Title: Schedule table generation for Load balancing in time-stepped simulation and Real-time scheduling of deferrable electric loads.

Full Name: Emerson Raja Jose Peeterson

Matric No.: A0195021N

Mobile No.: 82894340

Paper 1: Real-time load balancing scheduling algorithm for periodic simulation models

Abstract

In this simulation we develop a scheduling table for scheduling independent and periodic tasks. Scheduling of a static and dynamic model set in a time-stepped simulation is carried out using the time-stepped load balancing scheduling (TLS) algorithm. The algorithm ensures correct order of execution so that models meet deadlines. It is also able to balance the load during model addition or deletion to spread out tasks evenly across sub steps of periods and to balance the computation times and improve real-time reliability.

Short Introduction of the Problem, model, and objective(s)

In Combat simulations, an entity like a tank will have models such as radar sensors that are periodically scanning for enemies. As models maybe destroyed or added during combat, we combine offline scheduling and online adaptive methods to create a task schedule table. Tasks to run at each time step is decided by the TLS algorithm and updated in the scheduling table at run-time. Conventional algorithms like EDF calculate and sort deadlines online and this incurs a high overhead. Sometimes when models are created, or existing models are deleted, priority changes in EDF can occur and task computation times may exceed beyond its current period causing jitter. Jitter count, which is the variability in starting time or completion time of the same periodic task at different periods is higher with EDF when the number of model sets increases. A better scheduler is characterized by a smaller jitter count and fewer resource requirements to handle it. The issues of missing deadlines and infeasible schedules is tackled by improving real-time reliability.

Description of Input data / Parameters in detail

The input data is from table 1 of [1]. As the data is defence related it is confidential data. Requests for more specific data were declined by the authors. But the data provided is enough to validate the claims by the authors. A task is a periodic model and it is simply characterised by two parameters, its period, and its worst-case execution time.

The 'Original_model_parameters.csv' file contains the distinct models with periods and their execution times. Zero entries (Not model) are added to maintain a rectangular shape (same number of columns for each distinct period) of the task set as required by the pandas data frame. To add two models of the same period with same execution time just add them twice.

The 'model_addition.csv' and 'model_deletion.csv' files contains models belonging to a period and execution times that need to be added to or deleted from the schedule table.

Algorithm description

main.py

This is the main program that calls helper dependencies to implement TLS scheduling algorithm.

In ‘main.py’, model parameters are read from ‘Original_model_parameters.csv’, ‘model_addition.csv’ and ‘model_deletion.csv’ files and parameters period and computation times are also sorted using ‘extract_data.py’.

The ‘model_deletion.csv’ file includes an extra parameter called ‘Model no.’ which can be read from the generated schedule table. This is required to uniquely identify models that may have the same period and execution time but are distinct.

The maximum computation time is calculated from sorted computation times in ‘extract_data.py’.

Time step is calculated by time_step.py

The scheduling period, Ts repeats every major cycle. Major cycle is the largest period in the model set. Total steps is the number of steps inside the major cycle.

The TLS algorithm initially generates a static schedule table for the models. In the event of model addition, the schedule table is dynamically changed during runtime. During model addition load balancing is implicitly performed by placing the new model in the time step with the greatest idle rate. During model deletion the schedule table is checked if rebalancing is required to redistribute models from congested time steps through load balancing.   and It is implemented in tls_algorithm.py

Finally, we visualize the schedule table through ‘visualize.py’

Helper dependencies

extract_data.py

1) Sort the periods in ‘Original_model_parameters.csv’ and ‘model_addition.csv’ in ascending order in order to identify the smallest period which is the upper bound for the time step determination.
2) Sorts the computation times in ‘Original_model_parameters.csv’ and ‘model_addition.csv’ in ascending order in order to identify the largest computation time which is used as the lower bound for the time step determination.
3) The ‘detect_new_period(self):’ function checks for new distinct periods during model addition as this will require time step to be recalculated.

You can only add models of periods that exist in the ‘Original_model_parameters.csv’. If it does not exist, please add to ‘Original_model_parameters.csv’ file instead of the ‘model_addition.csv’ file

For model deletion, multiple models with the same period and/or computation times with distinct model numbers (Model no.) as it appears on the schedule table can be listed as rows in ‘model_deletion.csv’

time_step.py

In the ‘calc_time_step(self):’ function, time step is calculated based on upper and lower bounds given above and the condition $T_i = 2^t \times T_{step}$, $(t = 0,1,2, \dots.)$. In the program largest value of t is limited to 10. There can be more than one T_{step} within the bounds above that can satisfy the condition so we pick the largest time step so that number of time steps checked during load rebalancing is reduced.

tls_algorithm.py

In 'static_schedule(self):' the schedule table has time steps as rows (Total steps) and distinct periods as columns. 'self.idle' is the sum of the Idle rates over all the periods at each time step. It is intilaized to 1 for all time steps in the very beginning.

Every period has its computation times updated in the schedule table through 'generate_sch_table(self,p,comp_times_sorted,i):' function. Inside this function, we assign models to time steps with the largest idle rate and also update their multiples inside the major cycle. If any idle rates are negative an error is thrown, and the entire program is exited.

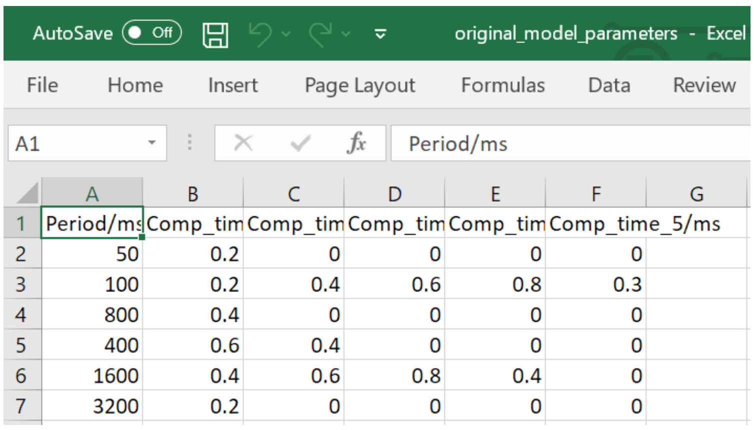
The function 'model_addition(self, new_per_flag, sorted_add_periods, sorted_add_comp_times):' first checks if a model with a period other than one of existing periods has been added as this will require recalculation of the time step. Then adds the new computations to the schedule table.

The function 'model_deletion(self, per_col,comp_col,m_col):' first identies the time step, r inside the first period time steps that contains the model to be deleted. The idle rate is increased for this time step and label in the schedule table is removed. Next disparity for any time step that is overloaded or underloaded after model deletion is checked using 'self.check_rebalancing_req(self, i,local_steps):'. If load rebalancing is required the function 'load_rebalancing(self,p,i,local_steps,max_sstep_idx, min_sstep_idx):' is called. Load refers to the computation time. Load balancing within the period is equivalent to globally rebalancing. Load rebalancing redistributes some models from sub step (time step within a period) with maximum total computation time to sub step with minimum computation time. The details are provided in code comments.

Simulation effort & Results discussions

After creating the 'load_bal' python environment as in 'Readme_paper1.txt' Please run 'main.py' to simulate. Initial Static Schedule table for model parameters in table 1 is shown in figure 1 below.

Table 1: combat simulation models task set


AutoSave Off
original_model_parameters - Excel
File
Home
Insert
Page Layout
Formulas
Data
Review
A1
Period/ms

	A	B	C	D	E	F	G
1	Period/ms	Comp_tim	Comp_tim	Comp_tim	Comp_tim	Comp_time_5/ms	
2	50	0.2	0	0	0	0	
3	100	0.2	0.4	0.6	0.8	0.3	
4	800	0.4	0	0	0	0	
5	400	0.6	0.4	0	0	0	
6	1600	0.4	0.6	0.8	0.4	0	
7	3200	0.2	0	0	0	0	

Figure 1: Initial static schedule table

M1 (0.2),	M2 (0.8), M5 (0.3),	M7 (0.6),			
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),	M8 (0.4),			
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),		M9 (0.4),		
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),			M10 (0.8),	
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),	M7 (0.6),			
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),	M8 (0.4),			
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),			M11 (0.6),	
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),			M12 (0.4),	
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),	M7 (0.6),			
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),	M8 (0.4),			
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),		M9 (0.4),		
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),			M13 (0.4),	
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),	M7 (0.6),			
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),	M8 (0.4),			
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),				M14 (0.2),
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),				
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),	M7 (0.6),			
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),	M8 (0.4),			
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),		M9 (0.4),		
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),			M10 (0.8),	
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),	M7 (0.6),			
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),	M8 (0.4),			
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),			M11 (0.6),	
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),			M12 (0.4),	
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),	M7 (0.6),			
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),	M8 (0.4),			
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),		M9 (0.4),		
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),			M13 (0.4),	
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),	M7 (0.6),			
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),	M8 (0.4),			
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),				
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2),	M2 (0.8), M5 (0.3),				
M1 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				

Figure 2:Dynamic schedule table after model addition

M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),	M7 (0.6),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),	M8 (0.4),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),		M9 (0.4),		
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),			M10 (0.8),	
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),	M7 (0.6),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),	M8 (0.4),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),			M11 (0.6),	
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),			M12 (0.4),	
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),	M7 (0.6),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),	M8 (0.4),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),		M9 (0.4),		
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),			M13 (0.4),	
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),	M7 (0.6),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),	M8 (0.4),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),				M14 (0.2),
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),				
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),	M7 (0.6),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),	M8 (0.4),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),		M9 (0.4),		
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),			M10 (0.8),	
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),	M7 (0.6),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),	M8 (0.4),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),			M11 (0.6),	
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),			M12 (0.4),	
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),	M7 (0.6),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),	M8 (0.4),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),		M9 (0.4),		
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),			M13 (0.4),	
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),	M7 (0.6),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),	M8 (0.4),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),				
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				
M1 (0.2), M15 (0.2),	M2 (0.8), M5 (0.3),				
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4), M6 (0.2),				

Figure 3:Dynamic schedule table after model deletion

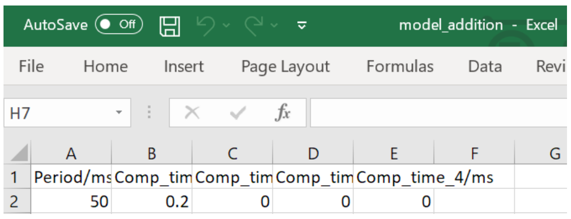
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),	M7 (0.6),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),	M8 (0.4),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),		M9 (0.4),		
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),			M10 (0.8),	
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),	M7 (0.6),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),	M8 (0.4),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),			M11 (0.6),	
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),			M12 (0.4),	
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),	M7 (0.6),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),	M8 (0.4),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),		M9 (0.4),		
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),			M13 (0.4),	
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),	M7 (0.6),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),	M8 (0.4),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),				M14 (0.2),
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),				
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),	M7 (0.6),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),	M8 (0.4),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),		M9 (0.4),		
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),			M10 (0.8),	
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),	M7 (0.6),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),	M8 (0.4),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),			M11 (0.6),	
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),			M12 (0.4),	
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),	M7 (0.6),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),	M8 (0.4),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),		M9 (0.4),		
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),			M13 (0.4),	
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),	M7 (0.6),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),	M8 (0.4),			
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),				
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				
M1 (0.2), M15 (0.2),	M6 (0.2), M5 (0.3),				
M1 (0.2), M15 (0.2),	M3 (0.6), M4 (0.4),				

The schedule table in figure 1 above has the same format as schedule table in figure 5 in [1]. The horizontal axis is the distinct periods. The vertical axis is the time step and it is also the timeline (passage of time). For example, in the 1st time step, models M1 and M2 and in the 3rd time step, models M1 and M4 and in the 8th time step models M1 and M9 will be executed. The length of the time step is 50ms resulting from 'time_Step.py'. Models in period 100ms (2nd column) repeat every 2 time steps.

Models with shorter periods and longer computation times are assigned first to largest idle rates as these have limited flexibility (fewer time steps) in the schedule table. Contribution from M1 to idle rate is fixed for every time step due to 0.2mS computation time. Note how M2 with highest computation time (0.8ms) is assigned to first sub step of period 100ms and M3 the next highest computation time is assigned to the second sub step as this has the highest idle rate (1) now. M4 is then assigned to sub step 2 and not sub step 1 as that step has lower idle rate than sub step 2 due to its higher computation time (0.8 ms compared to 0.6 mS). After Model 4 is added, the idle rate of sub-step 2 decreases due to 1ms computation time compared to 0.8mS computation time of sub step 1. This process goes on until period 100mS models are assigned. For M7 in period 400mS. It could be assigned to any odd sub-step within the 8 sub-steps of its period due to higher idle rate.Its assigned to 1st sub-step but for M8 now the idle rate of the 1st sub-steps and even sub-steps are higher than the next odd sub steps so it is assigned to sub-step 3. Similar process continues for periods 800mS, 1600mS and 3200mS.

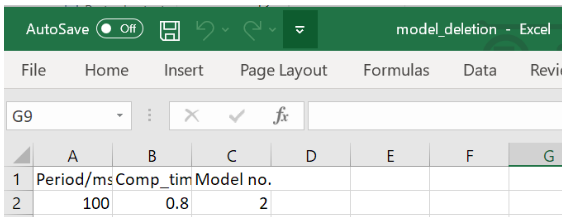
Schedule table after model addition as per model parameters in table 2 is shown in figure 2 above.

Table 2: model addition parameters



	A	B	C	D	E	F	G
1	Period/ms	Comp_tim	Comp_tim	Comp_tim	Comp_time_4/ms		
2	50	0.2	0	0	0		

Table 3: model deletion parameters



	A	B	C	D	E	F	G
1	Period/ms	Comp_tim	Model no.				
2	100	0.8	2				

As 50mS model is added it is given a unique id of M15, If model of a higher period is added the algorithm automatically finds the largest idle rate time step similar to the static scheduling table above.

Schedule table after Model deletion as per model parameters given in table 3 is shown in figure 3 above. When M2 (0.8mS computation time) of 100ms is deleted we can notice that the contents of models in sub steps 1 and 2 of period 100mS has changed. This is due to a large disparity created when a high computation task like 0.8mS is deleted. The load of 2nd sub-step ('max_sstep') now has (0.6+0.4+0.2 = 1.2mS) of computation time and this results in a low idle rate. In contrast, the load of 1st sub-step ('min_sstep') only has computation time of 0.3mS. Therefore the disparity is 1.2 – 0.3 = 0.9mS which is greater than the imbalance threshold [1] of 0.8mS. This results in transferring model M6 to the 1st sub step. For more details please see code comments. Now computation time of sub step 1 and 2 is 0.7 and 1.0. Now the idle rate of the 2nd sub step is improved and prevents models becoming un-schedulable in certain smaller periods. If rebalancing is not performed, during model addition overheads due to need for rebalancing can occur and this can sometimes cause models in smaller periods to miss their deadlines

Pitfalls/Disadvantages

Load rebalancing in TLS introduces time fragments which causes maximum processor utilization to be reduced. There can be more than one choice of time step so choosing a smaller time step may increase flexibility of task allocation as execution times in this setting are much smaller than their periods.

Paper 2: Real-time Scheduling of Deferrable Electric Loads

Abstract

In this real-time Electric vehicle charging schedule task, we analyse ways to reduce grid energy use and maximize renewable energy use to deliver energy over a servicing period to deferrable loads using real-time heuristic causal scheduling policies like EDF and Receding Horizon Control (RHC). RHC utilizes more renewable energy than EDF.

Short Introduction of the Problem, model, and objective(s)

Certain electric loads like electric vehicles are flexible in the sense that given the energy required to fully charge and maximum charging power we can schedule this process/task over some time. This flexibility allows us to incorporate variable energy sources like wind energy using power forecast data together with readily available grid energy sources.
Causal optimal scheduling policies do not exist where there are power constraints. This is because future power availability has an impact on power distribution to loads at present time. This is illustrated in theorem 1 in [2]. Currently, the grid energy is reserved in high quantities to counter renewable energy variability, but this is not a scalable solution. Therefore, heuristic causal scheduling policies through EDF and RHC algorithms are proposed and their performance evaluated using metrics of required renewable and grid energy.

Description of Input data / Parameters in detail

The electric vehicle dataset is obtained for a chevy volt battery in table 1 and figure 5 from [3]. In order to speed up real-time simulation the Energy of the battery is scaled down from 16Kw hour to 1.6Kw Second. And the maximum chargeable power from 1.4Kw to 0.14Kw and the charging time from 11 hours to (1.6KwS / 0.14Kw) = 11 seconds. A fleet operator may have these same cars and so they all have identical total energy to charge E (set to 1.6KwS), identical charging times (set to 20 S) forms deadline d from arrival time, maximum serviceable/chargeable power m (set to 0.14Kw). The arrival time can be any time between 0 and 90 seconds as wind power data is available for 0 to 105 seconds for 1 to 5 tasks.

Wind power data was obtained from [4]. A section of the data was rescaled in terms of Kw over every 1 second and used as shown below. It is stored in ‘wind_power_data.csv’.

Figure 4: Wind power profile and task set

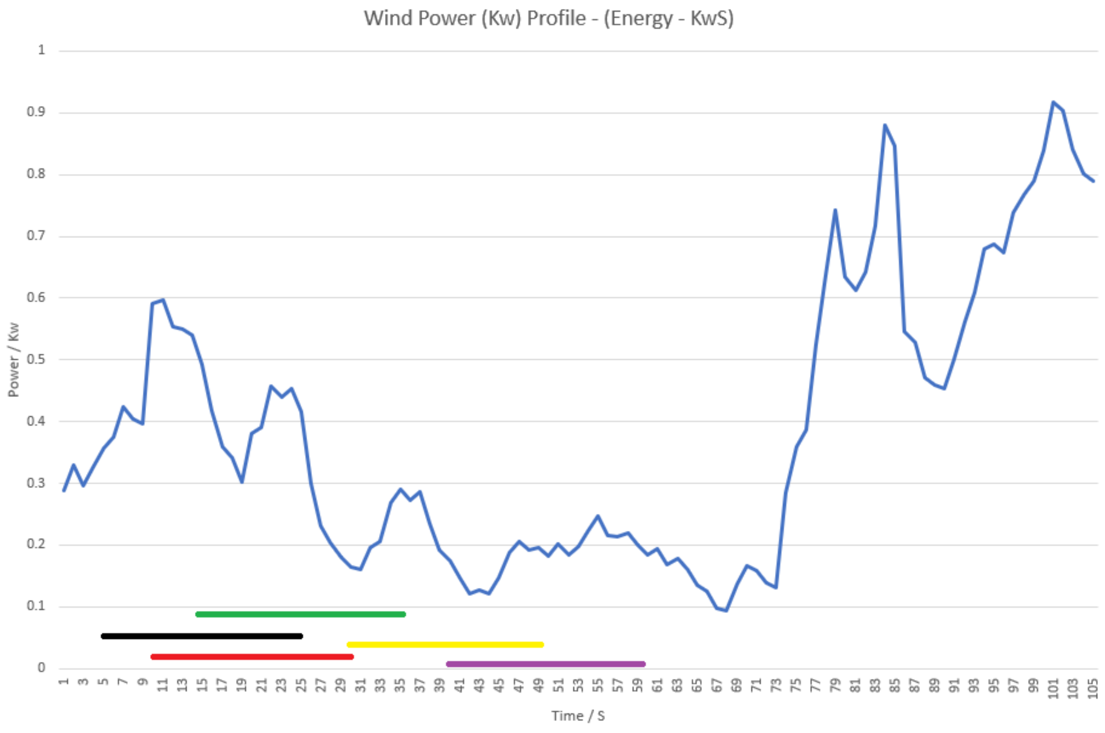
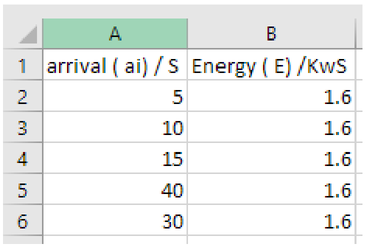


Table 4: simulated task set



	A	B
1	arrival (ai) / S	Energy (E) /KwS
2	5	1.6
3	10	1.6
4	15	1.6
5	40	1.6
6	30	1.6

The task set can be set in 'task_set.csv'.

Algorithm description

The Receding horizon control problem involving online optimization was modelled and solved using open-source tools the Pyomo module and the 'Octeract engine'.

In 'rhc.py', the task set is extracted from 'task_set.csv'. 20 seconds is added to every arrival time to create deadlines. 'del_t' refers to the time over which a fixed energy is allocated, and it is the sub step of the time horizon. 'rem_E' refers to remaining energy for every task. It has the same meaning as capital E in constraints 8 and 11 in [2].

Next, wind power data is extracted from 'wind_power_data.csv'. The function 'wind_pred(N):' makes predictions of the wind power over the next N-1 future time steps using currently measured wind power data. To make results repeatable gaussian sampling is not used rather multiples of 0.008 are subtracted from current wind power. These act as Upper bound in constraint 7 in [2].

The dimensions (M x N) of the matrix W and G are first computed in 'calc_M_N_final_d(d,rem_E):'. M represents number of active tasks/cars to recharge. N represents the number of 'del_t' time steps till final deadline is active task set. If a task's time between arrival and deadline overlap with the current time and its remaining energy 'rem_E' is positive (>0.01) then it is added to the active task set.

The real-time schedules are obtained by solving a constrained online convex optimization problem every step or 'del_t' (5 seconds) until the last deadline. In each step, if M is zero there are no active tasks at the present time and so we wait until M is non-zero. When there are active tasks, a pyomo 'ConcreteModel' object is created. Attributes such as variables like W,G, phi and energy (small e) are added to this model. Index sets are also created to allow iterations through the variables. These variables are initialized to zero before every optimization step.

NumPy predefined matrix operations are used to define and calculate the objective function in equation 6 of [2]. The objective function is specified using 'Objective(expr= obj_exp,sense= minimize)'. Next all the 5 constraints 7 – 11 in [2] are defined inside lists for every task i and time step k.

Finally, the optimization model 'm' is passed into the solver and solver name specified as 'octeract-engine'. The entire optimization is timed using 'st' and 'et'. This is to detect the k number of steps passed. A copy of the current optimization solution is saved in arrays old_W and old_G. The energy delivered in this k steps is subtracted from the 'rem_E' during next optimization calculation.

In order to see the full results Uncomment below line 235 "## display results in terminal" to display calculated W and G values in each step.

Simulation effort & Results discussions

After creating the 'pyomo' python environment and installing the octeract engine as in 'Readme_paper2.txt'. Please contact me if any issues. Please run 'rhc.py' to simulate.

In figure 4 above black (T1), red (T2), green (T3), purple (T4) and yellow (T5) lines represent charging tasks with arrival and deadlines as shown. The arrival times of the task set are indicated in table 4 They have identical total Energy requirements (of 1.6KwS). It represents 'E' in paper and 'rem_E' in code. They also have identical power requirements (of 0.14Kw). It represents 'm' in paper and 'maxP' in code. When 'rhc.py' is run it generates the real-time energy allocation schedule as shown in the following table.

The terminal output for this task set is copied to 'task_set_RESULTS.txt'

Table 5: RHC energy allocation

Current sim Time / S	Energy consumption for each task	Charging Task colours	M and N size	Total Energy (Wind + grid)

0	T1
	Wind = 0.0698 Grid used = 3.9e-09
		black	M = 1
			N = 4
				T1 = 0.0698

8.02	T1
	Wind = 0.282 Grid = 0.0001075
		black	M = 1
			N = 3
				T1 = 0.3518

10.66	T1
	Wind = 0.3715 Grid = 4.0559e-09
	T2
	Wind = 0.1999 Grid = 4.0559e-09
		Black and red	M = 2
			N = 3
				T1 = 0.7233
				
				T2 = 0.2

15.0	T1
	Wind = 0.2061, Grid = 0.4938
	
	T2
	Wind = 0.1829 Grid = 0.3784
	
	T3
	Wind = 0.02937 Grid = 0.03203
		Black, red and green	M = 3
			N = 4
				T1 = 1.4233
				
				T2 = 0.7613
				
				T3 = 0.0614

20.0	T1
	Wind = 0.07377 Grid = 0.10262
	T2
	Wind = 0.15865 Grid = 0.40133
		Black, red and green	M = 3
			N = 3
				T1 = 1.608
				Black task is over by deadline

	T3
	Wind = 0.158 Grid = 0.4013
				T2 = 1.3213
				
				T3 = 0.6200

25.0	T2
	Wind = 0.1089 Grid = 0.1696
	T3
	Wind = 0.1905Grid = 0.5094
		Red and green	M = 2
			N = 2
				T2 = 1.599
				Red task is over by deadline
				
				T3 = 1.320

30.0	T3
	Wind = 0.07008 Grid = 0.2085
	T4
	Wind = 0.08995 Grid = 0.61004
		Green and yellow	M = 2
			N = 4
				T3 = 1.600
				green task is over by deadline
				
				T4 = 0.700

35.0	T4
	Wind = 0.2729 Grid = 0.42700
		yellow	M = 1
			N = 3
				T4 = 1.399

40.0	T5
	Wind = 0.08726 Grid = 0.6127
	T4
	Wind = 0.05907 Grid = 0.14092
		Purple and Yellow	M = 2
			N = 4
				T5 = 0.6999
				
				T4 = 1.5999
				Yellow task is over by deadline

45.0	T5
	Wind = 0.1874 Grid = 0.51258
		Purple	M = 1
			N = 3
				T5 = 1.399

50.0	T5
	Wind = 0.1999 Grid = 1.1459e-08
		Purple	M = 1
			N = 2
				T5 = 1.5999
				Purple task is over BEFORE deadline

	Overall Grid and Wind energy consumption Grid = **5.000** KwS Wind = **2.999** KwS Grid + Wind = 7.99 ~= 1.6 x 5 = 8 KwS			

For the same tasks set above, Earliest Deadline First (EDF) algorithm is computed by hand to compare grid energy usage. Its results are shown in the table below.

The EDF scheduling policy first assigns available power p(t) to the active task Tj with the earliest deadline: $j(t) = \operatorname*{argmin}_{i \in A_t} d_i.$

Table 6: EDF energy allocation

Time / S	Energy consumption for each task	Charging Task colours	Task completion
5 – 10	T1: G = 0, W = 0.14 x 4 = 0.56	Black and red	

10 – 15	T1: G = 0, W = 0.14 x 4 = 0.56
	T1= 1.4
	T2: G = 0.00297 + 0.005516+ 0.01045+ 0.0334 = 0.25, W = 0.14 + 0.2076 = 0.34
	T2 = 0.4976
		Black and red	

15 - 25	T1: G = 0.1, W = 0.14+0.001 = 0.14
	T1 = 1.6
	T2: G = 0.593, W = 0.0692 + 0.0392 + 0.14*2 = 0.24
	T2 = 1.6
	T3: G = 0.14+0.14 + 0.64953 = 0.92953, W = 0.17046
	T3 = 1.399
		Black and red and green	Black and red

25 - 30	None		

30-35	T3: G = 0.36
	W = 0.0011
	T3 = 1.6
	T5: G = 0.14 x 4 + 1.1345 = 0.6145, W = 0.0055
	T5 = 0.7
		Green and yellow	Green

35 - 40	T5: G = 0.71978, W = 0.1364 + 0.1433 + 0.1172 + 0.096+ 0.0086 = 0.0802 T5=0.98	Yellow	

40 - 50	T4: G = 0.14*2 = 0.28 + 0.4405 = 0.792, W = 0.0679.
	T4 = 0.8599
	T5: G = 0.66, W = 0.0731+ 0.060 = 0.177
	T5=1.6
		Yellow and purple	Yellow

50 - 60	T4: G = 0.39881, W = 0.2412 T4 = 1.6	Purple	Purple
	Overall Grid and Wind energy consumption Grid = **5.41762** KwS Wind = **2.58238** KwS Grid + Wind = 7.98 ~= 1.6 x 5 = 8 KwS		

<u>Results analysis</u>

In comparing the overall grid energy usage, RHC draws less grid energy than EDF. In table 5 for RHC, between 15 and 20 seconds, substantial grid energy is used as wind power is shared amongst the three tasks and deadline is nearby at 20 S for task 1 followed by task 2. Task 3 however is deferred as it has a later deadline so it consumes lesser grid energy and RHC hopes that there will be more Wind (renewable energy) in future. Between 20 and 25 seconds again Wind energy is insufficient for all the tasks so grid energy is used because RHC knows that more tasks in active task set are going to cause congestion and it cannot afford to miss a deadline. No tasks missed the deadline.

The largest grid energy consumption for RHC is only 0.6127 KwS but for EDF it is 0.92953 KwS. EDF freely consumes large grid energy as it uses 0.719 KwS and 0.792 KwS as its only goal is to meet its immediate deadline without using any future information. There is no notion of task deferability, so it does not assess urgency of the tasks properly. Therefore, it was observed that EDF completes tasks faster than RHC within the deadlines. However, this is not our objective so it is not desired. Deadlines are not missed in both RHC and EDF because they can both can draw arbitrary amounts of grid energy but RHC is more conservative in its use of grid energy.

If deadlines are made different for each task and there are more task overlaps, RHC performs much better than EDF.

Pitfalls/Disadvantages

The time to calculate the optimal W and G is considered to be instantaneous, this is because of the difficulty in gauging the exact time for an optimisation step. From my experiments the same problems take varying times to solve each time, so it is difficult to factor this into the N time steps at the beginning of the optimization.

Conclusion

EDF and RHC were used to charge 5 electric vehicles through wind and grid energy sources over a 60 second period. Using EDF, there is higher grid energy consumption towards the end of the task service intervals as flexibility in task scheduling decreases. On the other hand, grid energy use is more balanced with RHC scheduling as it utilizes renewable generation forecasts in energy allocation. From the above experiment, RHC uses 8.3% less grid energy than EDF.

References

[1] Yulin Wu, Xiao Song, Gong, “Real-time load balancing scheduling algorithm for periodic simulation models”, Simulation Modelling Practice and Theory 52 (2015) 123–134

[2] A. Subramanian, M. Garcia, A. Dominguez-Garcia et. al, “Real-time Scheduling of Deferrable Electric Load”, 2012 American Control Conference

[3] S.Shao, M. Pipattanasomporn, and S. Rahman, “Challenges of PHEV penetration to the residential distribution network,” IEEE Power & Energy Society General Meeting, 2009.

[4] BPA Balancing Authority Load & Total Wind Generation at 5-minute intervals, last 7 days Dates: 16Mar2021 - 23Mar2021 (last updated 22Mar2021 04:00:34) Pacific Time https://transmission.bpa.gov/business/operations/Wind/baltwg.aspx

Coding effort

Paper 1 and Paper 2:

My coding effort is 100% for both the papers as I directly implement the papers without using any third-party code. The code comments provide description of code functionality.

Paper 2:

Open source tools:

Pyomo, python optimization modelling objects to create optimization objectives and constraints

Octeract: as optimization solver for online convex optimization.

Acknowledgements

Paper 1:

I am thankful to Xiao Song from Beihang University for explaining some technical details from the paper.

Paper 2:

I am thankful to Sai Ganesh from EPFL for clearing doubts with regards to mathematical optimization.

I am also thankful to Gabriel Lau from Octeract for his support and technical assistance in making the solver work. Link: https://octeract.co.uk/about/