**INFORME: “Laboratorio Virus”**

**Arián Benítez Ara**

**1. Enfoque Distribuido Utilizado**

El programa se basa en un modelo modular y concurrente para simular el funcionamiento de un robot aspirador en un "Laboratorio Virus". Para ello, se emplea el módulo estándar de Python **threading**, lo que permite distribuir las distintas tareas de forma simultánea. Cada componente importante del sistema se ejecuta en un hilo (thread) independiente:

* **Cálculo de áreas y tiempos de limpieza:**  
  Un hilo se encarga de calcular la superficie total a limpiar y estimar el tiempo teórico, manteniendo el proceso en ejecución para actualizaciones en tiempo real.
* **Lógica del robot (Roomba):**  
  Otro hilo gestiona el movimiento del robot en modo automático (utilizando un algoritmo BFS para encontrar la ruta óptima) o en modo manual, además de controlar colisiones con virus y enemigos.
* **Generación de virus y enemigos:**  
  Se han implementado hilos independientes para la dispersión de virus (ácaros) y enemigos (virus rojos). Estos hilos generan elementos de forma aleatoria dentro de las áreas definidas y se sincronizan mediante locks para evitar conflictos en el acceso a listas compartidas.
* **Incremento de radiación:**  
  Un hilo adicional aumenta progresivamente la radiación en el laboratorio y finaliza el juego cuando se alcanza un nivel crítico.

Este enfoque distribuido permite que cada componente realice su tarea de manera asíncrona, lo cual mejora la reactividad del sistema y la fluidez de la simulación, a pesar de tratarse de una aplicación basada en threads y no en procesos distribuidos a nivel de red.

**2. Elección de Herramientas**

En este proyecto se optó por el uso del módulo **threading** de Python en lugar de alternativas como **multiprocessing** o **concurrent.futures**. La elección se fundamenta en varios aspectos:

* **Naturaleza de la aplicación:**  
  La simulación se centra en tareas de I/O y en la actualización de la interfaz gráfica con Pygame, donde la sobrecarga asociada a la creación de nuevos procesos (como en multiprocessing) no resulta necesaria. El uso de threads permite compartir fácilmente variables y estructuras de datos (como el estado del juego) sin la complejidad de mecanismos de comunicación interprocesos.
* **Simplicidad y modularidad:**  
  La implementación con threads facilita la división en módulos independientes (por ejemplo, hilos para el robot, virus, enemigos, etc.), lo que favorece el mantenimiento y la escalabilidad del código. Además, el uso de locks garantiza la sincronización adecuada al acceder a datos compartidos.
* **Rendimiento en simulaciones interactivas:**  
  Dado que el programa no realiza tareas intensivas en CPU (como cálculos numéricos complejos), el Global Interpreter Lock (GIL) de Python no representa un problema crítico. La concurrencia basada en threads es suficiente para cumplir con los requerimientos de la simulación.

Aunque **concurrent.futures** ofrece una interfaz más moderna y sencilla para gestionar la concurrencia, en este caso el enfoque basado en threads resulta adecuado y se alinea con la naturaleza modular y altamente interactiva de la aplicación.

**3. Posibles Mejoras o Extensiones Futuras**

Existen diversas áreas en las que se puede mejorar o extender la funcionalidad del sistema:

* **Integración con sistemas de control de robots:**  
  Una extensión interesante sería integrar el programa con plataformas de control real de robots, como ROS (Robot Operating System). Esto permitiría que el algoritmo de navegación basado en BFS se aplique a robots físicos, facilitando la transición de una simulación a una aplicación práctica en entornos reales.
* **Manejo avanzado de errores y logging:**  
  Actualmente, la gestión de errores se realiza de forma básica (por ejemplo, mediante comprobaciones en cada iteración de los hilos). Se podría implementar un sistema de logging más robusto y centralizado para capturar excepciones, registrar eventos y facilitar el diagnóstico de fallos en tiempo real. Además, el uso de herramientas como **try/except** en cada módulo permitiría una recuperación más controlada ante errores inesperados.
* **Optimización y escalabilidad:**  
  Aunque el enfoque con threads es adecuado para esta simulación, en escenarios de mayor complejidad o cuando se requiera mayor rendimiento (especialmente en tareas CPU-bound), se podría considerar la utilización de **multiprocessing** o incluso la integración de soluciones basadas en **concurrent.futures** para gestionar pools de hilos/procesos. Esto ayudaría a optimizar el rendimiento y a aprovechar mejor los recursos del sistema.
* **Mejora en la inteligencia del algoritmo de navegación:**  
  El BFS actual se puede ampliar o mejorar utilizando heurísticas (por ejemplo, algoritmos A\*) para optimizar la búsqueda de rutas en entornos complejos, reduciendo la carga computacional y mejorando la capacidad de respuesta del robot en tiempo real.
* **Interfaz y experiencia de usuario:**  
  Ampliar la interfaz gráfica y la experiencia de usuario, integrando mejores visualizaciones, controles más intuitivos o incluso soporte para realidad aumentada, podría hacer que la simulación sea más interactiva y útil como herramienta educativa o de prueba.