Cyclic executive scheduling:

Cyclic Executive Scheduling is a popular real-time scheduling technique used in embedded systems. In this approach, the system's tasks are divided into fixed, predefined time slots or cycles, and each task is assigned a specific slot within the cycle. The tasks are then executed in a repetitive manner, following the assigned schedule.

1. Time Division: Cyclic Executive Scheduling divides time into fixed-size slots or cycles. Each cycle consists of one or more time slots, and the length of each slot is typically determined based on the requirements of the system and the tasks it needs to perform.
2. Task Assignment: Each task in the system is assigned a specific time slot within the cycle. The assignment is based on the priority and timing constraints of the tasks. Higher priority tasks are typically assigned shorter time slots to ensure timely execution.
3. Deterministic Execution: Cyclic Executive Scheduling follows a deterministic approach, meaning that the execution of tasks is predictable and repeatable. Once the schedule is defined, the system executes the tasks in the same order and for the same duration in each cycle.
4. Fixed-Priority Scheduling: Cyclic Executive Scheduling often uses a fixed-priority scheduling policy. Each task is assigned a priority level based on its importance or urgency. During execution, the tasks are executed in a priority-based order, ensuring that higher priority tasks are given precedence over lower priority ones.
5. Periodic Tasks: Cyclic Executive Scheduling is particularly useful for systems with periodic tasks. Periodic tasks are tasks that repeat at regular intervals, such as sampling sensor data or generating periodic output signals. By assigning dedicated time slots to these tasks, Cyclic Executive Scheduling ensures that they are executed regularly and predictably.
6. Determining Cycle Length: The cycle length is determined based on the worst-case execution time of the tasks, their deadlines, and any other timing constraints. The cycle length should be set to accommodate the longest task within the system.
7. Response Time Analysis: Cyclic Executive Scheduling allows for a straightforward analysis of task response times. Since the execution order and timing of tasks are deterministic, it becomes easier to calculate the maximum time taken by a task to respond to an external event or input.
8. Trade-offs: While Cyclic Executive Scheduling provides deterministic behavior and predictable response times, it may not be suitable for systems with a high degree of task dynamism or where tasks have varying execution times. It works best for systems with static task sets and well-defined timing requirements.

Overall, Cyclic Executive Scheduling is a widely used scheduling technique in embedded systems, particularly in applications where predictability and determinism are crucial. By dividing time into fixed cycles and assigning tasks to specific slots, it ensures that tasks are executed in a timely and repeatable manner, making it suitable for real-time applications.

Proposed solution to the problem:

To find the solution for scheduling a taskset there are many techniques. integer programming is one of them. in python we can use z3 module which can find solution to the problem if it is feasible.

Z3 Python Module:

The Z3 library is a powerful theorem prover developed by Microsoft Research that can be used to solve a wide range of mathematical and logical problems, including scheduling problems like the Cyclic Executive Schedule. While Z3 is not specifically designed for scheduling problems, it provides a flexible framework for formulating and solving such problems using constraints and logical formulas.

1. Formulating the Problem: The first step is to define the problem using variables, constraints, and objective functions. In the case of Cyclic Executive Scheduling, you would define variables representing tasks, time slots, task durations, task dependencies, and any other relevant parameters.
2. Encoding Constraints: Next, you need to encode the constraints of the Cyclic Executive Schedule problem using logical formulas. For example, you would encode constraints such as task execution time being less than or equal to the allocated time slot, ensuring that tasks do not overlap, meeting task deadlines, and satisfying task dependencies.
3. Defining the Objective: Depending on the specific objective of the scheduling problem, you can define an objective function. For example, you may aim to minimize the overall schedule length, maximize resource utilization, or optimize for meeting task deadlines. The objective function helps guide the search for an optimal solution.
4. Creating Z3 Solver: Once the problem is formulated and encoded, you create an instance of the Z3 solver. The solver acts as an interface to the Z3 library and provides methods for adding constraints, setting the objective function, and solving the problem.
5. Adding Constraints: You add the encoded constraints to the Z3 solver using its API. Z3 provides a rich set of functions and operators to express constraints, such as arithmetic operators, logical operators, quantifiers, and so on. You can specify constraints on variables, inequalities, equalities, and logical relationships between variables.
6. Setting the Objective: If you have defined an objective function, you can set it in the Z3 solver. The objective function guides the solver in finding the optimal solution based on the defined criteria. Z3 supports various types of objectives, including minimization and maximization.
7. Solving the Problem: Once the constraints and objective are added, you invoke the solver's solving method to find a solution. The solver uses various algorithms and techniques to search for a feasible or optimal solution that satisfies the defined constraints and objectives. Z3 employs advanced algorithms like SAT (Satisfiability) and SMT (Satisfiability Modulo Theories) solvers to efficiently search the solution space.
8. Interpreting the Solution: After the solver returns a solution, you can interpret and extract the values of variables representing the task schedule. You can analyze the schedule to verify its correctness, evaluate its performance metrics, or perform any further post-processing as needed.

Using the Z3 library to solve a Cyclic Executive Schedule problem requires formulating the problem, encoding constraints, setting objectives, and invoking the solver. The flexibility and

expressive power of Z3 make it a valuable tool for solving a wide range of scheduling problems, including Cyclic Executive Scheduling.

Reading and converting the taskset:

The taskset is stored in the memory in csv format. We can use pandas to read the csv file. Below is a snapshot of the taskset displayed.

Task	Computation	Period	Deadline
1	2	6	6
2	2	9	9
3	2	12	8
4	4	18	10

this taskset can then be converted to python dictionary for easily manipulating the data. `get_taskset_dict_list(taskset_df)` function does that. We get the following format.

```
[{'Task': 1, 'Computation': 2, 'Period': 6, 'Deadline': 6},
 {'Task': 2, 'Computation': 2, 'Period': 9, 'Deadline': 9},
 {'Task': 3, 'Computation': 2, 'Period': 12, 'Deadline': 8},
 {'Task': 4, 'Computation': 4, 'Period': 18, 'Deadline': 10}]
```

Finding the hyper period and Frame size:

The hyperperiod (H) is the LCM (Least Common Multiple) of the task periods, given by:

$$H = \text{LCM}(P_1, P_2, P_3, \ldots, P_n)$$
$$2 \cdot f - \gcd(f, T_i) \leq D_i$$

Suitable Frame size can be found using the GCD condition.

Code is written in Python and aims to determine the valid frame sizes for a Cyclic Executive Schedule based on a given taskset. It begins by calculating the hyperperiod, which represents the least common multiple (LCM) of the task periods. The minimum frame size is determined by finding the maximum computation time among all the tasks, while the maximum frame size is calculated as the minimum period value in the taskset.

Next, an array of possible frame sizes is created, ranging from the minimum to the maximum frame size. The code then calls the `get_valid_frame_sizes()` function with the array of possible frame sizes and the hyperperiod as inputs. Within this function, the frame sizes that are multiples of the hyperperiod are identified using the modulo operation, and an array of valid frame sizes is returned.

The code then initializes an empty list to store the final valid frame sizes. It iterates over each valid frame size and counts the number of tasks that satisfy a specific condition within that frame size. For each task, the condition is checked, which involves a calculation based on the frame size, task period, and deadline values. If the condition is satisfied for all tasks, the frame size is considered valid, and it is added to the final list of valid frame sizes.

Finally, the code outputs the array of final valid frame sizes. This code provides a practical approach for determining the valid frame sizes for a Cyclic Executive Schedule, taking into account the characteristics of the taskset and ensuring that the defined scheduling constraints are met.

Initializing z3 solver and optimization variables:

```
from z3 import Solver, Int
```

Sort the task set: The code starts by sorting the task set based on the deadlines of the tasks. This step ensures that tasks with earlier deadlines are considered first during the scheduling process.

Define start times: The code creates a list of integer variables called `start_times`. Each variable represents the start time of a task and is named accordingly. These start times will be determined by the solver to achieve a valid schedule.

Create a solver: An instance of the Z3 solver is created using `Solver()`. The solver acts as an interface to the Z3 library and provides methods for defining constraints, setting objectives, and solving the scheduling problem.

Overall, the code sets up the necessary components for solving the scheduling problem using the Z3 library.

Finding individual task instances start times:

1. Initialize an empty list called `release_instances` to store the instances of each task.
2. Iterate over each task in the `task_set`.
3. For each task, calculate the number of instances based on the hyperperiod and the task's period. This is done by dividing the hyperperiod by the task's period and taking the floor value.
4. Set the computation time, previous period, and deadline variables based on the task's properties.
5. Initialize an empty list called `instances` to store the individual instances of the task.
6. Set the previous deadline variable to the task's deadline initially.
7. Iterate from 0 to the number of instances calculated in step 3.
8. Within the loop, create an instance for the task by appending a dictionary containing the previous period, previous deadline, and computation time to the `instances` list.
9. Update the previous period by adding the task's period.
10. Update the previous deadline by adding the previous period and the task's deadline.
11. Append the `instances` list to the `release_instances` list, representing the instances of the current task.
12. Repeat steps 2-11 for each task in the `task_set`.

we get below list of dictionaries:

```
[[{'Period': 0, 'Deadline': 10, 'Computation': 2}],
 [{'Period': 0, 'Deadline': 5, 'Computation': 1},
  {'Period': 5, 'Deadline': 10, 'Computation': 1},
  {'Period': 10, 'Deadline': 15, 'Computation': 1},
  {'Period': 15, 'Deadline': 20, 'Computation': 1}],
 [{'Period': 0, 'Deadline': 6, 'Computation': 2},
  {'Period': 5, 'Deadline': 11, 'Computation': 2},
  {'Period': 10, 'Deadline': 16, 'Computation': 2},
  {'Period': 15, 'Deadline': 21, 'Computation': 2}]]
```

Adding constraints to the solver and optimization:

Iterate over each task in the sorted task set.

1. Initialize a variable `prev_st` to track the previous start time.
2. For each task, iterate over the instances of the task and their corresponding optimization variables.
3. Retrieve the start time, computation time, period, and deadline for the current instance.
4. Add constraints to the solver object (`smt`) for the current instance. These constraints include:

    - The start time must be non-negative.
    - The start time should not overlap with other instances of the same task (using the `Or` operator).
    - The start time must be greater than or equal to the period of the instance.
    - The end time (start time + computation time) must be less than or equal to the deadline of the instance.
5. Update the `prev_st` variable with the current start time.
6. Check if a solution exists by calling `smt.check()` on the solver object.
7. If a solution exists (indicated by `smt.check() == sat`), retrieve the model from the solver.
8. Create a list called `ploting_list` to store the schedules of each task.
9. For each task, create a schedule list by evaluating the optimization variables in the model and appending the start time, end time, and computation time to the schedule list. Sort the schedule list by the start time.
10. Append the schedule list to the `ploting_list`.
11. Print the schedule by iterating over the schedule list and displaying the task, start time, end time, and computation time.
12. If no feasible schedule exists (indicated by `smt.check() != sat`), print "No feasible schedule exists."

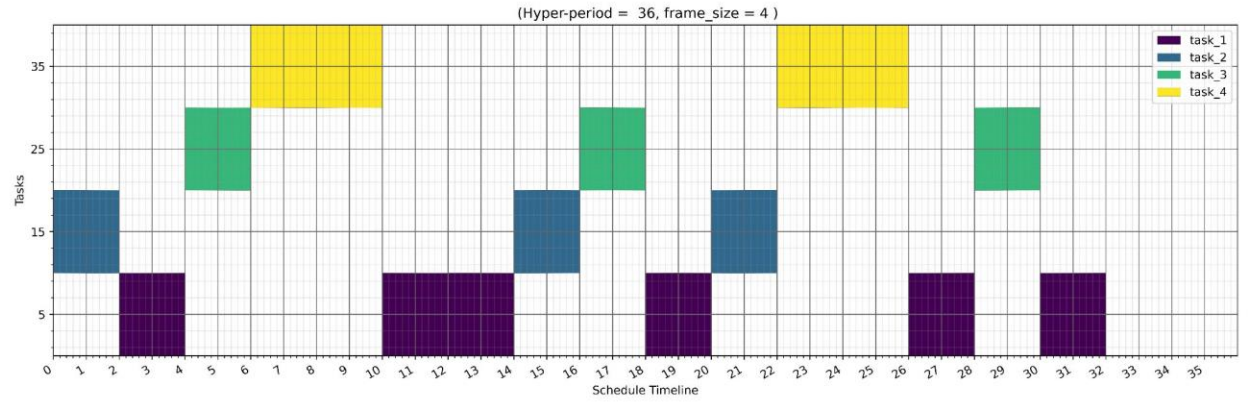
Plotting the results:

The code generates a visual representation of the schedule by plotting task execution bars on a timeline. Each task is represented by a colored bar, and the position and length of the bars indicate the start time and computation time of the respective tasks. The resulting plot provides an intuitive view of the schedule and allows for easy analysis and understanding.

Below are some of the results generated by this technique.

Example1:

Task	Computat	Period	Deadline
1	2	6	6
2	2	9	9
3	2	12	8
4	4	18	10



Example2:

Task	Computation	Period	Deadline
1	2	20	10
2	1	5	5
3	2	5	6