ICOM2022 – 2do Parcial

1 de diciembre de 2022

Notas:

1. Uso de prácticos: **se pueden utilizar los trabajos prácticos propios realizados.**
2. Utilización de la WEB: **solo para referencias**

Problema 1: Operaciones con árboles binarios

Se dice que dos árboles binarios son semejantes si tienen el mismo número de nodos y los valores de los nodos del primer árbol son los mismos valores de los nodos del segundo árbol, sin importar la relación de parentesco entre ellos.

Una forma en que se podría verificar la semejanza entre 2 árboles binarios es realizar una serialización de los nodos del árbol pasando los valores a un vector y comparar los vectores previamente ordenados.

Dado el esqueleto de implementación de árboles binarios ordenados dado en **arbol.cpp**, implemente la función:

```
bool arboles_semejantes(Tree t1, Tree t2);
```

Otro uso muy frecuente de árboles binario es imponerle orden y transformarlo en diccionarios binarios. Si el diccionario binario está balanceado (los elementos se distribuyen más o menos equilibradamente en las diferentes ramas), se optimiza su eficiencia de búsqueda.

Se solicita implementar una función que transforme un árbol binario (posiblemente sin orden) en un diccionario balanceado.

```
Tree balancear(Tree t);
```

Hint: Imagine un proceso de serialización entre medio.

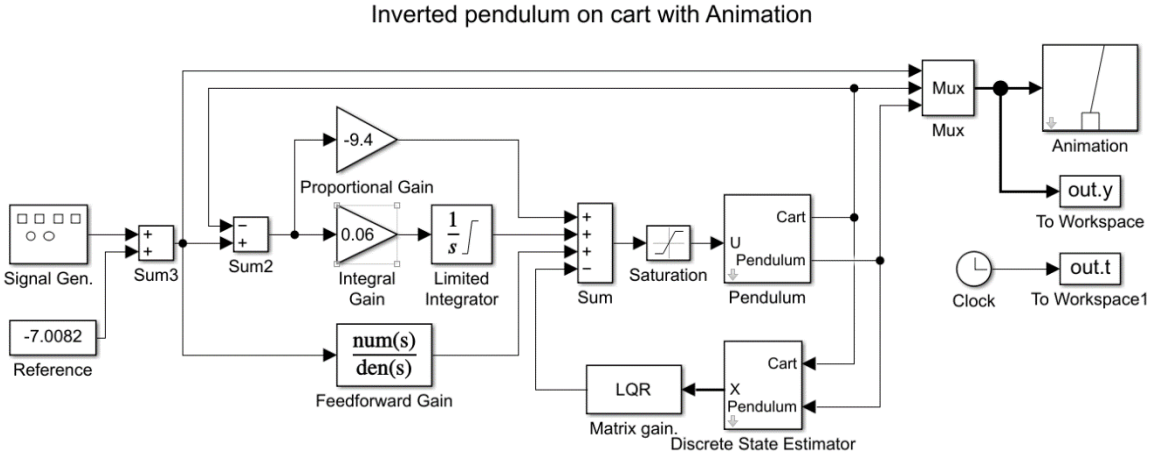
Problema 2: Matrices ralas

Las matrices ralas (sparse matrix) se caracterizan por ser de grandes dimensiones y muy pocos elementos no nulos (que son los únicos elementos almacenados).

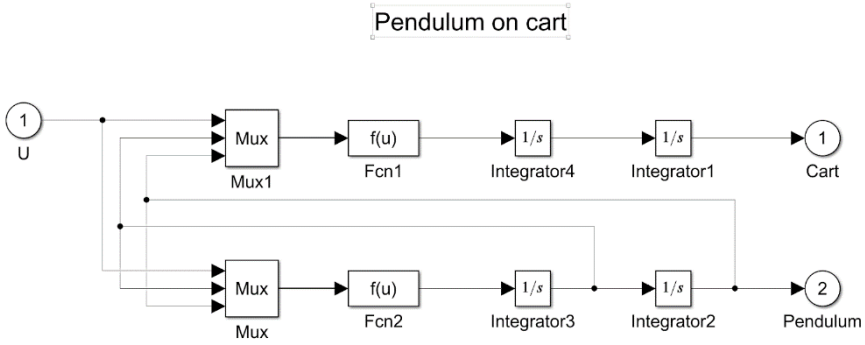
Se solicita terminar de diseñar e implementar el UDT SparseMatrix cuyo esqueleto se encuentra en **SparseMatrix.cpp** en donde se utiliza un unordered_map para mantener los elementos no-nulos.

Problema 3: Loops algebraicos

Un diagrama de simulación como el que se muestra en la siguiente figura se compone de bloques (todos con distinto nombre) que están conectados entre sí a través de señales.



A su vez, algunos bloques (**subsystem** o **subsistema**) pueden contener un diagrama, como es el caso del bloque **Pendulum** del diagrama anterior cuyo contenido se muestra en la figura siguiente. En estos casos, los nombres de los bloques internos llevan como prefijo el nombre del bloque que los contiene (por ejemplo, **Pendulum/Mux**).



El sistema de simulación determina el orden en que se ejecutarán todos los bloques, y en un esquema de tiempo discretizado, para cada paso de tiempo los ejecuta en ese orden. Para cada bloque se calculan las salidas (si las hubiera), se ingresan las entradas (si las hubiera) y se actualiza el estado (si lo hubiera). Existen dos tipos de bloques: los bloques cuya salida en un paso de tiempo depende directamente de las entradas en ese mismo paso (denominados **direct feedthrough** o **alimentación directa**) y los que no.
Si se sigue el flujo de una señal desde la salida de un bloque, puede ocurrir que termine ingresando como entrada en el mismo bloque, formando un **loop** o bucle. Cuando todos los bloques en ese loop son direct feedthrough, ocurre un **loop algebraico** que imposibilita al sistema determinar el orden de ejecución de los bloques.

En el archivo **`loop_algebraico.cpp`** hay un esqueleto de un programa que permite detectar loops algebraicos. Los archivos bloques_sin.txt y bloques_con.txt contienen los nombres de todos los bloques de un diagrama, junto con la información de si ese bloque es direct feedthrough (1 o 0). El diagrama de **`bloques_con.txt`** contiene loops algebraicos y el de **`bloques_sin.txt`** no.
El archivo **`matriz_adyacencia.txt`** contiene una matriz que indica la conexión entre los bloques. Si el elemento (i,j) de esa matriz es un 1, entonces la salida del bloque i se conecta a la entrada del bloque j.
Complete el programa **`loop_algebraico.cpp`** implementando todos los métodos marcados con **TODO**.