Universidad de Valladolid

Facultad de **Ciencias Económicas y Empresariales**

Trabajo de Fin de Grado

Grado en **Economía**

Teoría de juegos: una interfaz gráfica

Presentado por:

Enrique Sanjuán Calvo

Valladolid, 2 de junio de 2023

RESUMEN

La teoría de juegos es un área de las matemáticas que modeliza y estudia las interacciones entre varios agentes. Se trata de un campo con numerosas aplicaciones en la ciencia económica y que, como consecuencia, forma parte del programa formativo de cualquier estudiante de economía.

En el presente trabajo se exponen los modelos fundamentales de la teoría de juegos aplicada a la economía (Juegos Bimatriciales y modelos de Cournot, Stackelberg y Bertrand) y las soluciones de equilibrio de Nash relacionadas con cada uno de ellos. Asimismo, se desarrolla una aplicación en Python que implementa y resuelve estos modelos en una interfaz gráfica accesible con múltiples funcionalidades, destinada principalmente a profesores y estudiantes de este área. Se trata de ofrecer una herramienta que facilite la introducción a los modelos mencionados, generando y resolviendo ejercicios de cada campo mencionado con anterioridad.

Palabras clave: Python, Teoría de Juegos.

ABSTRACT

Game theory is an area of mathematics that models and studies the interactions between multiple agents. It is a field with numerous applications in economic science and, therefore, is part of the educational program for any economics student.

This paper presents the fundamental models of game theory applied to economics (Bimatrix Games and Cournot, Stackelberg, and Bertrand models) and the Nash equilibrium solutions associated with each of them. Additionally, a Python application is developed that implements and solves these models in a user-friendly graphical interface with multiple functionalities, primarily aimed at teachers and students in this field. The goal is to provide a tool that facilitates the introduction to the aforementioned models by generating and solving exercises in each of the mentioned areas.

Keywords: Python, Game Theory.

Clasificación JEL:

JEL: C72 - Juegos no cooperativos.

JEL: C88 - Otros programas informáticos.

ÍNDICE

ÍNDICE DE TABLAS

ÍNDICE DE ILUSTRACIONES

1. INTRODUCCIÓN Y ALCANCE DEL PROYECTO

El principal objetivo de este proyecto es el desarrollo de una aplicación que resuelva diversas cuestiones del ámbito de la teoría de juegos, a saber: equilibrio de Nash en juegos bimatriciales, modelo de Cournot, modelo de Stackelberg y modelo de Bertrand. Teniendo en cuenta que cada uno de estos campos tiene materia suficiente para disponer cada uno de su propia aplicación, la profundidad de la aplicación desarrollada en los campos seleccionados es relativa. Empero, el fin de este proyecto no es tanto estrictamente científico como educativo; se pretende crear una aplicación que trate los conceptos y modelos más básicos, prestando apoyo a estudiantes y profesores en la introducción a estas áreas. En la actualidad no existe ninguna alternativa que comprenda la resolución de todos los modelos mencionados en una sola aplicación o página web. Las pocas que existen, están centradas normalmente en un único modelo y no son tan accesibles para un usuario básico.

A este respecto quiero destacar la existencia de Gambit, una interfaz que permite construir y resolver juegos en forma estratégica o extensiva. Se trata de un programa muy maduro que ofrece una herramienta profunda para el análisis de estos juegos, pero que presenta los dos defectos mencionados anteriormente.

Además, como otro factor distintivo, la aplicación que se desarrolla en este trabajo ofrece la opción de exportar a Word todos los modelos resueltos, facilitando su manipulación y posterior impresión.

2. METODOLOGÍA

2.1. Python 3.10

Python es uno de los lenguajes de programación más populares del mundo en la actualidad. Es el lenguaje de programación con más porcentaje de búsqueda de tutoriales en internet (27.7%, PYPL Index), siendo el lenguaje con mayor crecimiento a este respecto en los últimos 5 años (7.1%, PYPL Index). También es el lenguaje más popular según el índice Tiobe, el cual tiene en cuenta las búsquedas en los motores de búsqueda más populares, número de ingenieros capacitados y número de cursos, entre otros.

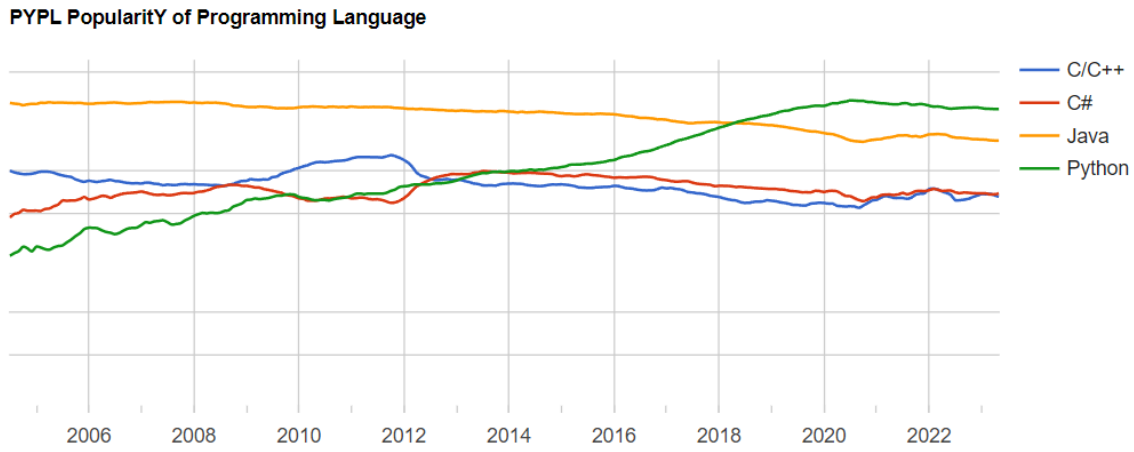


Ilustración 1: PYPL PopularitY of Programming Language.

Sus principales usos son: inteligencia artificial, ciencia y análisis de datos, proyectos de investigación… Sin embargo, la cantidad de bibliotecas de las que dispone extienden su uso hacia prácticamente todos los campos de la programación.

Entre sus ventajas nos encontramos con su sencillez y su amplia fama (que implica mayor documentación y proyectos). Es un lenguaje que prioriza la reducción del tiempo de desarrollo y la legibilidad. Es bien conocida la relativamente lenta velocidad de ejecución de Python en comparación con la de lenguajes como C o C++. Sin embargo, dadas las características de la aplicación a desarrollar y las numerosas bibliotecas optimizadas que posee, Python es más que suficiente para mi propósito en este trabajo.

Sumando lo ya mencionado a mi experiencia previa en el uso de Python, se hace evidente la elección de este lenguaje para llevar a cabo el proyecto. La elección de la versión (3.10) no es completamente arbitraria; Python 3.10 es la versión de Python más reciente compatible con la biblioteca Kivy. Estoy al tanto de la reciente salida de Python 3.11 y sus mejoras en eficiencia pero la incompatibilidad con Kivy dejan su uso apartado hasta un futuro, esperemos próximo.

2.2. Kivy 2.1.0

Kivy es una biblioteca de Python de código abierto que permite desarrollar aplicaciones con interfaces gráficas. Existen numerosas bibliotecas de Python que realizan funciones similares (PyQt, Tkinter, PySide2…), cada una con sus ventajas e inconvenientes. La elección de Kivy para este proyecto se basa fundamentalmente en estas 3 características:

- **Sencillez**: Kivy tiene un lenguaje específico relativamente fácil de aprender, con la particularidad de habilitar la escritura de código en “kv files”; reduciendo el número de palabras clave necesarias para cada proceso. Además, la legibilidad de las kv files es muy superior al mismo código escrito en un archivo Python, lo que juega en favor de su comprensión y mantenibilidad.
- **Flexibilidad**: Kivy cuenta con numerosos widgets que permiten una alta personalización de las aplicaciones. Además, he introducido en algunos apartados widgets de KivyMD (Material Design), que ofrecen funcionalidades específicas y combinan a la perfección con los widgets de Kivy.
- **Portabilidad**: las aplicaciones desarrolladas en Kivy son multiplataforma, es decir, se pueden ejecutar en dispositivos con Windows, Linux, Android, iOS, entre otros. (Kivy Documentation (2023)).

3. ESTRUCTURA

La aplicación cuenta con 10 archivos, 5 archivos con código Python (.py) y 5 con código Kivy (.kv). Se puede condensar todo el código en un único archivo Python pero ello, desde mi punto de vista, lo haría extremadamente caótico y poco mantenible. Cada archivo .py tiene asociado un archivo .kv, de esta manera existen 5 “grupos” de archivos que componen el código de la aplicación:

- **main.py** y **kivynomics.kv**: almacenan el código básico de la aplicación. Por un lado aparecen definidas la pantalla principal (de selección) y las pantallas del menú principal de ajustes e información. Además, en estos archivos quedan definidas las principales clases que luego usarán los demás objetos de la aplicación.
- **Resto**: un archivo .py y uno .kv para cada apartado de la aplicación. En cada uno de ellos se detalla la distribución de cada ventana así como los métodos para resolver cada modelo. Además, también se detallan las ventanas de más información específicas para cada apartado, donde aparecen instrucciones que explican cómo debe ser el input introducido y cómo interpretar el output obtenido. El contenido específico de cada apartado lo desarrollaré en el presente trabajo.

4. PANTALLA PRINCIPAL

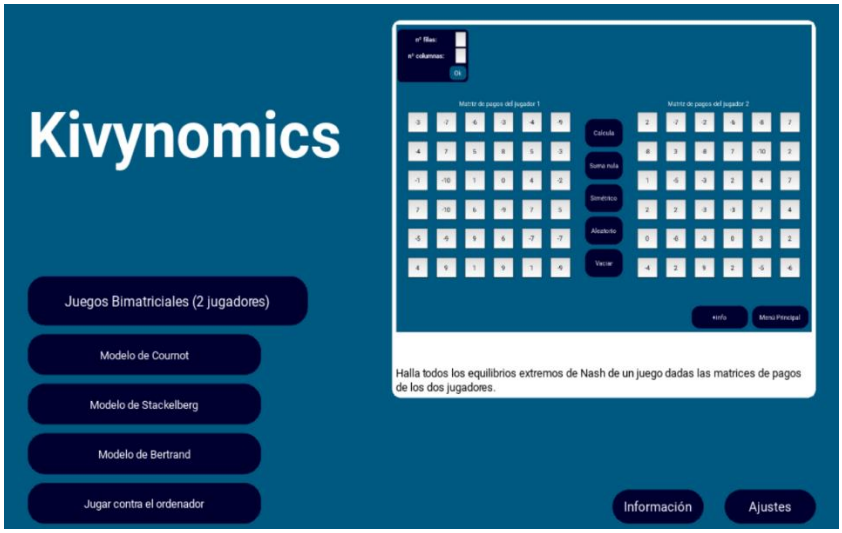


Ilustración 2: Menú principal de kivynomics.

La pantalla del menú principal de la aplicación nos permite navegar hacia cada apartado. Posando el puntero del ratón sobre un botón, se mostrará una previsualización de la ventana en cuestión junto a una breve descripción del modelo tratado en su interior. Dentro de "Información" podemos acceder a contenidos como el código fuente o mi página de Github, por donde se podrá contactar para solicitar ayuda con la instalación u ofrecer sugerencias.

Los botones de esta pantalla y el resto de la aplicación están construidos sobre la base de KivyMD, lo que les aporta el HoverBehaviour (comportamiento al posar el puntero del ratón sobre ellos), dando un carácter más actual a la aplicación.

El apartado de ajustes, en principio vacío, pretende habilitar configuraciones de la aplicación como la selección de otros idiomas o el tema (estilo) de la aplicación. Dada la extensión del trabajo desarrollado, este apartado queda vacío por el momento para, de cara a un futuro, poder introducir ajustes fácilmente. En el apartado de conclusiones se mencionan y estudian posibles aportes interesantes a futuro.

5. JUEGOS BIMATRICIALES

5.1. Teoría

Desde la segunda mitad del siglo XIX se han desarrollado trabajos en los que se intuyen las bases de la teoría de juegos (Cournot 1838, Edgeworth 1881, entre otros), sin embargo, el origen de la teoría de juegos como tal se le atribuye a la obra *"Theory of Games and Economic Behaviour" (1944)* de Von Neumann y Morgenstern. En ella se establecen las bases de la teoría de juegos, se

estudian las situaciones de conflicto entre 2 jugadores y se ofrecen resultados para los juegos de suma nula. En los años 50, John Forbes Nash, matemático estadounidense y premio Nobel de Economía en 1994, introduce conceptos muy relevantes en este mismo campo (equilibrio de Nash, negociación de Nash). En los años posteriores, investigadores como Selten profundizan más aún en el estudio de la teoría de juegos, estudiando juegos dinámicos, juegos con información incompleta y facilitando la integración de la teoría de juegos con la ciencia económica.

Podemos definir “juego” como una situación en la que determinados agentes (jugadores) deben tomar decisiones para obtener un resultado (pago), teniendo en cuenta que este resultado se verá influido por las decisiones de los demás jugadores.

Existen diferentes tipos de juegos pero todos tienen en común los siguientes elementos:

- **Jugadores**: participantes en el juego, que toman sus decisiones de forma racional (con el fin de maximizar su utilidad esperada). Son como mínimo dos.
- **Acciones de cada jugador**: decisiones que puede tomar cada jugador en cada momento en que le toque jugar. El número de acciones puede ser finito o infinito.
- **Resultados del juego**: modos en que puede concluir el juego para cada jugador.
- **Pagos**: representan la utilidad que el jugador atribuye al resultado de un juego, es decir, la valoración que para el jugador tienen las consecuencias de alcanzar un determinado resultado en el juego. Pagos más altos indicarán mayor utilidad y viceversa.
- **Estrategias. Perfiles de estrategias**: una estrategia es un plan completo de acciones con las que este podría proponerse participar en un juego. Un perfil de estrategias es un conjunto de estrategias, una por cada jugador. (Pérez Navarro, J. et al. (2013)).

Un juego puede ser representado en forma estratégica (matricial, centrado en las estrategias de los jugadores) o extensiva (en forma de árbol, poniendo el foco en el desarrollo secuencial del juego).

En cuanto a la solución de un juego, no existe una perspectiva única. Existen varios conceptos de solución que hacen hincapié en unos aspectos u otros. Sin embargo, el concepto de solución más extendido es el de equilibrio de Nash: perfil de estrategias del que ningún jugador tiene incentivos para desviarse, dadas las estrategias decididas por el resto de jugadores. Es decir, situación en la que cada jugador está ofreciendo una respuesta óptima (maximizadora de utilidad) ante las acciones de los demás.

Un equilibrio de Nash puede serlo en estrategias puras, los jugadores juegan una acción en cada momento de decisión, o en estrategias mixtas, los jugadores asignan una distribución de probabilidad sobre sus acciones en cada momento de decisión.

Además, en 1951 en el artículo Non-Cooperative Games de la revista *Annals of Mathematics*, el propio Nash demuestra que todo juego finito tiene al menos un equilibrio de Nash.

5.2. Algoritmos: Nashpy vs Gambit

Dado que el fin de este trabajo no está centrado únicamente en la resolución de juegos bimatriciales y que a este respecto ya existe mucha investigación, mi intención era implementar un algoritmo ya existente. La creación o implementación de un nuevo algoritmo para obtener equilibrios de Nash es una tarea compleja que probablemente ocuparía un TFG completo por sí solo.

Nashpy es una biblioteca de Python creada por el profesor de matemáticas de la Universidad de Cardiff, Vince Knight. Cuenta con una amplia documentación y un gran número de funciones que van desde la obtención de equilibrios de Nash hasta la implementación de procesos de Moran, entre otros. Su principal objetivo es proporcionar una biblioteca de Python que ofrezca las herramientas típicas para juegos de dos jugadores.

En un principio tenía pensado usar Nashpy, pues se trata de un proyecto de código abierto escrito completamente en Python. Para el cálculo de equilibrios de Nash, Nashpy usa 3 algoritmos (enumeración de soportes, Lemke-Howson y enumeración de vértices) con sus ventajas e inconvenientes. (Nashpy Documentation (2023)).

Sin embargo, a pesar de ser un proyecto muy prometedor, considero que Gambit está mucho más pulido para este propósito. Nashpy presenta problemas en la obtención de todos los equilibrios de Nash en algunos juegos degenerados

(como se informa en su documentación), esto es, juegos en los que alguna estrategia mixta con tamaño de soporte k tiene más de k respuestas óptimas puras.

Gambit por su parte es probablemente el proyecto de herramientas de software para teoría de juegos más popular que existe. Es también de código abierto y cuenta con una interfaz gráfica propia y una API para Python. Como se reconoce en la propia documentación de Nashpy, Gambit es un proyecto más maduro, sofisticado y eficiente.

Dentro de los diferentes algoritmos que ofrece Gambit, he seleccionado gambit-enummixed ya que garantiza encontrar todos los equilibrios extremos de Nash. En rigor no obtiene todos los equilibrios de Nash (ya que estos pueden ser infinitos en numerosas ocasiones) pero sí las estrategias puras que son puntos de equilibrio y mediante las cuales podemos formar combinaciones lineales para obtener todos los equilibrios. (Gambit Documentation (2023)).

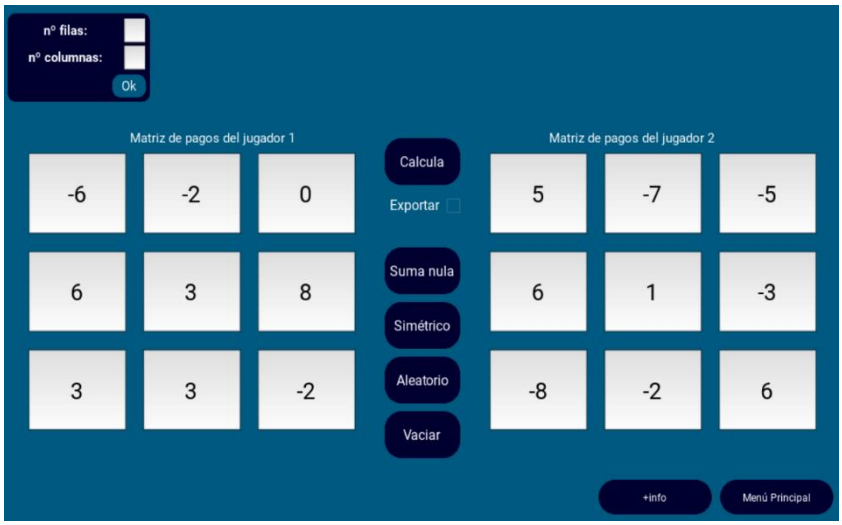


Ilustración 3: Pantalla "Juegos Bimatriciales" de kivynomics.

La aplicación toma como input las matrices de pagos de los 2 jugadores de dimensiones desde 2x2 hasta 6x6. En cada posición de cada matriz se admiten números formados por un máximo de 5 caracteres. Se puede navegar entre cada posición de las matrices de pagos mediante el tabulador, permitiendo una entrada rápida de los pagos. El botón “Calcula” almacena los datos introducidos por el usuario en matrices de Numpy, que serán usadas para computar los equilibrios de Nash por el algoritmo gambit-enummixed.

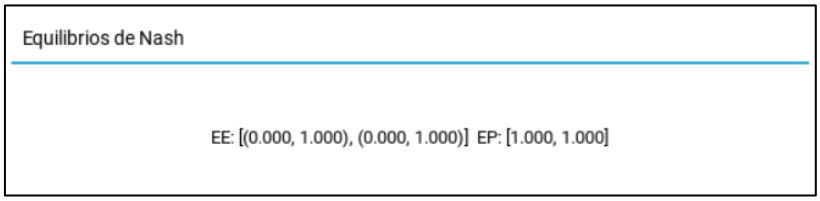


Ilustración 4: Output de "Juegos Bimatriciales" de kivynomics

Cada equilibrio de Nash del juego se mostrará como una fila en una ventana

emergente y se interpreta de la siguiente manera:

- **EE (Extreme Equilibria o Equilibrio Extremo):**
  - [(Estrategia del jugador 1), (Estrategia del jugador 2)] . Distribución de probabilidades con la que cada jugador jugará cada estrategia.
- **EP (Expected Payoff o Pago Esperado):**
  - [Pago esperado para j1, Pago esperado para j2] . Pago esperado para cada jugador en el equilibrio.

En el caso mostrado en la *Ilustración 4* obtenemos como resultado un único equilibrio de Nash, en estrategias puras.

Si se marca la opción "Exportar" antes de calcular, se creará un documento de Word con el juego representado en forma bimatricial y los equilibrios de Nash. Esto es posible gracias a la biblioteca Python-docx, que permite crear y editar documentos de Word desde código Python.

Dentro de esta pantalla se incluyen además:

- Un botón para generar juegos aleatorios: asigna pagos aleatorios a cada jugador mediante el uso de Numpy.
- Botones de "suma nula" y "simétrico": habiendo rellenado la matriz de pagos del jugador 1, haciendo click sobre estos botones se rellenan los pagos del jugador 2 formando un juego de suma nula o simétrico según el caso.
- Un botón para vaciar los campos.

6. DUOPOLIO DE COURNOT

6.1. Teoría

Como ya mencionábamos en la introducción, Antoine Augustin Cournot, matemático y economista francés, es uno de los primeros autores en intuir la teoría de juegos con su modelo de duopolio propuesto en su obra de 1838 *"Recherches sur les Principes Mathématiques de la Théorie des Richesses".*

El duopolio de Cournot es un modelo económico en el que dos empresas compiten determinando simultáneamente la cantidad de un bien que van a producir. Los supuestos básicos son:

- Ambas empresas producen un bien homogéneo.
- La variable de decisión es la cantidad.

- Los costes de cada empresa son una función creciente con la cantidad.
- Es un modelo de un solo periodo.
- Las empresas determinan simultánea e independientemente la cantidad producida, considerando que la producción de las demás es independiente de las decisiones propias.

Estos supuestos caracterizan al modelo de Cournot como un juego estático no cooperativo de dos jugadores (las empresas). Cada uno de ellos con un conjunto de estrategias idéntico: la cantidad producida, que tomará valores en el intervalo [0, ∞). Los pagos para cada empresa los define su función de beneficios. La solución del modelo es el equilibrio de Nash, donde cada empresa produce la cantidad que maximiza su beneficio, dada la producción de la rival (correspondencia de respuesta óptima).

El juego se resuelve obteniendo una función de respuesta óptima para cada empresa y resolviendo el sistema derivado de ambas funciones obteniendo así un perfil de estrategias [producción de la empresa 1, producción de la empresa 2] que es simultáneamente respuesta óptima para las dos empresas.

En la aplicación se implementa el siguiente modelo:

Sea x_1 la cantidad producida por la empresa 1, x_2 la cantidad producida por la empresa 2 y $X = x_1 + x_2$ la suma de las producciones de ambas empresas.

La función de demanda del mercado: $p(X) = a - bX = a - bx_1 - bx_2$.

La función de costes totales de la empresa 1: $CT_1(x_1) = cx_1$.

La función de costes totales de la empresa 2: $CT_2(x_2) = ex_2$.

Cada empresa trata de maximizar su beneficio total:

Empresa 1

$$\max \pi_1 = IT_1 - CT_1 = p(X)x_1 - cx_1 = (a - bx_1 - bx_2)x_1 - cx_1 .$$

Derivamos e igualamos a 0: $\frac{\partial \pi_1}{\partial x_1} = a - 2bx_1 - bx_2 - c = 0$.

Y despejamos para obtener la función de reacción. Esto es, la función que ofrece la respuesta óptima de la empresa 1 ante la cantidad producida por la empresa 2:

$$x_1 = \frac{a - bx_2 - c}{2b} = \frac{a - c}{2b} - \frac{x_2}{2} .$$

Empresa 2

Operamos del mismo modo para la empresa 2 y obtenemos también su función de reacción:

$$x_2 = \frac{a - bx_1 - e}{2b} = \frac{a - e}{2b} - \frac{x_1}{2}.$$

Equilibrio

Resolvemos el sistema sustituyendo una ecuación en la otra:

$$x_1 = \frac{a - c}{2b} - \frac{1}{2}\left(\frac{a - e}{2b} - \frac{x_1}{2}\right) = \frac{a - c}{2b} - \frac{a - e}{4b} + \frac{x_1}{4},$$

$$x_1 - \frac{x_1}{4} = \frac{3}{4}x_1 = \frac{a - c}{2b} - \frac{a - e}{4b} = \frac{2a - 2c - a + e}{4b} = \frac{a - 2c + e}{4b},$$

$$x_1 = \frac{4}{3} * \frac{a - 2c + e}{4b},$$

$$x_1^* = \frac{a - 2c + e}{3b}.$$

Sustituyendo este resultado en la función de reacción de la empresa 2 obtenemos:

$$x_2^* = \frac{a - 2e + c}{3b}.$$

Por tanto, el equilibrio de Cournot (equilibrio de Nash) es el par:

$$(x_1^*, x_2^*) = \left(\frac{a-2c+e}{3b}, \frac{a-2e+c}{3b}\right).$$

Sustituyendo las producciones óptimas en la función de demanda de mercado obtenemos el precio de equilibrio resultante.

El beneficio de cada empresa se calcula restando los costes totales a los ingresos totales, una vez hemos determinado la producción óptima y el precio de mercado:

$$\pi_1 = IT_1 - CT_1 = px_1^* - cx_1^*.$$

6.2. Código

Para tratar con este modelo y las diferentes funcionalidades que implementará la aplicación, he escogido las bibliotecas numpy, sympy, matplotlib y Python-docx.

Numpy servirá para asignar valores aleatorios en el método “aleatorio”, al igual que hacíamos con los juegos bimatriciales. Se asignan siempre números enteros en el intervalo [1, 10] para las funciones de costes y para la X de la función de demanda y enteros en [20, 150] para el término constante de la función de demanda.

Sympy nos permite “representar” las funciones de forma interna, para luego sustituir ciertos datos o graficarlas.

Matplotlib es una biblioteca de Python muy conocida, que, junto con su módulo backend para kivy, nos permite representar los gráficos de la aplicación. En concreto, esto se logra mediante el uso de la clase FigureCanvasKivyAgg.

Python-docx, como ya explicamos en el apartado de juegos bimatriciales, nos permite crear y editar documentos de Word directamente desde Python. Con el uso de esta biblioteca podemos generar el output deseado en Word, incluyendo el correspondiente gráfico con el equilibrio de Cournot.

El botón (y método) “vacía” elimina todo el contenido introducido en las celdas que recogen el input del usuario. “Aleatorio” ofrece valores aleatorios para cada cuadro de input del apartado, por si se desea generar un ejemplo de forma aleatoria. El método “obtenermasinfo” asociado al botón de “más información” nos dirige a una pantalla donde se explica cómo debe introducirse el input y cómo debe interpretarse el output.

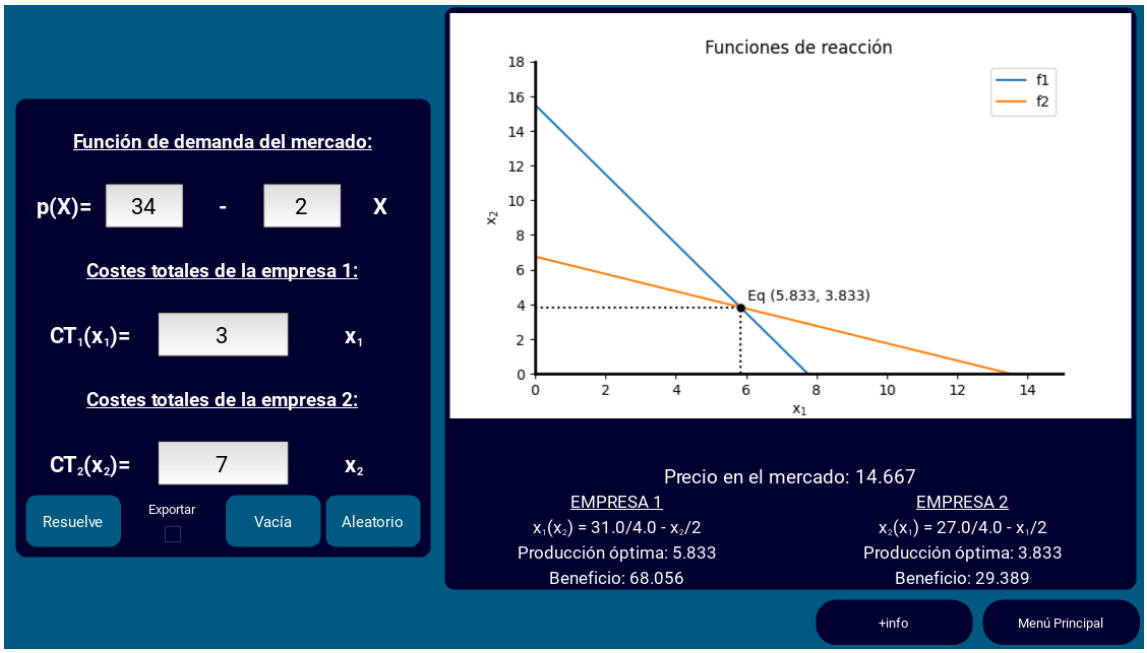


Ilustración 5: Pantalla "Modelo de Cournot" de kivynomics.

El método “calcula”, recoge la información de las entradas de texto (TextInput) del usuario y la almacena en variables. Con estas variables realiza las operaciones necesarias y produce el output mostrado en pantalla y, en su caso (marcando “exportar”), en Word. Las operaciones que realiza el programa, en concreto, son las descritas en el apartado teórico. En este caso sin necesidad de

derivar constantemente las funciones de beneficio puesto que, por la simplicidad del modelo, se implementan directamente las funciones de reacción.

7. MODELO DE STACKELBERG

7.1. Teoría

Heinrich Freiherr von Stackelberg, economista alemán, introduce su famoso modelo de duopolio en su obra *"Marktform und gleichgewicht"* publicada en 1934.

El duopolio de Stackelberg es también un modelo económico en el que dos empresas compiten decidiendo sobre su producción. Los supuestos básicos son muy parecidos a los del modelo de Cournot, con algunas variaciones:

- Ambas empresas producen un bien homogéneo.
- La variable de decisión es la cantidad.
- Los costes de cada empresa son una función creciente con la cantidad producida.
- Es un modelo de dos periodos.
- Las empresas determinan la cantidad producida de forma secuencial; una empresa (líder) determina su producción en el primer periodo. Esta decisión es irreversible: no puede cambiarse en el segundo periodo. Teniendo en cuenta la producción de la líder, la otra empresa (seguidora) determina su producción óptima en el segundo periodo.

El modelo es, por tanto, un juego dinámico con información perfecta: un jugador toma una decisión tras conocer la decisión que ha tomado el otro jugador. En este caso, la empresa líder toma una decisión sobre la cantidad a producir y la seguidora, conociendo la cantidad producida por la líder, decida su cantidad a producir. Al igual que en el modelo de Cournot, las empresas cuentan con un conjunto de estrategias idéntico: la cantidad producida, que tomará valores en el intervalo $[0, \infty)$. El juego puede representarse de forma extensiva de la siguiente manera:

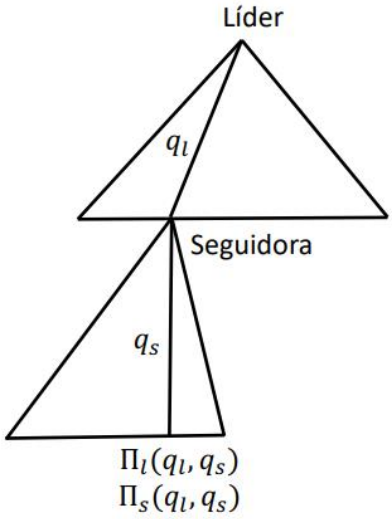


Ilustración 6: UC3M. Modelo de Stackelberg.

El juego se resuelve por inducción hacia atrás, hallando el Equilibrio de Nash Perfecto en Subjuegos:

Sea x_1 la cantidad producida por la empresa 1 (líder), x_2 la cantidad producida por la empresa 2 (seguidora) y $X = x_1 + x_2$ la suma de las producciones de ambas empresas.

La función de demanda del mercado: $p(X) = a - bX = a - bx_1 - bx_2$.

La función de costes totales de la empresa 1: $CT_1(x_1) = cx_1$.

La función de costes totales de la empresa 2: $CT_2(x_2) = ex_2$.

La empresa 1 determina su producción conociendo cómo será la respuesta de la empresa 2.

Empresa 2

La empresa 2 maximiza su beneficio obteniendo una función de reacción como la de Cournot, es decir, una función de respuesta óptima para cada cantidad producida por la empresa líder:

$$x_2 = \frac{a - bx_1 - e}{2b} = \frac{a - e}{2b} - \frac{x_1}{2}.$$

Empresa 1

La empresa 1 (líder) conoce cómo será la reacción de la empresa 2 ante su producción. Con esta información maximiza su beneficio:

$$\max \pi_1 = IT_1 - CT_1 = p(X)x_1 - cx_1 = (a - bx_1 - bx_2)x_1 - cx_1$$
$$= ax_1 - b{x_1}^2 - bx_1x_2 - cx_1 .$$

Sustituimos la función de reacción de la empresa 2 y agrupamos términos:

$$ax_1 - bx_1{}^2 - bx_1\left(\frac{a-e}{2b} - \frac{x_1}{2}\right) - cx_1 = ax_1 - bx_1{}^2 - \frac{x_1}{2}(a-e) + \frac{b}{2}x_1{}^2 - cx_1 .$$

Derivamos e igualamos a 0: $\frac{\partial \pi_1}{\partial x_1} = a - 2bx_1 - \frac{(a-e)}{2} + bx_1 - c = 0$.

Y despejamos para obtener la producción óptima de la empresa líder:

$$2bx_1 - bx_1 = a - \frac{(a-e)}{2} - c ,$$

$$bx_1 = \frac{a}{2} + \frac{e}{2} - c ,$$

$$x_1^* = \frac{a + e - 2c}{2b} .$$

Sustituyendo este resultado en la función de reacción de la empresa 2 obtenemos su producción óptima:

$$x_2^* = \frac{a-e}{2b} - \frac{1}{2}\left(\frac{a + e - 2c}{2b}\right) = \frac{a-e}{2b} - \frac{a + e - 2c}{4b} = \frac{2a - 2e - a - e + 2c}{4b},$$

$$x_2^* = \frac{a - 3e + 2c}{4b} .$$

Por tanto, el equilibrio de Stackelberg es el par:

$$(x_1^*, x_2^*) = \left(\frac{a+e-2c}{2b}, \frac{a-3e+2c}{4b}\right) .$$

Sustituyendo las producciones óptimas en la función de demanda de mercado obtenemos el precio de equilibrio resultante.

7.2. Código

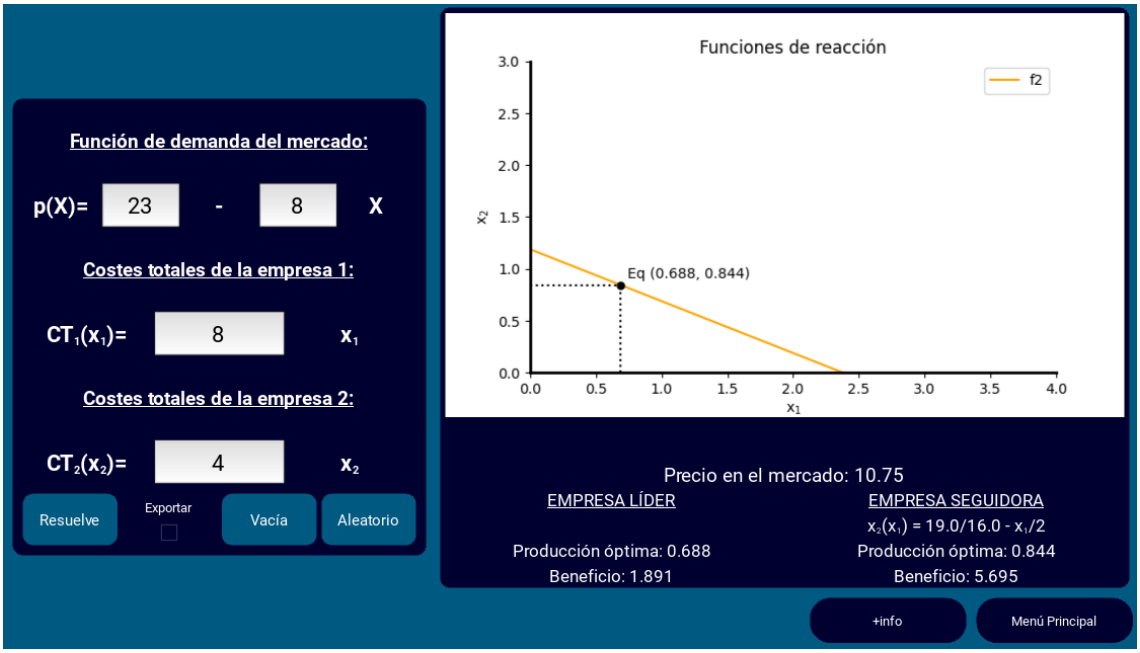


Ilustración 7: Pantalla "Modelo de Stackelberg" de kivynomics.

Dado el gran número de similitudes entre el modelo de Cournot y el de Stackelberg, no creo necesario rellenar este apartado explicando nuevamente todas las bibliotecas usadas (las mismas), ni el diseño de la ventana. Cualquier usuario de la aplicación se dará cuenta de que ambos apartados utilizan el mismo formato de ventana. Es más, la propia ventana de Stackelberg hereda la clase CournotVentana, cambiando solamente algún método:

- "vacía" y "aleatorio": permanecen igual.
- "obtenermasinfo": nos dirige ahora hacia la ventana de más información del modelo de Stackelberg.
- "calcula": como es de esperar, sufre cambios puesto que el modelo a resolver ahora es el de Stackelberg.

8. MODELO DE BERTRAND

8.1. Teoría

Joseph Louis François Bertrand, matemático y economista francés del siglo XIX, revisó el modelo de duopolio de Cournot en un artículo para la revista Journal des Savants en 1883, indicando que en una situación de duopolio las empresas acabarían compitiendo en precios, dando forma al conocido como modelo de Bertrand.

El modelo de Bertrand es un modelo de competencia entre dos o más empresas en el que, a diferencia de en los modelos anteriores, la variable de decisión es el precio. Esto es, las empresas deciden a qué precio venderán su producto y la cantidad será la determinada por la demanda del mercado para dicho precio. Los supuestos básicos son los siguientes:

- Ambas empresas producen un bien homogéneo.
- Las empresas presentan costes marginales constantes.
- La variable de decisión es el precio. Las empresas determinan simultáneamente el precio al que venden el bien en cuestión.
- No existen restricciones de capacidad; cada empresa puede satisfacer completamente toda la demanda del mercado.
- No existen costes de transacción ni de búsqueda para los consumidores, es decir, la decisión de los consumidores se fundamenta

únicamente en el precio establecido por las empresas: seleccionarán la empresa que ofrezca el bien al precio más barato.

Partiendo de costes marginales iguales, solo existe un equilibrio de Nash en el modelo: precio_1 = precio_2 = Coste Marginal. Siempre que el precio sea superior al coste marginal, existen incentivos para desviarse de la situación:

Imaginemos: precio_1 > precio_2 > Coste Marginal:

La empresa 2 está acaparando toda la demanda del mercado y la empresa 1 no tiene demanda. Esta última no está ofreciendo una respuesta óptima puesto que podría establecer un precio marginalmente inferior a precio_2, llevarse toda la demanda y obtener beneficios positivos.

Imaginemos ahora: precio_1 = precio_2 > Coste Marginal:

Los demandantes son indiferentes entre acudir a una empresa u otra por lo que asumiremos, sin pérdida de generalidad, que la demanda se distribuye equitativamente entre ambas empresas. Las dos empresas tienen incentivos para establecer un precio marginalmente inferior al presente, llevándose toda la demanda y obteniendo un beneficio mayor.

Imaginemos, por último: precio_1 > precio_2 = Coste Marginal:

La empresa 2 se lleva toda la demanda del mercado, pero sus beneficios son 0 (p = CMg). Tampoco es un equilibrio de Nash porque la empresa 2 puede ofrecer una respuesta mejor, estableciendo un precio marginalmente inferior al de la empresa 1, manteniendo toda la demanda del mercado pero consiguiendo unos beneficios positivos. La casuística es la misma si sustituimos en la demostración precio_2 por precio_1.

En el único equilibrio de Nash: precio_1 = precio_2 = Coste Marginal

Ambas empresas tienen beneficios iguales a 0 y la demanda se distribuye entre las dos. En este caso cada empresa está ofreciendo su respuesta óptima ante la decisión de la rival; ninguna tiene incentivos para reducir el precio puesto que obtendrían beneficios negativos (p<CMg), ni para aumentarlo puesto que seguirían sin beneficios porque ningún consumidor pagaría un precio superior al establecido por la otra empresa. Se produce lo que se conoce como la paradoja de Bertrand: en un modelo de duopolio, las empresas llegan a un equilibrio de competencia perfecta, donde p=CMg y, por tanto, los beneficios son normales (0).

En el modelo implementado en la aplicación se puede romper el supuesto de costes marginales idénticos, con el fin de determinar resultados más interesantes.

8.2. Código

El modelo de Bertrand se resuelve en la aplicación con una función que, dados los parámetros de la demanda del mercado y costes marginales, evalúa varias condiciones y genera un diccionario con los resultados:

Partimos de:

La función de demanda del mercado: $p(X) = a - bX$.

La función de costes totales de la empresa 1: $CT_1(x_1) = cx_1$.

La función de costes totales de la empresa 2: $CT_2(x_2) = ex_2$.

Lo primero que hace la función es comprobar que los parámetros introducidos son razonables:

```
def solve_bertrand(a, b, c, e):

    # Comprobamos si los parámetros son compatibles con una solución razonable del modelo
    if a <= c and a <= e:
        raise ValueError('Introduce parámetros válidos')
```

Si los costes marginales de la empresa 1 son menores que los de la empresa 2, esta tendrá el poder de decisión. Deberá elegir el precio en el intervalo [c, e-ε], de manera que mantenga toda la demanda y maximice sus beneficios. En realidad, solo debe comprobar si puede establecer su precio de monopolio, precio que maximiza sus beneficios cuando se enfrenta a toda la demanda del mercado, dejando fuera a la otra empresa y, si no, establecer un precio marginalmente inferior a los costes marginales de la empresa 2.

Punto de monopolio de la empresa 1:

$$\max \pi_1 = IT_1 - CT_1 = (a - bx_1)x_1 - cx_1 = ax_1 - b{x_1}^2 - cx_1 .$$

Derivamos e igualamos a 0: $\frac{\partial \pi_1}{\partial x_1} = a - 2bx_1 - c = 0$; ${x_1}^M = \frac{a-c}{2b}$.

¿Cuál es el precio de monopolio asociado a dicha cantidad?:

$$p^M = a - b{x_1}^M = a - b * \frac{a-c}{2b} = a - \frac{a-c}{2} = \frac{2a - a + c}{2} = \frac{a+c}{2}.$$

De ahí la condición establecida en el código: si el precio de monopolio ($\frac{a+c}{2}$) de la empresa 1 es superior o igual al coste marginal de la empresa 2, la empresa

1 no puede establecerlo porque permitiría entrar a su rival en el mercado. Dado el caso, su respuesta óptima sería establecer un precio marginalmente inferior (0.001 en nuestro caso) al coste marginal de la empresa 2:

```
if c < e:
    # La empresa 1 se lleva toda la demanda
    if (c + a) / 2 >= e:
        # Si no puede establecer precio de monopolio, establece
        # un precio marginalmente inferior al Coste Marginal de
la empresa 2
        p = e - .001

    else:
        p = (c + a) / 2

    x1_sol = (a - p) / b
    x2_sol = 0
```

En el caso opuesto, cuando los costes marginales de la empresa 2 son inferiores, es esta la que tiene el poder de decisión. Deberá tomar la misma decisión que la empresa 1 en el caso anterior: establecer su precio de monopolio o un precio marginalmente inferior al coste marginal de su rival:

```
elif c > e:
    # La empresa 2 se lleva toda la demanda
    if (e + a) / 2 >= c:
        # Si no puede establecer precio de monopolio, establece
        # un precio marginalmente inferior al Coste Marginal de
la empresa 2
        p = c - .001

    else:
        p = (e + a) / 2

    x1_sol = 0
    x2_sol = (a - p) / b
```

En el caso en el que los costes marginales sean idénticos entre las dos empresas, supondremos que la demanda se distribuye a partes iguales:

```
else:
    # Distribuimos la demanda a partes iguales
    p = c
    x1_sol = (a - c) / (2 * b)
    x2_sol = (a - c) / (2 * b)
```

Al final, calculamos los beneficios de cada empresa y pasamos los resultados a un diccionario, para poder leerlo luego desde otras funciones de la aplicación:

```
x_total = x1_sol + x2_sol
beneficio1 = (p - c) * x1_sol
beneficio2 = (p - e) * x2_sol

return {'p': p, 'q': x_total, 'q1': x1_sol, 'q2': x2_sol, 'pro-
fit1': beneficio1, 'profit2': beneficio2}
```

De la misma manera que en Cournot y Stackelberg, tenemos los botones con los métodos para vaciar, exportar y generar unos parámetros aleatorios. Esta ventana, como se puede apreciar en la imagen, también hereda la clase de la ventana de Cournot.

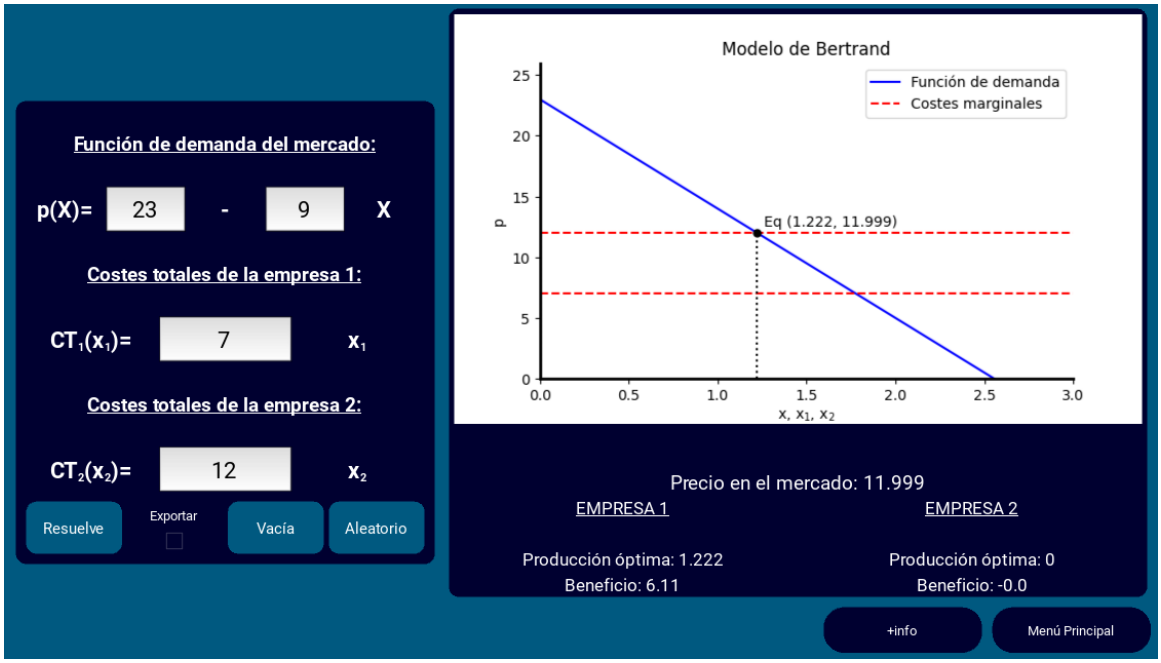


Ilustración 8: Pantalla "Modelo de Bertrand" de kivynomics.

9. JUEGO VS NPC

Quizá la implementación de los modelos anteriores sea incentivo suficiente para hacer la aplicación atractiva. Sin embargo, en aras de la originalidad, he decidido introducir un apartado más “creativo”, que mezcle los modelos ya presentados con un papel más activo por parte del usuario y dote de personalidad al programa. Ya que la aplicación está más centrada en un usuario del ámbito educativo (ya sea alumnado o profesorado) y están en auge nuevos métodos de enseñanza como la gamificación, me parece interesante la introducción de este apartado.

El juego implementado consiste en 3 etapas en las que tú, dueño de una empresa, competirás con uno de los tres NPCs (non-playable characters) disponibles, dueño de la empresa rival. Cada uno de estos jugadores tiene un

comportamiento predefinido distinto, cambiando la facilidad con la que podrás obtener beneficios en cada periodo:

- **John Forbes Nash**: siempre tomará las decisiones más óptimas, teniendo en cuenta todos los eventos. Buscará las soluciones de equilibrio de Nash para cada modelo.

- **Murray Rothbard**: responderá generalmente de forma acertada pero tiene una debilidad: la intervención estatal. Ignorará los impuestos establecidos y las subvenciones concedidas por el Gobierno porque las considera ilegítimas. Lo ignorará de cara a su decisión de producción, sin embargo, sus costes reales serán igual a los tuyos por lo que puedes aprovecharte de esta circunstancia. Ten en cuenta que su desaprobación de la intervención estatal es permanente, por lo que "acumulará" la ignorancia sobre las intervenciones pasadas, así que es recomendable ir apuntándolo para optimizar tus decisiones, por ejemplo:

CT = 10x

Etapa 1:

"El Gobierno introduce un impuesto sobre la producción de 2 u.m."

- Murray Rothbard producirá como si siguiera teniendo CT = 10x, cuando en realidad sus costes son mayores (12x).

Etapa 2:

"El Gobierno concede una subvención sobre la producción de 4 u.m."

- Murray Rothbard producirá como si siguiera teniendo CT = 10x (ignora tanto el impuesto del anterior periodo como la subvención de este), cuando en realidad sus costes reales son de 8x.

Dejo el fragmento del código donde se crea la clase NPCMurray, que contiene los métodos que definirán la actuación de este NPC en cada modelo:

Iniciamos la clase asignándole sus frases e imagen, incluyendo la variable tax, que será la que acumule el impacto de las actuaciones del Gobierno para hacer que este NPC no las tenga en cuenta.

```
class NPCMurray:
    # Variable que (des)informará al NPCMurray sobre los impuestos del
Gobierno
    tax = 0

    def __init__(self, img, frases):
        self.img = img
        self.frases = frases
        self.name = 'Murray Rothbard'
```

Definimos un método para cada modelo, teniendo en cuenta que este NPC descontará el impacto de las actuaciones estatales en cada caso:

```
# Cournot
@staticmethod
def cournot(a, b, c):
    return (a - (c + NPCMurray.tax)) / (3 * b)

# Líder
@staticmethod
def stackelberg1(a, b, c):
    return (a - (c + NPCMurray.tax)) / (2 * b)

# Seguidora
@staticmethod
def stackelberg2(a, b, c, x1):
    result = ((a - (c + NPCMurray.tax)) / (2 * b)) - (x1 / 2)
    if result > 0:
        return result
    else:
        return 0

# Bertrand
@staticmethod
def bertrand(c):
    return c + NPCMurray.tax
```

Cuando el NPC actúa como seguidor, introducimos una condicional que impide la producción negativa del NPC ya que si el jugador 1 (líder) decide producir una gran cantidad, la función de reacción del NPC le dictará producir una cantidad negativa (cambiamos la respuesta óptima a 0 porque en este caso el mercado estaría saturado por la producción de la empresa líder). El resto de los NPCs están construidos de la misma manera, introduciendo las particularidades mencionadas.

- **Karl Marx**: sobreestima sus costes con la intención de no extraer plusvalía de sus trabajadores. En concreto, producirá como si tuviera unos costes un 20% mayores a los reales. En este caso, el NPC (a diferencia de Rothbard) tiene en cuenta todos los eventos que sucedan. Además, ese 20% no es acumulativo, es decir, en cada periodo actuará como si sus costes fueran un 20% superiores a los tuyos para ese mismo periodo:

  CT = 10x

  Karl Marx producirá como si tuviera CT = 12x .

Además, se incluyen 3 frases célebres asociadas a cada uno de los personajes disponibles, una de las cuales será seleccionada aleatoriamente al inicio de cada partida para aparecer bajo su imagen en pantalla.

Cabe recordar que el enfrentamiento con el ordenador no es siempre rigurosamente equitativo puesto que, a pesar de tener los mismos costes totales en todo momento, en el modelo de Stackelberg el orden de entrada en el mercado desequilibra la balanza (favorece a la empresa líder considerablemente).

A lo largo del juego deberás decidir la cantidad a producir (modelos de Cournot y Stackelberg) o el precio que vas a establecer (modelo de Bertrand) teniendo en cuenta la personalidad del NPC, el modelo a tratar en la etapa y, por supuesto, las funciones de demanda del mercado y costes totales.

La selección del NPC rival deberás hacerla al iniciar el juego. La selección del modelo es completamente aleatoria entre las 4 opciones: modelo de Cournot, modelo de Stackelberg (como seguidora o líder) y modelo de Bertrand.

Además, como ya se puede intuir por las descripciones de los NPCs, se introducen eventos aleatorios en cada etapa, que afectarán a las funciones de costes o a la demanda del mercado. Existen, en concreto, 4 posibles eventos aleatorios que se pueden definir como:

- **Aumento de la demanda del mercado**: por un estudio favorable de la universidad de Massachusetts.
- **Reducción de la demanda del mercado**: por la propagación de una pandemia a nivel mundial.
- **Aumento de los costes totales**: como consecuencia de la introducción de un impuesto sobre la producción.
- **Reducción de los costes totales**: por la concesión de subvenciones para la producción.

Todos los jugadores se verán afectados por estos eventos. Los cambios que estos eventos conlleven serán introducidos actualizando los parámetros de las funciones de demanda y costes, según el caso. Se establece además una condicional que establece como precio de mercado 0.001 en los casos en los que el precio resultante fuera negativo (esto es, en los casos en los que la producción total sea tan alta que fuerce el precio a la baja excesivamente).

Al final del juego podrás exportar los resultados a Word. Ten en cuenta que el Word generado mostrará los costes reales de los jugadores, no los costes que el NPC tiene en cuenta para tomar sus decisiones (éstos deberás intuirlos según su “personalidad”).

10. KIVYNOMICS PARA ANDROID

Además de la aplicación para escritorio expuesta a lo largo del presente trabajo, aprovechando las características de kivy, también se desarrolla una aplicación para dispositivos Android. El alcance de esta es mucho más reducido; está solamente centrado en el apartado JUGAR vs NPC, por el momento. El código no es exactamente el mismo que el utilizado por la aplicación de escritorio (hay ciertas adaptaciones como el tamaño de letra fijo y la pérdida de la posibilidad de exportar a Word) pero a grandes rasgos funcionan de la misma manera.

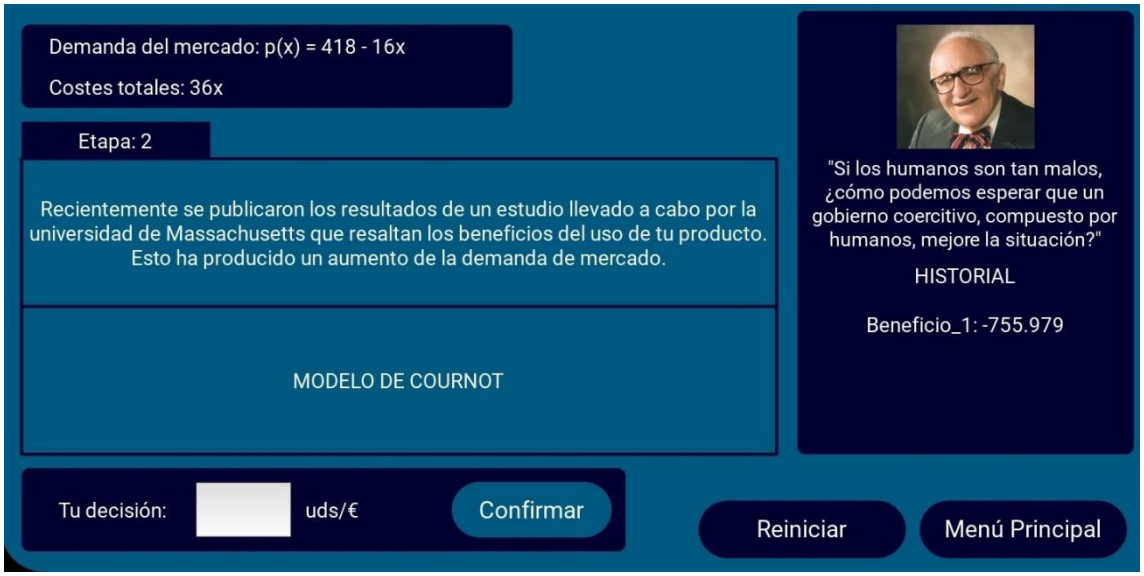


Ilustración 9: "Jugar vs NPC" de kivynomics en dispositivo Android.

La aplicación para Android consiste en 2 archivos de código: uno con código Python y otro con código Kivy. Para empaquetar la aplicación se ha utilizado Buildozer, una herramienta que permite crear un archivo apk a partir de unos parámetros determinados en un archivo de especificaciones (buildozer.spec) y el código de la aplicación en cuestión. (Buildozer Documentation (2023)). El código completo de la aplicación para Android está disponible en un repositorio de Github separado del que contiene la aplicación de escritorio. El archivo apk (a descargar para utilizar la aplicación) está almacenado en una carpeta de Google Drive, cuya referencia está en el apartado de Anexos. Asimismo, se incluye también una breve guía de instalación en el apartado de Anexos. En el apartado de conclusiones se incluye un comentario sobre el posible lanzamiento de la aplicación a la Play Store de Google o su implementación para dispositivos Apple.

10.1. Testeo

Dada la gran variedad de resoluciones de pantalla, relaciones de aspecto y otras especificaciones técnicas de todos los móviles con sistema operativo

Android, se hace preciso un testeo de la aplicación en varios dispositivos distintos. Este se ha llevado a cabo y, tras introducir los cambios pertinentes, la aplicación se ejecuta de forma exitosa en los siguientes dispositivos:

MARCA	MODELO	VERSIÓN ANDROID	RESOLUCIÓN	TAMAÑO DE PANTALLA	AÑO DE SALIDA
Samsung	Galaxy S10	12	1440x3040	6.1	2019
Samsung	Galaxy A20	9	720x1560	5.8	2019
Vodafone	Smart Prime 7	6.0.1	720x1280	5	2016
Oppo	A16s	12	720x1600	6.52	2021
Xiaomi	Redmi 7	10	720x1520	6.26	2019
Samsung	Galaxy S23 ultra	13	1440x3088	6.8	2023
Samsung	Galaxy Note 10	12	1080x2280	6.3	2019
Xiaomi	Pocophone F1	10	1080x2246	6.18	2018

Tabla 1: Dispositivos de testeo de kivynomics para Android. Datos de GSMARENA (2023).

Cabe destacar que la aplicación funciona sin problemas en el Vodafone Smart Prime 7, un móvil de 2016 con 1gb de memoria RAM y un procesador de 1.1 Ghz, es decir, un teléfono extraordinariamente antiguo cuyas especificaciones son superadas ampliamente por cualquier dispositivo actual. Huelga decir que el testeo no implica que la aplicación solamente se pueda utilizar desde uno de los dispositivos de estudio; la aplicación es universal para todos los móviles Android. El testeo certifica que la aplicación se ejecuta correctamente en los dispositivos mencionados, de lo que cabe deducir que se ejecutará sin problemas en la gran mayoría de teléfonos.

CONCLUSIONES

Tras casi nueve meses de trabajo he llegado a varias conclusiones que me parece relevante destacar:

- **Es importante delimitar claramente el alcance del proyecto desde un principio.** Es una de las cosas más difíciles (desde mi punto de vista) a la hora de desarrollar un proyecto por el estilo; las opciones a implementar son prácticamente infinitas y dejarse llevar por ellas puede desviar la atención del objetivo principal. Es importante señalar las funciones básicas de la aplicación y, una vez desarrolladas, centrarse en mejorar lo ya existente o crear nuevos apartados. La

teoría de juegos es un campo inmenso y la programación en Python puede producir casi cualquier cosa imaginable.

- **No existen tantos proyectos en este ámbito**. Si bien la teoría de juegos es un campo ampliamente estudiado, existen pocas implementaciones a nivel de programación al respecto, mucho menos implementaciones con interfaces gráficas y mucho menos en Python. Por ello, aunque pueda parecer desalentador la existencia de proyectos muy sofisticados, existe un amplio nicho aún sin explotar.

- **Python es un lenguaje de programación con un "alto ROI".** Se trata de un lenguaje de programación ideal para iniciarse en el mundillo que permite crear infinidad de scripts con diferentes funcionalidades. Con una sintaxis relativamente sencilla y numerosas bibliotecas, es una habilidad muy satisfactoria de adquirir que permite empezar a crear proyectos llamativos a los pocos meses de aprendizaje.

- **Posibles aportes a futuro.** Como comentaba en las conclusiones sobre el alcance del proyecto, por el límite de tiempo existente para desarrollar el trabajo, hay numerosas funcionalidades/aspectos que me habría gustado introducir/modificar pero debo dejar pospuestas para un futuro:

    - **La posibilidad de cambiar de idioma la aplicación (Español-Inglés).** Algo relativamente sencillo (bastaría con introducir un diccionario para cada idioma y acceder con la misma clave al valor correspondiente, habilitando un interruptor para seleccionar un diccionario u otro). No creo que llevara más de 5 horas de desarrollo.

    - **Más opciones en "Jugar vs NPC" y la app para Android.** Aquí se me ocurren numerosas ideas: habilitar la selección del número de etapas de una partida, habilitar el veto de algún modelo en cada partida, añadir nuevos eventos más complejos (como el ofrecimiento de creación de un cártel por el NPC) o incluso la introducción de nuevos personajes con personalidades más complejas. Se tratan de cosas relativamente fáciles de implementar, que supondrían añadir alguna clase más o cambiar algún aspecto de las funciones del juego. También se me ocurre la creación de partidas aleatorias por "semilla" de tal forma que introduciendo esa "semilla" los eventos y parámetros de la partida sean los mismos con el fin de comparar resultados con otras decisiones pasadas nuestras o de otras

personas. Con este mismo tono se podría crear un modo multijugador, sin embargo este sí sería un aporte más complicado de implementar. Por último, con el objetivo de aumentar el atractivo de este apartado (y la app de móvil), se podría introducir una tienda (con atuendos o desbloqueo de personajes) en la que se pudieran gastar los beneficios acumulados en las diferentes partidas. También se podría incluir un sencillo sistema de logros.

- **Resto de apartados.** Las posibles mejoras son evidentes: posibilidad de introducir funciones de costes cuadráticos en los modelos de Cournot y Stackelberg o aumento de la dimensión de las matrices en el apartado de Juegos Bimatriciales.
- **Ajustes.** Aparte del cambio de idioma, se pueden introducir más opciones de ajustes como la elección entre 2 paletas de colores para la aplicación (clara y oscura) u otras opciones de accesibilidad (paletas de colores para daltónicos…).
- **Creación de un paquete para dispositivos móviles Apple.** Esto sería relativamente sencillo ya que el código no tendría que ser cambiado y el propio Buildozer ofrece esta posibilidad. No lo he llevado a cabo porque no tenía apenas dispositivos con estas características a mi alcance para realizar pruebas.
- **Publicación de la app en la Play Store.** Simplemente con el objetivo de facilitar la descarga. Es probablemente una de las tareas que sí implemente personalmente en un futuro cercano.
- **Optimización del código en general.** Reducción de clases y métodos. Definición de funciones estáticas para los métodos de Cournot, Stackelberg y Juegos Bimatriciales, “traslado” a Python 3.11 y Kivy 2.2.0, …

REFERENCIAS BIBLIOGRÁFICAS

Bertrand, J. (1883) Review of “Theorie mathematique de la richesse sociale” and “Recherche sur les principes mathematiques de la theorie des richesses”. Journal des Savants, 499-508.

Carbonnelle, P. (2023), *PYPL PopularitY of Programming Language*. Fecha de consulta: 19/02/202. Recuperado de: https://pypl.github.io/PYPL.html

Cournot, A. (1838). Recherches sur les Principes Mathématiques de la Théorie des Richesses, Paris: Hachette.

GSMArena. (2023), *Phone finder.* Recuperado de: https://www.gsmarena.com/

Kivy’s Developers (2023), *Buildozer Documentation (release 0.11)*. Recuperado de: https://buildozer.readthedocs.io/_/downloads/en/latest/pdf/

Kivy’s Developers (2023), Kivy Documentation (release 2.2.0). Recuperado de: https://buildmedia.readthedocs.org/media/pdf/kivy/latest/kivy.pdf

Knight, V.(2023), *Nashpy Documentation (release 0.0.37)*. Recuperado de: https://nashpy.readthedocs.io/_/downloads/en/stable/pdf/

Ludwig von Mises Institute (2007), *Murray Rothbard in the 90's*, via Wikimedia Commons. Recuperado de: https://commons.wikimedia.org/wiki/File:MurrayBW.jpg

Nash, J. (1951). *Non-Cooperative Games. Annals of Mathematics*, 54(2), 286–295. https://doi.org/10.2307/1969529

Pérez Navarro, J., Jimeno Pastor, J. L., & Cerdá Tena, E. (2013). *Teoría de juegos.* (2ª ed.). Madrid: Editorial Garceta

Stackelberg H. V, (1934). *Marktform and Gleichgewicht*, Berlin: Springer-Verlag

The Gambit Project (2023), *Gambit Documentation (release 16.0.2)*. Recuperado de: https://gambitproject.readthedocs.io/_/downloads/en/latest/pdf/

TIOBE Software BV (2023), *TIOBE Index.* Fecha de consulta: 19/02/202. Recuperado de: https://www.tiobe.com/tiobe-index/

UC3M, *Juegos Dinámicos: Aplicaciones Económicas.* Recuperado de: https://www.eco.uc3m.es/docencia/new_juegos/doc/2.4%20Dina%CC%81micos%20aplicaciones.pdf

ANEXOS

Descarga del código fuente desde Github

El código fuente tanto de la aplicación de escritorio como de la aplicación para Android se encuentra disponible en un repositorio de Github en mi perfil. En los apartados de información de las ventanas principales de las aplicaciones se incluyen botones para acceder a mi perfil y ver los repositorios mencionados. Dejo también los siguientes enlaces aquí:

- Enlace a mi página de Github, desde donde se puede acceder a todos los repositorios de mi perfil: https://github.com/ENRSANJ
- Enlace al repositorio con el código para la versión de escritorio de Kivynomics: https://github.com/ENRSANJ/Kivynomics.git
- Enlace al repositorios con el código para la versión de Android de Kivynomics: https://github.com/ENRSANJ/Kivynomics_android.git

El repositorio de Github puede ser clonado como cualquier otro. Dejo enlace con la explicación de la documentación oficial de Github: https://docs.github.com/es/repositories/creating-and-managing-repositories/cloning-a-repository

Tras clonar el repositorio, será necesario instalar las bibliotecas requeridas para poder ejecutar la aplicación. Para ello, desde una terminal desde el repositorio de la aplicación, ejecutaremos el siguiente comando:

```
pip install -r requirements.txt
```

Además, por las particularidades de matplotlib como backend para kivy, tras instalar los requisitos debemos ejecutar también el siguiente comando:

```
garden install matplotlib --kivy
```

En Linux podríamos no tener permiso para ejecutar el comando por lo que deberíamos de dar permisos a nuestro usuario previamente ejecutando el siguiente comando:

```
chmod +x ruta_hacia_la_carpeta_garden
```

Cabe destacar que la aplicación está programada para ser utilizada con Python 3.10 y kivy 2.1.0 por lo que versiones posteriores de estas bibliotecas

podrían ser incompatibles. Por ejemplo, si se pretendiera usar Python 3.11 habría que utilizar la última versión de kivy (2.2.0).

Ahora sí, la aplicación está lista para ser ejecutada.

Además, se incluye una versión de la aplicación compilada con pyinstaller para Windows 10 en un directorio con un archivo ejecutable en la carpeta de drive:

https://drive.google.com/drive/folders/1T-A0Cij_Syigzmogbzj0r6OW5x4OvZQX?usp=sharing

Descargando la carpeta y haciendo click sobre el archivo “kivynomics.exe” se puede ejecutar la aplicación como cualquier otro programa de Windows, pudiendo incluso crear un acceso directo en el escritorio. De todas formas, esta forma de descarga y ejecución podría no funcionar (por falta de dependencias o problemas de detección de la versión de OpenGL) y es normalmente recomendable hacerlo mediante Github.

Por último, es recomendable leer siempre los archivos README por si hubieran cambiado algunas bibliotecas o la forma de instalar todas las bibliotecas del repositorio.

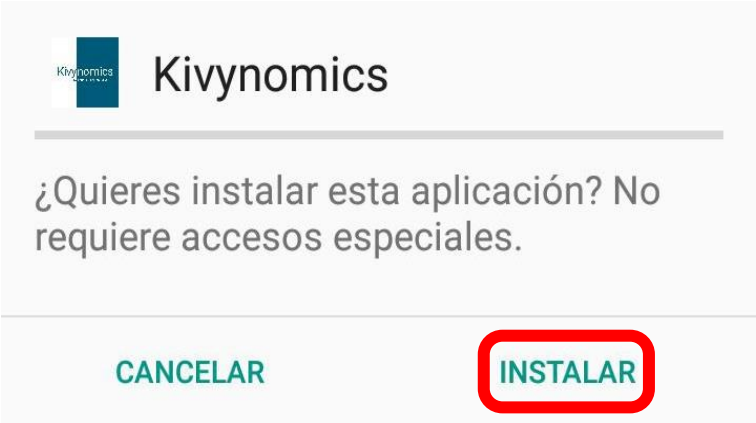
Instalación de la aplicación en Android

La aplicación en Android no ha sido publicada de momento en la Play Store por motivos de tiempo. Por lo tanto, la aplicación se puede instalar a través de un archivo apk. Dado que, aunque se trate de un procedimiento relativamente sencillo, no es común entre el usuario estándar, dejo una pequeña guía sobre la instalación:

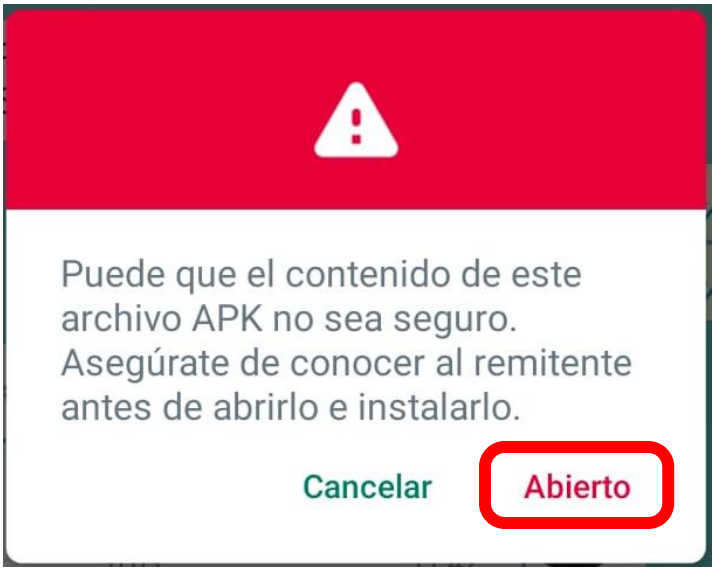
Desde el dispositivo en el que queramos instalar la aplicación debemos acceder a la carpeta de Drive en la que se encuentra el archivo a instalar. El enlace es el siguiente:

https://drive.google.com/drive/folders/1T-A0Cij_Syigzmogbzj0r6OW5x4OvZQX?usp=sharing

Descargamos el archivo “kivynomics.apk” y lo abrimos. Nos aparecerá una pantalla similar a la siguiente, donde pulsaremos en “Instalar”:



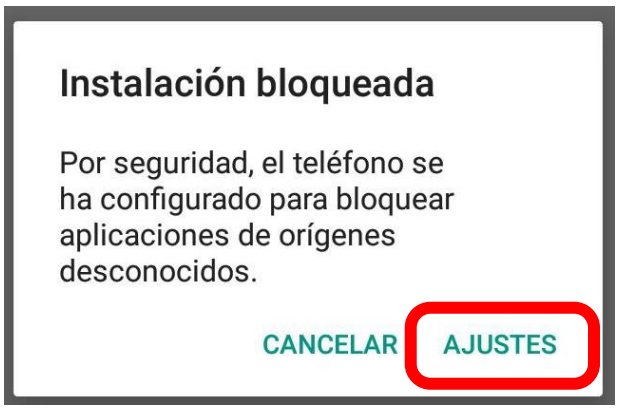
Es posible que previamente nos aparezca un aviso de seguridad puesto que no es una aplicación de la Play Store. Cualquier aviso de este tipo debe ser ignorado; el código fuente de la aplicación puede ser consultado en Github y no presenta ningún problema de seguridad.



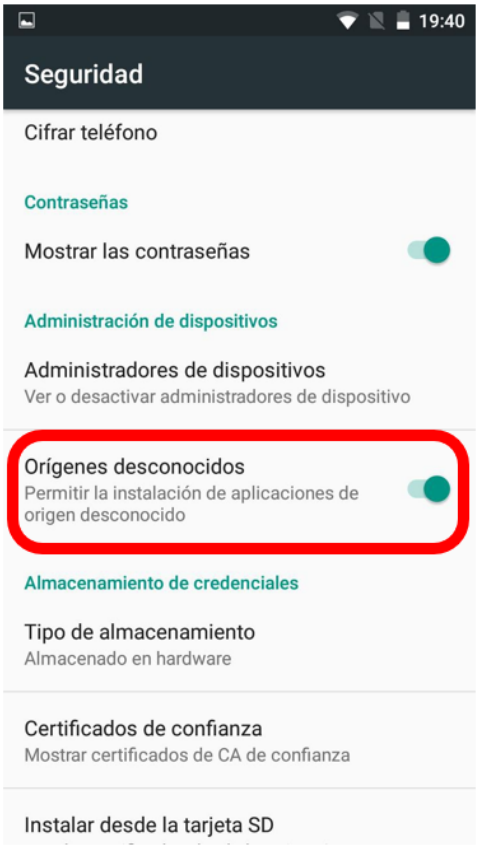
También podría aparecer un aviso de seguridad de Play Protect debido a que el desarrollador es desconocido:

En este caso, pulsaremos sobre “Instalar de todas formas”. Si por error pulsamos sobre “Entendido” se cancelará la instalación de la app.

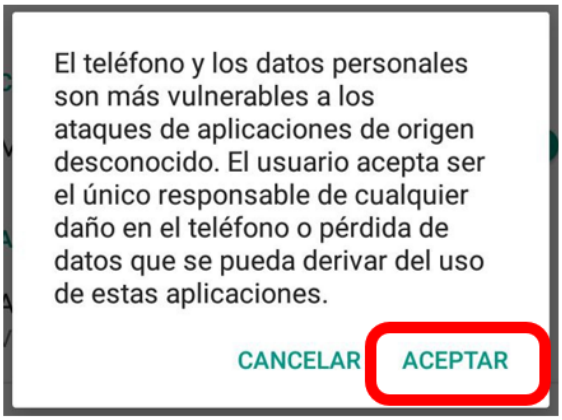
Si no tenemos habilitada la opción para instalar aplicaciones de orígenes externos, tras pulsar sobre instalar, nos aparecerá una pantalla similar a la siguiente:



Si es el caso, haremos click sobre “Ajustes” para habilitar la opción de orígenes desconocidos.



Al habilitar la opción nos aparecerá un aviso de seguridad que omitiremos pulsando en “Aceptar”.



Una vez habilitado “Orígenes desconocidos” volvemos a la pantalla de instalación para terminar de instalar la app. Al finalizar la instalación, la aplicación funcionará como cualquier otra: la podremos buscar por su nombre o icono en el cajón de aplicaciones y podremos crear un acceso directo en la pantalla de inicio.