# main

April 17, 2023

## 1 FARYADELL SIMULATION FRAMEWORK

The `Faryadell Simulation Framework` is a simple tool that facilitates simulations expeditiously and effortlessly, using invaluable instruments for modeling linear and nonlinear dynamic systems, irrespective of their time dependence or independence. This formidable instrument is capable of executing `Dynamic Mathematical Models` and is advantageous for control engineering applications, estimation, and prediction. Moreover, it is applicable to Machine Learning and AI domains, including neural networks. Researchers who wish to delve into the realm of dynamic and control systems will find this package to be an invaluable resource. Enjoy!

"*Successful people are those who can build solid foundations with the bricks others throw at them.*"

```
Creator: Abolfazl Delavar
Website: https://github.com/abolfazldelavar
```

### 1.1 Requirements

All external **dependencies** utilized in the project should be documented in this section. To ensure comprehensive access to libraries, any additional libraries can be added as objects (using `struc()`) for improved control and management. Additionally, the `coreLib.py` file contains two libraries, `clib` and `plib`, which provide a variety of useful functions for tasks such as drawing.

[ ]:
```
# To generate plots in an independent window outside this page
# %matplotlib qt

# Dependencies
from core.lib.pyRequirement import *
from core.lib.coreLib import *
```

### 1.2 Custom Functions

If you need to define any custom functions for your project, this is the ideal location to do so. This section serves as a repository for pre-defined functions that can be accessed and utilized throughout the entire project.

[ ]:
```
def test():
    '''
    DocString `is` a text to help using this function.
    '''
```

```
    pass
```

### 1.3 Parameters

This segment confers the prerogative to establish **invariable** quantities, typically employed with a singular value throughout the entire project. It is imperative to note that they must be delineated as a component of the `params` variable, which constitutes a `struct` collection.

[ ]:
```
params = struct()

# Set the time-step and simulation time as needed.
# Note: These variables may need to be changed arbitrarily.
params.step = 0.0001 #(double)
params.tOut = 6      #(double)

# Calculate the number of steps required for the simulation.
# Note: This variable is dependent on the time-step and simulation time, and␣
 ↪should not be changed.
params.n = int(params.tOut/params.step) #(DEPENDENT)

# Specify the folders for input data, output data, and logs.
params.dataPath = 'data'
params.loadPath = params.dataPath + '/inputs'  #(string)
params.savePath = params.dataPath + '/outputs' #(string)

# Determine whether to save a diary after each simulation and specify the␣
 ↪directory and name of the diary file.
# Note: The diary file name is normally created using the current time, but can␣
 ↪be changed arbitrarily.
params.makeDiary = True   #(logical)
params.diaryDir  = 'logs' #(string)
params.diaryFile = clib.getNow(1, '-') #(string)

# Set the amount of time (in seconds) between each string printed on the␣
 ↪command prompt.
params.commandIntervalSpan = 2 #(int)

# Determine whether to save data with a unique name based on the time of saving.
params.uniqueSave = False #(logical)

# Specify the default image format for saving.
# Note: Allowed formats are "jpg", "png", and "pdf".
params.defaultImageFormat = 'png' #(string)

# Add any additional parameters as needed ~~~>
```

## 1.4 Signals

This section is designated for the definition of any signals and array variables.

[ ]:
```
signals = struct()

# Generate a time vector for the simulation using the specified time-step and␣
 ↪simulation time.
signals.tLine = np.arange(0, params.tOut, params.step)

# Insert signal data here ~~~>
```

## 1.5 Models

Dynamic objects and those that are not as elementary as an array must be delineated as a component of the `models` variable. This includes objects such as estimators and controllers.

[ ]:
```
models = struct()

# Add your desired models to the struct ~~~>
```

## 1.6 Main

### 1.6.1 Oscilloscope set up

Firstly, `scope` objects that are instrumental in observing signals should be defined herein.

[ ]:
```
# Insert signal trackers here ~~~>
```

### 1.6.2 Simulation

The principal function of the project, indubitably, can be identified as the `simulation` function delineated below. In this segment, given the accessibility of all variables (`params`, `signals`, and `models`), you possess the capability to code the primary objective of this project herein. It is imperative to note that there exists a loop entitled `Main loop`, which can be employed as a time step loop, albeit its utilization may not be requisite in numerous projects.

[ ]:
```
def simulation(params, signals, models):
    # This function is the main code for your simulation, containing a␣
 ↪time-loop and utilizing model blocks and signals.
    # The order of the input parameters should be (Parameters, Signals, Models).
    # You can initialize before the main loop and finalize after it if needed.

    ## Initial options
    # Call the sayStart method from clib class with params as input and assign␣
 ↪it to st variable
    st   = clib.sayStart(params)
    # Create a trigger to report steps in command
    trig = [st, params.commandIntervalSpan, -1]
```

```
    ## Main loop
    for k in range(0, params.n):
        # Display the iteration number on the command window using disit method from clib class
        trig = clib.disit(k, params.n, trig, params)

        # Put your codes here ~~~>

    ## Finalize options
    # Report the simulation time after running has finished
    clib.disit(k, params.n, [st, 0, trig[2]], params)
    # Call the sayEnd method from clib class
    clib.sayEnd(st, params)
    # Return the output as a list of params, signals, models
    return [params, signals, models]
```

## 1.7 Execution

To run the project, the subsequent code snippet is furnished. In elementary projects, a single execution may suffice, whereas, in more intricate ones, multiple simulations may be necessitated. To accomplish this, loops and other discretionary techniques can be employed to invoke the `simulation` function with altered inputs such as `params`, etc.

[ ]:
```
[params, signals, models] = simulation(params, signals, models)
```

### 1.7.1 Analysis

Should you necessitate an evaluation of your project's timing and an identification of the most time-intensive segments of your program, you may uncomment the contents of this section and scrutinize your coding performance. It is pertinent to note that the preceding code block should be commented.

To utilize this feature in `Command Prompt`, execute the subsequent commands to generate and display the results: * `python -m cProfile -o logs\profiler\cprofiler.prof main.py` * `snakeviz logs\profiler\cprofiler.prof`

[ ]:
```
# # Run the simulation and save the results into a file
# simulationorder = 'simulation(params, signals, models, lib)'
# profilename = params.diaryDir + '/profiler/' + params.diaryFile + '.prof'
# cProfile.run(simulationorder, profilename)

# # Visualize the profiling results using snakeviz
# %loadext snakeviz
# %snakeviz profilename
```

### 1.8 Illustration

Should you necessitate the exhibition of the results procured, this section can be employed to facilitate superior organization.

[ ]:
```
## Initialize the variables and parameters
n = params.n                 # number of elements
nn = np.arange(0, n)         # create a vector from 0 to n-1
tLine = signals.tLine[nn]    # create a time-line vector with length n
plib.initialize()            # initialize the library

## Write your codes here ~~~>
```

### 1.9 Preservation

Utilize the subsequent section to store data.

[ ]:
```
## Write your codes here ~~~>
```