumenta la expresión génica.
- **Represor** cuando disminuye la expresión génica.

La complejidad de los sistemas biológicos hace que sea inviable simular estas interacciones sin asumir una serie de simplificaciones, siendo una de ellas considerar constante el efecto que tiene un regulador sobre la expresión génica cuando en realidad puede depender del contexto, como por ejemplo en el caso de metilación, que puede inhibir la expresión de un gen o favorecerla según la localización de las islas CpG dentro del mismo pero que, como se verá posteriormente, es considerado en todo momento un represor a efectos de la simulación.

No todos los reguladores afectan en todo momento a la expresión de todos los genes, pudiéndose distinguir la presencia de distintos **programas regulatorios** que pueden ser considerados como la participación o influencia de uno o varios reguladores en la expresión de un determinado grupo de genes. Durante la etapa de inicialización del simulador se configura un determinado número de programas regulatorios, cada uno de ellos asociado a una clase de genes (pudiendo tener una sola clase varios programas regulatorios asociados) y donde se define qué reguladores tendrán algún efecto en la expresión de los mismo, como ilustra la tabla 4.5. Esta configuración generada por el algoritmo puede ser guardada y modificada por el usuario para un uso posterior, pudiendo modificar, entre otras, cada uno de los programas regulatorios.

Tabla 4.5: Ejemplo de programas regulatorios.

El número de programas regulatorios ha de ser igual o superior al de clases de genes, por lo que una clase de genes puede constar de varios programas regulatorios.

Clase	Prog. reg	Methyl-seq	miRNA-seq	DNase-seq	ChIP-seq
1	1	✓	×	✓	×
	2	✓	✓	✓	✓
2	3	✓	×	✓	×
3	4	×	×	✓	×
	5	✓	×	×	✓
4	6	×	✓	✓	✓

El procedimiento de integración se acopla en dos puntos. El primero de ellos es tras el tercer paso general de simulación (figura 4.1), una vez generados los valores de la muestra inicial para

cada una de las condiciones, momento en que el algoritmo, cuando detecta que se trata de un regulador y que está activado para el programa regulatorio considerado en ese momento, modifica los valores correspondientes de la muestra inicial. El segundo paso, dependiente del primero, es en la generación de expresión temporal, seleccionando un patrón determinado y condicionado por el de expresión, como se verá a continuación.

Esa primera modificación mencionada depende de cada regulador y tiene en cuenta los valores de expresión que tendrán los genes una vez se conviertan en series temporales, es decir, su patrón temporal. Así, según la naturaleza activadora o inhibitoria del regulador, el patrón asociado a sus regiones o transcritos es similar u opuesto al correspondiente a los genes de RNA-seq, como indica la tabla 4.6a.

Tabla 4.6: Integración de reguladores.

(a) Asignación de patrones a reguladores.

Expresión	**Represor**	**Activador**
		
		
...	...	...

(b) Propiedades de ómicas simuladas.

Ómica	**Regulador**	**Efecto**	**Tipo**
RNA-seq	No	–	Genes
Methyl-seq	Sí	Inhibidor	Regiones
DNase-seq	Sí	Activador	Regiones
ChIP-seq	Sí	Activador	Regiones
miRNA-seq	Sí	Inhibidor	miRNA

(a) relación patrón de expresión con el de regulador según su comportamiento.

(b) reguladores disponibles en el simulador, efecto sobre expresión y tipo de identificador en sus datos.

Como ejemplo, supóngase la situación de la tabla 4.7a. El gen X, con un patrón de inducción continua, tiene un valor de expresión en la muestra inicial de 500; al generar la serie temporal (descrito en el apartado 4.2.6) el valor a tiempo inicial será ~ 0 e irá incrementándose con el tiempo. Considérese que dicho gen, localizado en la región X, está regulado por metilación (pudiendo tomar valores entre 0 y 1), y que por azar la misma tiene un valor de 0,1 en la muestra inicial. Dado el carácter inhibitorio de la metilación, el patrón asignado es el opuesto (represión continua), por lo que al modelar su serie temporal en el instante inicial su valor sería el máximo de la serie (0,1) e iría disminuyendo con el tiempo hasta llegar a ~ 0. En general, al estudiar datos reales, se ve que cuando un gen no se expresa y hay metilación en su región asociada, los porcentajes de ésta suelen ser elevados, casi cercanos al 100 %.

Considerando este hecho, aunque el comportamiento que se sigue es en principio correcto (cuando los niveles de metilación disminuyen la expresión de gen aumenta) para establecer una relación consistente el valor de metilación de la región X a tiempo inicial (cuando el gen X está totalmente reprimido) debería ser mucho más alto ($\sim 0{,}9$), y para conseguirlo el valor de la muestra inicial debería ser este. El resultado obtenido tras el paso de modificación, reflejado en la tabla 4.7b, muestra una situación más acorde con la realidad.

En la modificación, además de considerar el patrón temporal también se tienen en cuenta los posibles cambios de FC producidos en cada condición y clase de genes.

Tabla 4.7: Ejemplo de modificación de muestra inicial.

$\boldsymbol{\mu_i}$: conteo en la muestra inicial.

(a) Sin modificación.

ID	**Ómica**	**Patrón**	$\boldsymbol{\mu_i}$	**Tiempos**		
				t_0	...	t_n
Gen X	RNA-seq		500	0	...	500
Reg. X	Methyl-seq		0,1	0,1	...	0

(b) Con modificación.

...	**Patrón**	$\boldsymbol{\mu_i}$	**Tiempos**		
			t_0	...	t_n
...		500	0	...	500
...		0,9	0,9	...	0

Según lo expuesto hasta ahora sobre reparto en clases de genes, existencia de uno o varios programas regulatorios para cada una de ellas, y combinando los ejemplos de las tablas 4.2b y 4.5, finalmente los patrones asignados a cada una de las partes del sistema serían los reflejados en la tabla 4.8, donde se ha omitido la condición número dos para facilitar la visualización.

Tabla 4.8: Ejemplo de integración de reguladores.

CL (clase de genes), PR (programa regulatorio), C1 (condición 1).

CL	**PR**	**RNA-seq**	**Methyl-seq**	**miRNA-seq**	**DNase-seq**	**ChIP-seq**
		C1	**C1**	**C1**	**C1**	**C1**
1	1					
	2					
2	3					
3	4					
	5					
4	6					

4.3.1. miRNA-seq, DNase-seq y ChIP-seq

Estos tres simuladores siguen el mismo procedimiento descrito para RNA-seq, basado en la modelización de los valores por una distribución binomial negativa parametrizada según media y dispersión, expresado este último en términos de media y desviación típica. Las tablas empleadas con los parámetros necesarios para hallar la desviación típica a través de una distribución Gamma fueron generadas de la misma forma explicada con anterioridad.

Las diferencias con respecto a la simulación de los datos de RNA-seq son las siguientes:

– **Datos iniciales** La corrección por longitud, si se realiza, tiene en cuenta el tamaño de las regiones en ChIP-seq y DNase-seq, mientras que está desactivada en miRNA-seq.

– **Diferencia de expresión entre condiciones** Los cambios de FC no alteran los valores directamente como lo hacían en RNA-seq, pero sí se tienen en cuenta en el primer paso de integración (modificación de muestra inicial) descrito en la sección anterior. Para ChIP-seq y DNase-seq (activadores) un cambio positivo en FC derivará en un incremento de sus valores, mientras que en el caso de miRNA-seq (represor) serán reducidos; si por el contrario el cambio en FC es negativo se invertirá lo anterior.

– **Traducción de identificadores** Dado que la inicialización del algoritmo configura el reparto de programas regulatorios, clases de genes, etc. basándose en los identificadores de los genes proporcionados para RNA-seq, el resto de simuladores realiza una búsqueda en la tabla de asociación proporcionada para conocer qué regiones (ChIP-seq y DNase-seq) o transcritos (miRNA-seq) están asociados a esos genes.

4.3.2. Methyl-seq

Para la adaptación del sistema WGBSSuite al algoritmo fue necesario realizar diversos cambios al código:

- El primer paso correspondiente a la generación de las localizaciones de los sitios CpG fue sustituido para incorporar en su lugar las correspondientes al conjunto de datos reales proporcionado, y que en sus identificadores cuentan con el cromosoma y la posición del sitio CpG dentro de éste.
- El algoritmo original únicamente simula sitios CpG sin considerar localización cromosómica. Aprovechando la información derivada de los datos reales, se modificó el script para realizar simulaciones independientes por cada uno de los cromosomas existentes, usando las posiciones de los sitios CpG de este. Finalmente todo se integra en una sola matriz de datos con identificadores adecuados, incluyendo el cromosoma.
- El simulador original devuelve varias columnas de datos, de estas, una corresponde al porcentaje de metilación, denominado valores β; sin embargo en análisis reales se realizan una serie de transformaciones que mitigan algunos problemas estadísticos de los mismos (rango de valores limitados y una distribución fuertemente bimodal) al coste de una interpretabilidad biológica reducida. Sea b_i el valor simulado β para cada sitio CpG i considerado y T un valor umbral definido en configuración; b_i se corrige de forma que quede comprendido en el rango $[T, 1-T]$ y posteriormente se realiza una transformación logarítmica $log2(\frac{b_i}{1-b_i})$ para todo b_i distinto de 0. Estos valores transformados se conocen

como valores M, y son devueltos junto a los valores β en una lista para que el usuario pueda seleccionar entre ellos.

- La generación de gráficos incrustada en el código también fue desactivada para no interferir con la simulación.
- El resto de modificaciones consistieron en la integración en el esquema general de simulación, añadiendo soporte para series temporales, modificación de la muestra inicial, ajuste por profundidad de secuenciación, así como búsqueda en la tabla de asociación para relacionar los sitios CpG con genes, de forma similar para la descrita en el resto de simuladores.
- La modificación de datos iniciales en el proceso de integración deriva en porcentajes de metilación casi máximos para los genes no expresados, y valores mínimos para los expresados, imitando los datos reales donde la mayoría de valores observados son altos o bajos, con poca cantidad que se sitúe en valores intermedios.

4.4. Ejemplos de funcionamiento

La ejecución del simulador devuelve una instancia de una clase tipo S4 "Simulation", conteniendo diversos apartados accesibles por el operador "@" de R. Muchos de estos apartados son listas de valores, a cuyos elementos se accede con el operador "$".

Como ejemplo, supóngase la ejecución del siguiente código:

```
depth = 25

opciones_simuladores <- list(
        "SimRNAseq" =  list("depth"=depth),
        "SimMethylseq" = list("depth"=depth),
        "SimMiRNAseq" = list("depth"=depth),
        "SimChIPseq" = list("depth"=depth),
        "SimDNaseseq" = list("depth"=depth)
)

stSim <- STATegRaSimulation(
        simulators = opciones_simuladores,
        nGeneClasses = 6,
        nRegPrograms = 9
)

informe <- reportData(stSim)
```

La variable *stSim* contendrá el resultado de la simulación, entre las que se incluyen los parámetros de configuración generados, referencias a los objetos de clase "simulador" inicializados, que a su vez disponen de las tablas de asociación regiones-genes, muestras iniciales, etc.

A continuación se incluyen unos ejemplos de acceso a la información referente a la configuración generada, con el código ejecutado y su salida:

```
# Acceso a lista de configuración generada:
ls(stSim@simSettings)
# [1] "DEG"    "NoDEG"  "NoExpr"

# Lista de opciones disponibles para bloque de genes diferencialmente expresados
ls(stSim@simSettings$DEG)
# [1] "RPtoGeneClass"     "fcMatrix"          "geneClassNumber"   "geneClassPatterns"
 ↪   "geneClassToRP"     "geneGroups"        "regPrograms"       "tProfiles"

# Consulta de programas regulatorios
head(stSim@simSettings$DEG$regPrograms)
#             SimMiRNAseq SimChIPseq SimDNaseseq SimMethylseq
# [1,]           0          1           1            0
# [2,]           0          1           1            0
# [3,]           1          1           1            0
# [4,]           0          1           0            1
# [5,]           0          0           1            1
# [6,]           0          0           1            1
```

Los datos simulados están presentes en *simData*, clasificados según la clase correspondiente a cada ómica:

```
# Consultas ómicas simuladas
ls(stSim@simData)
#[1] "SimChIPseq"   "SimDNaseseq"  "SimMethylseq" "SimMiRNAseq"  "SimRNAseq"

# Consulta de genes simulados en RNA-seq
head(stSim@simData$SimRNAseq)
#                     S1T0R1 S1T0R2 S1T0R3 S1T2R1 S1T2R2 S1T2R3  ...
#ENSMUSG00000095599      0      0      0      0      0      0    ...
#ENSMUSG00000037169      0      0      0      0      0      0    ...
#ENSMUSG00000085049      0      0      0      0      0      0    ...
#ENSMUSG00000091227      0      0      0      0      0      0    ...
#ENSMUSG00000045815      0      0      0      0      0      0    ...
#ENSMUSG00000091006      0      0      0      0      0      0    ...
# (Valores correspondientes a genes no expresados)
```

El valor *simData* devuelve los datos generados para ser utilizados directamente en los programas de análisis, sin embargo, la función *reportData* genera una serie de listas con los datos organizados según bloque, programa regulatorio... además de permitir la consulta de los conteos iniciales usados, entre otros. Siguiendo con el ejemplo anterior, siendo *informe* el valor devuelto por esta función:

```
# Índice del informe
ls(informe)
```

```
#[1] "DEG"    "Global" "NoDEG"  "NoExpr"

# Consultar información sobre RNA-seq
head(informe$Global$SimRNAseq)
#                     Group Gene class Reg. program Pattern Expr. sample 1 Expr. sample 2
# ENSMUSG00000095599 "DEG"    "1"        "1"          "1"     "-1"           "1"
# ENSMUSG00000037169 "DEG"    "1"        "1"          "1"     "-1"           "1"
# ENSMUSG00000085049 "DEG"    "1"        "1"          "1"     "-1"           "1"

# Opciones disponibles para bloque DEG en RNA-seq
ls(informe$DEG$SimRNAseq)
# [1] "ALL"  "BC1"  "BC2"  "RP1"  "RP10" "RP2"  "RP3"  "RP4"  "RP5"  "RP6"  "RP7"  "RP8"  "RP9"

# Consultar valor de genes para programa regulatorio 1 en simulador RNA-seq con bloque DEG
head(informe$DEG$SimRNAseq$RP1)
# (Salida manipulada eliminando columnas para mejorar legibilidad)
# Condición 1: no expresado
# Condición 2: expresado con inducción continua
#                     S1T0R1 S1T0R2 S1T0R3 S1T2R1 S1T2R2 S1T2R3 S1T6R1 S1T6R2  ...
#ENSMUSG00000095599      0      0      0      0      0      0     .0          0    ...
#ENSMUSG00000037169      0      0      0      0      0      0      0      0    ...
#ENSMUSG00000085049      0      0      0      0      0      0      0      0    ...
#                     S2T0R1  S2T0R2 S2T0R3 S2T2R1 S2T2R2 S2T2R3 S2T6R1 S2T6R2 ...
#ENSMUSG00000095599      0      0       0    345    678    787   1567   2378   ...
#ENSMUSG00000037169      0      0      29     14     54     42    456    657   ...
#ENSMUSG00000085049      0      2       0    108    234    167    623    713   ...
```

Para conseguir otra información, por ejemplo los genes asociados a unas determinadas regiones, pueden emplearse funciones de los simuladores:

```
# Ejemplo de regiones DNase-seq
regiones <- sample(rownames(informe$DEG$SimDNaseseq$ALL), 3)
# [1] "19_15837114_15837312" "13_85189968_85190120" "19_60608901_60609042"

# Simulador cargado de DNase-seq
simDNase <- stSim@simulators$SimDNaseseq

# Genes asociados a las regiones
IDtoGenes(simDNase, regiones)·
# [1] "ENSMUSG00000036545" "ENSMUSG00000086509" "ENSMUSG00000081266"
```

Existen otras funciones de utilidad en diferentes estados de implementación, como la generación de gráficas representando los comportamientos de los genes por bloques. La documentación de todas ellas estará disponible en el manual de usuario de STATegRa.

5 Conclusiones

La popularización de las nuevas tecnologías de secuenciación y la mejora de protocolos está dando lugar a la aparición de nuevas fuentes de información sobre los mecanismos celulares en cantidades cada vez mayores. Resulta imprescindible reorientar la forma de analizar estos datos ómicos y considerar al mismo tiempo todas las partes del sistema elaborando nuevos modelos matemáticos o adaptando los antiguos para maximizar los resultados.

Como parte de esta nueva tendencia surge la necesidad de utilizar herramientas compatibles pero, en el momento de presentar este texto, no existen opciones adecuadas disponibles para procedimientos como la validación *in silico* de estos modelos de integración de datos ómicos. Por ello, en este trabajo se ha desarrollado un algoritmo de simulación de datos ómicos dentro del marco del proyecto europeo STATegra que permitirá estimar la potencia estadística de nuevos modelos.

El resultado final ha sido una herramienta funcional programada en R para ser incluida dentro del paquete de R STATegRa, del repositorio público Bioconductor. Esta herramienta permite simular datos de diferentes ómicas obtenidos mediante técnicas de secuenciación masiva: RNA-seq, miRNA-seq, methyl-seq, ChIP-seq y DNase-seq, así como la regulación de la expresión génica (RNA-seq) llevada a cabo por distintos elementos: miARNs (miRNA-seq), metilación (methyl-seq), factores de transcripción (ChIP-seq) y accesibilidad de la cromatina (DNase-seq). De este modo, las opciones de integración que ofrece no están presentes en ninguno de los algoritmos de simulación de datos ómicos disponibles hasta el momento. La simulación de estos datos de forma realista ha sido posible gracias al estudio en profundidad de los datos experimentales procedentes del proyecto STATegra.

Además, el algoritmo de simulación presentado en este trabajo proporciona una gran flexibilidad en cuanto al diseño experimental, permitiendo generar datos de una o varias condiciones experimentales, así como series temporales.

Sin embargo, debido al alcance de este trabajo, se es consciente de las limitaciones de esta primera versión del algoritmo. Aunque abarca muchas posibilidades y es plenamente funcional, se han detectado potenciales mejoras a implementar en futuras versiones del mismo. La versión actual está preparada para dar soporte a más de dos condiciones experimentales pero es necesario ajustar mejor esta opción. En esta versión, es necesario proporcionar inicialmente una muestra de datos reales o bien utilizar los datos proporcionados por defecto (procedentes del proyecto STATegra). El simulador de RNA-seq está programado para poder simular aleatoriamente dichos datos iniciales, sin necesidad de proporcionarlos o tomar los que hay por defecto. Sería deseable poder disponer de esta posiblidad para todas las ómicas, lo que requeriría un estudio en profundidad de las características de cada tipo de datos. Y, en general, se podría mejorar y

ampliar la modelización de los programas regulatorios para abarcar mayor número de escenarios reales, así como ajustar mejor las distintas opciones de patrones temporales, cambios de expresión o señal, etc.

A pesar de estas limitaciones, se ha realizado un gran esfuerzo por proporcionar un método útil para la simulación de datos multiómicos que servirá de base para futuras versiones, y que proporciona una herramienta indispensable para evaluar el funcionamiento de los modelos estadísticos que se desarrollen en el contexto de la integración de datos ómicos.

Bibliografía

AKALIN, A., KORMAKSSON, M., LI, S., GARRETT-BAKELMAN, F. E., FIGUEROA, M. E., MELNICK, A. y MASON, C. E. (2012). «methylKit: a comprehensive R package for the analysis of genome-wide DNA methylation profiles». En: *Genome Biology* 13.10, R87. ISSN: 1465-6906. DOI: 10.1186/gb-2012-13-10-r87. URL: http://genomebiology.com/2012/13/10/R87.

ALEKSIC, J., CARL, S. H. y FRYE, M. (2014). «Beyond library size: a field guide to NGS normalization». En: *bioRxiv*. DOI: 10.1101/006403. URL: http://biorxiv.org/content/early/2014/06/19/006403.abstract.

ANDERS, S. y HUBER, W. (2012b). «Differential expression of RNA-Seq data at the gene level–the DESeq package». En: *EMBL, Heidelberg, Germany*. URL: http://gga01.med.wayne.edu/online%5C_help/help%5C_regionminer/DESeq.pdf.

BARTEL, D. P. (2004). «MicroRNAs: Genomics, Biogenesis, Mechanism, and Function». En: *Cell* 116.2, págs. 281-297. ISSN: 00928674. DOI: 10.1016/S0092-8674(04)00045-5.

BOCK, C. (2012). «Analysing and interpreting DNA methylation data». En: *Nature Reviews Genetics* 13.10, págs. 705-719. ISSN: 1471-0056. DOI: 10.1038/nrg3273. URL: http://dx.doi.org/10.1038/nrg3273.

BOYLE, A. P., DAVIS, S., SHULHA, H. P., MELTZER, P., MARGULIES, E. H., WENG, Z., FUREY, T. S. y CRAWFORD, G. E. (2008). «High-Resolution Mapping and Characterization of Open Chromatin across the Genome». En: *Cell* 132.2, págs. 311-322. ISSN: 00928674. DOI: 10.1016/j.cell.2007.12.014.

FURUSAWA, C. y KANEKO, K. (2003). «Zipf's law in gene expression.» En: *Physical review letters* 90.8, pág. 088102. ISSN: 0031-9007. DOI: 10.1103/PhysRevLett.90.088102. arXiv: 0209103 [physics].

GENTLEMAN, R. C., GENTLEMAN, R. C., CAREY, V. J., CAREY, V. J., BATES, D. M., BATES, D. M., BOLSTAD, B., BOLSTAD, B., DETTLING, M., DETTLING, M., DUDOIT, S., DUDOIT, S., ELLIS, B., ELLIS, B., GAUTIER, L., GAUTIER, L., GE, Y., GE, Y., GENTRY, J., GENTRY, J., HORNIK, K., HORNIK, K., HOTHORN, T., HOTHORN, T., HUBER, W., HUBER, W., IACUS, S., IACUS, S., IRIZARRY, R., IRIZARRY, R., LEISCH, F., LEISCH, F., LI, C., LI, C., MAECHLER, M., MAECHLER, M., ROSSINI, A. J., ROSSINI, A. J., SAWITZKI, G., SAWITZKI, G., SMITH, C., SMITH, C., SMYTH, G., SMYTH, G., TIERNEY, L., TIERNEY, L., YANG, J. Y. H., YANG, J. Y. H., ZHANG, J. y ZHANG, J. (2004). «Bioconductor: open software development for computational biology and bioinformatics.» En: *Genome biology* 5.10, R80. ISSN: 1465-6914. DOI: 10.1186/gb-2004-5-10-r80. URL: http://www.ncbi.nlm.nih.gov/pubmed/15461798.

GOMEZ-CABRERO, D., ABUGESSAISA, I., MAIER, D., TESCHENDORFF, A., MERKENSCHLAGER, M., GISEL, A., BALLESTAR, E., BONGCAM-RUDLOFF, E., CONESA, A. y TEGNÉR, J. (2014). «Data integration in the era of omics: current and future challenges». En: *BMC Systems Biology* 8.Suppl 2, pág. I1. ISSN: 1752-0509. DOI: 10.1186/1752-0509-8-S2-I1. URL: http://www.biomedcentral.com/1752-0509/8/S2/I1.

HAFNER, M., LANDGRAF, P., LUDWIG, J., RICE, A., OJO, T., LIN, C., HOLOCH, D., LIM, C. y TUSCHL, T. (2008). «Identification of microRNAs and other small regulatory RNAs using cDNA library sequencing». En: *Methods* 44.1, págs. 3-12. ISSN: 10462023. DOI: 10.1016/j.ymeth.2007.09.009.

JONES, P. a. (2012). «Functions of DNA methylation: islands, start sites, gene bodies and beyond». En: *Nature Reviews Genetics*.

KOBOLDT, D. C., STEINBERG, K. M., LARSON, D. E., WILSON, R. K. y MARDIS, E. R. (2013). «XThe next-generation sequencing revolution and its impact on genomics». En: *Cell*

155.1, págs. 27-38. ISSN: 00928674. DOI: 10.1016/j.cell.2013.09.006. URL: http://dx.doi.org/10.1016/j.cell.2013.09.006.

LAIRD, P. W. (2010). «Principles and challenges of genomewide DNA methylation analysis.» En: *Nature reviews. Genetics* 11.3, págs. 191-203. ISSN: 1471-0056. DOI: 10.1038/nrg2732. URL: http://dx.doi.org/10.1038/nrg2732.

LIU, L., LI, Y., LI, S., HU, N., HE, Y., PONG, R., LIN, D., LU, L. y LAW, M. (2012). «Comparison of next-generation sequencing systems». En: *Journal of Biomedicine and Biotechnology* 2012. ISSN: 11107243. DOI: 10.1155/2012/251364.

METZKER, M. L. (2010). «Sequencing technologies - the next generation.» En: *Nature reviews. Genetics* 11.1, págs. 31-46. ISSN: 1471-0056. DOI: 10.1038/nrg2626. arXiv: 209. URL: http://dx.doi.org/10.1038/nrg2626.

NATIONAL CANCER INSTITUTE y NATIONAL HUMAN GENOME RESEARCH INSTITUTE (2015). *The cancer genome atlas.* URL: http://cancergenome.nih.gov/ (visitado 01-01-2015).

NUEDA, M. J., TARAZONA, S. y CONESA, A. (2014). «Next maSigPro: updating maSigPro bioconductor package for RNA-seq time series.» En: *Bioinformatics (Oxford, England)*, págs. 1-5. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btu333. URL: http://www.ncbi.nlm.nih.gov/pubmed/24894503.

OUYANG, Z., ZHOU, Q. y WONG, W. H. (2009). «ChIP-Seq of transcription factors predicts absolute and differential gene expression in embryonic stem cells.» En: *Proceedings of the National Academy of Sciences of the United States of America* 106.51, págs. 21521-21526. ISSN: 1091-6490. DOI: 10.1073/pnas.0904863106.

PARK, P. J. (2009). «ChIP-seq: advantages and challenges of a maturing technology.» En: *Nature reviews. Genetics* 10.10, págs. 669-680. ISSN: 1471-0056. DOI: 10.1038/nrg2641. URL: http://dx.doi.org/10.1038/nrg2641.

R CORE TEAM (2014a). «R : A Language and Environment for Statistical Computing». En: 1. ISSN: 16000706. DOI: 10.1007/978-3-540-74686-7. URL: http://www.r-project.org/.

— (2014b). «R Language Definition V. 3.1.1». En: 3.1.1, pág. 55. URL: http://mirror.fcaglp.unlp.edu.ar/CRAN/doc/manuals/r-patched/R-lang.pdf$%5Cbackslash$nhttp://cran.r-project.org/doc/manuals/r-release/R-lang.pdf.

RAJKUMAR, A. P., QVIST, P., LAZARUS, R., LESCAI, F., JU, J., NYEGAARD, M., MORS, O., BØRGLUM, A. D., LI, Q. y CHRISTENSEN, J. H. (2015). «Experimental validation of methods for differential gene expression analysis and sample pooling in RNA-seq». En: *BMC Genomics* 16.1, pág. 548. ISSN: 1471-2164. DOI: 10.1186/s12864-015-1767-y. URL: http://www.biomedcentral.com/1471-2164/16/548.

ROBLES, J. a., QURESHI, S. E., STEPHEN, S. J., WILSON, S. R., BURDEN, C. J. y TAYLOR, J. M. (2012). «Efficient experimental design and analysis strategies for the detection of differential expression using RNA-Sequencing». En: *BMC Genomics* 13.1, pág. 484. ISSN: 1471-2164. DOI: 10.1186/1471-2164-13-484.

SANGER, F., NICKLEN, S. y COULSON, a. R. (1977). «DNA sequencing with chain-terminating inhibitors.» En: *Proceedings of the National Academy of Sciences of the United States of America* 74.12, págs. 5463-5467. ISSN: 0027-8424. DOI: 10.1073/pnas.74.12.5463.

SHENDURE, J. y JI, H. (2008). «Next-generation DNA sequencing.» En: *Nature biotechnology* 26.10, págs. 1135-1145. ISSN: 1087-0156. DOI: 10.1038/nbt1486. arXiv: 1111.6189v1.

SMYTH, G. (2005). «limma: Linear Models for Microarray Data». En: *Bioinformatics and Computational Biology Solutions Using R and Bioconductor*, págs. 397-420. ISSN: 1544-6115. DOI: citeulike-article-id:5722720. URL: http://dx.doi.org/10.1007/0-387-29362-0%5C_23.

SONESON, C. (2014b). «compcodeR - an R package for benchmarking differential expression methods for RNA-seq data.» En: *Bioinformatics (Oxford, England)*, págs. 1-2. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btu324. URL: http://www.ncbi.nlm.nih.gov/pubmed/24813215.

SONESON, C. y DELORENZI, M. (2013a). «A comparison of methods for differential expression analysis of RNA-seq data». En: *BMC Bioinformatics* 14.1, pág. 91. ISSN: 1471-2105. DOI: 10.1186/1471-2105-14-91. URL: http://www.biomedcentral.com/1471-2105/14/91/abstract$%5Cbackslash$nhttp://www.biomedcentral.com/1471-2105/14/91$%5Cbackslash$nhttp://www.biomedcentral.com/content/pdf/1471-2105-14-91.pdf.

TARAZONA, S., FURI, P., FERRER, A. y CONESA, A. (2015a). «NOISeq : Differential Expression in RNA-seq». En: págs. 1-26.

TARAZONA, S., FURIÓ-TARÍ, P., TURRÀ, D., DI PIETRO, A., NUEDA, M. J., FERRER, A. y CONESA, A. (2015b). «Data quality aware analysis of differential expression in RNA-seq with NOISeq R/Bioc package». En: *Nucleic Acids Research*. DOI: 10.1093/nar/gkv711. eprint: http://nar.oxfordjournals.org/content/early/2015/07/15/nar.gkv711.full.pdf+html. URL: http://nar.oxfordjournals.org/content/early/2015/07/15/nar.gkv711.abstract.

TARAZONA, S., PRADO-LÓPEZ, S., DOPAZO, J., FERRER, A. y CONESA, A. (2012). «Variable selection for multifactorial genomic data». En: *Chemometrics and Intelligent Laboratory Systems* 110.1, págs. 113-122.

THE ENCODE PROJECT CONSORTIUM (2004). «The ENCODE (ENCyclopedia Of DNA Elements) Project.» En: *Science (New York, N.Y.)* 306.5696, págs. 636-640. ISSN: 0036-8075. DOI: 10.1126/science.1105136.

THE STATEGRA CONSORTIUM (2007). «STATegra White paper». En: September, págs. 1-9.

WANG, Z., GERSTEIN, M. y SNYDER, M. (2009). «RNA-Seq: a revolutionary tool for transcriptomics.» En: *Nature reviews. Genetics* 10.1, págs. 57-63. ISSN: 1471-0056. DOI: 10.1038/nrg2484. arXiv: NIHMS150003.

Apéndices

A

Algoritmo

A.1. Jerarquía de archivos

- **AllClass.R**
- **AllGeneric.R**
- **Simulation.R**
- **Simulator.R**
- **SimulatorRegion.R**
- **simulators/ChIP-seq.R**
- **simulators/DNase-seq.R**
- **simulators/Methyl-seq.R**
- **simulators/RNA-seq.R**
- **simulators/miRNA-seq.R**

A.2. Código fuente

A.2.1. AllClass.R

```
#
#' Simulation class object
#'
#' This class manages the global simulation process, like associating genes
#' with gene classes, regulatory programs and other settings. Finally it will
#' initialize the simulators with their options that will use the previously
#' generated settings to simulate the data.
#'
#' @slot simulators list. Vector containing either S4 initialized classes of simulators
```

```
#'  or a list with the class name as keys, and its options as value, see example.
#' @slot totalGenes numeric. Total number of genes including not expressed. Overwrited
#'   if a genome reference is provided. Currently not used as we force to provide real data.
#' @slot diffGenes numeric. Total number of differential genes (if value > 1) or % or
#'  total genes (if value < 1).
#' @slot numberReps numeric. Number of replicates of the experiment.
#' @slot numberSamples numeric. Number ofsamples considered on the experiment. Currently forced
↪   to 2.
#' @slot nGeneClasses numeric. Number of gene classes to take into account. Must be an even
↪   number.
#' @slot randomSeed numeric. Random seed for random number generator state.
#' @slot times vector. Numeric vector containing the measured times. If numberSamples < 2,
#'  the number of times must be at least 2.
#' @slot geneNames character. List containing the IDs of the genes. Overwrited by the
#'  genome reference if provided. Currently not used as we force to provide real data.
#' @slot defaultPatterns list. New default patterns to be used if needed. If passed,
#'  every simulator loaded must be provided with new patterns.
#' @slot simSettings list. Overrides initializing the configuration of the simulation
#'  by passing a previously generated list. This could be used to tweak by hand the
#'  assigned profiles, genes, regulatory programs, etc.
#' @slot simData list. Simulated data, useful to load previously simulated data
#'  and generate some plots.
#' @slot genomeSize numeric. Number of pair bases of the genome. The reference genome
#'  generated is extremely basic. Currently not used as we force to provide real data.
#' @slot meanGeneSize numeric. Average gene size used to generate the reference genome.
#'  Currently not used as we force to provide real data.
#' @slot GRanges GRanges. GRanges object to be used in gene-region associations. If a reference
#'  file is provided it will be extracted from it.
#' @slot GTF ANY. GFF file format containing genes information, both exon ID and
#'  chromosome regions.
#' @slot organism ANY. Parameter used o load the GFF file.
#' @slot exprGenes numeric. Total number of expressed genes (if value > 1) or %
#'  of the total genes (if value < 1).
#' @slot nRegPrograms numeric. Number of regulatory programs to consider.
#'
#' @export
#'
#'
setClass(
    "Simulation",
    slots = c(
        simulators = "list", totalGenes = "numeric",
        diffGenes = "numeric", numberReps = "numeric",
        numberSamples = "numeric", env = "environment",
        .nCols = "numeric", nGeneClasses = "numeric",
        randomSeed = "numeric", times = "vector",
        geneNames = "character", defaultPatterns = "list",
        simSettings = "list", simData = "list",
        genomeSize = "numeric", meanGeneSize = "numeric",
        GRanges = "ANY", GTF = "ANY", organism = "ANY",
        exprGenes = "numeric", nRegPrograms = "numeric",
```

```
        defaultData = "list"
    ),
    prototype = list(
        totalGenes = 20000, diffGenes = .15,
        numberReps = 3, numberSamples = 2,
        sdNoise = 0.3, nRegPrograms = 10,
        nGeneClasses = 6, randomSeed = 12345,
        times = c(0, 2, 6, 12), meanGeneSize = 10000,
        genomeSize = 5000000, GTF = NULL,
        organism = NA, exprGenes = 0.5
    )
)

#' Simulator object
#'
#' Virtual class containing common methods and slots for child classes.
#'
#' @slot name character. Name of the simulator to be used in debug messages.
#' @slot data vector. Initial sample to be used.
#' @slot simulation Simulation. Previously initialized Simulation object.
#' @slot .sdata list. Internal container for storing data.
#' @slot patternCoefficients list. Pattern coefficients. They must follow the same
#'  structure as the ones defined in the Simulation class.
#' @slot regulator logical. Flag to distinguish a regulator from RNA-seq (at this moment).
#' @slot enhancer logical. Flag to define the effect of the regulator on gene
#'  expression.
#' @slot idToGene list. Association table between genes and other IDs (miRNAs, regions...)
#' @slot minValue numeric. The simulated data will be corrected to prevent having
#'  values below this number.
#' @slot maxValue numeric. The simulated data will be corrected to prevent having
#'  values above this number.
#' @slot gammaTable ANY. Internal reference to the gamma distribution table parameters.
#' @slot depth numeric. Sequencing depth to simulate.
#' @slot noise numeric. Noise value for the NB mean.
#' @slot depthRound numeric. Number of decimal places to round when adjusting depth.
#' @slot depthAdjust logical. Whether to adjust by sequencing depth or not.
#'
#' @export
#'
setClass(
    "Simulator",
    slots = c(
        name = "character", data = "vector",
        # simulation = "Simulation", .sdata = "list",
        simulation = "environment", .sdata = "list",
        patternCoefficients = "list", regulator = "logical",
        enhancer = "logical", idToGene = "matrix",
        minValue = "numeric", maxValue = "numeric",
        depth = "numeric", gammaTable = "ANY",
        noise = "numeric", variability = "character",
        depthRound = "numeric", lengthBias = "logical",
```

```
        lengthBiasFunction = "ANY", depthAdjust = "logical",
        coeffIncrease = "numeric"
    ),
    prototype = list(
        regulator = TRUE, virtual = TRUE,
        enhancer = FALSE, minValue = 0,
        depth = 30, noise = 0.1,
        variability = "low", gammaTable = NULL,
        depthRound = 0, lengthBias = FALSE,
        lengthBiasFunction = "lineal", depthAdjust = TRUE,
        coeffIncrease = 10
    )
)

#' SimulatorRegion object
#'
#' Virtual class containing general methods for simulators based on
#' regions of the chromosomes, like DNase-seq, ChIP-seq or Methyl-seq
#'
#' @slot locs ANY. Vector containing the list of locations of the sites.
#' @slot chGRanges ANY. List of GRanges object, one for every chromosome.
#' @slot nLocs numeric. Number of sites to simulate.
#' @slot locsName character. Type of the site to simulate, only for debug.
#'
#' @export
#'
setClass(
    "SimulatorRegion",
    slots = c(locsName = "character", locs = "ANY",
              chGRanges = "ANY", nLocs = "numeric"),
    prototype = list(
        virtual = TRUE
    ),
    contains = "Simulator"
)

#' SimRNAseq
#'
#' Class to simulate RNA-seq data
#'
#' @slot beta numeric. "shape2" parameter of the beta distribution used to model
#'  FC diffs.
#' @slot nGenes numeric. Internal value containing the number of genes simulated.
#' @slot geneLength matrix. A matrix with rownames as gene ID and a column with their size.
#'
#' @export
#'
setClass(
    "SimRNAseq",
    slots = c(beta = "numeric", nGenes = "numeric",
              geneLength = "matrix"),
```

```
    prototype = list(
        name = "RNA-seq", idToGene = matrix(NA),
        beta = 6, nGenes = 0, regulator = FALSE
    ),
    contains = "Simulator"
)

#' SimMiRNAseq
#'
#' Class to simulate miRNA-seq
#'
#' @slot nmiRNA numeric. Number of RNAs to simulate
#'
#' @export
#'
setClass(
    "SimMiRNAseq",
    slots = c(nmiRNA = "numeric"),
    prototype = list(
        name = "miRNA-seq", enhancer = FALSE,
        regulator = TRUE, nmiRNA = 0,
        idToGene = matrix()
        ),
    contains = "Simulator"
)

#' SimChIPseq
#'
#' Class to simulate ChIP-seq data
#'
#' @export
#'
setClass("SimChIPseq",
         #slots=c(bsSize = "numeric"),
         prototype = list(
             name = "ChIP-seq", locsName = "binding sites",
             enhancer = TRUE, idToGene = matrix()),
         contains = "SimulatorRegion")

#' SimDNaseseq
#'
#' Class to simulate DNase-seq data
#'
#' @export
#'
setClass("SimDNaseseq",
         #slots=c(DHSlocs = "ANY"),
         prototype = list(
             name = "DNase-seq", locsName = "DHS",
             enhancer = TRUE, idToGene = matrix()),
         contains = "SimulatorRegion")
```

```r
#' SimMethylseq
#'
#' Class to simulate Methyl-seq data.
#'
#' @slot nCpG numeric. Number of CpG sites to simulate.
#' @slot pSuccessMethReg numeric. Probability of success in methylated region.
#' @slot pSuccessDemethReg numeric. Probability of success in non methylated region
#' @slot errorMethReg numeric. Error rate in methylated region
#' @slot errorDemethReg numeric. Error rate in methylated region
#' @slot nReadsMethReg numeric. Mean number of reads in methylated region.
#' @slot nReadsDemethReg numeric. Mean number of reads in non methylated regions.
#' @slot phaseDiff numeric. Phase difference in the differentially methylated regions between
↪  two samples
#' @slot balanceHypoHyper numeric. Balance of hypo/hyper methylation
#' @slot ratesHMMMatrix numeric. Matrix of values that describes the exponential
#'  decay functions that define the distances between CpG values.
#' @slot distType character. Distribution used to generate replicates:
#' @slot transitionSize numeric.
#' @slot PhiMeth matrix. Transition matrix for CpG locations.
#' @slot PhiDemeth matrix. <Not used>
#' @slot typesLocation numeric. <Not used>
#' @slot returnValue character. Selected column:
#' @slot betaThreshold numeric. Beta threshold value used to calculate M values.
#'
#' @export
#'
setClass(
    "SimMethylseq",
    slots = c(
        nCpG = "numeric", pSuccessMethReg = "numeric",
        pSuccessDemethReg = "numeric", errorMethReg = "numeric",
        errorDemethReg = "numeric", nReadsMethReg = "numeric",
        nReadsDemethReg = "numeric", phaseDiff = "numeric",
        balanceHypoHyper = "numeric", nRepeats = "numeric",
        ratesHMMMatrix = "numeric", distType = "character",
        transitionSize = "numeric", PhiMeth = "matrix",
        PhiDemeth = "matrix", typesLocation = "numeric",
        returnValue = "character", betaThreshold = "numeric"
    ),
    prototype = list(
        name = "methyl-seq", transitionSize = 0, nCpG = 5000,
        PhiMeth = matrix(c(0.65, 0.35,0.2, 0.8), byrow =
                             TRUE, nrow = 2),
        PhiDemeth = matrix(c(0.9,0.1,0.1,0.9), byrow =
                               TRUE, nrow = 2),
        typesLocation = 2, ratesHMMMatrix = c(0.019,0.002),
        pSuccessMethReg = 0.9203, pSuccessDemethReg = 0.076,
        errorMethReg = 0.1, errorDemethReg = 0.1,
        nReadsMethReg = 29, nReadsDemethReg = 29,
        phaseDiff = 0.1, balanceHypoHyper = 0.5,
```

```
        distType = "binomial", returnValue = "proportionReads",
        betaThreshold = 0.01, locsName = "CpG",
        enhancer = FALSE, maxValue = 1, nRepeats = 1,
        idToGene = matrix(), depthAdjust = FALSE,
        coeffIncrease = 10
    ),
    contains = "SimulatorRegion"
)
```

A.2.2. AllGeneric.R

```
################################################################################
# General
################################################################################
#' Simulate [internal use]
#'
#' This generic can dispatch methods from two types of classes:
#' - Simulation: in which case triggers the simulation on every simulator
#'  loaded.
#' - Simulator: performs the simulation based on already initialized simulation
#'  settings.
#'
#' @param object Object of class Simulator.
#' @param ... Extra parameters for extensibility.
#' @param simSettings List containing the simulation settings.
#' @param force If true, the simulation will be performed again even if there's
#'  previously saved data.
#'
#' @return A Simulator object containing the simulation data inside @.sdata$svalues
#'  if called on a Simulator class, or a Simulation object with all @simulators
#'  containing simulated data.
#'
#' @examples
setGeneric("simulate", function(object, ...) standardGeneric("simulate"))

#' reportData
#'
#' This generic can dispatch methods from two types of classes:
#' - Simulation: list containing classifieds matrix values for every simulator,
#' regulatory program, gene class, etc.
#'
#' - Simulator: initial sample and simulated values
#'
#' @param object Instance of Simulation or Simulation class.
#'
#' @return For Simulator class a list containing the following entries:
#'    - simValues: simulated values
#'    - bData: initial sample values.
#' For Simulation class:
#'
```

```
#' - First level: blocks of genes to retrieve info, DEG, Global, NoDEG, NoExpr.
#' - Second level: inside every BLOCK, we have.
#'      - <SimulatorName>: for every simulator initialized.
#'          - ALL: all simulated values for genes-regions inside of BLOCK.
#'          - BC_i: initial sample values for sample "i" for genes/regions inside of BLOCK.
#'          - RP_n: simulated values for genes associated to regulatory program "n" and inside
↪  of BLOCK.
#'      - tratRP: matrix containing info of patterns associated to regulatory programs, a column
↪  for
#'          every sample (-1: not expressed, 1: expressed, 0: flat) and then a pattern column.
#'      - FCMatrix: only on DEG block, show the changes of FC between samples, associated with a
↪  reg. program.
#'
#'  In the case on "Global" block, on second level we have an entry for every simulator that
↪  contains
#'  all the genes with the group (DEG, NoDEG, NoExpr), gene class, reg. program, pattern, and
↪  state of expression
#'  (expressed, not expressed, flat) on all samples.
#'
#'
#' @examples
setGeneric("reportData", function(object) standardGeneric("reportData"))

# Simulation class

#' sampleGeneNames
#'
#' Retrieves a random sample of genes from the available IDs.
#'
#' @param object Initialized instance of Simulation class.
#' @param size Number of names to retrieve.
#' @param type Limit the pool to a kind of gene: DEG, NoDEG, NoExpr or c('DEG',...)
#'
#' @return A character vector containing 'size' gene names.
#'
#' @examples
setGeneric("sampleGeneNames", function(object, size, type) standardGeneric("sampleGeneNames"))

#  ------------------------------------------------------------------------


##############################################################################
# Simulator
##############################################################################
#' initializeData
#'
#' Generates the initial sample of data.
#'
#' @param object Instance of Simulator class.
#'
#' @return A list with "values" key containing the data, and "idToGene" as
```

```
#'  association table (eg: region-gene).
#' @export
#'
#' @examples
setGeneric("initializeData", function(object) standardGeneric("initializeData"))

#' configureSimulator
#'
#' Method called on class constructor, after settings the slots and defaultData
#' but before everything else.
#'
#' @param object Instance of Simulator class.
#' @param ...
#'
#' @return An instance of Simulator class with modified options.
#'
#' @examples
setGeneric("configureSimulator", function(object, ...) standardGeneric("configureSimulator"))


#' patternCoefficients
#'
#' Generic for dispatching method that will transform the parameters of the expression
#' defining a
#'
#' @param object Instance of Simulator class
#' @param coefs New set of pattern coefficients to overwrite the default ones.
#'
#' @return A numeric vector of the same length as the number of times considered
#'  by the experimental design.
#'
#' @examples
setGeneric("patternCoefficients", function(object, coefs=NULL)
 ↪  standardGeneric("patternCoefficients"))

#' groupSimulation
#'
#' Generic to dispatch method that transform the initial sample on the final
#' values with the correct number of replicates, added noise, etc. This is called
#' for every factor.
#'
#' @param object Instance of simulator class.
#' @param geneNames IDs (genes, regions...) to change.
#' @param timeCoefficients Numbers to multiply the base value (initial count) to simulate
#'  the times.
#' @param ... Extra parameters, if needed in the future.
#'
#' @return A matrix of data with N rows, corresponding to the IDs present on the data
#'  and C cols, number of times * number of replicates.
#'
#' @examples
```

```
setGeneric("groupSimulation", function(object, geneNames, timeCoefficients, ...)
 ↪  standardGeneric("groupSimulation"))

# setGeneric("interactionMatrix", function(object, values) standardGeneric("interactionMatrix"))


#' postSimulation
#'
#' Generic to dispatch method for making final modifications to the data, like round
#' values, put negative numbers to zero, etc.
#'
#' @param object Instace of class simulator
#' @param simData Simulated data
#'
#' @return Modified simulated data
#'
#' @examples
setGeneric("postSimulation", function(object, simData) standardGeneric("postSimulation"))

#' patternValues
#'
#' Generic to dispatch a method that returns functions to transform the initial
#' samples of the regulators based on the gene expression
#'
#' @param object Instance of a simulator class
#' @param ...
#'
#' @return A list containing 3 elements:
#'  - methods: a list with time patterns as ID (contind, contrep, etc) and a function to
#'  modify the data.
#'  - noise: a list with time patterns as ID (contind, contrep, etc) and a function to
#'  apply noise to the modified the data.
#'  - changeFC: list with "up" and "down" as names and a function to
#'  modify the data considering the FC changes.
#'
#' @examples
setGeneric("patternValues", function(object, ...) standardGeneric("patternValues"))

#' updateData
#'
#' Generic to dispatch method for updating the initial sample of regulators based
#' on the functions returned by calling patternValues
#'
#' @param object Instance of a simulator class
#' @param updatePattern Type of pattern (contind, contrep, ...)
#' @param geneNames IDs to modify
#' @param sampleNumber Number of sample
#' @param changeFC If there's is a change of FC (1, -1.5, 1.5) on the genes
#' @param regWeight <Not used>
#'
#' @return Instance of a simulator class with the initial data modified.
```

```
#'
#' @examples
setGeneric("updateData", function(object, updatePattern, geneNames, sampleNumber, changeFC,
↪  regWeight) standardGeneric("updateData"))

#' IDfromGenes
#'
#' Method for transforming genes to IDs (regions, miRNA...)
#'
#' @param object Instance of a Simulator class
#' @param geneNames Names of genes to look up in the association table.
#'
#' @return IDs corresponding to the genes.
#'
#' @examples
setGeneric("IDfromGenes", function(object, geneNames) standardGeneric("IDfromGenes"))


#' IDtoGenes
#'
#' Method for transforming IDs (regions, miRNAs, ...) to genes
#'
#' @param object Instance of a simulator class.
#' @param idNames IDs to look up in the association table.
#'
#' @return Genes corresponding to the IDs.
#'
#' @examples
setGeneric("IDtoGenes", function(object, idNames) standardGeneric("IDtoGenes"))

#' getLengths
#'
#' Get the length of the genes or regions.
#'
#' @param object
#'
#' @return A list containing the region or gene lengths.
#'
#' @examples
setGeneric("getLengths", function(object) standardGeneric("getLengths"))

################################################################################
# SimulatorRegion
################################################################################
#' locGRanges
#'
#' Creates a very basic GRanges object based on locations
#'
#' @param object Instance of a SimulatorRegion class
#' @param locs Positions of locations
#'
```

```
#' @return GRanges object with 1 as seqname and locs as starting positions.
#'
#' @examples
setGeneric("locGRanges", function(object, locs) standardGeneric("locGRanges"))

#' regionNames
#'
#' Creates a valid string of a region
#'
#' @param object Instace of a SimulatorRegion class
#' @param chrNumber IDs of the chromosomes
#' @param start Starts positions of the regions
#' @param end End positions of the regions
#'
#' @return A character vector with the format "<chr>_<start>_<end>"
#'
#' @examples
setGeneric("regionNames", function(object, chrNumber, start, end=NULL)
 ↪  standardGeneric("regionNames"))

#' nearestGenes
#'
#' By comparing two GRanges objects, returns the nearest genes of the regions.
#'
#' @param object Instance of a SimulatorRegion class.
#' @param refGR Reference genome GRanges class.
#' @param simGR GRanges object corresponding to the regions.
#'
#' @return EXONNAME column from the refGR corresponding to the nearest genes of the simGR
 ↪  regions.
#'
#' @examples
setGeneric("nearestGenes", function(object, refGR, simGR) standardGeneric("nearestGenes"))


#' sMessage
#'
#' A method for displaying warning messages with support for matrix and other things.
#'
#' @param ... Elements to display on console
#'
#' @export
#'
smessage <- function(...) {
    sParams <- list(...)
    sOutput <- c()

    for (sOpt in sParams) {
        if (is.null(dim(sOpt)) && ! is.list(sOpt)) {
            sOutput <- c(sOutput, sOpt)
        } else {
```

```
            sOutput <- c(sOutput, paste(capture.output(print(sOpt)), collapse="\n", "\n"))
        }
    }

    message(paste(sOutput, collapse=" "))
    loginfo(paste(sOutput, collapse=" "))
}

#' sloginfo
#'
#' A method for writing to a a log file (if it's configured before calling simulation)
#'
#' @param ... Elements to write
#'
#' @export
#'
sloginfo <- function(...) {
    sParams <- list(...)
    sOutput <- c()

    for (sOpt in sParams) {
        if (class(sOpt) == 'character') {
            sOutput <- c(sOutput, sOpt)
        } else {
            sOutput <- c(sOutput, paste(capture.output(print(sOpt)), collapse="\n"))
        }
    }

    loginfo(paste(sOutput, collapse=" "))
}
```

A.2.3. Simulation.R

```
.Simulation.defaultDataFilename <- "lib/data/sampleData.RData"

.Simulation.initialize <- function(.Object, ...) {
    # Set slots
    .Object <- callNextMethod()

    # RNA-seq simulator must always be present
    if (! exists('SimRNAseq', where=.Object@simulators)) {
        .Object@simulators[['SimRNAseq']] <- list()
    }

    # If we provide real data it needs to be done for every loaded
    # simulator and also with the association list.
    # (Note: An empty matrix has a length of 1)
    dataProvided <- sapply(.Object@simulators, function(s) length(s$data) > 1)

    # Check the provided association list. RNAseq is the only one with "NA" by default because
```

```
    # it doesn't need one.
    assocProvided <- sapply(.Object@simulators, function(s) length(s$idToGene) > 1 ||
    ↪  suppressWarnings(is.na(s$idToGene)))

    # If at least one simulator has provided data, all of them must have
    # both data and association list.
    if (! any(dataProvided)) {
        # No data: load default values
        #   file: sampleData.RData
        #   value: sampleData
        load(.Simulation.defaultDataFilename)

        .Object@defaultData <- sampleData

        # Assign parameters based on RNA-seq data
        exprData <- sampleData$SimRNAseq$data

        .Object@geneNames <- rownames(exprData)
        .Object@totalGenes <- nrow(exprData)

        # GRanges not needed in this version
        .Object@GRanges <- NULL

    } else if (! all(dataProvided & assocProvided)) {
        stop("If you provide an initial sample for one simulator, you need to do
             it for all the other ones, as well as include the association list (except with
↪   RNA-seq).")
    }

    # Check gene number
    .Object@exprGenes <- min(.Object@totalGenes, .Object@exprGenes)

    if (.Object@exprGenes < 1) {
        .Object@exprGenes <- round(.Object@exprGenes * .Object@totalGenes)
    }

    .Object@.nCols <- .Object@numberSamples*.Object@numberReps*length(.Object@times)

    smessage("\nInitializing Simulation object:")
    smessage("- Total genes -> ", .Object@totalGenes)
    smessage("- Expressed genes -> ", .Object@exprGenes)
    smessage("- Dif. expressed genes -> ", .Object@diffGenes*.Object@exprGenes)
    smessage("- Replicates -> ", .Object@numberReps)
    smessage("- Factor levels (samples) -> ", .Object@numberSamples)
    smessage("- Time vector length -> ", length(.Object@times))
    smessage("- Total column number -> ", .Object@.nCols, "\n")

    # Set random number generator seed
    set.seed(.Object@randomSeed)

    # Force numberSamples to 2
```

```
# This will be removed on future versions.
smessage("Forcing number of conditions to 2.")
.Object@numberSamples <- 2

# Checking
if (.Object@numberSamples < 2 && length(.Object@times) < 2) {
    stop("Simulating one condition requires a time series.")
}

# Disable GTF file
if (! is.null(.Object@GTF)) {
    warning("Support of GTF files is temporarily disabled.")
    .Object@GTF <- NULL
}

# Retrieve gene names and ranges from GTF file if possible
if (! is.null(.Object@GTF)) {
    # Load GTF from filename
    if (! class(.Object@GTF) == 'TxDb') {
        smessage("Trying to load GTF file: ", .Object@GTF)

        .Object@GTF <- makeTxDbFromGFF(.Object@GTF, format="gtf", organism =
        ↪  .Object@organism)
    }

    .Object@GRanges <- exons(.Object@GTF, columns = c("EXONID", "EXONNAME"))
    .Object@geneNames <- mcols(.Object@GRanges)$EXONNAME

    smessage("Gene names and GRanges object loaded from GTF file.")

    # Override the total number of genes
    .Object@totalGenes <- length(.Object@geneNames)

} else {
    # Random generated IDs
    if (length(.Object@geneNames) == 0) {

        smessage("Generating gene names")
        .Object@geneNames <- paste("g", formatC(1:.Object@totalGenes, width =
        ↪  nchar(.Object@totalGenes),
                                                format = "d", flag = "0"), sep ="")
    }
    else if (length(.Object@geneNames) != .Object@totalGenes) {
        smessage("The number of genes names passed is not equal to the number of total
        ↪  genes. Adjusting.")
        .Object@totalGenes <- length(.Object@geneNames)
    }

    # GRanges object needed for simulators based on locations rather tan genes
    if (length(.Object@GRanges) == 0 && ! is.null(.Object@GRanges)) {
        smessage("Generating random GRanges object")
```

```
        .Object@genomeSize <- max(.Object@genomeSize,
        ↪   .Object@totalGenes*.Object@meanGeneSize)
        .Object@GRanges <- .Simulation.GRanges(.Object)
    }
}

# Create the simulation settings only if they haven't been passed as options.
if (length(.Object@simSettings) == 0) {

    smessage("Generating simulation settings")

    # Number of times considered
    nt <- length(.Object@times)

    # Number of columns
    cond <- .Object@numberReps*.Object@numberSamples*nt

    # Average size of "gene classes" (DE genes): number_DE_genes/number_classes
    M <- .Object@diffGenes*.Object@exprGenes/.Object@nGeneClasses

    smessage("\t- Number of gene classes: ", .Object@nGeneClasses)
    smessage("\t- Avg size of gene classes: ", M)

    # Initialize default expression patterns
    if (length(.Object@defaultPatterns) == 0) {
        .Object@defaultPatterns <- .Simulator.defaultPatternCoefficients(.Object)
    } else {
        .Object@defaultPatterns <- .Object@defaultPatterns
    }

    patternNumber <- length(.Object@defaultPatterns)
    patternInduction <- seq(from=1, to=patternNumber, by=2)

    # No expr, flat, expr
    DEGsampleOptions <- c(-1, 0, 1)

    # Only flat for NoDEG genes
    NoDEGsampleOptions <- c(0) #c(0, 1)
    NoExprsampleOptions <- c(-1)

    # Limit the availabily of patterns
    DEGpatterns <- .Object@defaultPatterns
    NoDEGpatterns <- NULL
    NoExprpatterns <- NULL

    # Min number of gene classes = number of patterns
    nGeneClassesDEG <- max(.Object@nGeneClasses, patternNumber)

    # Possible number of combinations: possibleValues^numberOfSamples
    nCombinations <- (length(DEGsampleOptions)^.Object@numberSamples - 1) * patternNumber
```

```
# TODO: change this limit (possible combinations * patternNumber)
if (nGeneClassesDEG %% 2 > 0 || nGeneClassesDEG > nCombinations) {
    warning("Invalid number of gene classes: must be even and not greater than
     ↪  nOptions^nSamples [", nCombinations, "]")
    smessage("Adjusting number of classes to maximum allowed.")

    nGeneClassesDEG <- nCombinations
    .Object@nGeneClasses <- nGeneClassesDEG
}

# Possible number of combinations: possibleValues^numberOfSamples
# Force to 2 to maintain an even value
nCombinationsNoDEG <- max((length(NoDEGsampleOptions)^.Object@numberSamples - 1) *
 ↪  patternNumber, 2)

# Max number of gene classes for non-DEG (max 2 combinations)
nGeneClassesNoDEG <- min(nCombinationsNoDEG, nGeneClassesDEG)

# No expressed genes
nCombinationsNoExpr <- 2
nGeneClassesNoExpr <- 2

# Simulators provided excluding RNA-seq
omicSimulators <- names(.Object@simulators)[which(names(.Object@simulators) !=
 ↪  "SimRNAseq")]

# Only RNA-seq: no regulatory programs
if (length(omicSimulators) < 1) {
    # Dummy regulatory program to be used only to keep things in order.
    # Because there's no loaded Simulator called "Dummy" it won't affect
    # the results.
    omicSimulators <- list("Dummy")

    .Object@nRegPrograms <- 0
}

# Adjust the number of regulatory programs at least to the same
# number of gene classes
nRegProgramsDEG <- max(nGeneClassesDEG, .Object@nRegPrograms)
nRegProgramsNoDEG <- max(nGeneClassesNoDEG, .Object@nRegPrograms)
nRegProgramsNoExpr <- max(nGeneClassesNoExpr, .Object@nRegPrograms)

# TODO: limit the number of RP?
nDEG <- round(.Object@diffGenes * .Object@exprGenes, 0)
nNoDEG <- .Object@exprGenes - nDEG
nNoExpr <- .Object@totalGenes - (nDEG + nNoDEG)

# Define all the possible combinations of 1/0 with the number of considered simulators.
# TODO: remove (0, 0, 0...) from them?
```

```
RPOptions <- as.list(data.frame(t(permutations(2, length(omicSimulators), 0:1,
 ↪  repeats.allowed=T))))

condRunif <- function(n, total, use.avg = TRUE, use.jitter = TRUE) {
    if (use.avg) {
        mRD <- rnorm(total)
        mKQ <- quantile(mRD, 0:n/n)
        x <- as.vector(table(cut(mRD, mKQ, include.lowest=TRUE)))

        if (use.jitter) {
            x <- jitter(x, factor=n)

            x[x<=0] <- 1
        }
    } else {
        x <- runif(n, min=1, max=total-n)
    }

    x * total / sum(x)
}

distributePatterns <- function(nOptions, nGeneClasses, nRegPrograms, nGenes, patterns) {

    # Direct assignation (GC1 -> Pattern1, GCn -> PatternN)
    geneClassToPattern <- 1:nGeneClasses

    # Every pattern can be present 3 times: condition 1, condition 2 or both
    # If gene class numbers is greater than number of patterns, the later
    # ones are "recycled", we allow them to appear 1 or 2 more times, depending
    # on DE/NO DE genes
    patternNumber <- length(patterns)

    oddPositions <- seq(patternNumber+1, nGeneClasses, by=2)

    # After reaching the number of patterns, assign them randomly and by
    # blocks of induction/repression (odd/even indexes)
    geneClassToPattern[oddPositions] <-
        sample(rep(patternInduction, max(nOptions - 1, 1)),
         ↪  (nGeneClasses-patternNumber)/2, replace=F)
    geneClassToPattern[oddPositions+1] <- geneClassToPattern[oddPositions] + 1

    # Number of regulatory programs per gene class
    # TODO: change to min=avg number
    geneClassNRP <- .Simulation.roundfixS(condRunif(nGeneClasses, nRegPrograms,
     ↪  use.jitter=FALSE))

    # List linking every gene class to one or more regulatory program
    geneClassToRP <- split(1:nRegPrograms,
                           rep(seq_along(geneClassNRP), times=geneClassNRP))

    RPtoGeneClass <- vector("list", nRegPrograms)
```

```
    for (gc in names(geneClassToRP)) {
        RPtoGeneClass[geneClassToRP[[gc]]] <- gc
    }

    # Randomly activate omics in every regulatory program
    # TODO: check that RP from the same gene class are DIFFERENT, but first
    # decide what to do when there's more reg. programs than possible combinations
    # between the omics.
    regPrograms <- matrix(mapply(rbind, RPOptions[sample(1:length(RPOptions),
    ↪  nRegPrograms, replace=T)]), ncol=length(omicSimulators))

    # Force matrix when regPrograms is a vector (one simulator)
    colnames(regPrograms) <- omicSimulators

    # In every regProgram count the number of activated regulators
    # and define the weight of every one of them on gene expression.
    # DISABLED for the time being
    #             regProgramsDist <- t(apply(regPrograms, 1, function(lRow) {
    #                rInd <- which(lRow == 1)
    #                rWeight <- 1/length(rInd)
    #
    #                lRow[rInd] <- rWeight
    #                lRow
    #             }))
    #
    #             colnames(regProgramsDist) <- omicSimulators

    # Divide the gene number between the different regulatory programs
    geneClassNumber <- .Simulation.roundfixS(condRunif(nRegPrograms, nGenes))

    list(
        'geneClassToPattern'=geneClassToPattern,
        'geneClassToRP'=geneClassToRP,
        'RPtoGeneClass'=RPtoGeneClass,
        # 'regProgramsDist'=regProgramsDist,
        'regPrograms'=regPrograms,
        'geneClassNumber'=geneClassNumber
    )
}

# DEG block
dataDEG <-
    distributePatterns(
        nOptions = length(DEGsampleOptions), nGeneClasses = nGeneClassesDEG,
        ↪  nRegPrograms = nRegProgramsDEG, nGenes = nDEG, patterns = DEGpatterns
    )

# Flat block
dataNoDEG <-
    distributePatterns(
```

```
        nOptions = length(NoDEGsampleOptions), nGeneClasses = nGeneClassesNoDEG,
        ↪  nRegPrograms = nRegProgramsNoDEG, nGenes = nNoDEG, patterns = NoDEGpatterns
    )

# No expr. block
dataNoExpr <-
    distributePatterns(
        nOptions = length(NoExprsampleOptions), nGeneClasses = nGeneClassesNoExpr,
        ↪  nRegPrograms = nRegProgramsNoExpr, nGenes = nNoExpr, patterns =
        ↪  NoExprpatterns
    )

smessage("DEG information: ")
smessage(dataDEG)

smessage("NO DEG information:")
smessage(dataNoDEG)

smessage("NO Expr information:")
smessage(dataNoExpr)

# Possible options:
# -1 => not expressed
#  0 => flat
#  1 => expressed
#
#  Generate all possible combinations in N conditions.
sampleValues <- as.list(data.frame(t(permutations(length(DEGsampleOptions),
↪  .Object@numberSamples, DEGsampleOptions, repeats.allowed=T))))

# Remove invalid values (like not expressed-...-not expressed)
invalidValues <- list(rep(-1, .Object@numberSamples))
sampleValues <- sampleValues[! sampleValues %in% invalidValues]

# Flat & no expr genes
# The same on every sample
sampleValuesNoDEG <- lapply(NoDEGsampleOptions, function(o) rep(o,
↪  .Object@numberSamples))
sampleValuesNoExpr <- lapply(NoExprsampleOptions, function(o) rep(o,
↪  .Object@numberSamples))

# Assign a different expression scheme (by condition) when the
# pattern of expression is the same.
generateTratmatrix <- function(sampleValues, nGeneClasses, geneClassToPattern) {
    # Zero filled matrix by default
    trat <- matrix(0, ncol = nGeneClasses, nrow = .Object@numberSamples)

    # For every induction pattern (odd indexes)
    for (i in patternInduction) {
        # Check the number of appearances
        # Index = number of column
```

```
        patternIndexes <- which(geneClassToPattern == i)

        # Induction (DEG)
        trat[, patternIndexes] <- unlist(sample(sampleValues, length(patternIndexes),
        ↪  replace=F))
        # Repression (DEG)
        trat[, patternIndexes + 1] <- trat[, patternIndexes]
    }

    trat
}

tratDEG <-
    generateTratmatrix(
        sampleValues = sampleValues, nGeneClasses = nGeneClassesDEG, geneClassToPattern
        ↪  = dataDEG$geneClassToPattern
    )

tratNoDEG <-
    generateTratmatrix(
        sampleValues = sampleValuesNoDEG, nGeneClasses = nGeneClassesNoDEG,
        ↪  geneClassToPattern = dataNoDEG$geneClassToPattern
    )

tratNoExpr <-
    generateTratmatrix(
        sampleValues = sampleValuesNoExpr, nGeneClasses = nGeneClassesNoExpr,
        ↪  geneClassToPattern = dataNoExpr$geneClassToPattern
    )

# If the genes are expressed in both conditions (1-1) modify the
# fold change in one of them to make them dif. expressed.
FCsampleOptions <- c(1.5, 1.0, -1.5)
FCsampleValues <- as.list(data.frame(t(permutations(length(FCsampleOptions),
↪  .Object@numberSamples, FCsampleOptions, repeats.allowed=T))))

# Remove invalid values (like not expressed-...-not expressed)
FCinvalidValues <- list(rep(-1.5, .Object@numberSamples),
                        rep(1.5, .Object@numberSamples),
                        rep(1.0, .Object@numberSamples))
FCsampleValues <- FCsampleValues[! FCsampleValues %in% FCinvalidValues]


tratFC <- matrix(1.0, ncol=ncol(tratDEG), nrow=nrow(tratDEG))

tratFC <- apply(tratDEG, 2, function(col) {
    # All elements equal
    if (length(unique(col)) == 1) {
        col <- unlist(sample(FCsampleValues, 1))
    } else {
        col <- rep(1, .Object@numberSamples)
```

```
    }

    col
})

# Overwrite even positions
inductionIndexes <- seq(1, ncol(tratFC), by=2)
tratFC[, inductionIndexes + 1] <- tratFC[, inductionIndexes]

smessage("\t- Treatments & expression matrix (DEG):")
smessage(tratDEG)
smessage(dataDEG$geneClassToPattern)

smessage("\t- Treatments & expression matrix (No DEG):")
smessage(tratNoDEG)
smessage(dataNoDEG$geneClassToPattern)

smessage("\t- Treatments & expression matrix (No Expr):")
smessage(tratNoExpr)
smessage(dataNoExpr$geneClassToPattern)

smessage("\t- Treatments & expression matrix (FC):")
smessage(tratFC)

assignGenes <- function(geneClassNumber, geneIDs) {
    geneGroups <- list()

    for (i in 1:length(geneClassNumber)) {
        sampleSize <- geneClassNumber[i]
        geneSample <- sample(geneIDs, sampleSize)
        geneGroups[[i]] <- geneSample

        geneIDs <- setdiff(geneIDs, geneSample)
    }

    geneGroups
}

# Randomly assign genes to a certain expression pattern
geneGroupsDE <- assignGenes(dataDEG$geneClassNumber, .Object@geneNames)
geneGroupsNoDE <- assignGenes(dataNoDEG$geneClassNumber, setdiff(.Object@geneNames,
↪  unlist(geneGroupsDE)))
geneNoExpr <- assignGenes(dataNoExpr$geneClassNumber, setdiff(.Object@geneNames,
↪  c(unlist(geneGroupsDE), unlist(geneGroupsNoDE))))

.Object@simSettings <- list(
    'DEG'=list(
        'tProfiles'=tratDEG,
        'geneClassNumber'=dataDEG$geneClassNumber,
        'geneGroups'=geneGroupsDE,
        'geneClassPatterns'=dataDEG$geneClassToPattern,
```

```
                'geneClassToRP'=dataDEG$geneClassToRP,
                'RPtoGeneClass'=dataDEG$RPtoGeneClass,
                'regPrograms'=dataDEG$regPrograms,
                # 'regProgramsDist'=dataDEG$regProgramsDist,
                "fcMatrix"=tratFC
            ),
            'NoDEG'=list(
                'tProfiles'=tratNoDEG,
                'geneClassNumber'=dataNoDEG$geneClassNumber,
                'geneGroups'=geneGroupsNoDE,
                'geneClassPatterns'=dataNoDEG$geneClassToPattern,
                'geneClassToRP'=dataNoDEG$geneClassToRP,
                'RPtoGeneClass'=dataNoDEG$RPtoGeneClass,
                'regPrograms'=dataNoDEG$regPrograms
                # 'regProgramsDist'=dataNoDEG$regProgramsDist
            ),
            'NoExpr'=list(
                'geneGroups'=geneNoExpr,
                'tProfiles'=tratNoExpr,
                'geneClassNumber'=dataNoExpr$geneClassNumber,
                'geneClassPatterns'=dataNoExpr$geneClassToPattern,
                'geneClassToRP'=dataNoExpr$geneClassToRP,
                'RPtoGeneClass'=dataNoExpr$RPtoGeneClass,
                'regPrograms'=dataNoExpr$regPrograms
            )
        )
    }

    # Create an environment to prevent R from eating large quantities of
    # memory by creating copies from the objects.
    .Object@env <- new.env(parent=emptyenv())
    # This will copy the object to the environment
    .Object@env$instance = .Object

    # Create simulators
    # Work on the instance of the environment
    for (simName in names(.Object@env$instance@simulators)) {
        if (! inherits(.Object@env$instance@simulators[[simName]], "Simulator")) {
            .Object@env$instance@simulators[[simName]]$simulation <- .Object@env
            .Object@env$instance@simulators[[simName]] <- do.call(new, c("Class"=simName,
            ↪   .Object@env$instance@simulators[[simName]]))
        }
    }

    .Object@env$instance
}

.Simulation.GRanges <-  function(object) {
    # This method requires a lot of work, it simulates a very basic genome so
    # it shouldn't be used right now.
    #
```

```
    # Taken from biostars post

    # assuming read lengths are poisson distributed around the mean length, you could use a
    ↪  different random function
    # assuming reads are uniformly distributed wrt. start position
    ir <- IRanges(start=as.integer(runif(object@totalGenes, min=1, max=object@genomeSize)),
                  width=rpois(object@totalGenes, object@meanGeneSize),
                  names=object@geneNames)

    # cut off reads at the end
    ir <- restrict(ir, 1, object@genomeSize)

    # Only one chromosome in seqnames
    gr <- GRanges(ranges=ir , seqnames="1")

    gr
}

.Simulation.simulate <- function(object, force = FALSE, freeMemory = FALSE, ...) {
    # Don't simulate unless there's no simulation data available
    # or we force it.
    if (length(object@simData) == 0 || force) {
        # Initialize data container
        simData <- list()

        # Make sure SimRNAseq is the first to be simulated.
        simOrder <- unique(c(list("SimRNAseq"), names(object@simulators)))

        # Simulation for every simulator loaded
        for (simName in simOrder) {
            # Keep an updated copy of the simulator object after updating
            # base values
            object@simulators[[simName]] <- simulate(object@simulators[[simName]])

            # Simulate method saves the simulated values on the internal
            # container .sdata under the key svalues.
            #
            # We put them in simulation simData slot for easier access.
            simData[[simName]] <- object@simulators[[simName]]@.sdata$svalues

            # Remove the object after the simulation
            # TEMP. DISABLED
            #             if (freeMemory) {
            #                 #rm()
            #                 object@simulators[[simName]] <- NULL
            #             }


            # Update instance from the environment to reflect latest changes
            # before calling others simulators.
            object@simData <- simData
```

```
            object@env$instance <- object
        }
    }

    object
}

# Function from mist-goodies.R (package sfsmisc available at CRAN)
.Simulation.roundfixS <- function(x, method = c("offset-round", "round+fix", "1greedy"))
{
    ## Purpose: y := r2i(x) with integer y  *and* sum(y) == sum(x)
    ## Author: Martin Maechler, 28 Nov 2007
    n <- length(x)
    x0 <- floor(x)
    e <- x - x0 ## == (x %% 1) in [0, 1)
    S. <- sum(e)
    stopifnot(all.equal(S., (S <- round(S.))))
    method <- match.arg(method)

    ## The problem is equivalent to transforming
    ##   e[] \in [0,1)  into  f[] \in {0,1},  with sum(e) == sum(f)
    ## Goal: transform e[] into f[] gradually, by "shifting" mass
    ##       such that the sum() remains constant

    switch(method,
           "offset-round" = {
               ## This is going to be equivalent to
               ##  r := round(x + f)  with the correct     f \in [-1/2, 1/2], or
               ##  r == floor(x + f + 1/2) = floor(x + g), g \in    [0, 1]
               ##
               ## Need  sum(floor(e + g)) = S;
               ## since sum(floor(e)) == 0, sum(floor(e+1)) == n,
               ## we just need to floor(.) the S smallest, and ceiling(.) the others
               if(S > 0) {
                   r <- numeric(n) # all 0; set to 1 those corresponding to large e:
                   r[sort.list(e, decreasing=TRUE)[1:S]] <- 1
                   x0 + r
               } else x
           }, ## end{offset-round}

           "round+fix" = {
               r <- round(e)
               if((del <- S - sum(r)) != 0) { # need to add +/- 1 to 'del' entries
                   s <- sign(del) ## +1 or -1: add +1 only to r < x entries,
                   aD <- abs(del) ##          and -1 only to r > x entries,
                   ## those with the "worst" rounding are made a bit worse
                   if(del > 0) {
                       iCand <- e > r
                       dx <- (e - r)[iCand] # > 0
                   } else {                 ## del < 0
                       iCand <- e < r
```

```
                dx <- (e - x)[iCand] # > 0
            }
            ii <- sort.list(dx, decreasing = TRUE)[1:aD]
            r[iCand][ii] <- r[iCand][ii] + sign(del)
        }

        return(x0 + r)

    }, ## end{round+fix}

    "1greedy" = {
        ii <- e != 0
        while(any(ii)) {
            ci <- cumsum(ii) # used to revert  u[ii] subsetting
            m <- length(e. <- e[ii])
            ie <- sort.list(e.)  # both ends are relevant
            left <- e.[ie[1]] < 1 - e.[ie[m]]
            iThis  <- if(left) 1 else m
            iother <- if(left) m else 1
            J <- which.max(ci == ie[iThis]) ## which(.)[1]  but faster
            I <- which.max(ci == ie[iother])
            r <- x[J]
            x[J] <- k <- if(left) floor(r) else ceiling(r)
            mass <- r - k         # if(left) > 0 else < 0
            if(m <= 2) {   # short cut and evade rounding error
                if(m == 1) {      # should happen **rarely**
                    if(!(min(abs(mass), abs(1-mass)) < 1e-10))
                        warning('m==1 in "1greedy" w/ mass not close to {0,1}')
                } else { ## m==2
                    x[I] <- round(x[I] + mass)
                }
                break ## ii <- FALSE
            }
            else { ## m >= 3
                e[J] <- if(left) 0 else 1
                ii[J] <- FALSE
                ## and move it's mass to the other end:
                e.new <- e[I] + mass
                if(e.new > 1)
                    stop("e[I] would be > 1 -- internal error")
                else if(e.new < 0)
                    stop("e[I] would be < 0 -- internal error")
                x[I] <- x[I] + mass
                e[I] <- e.new
            } ## m >= 3
        }     ## end{while}
        x

    }) # end{switch}
}## roundfixS
```

```
.Simulation.sampleGeneNames <- function(object, size = 0, type = c('DEG', 'NoDEG')) {
    genePool <- unlist(lapply(object@simSettings[type], '[[', 'geneGroups'))

    # Don't use replace (we need unique results)
    geneSample <- sample(genePool, size, replace = FALSE)
    geneSample
}

.Simulation.generateGraphics <- function(object, reportData=NULL, blockSize=NULL) {

    # TEMP FIX UNTIL THE DEV VERSION OF ggplot REACHES CRAN SERVERS
    # BYPASS CLASS CHECK
    ggsave <- ggplot2::ggsave; body(ggsave) <- body(ggplot2::ggsave)[-2];

    if (is.null(reportData)) {
        reportData <- .Simulation.reportData(object)
    }

    # Convert matrix:
    #       Cond1   ... CondN
    #   g1
    #   ..
    #   gn
    #
    #  To DF:
    #
    #
    .prepareGGplotDF <- function(simuData) {
        # In THIS case, a class diferent than matrix implies that we are
        # analyzing only 1 record.
        if (class(simuData) != 'matrix') {
            simuData <- matrix(simuData, nrow=1)
        }

        # Change names of columns
        colnames(simuData) <- rep(rep(object@times, each=object@numberReps),
         ↪  object@numberSamples)

        sLongData <- data.frame()

        # Number of columns per sample
        cColumnSize <- object@numberReps*length(object@times)

        for (c in 1:object@numberSamples) {
            # Subset of one sample
            cSimuData <- simuData[, ((c-1)*cColumnSize + 1):(c*cColumnSize)]

            for (r in 1:object@numberReps) {
                # Subset of one replicate
                rIndexes <- seq(from=r, to=cColumnSize, by=object@numberReps)
```

```
            rSimuData <- melt(cSimuData[, rIndexes], varnames=c('gene', 'time'))
            rSimuData$sample <- c
            rSimuData$rep <- r

            sLongData <- rbind(sLongData, rSimuData)
        }
    }

    sLongData
}

.getPattern <- function(simulator, group, settings) {
    sGeneClass <- settings$RPtoGeneClass[[group]]
    sPattern <- settings$geneClassPatterns[as.integer(sGeneClass)]

    if (simulator@regulator && ! simulator@enhancer) {
        sPattern <- if(sPattern %% 2 == 0) (sPattern - 1) else (sPattern+1)
    }

    sNames <- names(object@defaultPatterns)

    sNames[[sPattern]]
}

# Layout calculation
# Add one row for the global title
pageRows <- length(object@simData) + 1
pageColumns <- 1

# Loaded simulators
simNames <- names(object@simData)

for (gType in c('DEG', 'NoDEG', 'NoExpr')) {
    sSettings <- object@simSettings[[gType]]

    for (nG in 1:length(sSettings$geneGroups)) {
        # Genes for this gene group
        gGroup <- sSettings$geneGroups[[nG]]

        # Divide into chunks of the size specified
        if (! is.null(blockSize)) {
            gBlocks <- split(gGroup, ceiling(seq_along(gGroup)/blockSize))
        } else {
            # gBlocks <- list(list(gGroup))
            gBlocks <- list(gGroup)
        }

        # Grob list for multiplot
        sGrobs <- list()
```

```
# Create new graphics
pdfFilename <- paste("graphics/RP_", nG, "_", format(Sys.time(), "%m_%d_%Y_%H:%M"),
↪  ".pdf", sep="")

# Graphics for every block
for (nBlock in 1:length(gBlocks)) {
    # Repeat for every simulator
    for (simu in 1:length(simNames)) {
        simClass <- simNames[[simu]]
        simObject <- object@simulators[[simClass]]

        groupIDs <- IDfromGenes(simObject, gBlocks[[nBlock]])

        .setLabels <- function(variable, values){
            # Sample label
            if (variable == 'sample') {

                expNames <- list('-1'="No expr.", '0'="Flat", '1'="Expr")

                sLabel <- sapply(values, function(value) {
                    # Check status
                    expStatus <- sSettings$tProfiles[nG, value]

                    sLabel <- paste("Sample ", value, "\n[",
                    ↪  expNames[[as.character(expStatus)]],
                                "]\n(", .getPattern(simObject, nG, sSettings),
                                ↪  ")", sep="")
                })
            } else {
                # Rep label
                sLabel <- paste("Rep", values)
            }

            sLabel
        }

        # Values for this simulator and gene group
        allData <- reportData[[gType]][[simClass]][[paste("RP", nG, sep="")]]
        blockData <- allData[intersect(groupIDs, rownames(allData)),]

        if (length(blockData) > 0) {
            ggDataFrame <- .prepareGGplotDF(blockData)

            sPlot <- ggplot(ggDataFrame, aes(x=time, y=value, colour=gene)) +
                        geom_line() +
                        facet_grid(rep ~ sample, labeller=.setLabels) +
                        labs(x="Time", y="Value") +
                        ggtitle(simClass)
        } else {
            sPlot <- ggplot(data.frame()) +
                        geom_point() +
```

```
                                    xlim(0, 10) +
                                    ylim(0, 100) +
                                    ggtitle(paste("No genes present on", simClass))
                    }

                    blockData <- NULL

                    sGrobs <- append(sGrobs, list(sPlot))
                }
            }

            mPlots <- marrangeGrob(sGrobs, nrow = 1, ncol = 1)

            ggsave(pdfFilename, mPlots)
        }
    }
}

.Simulation.reportData <- function(object) {
    reportData <- list()

    # Create a set of matrices for globaldata (all genes) with the following columns:
    cFields <- c("Group", "Gene class", "Reg. program", "Pattern", paste("Expr. sample ",
    ↪  1:object@numberSamples, sep=""))

    globalData <- lapply(names(object@simulators),
                         function(n) matrix(0, ncol=length(cFields), nrow=0, dimnames =
                         ↪  list(NULL, cFields)))
    names(globalData) <- names(object@simulators)

    for (gType in c('DEG', 'NoDEG', 'NoExpr')) {
            genData <- with(object@simSettings[[gType]], {
                rData <- list()

                # Trat matrix based on regulatory programs
                tratRP <- matrix(0, nrow=length(geneGroups), ncol=object@numberSamples + 1)

                for (i in 1:nrow(tratRP)) {
                    geneClass <- as.integer(RPtoGeneClass[[i]])

                    tratRP[i, 1:object@numberSamples] <- tProfiles[, geneClass]
                    tratRP[i, object@numberSamples + 1] <- geneClassPatterns[geneClass]
                }

                colnames(tratRP) <- c(paste("Cond", 1:object@numberSamples), "Pattern")

                rData$tratRP <- tratRP

                if (gType == 'DEG') {
                    rData$FCmatrix <- fcMatrix
                }
```

```
        for (sim in names(object@simulators)) {

            simRepdata <- reportData(object@simulators[[sim]])
            sGeneGroups <- lapply(geneGroups, function(g) {
             ↪  IDfromGenes(object@simulators[[sim]], g)})
            sintGeneGroups <- lapply(sGeneGroups, function(g) {
             ↪  intersect(rownames(simRepdata$simValues), g)})

            # All genes (inside DEG or NoDEG)
            rData[[sim]]$ALL <- simRepdata$simValues[unlist(sintGeneGroups),]

            # Base values
            for (c in 1:object@numberSamples) {
                rData[[sim]][[paste("BC", c, sep="")]] <-
                 ↪  as.matrix(simRepdata$baseData[[c]])
            }

            # Gene groups
            for (i in 1:length(sGeneGroups)) {
                RPdata <- simRepdata$simValues[sintGeneGroups[[i]],]

                rData[[sim]][[paste("RP", i, sep="")]] <- RPdata
                # Gtype, geneClass, regProgram, sample1status, ..., sampleNstatus
                sampleStatus <- sapply(1:object@numberSamples, function(s) tratRP[i, s])
                group <- gType
                geneClass <- RPtoGeneClass[[i]]
                regProgram <- i
                gcPattern <- tratRP[i, "Pattern"]

                gdataInfo <- matrix(c(group, geneClass, regProgram, gcPattern,
                 ↪  sampleStatus),
                                    byrow=TRUE,
                                    ncol=length(cFields),
                                    nrow=nrow(RPdata),
                                    dimnames=list(rownames(RPdata), cFields)
                                    )

                globalData[[sim]] <- rbind(globalData[[sim]], gdataInfo)
            }
        }

        list(
            localData=rData,
            globalData=globalData
        )
    })

    reportData[[gType]] <- genData$localData
    globalData <- genData$globalData
}
```

```
    reportData$Global <- globalData

    reportData
}

#' multiOmicSimulation
#'
#' Performs a multiomic simulation by chaining two actions:
#'  1) Creating the "Simulation" class with the provided params.
#'  2) Calling "simulate" method on the initialized object.
#'
#' @param ... Options to be passed to simulation object
#'
#' @return Instance of class "Simulation" with slot "simData" containing the multiomic
↪   simulation data.
#' @export
#'
#' @examples
#'
#' Start empty simulation with default params:
#'
#'  moSimulation <- multiOmicSimulation()
#'  # Retrieve simulated data, it will only contain a "SimRNAseq" key.
#'  moSimulation@simData
#'
#'
#' Simulation with every omic and some custom options:
#'
#'  simulatorOptions <- list(
#'      'SimRNAseq'=list(
#'          'depth'=25
#'      ),
#'      'SimMethylseq'=list(
#'      ),
#'      'SimMiRNAseq'=list(),
#'      'SimDNaseseq'=list(),
#'      'SimChIPseq'=list()
#'  )
#'
#'  moSimulation <- multiOmicSimulation(
#'      numberReps = 3,
#'      times = c(0, 2, 6, 12, 24),
#'      randomSeed = 1234,
#'      simulators = simulatorOptions
#'  )
#'
#'  # simData will contain a key for every simulator
#'  dataRNAseq <- moSimulation@simData$SimRNAseq
#'
#'  # Methylseq is one exception, having both "beta" and "M" keys
```

```
#'  dataMethylSeqBeta <- moSimulation@simData$SimMethylseq$beta
#'
multiOmicSimulation <- function(...) {
    oSim <- do.call(new, c("Class"="Simulation", list(...)))
    oSim <- simulate(oSim, freeMemory = FALSE)

    oSim
}

setMethod("initialize", signature="Simulation", .Simulation.initialize)
setMethod("sampleGeneNames", signature="Simulation", .Simulation.sampleGeneNames)
setMethod("simulate", signature="Simulation", .Simulation.simulate)
setMethod("reportData", signature="Simulation", .Simulation.reportData)
```

A.2.4. Simulator.R

```
# This function will be used by Simulation class to distribute the genes
# between different groups based on their expression profile.
#
# Child classes override the function "patternCoefficients" but the
# returned value must contain at least the names of the list returned by
# this one.
.Simulator.defaultPatternCoefficients <- function(simulationObject) {
    nt <- length(simulationObject@times)

    x <- c(0:(nt-1))

    a1 <- 0
    b1 <- 1/x[nt]

    a2 <- 0
    b2 <- 4*(x[nt])/(x[nt]*x[nt])
    c2 <- -4/(x[nt]*x[nt])


    list(
        contind=c(a1,b1,0),
        contrep=c(a1,b1,0),
        tranind=c(a2,b2,c2),
        tranrep=c(a2,b2,c2)
    )
}

.Simulator.initialize <- function(.Object, ...) {
    .Object <- callNextMethod()

    # Load default data if not provided. This is done before calling configureSimulator
    # because doesn't need any modifications but could be needed for some tweaks
    # in specific simulators.
    sName <- class(.Object)
```

```
# If no data is provided, simulation@defaultData must be initialized
if (length(.Object@data) < 2 && exists(sName,
↪   where=.Object@simulation$instance@defaultData)) {
    smessage("Loading default values for ", .Object@name)

    dData <- .Object@simulation$instance@defaultData[[sName]]

    # If data is not available (methylation) return a vector of length 0
    .Object@data <- if(exists('data', dData)) dData$data else vector()

    # If association table is not available (RNA-seq) return NA
    # (Simulator options are contained on a list, and a list can't have NULL
    # values so we use NA)
    .Object@idToGene <- if(exists('idToGene', dData)) dData$idToGene else matrix(NA)

    if (! .Object@regulator && exists('geneLength', dData)) {
        .Object@geneLength <- dData$geneLength
    }
}

# Child classes use another function to initialize the instance
# called "configureSimulator" instead of the predefined constructor.
# This way we can benefit from callNextMethod to assign the slots but
# also configure required things _before_ calling initializeData here.
.Object <- configureSimulator(.Object, ...)

# Set required gammaTable if exists and hasn't been defined yet.
gammaValue <- paste0(".", class(.Object), ".gammaTable", collapse="")

if (is.null(.Object@gammaTable) && exists(gammaValue)) {
        .Object@gammaTable <- get(gammaValue)
}

# Complete data initialization
bData <- initializeData(.Object)

.Object@data <- bData$values
.Object@idToGene <- bData$IDtable

# Be sure to have a copy of initial data for every factor in the form of a list.
if (class(.Object@data) != "list" || length(.Object@data) <
↪   .Object@simulation$instance@numberSamples) {
    .Object@data <- lapply(1:.Object@simulation$instance@numberSamples, function(r)
    ↪   .Object@data)
}

# Apply length bias to base values
if (.Object@lengthBias) {
    # Lineal function
    .lBiasLineal <- function (counts, lengths) {
```

```
            counts + counts*(lengths-mean(lengths, na.rm = TRUE))/mean(lengths, na.rm = TRUE)
        }

        # Quadratic function
        .lBiasQuad <- function (counts, lengths) {
            counts +
                counts*(lengths-mean(lengths, na.rm = TRUE))/mean(lengths, na.rm = TRUE) -
                counts*((lengths-mean(lengths, na.rm = TRUE))/mean(lengths, na.rm = TRUE))^2
        }

        lBiasPredef <- list(
            'lineal'=.lBiasLineal,
            'quadratic'=.lBiasQuad
        )

        rowLengths <- getLengths(.Object)

        if (! is.null(rowLengths)) {
            if (is.function(.Object@lengthBiasFunction)) {
                .Object@data <- lapply(.Object@data, function(sampleCounts) {
                ↪  .Object@lengthBiasFunction(sampleCounts, rowLengths) })
            } else {
                .Object@data <- lapply(.Object@data, function(sampleCounts) {
                ↪  lBiasPredef[[.Object@lengthBiasFunction]](sampleCounts, rowLengths) })
            }
        }
    }

    .Object
}

.Simulator.configureSimulator <- function(object, ...) {
    # Adjust sequencing depth
    object@depth <- object@depth*10^6

    object
}

.Simulator.initializeData <- function(object) {
    list(
        values=object@data,
        IDtable=object@idToGene
    )
}

# Return pattern coefficients defined by default (defaultPatternCoefficients)
.Simulator.patternCoefficients <- function(object, coefs=NULL) {

    if (is.null(coefs)) {
        coefs <- object@simulation$instance@defaultPatterns
    }
```

```
    nTimes <- length(object@simulation$instance@times)
    x <- c(0:(nTimes-1))

    timeMatrix <- rbind(c(rep(1, nTimes)), x, x*x)
    timeMatrixRev <- timeMatrix[, ncol(timeMatrix):1]

    # Multiply the inductions patterns by timeMatrix, and
    # the opposite by the reverse.
    #
    # If the regulator activates expression, the patterns will follow
    # the induction/repression block, however, if it isn't, that will
    # change to repression/induction, talking of indexes
    divRest <- if(! object@regulator || object@enhancer) 0 else 1

    for (i in 1:length(coefs)) {
        coefs[[i]] <- coefs[[i]] %*% (if(i %% 2 == divRest) timeMatrixRev else timeMatrix)
    }

    smessage("Defining pattern coefficients for simulator ", object@name)
    smessage("Regulator: ", object@regulator)
    smessage("Activator: ", object@enhancer)
    smessage("Coefs:")
    smessage(paste(coefs, sep="\n"))
    smessage("Increasing size of coefficients x :", object@coeffIncrease)
    coefs <- lapply(coefs, function(c) c*object@coeffIncrease)
    smessage(paste(coefs, sep="\n"))

    coefs
}

.Simulator.simulate <- function(object, simSettings = object@simulation$instance@simSettings,
↪   force = FALSE, ...) {
    if (force || length(object@.sdata$svalues) == 0) {

        smessage("\n\n\nStarting ", object@name, " simulation: \n")

        pCoeff <- patternCoefficients(object)

        # Check the validity of the pattern coefficients comparing with
        # the default ones. All the default profiles must be present.
        defaultPCoeff <- .Simulator.defaultPatternCoefficients(object@simulation$instance)

        if ( ! all(names(defaultPCoeff) %in% names(pCoeff)) ) {
            stop("The profile coefficients defined in ", object@name, " are invalid.")
        }

        fSimulation <- NULL

        # Repeat for all blocks of genes
        for (gType in c('DEG','NoDEG','NoExpr')) {
```

```
smessage("\nStarting loop for ", gType, ".")

fSimulation <- with(simSettings[[gType]], {
    # Simulate every gene associated to an expression class
    for (i in 1:length(geneClassToRP)) {
        smessage("\n- Entering gene group ", i)

        # Every gene group can have 1 or more RP associated
        for (r in 1:length(geneClassToRP[[i]])) {
            # Number of regulatory program
            nRP <- geneClassToRP[[i]][r]

            smessage("\n\tEntering regulatory program ", nRP, " with ",
             ↪  length(geneGroups[[nRP]]), " genes. Pattern=",
             ↪  geneClassPatterns[i])
            groupSimulation <- NULL

            for (j in 1:object@simulation$instance@numberSamples) {

                # Delegate simulation to specific simulators providing
                # them with the time coefficients.
                #
                # Trat matrix flags:
                # -1 -> not expressed
                #  0 -> flat
                #  1 -> expressed
                #
                groupStatus <- tProfiles[j, i]

                # For regulatory simulators (all except RNA-seq) keep
                # a flat expression profile when the genes are not expressed
                # or they do it with flat pattern. We do this initializing
                # a cTimeOps to 1 (flat) by default.
                cTimeOps <- rep(1, length(object@simulation$instance@times))
                updatePattern <- NULL
                geneIDs <- geneGroups[[nRP]]

                if (object@regulator) {
                    isEnabled <- regPrograms[r, class(object)]
                    smessage("\t[Omic ", object@name, " enabled in this RP: ",
                     ↪  isEnabled, "]")
                }

                # Group not expressed on this condition
                if (groupStatus < 0) {

                    smessage("\tGroup not expressed [Omic ", object@name, " using
                     ↪  pattern noexpr]")

                    # If it's a regulator, keep the "flat" profile but
                    # change the base values (or not, but give the possibility)
```

```
    if (object@regulator) {
        if (isEnabled) {
            updatePattern <- 'noexpr'
        }
    } else {
        # Non-expressed on RNA-seq must be filled with zeros
        cTimeOps <- cTimeOps*0
    }
}
# Group expressed in this condition and not flat
else if (groupStatus > 0) {
    # If it's a regulator, check if it's activated for
    # this regulatory program:
    #
    # - Enabled: modify base values according to
    #   the function patternValues and the pattern.
    #
    # - Disabled: modify base values using the
    #   'disabled' pattern and keep the flat
    #   profile.
    #
    if (object@regulator) {
        if (isEnabled) {
            updatePattern <- geneClassPatterns[i]
            cTimeOps <- pCoeff[[geneClassPatterns[i]]]
        } else {
            updatePattern <- 'disabled'
        }
    } else {
        cTimeOps <- pCoeff[[geneClassPatterns[i]]]
    }
}

if (! is.null(updatePattern)) {
    FCmod <- if (gType == 'DEG') fcMatrix[j, i] else 1
    # regWeight <- regProgramsDist[r, class(object)]

    object <- updateData(object, updatePattern, geneIDs,
     ↪  sampleNumber=j, changeFC=FCmod)#, regWeight = regWeight)
}

smessage("\t* [", object@name, "][", gType, "] Entering factor", j,
 ↪  " [", groupStatus, "][",
        paste(cTimeOps, collapse=", "), "] (Regulator: ",
        object@regulator, ")")

geneIDs <- IDfromGenes(object, geneIDs)

iGroupSimulation <- groupSimulation(object, geneNames=geneIDs,
                                    timeCoefficients=cTimeOps,
                                     ↪  sampleNumber = j,
```

```
                                                    geneGroup = nRP, geneBlock =
                                                    ↪  gType)

                        # Check that the returned value is not null (for
                        # example in regulators based on regions without
                        # an entry on region-gene table) and, if it is,
                        # create a dummy matrix with 0 rows and the appropiate
                        # number of columns or else "rbind" will fail.
                        if (! is.null(iGroupSimulation)) {
                            # Add small jittering when the profile is flat
                            # TODO: do we need this anymore? Disabled
                            # if (sum(cTimeOps - 1) == 0) {
                            #    iGroupSimulation <- jitter(iGroupSimulation, factor=1)
                            # }
                        } else {
                            iGroupSimulation <- matrix(nrow=0,
                            ↪  ncol=length(cTimeOps)*object@simulation$instance@numberReps)
                        }

                        colnames(iGroupSimulation) <- paste0("S", j, "T",
                        ↪  rep(object@simulation$instance@times,

                                                                        ↪  each=object@simulation
                                                    "R",
                                                    ↪  1:object@simulation$instance@numberReps

                        # Merge by columns (factor_1/.../factor_n)
                        groupSimulation <- if(is.null(groupSimulation))
                                                iGroupSimulation
                                            else cbind(groupSimulation, iGroupSimulation)
                    }

                    # Merge by rows (group_1|...|group_n)
                    fSimulation <- rbind(fSimulation, groupSimulation)
                }
            }

            fSimulation
        })
    }

    # Added group NoExpr to the main loop, maybe it's time to remove this?
    if (object@regulator) {
        simulatedIDs <- rownames(fSimulation)
        originalIDs <- rownames(object@data[[1]])

        remainingIDs <- setdiff(originalIDs, simulatedIDs)

        smessage("\n\t[", object@name, "] Adding remaining IDs (n°: ", length(remainingIDs),
        ↪  ") to simulated data with a 'flat' profile.")
```

```
            # Simulate a flat profile
            if (length(remainingIDs) > 0) {
                fSimulation <- rbind(fSimulation, do.call(cbind,
                 ↪  lapply(1:object@simulation$instance@numberSamples, function(s) {
                                                    smessage("\t\t- Sample ", s)
                                                    groupSimulation(object,
                                                     ↪  geneNames=remainingIDs,
                                                    timeCoefficients=rep(1,
                                                     ↪  length(object@simulation$instance@times)),
                                                     ↪  sampleNumber = s,
                                                    geneGroup = 'remaining')
                }))
            }
        }

        fSimulation <- postSimulation(object, fSimulation)

        object@.sdata$svalues <- fSimulation

        object
    }
}

.Simulator.groupSimulation <- function(object, geneNames, timeCoefficients, sampleNumber,
 ↪  geneGroup, geneBlock, ...) {

    #
    # geneBlock: DEG, NoDEG, NoExpr
    #
    # A no expr gene (timeCoefficients=0) could be not expressed on ONE sample,
    # or simply not expressed.
    #
    # At this moment both are treated the same way, and that includes the
    # possibility of appear with very low numbers (0.1)
    #
    # Maybe we should change this to force all genes from "NoExpr" group to be
    # always 0, returning a new matrix like this:
    #             noExprData <- matrix(0, ncol=object@simulation$instance@.nCols,
    #                                  nrow=length(geneNames),
    #                                  dimnames = list(geneNames, NULL))
    groupCounts <- c()

    # Make sure we only select gene names in common
    geneNames <- intersect(geneNames, rownames(object@data[[sampleNumber]]))

    # Return a zero filled matrix if pattern = no expr
    # TODO: is okay for every simulator? Check the asigned pattern when RNA-seq=noexpr (must be
     ↪  flat)
    if (length(which(timeCoefficients!=0)) > 0) {
        geneCounts <- object@data[[sampleNumber]][geneNames,]
```

```
    sloginfo(paste(object@name, "[Gene group", geneGroup, "][Sample ", sampleNumber, "]
    ↪  Genes and counts:"))
    sloginfo(geneNames)
    sloginfo(geneCounts)

    # Add 5 to prevent 0 counts
    G <- (as.matrix(geneCounts) + 5) %*% timeCoefficients
    sloginfo(paste(object@name, "Multiplying by", paste(timeCoefficients, collapse=",
    ↪  "),"equals to:"))
    sloginfo(G)

    groupCounts <- G

} else {
    smessage("\t\t(", paste(object@name, "Gene group", geneGroup, "from sample",
    ↪  sampleNumber, "has 0 time coefficients. Passing zero filled matrix to BN.)"))
    groupCounts <- matrix(0,
                          nrow=length(geneNames),

                          ↪  ncol=length(object@simulation$instance@times),#*object@simulation$instance
                          dimnames = list(geneNames, NULL))
}

# groupCounts: matrix with genes on rows and t1...tn counts on columns
# Next step: generating N replicates for every time (every column) adding noise.
simReplicates = do.call(cbind, lapply(1:ncol(groupCounts), function(columnNumber, object,
↪  groupCounts) {
    # Simulate replicates and add noise
    columnCounts <- groupCounts[, columnNumber]

    # Allowing for some noise in the NB mean
    counts.noise <- t(sapply(columnCounts, function(x) { x + c(-1,1)*object@noise*x }))
    counts.noise[which(counts.noise < 0)] <- 0

    mu.noise <- apply(counts.noise, 1, function (x) { runif(1, x[1], x[2]) })

    # Transform to CPM (rgamma tables based on CPM bins)
    mu.noise.cpm <- 10^6 * mu.noise / sum(mu.noise)

    # Replace NA for 0
    mu.noise.cpm[is.na(mu.noise.cpm)] <- 0.1
    mu.noise.cpm[mu.noise.cpm < 0.1] <- 0.1

    stdev <- sapply(mu.noise.cpm, function(x) {
        # Select row of table using mean column as intervals
        rgammaIndex <- which.max(x <= object@gammaTable$mean)

        rgamma(1, shape=object@gammaTable$shape[rgammaIndex],
        ↪  scale=object@gammaTable$scale[rgammaIndex])
    })
```

```
        nReps <- object@simulation$instance@numberReps

        temp <- t(apply(cbind(mu.noise.cpm, stdev), 1, function (x) {
            if (x[1] == x[2]^2) {
                replis <- rpois(n = nReps, lambda = x[1])
            } else {
                # replis <- rnbinom(n = object@simulation$instance@numberReps, size =
                ↪  max(x[1]^2/abs(x[2]^2 - x[1]), 0.001),  mu = max(x[1], 0.1))
                replis <- rnbinom(n = nReps, size = x[1]^2/abs(x[2]^2 - x[1]),  mu = x[1])
            }

            return(replis)
        }))

        # Transform to matrix when there's only 1 gene (it could happen...)
        if (is.null(ncol(temp))) {
            temp <- matrix(temp, ncol=object@simulation$instance@numberReps)
        }

        temp
    }, object, groupCounts))

    # Adjust CPM to depth
    simReplicates <- simReplicates*object@depth/1E6

    simReplicates
}

.Simulator.adjustDepth <- function(object, data) {
    if (object@depthAdjust) {
        smessage("Adjusting to sequencing depth ", object@depth, " on ", object@name)

        round(object@depth * data/sum(data), object@depthRound)
    } else {
        data
    }
}

.Simulator.postSimulation <- function(object, simData) {
    # Faster alternative: (abs(x)+x)/2
    simData[simData < object@minValue] <- object@minValue

    if (length(object@maxValue)) {
        simData[simData > object@maxValue] <- object@maxValue
    }

    simData <- .Simulator.adjustDepth(object, simData)

    simData
}
```

```
.Simulator.reportData <- function(object) {
    list(
        'simValues'=object@simulation$instance@simData[[class(object)]],
        'baseData'=object@data
    )
}

.Simulator.updateData <- function(object, updatePattern, geneNames, sampleNumber, changeFC,
↪   regWeight=NULL) {
    # Provide the RNAseq values to know how the expression
    # levels are for this group.
    #
    # Note: the simulation of SimRNAseq simulator needs to be
    # the first.
    exprData <- object@simulation$instance@simData$SimRNAseq[geneNames, ]

    # Transform to IDs
    geneNames <- IDfromGenes(object, geneNames)

    # Get functions/noise from the object
    patValues <- patternValues(object)

    # Original values
    # Force matrix and geneNames
    # Select only present IDs
    presentIDs <- intersect(geneNames, rownames(object@data[[sampleNumber]]))

    # Copy the initial sample of the regulator to a new matrix.
    uData <- matrix(object@data[[sampleNumber]][presentIDs,],
                    ncol=ncol(object@data[[sampleNumber]]),
                    dimnames = list(presentIDs, NULL))

    # Get new base values using the regulator function
    bFunction <- patValues$methods[[updatePattern]]

    if (! is.null(bFunction)) {
        smessage("\tUpdating data for ", object@name, " on sample ", sampleNumber,
                 " with pattern ", updatePattern)

        uData <- bFunction(uData, exprData)
    }

    if (changeFC != 1 && updatePattern != 'disabled') {
        patFC <- patValues$changeFC

        # Default values if not provided by the regulator
        if(is.null(patFC)) {

            .fcUp <-  function(m, exprData) {
                m * 1.2
            }
```

```
            .fcDown <-  function(m, exprData) {
                m * 0.8
            }

            patFC <- list(
                up=if(object@enhancer)  .fcUp else .fcDown,
                down=if(object@enhancer) .fcDown else .fcUp
            )
        }

        modMult <- if(changeFC < 1) patFC$down else patFC$up

        smessage("\tApplying modification due to FC change ", changeFC,
                 ". Changing baseValues")

        uData <- fcFunction(uData, exprData)# * modMult

    }

    # Apply function of noise. This must be at this step because
    # some simulators have more than one column, and baseValues is
    # only a vector.
    if (! is.null(patValues$noise[[updatePattern]])) {
        uData <- patValues$noise[[updatePattern]](uData)
    }

    object@data[[sampleNumber]][presentIDs,] <- uData

    object
}

.Simulator.IDfromGenes = function(object, geneNames) {
    regTable <- object@idToGene

    # matrix(NA) has length = 1
    if (length(regTable) < 2){
        geneNames
    } else {
        # Table: list with regions as names and genes as values
        rownames(regTable)[regTable %in% geneNames]
    }
}

.Simulator.IDtoGenes = function(object, idNames) {
    regTable <- object@idToGene

    if (length(regTable) < 2){
        idNames
    } else {
        # Table: list with regions as names and genes as values
```

```
        regTable[rownames(regTable) %in% idNames]
    }
}

.Simulator.patternValues <- function(object, ...) {

    patValues <- list(
        # C. induction:
        contind = NULL,
        # C. repression:
        contrep = NULL,
        # T. induction:
        tranind = NULL,
        # T. repression:
        tranrep = NULL,
        # Regulator not enabled
        disabled = NULL,
        # Not expressed:
        noexpr   = NULL
    )

    patNoise <- NULL

    patFC <- NULL

    list(
        methods=patValues,
        noise=patNoise,
        changeFC=patFC
    )
}

setMethod("initialize", signature="Simulator", .Simulator.initialize)
setMethod("simulate", signature="Simulator", .Simulator.simulate)
setMethod("patternCoefficients", signature="Simulator", .Simulator.patternCoefficients)
setMethod("postSimulation", signature="Simulator", .Simulator.postSimulation)
setMethod("updateData", signature="Simulator", .Simulator.updateData)
setMethod("IDfromGenes", signature="Simulator", .Simulator.IDfromGenes)
setMethod("IDtoGenes", signature="Simulator", .Simulator.IDtoGenes)
setMethod("patternValues", signature="Simulator", .Simulator.patternValues)
setMethod("reportData", signature="Simulator", .Simulator.reportData)
setMethod("groupSimulation", signature="Simulator", .Simulator.groupSimulation)
setMethod("configureSimulator", signature="Simulator", .Simulator.configureSimulator)
setMethod("initializeData", signature="Simulator", .Simulator.initializeData)
```

A.2.5. SimulatorRegion.R

```
.SimulatorRegion.configureSimulator = function(object, ...) {

    object <- callNextMethod(object, ...)
```

```
smessage("Configuring simulator", object@name)

.splitRegions <- function(rLocs) {
    rData <- strsplit(rLocs, '_')

    rChr <- unlist(lapply(rData, '[[', 1))

    rStart <- lapply(rData, '[[', 2)

    rEnd <- if(length(rData[[1]]) > 2)
                lapply(rData, '[[', 3)
            else
                rStart
    list(
        chr=rChr,
        start=as.numeric(rStart),
        end=as.numeric(rEnd)
    )

    # Clean locs from NA
    #         invalidIndexes <- unique(c(which(is.na(rStart)),
    #                                    which(is.na(rEnd))))
    #         if (length(invalidIndexes) > 0) {
    #             object@locs <- lapply(object@locs, function(x){ x[- invalidIndexes] })
    #         }
}

# Check rownames format
if (length(object@data) > 0) {

    rLocs <- rownames(object@data)

    # If not null, the rownames must follow the scheme:
    # chr_start_<end>
    #
    # Being end optional when there's no associated range, only the
    # location of 1 pb.
    if (! is.null(rLocs)) {
        object@locs <- .splitRegions(rLocs)
    } else {
        stop("Rownames on data are required on simulators with region names.")
    }
}

if (length(object@locs) > 0) {

    if (class(object@locs) == 'character') {
        # Filename
        if (length(object@locs) == 1) {
            object@locs <- import.bed(object@locs)
```

```
            } else {
                # Make a list from the vector
                object@locs <- .splitRegions(object@locs)
            }
        }

        if (class(object@locs) == 'list') {
            object@locs <-  GRanges(seqnames=Rle(object@locs$chr),
                                    ranges=IRanges(
                                        start=object@locs$start,
                                        end=object@locs$end),
                                    strand=Rle(strand('*')))
        }

        object@locs <- sort(object@locs)
        object@chGRanges <- split(object@locs, seqnames(object@locs))
        object@nLocs <- length(object@locs)

        smessage(object@locsName, " locations from ", length(object@chGRanges), " seqnames.")
    } else {
        smessage("No ", object@locsName, " locations provided.")

        object@chGRanges <- list(NULL)
        object@locs <- NULL
    }

    object
}

.SimulatorRegion.locGRanges <- function(object, locs) {
    GRanges(
        seqnames = Rle('1'),
        ranges = IRanges(start = locs, width = 1),
        strand = Rle(strand('*'))
    )
}

.SimulatorRegion.regionNames <- function(object, chrNumber, start, end=NULL) {
    end <- if(is.null(end)) start else end

    paste(chrNumber, start, end, sep="_")
}

.SimulatorRegion.nearestGenes <- function(object, refGR, simGR) {
    nearReg <- refGR[nearest(simGR, refGR)]
    chrRegTable <- mcols(nearReg)$EXONNAME

    # In case exon name is not provided
    if (is.null(chrRegTable)) {
        chrRegTable <- names(nearReg)
    }
```

```
    chrRegTable
}

.SimulatorRegion.getLengths <- function(object) {
    width(object@locs)
}

setMethod("configureSimulator", signature="SimulatorRegion",
 ↪   .SimulatorRegion.configureSimulator)
setMethod("regionNames", signature="SimulatorRegion", .SimulatorRegion.regionNames)
setMethod("locGRanges", signature="SimulatorRegion", .SimulatorRegion.locGRanges)
setMethod("nearestGenes", signature="SimulatorRegion", .SimulatorRegion.nearestGenes)
setMethod("getLengths", signature="SimulatorRegion", .SimulatorRegion.getLengths)
```

A.2.6. simulators/ChIP-seq.R

```
.SimChIPseq.gammaTable <- list(
    mean=c(0.5,1,3,5,10,20,1e+06),

     ↪   shape=c(0.0769368756183698,4.87463148740777,1.77614021569757,1.27135510976813,1.10859564566844,1.114

     ↪   scale=c(0.668787864915357,0.192535468225583,1.05202204909108,2.15309258180485,2.66610983570308,3.114
)

.SimChIPseq.patternValues <- function(object, ...) {
    defChipUp <- function(m, exprData) {
        #m[,] <- m[,]*0.97
        m
    }

    defChipDown <- function(m, exprData) {
        #m[,] <- m[,]*0.97
        m
    }

    defChipNoExpr <- function(m, exprData) {
        #m[,] <- m[,]*0.97
        #m*0.8
        m
    }

    patValues <- list(
        # C. induction:
        contind = defChipUp,
        # C. repression:
        contrep = defChipDown,
        # T. induction:
```

```
        tranind = defChipUp,
        # T. repression:
        tranrep = defChipDown,
        disabled = NULL,
        # Not expressed:
        noexpr   = defChipNoExpr
    )

    patNoise <- NULL

    patFC <- NULL

    list(
        methods=patValues,
        noise=patNoise,
        changeFC=patFC
    )
}

setMethod("patternValues", signature="SimChIPseq", .SimChIPseq.patternValues)
```

A.2.7. simulators/DNase-seq.R

```
.SimDNaseseq.gammaTable <- list(
    mean=c(5,10,20,30,40,50,100,1e+06),

    ↪ shape=c(2.65652390488234,2.61251029572217,2.32879608523843,2.26578855649232,2.24077370888784,2.22805

    ↪ scale=c(0.956687532320765,1.41065433018424,2.12865961108064,2.66203156475529,3.30442073165964,3.8963
)

.SimDNaseseq.patternValues <- function(object, ...) {
    defDNaseUp <- function(m, exprData) {
        #m[,] <- m[,]*0.97
        m
    }

    defDNaseDown <- function(m, exprData) {
        #m[,] <- m[,]*0.97
        m
    }

    defDNaseNoExpr <- function(m, exprData) {
        #m[,] <- m[,]*0.97
        #m*0.8
        m
    }
```

```
    patValues <- list(
        # C. induction:
        contind = defDNaseUp,
        # C. repression:
        contrep = defDNaseDown,
        # T. induction:
        tranind = defDNaseUp,
        # T. repression:
        tranrep = defDNaseDown,
        disabled = NULL,
        # Not expressed:
        noexpr   = defDNaseNoExpr
    )

    patNoise <- NULL

    patFC <- NULL

    list(
        methods=patValues,
        noise=patNoise,
        changeFC=patFC
    )
}

setMethod("patternValues", signature="SimDNaseseq", .SimDNaseseq.patternValues)
```

A.2.8. simulators/Methyl-seq.R

```
#
# Methyl-set simulator
#
# Based on simulate_WGBS.R script from WGBSScuite V 0.3 (owen.rackham@imperial.ac.uk)
#
.SimMethylseq.configureSimulator <- function(object, ...) {

    smessage("Configuring MethylSeq simulator")

    # Load required libraries
    # TODO: take care of this at NAMESPACE?
    if (! require(zoo))
        stop("Library 'zoo' is required by ", object@name, " simulator.")

    if (! require(HiddenMarkov))
        stop("Library 'HiddenMarkov' is required by ", object@name, " simulator.")

    # Correspondence between value and column number
    tReturnValues = list(
        methReads = 1,
        demethReads = 2,
```

```
    totalReads = 3,
    proportionReads = 4
)

# Unless provided, use idToGene by default
if (length(object@data) < 2) {
    # By default we don't include real data methylation, however,
    # the association list it's included and contains the identified
    # regions as rownames chr_start_end.
    object@locs <- rownames(object@idToGene)
}

object <- callNextMethod(object, ...)

# Link S4 slot names with variables used by code extracted from
# simulate_WGBS.R
#
# It's better to keep them like that so the code can be copied straight from
# future versions of WGBSSuite with minor modifications.
#

#     Arg1: multi - This indicates that you wish to perform repeats of the same settings in
 ↪  order to produce an ROC/AUC analysis.
#     Arg2: 5000 - This is the number of CpG sites to simulate
#     Arg3: 0.9203 - This is the probability of success in the methylated region. This can
 ↪  be approximated using the analysis script as explained above.
#     Arg4: 0.076 - This is the probability of success in the de-methylated region. This can
 ↪  be approximated using the analysis script as explained above.
#     Arg5: 0.1 - This is the size of the error in the methylated region. This can be
 ↪  approximated using the analysis script as explained above.
#     Arg6: 0.1 - This is the size of the error in the de-methylated region. This can be
 ↪  approximated using the analysis script as explained above.
#     Arg7: 29 - This is the average number of reads in the methylated region. This can be
 ↪  approximated using the analysis script as explained above.
#     Arg8: 29 - This is the average number of reads in the de-methylated region. This can
 ↪  be approximated using the analysis script as explained above.
#     Arg9: 3 - This is the number of replicates to be simulated.
#     Arg10: 2 - This is the number of samples to be simulated.
#     Arg11: 0.1 - This is the phase difference in the differentially methylated regions.
#     Arg12: 0.5 - This is the balance of hypo/hyper methylation.
#     Arg13:  0.19, 0.02 - This is the matrix of values that describes the exponential decay
 ↪  functions that define the distances between CpG values.
#     Arg14: /tmp/my WGBSanalysisfiles - This is the location and prefix of the files that
 ↪  will be written out.
#     Arg15: binomial - This can be either binomial or truncated and defines the
 ↪  distribution of methylated reads at each CpG.
#     Arg16: 10 - The number of repeats to run for the ROC analysis.
#
object@.sdata$WGBSsymbolTable <- list(
    # Phase difference in the differentially methylated regions between two samples
    phase_diff = c(0, object@phaseDiff),
```

```
        #error_rate_in_differentially_methylated_region = 0.1
        # File to write results
        outfile = '/tmp/', #output_path
        probs = c(1,1,0.9,0.8,0.7,0.6),
        # Transition size
        transition_size = object@transitionSize,
        # Number of repeats (NOT REPLICATES!)
        m = object@nRepeats,
        # Transitio matrix for CpG locations
        Pi_m = object@PhiMeth,
        # Transition matrix for probability distributions
        Pi_d = object@PhiDemeth,
        # Type of locations
        type_of_locations = object@typesLocation,
        # Number of CpGs: this is the number of CpG sites to simulate
        n = object@nLocs,
        # Mean number of reads in methylated region
        mean_m = object@nReadsMethReg,
        # Mean number of reads in non methylated regions
        mean_d = object@nReadsDemethReg,
        # Probability of success in  methylated region
        prob_m = object@pSuccessMethReg,
        # Probability of success in non methylated region
        prob_d = object@pSuccessDemethReg,
        # Error rate in dif. methylated region
        error_m = object@errorMethReg,
        # Error rate in non methylated region
        error_d = object@errorDemethReg,
        # Number of replicas
        # number_of_replicas = object@simulation$instance@numberReps,
        # Number of samples
        # number_of_samples = object@simulation$instance@numberSamples,
        # Balance: this is the balance of hypo/hyper methylation
        balance = object@balanceHypoHyper,
        # Rates for HMM for CpG locations: this is the matrix of values that
        # describes the exponential decay functions that define the distances
        # between CpG values.
        rates_for_HMM_for_CpG_locations = object@ratesHMMMatrix,
        # Type: this can be either binomial or truncated and defines the
        # distribution of methylated reads at each CpG.
        type = object@distType,
        returnColumn = tReturnValues[[object@returnValue]],
        chrLocs = object@chGRanges
    )

    object
}

.SimMethylseq.initializeData <- function(object) {
    ############################################################
    # Code from function generate_sim_set [simulate_WGBS.R]
```

```
    # Version 0.3 (01 June 2015)
    ############################################################
    with(object@.sdata$WGBSsymbolTable, {
        # Load the script functions with local scope.
        # Currently the script needs to modify all
        source("lib/WGBS/simulate_WGBS_functions.R")

        methData <- vector("list", object@simulation$instance@numberSamples)
        regTable <- list()

        # Link region to gene name
        simGR <- object@simulation$instance@GRanges

        # Repeat process for every chromosome
        for (i in 1:2) {
        # for (i in 1:length(chrLocs)) {
            # If chrLocs is null, simulate the locations. In this case the
            # loop will have only one cycle.
            if (is.null(chrLocs[[i]])) {
                smessage("Simulating CpG locations.")
                locs <- create_simulated_locations(n, Pi_m, rates_for_HMM_for_CpG_locations,
                 ↪  seed=object@simulation$instance@randomSeed)
                # methGR <- NULL
                methGR <- locGRanges(object, locs)
                # If the regions aren't passed, we generate the locs but only
                # on one chromosome.
#                 if (is.null(methGR)) {
#                     methGR <- locGRanges(object, locs)
#                 }
            } else {
                smessage("Simulating methylation data from CpG locations in seqname index ", i)
                methGR <- chrLocs[[i]]

                locs <- start(methGR)
                n <- length(methGR)

                # This should only happen if we are subsetting the original
                # GRanges (for testing purposes for example)
                if (n == 0) {
                    smessage("Number of CpG locations = 0. Skipping.")
                    next
                }
            }

            #simulate the state transition based on the location of the CpGs
            a <-
             ↪  simulate_state_transition((n*m),c(0.01,0.99,0.08,0.99),locs,0.5,transition_size)

            #extract the postions of the blocks
            state_blocks <- find_the_blocks(a)
```

```
#set the percentage of DM in each block type
percs_for_diff <- c(0,0,0,0.5)

#update the blocks to be differentially methylated
diff_methed <- make_differential(state_blocks,percs_for_diff,a)

#create the simulated reads methylated/unmethylated at each CpG, at the moment this
↪  is hard coded to be 3 replicates of each type
#The phase diff param control how different the methylation is in the differentially
↪  methylated regions.
errors <-
    list(error_m,((error_d + error_m) / 2),error_d,((error_d + error_m) / 2))
means <-
    list(mean_m,((mean_d + mean_m) / 2),mean_d,((mean_d + mean_m) / 2))


# Generate data for every sample and every rep
repData <- list()
repDataNames <- regionNames(object, chrNumber=i, start=locs)


#sIRanges <- reduce(IRanges(start=locs, width=250))


# NOTE: if we are providing data for the GRanges object of simulation
# and methylation, they must contain information about the chr
# or GenomicRanges will not return results.
#rownamesRepData <- mcols(simGR[nearest(methGR, simGR)])$EXONNAME

# TEMP: first (prob_m_nonmod), rest (prob_m_mod)
diffPhase <- c(0, rep(0.1, object@simulation$instance@numberSamples - 1))
probs <- vector("list", object@simulation$instance@numberSamples)

prob_d_factor <- lapply(1:object@simulation$instance@numberSamples, function(f) {
    list(
        d=hypo_hyper_diffs(prob_d, diffPhase[f], diff_methed, balance),
        m=hypo_hyper_diffs(prob_m, diffPhase[f], diff_methed, balance)
    )
})

# Get one "non-mod" data
# prob_nonmod <- prob_d_factor[[which(diffPhase == 0)[1]]]
prob_nonmod <- prob_d_factor[[which.max(diffPhase == 0)]]

for (f in 1:object@simulation$instance@numberSamples) {
    # Diff. sites for every factor
    prob_f <- prob_d_factor[[f]]

    # Non-mod
    if (diffPhase[f] == 0) {
```

```
            prob_f <- list(prob_f$m, ((prob_nonmod$m + prob_nonmod$d) / 2), prob_f$d,
             ↪  ((prob_f$m + prob_f$d) / 2))
        } else {
            prob_f <- list(prob_f$m, ((prob_f$m + prob_f$d) / 2), prob_f$d, ((prob_f$m +
             ↪  prob_f$d) / 2))
        }

        probs[[f]] <- prob_f

        # New matrix for every sample
        repData <- matrix(data=0, nrow=object@simulation$instance@numberReps, ncol=n)

        for (r in 1:object@simulation$instance@numberReps) {
            if (object@distType == 'binomial') {
                sRepData <-
                    generate_replicat_methyl_bin_data(a, probs, means, 0, errors, locs,
                     ↪  f, output_path)
            } else if (object@distType == 'truncated') {
                sRepData <-
                    generate_replicat_methyl_truncated_nbin_data(a, probs, means, 0,
                     ↪  errors, locs, f, 20, output_path)
            } else{
                sRepData <-
                    generate_replicat_methyl_nbin_data_model_3(a, probs, means, 0,
                     ↪  errors, locs, f, 30)
            }
            # Columns:
            #         V1: location in base pairs
            #         V2: differentially methylated flag
            #
            # Blocks of 4 columns for each replica as follows:
            #
            #         Vn: Number of methylated reads
            #         Vn+1: Number of de-methylated reads
            #         Vn+2: Total number of reads
            #         Vn+3: Proportion of methylated vs de-methylated reads
            sRepData <- sRepData[returnColumn,]

            repData[r,] <- sRepData
        }

        repData <- t(repData)

        # Add rows to the factor data
        rownames(repData) <- repDataNames

        # Repeating for every chromosome
        methData[[f]] <- rbind(methData[[f]], repData)
    }

    if (length(object@idToGene) == 0) {
```

```
                # Make region-gene table
                chrRegTable <- nearestGenes(object, simGR, methGR)
                names(chrRegTable) <- repDataNames

                regTable <- c(regTable, chrRegTable)
            }
        }

        list(
            values=methData,
            IDtable=if(length(object@idToGene) == 0) regTable else object@idToGene
        )

    })
}

.SimMethylseq.patternCoefficients <- function(object) {
    callNextMethod()
}

.SimMethylseq.groupSimulation <- function(object, geneNames, timeCoefficients, sampleNumber,
↪   geneGroup, ...) {

    with(object@.sdata$WGBSsymbolTable, {
        groupCounts <- NULL #c()

        if (length(geneNames) > 0) {

            dCont <- object@data[[sampleNumber]]

            # Select only present regions on data
            commonRegions <- intersect(geneNames, rownames(dCont))

            # Every column of the data is a replica, so we need to transform
            # that into a time series and then reorder the columns.
            groupCounts <- do.call(cbind, lapply(1:ncol(dCont), function(rep) {
                as.matrix(dCont[commonRegions, rep]) %*% timeCoefficients
            }))

            # Reorder columns like:
            # t0.r1 t0.r2 t0.r3  ... tn.r1 tn.r2 tn.r3
            lStep <- ncol(groupCounts) %/% object@simulation$instance@numberReps

            columnReorder <- unlist(lapply(1:lStep, seq, by=lStep,
            ↪   length.out=object@simulation$instance@numberReps))

            groupCounts <- matrix(groupCounts[, columnReorder],
                                  ncol=length(columnReorder),
                                  dimnames = list(rownames(groupCounts), NULL))
        }
```

```
        groupCounts
    })
}

.SimMethylseq.postSimulation <- function(object, simData) {

    simData <- callNextMethod()

    # Add beta-values
    # methLevel & rowData comes from BiSeq package?
    betaThreshold <- object@betaThreshold
    #beta = pmin(pmax(methLevel(smooth.clust.lim), betaThreshold), 1 - betaThreshold)
    beta = pmin(pmax(simData, betaThreshold), 1 - betaThreshold)
    colnames(beta) <- colnames(simData)
    #paste(seqnames(smooth.clust.lim@rowData),end(smooth.clust.lim@rowData), sep=":")
    rownames(beta) <- rownames(simData)
    #         - remove rows with zero variance:
    beta <- beta[rowVars(beta, na.rm=T)!=0,]
    M <- log2(beta/(1-beta))

    # Return M values
    list(
        raw=simData,
        beta=beta,
        M=M
    )
}

.SimMethylseq.reportData <- function(object) {
    list(
        'simValues'=object@simulation$instance@simData$SimMethylseq$M,
        'baseData'=object@data
    )
}

.SimMethylseq.patternValues <- function(object, ...) {

    # For methylation, we assume a value of ~0.9 is associated with a not expressed gene,
    # and a value of ~0.1 with a expressed one.
    #
    # In all profile patterns we return the value 0.9 that will be adjusted
    # to ~0 using the time coefficients.
    #
    # TODO: change noise function
    methValue <- 0.9
    nomethValue <- 0.9

    defMethvalue <- function(m, exprData) {
        m[,] <- methValue;
        m
    }
```

```
    defNoMethvalue <- function(m, exprData) {
        m[,] <- nomethValue;
        m
    }

    defNoise <- jitter

    patValues <- list(
        # C. induction: not expressed at t=0, methylated
        contind = defMethvalue,
        # C. repression: expressed at t=0, not methylated
        contrep = defNoMethvalue,
        # T. induction: not expressed at t=0, methylated
        tranind = defMethvalue,
        # T. repression: expressed at t=0, not methylated
        tranrep = defNoMethvalue,

        # TODO: decide what to do when methylation is not part of the
        # regulatory program.
        disabled = NULL,
        # Not expressed: methylated
        noexpr   = defMethvalue
    )

    patNoise <- list(
        # C. induction: not expressed at t=0, methylated
        contind = defNoise,
        # C. repression: expressed at t=0, not methylated
        contrep = defNoise,
        # T. induction: not expressed at t=0, methylated
        tranind = defNoise,
        # T. repression: expressed at t=0, not methylated
        tranrep = defNoise,
        disabled = NULL,
        # Not expressed: methylated
        noexpr   = defNoise
    )

    # Functions:
    # - up: when modeling +1.5 FC
    # - down: for -1.5 FC
    patFC <- NULL

    list(
        methods=patValues,
        noise=patNoise,
        changeFC=patFC
    )
}
```

```
.SimMethylseq.getLengths <- function(object) {
    warning("Ignoring length bias on Methyl-seq")
    NULL
}

setMethod("configureSimulator", signature="SimMethylseq", .SimMethylseq.configureSimulator)
setMethod("initializeData", signature="SimMethylseq", .SimMethylseq.initializeData)
setMethod("groupSimulation", signature="SimMethylseq", .SimMethylseq.groupSimulation)
setMethod("patternCoefficients", signature="SimMethylseq", .SimMethylseq.patternCoefficients)
setMethod("postSimulation", signature="SimMethylseq", .SimMethylseq.postSimulation)
setMethod("reportData", signature="SimMethylseq", .SimMethylseq.reportData)
setMethod("patternValues", signature="SimMethylseq", .SimMethylseq.patternValues)
setMethod("getLengths", signature="SimMethylseq", .SimMethylseq.getLengths)
```

A.2.9. simulators/RNA-seq.R

```
.SimRNAseq.gammaTable <- list(
    mean=c(1,5,10,15,25,50,75,100,500,5000,15000,1e+06),

    shape=c(5.12069695369583,2.31111819433427,2.79973960715549,2.85912994880808,
            2.92544440726716,2.71844286130629,1.79038505963967,2.93199534788786,
            1.58803070351186,1.53988847271216,1.86332612328476,0.580739807001836),

    scale=c(0.0881209571080616,0.804836647784225,1.64942276542406,2.87435701954641,
            5.21636953504161,8.65961524646501,15.7834953443805,13.6005097645718,
            53.8061030463463,301.806512019322,622.399272725056,16859.692892429)
)

.SimRNAseq.configureSimulator <- function(object, ...) {
    object <- callNextMethod(object, ...)
    object
}

.SimRNAseq.initializeData <- function(object) {
    #
    #     potencia <- 0.5
    #
    #     # TODO: change upper limit?
    #     x <- 1:round(object@depth/1000, 0)
    #     myprob <- x^(-potencia)
    #
    #     # Create values for expressed genes
    #     baseCounts <- sample(x, size = object@nGenes, replace = TRUE, prob = myprob)
    #     names(baseCounts) <- sampleGeneNames(object@simulation$instance, size =
    ↪   length(baseCounts), type = c('DEG', 'NoDEG'))
    #
    # Set FC
    baseCounts <- object@data
    data <- vector("list", object@simulation$instance@numberSamples)
```

```
# Modify values based on FC (only for DEG genes)
data <- with(object@simulation$instance@simSettings$DEG, {

    forma1 <- 1.5
    forma2 <- object@beta
    # to avoid multiply foldchange by 0 or very low counts
    k1 <- k2 <- 5

    # For every condition
    for (f in 1:nrow(fcMatrix)) {

        fData <- baseCounts

        # Change object@data (original counts)
        #
        # For every gene group
        #
        # Skip changing FC if we are considering the simulator a regulator
        # (child classes)
        if (! object@regulator) {
            for (c in 1:ncol(fcMatrix)) {
                FCvalue <- fcMatrix[f, c]

                if (FCvalue != 1) {

                    smessage("[", object@name, "] Adjusting FC ", FCvalue, " for gene group
                    ↪  ", c, " on condition ", f)

                    # Retrieve gene names for every regulatory program associated
                    # to this particular gene group.
                    geneNames <- unlist(geneGroups[geneClassToRP[[c]]])

                    sloginfo(paste("[", object@name, "] Values pre-adjusting FC to ",
                    ↪  FCvalue))
                    sloginfo(fData[geneNames,])

                    # If is up-regulation, modify the gene values directly, if it's down
                    ↪  regulation, increase the FC of
                    # the remaining genes. Finally we will round to desired depth.
                    if (FCvalue < 0) {
                        geneNames <- setdiff(rownames(fData), geneNames)
                    }

                    dataChange <- rbeta(length(geneNames), shape1 = forma1, shape2 =
                    ↪  forma2)*100 + 1.5 # FCvalue

                    fData[geneNames,] <- (fData[geneNames,] + k1) * dataChange

                    loginfo("[", object@name, "] Values post-adjusting FC to ", FCvalue)
                    sloginfo(fData[geneNames,])
                }
```

```
                }
            }

            fData <- .Simulator.adjustDepth(object, fData)

            data[[f]] <- as.matrix(fData)
        }

        data
    })

    list(
        values=data,
        IDtable=matrix(NA)
    )
}

.SimRNAseq.getLengths <- function(object) {
    object@geneLength
}

.SimRNAseq.patternCoefficients <- function(object) {
    callNextMethod()
}

.SimRNAseq.postSimulation <- function(object, simData) {
    simData <- callNextMethod()

    simData <- round(simData)

    simData
}

setMethod("configureSimulator", signature="SimRNAseq", .SimRNAseq.configureSimulator)
setMethod("initializeData", signature="SimRNAseq", .SimRNAseq.initializeData)
setMethod("patternCoefficients", signature="SimRNAseq", .SimRNAseq.patternCoefficients)
setMethod("postSimulation", signature="SimRNAseq", .SimRNAseq.postSimulation)
setMethod("getLengths", signature="SimRNAseq", .SimRNAseq.getLengths)
```

A.2.10. simulators/miRNA-seq.R

```
.SimMiRNAseq.gammaTable <- list(
    mean=c(0.5,1,2,30,1e+06),
    ↪ shape=c(0.0912798260161212,2.57146261146076,2.73546025404724,1.43199997857507,0.238274308304151),
    ↪ scale=c(0.170643135401093,0.116652504719918,0.207873707983415,1.74063989947033,5842.72780240922)
)
```

```
.SimMiRNAseq.patternValues <- function(object, ...) {

    # miRNA inhibits expression
    #
    # When the pattern is induction, we *in theory* must
    # decrease miRNA values... but that is taken into account
    # by groupSimulation on every time.
    defMiRNAUp <- function(m, exprData) {
        #m[,] <- m[,]*0.97
        m
    }

    defMiRNADown <- function(m, exprData) {
        #m[,] <- m[,]*0.97
        m
    }

    defMiRNANoExpr <- function(m, exprData) {
        #m[,] <- m[,]*0.97
        #m*1.2
        m
    }

    patValues <- list(
        # C. induction:
        contind = defMiRNAUp,
        # C. repression:
        contrep = defMiRNADown,
        # T. induction:
        tranind = defMiRNAUp,
        # T. repression:
        tranrep = defMiRNADown,
        disabled = NULL,
        # Not expressed:
        noexpr   = defMiRNANoExpr
    )

    patNoise <- NULL

    patFC <- NULL

    list(
        methods=patValues,
        noise=patNoise,
        changeFC=patFC
    )
}

.SimMiRNAseq.getLengths <- function(object) {
    warning("Ignoring length bias on miRNA-seq")
```

```
    NULL
}

setMethod("patternValues", signature="SimMiRNAseq", .SimMiRNAseq.patternValues)
setMethod("getLengths", signature="SimMiRNAseq", .SimMiRNAseq.getLengths)
```

B

Análisis de datos

B.1. Script de generación de gráficos

```
library(ggplot2)
library(reshape2)
library(grid)
library(gridExtra)

# TEMP FIX UNTIL THE DEV VERSION OF ggplot REACHES CRAN SERVERS
# BYPASS CLASS CHECK
ggsave <- ggplot2::ggsave; body(ggsave) <- body(ggplot2::ggsave)[-2];

.OMICS <- c("RNA_seq", "DNase_seq", "ChIP_seq", "miRNA_seq")
# Grupos obtenidos tras el análisis de la gráfica sd.
# Contiene número de columnas
.OMICScache <- list()
.OMICScfg <- list(
    "RNA_seq"=list(
        samples=c('Ctrl', 'IK'),
        nReps=3,
        times=c(0, 2, 6, 12, 18, 24),
        normLog=FALSE,
        sdGroups=list(
            'IK24h'=34:36,
            'IK0h'=19:21,
            'Ctrl12h'=10:12,
            'Ctrl6h'=7:9
        )
    ),
    "DNase_seq"=list(
        samples=c('Ctrl', 'IK'),
        nReps=3,
        times=c(0, 2, 6, 12, 18, 24),
        normLog=FALSE,
        sdGroups=list(
            'IK2h'=22:24,
            'Ctrl18h'=13:15,
            'Ctrl6h'=7:9
```

```
        )
    ),
    "ChIP_seq"=list(
        samples=c('Ctrl', 'IK'),
        nReps=2,
        times=c(24),#, 2, 6, 12, 18, 24),
        normLog=TRUE,
        sdGroups=list(
            'IK24h'=3:4,
            'Ctrl24h'=1:2
        )
    ),
    "miRNA_seq"=list(
        samples=c('Ctrl', 'IK'),
        nReps=3,
        times=c(0, 2, 6, 12, 18, 24),
        normLog=FALSE,
        sdGroups=list(
            'IK18h'=31:33,
            'IK6h'=25:27,
            'IK2h'=22:24,
            'Ctrl12h'=10:12
        )
    )
)

multiplotOmic <- function(omics, title, outFilename, preFunction, drawFunction, cache = NULL,
↪  normalizedData = FALSE) {
    sGrobs <- lapply(omics, .plotOmic, preFunction=preFunction, drawFunction=drawFunction,
    ↪  cache=cache, normalizedData=normalizedData)

    mPlots <- marrangeGrob(grobs=sGrobs[!sapply(sGrobs, is.null)], nrow = 1, ncol = 1, top =
    ↪  title)

    ggsave(outFilename, mPlots)
}

.cfg <- function(omic) {
    .OMICScfg[[omic]]
}

.caching <- function(operation, omic, type, data=NULL) {
    if (operation == 'get') {
        .OMICScache[[omic]][[type]]
    } else {
        .OMICScache[[omic]][[type]] <- data

        assign('.OMICScache', .OMICScache, envir = .GlobalEnv)
    }
}
```

```
.plotOmic <- function(name, preFunction, drawFunction, cache, normalizedData) {

    # Variable name: counts.OmicName{norm} (with underscore)
    if (normalizedData) {
        dataFile <- paste("counts_", name, "_norm.RData", sep="")
        valueName <- paste('counts', name, 'norm', sep=".")
        ggTitle <- paste(gsub('_', '-', name), '')
    } else {
        dataFile <- paste("counts_", name, ".RData", sep="")
        valueName <- paste('counts', name, sep=".")
        ggTitle <- gsub('_', '-', name)
    }

    # Load counts_OmicName.RData
    message("\nLoading ", dataFile)
    load(dataFile)

    dPlot <- NULL
    valueData <- get(valueName)

    message("Loaded ", valueName, " (Rows: ", nrow(valueData), ")")

    oData <- preFunction(valueData, omic=name, normalizedData=normalizedData)

    if (! is.null(oData)) {

        message("Prefunction on ", valueName, " (Rows: ", nrow(as.matrix(oData)), ")")

        if (! is.null(cache)) {
            .caching('set', name, cache, oData)
        }

        if (! is.data.frame(oData)) {
            oData <- melt(data.frame(oData))
        }

        dPlot <- drawFunction(oData, ggTitle)
    }

    dPlot
}

# Drawing functions
.drawDensity <- function(data, title) {
    ggplot(data, aes(x=value)) +
        geom_density(aes(group=variable, colour=variable)) +
        ggtitle(title) +
        guides(colour=guide_legend(ncol=1)) +
        xlab("Desviación típica") +
        ylab("Densidad") +
        theme(legend.key.size=unit(10, "pt")) +
```

```
        theme(legend.title=element_blank())
}

.drawLines <- function(data, title) {
    ggplot(data, aes(x=xvalue, y=yvalue)) +
        geom_point() +
        ggtitle(title) +
        xlab("Avg. expression (rowMeans)") +
        ylab("SD reps") +
        facet_grid(variable ~ .)
}

.drawLinesNoFacet <- function(data, title) {
    ggplot(data, aes(x=xvalue, y=yvalue)) +
        geom_point() +
        ggtitle(title) +
        xlab("Expresión media (CPM)") +
        ylab("Desviación típica")
}


# Distribution of counts
.logPF <- function(d, ...) {
    # Matrix: force melt
    as.matrix(log(d+1))
}

# Skip:
multiplotOmic(.OMICS, title="Reads distribution (log)", "graphics/counts_distribution.pdf",
 ↪  preFunction=.logPF, drawFunction=.drawDensity)
multiplotOmic(.OMICS, title="Reads distribution (Normalized)",
 ↪  "graphics/counts_distribution_norm.pdf", preFunction=.logPF, drawFunction=.drawDensity,
 ↪  normalizedData=TRUE)

# SD between reps
.sdPF <- function(d, omic, normalizedData, ...) {
    cfgOmic <- .cfg(omic)

    nReps <- cfgOmic$nReps
    nTimes <- length(cfgOmic$times)

    nPos <- seq(from=1, to=nReps*nTimes*2, by=nReps)

    sdColnames <- paste(rep(cfgOmic$samples, each=nTimes), cfgOmic$times, "h", sep="")

    # sd for groups of cols
    sdCols <- do.call(cbind, lapply(nPos, function(nCol) {
        start <- nCol
        end <- nCol + nReps - 1
```

```
        message("SD REP ", omic, " columns ", start, ":", end, " [",
         ↪  sdColnames[ceiling(start/nReps)] , "]")

        apply(d[,start:end], 1, sd)
    }))

    colnames(sdCols) <- sdColnames

    if (normalizedData && cfgOmic$normLog)
        rValues <- log(na.omit(sdCols) + 1)
    else
        rValues <- na.omit(sdCols) + 1

    # Matrix: force melt
    as.matrix(rValues)
}

# Skip:
multiplotOmic(.OMICS, title="Rep. variability", "graphics/sdrep.pdf", preFunction=.sdPF,
 ↪  drawFunction=.drawDensity, cache='sdrep')
multiplotOmic(.OMICS, title="Variabilidad entre réplicas", "graphics/sdrep_norm.pdf",
 ↪  preFunction=.sdPF, drawFunction=.drawDensity, cache='sdrepNorm', normalizedData=TRUE)

# SD vs mean expression
.sdExp <- function(d, omic, normalizeData, ...) {
    cfgOmic <- .cfg(omic)

    if (! exists('sdGroups', where=cfgOmic)) {
        ggData <- NULL
    } else {
        sdDens <- .caching('get', omic, 'sdrepNorm')

        ggData <- data.frame(avgExp=numeric(), sdGroup=numeric(), group=character())

        for (group in names(cfgOmic$sdGroups)) {
            groupData <- na.omit(d[, cfgOmic$sdGroups[[group]]])

            avgExp <- rowMeans(groupData)
            sdGroup <- sdDens[, group]

            ggData <- rbind(ggData, data.frame(xvalue=avgExp,
                                               yvalue=sdGroup,
                                               variable=rep(group, length(sdGroup))))
        }
    }

    # Dataframe: no melt
    ggData
}
```

```
multiplotOmic(.OMICS, title="Rep. variability vs Avg. expression", "graphics/sdrep_avgexp.pdf",
 ↪  preFunction = .sdExp, drawFunction = .drawLines, cache='SDvsAvgExp', normalizedData = TRUE)

multiplotOmic(.OMICS, title="Rep. variability vs Avg. expression (CPM)",
 ↪  "graphics/sdrep_avgexp_cpm.pdf", preFunction = function(d, omic, normalizeData, ...) {
    d <- d/sum(d)*1E6
    .sdExp(d, omic, normalizeData, ...)
}, drawFunction = .drawLinesNoFacet, cache='SDvsAvgExpCPM', normalizedData = TRUE)


# # Save plotting data to file
# plotData <- .OMICscache
# save(plotData, file="plotData.RData")
```