ChatGPT

Auditoría de Código de KomBox

Errores Detectados

- **Inconsistencia en atributos de proyección:** En la función para habilitar la proyección de restricciones (`Simulator.enable_constraint_projection`), se asigna el atributo interno `_proj_max_iter`, pero luego la lógica de iteración usa `_proj_maxit` en lugar del nombre correcto. Esto provoca que el valor `max_iter` configurado por el usuario nunca se utilice (siempre se iteraría el número por defecto, 5). En el código se ve que se setea `self._proj_max_iter = int(max_iter)` [1] pero al iterar se usa `getattr(self, "_proj_maxit", 5)` [2]. La solución sería unificar el nombre del atributo (por ejemplo, usar siempre `_proj_maxit`) o corregir la referencia en la iteración para usar `_proj_max_iter`.

- **Entradas no conectadas pueden causar errores:** Algunos bloques asumen que todas sus entradas están conectadas y disponibles. Por ejemplo, el bloque `Spring.ode` accede directamente a `inputs["x"]` [3]; si el usuario no conectó la entrada `x` (o no la proporcionó como external), esto lanzaría un `KeyError`. Si bien la fase A de la simulación rellena entradas faltantes con ceros mediante `_complete_inputs_with_zeros` al calcular salidas [4], en la fase B (integración de estados) la implementación actual podría llamar a `ode` o `update` con entradas incompletas. De hecho, en `DCMotor.ode` se usó `inputs.get("tau_load", torch.zeros_like(i))` para evitar errores si falta esa entrada [5], pero otros bloques (Spring, Damper, etc.) no tienen esta prevención. Esto revela una inconsistencia: entradas no cableadas podrían provocar fallos en ciertos bloques. Lo ideal sería unificar el tratamiento de entradas faltantes (por ejemplo, siempre rellenar con ceros antes de llamar a `ode`/`update`, o exigir explícitamente todas las conexiones).

- **Flush de datos al usar NPZChunkRecorder:** Cuando se usa `NPZChunkRecorder` para grabar en disco por chunks, si la simulación termina con un chunk incompleto, esos datos quedan en memoria y **no se vuelcan al archivo** a menos que el usuario llame manualmente a `recorder.close()`. El método `close()` llama a `_flush_chunk()` para escribir el último bloque [6], pero actualmente `Simulator.simulate` no invoca `close()` automáticamente al terminar. Esto puede llevar a pérdida de datos si el usuario olvida cerrar el recorder. Sería conveniente que la simulación llamara a `recorder.close()` al finalizar (si el recorder tiene ese método) o que el propio recorder manejase la escritura final en `__exit__` (context manager) para no depender del usuario.

- **Parámetros matriciales no plenamente soportados en grid:** La API de parámetros asume que cada parámetro tiene una dimensión "ancho" fija D. Para parámetros escalares o vectores funciona bien, pero en casos de matrices como los de `StateSpace` (A, B, C, D), la lógica de grid puede fallar. Por ejemplo, la matriz A se declara con shape (n,n); internamente se almacena tal cual (p.ej. 3×3). El método `apply_parameter_grid` trata de expandir parámetros creando tensores 2D de shape (B,D) [7], donde D es el número de columnas declarado. En el caso de A, `D = n` (e.g. 3), no *n*×*n*. Si quisiéramos variar A en una grid de valores, probablemente *fallaría* porque `_to_row_vector` espera secuencias de longitud D=3 [8], no matrices 3×3 completas. Esto indica que actualmente no es posible variar fácilmente matrices completas con la malla de parámetros. No es un error crítico (podría considerarse una limitación conocida), pero conviene

documentarlo o mejorar la función de grid para soportar **parámetros multi-dimensionales** (por ejemplo, aceptando tensores 2D en la lista de valores y adaptando `_to_row_vector` para matrices).

- **Cierre de lazo algebraico y selección de *solver*:** Cuando el modelo tiene lazos algebraicos puros (ciclos directos de alimentación), el simulador lanza un error si se intenta usar un solver explícito (por bloque). De hecho, en `Simulator.step` hay un chequeo que produce `ValueError` si `model._has_pure_algebraic_cycle` es True y `solver.is_global` es False [9]. Aunque esto previene resultados incorrectos, la forma de notificarlo al usuario podría mejorarse. Actualmente el mensaje sugiere usar un solver implícito/global [10], pero dado que el usuario "no debería tener que saber de solvers", sería mejor que el framework **lo manejara automáticamente**. Este punto se amplía en las mejoras, pero lo mencionamos aquí porque, tal como está, puede sorprender al usuario con una excepción que no sabrá resolver sin modificar opciones internas.

Propuestas de Mejora

- **Experiencia de uso simplificada (tipo Simulink):** Sería beneficioso proveer atajos para que el usuario configure y ejecute simulaciones en menos pasos. Por ejemplo, añadir un método de alto nivel como `Model.simulate(...)` que internamente haga `build()`, `initialize()` y ejecute el `Simulator.simulate()`, retornando directamente los resultados. Esto permitiría correr una simulación con una sola línea de código una vez definidos los bloques y conexiones. Igualmente, se podría ofrecer una **configuración por defecto de solver y dt**: actualmente el usuario debe elegir un `dt` y número de pasos/tiempo total. Para asemejarse a Simulink, KomBox podría seleccionar automáticamente un `dt` razonable (o usar por defecto un solver adaptativo) si el usuario no lo especifica. Por ejemplo, si no hay bloques discretos con período definido, se podría default a un solver variable (tipo RK45 o `torchdiffeq` dopri5 con tolerancias) evitando que el usuario deba conocer los detalles. Estas mejoras acercarían la herramienta a la filosofía "plug-and-play" de Simulink, donde el usuario solo coloca bloques y ejecuta la simulación sin configurar aspectos numéricos a menos que quiera.

- **Selección y configuración automática de solver:** Relacionado con lo anterior, la función `auto_solver_for(model)` ya elige un solver implícito (Trapezoidal + Newton-Krylov) si detecta lazos algebraicos [11], lo cual es excelente. No obstante, habría que asegurar que esto siempre cubra los casos necesarios. Si por alguna razón el solver implícito no está disponible (p.ej. falta import porque no se instaló alguna dependencia), la función cae a Euler explícito [12]. En ese escenario de *fallback*, sería conveniente activar automáticamente la **proyección de restricciones post-paso** en el simulador (equivalente a llamar a `enable_constraint_projection(True)` internamente) en vez de simplemente proceder con Euler y luego fallar con el error de lazo algebraico. De esta manera, KomBox podría resolver lazos algebraicos mediante iteración correctora aun con solvers explícitos (sacrificando eficiencia pero ganando robustez para el usuario novato). En resumen, mejorar la **robustez automática**: si hay lazos, intentar solver implícito; si no se puede, activar proyección/iteraciones correctoras de modo transparente al usuario.

- **Tratamiento uniforme de entradas no conectadas:** Para hacer la herramienta más amigable, se propone que los bloques continuous/discrete no fallen por entradas faltantes. Actualmente, la clase base `Block._complete_inputs_with_zeros` rellena con ceros cualquier entrada no proporcionada [13], pero esto solo se usa en `_expose_outputs` (fase A) y no necesariamente en la integración de estados en fase B. Una mejora sería llamar también a

`_complete_inputs_with_zeros` dentro de los métodos `_advance` antes de integrar, garantizando que incluso si el usuario no conectó alguna entrada, el bloque vea un tensor de ceros en su lugar. Alternativamente, documentar y adoptar la convención de que un bloque debe usar `inputs.get("nombre", default)` al leer sus entradas (como hizo DCMotor). Estandarizar esto evitará errores silenciosos y reproduce el comportamiento de herramientas gráficas, donde una entrada no conectada equivale a cero. Esta mejora de usabilidad permitiría que, por ejemplo, un usuario pueda dejar sin conectar una entrada de perturbación (p.ej. `tau_load` en DCMotor) asumiendo que será cero por defecto, sin necesidad de código adicional.

- **Extender capacidad del solver RK45 (paso adaptativo):** La implementación actual de `RK45Solver` calcula el error de truncamiento local (`last_error`) pero no ajusta el tamaño de paso [14] [15]. Para estar "a la altura de Simulink", sería deseable soportar integración de paso variable automáticamente. Se podría implementar un mecanismo simple de control de paso: por ejemplo, en `Simulator.step` detectar si el solver actual tiene atributo `last_error` y, en tal caso, rehacer el paso con `dt` menor si el error excede cierta tolerancia (y quizá aumentarlo si el error es muy pequeño). Aunque esto añade complejidad, ofrecer un solver adaptativo tipo RK45 mejoraría la precisión sin que el usuario tenga que experimentar manualmente con `dt`. Alternativamente, documentar cómo el usuario puede usar `TorchDiffEqSolver` (basado en `torchdiffeq`) para lograr adaptatividad global. En todo caso, brindar **opciones de paso adaptativo** al nivel de la API acercará la experiencia a la de Simulink, donde ODE45 (Dormand-Prince) por defecto ajusta el paso automáticamente.

- **Mejoras en el registro y visualización de resultados:** Dado que la filosofía es facilitar al máximo la obtención de resultados, se podría considerar **registrar salidas por defecto** o simplificar la recolección de datos. Por ejemplo, si el usuario no pasa un recorder a `simulate()`, el framework podría por defecto almacenar todas las salidas externas en memoria (equivalente a un `MemoryRecorder` de las señales expuestas). Así, tras la simulación, el usuario tendría acceso a un diccionario de resultados sin pasos adicionales. Otra mejora pequeña: en el recorder de memoria, los nombres de claves para estados con slice podrían abreviarse. Actualmente produce etiquetas como `"bloque.state:slice(0, 2, None)"` [16] [17] si se pidió grabar un slice; sería más limpio formatearlo como, por ejemplo, `"bloque.state:0-1"`. Esto es cosmético pero ayuda en la presentación profesional de los datos. Además, como se mencionó en errores, **asegurar el flush de NPZChunkRecorder** automáticamente al final aporta confiabilidad en entornos de producción.

- **Soporte ampliado para variaciones paramétricas:** KomBox ya permite ejecutar simulaciones en batch variando parámetros (mallas cartesianas). Para equipararse a herramientas profesionales, podría ampliarse esta funcionalidad. Por un lado, resolver el mencionado soporte a matrices: permitir que `apply_parameter_grid` maneje matrices completas (quizá detectando si el valor es un tensor 2D del mismo shape que el parámetro original). Por otro lado, se podría incorporar *sweeps* no cartesianos o distribuciones aleatorias de parámetros para **análisis de sensibilidad** Monte Carlo, etc. Otra idea es ofrecer utilidades para barrer un rango temporal (simulaciones consecutivas aumentando el tiempo) o loops que automaticen pruebas de convergencia en `dt`. Estas capacidades adicionales harían de KomBox una herramienta más poderosa para exploración de modelos, acercándola a la versatilidad de Amesim en estudios paramétricos.

- **Refinamientos de código y documentación profesional:** Para posicionar a KomBox a la altura de herramientas industriales, es importante pulir detalles de consistencia y claridad:

- *Lenguaje y nombrado:* El código mezcla nombres y mensajes en español e inglés. Considerar unificar el idioma del código (p.ej. todo en inglés, con documentación bilingüe si se desea). Por ejemplo, excepciones como `"Bloque 'X' no existe"` frente a `FloatingPointError("NaN/Inf")`. Una base de código consistente luce más profesional y facilita la adopción por comunidades globales.
- *README y ejemplos actualizados:* Dado que el README está desactualizado, conviene revisarlo para reflejar la API actual, con ejemplos sencillos de uso (crear un modelo simple, conectarlo y simular). Incluir comparativas de sintaxis con Simulink ("si en Simulink harías X, en KomBox se hace Y") podría atraer usuarios familiarizados con esa herramienta.
- *Tests automáticos:* Incluir una suite de pruebas unitarias/regresión para el core (por ejemplo, probar que un modelo simple masa-muelle amortiguador produce la salida esperada) incrementará la confiabilidad al hacer mejoras futuras. Esto es estándar en herramientas profesionales.

- *Mensajes de error más explicativos:* Aunque en general los errores están bien descritos, siempre se puede mejorar pensando en el usuario novel. Por ejemplo, si falta llamar a `initialize()`, el error actual es `"Model: no hay estados. Llama a initialize() primero."` – esto está bien, pero podría atraparse antes (cuando el usuario llama a `Simulator(m)` sin inicializar) y arrojar un mensaje más guiado. Pequeños refinamientos así contribuyen a una mejor UX.

- **Visualización y GUI (a mediano plazo):** Si bien *"salvando las distancias de la interfaz gráfica"* indica que de momento no se espera un entorno gráfico completo, podría evaluarse alguna solución intermedia para visualización de esquemas. Por ejemplo, ofrecer un método que genere un diagrama de bloques a partir del modelo (usando Graphviz o similares) ayudaría a depurar y a presentar el modelo de forma comprensible, sin construir una GUI desde cero. Del lado de la simulación, se podría integrar algún visor sencillo de resultados – por ejemplo, una función para plotear rápidamente las series temporales grabadas (similar a cómo Simulink tiene scopes). Estas utilidades no son críticas, pero mejoran la **comodidad del usuario** y la percepción de "herramienta completa". En última instancia, si el proyecto crece, desarrollar un front-end gráfico (quizá sobre un notebook Jupyter interactivo o una aplicación web) haría a KomBox mucho más competitivo frente a Simulink/Amesim, atrayendo a usuarios que prefieren evitar la programación textual.

En conclusión, el código de KomBox está bien estructurado y ofrece ya muchas funcionalidades avanzadas (soporte de PyTorch, solvers personalizables, grabadores, etc.). Corrigiendo los pequeños errores detectados y enfocando las mejoras en la **simplicidad de uso** y **robustez automática**, el proyecto se acercará a la visión de una herramienta profesional al nivel de Simulink/Amesim, pero en entorno Python. Cada propuesta anterior busca reducir la carga sobre el usuario final – que este pueda modelar y simular sistemas dinámicos complejos con mínima configuración manual, confiando en que KomBox gestione los detalles técnicos de forma inteligente. Así, KomBox podría distinguirse como una biblioteca potente pero *user-friendly* dentro del ecosistema Python.

---

1 2 9 10 simulator.py
file://file-PqT8DypQ4NJBd8JD6v1bDJ

3 mechanical.py
file://file-7jgcqkKGsGUqvj3FWbqjbj

4 13 block.py
file://file-2qzmw6NYxvN9PWzVotXTS4

5 electrical.py
file://file-CURsvPrVT6xKMguW1QMStR

6 16 17 recorders.py
file://file-2RFhEsTEvwGeKKvSoviH2e

7 8 utils.py
file://file-5CGoHpTd8Dgnb2gMYBzoqW

11 12 14 15 solvers.py
file://file-2ozu1DwGP3QVEQKJJBuzC8