Universidad Autónoma de Madrid
Escuela politécnica superior

Doble Grado en Ingeniería Informática y Matemáticas

Trabajo Fin de Grado

SIMULACIÓN DE MODELOS EVOLUTIVOS EN TUMORES CON R Y C++

Autor: Alberto Parramón Castillo
Tutor: Ramón Díaz Uriarte
Ponente: Iván González Martínez

Junio 2016

SIMULACIÓN DE MODELOS EVOLUTIVOS EN TUMORES CON R Y C++

Autor: Alberto Parramón Castillo
Tutor: Ramón Díaz Uriarte
Ponente: Iván González Martínez

Dpto. de Bioquímica
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio 2016

Resumen

El cáncer se ha convertido en una enfermedad muy temida por la población mundial. Es por ello por lo que el interés del mundo científico en buscar una cura ha aumentado en los últimos años.

En este trabajo se va a desarrollar un algoritmo en R y C++, que, basado en un modelo matemático en tiempo discreto, simule el crecimiento de un tumor en un individuo a partir de unos datos y parámetros previos que podrán variarse para probar situaciones diferentes. El modelo será implementado utilizando la librería `Rcpp` que permite conectar código en R con código en C++.

Este modelo será integrado en un paquete en R (*OncoSimulR*) ya existente y que contiene un modelo en tiempo continuo y código relacionado con este problema. Previamente se realizará un estudio de los modelos existentes, y de cómo estos pueden simular este proceso. Se estudiará la nomenclatura utilizada en este campo y se analizarán las diferencias entre unos modelos y otros tanto a nivel computacional como a nivel biológico.

Una vez diseñado el algoritmo se comprobará su correcto funcionamiento. Para ello, se diseñará un conjunto de tests que probarán la respuesta del algoritmo a partir de diferentes valores de entrada. A continuación, se utilizará para simular un escenario clásico (el propuesto por Ochs y Desai) y se analizarán los resultados obtenidos.

Para finalizar, se expondrán las conclusiones que se pueden extraer de este trabajo, así como las líneas de trabajo futuro.

Palabras Clave

cáncer, modelo, *OncosimulR*, tiempo discreto, mutación, evolución

Abstract

Cancer has become a very dreaded disease by the world population. That is why the scientific community shows increasing interest in searching for a cure in recent years.

This paper focuses on the development of an algorithm in R and C++ which ?based on a mathematical discrete time model? simulates the growth of a tumor in an individual from previous data and parameters that can be varied to test different situations. The model will be implemented using the `Rcpp` library that allows us to connect R with C++ code.

This model will be integrated into an existing package in R (*OncoSimulR*) which contains a continuous time model and code related to this problem. Previously, existing models and how they can simulate this process will be observed. The nomenclature used in this field will be studied and the differences between various models will be analyzed both computationally and biologically.

Once the algorithm is designed, its correct performance will be examined. For this purpose, a set of tests that will check the response of the algorithm from different input values will be designed. Then, the algorithm will be used to simulate a classic scenario (the one proposed by Ochs and Desai) and the results will be analyzed.

Lastly, the conclusions drawn from this paper and futures strands of work will be exhibited.

Key words

cancer, model, *OncosimulR*, discrete time, mutation, evolution

Índice general

Índice de Figuras

Índice de Tablas

1

Introducción

En este capítulo se detallan las necesidades que dan lugar a la realización de este Trabajo de Fin de Grado, planteando los objetivos que habrán de cumplirse para llevarlo a cabo. Al final del capítulo se explicará la estructura de este documento.

1.1. Motivación del proyecto

El cáncer se encuentra cada vez más presente en nuestro día a día. El temor a padecerlo es un hecho que la mayoría de las personas experimenta en algún momento de su vida, ya sea por la proximidad con familiares o amigos que lo han tenido, o por alcanzar edades en las que la frecuencia de estas enfermedades es cada vez mayor.

Pese a los numerosos avances que se han producido en los últimos años en su diagnóstico, tratamiento y prevención, el término cáncer esta intimamente asociado a la muerte por la mayoría de la población. El número de diagnósticos que se producen cada año aumenta, lo que indica que cada vez es más probable que cualquiera pueda llegar a sufrir algún tipo de cáncer a lo largo de su vida.

Según lo publicado por la Sociedad Española de Oncología Médica, en 2012 se diagnosticaron 14 millones de casos nuevos y se espera que aumente en un 70 % en las próximas décadas. Además se estima que cerca del 39,6 % de mujeres y hombres recibirán un diagnóstico de cáncer en algún momento de sus vidas. También se aprecian diferencias entre hombres y mujeres, mientras que en el caso de las mujeres se estima que padecerán cáncer una de cada tres, en el caso de los hombres se estima que serán uno de cada dos.

Además, en la actualidad, muere más del doble de personas a causa del cáncer de las que mueren por SIDA, paludismo (malaria) y tuberculosis juntas. La Organización Mundial de la Salud reporta que sin una acción inmediata, el número global de muertes por cáncer aumentará en aproximadamente el 80 % para el año 2030.

El cáncer es un conjunto de cientos de enfermedades diferentes. Depende de muchos factores, como el órgano en el que surja, el tipo de células o la malignidad. Aquí se pueden ver un par de gráficos en los que se muestran los cánceres más comunes así como los más mortales:

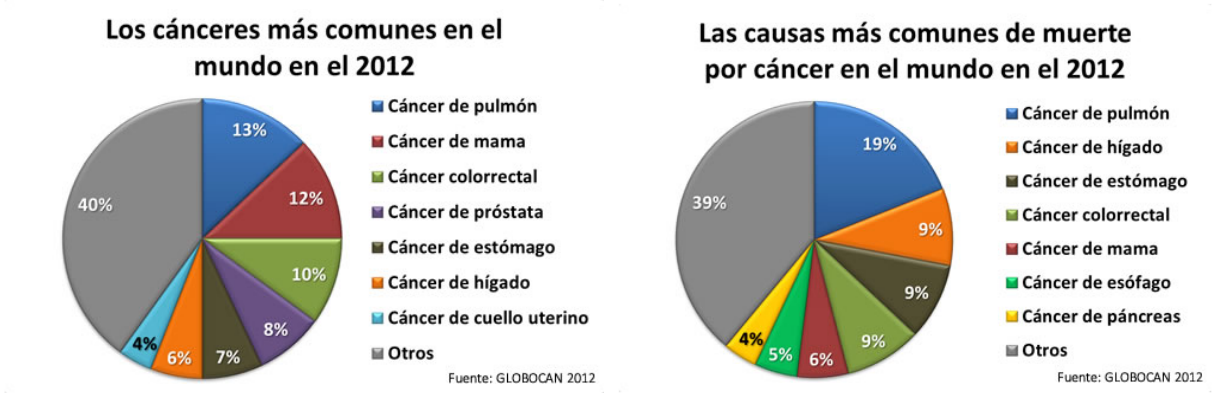

Figura 1.1: Cánceres más comunes y más mortales en el mundo

Pese a que sean diferentes unos de otros, todos tienen algo en común: son debidos al crecimiento anormal de un conjunto de células. El cáncer surge porque en algún momento se produce en la división celular una mutación que provoca que las células se dividan de forma descontrolada y el cuerpo no sea capaz de controlarlo, formándose así una masa tumoral.

Lo que no se sabe aún es qué es lo que provoca cáncer. Se sabe que hay factores de riesgo (como la radiación o la dieta) y que algunos son hereditarios.

Por todo esto, y con el objetivo de aportar su granito de arena en la lucha contra el cáncer, surgen los modelos matemáticos sobre la replicación y mutación celular, es decir, sobre el crecimiento tumoral.

La modelización matemática se ha vuelto cada vez más abundante en la investigación sobre el cáncer. La complejidad del cáncer se adapta bien a estos modelos, que ayudan a dilucidar los mecanismos y a proporcionar predicciones cuantitativas sobre la expansión tumoral. Estos modelos intentan obtener respuestas a la causa de la iniciación del tumor, su progresión y su metástasis. Así como a la respuesta a tratamientos y la resistencia de las mutaciones a los mismos.

Los modelos matemáticos no solo permiten complementar los estudios experimentales y clínicos sino también desafiar paradigmas actuales, redefinir nuestra comprensión de los mecanismos tumorales y dar forma a las futuras investigaciones en la biología del cáncer.

El poder del modelado matemático reside en su capacidad para revelar conocimientos previamente desconocidos o contrarios a la intuición, que podrían haber pasado desapercibidos por un enfoque cualitativo más propio de la biología.

1.2. Objetivos y enfoque

Para lograr los objetivos que se proponen alcanzar con este trabajo, se tendrá que realizar un estudio previo de la situación y de las tecnologías que se utilizarán. Este estudio nos permitirá:

- Tener una visión general de lo que es un modelo matemático.
- Entender el papel de los modelos matemáticos en la ayuda para la investigación contra el cáncer.
- Comprender la utilidad de implementar algoritmos informáticos basados en estos modelos.

- Aprender la nomenclatura utilizada en este campo, tanto para definir modelos como para referirse a comportamientos biológicos.
- Desarrollar una visión panorámica de los diferentes tipos de modelos y algoritmos que hay, así como las diferencias entre ellos y sus ventajas y desventajas.

Como se aprecia, estas capacidades están menos relacionadas con la Informática, y más con la Biología y las Matemáticas. Esto es porque se considera importante conocer la base teórica del problema, y al ser un problema relacionado con la biología y en el que las matemáticas juegan un papel fundamental, es necesario introducir todos los conceptos necesarios de estos dos ámbitos.

Una vez conocidos estos, se procederá a desarrollar un nuevo modelo e incorporarlo al paquete de R *OncoSimulR* ([1]una librería de R con código relacionado con este problema, de la que hablaremos más adelante). Por ello, los objetivos del trabajo pueden resumirse en:

- Entender la funcionalidad actual del paquete *OncoSimulR*.
- Desarrollar un modelo matemático en tiempo discreto e implementar un algoritmo informático que simule la replicación, mutación y proliferación de una población de células.
- Entender los fundamentos matemáticos en los que se rige el modelo, así como el algoritmo computacional que lo pone en práctica.
- Aprender la estructura y creación de un paquete de R.
- Comprender la utilidad de la mezcla de los lenguajes R y C++ gracias a la librería Rcpp.
- Integrar el modelo en el paquete *OncoSimulR*.
- Explicar cómo un usuario puede utilizar dicho modelo con la funcionalidad que ofrece el resto del paquete.

Como se puede apreciar, estos objetivos están más relacionados con la informática y los métodos de implementación.

A lo largo de la lectura del documento se mezclarán conceptos biológicos con conceptos informáticos y matemáticos.

1.3. Organización de la memoria

Dado el amplio ámbito de contenidos que se van a tocar, es importante estructurar el documento de tal forma que se vayan aprendiendo los objetivos marcados en orden y de manera incremental.

En la sección siguiente se van a tratar los objetivos del primer bloque, es decir, se va a aprender lo que es un modelo matemático, a comprender los argumentos biológicos y matemáticos que hay detrás de ellos y se verán varios tipos distintos de modelos. Por último se revisará el contenido del paquete *OncoSimulR* sobre el que se implementará el nuevo modelo.

A continuación se entrará en el segundo bloque de objetivos. Se desarrollará un modelo matemático en tiempo discreto y se explicará cómo se ha llevado a cabo la implementación en R y en C++, así como la incorporación del mismo al paquete *OncoSimulR*.

Tras este paso, se realizarán tests para comprobar su correcto funcionamiento y se probará con un par de escenarios clásicos.

Por último se incorporará una última sección con las conclusiones del trabajo y con una breve exposición de las líneas de trabajo futuro que surgen en este ámbito a partir del trabajo realizado.

2

Estado del arte

Durante este capítulo se va a introducir el concepto de modelo matemático y a definir su papel en la investigación sobre el cáncer. A continuación se repasarán los diferentes tipos de algoritmos informáticos y modelos en los que se basan, y se describirán con detalle algunos de ellos. Finalmente se hablará del paquete *OncoSimulR* sobre el cual se implementará, en el siguiente capítulo, el nuevo algoritmo.

2.1. Modelos matemáticos y cáncer

La aplicación de las matemáticas al cáncer es un tema que en los últimos años se ha tratado mucho. Las matemáticos han pasado a formar una parte importante en esta investigación, sobre todo a partir de la modelización. Un proceso de modelización consiste en lo siguiente:

- Identificación de un problema o situación compleja que necesita ser simulada, optimizada o controlada.
- Elección del tipo de modelo que se va a usar. Para ello hay que saber qué tipo de respuesta se quiere obtener y cuáles son los factores de los que depende el modelo.
- Formalización del modelo, es decir, detallar cuáles serán los datos de entrada, así como las fórmulas matemáticas en las que se basará. En esta fase posiblemente se introduzcan las simplificaciones necesarias para que el problema matemático sea tratable computacionalmente, por ejemplo, mediante la implementación de un algoritmo.
- Análisis el modelo a partir de los resultados y obtener información de los mismos comparándolos con situaciones o datos reales.

En este caso, tratándose de un modelo que simule el crecimiento tumoral, este debe tener el cuenta el tamaño en células del tumor. El número de células de cáncer en un tumor es difícil de estimar debido a que está sometido a continuos cambios en el tiempo. Las células del tumor pueden morir, permanecer estables o proliferar.

Cuántas y cómo de a menudo las células proliferan, mueren o mutan es lo que se encarga de decidir el modelo que se elija a partir de los parámetros que se le pasen.

Por tanto, como primera aproximación, se puede decir que un modelo se encarga de simular lo que ocurre con un conjunto de células en un tiempo dado. El modelo provocará la reproducción, mutación o muerte de esas células de acuerdo a unos patrones concretos.

El aspectos más crítico en la aparición de cáncer es la mutación de las células. Una mutación puede provocar a esa célula una ventaja evolutiva frente a las demás, o por el contrario, una desventaja. En la primera de las situaciones, lo más probable es que con el paso del tiempo esta célula se reproduzca y que sus genes se hagan cada vez más abundantes en la población, imponiéndose sobre los demás. En el segundo de los casos, se espera todo lo contrario, es decir, que las células que experimenten esa mutación terminen por desaperecer.

Dicho esto, es importante definir algunos conceptos importantes muy utilizados en el mundo de los modelos matemáticos sobre crecimiento de tumores [2] [3]:

- Genotipo: se define como genotipo al conjunto de todos los genes que tienen las células. Los modelos distinguen la posibilidad de que un gen esté mutado o esté sin mutar, de esta manera pueden diferenciar dos genotipos de dos células diferentes mirando qué genes están mutados y cuáles no.

- Fitness o aptitud: este concepto surge a la hora de valorar cuánto de ventajoso es un genotipo frente a otro. Suele ser un número que se le asocia a un genotipo, a mayor valor, más ventajoso es el genotipo, esto es, la célula será resistente y tendrá mayor capacidad de reproducción. Las células con mayor fitness son las células que provocan el crecimiento tumoral y la expansión del mismo.

- Mutaciones pasajeras (*Passenger mutations*): son cambios genéticos que no están directamente relacionados con el desarrollo del cáncer, ya que no influyen positivamente en el fitness.

- Mutaciones conductoras (*Driver mutations*): son cambios genéticos que están muy relacionados con el desarrollo del cáncer. Típicamente producen una ventaja evolutiva y un aumento del fitness.

- Fijación: la fijación es la expansión de un genotipo en toda la población de células. Es decir, una mutación se ha fijado cuando su frecuencia en la población es 1.

- Eventos celulares: la mutación, la reproducción y la muerte de células son ejemplos de eventos celulares.

Por tanto, uno de los objetivos de estos modelos en el ámbito del cáncer es estudiar las probabilidades de fijación de los diferentes genotipos de un conjunto de células. Este problema se puede abordar matemáticamente cuando el conjunto de variables es pequeño, pero cuando se simulan situaciones más realistas con muchos genotipos diferentes la simulación computacional sirve de gran ayuda para lograr este objetivo.

En este punto es importante hacer una breve distinción que puede confundir ligeramente al lector. En algunas ocasiones se habla de modelo en general como algo que está formado tanto por la parte matemática como por la informática; mientras que otras veces se considera por una parte el modelo matemático y por otra el modelo informático, que suele ser un algoritmo computacional que se basa en ese modelo. Ambos puntos de vista son válidos y equivalentes.

Así, dado un modelo matemático, se trata de formalizarlo mediante un algoritmo basado en él, para posteriormente probarlo con diferentes medidas y parámetros, y analizar los datos obtenidos.

2.2. Tipos de modelos y de algoritmos

Antes de estudiar modelos, se considera importante hacer una clasificación de los mismos comprobando si cumplen unas características u otras. Así, se van a diferenciar distintos tipos de casos.

En primer lugar, dentro de la parte matemática del modelo se pueden observar diferentes posibilidades:

En función de cómo se trate el tiempo:

- Modelo en tiempo discreto [4] [5] [6] [7] [8] : cuando solo se muestrean los valores de salida en un conjunto discreto (de cardinal finito o numerable) de instantes de tiempo. Llevado a este campo, cuando se usan estos modelos lo que sucede es que en cada instante de tiempo se pueden producir uno, ninguno o varios eventos.
- Modelo en tiempo continuo [9] [10]: si se quieren conocer los valores de salida en todos los instantes de un intervalo de tiempo. Los modelos continuos suelen estar basados en ecuaciones diferenciales, tanto ordinarias como en derivadas parciales. Llevado a este campo, normalmente estos modelos lo que hacen es calcular el siguiente instante de tiempo en el que se va a producir algún evento.

En función de si hay aleatoriedad o no a partir de la situación inicial:

- Modelo determinista: si dada una misma condición inicial específica siempre da la misma salida, es decir, no hay aleatoriedad. Pero una pequeña desviación en la situación inicial sí puede producir efectos muy diferentes.
- Modelo estocástico o probabilístico: si una condición inicial puede dar diferentes situaciones finales, es decir, el azar interviene en el modelo.

En función del tipo de representación:

- Modelos cualitativos: en general predicen si el estado del sistema irá en determinada dirección o si aumentará o disminuirá alguna magnitud, sin importar exactamente la magnitud concreta.
- Modelos cuantitativos: usan números para representar aspectos del sistema modelizado, y generalmente incluyen fórmulas y algoritmos matemáticos más o menos complejos que relacionan los valores numéricos.

En función de su uso y objetivo:

- Modelo de simulación: aquel que intenta adelantarse a un resultado en una determinada situación. Este tipo de modelos pretende predecir qué sucede a partir de una situación concreta dada.
- Modelo de optimización: este tipo de modelos compara diversas condiciones, casos o posibles valores de un parámetro y mira cual de ellos resulta óptimo según el criterio elegido.
- Modelo de control: este tipo de modelos tiene como objetivo ayudar a decidir qué nuevas medidas o qué parámetros deben ajustarse para lograr un resultado o estado concreto del sistema modelado.

Los modelos matemáticos relacionados con el cáncer, se suelen clasificar típicamente como modelos de simulación, cuantitativos y estocásticos. Por tanto, la diferencia más importante

entre unos y otros suele ser si son de tiempo continuo o de tiempo discreto. Más adelante se verán ejemplos de ambos tipos de modelos y se explicarán las ventajas y desventajas de cada uno.

Ahora se va a hablar de la parte informática del problema, y se van a mencionar los algoritmos (o modelos informáticos) más utilizados en este campo:

En función de los factores de los que dependen los eventos de mutación, reproducción o muerte:

- Modelos en que asumen que los eventos celulares (tales como replicación, mutación y muerte) no se influencian los unos a los otros, es decir, son independientes en cada célula. A este tipo de modelos se les suele llamar *Branching Process Model*.
- Modelos en los que los eventos celulares se influencian de factores microambientales y del entorno que les rodea, por ejemplo, de otras células cercanas.
- Modelos híbridos: aquellos que tienen en cuenta ambas diferenciaciones.

En función de cómo avanza el algoritmo en cada iteración [5]. Para este caso se va a asumir una población inicial de N individuos y se van a considerar sólo los procesos de muerte y reproducción.

- Modelo Wright-Fisher (*Wright-Fisher model*): estos modelos asumen que todos los individuos de la población mueren en cada generación y son remplazados por una descendencia de exactamente N individuos. El genotipo de estos nuevos N individuos vendrá dado en función del fitness del genotipo de sus antecesores, de manera que los individuos con más fitness se reproducirán más que los de menos fitness. El aspecto más relevante de estos modelos es que el tamaño se mantiene constante.
- Modelo de Moran (*Moran model*): estos modelos se caracterizan porque en cada iteración un solo individuo es elegido al azar para reproducirse, y otro para morir. Por tanto la población se mantiene constante.
- *Branching Process Model*: según estos modelos, en cada iteración, cada individuo de la población tiene una probabilidad de reproducirse o de morir. En estos modelos se considera (como hemos visto en la clasificación anterior) que la descendencia de cada célula es independiente de las demás. Se asume que cada individuo con un fitness w_i se reproduce siguiendo una distribución de Poisson de parámetro proporcional al fitness. Para ver cómo fluctúa la población se van a distinguir dos versiones de este modelo:

  1. Branching Process Model (clásico): en este modelo la población puede crecer indefinidamente o incluso decrecer hasta extinguirse. Depende de lo que el azar dado por la distribución de Poisson dictamine en cada caso.
  2. Logistic Branching Process Model: esta versión del Branching Process Model clásico añade un parámetro para regular el tamaño de la población, de manera que esta se mantenga cercana a la población inicial. Se verá con detalle más adelante.

En función del espacio dimensional en el que se trabaje:

- Modelos adimensionales: se considera un conjunto de N individuos y no se diferencia la posición que ocupan unos y otros.
- Modelos espaciales [11] [12]: se consideran modelos espaciales aquellos que están representados en 2 o 3 dimensiones. Necesariamente, en estos modelos los eventos celulares suelen depender del entorno que rodea a las células. Cada individuo, aparte de caracterizarse

por su genotipo, tiene asociada una posición única, además suelen tener la posibilidad de moverse a espacios no ocupados por otras células. Estos modelos son por tanto mucho más complejos. Se verá un ejemplo de un modelo de este tipo en este capítulo.

Un tema que no se ha mencionado apenas es la mutación. La mayor parte de los modelos suelen considerar un genotipo, que está formado por un conjunto de genes, cada uno de los cuales tiene una probabilidad concreta de mutar cuando es evaluado. Se distinguen dos casos:

- Mutación simple: cuando sólo se permite una mutación por iteración. Esto suele ser muy habitual en los modelos en tiempo continuo, ya que el modelo en tiempo continuo se caracteriza por calcular el instante en el que se producirá el siguiente evento, y dos mutaciones serían consideradas como dos eventos, por lo que se harían en iteraciones distintas.
- Mutación múltiple: cuando se permite la mutación de varios genes por iteración. Lo habitual en modelos en tiempo discreto es que al evaluar al individuo también se compruebe si se muta alguno o varios de sus genes o no.

La mayor parte de los modelos se pueden clasificar según estos prototipados.

Así, a grandes rasgos, la parte matemática del modelo consiste en unas fórmulas que se encargan de definir cuándo se producen los eventos celulares y si las células se reproducen, mueren o mutan; y la parte informática consiste en un algoritmo que va recorriendo el conjunto de células, de una manera u otra, y aplica estás fórmulas.

Aún así, hay modelos que difieren un poco de esta clasificación, o que presentan situaciones iniciales diferentes, más específicas, y que abordan el problema desde un punto de vista distinto.

Una vez estudiadas las diferentes características que pueden tener unos modelos u otros, es momento de ver algunos ejemplos concretos. Es a eso a lo que se dedicará la siguiente sección.

2.3. Algunos ejemplos de modelos

Se va a dedicar esta sección a ver con algo más de detalle algunos ejemplos de modelados:

2.3.1. Modelo espacial de Waclaw

El modelo espacial de Waclaw [12] es un modelo de tiempo discreto en 3 dimensiones. La idea que distingue a este modelo de cualquier modelo adimensional es que ahora los individuos tienen asociada una posición en un espacio de 3 dimensiones y que pueden moverse. En este modelo se considera que las células tendrán una capacidad de reproducirse proporcional al número de posiciones vacías que haya a su alrededor (aparte de al fitness). Por tanto, se espera que la masa tumoral adquiera forma de bola más o menos esférica y que la reproducción sea mayor en la superficie de la misma.

Como en otros modelos, las células tendrán una probabilidad d de morir, de manera que dejarían un hueco vacío en la bola. Además, este modelo incorpora una probabilidad M de que las células se muevan tras la replicación. Si $M = 0$ la forma del tumor tendrá forma de bola, y si $M > 0$ se parecerá a un conglomerado de bolas. Esto permite sacar otra conclusión: tal y como se ha definido la probabilidad de replicación, se puede decir que a mayor valor de M mayor velocidad de crecimiento y expansión del tumor.

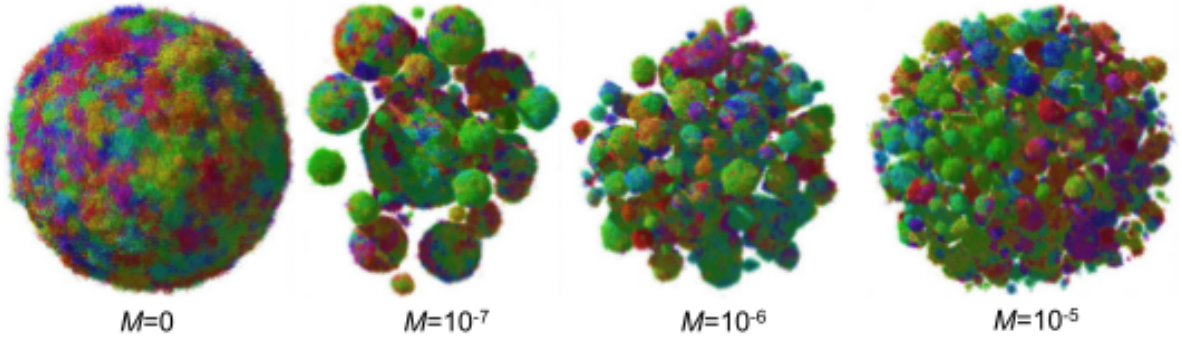

Figura 2.1: Visualización del tumor en forma de bolas para diferentes valores de M (Waclaw et al [12]).

Una característica importante de los modelos es su flexibilidad para poder simular diferentes escenarios. Esto se puede conseguir introduciendo distintos parámetros en el algoritmo informático. Este modelo incorpora la posibilidad de simular varias situaciones interesantes.

Se pueden considerar dos situaciones iniciales distintas:

1. Cuando ya hay un grupo de células con una ventaja genética (y por tanto mayor fitness que las demás) en situación de metástasis, es decir, que ya se han separado del tumor inicial y se disponen a invadir otras zonas del cuerpo.

   En esta primera situación inicial se considera que las nuevas alteraciones genéticas que se produzcan no mejoran el fitness (mutaciones pasajeras). Esto se hace con la intención de asemejarse a lo que ocurre realmente: cuando un tumor se ha desarrollado y un conjunto de células con un genotipo determinado se ha impuesto sobre los demás, es porque este genotipo ha dado a esas células una ventaja genética lo suficientemente importante. Por tanto, al moverse a otra zona del cuerpo en la que no hay mutaciones, es natural pensar que este genotipo va a dominar sin demasiados problemas.

2. Cuando se empieza en el tumor inicial: en esta situación se elimina la consideración anterior de que las alteraciones genéticas no modifican el fitness. Por tanto es un estado más complejo ya que las células tumorales están adquiriendo constantemente mutaciones conductoras (*driver mutations*).

También se pueden cambiar otros parámetros para generar escenarios alternativos:

1. Se puede indicar que las células se replicarán con una probabilidad constante en cuanto que haya una sola posición vacía a su alrededor.

2. Se puede indicar que cualquier célula puede replicarse aunque no tenga posiciones vacías a su alrededor, en ese caso la célula replicada empujaría a las demás para hacerse hueco.

3. Se puede indicar que la replicación y la muerte ocurran sólo en la superficie de la bola. Esto provocará mayor heterogeneidad de células, ya que al no haber renovación celular no habrá opción de que las células de mayor fitness se impongan a las demás.

El algoritmo informático que formaliza y simula este modelo (en su escenario básico) es el siguiente:

Por cada instante de tiempo:

1. Se elige una célula i al azar. A su genotipo e le denomina g.

2. Una zona alrededor de la célula i es elegida al azar y si esta vacía, la célula i se replicará y creará una nueva célula j con probabilidad proporcional a la tasa de replicación de la célula i dividido por la máxima tasa de replicación de las células presentes en el tumor: $\frac{b_g}{b_{max}}$. Al depender de que la zona elegida este vacía, también es proporcional al número de posiciones vacías alrededor de i. En caso de no realizarse la replicación se va al paso 5.

3. Cada una de las dos células i y j, reciben una cantidad n_i y n_j de nuevas mutaciones (número elegido al azar a través de una distribución de Poisson). Si n_i es 0, la célula i no cambiará su genotipo. Igual ocurrirá con la célula j si n_j es 0.

4. Con probabilidad M, la nueva célula j se moverá y formará una nueva bola en una zona cercana a la bola de la que procede (la de la célula i). La nueva posición en la que se queda la nueva bola creada tendrá su centro en $X_j = (X_{1,j}, X_{2,j}, X_{3,j})$ (X_j denota el centro de la bola que tiene, entre otras, a j como célula que la compone) y se calculará: $X_j = X_i + R_i \frac{x_i}{|x_i|}$ donde X_i denota la posición del centro de la bola de la célula i original, R_i es el radio de la bola, y x_i es la posición de la célula i relativa al centro de la bola X_i.

5. La célula i muere con probabilidad igual a su tasa de muerte dividido por la tasa de muerte máxima de las células que componen el tumor: $\frac{d_g}{d_{max}}$. Si todas las células de una bola mueren, la bola es eliminada.

6. El tiempo es incrementado $dt = \frac{1}{b_{max} N}$ donde N es el número de células del tumor.

Se trata por tanto de un modelo en tiempo discreto un tanto especial ya que el incremento de tiempo no es constante.

Este modelo se puede clasificar como un modelo híbrido ya que los eventos celulares dependen tanto de los factores microambientales (las posiciones vacías alrededor de la célula) como del genotipo de la propia célula. Además se puede deducir que sigue al modelo de Moran en cuánto a que en cada iteración de tiempo un sólo individuo se elige al azar para reproducirse, mutarse o morir.

2.3.2. Modelo de Mather

El modelo de Mather [10] es un ejemplo de modelo en tiempo continuo que se basa en todo un clásico: el algoritmo de Gillespie, que es el algoritmo utilizado en modelos de tiempo continuo por excelencia. Mather consigue una mejora sustancial de la velocidad de simulación con respecto a este modelo, gracias al algoritmo BNB (Binomial Negative-Binomial).

Hay varias formas de implementarlo diferentes, tales como el *Gillespies's Direct Method* y el *Gillespie's First Reaction Method*. Se va a estudiar en primer lugar en qué consiste el primero de ellos, y luego se verá el *Gillespie's First Reaction Method* que es el que implementa Mather. También existe una versión mejorada llamada *Gillespie's Next Reaction Method* algo más eficiente ya que aprovecha al máximo los datos tratando de actualizar solo los que sean necesarios.

El funcionamiento del método directo es el siguiente:

Se tiene una función $P(\mu, \tau)d\tau$ que nos da la probabilidad de que una reacción μ ocurra en tiempo τ. Si se integra $P(reaction = \mu)$ respecto de τ, se obtiene la probabilidad en el que se puede producir la reacción μ. Y si se integra $P(\tau)$ respecto de τ, se obtiene cuándo se va producir la siguiente reacción. El algoritmo quedaría:

1. Se inicializa $t \leftarrow 0$ y el número de moléculas de cada tipo.

2. Se elige μ y τ de acuerdo a lo dicho anteriormente. A esta fase también se la llama paso de Monte Carlo, se generan dos números aleatorios para determinar cuál es la siguiente reacción y en qué momento ocurrirá. La probabilidad de que una reacción dada sea elegida es proporcional al número de moléculas de los reactivos.

3. Se incrementa el tiempo y se actualizan las moléculas en consecuencia de la ejecución de la reacción μ.

4. Se vuelve al paso 2.

En este caso, y tratándose de un modelo que simula la reproducción, muerte y mutación de las células, se pueden considerar como reacciones precisamente los eventos celulares, esto es, la reproducción, la mutación y la muerte. Además, estos eventos dependen del fitness de las células, por lo que se podría separar el conjunto de células en grupos con el mismo genotipo. Así, se llamará N al número de conjuntos diferentes de genotipos. Por tanto, se tendrían $3N$ tipos diferentes de reacciones, y habría que calcular los tiempos de aparición de cada uno de ellos para ver cuál es el mínimo, y ejecutar esa reacción.

Pero para simplificar el problema, este modelo sugiere que como reacciones solo se van a considerar las mutaciones, por tanto, en cada iteración el avance de tiempo será el que transcurra entre una mutación y otra.

El algoritmo implementado (en Mather et al [10]) es el BNB (Binomial, Negative-Binomial) y está basado en el *Gillespie's First Reaction Method*. Lo vemos con más detalle:

1. Se inicializa el sistema con N clases de especies según el genotipo. Se especifica la población de cada especie con n_i,$i = 1, ..., N$. Cada especie tendrá asignados sus propias tasas de nacimiento (g_i), muerte (λ_i) y mutación (μ_i).

2. Para cada clase se generarán N números r_i uniformemente distribuidos entre 0 y 1. Y para cada $i = 1, ..., N$, se generará el tiempo t_i que quedaría hacia la siguiente mutación. Estos tiempos se calculan de acuerdo a unas ecuaciones matemáticas que se pueden ver con detalle en el anexo A de [10].

3. Se coge $t_m = \text{mín}(t_i)$ y se actualiza el tiempo $t = t + t_m$.

4. Se actualizarán el número de individuos de cada especie siguiendo el proceso que se describe más adelante.

5. Se selecciona la mutación que ocurre. Si esta genera un miembro que no pertenece a ninguna de las especies existentes, se crea una nueva especie con $n_{N+1} = 1$. En caso contrario se añade 1 a la correspondiente especie.

6. Las especies con 0 individuos se eliminan y se actualiza N al número actual de especies distintas.

7. Se vuelve al paso 2

El algoritmo de actualización del número de individuos es el siguiente: Sean g, λ y μ las tasas de nacimiento, muerte y mutación medias. Sea $p_M(t)$ la probabilidad de que ninguna mutación se haya producido, $p_E(t)$ es la probabilidad de extinción y $p_B(t) = \frac{gp_E(t)}{\lambda}$.

1. Para la especie mutada se genera un número aleatorio $\hat{m}$ de una binomial $B(n_0 - 1, 1 - (p_E(t_m)/p_M(t_m)))$. A continuación se actualiza $n_{i_m} = \hat{m} + 1 + NB(\hat{m} + 2, p_B(t))$ donde NB es una binomial negativa.

2. Para el resto de las especies se genera un numero aleatorio $\hat{m}$ de una binomial $B(n_0, 1 - (p_E(t_m)/p_M(t_m)))$. Si $\hat{m} = 0$ entonces se da a la especie por extinguida. En caso contrario se actualiza $n_i = \hat{m} + 1 + NB(\hat{m} + 2, p_B(t))$ $\forall i = 1, ..., N$, donde NB es una binomial negativa.

Al ser un modelo en tiempo continuo tiene una carga matemática mayor. Además, a diferencia del modelo anterior, no hay mutación múltiple, en cada iteración solo se realiza una mutación. Por otro lado, es semejante a un modelo de Moran en cuanto a que se escoge una célula al azar para ser mutada, pero difiere en la parte en la que se considera que en cada iteración de tiempo se modifica el tamaño del resto de las especies, ya que esto es lo mismo que aplicar una probabilidad de muerte y replicación a cada una de las células.

2.4. Paquete OncoSimulR

El modelo que se implementará estará integrado en el paquete *OncoSimulR* [1] (`https://github.com/rdiaz02/OncoSimul`). Este paquete ha sido desarrollado por Ramón Díaz Uriarte, Profesor Titular del Dpto. de Bioquímica de la Universidad Autónoma de Madrid.

Pertenece a Bioconductor, que es un proyecto de código abierto y que tiene el objetivo de desarrollar e integrar software para el análisis estadístico de datos de laboratorio en biología molecular. Este proyecto esta escrito en el lenguaje de programación R, un código de muy utilizado para el análisis estadístico y representación de datos.

OncoSimulR contiene código en R y en C++ relacionado con la simulación de modelos matemáticos para la investigación del cáncer. Al código en C++ se le llama desde R mediante la librería `Rcpp`, se utiliza C++ en el núcleo del algoritmo debido a que es mucho mas rápido que R.

Pero uno de las grandes funcionalidades que ofrece este paquete, la cual se utilizará en nuestro modelo, es la gran flexibilidad a la hora de indicar cómo se calcula el fitness de un individuo a partir de su genotipo.

Este es un aspecto en el que no se ha profundizado demasiado hasta ahora, y es el momento de hacerlo. El inicio del problema es: ¿qué se quiere diseñar? Se quiere diseñar un modelo que permita simular la evolución de un conjunto de células a lo largo del tiempo. Y ¿de qué factores depende ese crecimiento? Depende de cuál es la célula más fuerte. Las células con mayores ventajas evolutivas serán las que dominen en el conjunto de células y se terminen expandiendo. Pero ese parámetro se mide mediante el fitness, y este depende a su vez del genotipo de la misma.

Por tanto, cuánto más flexible y compleja se consiga hacer la función que calcula el fitness de una célula, mayor cantidad de situaciones diferentes y de mayor complejidad se podrán simular. Por ello, se va a dedicar el resto de la sección en estudiar brevemente las diferentes opciones de especificación de fitness que aporta el paquete *OncoSimulR*.

Para ello se utiliza la siguiente función en R:

```
allFitnessEffects <- function(rT = NULL,
                              epistasis = NULL,
                              orderEffects = NULL,
                              noIntGenes = NULL,
                              geneToModule = NULL,
```

```
                                    drvNames = NULL,
                                    keepInput = TRUE)
```

La salida de esta función se le pasa a C++ en forma de una estructura. En esta estructura se almacenan todos los efectos del fitness que aporta cada gen.

En C++, dada una variable `g` que contiene el genotipo de una célula y otra variable `fE` que contiene la estructura mencionada, se utiliza la siguiente instrucción para calcular el fitness asociado al genotipo `g`:

```
prodFitness(evalGenotypeFitness(g, fE));
```

Donde `evalGenotypeFitness` devuelve un vector de `double` con los efectos del fitness que se van a aplicar al genotipo dado. Y `prodFitness`:

```
double prodFitness(const std::vector<double>& s) {
        return accumulate(s.begin(), s.end(), 1.0,
                []( double x, double y) {return (x * std::max(0.0, (1 + y)));});
}
```

es la implementación de este productorio:

$$\prod_{i=1}^{length(s)} (1 + s_i)$$

En la función principal `allFitnessEffects`, los parámetros `geneToModule`, `drvNames`, y `keepInput` no interesan en este momento, por lo que se van a explicar los otros cuatro:

1) `noIntGenes`: El nombre viene de 'genes que no interactúan', este parámetro es una lista de genes, cada cual lleva asociados su aportación al fitness si está mutados. Se muestra un ejemplo para aclararlo:

Indicamos los efectos del fitness con la función `allFitnessEffects` y nos ayudamos de la función `evalAllGenotypes` para evaluar los fitness de todos los genotipos posibles:

```
fe <- allFitnessEffects(noIntGenes = c("A" = -0.03, "B" = 0.05, "C" = 0.08))

evalAllGenotypes(fe, order = FALSE)
##  Genotype Fitness
##1        A 0.97000
##2        B 1.05000
##3        C 1.08000
##4     A, B 1.01850
##5     A, C 1.04760
##6     B, C 1.13400
##7  A, B, C 1.09998
```

2) `orderEffects`: Sirve para indicar los efectos del fitness en los que importa el orden de mutación de los genes. Este parámetro es una lista de expresiones, cada cual lleva asociada su aportación al fitness. Se muestra un ejemplo para aclararlo:

```
fe <- allFitnessEffects(orderEffects = c("A > B" = 0.4, "B > A" = 0.1,
                                         "C > B > A" = 0.2, "A > C" = -0.3))

evalAllGenotypes(fe, order = TRUE)
##    Genotype Fitness
##1          A    1.00
##2          B    1.00
##3          C    1.00
##4      A > B    1.40
##5      A > C    0.70
```

```
##6       B > A    1.10
##7       B > C    1.00
##8       C > A    1.00
##9       C > B    1.00
##10 A > B > C    0.98
##11 A > C > B    0.98
##12 B > A > C    0.77
##13 B > C > A    1.10
##14 C > A > B    1.40
##15 C > B > A    1.32
```

Por ejemplo, en el número 10, se cumplen $A > B$ y $A > C$, por lo que queda: $1{,}4 \cdot 0{,}7 = 0{,}98$.

3) `epistasis`: Sirve para indicar los efectos del fitness en los que un gen mutado puede perder o ganar ventaja en función de si hay otro mutado. Este parámetro es una lista de expresiones, cada cual lleva asociada su aportación al fitness. Se muestra un ejemplo para aclararlo:

```
fe <- allFitnessEffects(epistasis = c("A:-B" = 0.1, "B:-A" = 0.4, "A : B" = 0.2,
                                      "C:-A:-B" = 0.5, "C:A"=-0.5, "C:B"=0.4))

evalAllGenotypes(fe, order = FALSE)
##  Genotype Fitness
##1        A    1.10
##2        B    1.40
##3        C    1.50
##4     A, B    1.20
##5     A, C    0.55
##6     B, C    1.96
##7  A, B, C    0.84
```

Por ejemplo, en el número 6, se cumplen $B : -A$ y $C : B$, por lo que queda: $1{,}4 \cdot 1{,}4 = 1{,}96$.

4) `rT`: Esta sea quizás la forma más curiosa de especificar el fitness, ya que consiste en una estructura data.frame de R que crea un grafo. Cada nodo del grafo es un gen y tiene unos efectos de fitness asociados si se cumplen unas condiciones y otros si no se cumplen. Esto se indicar mediante el parámetro `typeDep`:

- `typeDep = "MN"`: los efectos de fitness asociados a un nodo se cumplen si todos sus nodos padres están mutados, esto es equivalente a una relación de tipo AND.
- `typeDep = "SM"`: los efectos de fitness asociados a un nodo se cumplen si al menos un padre está mutado, esto es equivalente a una relación de tipo OR.
- `typeDep = "XMPN"`: los efectos de fitness asociados a un nodo se cumplen si sólo uno de los padres están mutados, esto es equivalente a una relación de tipo XOR.

Se muestran algunos ejemplos:

Mediante este código:

```
cs <-data.frame(parent = c(rep("Root", 2), "a","b"),
                child = c("a", "b", "c","c"),
                s = c(0.1,0.2,0.3,0.3),
                sh = c(-0,8,-0.7,-0.5,-0.5),
                typeDep = "MN")

fe <- allFitnessEffects(cs)
```

Se genera el siguiente grafo al ejecutar `plot(fe)`:

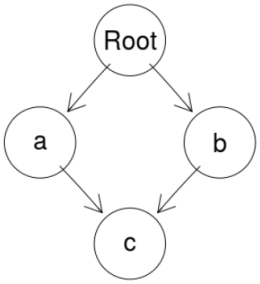

Figura 2.2: Grafo de ejemplo

El nodo *Root* siempre debe especificarse y tiene la propiedad de que siempre está mutado.

Como ya se ha explicado antes, el parámetro `typeDep="MN"` indica que los efectos de fitness asociados a los nodos se tienen en cuenta sólo si el nodo de llegada tiene todos sus padres mutados, es decir, se tienen que cumplir todas las dependencias, como una relación AND.

El parámetro `s` indica cuales los efectos de fitness de los nodos si se cumplen las condiciones según lo indicado por el parámetro `typeDep` ya explicado. Mientras que el parámetro `sh` indica cuales los efectos de fitness si no se cumplen.

Una observación importante es que tanto `s` como `sh` deben ser consistentes con la forma de grafo. Es decir en este caso si se hubiera puesto `s = c(0.1,0.2,0.3,0.4)`, hubiese dado error, ya que habría ambigüedad sobre que fitness coger cuando se cumplan las condiciones para el nodo c, si 0,3 o 0,4.

Se pintan las evaluaciones del fitness para explicarlas a continuación:

```
evalAllGenotypes(fe, order = FALSE)
##  Genotype Fitness
##1        a   1.100
##2        b   1.200
##3        c   0.500
##4     a, b   1.320
##5     a, c   0.550
##6     b, c   0.600
##7  a, b, c   1.716
```

- Los casos 1 y 2 son sencillos, un gen mutado en cada caso y se cumplen las dependencia ya que *Root* siempre está mutado.
- El caso 3 es también sencillo, está mutado el gen c, pero no están mutados a y b, por lo que se aplica `sh`$= -0{,}5$.
- En el caso 4 se encuentran mutados los genes a y b y cumplen las dependencias ya que *Root* siempre está mutado. El fitness total es $1{,}2 \cdot 1{,}1 = 1{,}32$.
- En el caso 5, para el gen a sí se cumplen las dependencias, pero para el gen c no, ya que b no está mutado. EL fitness asociado es $1{,}1 \cdot 0{,}5 = 0{,}55$. El caso 6 es análogo: $1{,}2 \cdot 0{,}5 = 0{,}6$.
- En el caso 7, se encuentran mutados todos los genes, se cumplen por tanto todas las condiciones y el fitness es $1{,}2 \cdot 1{,}1 \cdot 1{,}3 = 1{,}716$.

Se repite la operación con `typeDep="SM"`.

```
cs <-data.frame(parent = c(rep("Root", 2), "a","b"),
                child = c("a", "b", "c","c"),
```

```
                  s = c(0.1,0.2,0.3,0.3),
                  sh = c(-0.8,-0.7,-0.5,-0.5),
                  typeDep = "SM")

fe <- allFitnessEffects(cs)
evalAllGenotypes(fe, order = FALSE)
##  Genotype Fitness
##1        a   1.100
##2        b   1.200
##3        c   0.500
##4     a, b   1.320
##5     a, c   1.430
##6     b, c   1.560
##7  a, b, c   1.716
```

Y por último con `typeDep = "XMPN"`:

```
cs <-data.frame(parent = c(rep("Root", 2), "a","b"),
                  child = c("a", "b", "c","c"),
                  s = c(0.1,0.2,0.3,0.3),
                  sh = c(-0.8,-0.7,-0.5,-0.5),
                  typeDep = "XMPN")

fe <- allFitnessEffects(cs)
evalAllGenotypes(fe, order = FALSE)
##  Genotype Fitness
##1        a    1.10
##2        b    1.20
##3        c    0.50
##4     a, b    1.32
##5     a, c    1.43
##6     b, c    1.56
##7  a, b, c    0.66
```

Aquí se observa la diferencia en el caso 7. Al no cumplirse las condiciones para c queda como fitness $1{,}1 \cdot 1{,}2 \cdot 0{,}5 = 0{,}66$.

Por supuesto, se pueden especificar los efectos del fitness mezclando todas estas formas, e incorporando especificaciones con grafos, con efectos de epistasis, con efectos de orden y/o con genes que no interactúan. Esto da gran flexibilidad y permite considerar situaciones tan complejas como se quiera.

3

Diseño y desarrollo

A lo largo de este capítulo se va a explicar cómo se ha llevado a cabo el diseño y desarrollo de un modelo de evolución tumoral en tiempo discreto. Se explicará con detalle el modelo diseñado, tanto la parte matemática como la parte computacional, entendiendo sus parámetros de entrada y de salida, así como su funcionalidad. También se comentarán las distintas fases por las que ha pasado el diseño y el por qué se ha elegido un camino u otro.

3.1. Julia

El objetivo que se plantea es cómo integrar un nuevo modelo de crecimiento tumoral en el paquete *OncoSimulR*, que como ya se mencionó anteriormente, está escrito en R y C++, relacionando ambos lenguajes con la librería Rcpp. El primer planteamiento que se hizo fue el de hacer el nuevo modelo en un nuevo lenguaje de programación llamado *Julia*, con el objetivo de ir poco a poco migrando parte del código a este lenguaje. ¿Por qué *Julia*? Quizás esa pregunta este contestada por la siguiente imagen:

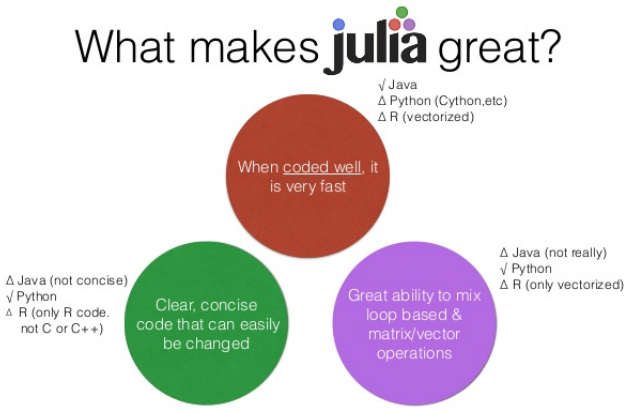

Figura 3.1: Algunas ventajas de Julia respecto a otros lenguajes.

Julia es un lenguaje de programación de alto nivel y rendimiento. Tiene una sintaxis bastante familiar para los usuarios de otros entornos de computación técnica. Además, proporciona un

compilador sofisticado, con ejecución en paralelo, gran precisión numérica y una extensa librería de funciones matemáticas.

Se habla de que puede ser un lenguaje muy utilizado en el futuro. Tiene una gran versatilidad y eficiencia en el análisis de datos y la visualización de resultados. Además permite la comunicación de forma sencilla con otras plataformas como Python (y en su próxima versión, para C++).

Aquí se puede ver un gráfico que compara el rendimiento de C con otros lenguajes de programación, entre ellos Julia:

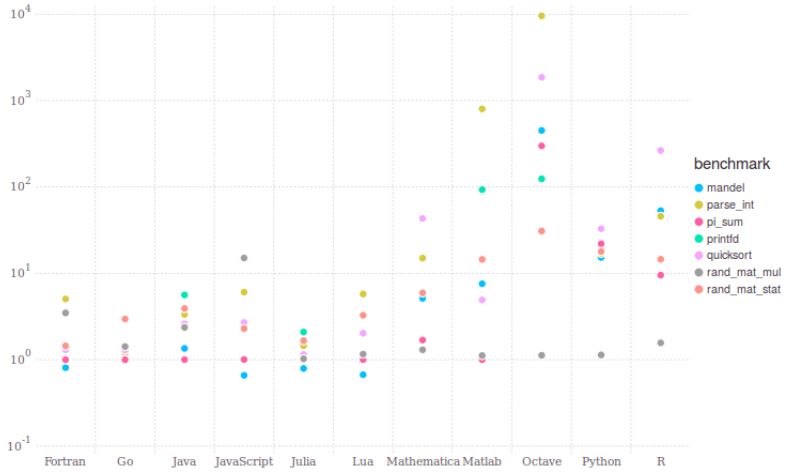

Figura 3.2: Comparación de tiempos con C (Cuanto menos mejor, el rendimiento de C es 1.0).

Parece por tanto bastante razonable sustituir Julia por R, ya que es bastante más rápido y en un futuro permitirá comunicarlo con C++. Además por comodidad y facilidad a la hora de programar, es esperable que también pueda ser sustituido por el código en C++, aunque aquí se pierda algo de velocidad, se ganará en claridad y estructuración de código.

Así, se comenzó la implementación del modelo en tiempo discreto con Julia con la intención de probar este nuevo lenguaje y, en caso de éxito, llevar a cabo la migración completa del código más adelante. Pero llegado un punto del desarrollo se vio que éste requería gran parte de la funcionalidad implementada en C++ y en R.

Finalmente se desechó esta idea, debido a que Julia aún esta en un desarrollo bastante prematuro, no tiene aún disponible la comunicación con C++ y no se veía claro usar una solución intermedia mezclando Julia con lo que hay actualmente en R y C++.

3.2. R y C++

Por tanto, se decidió que el nuevo modelo seguiría la línea de desarrollo que tiene el paquete actualmente, esto es, mezclando los lenguajes de R y C++ mediante la librería Rcpp.

Puesto que el código genera en última instancia un paquete de R, la llamada a las funciones y el paso de parámetros se debe realizar desde R. El motivo de utilizar C++ es porque el núcleo del algoritmo suele ser un proceso bastante complejo y que lleva bastante tiempo, e implementándolo en C++ se consigue que el proceso sea más de 10 veces más rápido que haciendo el mismo algoritmo en R.

Así, el diseño que se ha decidido utilizar es el siguiente:

1. Se ha creado una función de alto nivel en R, llamada `simulDiscrete()` que recibe como argumentos una serie de parámetros que el usuario puede especificar desde R:

```
simulDiscrete <- function(rFE, mu, popIni, tMax, seed, itInfo, verbosity)
```

2. Está función, utilizando la librería Rcpp, llama a una función de bajo nivel, llamada `discreteModel()`, escrita en C++, pasándole los parámetros en un formato legible para C++. Está función no es visible para el usuario, si no que sólo puede ser llamada dentro de `simulDiscrete()`, la cuál sí que es visible para el usuario.

```
// [[Rcpp::export]]
Rcpp::List discreteModel(Rcpp::List rFE,
                         Rcpp::NumericVector mu_,
                         double popIni,
                         int tMax,
                         double seed,
                         int itInfo,
                         int verbosity)
```

3. La función `discreteModel()` es la que realiza el grueso de la simulación. Además, utilizando la librería `Rcpp` envía mensajes a R informando del estado de la simulación.

4. Al finalizar el algoritmo esta función devuelve los resultados en forma de estructura legible para R que el usuario podrá utilizar.

De esta forma se mantiene la estructura original del paquete, haciendo uso de las ventajas que ofrece el lenguaje C++, como son su velocidad de ejecución y su tratamiento de objetos.

3.3. Aspectos generales del nuevo modelo

En esta sección se va a explicar que tipo de modelo se ha escogido y por qué.

Puesto que en el paquete ya había un modelo en tiempo continuo, se ha optado por la implementación de un modelo en tiempo discreto.

Un modelo en tiempo discreto aporta una forma diferente de estudiar la evolución tumoral de la que aporta un modelo en tiempo continuo.

La dos principales ventajas que aporta la implementación de un modelo discreto frente a otro continuo son:

1. La incorporación de mutación múltiple. Los modelos continuos se basan en ir de un evento celular en otro, y al tratarse de crecimiento tumoral, lo más lógico es que la mutación sea considerada un evento, por tanto, como se ha explicado anteriormente, dos mutaciones ocurrirían en dos iteraciones distintas. Mediante un modelo en tiempo discreto, en cada iteración de tiempo se pueden haber producido un número aleatorio de mutaciones diferentes.

2. La opción de ampliarlo a un modelo espacial en un futuro. En un modelo en tiempo espacial, se añade un evento celular más: el movimiento. La idea de introducir movimiento más un espacio tridimensional a un modelo continuo es bastante complicada, habría que diseñar diferentes ecuaciones del movimiento en función del tiempo y sería un asunto bastante enrevesado. Pero introducir la componente espacial a un modelo discreto es mucho más sencillo, el algoritmo de Waclaw es un claro ejemplo de ello.

Siguiendo con la clasificación realizada anteriormente, se va a elegir un algoritmo al estilo del *Logistic Branching Process Model*, de manera que la población total se mantenga estabilizada entre unos ciertos valores. Esta elección se debe a que trabajar con poblaciones de tamaño controlado es bastante cómodo computacionalmente y que esta elección no va a influir en el resultado. Puede que con otro modelo el genotipo que se imponga a los demás tenga mayor representación en número total, pero la cantidad relativa con respecto a las demás poblaciones será parecida en ambos modelos.

Por otro lado, siguiendo la línea de trabajo del paquete *OncoSimulR*, se van a mantener las especificaciones de fitness que en él se concretan. Además, los eventos celulares (tales como replicación, muerte y mutación) dependerán del genotipo (y por tanto del fitness) de las células y no de otros factores microambientales.

Además, se considera que las mutaciones son irreversibles, es decir, si un gen se ha mutado en una célula, esa misma célula nunca podrá volver a tener ese gen sin mutar.

3.4. Implementación

En esta sección se van a indicar los fundamentos matemáticos en los que se rige el modelo y a detallar los aspectos de la implementación del algoritmo que lo formaliza.

3.4.1. Estructuras de datos

Para guardar los datos de toda la población de células se utilizarán las siguientes estructuras de datos:

- `Genotype`: en esta estructura se guarda el genotipo de cada individuo. El genotipo está representado por tres vectores de enteros que representan los identificadores de los genes mutados.

```
struct Genotype {
        std::vector<int> orderEff;
        std::vector<int> epistRtEff; //always sorted
        std::vector<int> rest; // always sorted
};
```

- `Clon`: representa el número de individuos que hay con el mismo genotipo en la población de células. Se guarda el genotipo que tienen, su tamaño y el fitness.

```
struct Clon {
        Genotype genotype;
        double popSize;
        double absfitness; //absolute fitness
};
```

- `DiscreteModel`: guarda toda la información de la simulación en cada iteración. Por orden: población máxima, población actual, tiempo máximo de ejecución, tiempo actual, lista de clones, fitness medio de la población, vector con las probabilidades de mutación de cada gen y especificación de los efectos del fitness.

```
struct DiscreteModel {
        double popMax;
```

```
        double popCurrent;
        int tMax;
        int tPreset;
        std::vector<Clon> listClones;
        double avefitness; //average fitness
        std::vector<double> mu;
        fitnessEffectsAll fE;
};
```

Se ha definido además un fichero *discrete_ model.cpp* y otro *discrete_ model.h* con la funciones nuevas que se han implementado para este modelo. Dichas funciones son:

- Función para calcular el tamaño total de la población:

```
int calculateSizePopulationModelo(const DiscreteModel &dm);
```

- Función para calcular el fitness medio de la población:

```
double calculateFitnessAverageModelo(const DiscreteModel &dm);
```

- Función que calcula el fitness dado un genotipo `g` y los efectos de fitness `fE`:

```
double calculateFitnessGenotype(const Genotype &g,
        const fitnessEffectsAll &fE);
```

- Función que simula la mutación de las células:

```
DiscreteModel mutation(const DiscreteModel &dm,
        const std::vector<int> &muAux, std::mt19937 &ran_gen);
```

- Función que simula la replicación y muerte de las células:

```
int populationVariation(DiscreteModel &dm);
```

- Función que añade una célula de un genotipo `g` al modelo `dm`.

```
void addCellToModel(DiscreteModel &dm2, const Genotype &g);
```

- Función que compara dos genotipos:

```
bool eqGenotypes(const Genotype &g1, const Genotype &g2);
```

- Función para sacar por pantalla el estado del modelo `dm`.

```
void printDiscreteModel(const DiscreteModel &dm);
```

- Función encargada de elaborar la estructura que devuelve la función `discreteModel` al final de la simulación. Será una lista de parámetros legible para R:

```
Rcpp::List structDiscreteModel(const DiscreteModel &dm);
```

3.4.2. Parámetros de entrada

La función `simulDiscrete()`, visible para el usuario desde R al incorporar el paquete *OncoSimulR* admite los siguientes parámetros:

- `rFE`: corresponde con la especificación del fitness, es decir, qué reglas hay que aplicar para calcular el fitness de una célula dado su genotipo. La forma de especificarlo ha sido explicada en la sección 2.4. En concreto, este parámetro corresponde a la salida de la función `allFitnessEffects()`.
- `mu`: vector tan largo como número de genes compongan el genotipo, cada valor de este vector representa la probabilidad que tiene su gen asociado de mutar en cada iteración.
- `popIni`: número de individuos que tendrá la población al principio del algoritmo.
- `tMax`: número de iteraciones máximas que se realizarán.
- `seed`: este parámetro es necesario para ciertos cálculos probabilísticos.
- `itInfo`: indica cada cuántas iteraciones desea el usuario que se le informe de cómo transcurre la simulación.
- `verbosity`: a mayor valor, más detallada será la información que el usuario reciba sobre el transcurso de la simulación. Habrá cambios notorios en la cantidad de información según valga 0, 1, 2 o 3.

3.4.3. Algoritmo

Los parámetros se pasan desde R a C++ a través de la función `discreteModel()` que es llamada desde `simulDiscrete()`.

En ese momento se inicializan las estructuras correspondientes. Para ello se supone una población inicial sin mutaciones. El fitness es calculado a través de la función `calculateFitnessGenotype()`, la cual llama a funciones de C++ ya implementadas en el código desarrollado dentro del paquete *OncoSimulR*, de está forma se consigue reutilizar el código existente e integrar el nuevo algoritmo de manera eficiente. Cuando no hay ningún gen mutado, el fitness es 1.

El fitness medio de la población se calcula llamando a la función `calculateFitnessAverageModelo()`.

A continuación se comienza el bucle principal del algoritmo, que es el siguiente:

```
for(dm.tPreset; dm.tPreset<dm.tMax; dm.tPreset++){
        if(populationVariation(dm)==-1){
                //if population < 1 ---> end
                return List::create(Named("finalTime") = dm.tPreset);
        }
        dm = mutation(dm, muAux, ran_gen);

        if(itInfo > 0 && verbosity >= 2){
                if(((dm.tPreset+1)%itInfo)==0){
                        Rcpp::Rcout << "\n After mutation " //[...] línea oculta
                        if(verbosity >= 3){
                                printDiscreteModel(dm);
                        }
                }
        }
}
```

Por cada iteración de tiempo se llevan a cabo dos pasos:

1. Replicacion y muerte: a través de la función `populationVariation()`.
2. Mutación: a través de la función `mutation()`.

La función `populationVariation()` modifica el tamaño de la población simulando los procesos de replicación y muerte. Consiste en un bucle que recorre la lista de clones y modifica su parámetro `popSize`. Llamando s a la longitud de la lista de clones, la variación vendrá dada por una distribución de Poisson de parámetro λ_i con:

$$\lambda_i = C(x_i)n_i \quad \forall i = 1, ..., s$$

Donde:

- n_i representa el tamaño de la población del clon i.
- $x_i = \frac{w_i}{\hat{w}}$ es el fitness relativo del clon i. w_i es el fitness absoluto del clon i y $\overline{w}$ es el fitness medio de toda la población de células.
- $C = \frac{popIni}{\sum_{i=1}^{s} n_i}$ es el parámetro de regulación de la población, gracias al cuál la población se mantiene siempre entorno a unos valores cercanos a $popIni$, que es la población inicial.

Por tanto, dado que el modelo tiene la propiedad de mantener su población en torno a la población inicial, se debe cumplir que:

$$\sum_{i=1}^{s} \lambda_i = popIni$$

Demostración.

$$\sum_{i=1}^{s} \lambda_i = \sum_{i=1}^{s} C\left(x_i\right) n_i = \sum_{i=1}^{s} \frac{popIni}{\sum_{i=1}^{s} n_i} \left(\frac{w_i}{\overline{w}}\right) n_i = \frac{popIni}{\sum_{i=1}^{s} n_i} \sum_{i=1}^{s} n_i \left(\frac{w_i}{\frac{\sum_{i=1}^{s} n_i w_i}{\sum_{i=1}^{s} n_i}}\right) =$$

$$= \frac{popIni}{\sum_{i=1}^{s} n_i} \sum_{i=1}^{s} n_i w_i \left(\frac{\sum_{i=1}^{s} n_i}{\sum_{i=1}^{s} n_i w_i}\right) = popIni \cdot \left(\frac{\sum_{i=1}^{s} n_i w_i \sum_{i=1}^{s} n_i}{\sum_{i=1}^{s} n_i w_i \sum_{i=1}^{s} n_i}\right) = popIni$$

□

Tras simular la replicación y la muerte, se simula la mutación, la cual es realizada por la función `mutation()`, que modifica el genotipo de las células de la población de acuerdo a las probabilidades indicadas por el vector `mu` que se pasa como parámetro.

Al ejecutar esta función se crea una nueva estructura `DiscreteModel` a la que se le rellenan los campos `popIni`, `popCurrent`, `tMax`, `tPreset`, `mu` y `fE` con los mismos datos que la estructura anterior. El campo `listClones` se inicializa con una lista vacía y `avefitness` se deja sin rellenar.

Posteriormente se inicia un bucle que recorre una a una todas las células de la población y para cada una de ellas hace lo siguiente:

1. Obtiene las posiciones de los genes no mutados de su genotipo.
2. Calcula tantos números aleatorios como genes no mutados tenga el genotipo de la célula. Los números aleatorios calculados corresponden a una distribución uniforme entre 0 y 1.

3. Compara estos números aleatorios con los valores correspondientes del vector `mu` de probabilidades. Si el número aleatorio calculado es menor que la coordenada del vector `mu` que le corresponde, el gen muta.

4. A continuación se llama a la función `addCellToModel()`. Si la lista de clones está vacía, se crea un nuevo clon con ese genotipo, se calcula su fitness (`calculateFitnessGenotype()`), se le asigna `popSize=1` y se añade a la lista (está acción sera la que se realice cuando se llame a esta función por primera vez en cada iteración). Si no, se recorre la lista de clones y se compara el genotipo resultante con los genotipos de los clones existentes (con la función `eqGenotypes()`), si alguno coincide se suma uno al parámetro `popSize` de ese clon. Si no, se crea un nuevo clon con ese genotipo, se asigna `popSize=1`, se calcula su fitness y se añade a la lista de clones.

3.4.4. Parámetros de salida

Tras finalizar su ejecución, la función `simulDiscrete`, devuelve una estructura en R con los siguientes campos:

- `TotalPopSize`: población final.
- `Mu`: vector de probabilidades utilizado.
- `Avefitness`: fitness medio de la población final.
- `NumClones`: número de clones en la población final.
- `PopSizeClones`: vector con los tamaños de población de los clones.
- `AbsfitnessClones`: vector con los fitness de cada uno de los clones.
- `GenotypeClones`: vector con los genotipos de los clones.
- `LargestClonePopSize`: tamaño de población del clon con mayor número de individuos.
- `LargestCloneFitness`: fitness del clon con mayor número de individuos.
- `LargestCloneGenotype`: genotipo del clon con mayor número de individuos.

Además, la ejecución de dicha función de R, emitirá una salida por pantalla de acuerdo a los parámetros `itInfo` y `verbosity` explicados anteriormente. Se verán algunos ejemplos en el siguiente capítulo.

4

Integración, pruebas y resultados

Tras implementar el código, se va a contar brevemente como se realiza la integración del mismo en un paquete de R con la peculiaridad del uso de C++ y de la librería Rcpp. Posteriormente se verá el conjunto de pruebas realizado para asegurarse del buen funcionamiento del código. Se finalizará el capítulo aplicando el modelo a un escenario clásico y analizando los resultados obtenidos.

4.1. Integración

La aportación de este trabajo al paquete *OncoSimulR* ha sido la creación de un modelo en tiempo discreto, cuyo código se encuentra en los ficheros *discrete_model.h* y *discrete_model.cpp*. Aparte de esos dos ficheros se ha añadido un fichero llamado *discreteModel.R* con la función de R visible por el usuario.

La peculiaridad de este paquete es el uso del lenguaje C++ y de la librería Rcpp para llamar desde R a C++.

Si no se tratase de un paquete de R, el uso de la librería Rcpp es sencillo. Consiste en crear un fichero con una función en C++. Añadirle `// [[Rcpp::export]]` como encabezado, y cargar el fichero usando el comando `sourceCpp`. Con estas 4 sencillas instrucciones se podría ejecutar desde R la función `discreteModel()` (suponiéndola sin parámetros) escrita en C++:

```
library(Rcpp)
Sys.setenv("PKG_CXXFLAGS"="-std=c++11") #Flag de compilación
setwd("<path_to_cppFile>.cpp")
sourceCpp("<cppFile>.cpp")
discreteModel()
```

Pero la complejidad es mayor en este caso, ya que la función en C++ sólo se quiere que sea visible para las funciones de dentro del paquete de R, pero no para el usuario. El modo de conseguirlo ha sido el expuesto en [13]. Además de crear la función en C++ poniéndole como encabezado `// [[Rcpp::export]]` tal y como se ha indicado anteriormente, se crean dos archivos: *RcppExports.R* y *RcppExports.cpp*, en los que se encuentran las instrucciones que llevan a cabo esta conexión entre R y C++.

Tal y como se explica en [13] y en [14] estos pasos son realizados automáticamente por la función `Rcpp.package.skeleton()` cuando creas un paquete desde el inicio.

Una vez hecho esto, basta con crear una función en R (en nuestro caso `simulDiscrete()`) que realice una llamada a la función en C++ (en nuestro caso `discreteModel()`) y añadirla a la lista de funciones visibles al cargar el paquete. Esto es, al fichero *NAMESPACE*.

4.2. Pruebas

Para probar el correcto funcionamiento del modelo se han diseñado una serie de pruebas. Para ello se ha utilizado la librería de R `testthat`, que facilita el código y las funciones necesarias para poder crear una batería de tests elegantes y con todas las variaciones que el lenguaje de R puede requerir.

Así, se comprueba que:

- El vector de probabilidades `mu` recibido como argumento tiene todos sus valores positivos y es consistente con la definición de los efectos de fitness.
- La población inicial (`popIni`) recibida como argumento es un número mayor que 0.
- El número de iteraciones (`tMax`), recibido como argumento, que tendrá el algoritmo es un número mayor que 0.
- El tamaño de la población se mantiene en torno al valor inicial.
- El parámetro `verbosity` muestra la información que se espera para sus diferentes valores.
- El parámetro `itInfo` muestra la información en las iteraciones correctas.
- En escenarios sencillos el modelo encuentra la combinación más ventajosa de los genes, es decir, la que provoca mayor fitness.

Para empezar las pruebas, se comienza inicializando una serie de valores:

```
##alternative definitions of fitness effects:
orderFE <- allFitnessEffects(orderEffects = c("F > D" = -0.2, "D > F" = 0.2))

mu <- runif(2,0,0.0001)
popIni <- 100000
tMax <- 1000
verbosity <- 0
seed <- runif(1,0,10)
itInfo <- 0
```

Se van a enseñar algunos de los tests:

1) Probar que el tamaño de la población se mantiene en torno al valor inicial. Este test puede fallar debido a que se basa en el azar, pero se ha diseñado de tal forma que su probabilidad de fallo sea aproximadamente de 1 de cada 10^9 intentos.

Dada una muestra $X_1, ..., X_n$, el estadístico:

$$z = \frac{\overline{X} - \mu}{\sigma/\sqrt{n}}$$

se distribuye según una normal estándar. Por tanto, aplicando el método del pivote se puede construir la expresión:

$$P\left(-z_{\alpha/2} \leq \frac{\overline{X}-\mu}{\sigma/\sqrt{n}} \leq z_{\alpha/2}\right) = 1-\alpha$$

Donde $z_{\alpha/2}$ es el valor de una distribución normal estándar que deja a su derecha una probabilidad de $\alpha/2$. A partir de esto se puede construir el siguiente intervalo de confianza:

$$\mu - z_{\alpha/2}\frac{\sigma}{\sqrt{(n)}} \leq \overline{X} \leq \mu + z_{\alpha/2}\frac{\sigma}{\sqrt{(n)}}$$

La media de una distribución de Poisson de parámetro λ es λ y su desviación típica es $\sqrt{\lambda}$. Si se realiza la simulación 400 veces, con un tamaño de población de 100 individuos (por tanto $\lambda = \mu = 100$ y $\sigma = 10$) y se escoge $\alpha = 0{,}05$, nos queda:

$$100 - 1{,}96\frac{10}{20} \leq \overline{X} \leq 100 + 1{,}96\frac{10}{20}$$
$$99{,}02 \leq \overline{X} \leq 100{,}98$$

Esto quiere decir que la media de poblaciones finales de las 400 simulaciones estará en el intervalo $[99{,}02, 100{,}98]$ el 95 % de las veces. Se considerará que la prueba ha fallado si al repetir este proceso siete veces, la media cae fuera de ese intervalo las siete veces. Y eso, teóricamente tiene una probabilidad de ocurrir de 1 de cada $\frac{1}{0{,}05^7} = 1{,}28 \cdot 10^9$ intentos.

Se considera, sin pérdida de generalidad, `tMax=1`:

```
test_that("The number of cells is maintained around the maximum population",{
    popAcc <- 0
    popIniAux <- 100
    flag <- TRUE
    it <- 0

    while(flag && it<7){
        for(i in 1:400){
            s <- simulDiscrete(rFE = orderFE,
                               mu = mu,
                               popIni = popIniAux,
                               tMax = 1,
                               seed = seed,
                               itInfo = 0,
                               verbosity = 0)
            popAcc <- popAcc + s$TotalPopSize
        }
        popAcc = popAcc / 400
        if(popAcc >= 99.02 && popAcc <= 100.98){
            flag <- FALSE
        }
        it <- it+1
    }
    expect_true(!flag)
})
```

2) Probar que el parámetro `verbosity` funciona correctamente. Se definen para ello los siguientes efectos de fitness:

```
mixFE <- allFitnessEffects(orderEffects = c("F > D" = -0.2, "D > F" = 0.2),
                           epistasis = c("A:-B" = 0.1, "B:-A" = 0.4,
                                         "A : B" = 0.2),
                           noIntGenes = c("X" = -0.03, "Y" = 0.05, "Z" = 0.0))
```

Se muestra un ejemplo de con el parámetro en su máximo valor:

```
muAux <- runif(7,0,0.01)
s <- simulDiscrete(rFE = mixFE, mu = muAux, popIni = 100, tMax = 50, seed = seed,
                   itInfo = 20, verbosity = 3)
```

Aparte de la estructura que devuelve, y que se ha explicado anteriormente, su ejecución provoca la siguiente salida por pantalla:

```
Start --> Population size: 100---- average fitness: 1 ---- clones: 1
    Clon 0 ---- size: 100 ---- genotype:  _  ---- fitness: 1
After mutation 20--> Population size: 97---- average fitness: 1.32031 ---- clones: 8
    Clon 0 ---- size: 12 ---- genotype:  _  ---- fitness: 1
    Clon 1 ---- size: 11 ---- genotype:  _ Y ---- fitness: 1.05
    Clon 2 ---- size: 53 ---- genotype:  _ B ---- fitness: 1.4
    Clon 3 ---- size: 1 ---- genotype:  _ B, Z ---- fitness: 1.4
    Clon 4 ---- size: 3 ---- genotype:  _ B, Y ---- fitness: 1.47
    Clon 5 ---- size: 1 ---- genotype: D _ B, Y ---- fitness: 1.47
    Clon 6 ---- size: 8 ---- genotype:  _ A, B ---- fitness: 1.2
    Clon 7 ---- size: 8 ---- genotype: D > F _ B ---- fitness: 1.68
After mutation 40--> Population size: 98---- average fitness: 1.40582 ---- clones: 6
    Clon 0 ---- size: 76 ---- genotype:  _ B ---- fitness: 1.4
    Clon 1 ---- size: 12 ---- genotype: D _ B ---- fitness: 1.4
    Clon 2 ---- size: 1 ---- genotype:  _ A, B ---- fitness: 1.2
    Clon 3 ---- size: 1 ---- genotype: F _ B ---- fitness: 1.4
    Clon 4 ---- size: 1 ---- genotype: D > F _ B ---- fitness: 1.68
    Clon 5 ---- size: 7 ---- genotype: D _ B, Y ---- fitness: 1.47
End: --> Population size: 105---- average fitness: 1.39387 ---- clones: 7
    Clon 0 ---- size: 83 ---- genotype:  _ B ---- fitness: 1.4
    Clon 1 ---- size: 2 ---- genotype: F _ B ---- fitness: 1.4
    Clon 2 ---- size: 2 ---- genotype:  _ B, X ---- fitness: 1.358
    Clon 3 ---- size: 3 ---- genotype:  _ A, B ---- fitness: 1.2
    Clon 4 ---- size: 13 ---- genotype: D _ B ---- fitness: 1.4
    Clon 5 ---- size: 1 ---- genotype:  _ B, Z ---- fitness: 1.4
    Clon 6 ---- size: 1 ---- genotype: D > F _ A, B ---- fitness: 1.44
```

Las diferencias para los 4 posibles valores de este parámetro son:

- `verbosity <= 0`: no saca ningún resultado por pantalla.
- `verbosity = 1`: muestra únicamente el estado inicial y final.
- `verbosity = 2`: además de lo anterior muestra información detallada de los clones al final de la simulación, así como información básica cada `itInfo` iteraciones. (la línea *After mutation...*)
- `verbosity >= 3`: además de lo anterior muestra información detallada de los clones cada `itInfo` iteraciones.

Y el test es:

```
test_that("parameter verbosity is ok", {
    muAux <- runif(7,0,0.01)
    expect_output(simulDiscrete(rFE = mixFE,
                                mu = muAux,
                                popIni = 100,
                                tMax = 50,
                                seed = seed,
                                itInfo = 20,
                                verbosity = 1),
                  "Start", "End", fixed = TRUE)
```

```
    expect_output(simulDiscrete(rFE = mixFE,
                                mu = muAux,
                                popIni = 100,
                                tMax = 50,
                                seed = seed,
                                itInfo = 20,
                                verbosity = 2),
                  "After mutation", fixed = TRUE)
})
```

3) Probar que el parámetro `itInfo` funciona correctamente. Para ello se le asigna un número primo y se fuerza la situación para que sólo aparezca una vez. También se asigna `verbosity=2`:

```
test_that("parameter itInfo is ok", {
    expect_output(simulDiscrete(rFE = orderFE,
                                mu = mu,
                                popIni = 100,
                                tMax = 12,
                                seed = seed,
                                itInfo = 11,
                                verbosity = 2),
                  "After mutation 11", fixed = TRUE)
})
```

4) Probar que el modelo encuentra la combinación más ventajosa de los genes. Para ello se prueba el modelo con ejemplos sencillos, como los efectos de fitness definidos al principio de esta sección, en los que resulta sencillo encontrar la mejor combinación de los genes:

```
test_that("example with order effects works well", {
    expect_true(simulDiscrete(rFE = orderFE, mu = mu, popIni = popIni,
                              tMax = tMax, seed = seed, itInfo = itInfo,
                              verbosity = verbosity)$LargestCloneGenotype ==
                              "D > F _ ")
})
```

4.3. Resultados

Se va a dedicar esta sección a probar el modelo con un ejemplo clásico.

Este es el ejemplo propuesto por Ochs and Desai [15]. Proponen un escenario en el que ciertas combinaciones de genes mutados provocan directamente la muerte de la célula en la siguiente iteración. Esto se consigue mediante una epistasis que hace que el fitness de esa combinación sea 0.

La definición de los efectos de fitness es como la siguiente:

```
fe <- allFitnessEffects(
    data.frame(parent = c("Root", "Root", "i"),
               child = c("u", "i", "v"),
               s = c(0.1,-0.05, 0.25),
               sh = -1,
               typeDep = "MN"),
    epistasis = c("u:i" = -1,"u:v" = -1))
evalAllGenotypes(fe, order = FALSE, addwt = TRUE)
##  Genotype Fitness
##1       WT  1.0000
##2        i  0.9500
##3        u  1.1000
##4        v  0.0000
##5     i, u  0.0000
```

```
##6      i , v   1.1875
##7      u , v   0.0000
##8   i , u , v   0.0000
```

Por tanto, todas las combinaciones llevan a la muerte celular menos 4. Y para alcanzar la situación de mayor fitness previamente se debe pasar por una mutación que produce desventaja.

Se van a probar estos efectos de fitness cambiando el parámetro `mu` para ver cómo varía el resultado:

- `mu = runif(3,0,0.0001)`:

```
Start --> Population size: 10000---- average fitness: 1 ---- clones: 1
    Clon 0 ---- size: 10000 ---- genotype:  ---- fitness: 1
After mutation 100--> Population size: 10033---- average fitness: 1.09092 ---- clones: 3
After mutation 200--> Population size: 9975---- average fitness: 1.1 ---- clones: 1
After mutation 300--> Population size: 9976---- average fitness: 1.09989 ---- clones: 2
After mutation 400--> Population size: 9926---- average fitness: 1.09978 ---- clones: 3
After mutation 500--> Population size: 10027---- average fitness: 1.1 ---- clones: 1
After mutation 600--> Population size: 10065---- average fitness: 1.1 ---- clones: 1
After mutation 700--> Population size: 9960---- average fitness: 1.09989 ---- clones: 2
After mutation 800--> Population size: 9948---- average fitness: 1.09989 ---- clones: 2
After mutation 900--> Population size: 10060---- average fitness: 1.09978 ---- clones: 3
After mutation 1000--> Population size: 10000---- average fitness: 1.1 ---- clones: 1
End: --> Population size: 10000---- average fitness: 1.1 ---- clones: 1
    Clon 0 ---- size: 10000 ---- genotype: u ---- fitness: 1.1
```

- `mu = runif(3,0,0.001)`:

```
Start --> Population size: 10000---- average fitness: 1 ---- clones: 1
    Clon 0 ---- size: 10000 ---- genotype:  ---- fitness: 1
After mutation 100--> Population size: 9917---- average fitness: 1.09784 ---- clones: 4
After mutation 200--> Population size: 10128---- average fitness: 1.09859 ---- clones: 3
After mutation 300--> Population size: 9839---- average fitness: 1.09944 ---- clones: 3
After mutation 400--> Population size: 9872---- average fitness: 1.09967 ---- clones: 3
After mutation 500--> Population size: 9938---- average fitness: 1.09911 ---- clones: 3
After mutation 600--> Population size: 9968---- average fitness: 1.0989 ---- clones: 3
After mutation 700--> Population size: 10027---- average fitness: 1.09901 ---- clones: 3
After mutation 800--> Population size: 9933---- average fitness: 1.09922 ---- clones: 3
After mutation 900--> Population size: 10011---- average fitness: 1.09923 ---- clones: 3
After mutation 1000--> Population size: 10022---- average fitness: 1.09923 ---- clones: 2
End: --> Population size: 10022---- average fitness: 1.09923 ---- clones: 2
    Clon 0 ---- size: 10015 ---- genotype: u ---- fitness: 1.1
    Clon 1 ---- size: 7 ---- genotype: i, u ---- fitness: 0
```

- `mu = runif(2,0,0.01)`:

```
Start --> Population size: 10000---- average fitness: 1 ---- clones: 1
    Clon 0 ---- size: 10000 ---- genotype:  ---- fitness: 1
After mutation 100--> Population size: 9993---- average fitness: 1.1746 ---- clones: 5
After mutation 200--> Population size: 9988---- average fitness: 1.17585 ---- clones: 2
After mutation 300--> Population size: 9884---- average fitness: 1.17717 ---- clones: 2
After mutation 400--> Population size: 10015---- average fitness: 1.17884 ---- clones: 2
After mutation 500--> Population size: 10057---- average fitness: 1.17817 ---- clones: 2
After mutation 600--> Population size: 10140---- average fitness: 1.17661 ---- clones: 2
After mutation 700--> Population size: 9763---- average fitness: 1.17631 ---- clones: 2
After mutation 800--> Population size: 10038---- average fitness: 1.17768 ---- clones: 2
After mutation 900--> Population size: 10123---- average fitness: 1.17565 ---- clones: 2
After mutation 1000--> Population size: 10126---- average fitness: 1.17683 ---- clones: 2
End: --> Population size: 10126---- average fitness: 1.17683 ---- clones: 2
    Clon 0 ---- size: 10035 ---- genotype: i, v ---- fitness: 1.1875
    Clon 1 ---- size: 91 ---- genotype: i, u, v ---- fitness: 0
```

Se aprecia que al aumentar la probabilidad de mutación se consigue alcanzar el genotipo más ventajoso con mayor facilidad.

Para tener mayor información de los resultados, se va a realizar una simulación más detallada. Se van a probar diferentes combinaciones de los tres parámetros principales: tamaño de población inicial, número máximo de iteraciones y vector de probabilidades, y se va a ejecutar 10000 veces la simulación para cada combinación. Los resultados se pueden observar en la siguiente tabla:

`mu`	`popIni`	`tMax`	WT	i	u	i,v	otro
[0,0.01]	100	1000	83	0	8768	1149	0
[0,0.01]	100	10000	3	0	8842	1155	0
[0,0.01]	1000	1000	2	0	5636	4362	0
[0,0.01]	1000	10000	0	0	5560	4440	0
[0,0.001]	100	1000	2436	0	7539	25	0
[0,0.001]	100	10000	280	0	9631	89	0
[0,0.001]	1000	1000	165	0	9740	95	0
[0,0.001]	1000	10000	6	0	9828	166	0

Cuadro 4.1: Resultados de la simulación del esceneario de Desai con diferentes valores de los parámetros

Con los resultados de la primera mitad de la tabla se pueden hacer algunas observaciones:

- Cuando la probabilidad de mutación es alta, se observan resultados muy semejantes al variar el parámetro `tMax`. Esto parece indicar que el resultado de la simulación no varía al modificar dicho parámetro a partir de un valor. Por tanto, debería ser una práctica importante encontrar el valor óptimo de ese parámetro mediante el cual se puede calcular la probabilidad de fijación de los diferentes tipos de genotipos para una población de células de cierto tamaño, y con una probabilidad de mutación `mu`. Para comprobar este resultado se va a repetir la prueba con valores de `tMax` menores que 1000:

`mu`	`popIni`	`tMax`	WT	i	u	i,v	otro	$\frac{P(i)}{P(i)\cup P(i,v)}$	$\frac{P(i,v)}{P(i)\cup P(i,v)}$
[0,0.01]	100	1000	83	0	8768	1149	0	88,42 %	11, 58 %
[0,0.01]	100	900	118	1	8718	1163	0	88,23 %	11,77 %
[0,0.01]	100	800	139	1	8744	1116	0	88,68 %	11,31 %
[0,0.01]	100	700	163	2	8714	1121	0	11,39 %	88,61 %
[0,0.01]	100	600	196	0	8769	1035	0	89,44 %	10,55 %
[0,0.01]	100	500	239	0	8709	1052	0	89,23 %	10,77 %
[0,0.01]	100	400	329	1	8610	1060	0	89,04 %	10,96 %
[0,0.01]	100	300	518	1	8482	999	0	89,46 %	10,54 %
[0,0.01]	100	200	817	0	8274	905	0	90,14 %	9,86 %
[0,0.01]	100	100	1961	3	7372	664	0	91,73 %	8,27 %

Cuadro 4.2: Resultados de la simulación del esceneario de Desai variando el parámetro tMax

Se observa que al disminuir el número de iteraciones, lo que aumenta es la cantidad de células que presentan un genotipo sin mutaciones. Además lo que se extrae de las última 2 columnas es que en este ejemplo, los porcentajes de fijación de los genotipos más dominantes se mantienen mas o menos en la misma línea entre ellos al disminuir el número de iteraciones (observándose una ligera desventaja a menos cantidad de iteraciones para el genotipo que necesita tener dos mutaciones).

- La población inicial juega un papel importante, se observa que el genotipo más ventajoso $'i, v'$ aparece mucho más en poblaciones de células mayores que en poblaciones pequeñas. Esto puede ser debido a que la presencia de más individuos aumenta la diversidad de especies, lo que provoca que sea más probable la aparición del genotipo más ventajoso y a partir de ahí es más probable que se pueda imponer.

  Para comprobar que esta observación es cierta se va a realizar el experimento con un valor mayor de la población, con el fin de ver que la probabilidad de fijación del genotipo i, v aumenta:

`mu`	`popIni`	`tMax`	WT	i	u	i,v	otro
[0,0.01]	10000	1000	0	0	1302	8698	0

Cuadro 4.3: Resultados de la simulación del esceneario de Desai con tamaño de población alto.

Por lo tanto se sacan dos conclusiones sobre este ejemplo:

- El tamaño de la población inicial es clave para que un genotipo más ventajoso se imponga sobre los demás.
- El número de iteraciones no juega un papel importante en decidir cuál es el genotipo dominante, pero sí es necesario adecuar este parámetro a la probabilidad de mutación, pues si esta es muy pequeña, las células necesitarán más iteraciones para acumular mutaciones.

Lo que se observa en la segunda mitad de la tabla, con probabilidades de mutación mucho más pequeñas, no contradice estas conclusiones:

Además, se observa, al igual que en el ejemplo anterior, que una menor probabilidad de mutación provoca en la población final menos cantidad de células con los dos genes mutados, y más con ningún gen mutado.

También se puede ver que con una tasa de mutación baja se necesitan más iteraciones o mayor tamaño de la población para ver unos resultados razonables, ya que con `popIni=100` y `tMax=1000` predomina en muchas ocasiones el genotipo sin mutaciones, mientras que al aumentar el número de iteraciones cambia el resultado. Sin embargo en las dos últimas filas de la tabla no se observa tanta diferencia al aumentar el número de iteraciones, de aquí se deduce que con `popIni=1000` hacen faltas menos iteraciones para entender el comportamiento del modelo ante ese escenario.

5

Conclusiones y trabajo futuro

A lo largo de este proyecto se ha desarrollado un modelo de simulación de crecimiento tumoral en tiempo discreto. Se han dado las bases matemáticas del mismo y se ha explicado el algoritmo informático que lo formaliza. Además se ha integrado en el paquete *OncoSimulR* y se ha explicado el funcionamiento del mismo y de la librería `Rcpp`.

Estos métodos de simulación son muy útiles para examinar el resultado de escenarios que son intratables matemáticamente. Mediante el muestreo de los datos que producen y el análisis estadístico de los resultados se puede predecir el comportamiento de la población ante un escenario concreto de una forma sencilla y rápida [16] [17] [18] [19].

Además, el resultado final de las simulaciones no es tan trivial como suponer que se va a imponer el genotipo más ventajoso. Cómo se ha visto en la sección anterior, hay situaciones en que la llegada al genotipo más ventajoso es complicada y en poblaciones de células pequeñas o de baja probabilidad de mutación.

También se ha observado que en un escenario con los mismos efectos de fitness y los mismos posibles genotipos, una pequeña variación en los tres parámetros principales (tiempo, población y probabilidad de mutación) puede producir desenlaces diferentes.

Otra conclusión que se obtiene es que es posible optimizar el tiempo de prueba de un escenario concreto, ya que se observa que con una población inicial y una probabilidad de mutación fijas, se obtienen los mismos resultados con un tiempo de vida de 10000 iteraciones que con uno de 1000.

5.1. Líneas de trabajo futuro

La causa por la que se decidió añadir un modelo discreto al paquete *OncoSimulR* existiendo ya un modelo continuo fue porque el modelo discreto permite explorar situaciones distintas que con el continuo no se podría, como la mutación múltiple.

La siguiente razón importante era la posibilidad de incorporar un modelo espacial. Es mucho más sencillo y tiene más sentido añadir componente espacial a un modelo en tiempo discreto que a uno en tiempo continuo.

Por tanto la continuación natural de este proyecto sería incorporar componente espacial al modelo.

Aparte de esto, habría otras líneas de trabajo que van encaminadas a mejorar la usabilidad y eficiencia del paquete *OncosimulR* y de sus métodos de simulación como son:

- Ampliar la información de salida que ofrece el modelo. Incorporar información de la evolución del modelo por iteraciones de tiempo y no solo del resultado final.
- Añadir funciones de procesado estadístico de los modelos para obtener las probabilidades de fijación de los genotipos en diferentes condiciones.
- Añadir una interfaz de usuario (por ejemplo con *Shiny*) que mejore visualmente la salida y entrada del programa.

Bibliografía

[1] Ramon Uriarte-Díaz. Oncosimulr: forward genetic simulation in asexual populations with arbitrary epistatic interactions and a focus on modeling tumor progression. 2016.

[2] Philipp M.Altrock, LIn L.Liu, and Franziska Michor. The mathematics of cancer: integrating quantitative models. *Nature*, 2015.

[3] Niko Beerenwinkel, Roland F.Schwarz, Moritz Gerstung, and Florian Markowetz. Cancer evolution: Mathematical models and computational inference. 2012.

[4] R.B Campbell. A logistic branching process for population genetics. *Journal of Theoretical Biology*, 2002.

[5] Wonpyong Gill. Modified fixation probability in multiple alleles models in the asymmetric sharply-peaked landscape. *Journal of the Korean Physical Society*, 2009.

[6] John H. Gillespie. Substitution processes in molecular evolution. i. uniform and clustered substitutions in a haploid model. *Genetics*, 1993.

[7] Benjamin H.Good, Igor M.Rouzine, Daniel J.Balick, Oskar Hallatschek, and Michael M.Desai. Distribution of fixed beneficial mutations and the rate of adaptation in asexual populations. *PNAS*, 2012.

[8] Christoper J.R.Illingworth and Mustonen Ville. Distinguishing driver and passenger mutations in an evolutionary history categorized by interference. *Genetics*, 2011.

[9] Michael A. Gibson and Bruck Jehoshua. Efficient exact stochastic simulation of chemical systems with many species and many channels. 2000.

[10] William H.Mather, Jeff Hasty, and Lev S.Tsimming. Fast stochastic algorithm for simulating evolutionary population dynamics. 2012.

[11] Erik A Martens, Rumen Kostadinov, and Carlo C Maley. Spatial structure increases the waiting time for cancer. *New Journal of Physics*, 2011.

[12] Bartlomiej Waclaw, Ivana Bozic, Meredith E.PittMan, Ralph H.Hruban, bert Volgelstein, and Martin A.Nowak. A spatial model predicts that dispersal and cell turnover limit intratumour heterogeneity. *Nature*, 2015.

[13] Dirk Eddelbuettel and Romain François. Writing a package that uses rcpp. 2016.

[14] Dirk Eddelbuettel, Romain Francois, JJ Allaire, Kevin Ushley, Qiang Kou, DOuglas Bates, and John Chambers. Package 'rcpp'. 2016.

[15] Ian E Ochs and M Michael Desai. The competition between simple and complex evolutionary trajectories in asexual populations. *BMC Evolutionary Biology*, 2015.

[16] K.R. Thornton. A c++ template library for efficient forward-time population genetic simulation of large populations. *Genetics*, 2013.

[17] D. Kessner. Forqs: Forward-in-time simulation of recombination, quantitative traits and selection. *Bioinformatics*.

[18] X. Yuan, D.J. Miller, J. Zhang, D Herrington, and Y Wang. An overview of population genetic data simulation. journal of computational biology. 2012.

[19] Ramón Díaz Uriarte. Identifying restrictions in the order of accumulation of mutations during tumor progression: effects of passengers, evolutionary models, and sampling. 2015.