IMMC 2025

Problem E

When Watts Meet Bits: The Power-Computing Collaborative Scheduling

St. Joseph's College

Teacher	Dr. Cheung Wai Shan	張慧珊
Students	Zhang Chenge	張宸歌
	Tang Ka Lam	鄧嘉琳
	Lee Hin Chun	李衍俊
	Tong Wai Hang	唐煒恆

SJC - PROBLEM E

SUMMARY

1 - Introduction

Our project optimizes energy usage and task scheduling using a Python-based mathematical model over 24 hours. It balances renewable and traditional energy sources while maximizing task performance and minimizing costs.

2 - Setup

We define data, import key modules, and create tasks with attributes like start time, priority, workload, and delay.

3 - Optimisation

The core focuses on finding optimal solutions through mathematical approaches, iterating through tasks over 10,000 simulations. Energy consumption is calculated hourly, tasks are delayed if needed, and costs are minimized.

4 - Best-fit Curve

Statistical analysis is performed on optimized task schedules, categorized by priority. Data is visualized using scatter plots and fitted to polynomial curves to identify task distribution trends.

5 - Conclusion

Our project offers insights into energy optimization and task scheduling, highlighting challenges in real-world applications and suggesting avenues for future improvement.

SJC - PROBLEM E

PROBLEM INTRODUCTION

1 - Introduction

We chose problem E as our focusing problem this time. The task is about creating an optimization model that can maximize the green energy used, and minimize:

- the cost of purchasing power, both green power and traditional power,
- the delay for each task
- the excess / short of green energy for each hour

The amount of green energy available, prices of energy supply, and task demand for each hour are given. Therefore, the goal is to create a model (function) that represents the number of tasks for each hour. The task management system can thus use this model to decide the amount of tasks that are suitable for processing in the current hour, and decide whether to delay outstanding tasks, in order to save cost, energy and promote green energy.

Our solution should be based on the following principles:

1. Tasks should be **delayed as little as possible**, as delays for a server in the modern world is expensive [REF 1].
2. Even if there is a delay, higher-urgency tasks should be delayed the least.
3. **Green energy should be applied as much as possible**, reducing surpluses and shortages.
4. The **cost of the power supply should be as low as possible**, reducing cost for tasks.

2 - Strategy

2-1 - Identify

We noticed that,

- As the power drawn out is linear, the energy should be described by $E = Pt$. As the power is given hourly, the working units should be in kWh. The power given in MW can be transformed to kWh by multiplying by 1000.
- Delaying high-urgency tasks costs more, so lower-urgency ones need to be considered first while calculating delay.
- Delaying can never occur after 23:00, as the question is about creating a 24-hour plan. Therefore, all tasks must be done in the 24-hour span.
- As green energy is prioritized, tasks should be balanced to use most of the green energy available in the hour.

2-2 - Set-up

Based on our observation, we agreed that creating a simulation program that repetitively simulates planning of a day can best solve this problem.

Electricity Price Coordination The electricity price and availability for each hour is different. Our solution calculates the energy needed for an hour. Then, by $E = Pt$, we can obtain the total amount of electricity available for each hour. We can then compute the electricity costs for each hour.

Delay and Cost Optimization Since tasks needs to be delayed as little as possible, we try to discourage delays by adding awards and punishments. If there is a delay to a task, its corresponding score will be reduced. Our score reduction system is defined as follows

$$P = \begin{cases} 100\% \cdot 0.6 \cdot e^d, & \text{if } u = \text{HIGH} \\ 100\% \cdot 0.7 \cdot e^d, & \text{if } u = \text{MED} \\ 100\% \cdot 0.8 \cdot e^d, & \text{if } u = \text{LOW} \end{cases}, \text{where } \begin{cases} d & = \text{The delay in hours} \\ u & = \text{The urgency} \end{cases}$$

The cost to run the tasks should be optimized as well. Therefore, for each iteration, if $\Sigma_{i=1}^{n} T_{mark,i} > (\Sigma T_{mark})_{max}$, where T_{mark} is the mark of a task, and $\Sigma_{i=1}^{n} E_i < (\Sigma E_i)_{min}$, where E is the price of the energy consumed in an hour, the iteration is considered to be better than the last one. $(\Sigma T_{mark})_{max}$ and $(\Sigma E)_{max}$ should be replaced with $\Sigma_{i=1}^{n} T_{mark,i}$ and $\Sigma_{i=1}^{n} E_i$ respectively.

Green Energy Optimization As green energy should be used primarily, if there is not enough green energy to run all the tasks, lower-urgency tasks can be delayed.

Randomization To create variance, random numbers are used. The tasks are generally delayed at a $1/4$ chance. When green power is not enough, they are delayed at a $1/2$ chance.

2-3 - Expected outcome

We hope to obtain three arrays of integers, each describing the number of high-urgency, medium-urgency and low-urgency tasks. Then, by curve fitting, we would obtain three functions, $N_{high}(t)$, $N_{medium}(t)$, and $N_{low}(t)$. The three functions would allow interpolation in the dataset.

REFERENCES

REF 1 - Alex Hawkes (n.d.). **The real cost of latency**. Retrieved from https://blog.consoleconnect.com/the-real-cost-of-latency

1 - Implementation

Python [REF 1] on Jupyter Notebook [REF 2] are used to implement the solution. See Appendix A for more information on the code.

We mainly used the following libraries:

Numpy [REF 3] Provides convenient data storage and operation functions, allowing us to calculate exponents, standard deviations, etc, without writing extra code. Additionally, the `polyfit()` function provides polynomial curve fitting functions, which are crutial to our goal.

Matplotlib [REF 4] Provides graphing functions, allowing us to generate graphs for output. We can then use the graphs for inspection and presentation.

SymPy [REF 5] Provides symbolic math functions, for us to convert Numpy-generated polynomial coefficients into polynomials with unknowns, and display them as well.

2 - Method

This code defines two classes: `Priority` (an enum for task priority with associated energy consumption) and `Task` (containing start time, priority, workload, and delay, with a method `calc_mark` calculating a task's mark based on priority and delay using an exponential function). This code defines two classes: `Priority` (an enum for task priority with associated energy consumption) and `Task` (containing start time, priority, workload, and delay, with a method `calc_mark` calculating a task's mark based on priority and delay using an exponential function). A list `ORIG_TASKS` is created, populating it with multiple `Task` instances with varying parameters. An optimization process iteratively delays tasks based on probabilities (1/4 initially, 1/2 if renewable energy is insufficient), and awards points randomly for delays, calculates energy costs, and updates the total marks.

For each hour delay, we will deduct 20,30 and 40 percent of the score each hour a low, medium, high urgency task is delayed respectively

The algorithm aims to maximize total marks while minimizing energy costs, iterating 10,000 times to find an optimal solution. Finally, statistical analysis is performed on the results, summing task values, and fitting curves (9th-power polynomials) to the number of tasks per hour for each priority level.

3 - Evaluation

The results will be evaluated with the following

Residue The residue [REF 6] is differences between the response data and the fit to the response data at each predictor value.

RSS The residual sum of squares [REF 7] is a way to evaluate curve fitting, and measures the level of variance in the error term, or residuals, of a regression model. Lower values mean that the covergence is higher.

Error Variance Error variance [REF 8] is the variance of errors between the fitted line and the known data points. Smaller error variance means that the fitted curve maintains a smaller distance from known data points.

REFERENCES

REF 1 - Author unknown (n.d.). **Welcome to Python.org**. Retrieved from https://python.org

REF 2 - Author unknown (n.d.). **Project Jupyter | Home**. Retrieved from https://jupyter.org/

REF 3 - Author unknown (n.d.). **NumPy**. Retrieved from https://numpy.org/

REF 4 - Author unknown (n.d.). **Matplotlib — Visualization with Python**. Retrieved from https://matplotlib.org/

REF 5 - Author unknown (n.d.). **SymPy**. Retrieved from https://www.sympy.org/en/index.html

REF 6 - Author unknown (n.d.). **Residual Analysis** . Retrieved from https://www.mathworks.com/help/curvefit/residual-analysis.html

REF 7 - Retrieved from https://www.investopedia.com/terms/r/residual-sum-of-squares.asp

REF 8 - Simske, S. (2019). **Modeling and model fitting**. Meta-Analytics, (), 217-228. https://doi.org/10.1016/B978-0-12-814623-1.00007-1

After running the code, we have the following results.

1 - Task arrangement

Our most optimized solution for task arrangement is described as follows.

Hour	Tasks (High)	Tasks (Medium)	Tasks (Low)
0	0	0	0
1	0	6.66666667	20
2	0	13.33333333	10
3	0	6.66666667	0
4	0	6.66666667	20
5	0	6.66666667	0
6	0	0	10
7	0	55	70
8	28.5	18	0
9	28.5	18	0
10	0	18	0
11	28.5	0	0
12	55.5	0	0
13	0	113	0
14	0	0	0
15	0	40	0
16	76	0	0
17	38	20	0
18	38	32.5	10
19	0	12.5	10

Hour	Tasks (High)	Tasks (Medium)	Tasks (Low)
20	12.5	12.5	10
21	25	12.5	10
22	39.5	10	0
23	0	10	15

The following bar plot shows the number of tasks for each hour.

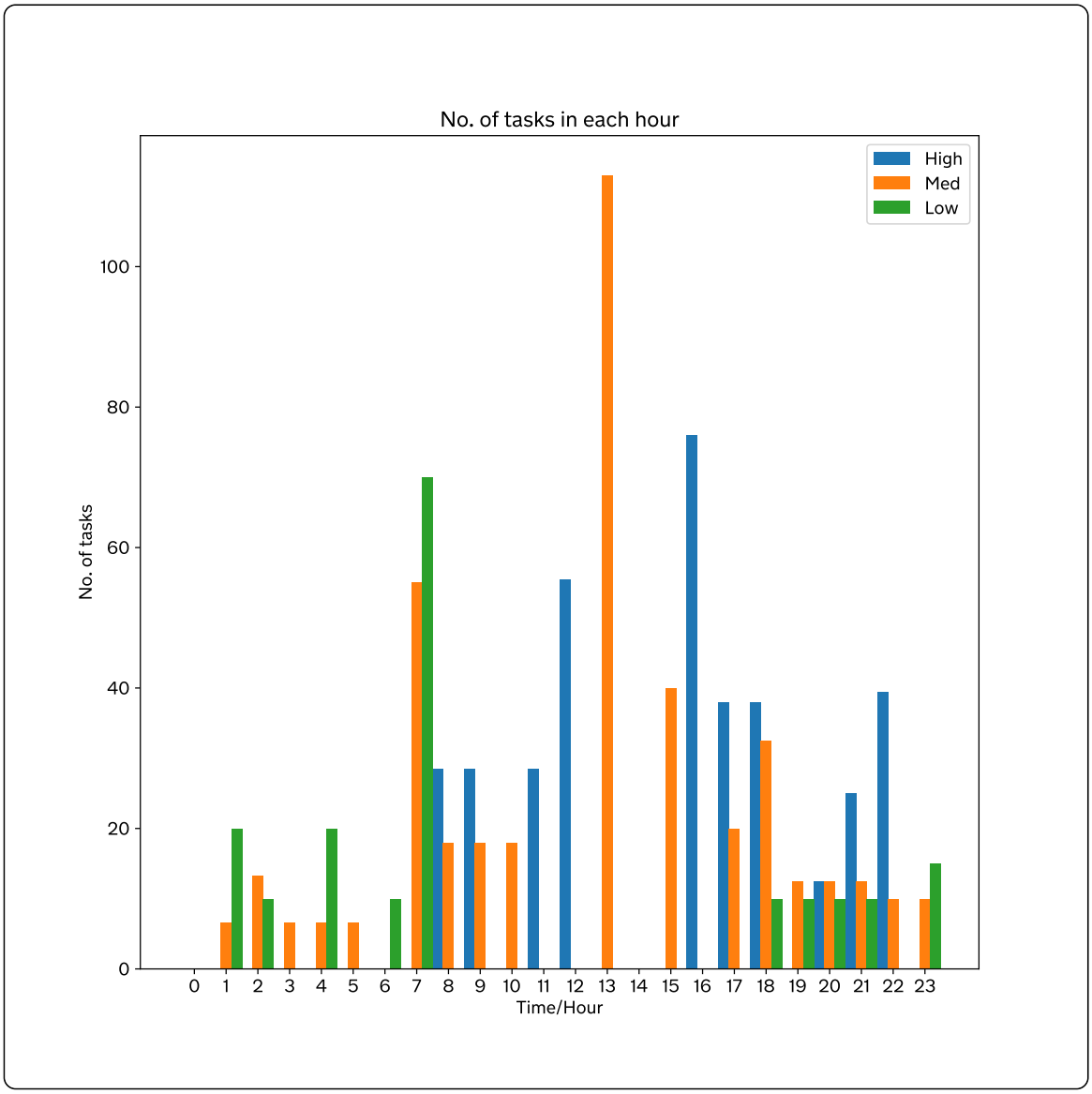


FIG. 1-1 The tasks described in a graph.

2 - Energy planning

The energy consumed by tasks are shown as follows.

Hour	Total energy (kWh)	Green energy (kWh)	Green energy %
0	300	300	100.0

Hour	Total energy (kWh)	Green energy (kWh)	Green energy %
1	933.3333333333334	933.3333333333334	100.0
2	1900.0	1900.0	100.0
3	2233.3333333333335	2233.3333333333335	100.0
4	3166.666666666667	2666.666666666667	84.21052631578948
5	3800.0000000000005	2400.0000000000005	63.15789473684211
6	4850.0	3050.0	62.88659793814433
7	8650.0	6550.0	75.72254335260115
8	11830.0	9430.0	79.71259509721048
9	15010.0	12610.0	84.01065956029313
10	15910.0	13110.0	82.40100565681962
11	18190.0	14990.0	82.40791643760308
12	22630.0	19230.0	84.97569597878922
13	28280.0	24980.0	88.33097595473834
14	28280.0	25180.0	89.03818953323905
15	30280.0	27380.0	90.42272126816381
16	36360.0	33760.0	92.84928492849285
17	40400.0	37900.0	93.8118811881188
18	45365.0	43065.0	94.93001212388405
19	46290.0	44790.0	96.75955930006481
20	48215.0	47215.0	97.92595665249404
21	51140.0	51140.0	100.0
22	54800.0	54800.0	100.0
23	55750.0	55750.0	100.0

The following bar graph shows the trend of energy usage.

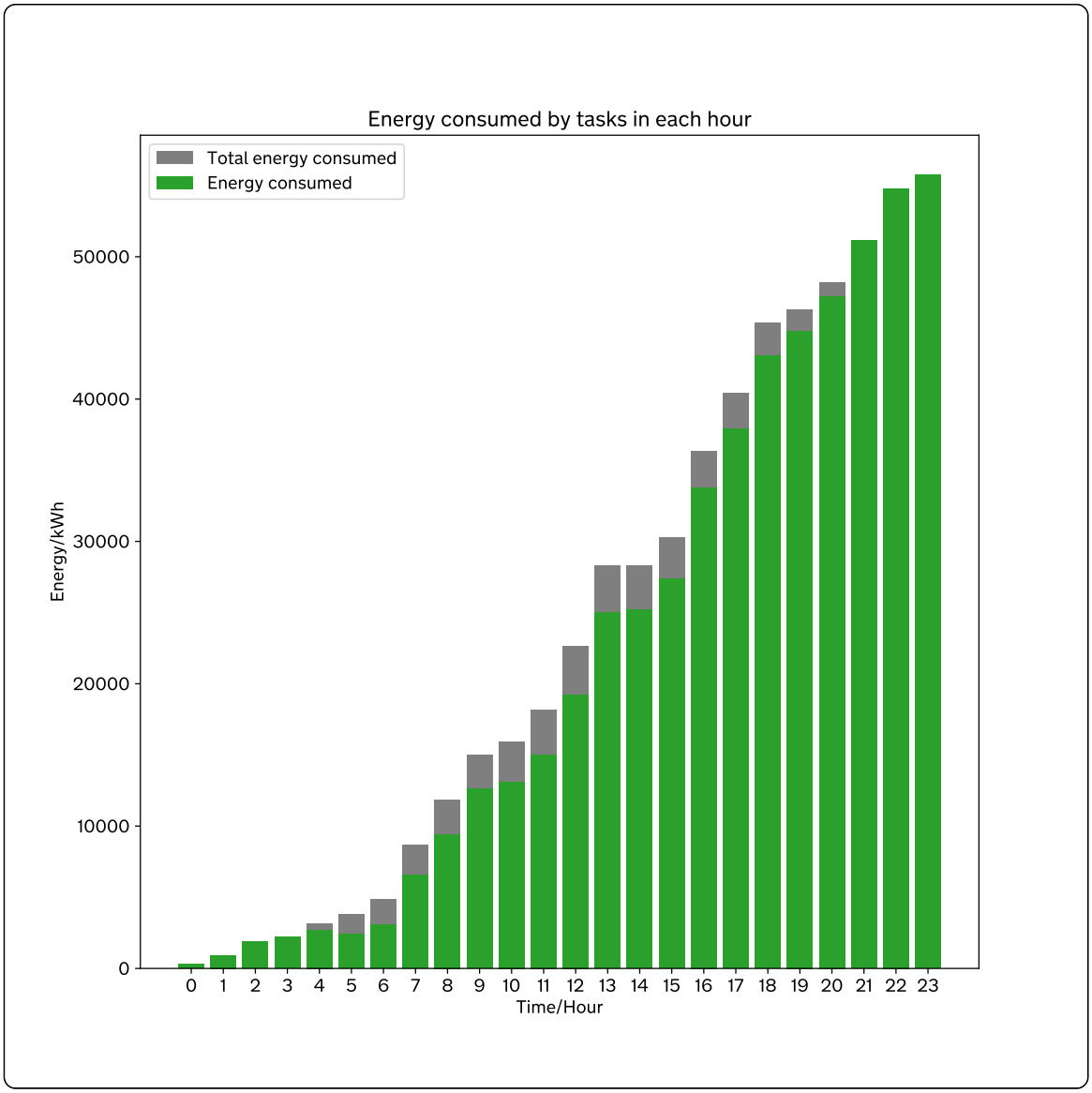


FIG. 2-1 The energy consumed described in a graph.

3 - Curve fitting

After curve fitting with a 9th-power polynomial, we obtained the following.

3-1 - Functions

The following are our polynomials after curve fitting with a 9th-power polynomial.

$$\begin{aligned}N_{high}(t) = 0.297808346409592t^{9}\\ +45.6485182325012t^{8}\\ -60.1924792216226t^{7}\\ +29.253711085704t^{6}\\ -6.79430967517511t^{5}\\ +0.866438823805813t^{4}\\ -0.064303494446529t^{3}\\ +0.00277294882625502t^{2}\\ -6.4463137361545 \cdot 10^{-5}t\\ +6.24881275450244 \cdot 10^{-7}\end{aligned}$$

$$\begin{aligned}N_{medium}(t) = 7.24718573883481t^{9}\\ +3.23059528348113t^{8}\\ -11.1504146340085t^{7}\\ +6.80560972459648t^{6}\\ -1.72380342413603t^{5}\\ +0.230262607365821t^{4}\\ -0.0175234269027381t^{3}\\ +0.000762758194576554t^{2}\\ -1.76748679195197 \cdot 10^{-5}t\\ +1.6908110086406 \cdot 10^{-7}\end{aligned}$$

$$\begin{aligned}N_{low}(t) = 0.297808346409592t^{9}\\ +45.6485182325012t^{8}\\ -60.1924792216226t^{7}\\ +29.253711085704t^{6}\\ -6.79430967517511t^{5}\\ +0.866438823805813t^{4}\\ -0.064303494446529t^{3}\\ +0.00277294882625502t^{2}\\ -6.4463137361545 \cdot 10^{-5}t\\ +6.24881275450244 \cdot 10^{-7}\end{aligned}$$

The following scatter graph shows the overall trend of the tasks, and the lines represent the fitted curves.

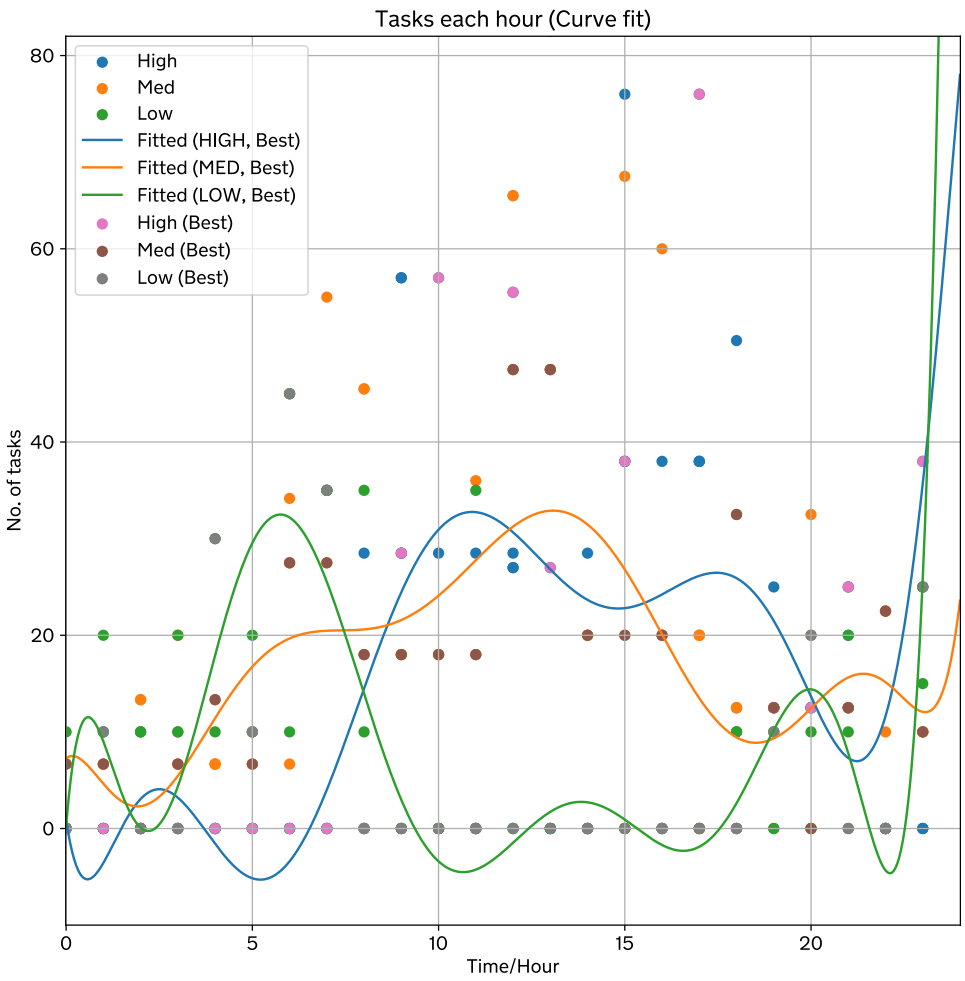


FIG. 3-1 Overall trend of all tasks, the lines show the fitted curves.

4 - Error evaluation

RSS The following are the RSS of our scenario.

Tasks (High)	Tasks (Medium)	Tasks (Low)
9488.170158471446	12297.42089022263	4053.669801287673

Error variance The following are the error variance (σ^2) in our scenario.

Tasks (High)	Tasks (Medium)	Tasks (Low)
395.3404232696436	512.3925370926096	168.90290838698635

Residuals The following graph shows the trend of residuals in the curve fitting process.

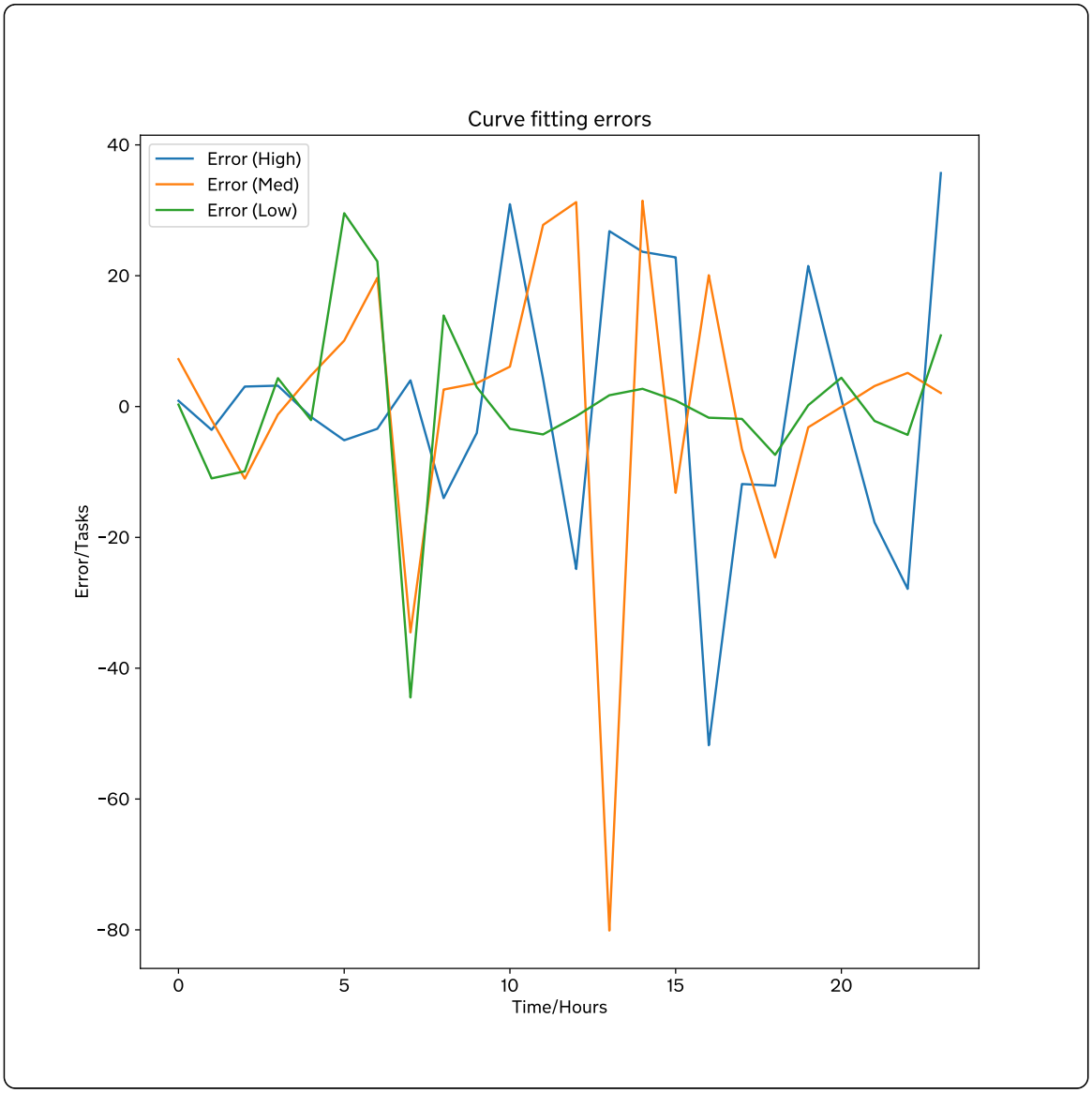


FIG. 4-1 Residuals in curve fitting.

SJC - PROBLEM E

LIMITATIONS

1 - Introduction

Mathematical modeling is indeed a powerful tool for solving complex problems, which unfortunately comes with several inherent limitations that can affect its applicability and accuracy in real-world scenarios. The limitations of the mathematical modeling approach described in the Appendix A are inherent in both the methodology and the practical implementation of the

task optimization process. These limitations highlight the challenges of balancing theoretical models with real-world constraints and achieving optimal solutions. Below is a detailed discussion of the most significant limitations:

Solution may not be optimized Our model often relies on loops to iterate through tasks and evaluate solutions. While this approach is computationally effective, it is not possible to fully explore all combinations of task delays and assignments, so it is not easy to find fully optimized solutions. This limitation is especially problematic for complex systems when the solution space is vast, and a more exhaustive search might be required to find more accurate results. As we unfortunately do not have enough time for long searches, it is not easy to find the optimal solution.

The use of randomization in delaying tasks and awarding marks also results in a stochastic process that prioritizes certain solutions over others purely by chance. While the program iterates through 10,000 simulations, even this number of iterations may not be sufficient to find the true global optimum in a highly complex solution space. Moreover, the model's reliance on heuristics, such as delaying tasks only until 23:00, imposes additional constraints that limit the scope of potential solutions.

Errors in the best-fit curve When we analyse the data, the program applies curve fitting to model the distribution of tasks across different priority levels (High, Medium, Low) over time. However, the best-fit curves are power polynomials, which, while useful for approximating trends, rarely represent real-world data perfectly. Although the residue sum of squares (RSS) is already minimised, deviations between the actual data points and the fitted curve can still introduce inaccuracies, leading to misinterpretation of task distributions or energy usage patterns. These mini errors made the model a little less reliable.

Not possible to fully balance usage of green energy and production costs One of the objectives of our model is to balance conflicting goals, such as minimizing energy costs and maximizing task marks. However, this balance is difficult to achieve in practice because the weights assigned to different priorities (e.g. energy consumption, delays, and task marks) are subjective and may not reflect real-world trade-offs accurately. For example, prioritizing renewable energy usage might result in higher delays, while minimizing delays might lead to increased reliance on traditional energy sources, which are costlier and less sustainable. We must admit that the model is unable to achieve a perfect equilibrium.

Time constraints and delays. The program enforces a strict rule that no task can be delayed beyond 23:59, as the question specifies a 24-hour plan. While this constraint simplifies the modeling process, it introduces a significant limitation in reflecting real-world scenarios. In many real-life applications, tasks can be deferred to subsequent days or periods to optimize resource utilization and cost. By restricting delays to a single day, the model sacrifices realism and flexibility, which might undermine its applicability to scenarios where tasks span multiple days.

Assumptions being made about energy supply and costs. The energy supply data (both renewable and traditional) and their respective costs are predefined and static in the model. While this simplifies calculations, it does not account for dynamic changes in energy availability or price fluctuations that occur in real-world markets. For instance, the renewable energy availability might vary due to weather conditions, or traditional energy prices could shift based on demand. The static nature of these inputs limits the model's ability to adapt to realistic energy supply scenarios.

Also, a power supply system has energy dissipated into the surrounding environment, so the estimated energy usage is less than the actual amount. Assuming linear power usage also cannot reflect the real-world scenarios, as computers use varying amount of energy when processing a task.

2 - Conclusion

In conclusion, while our approach provides a structured framework for solving the problem, it is constrained by several limitations. The reliance on looping mechanisms and randomization, errors introduced through curve fitting, and the inability to achieve truly optimal solutions all contribute to the model's shortcomings. Additionally, the rigid time constraints, difficulty in balancing competing objectives, and static assumptions about energy supply further reduce its applicability to real-world scenarios. Addressing these limitations through more sophisticated algorithms, dynamic inputs, and better optimization strategies would enhance the model's accuracy and usability.

Appendix A - Source Code

This is an attempt to use Python to solve problem E in IMMC 2024-25. This notebook can be split into the following parts:

1. Set up
    A. Import modules and set up formatting
    B. Set up energy supply data
    C. Define classes
    D. Set up variables
    E. Seeds a random-number generator
2. Try to find an optimal solution
3. Statistics
    A. Sum up all task values
    B. Curve fitting and output

1. Set-up

This part sets up everything needed for solving.

1.1 Importing modules

This part imports the necessary modules. It also loads fonts to make `matplotlib` graphs look a bit better.

In [1]:
```
import copy
import enum
import random
from time import time
from typing import Dict, List

import matplotlib.pyplot as plt
import numpy as np
import sympy as sp
from IPython.display import display
from sympy.abc import t
from tqdm.notebook import trange
from matplotlib import font_manager

sp.init_printing()
random.seed(time())

font_path = "TN-Normal.otf"
font_manager.fontManager.addfont(font_path)
prop = font_manager.FontProperties(fname=font_path)

plt.rcParams["font.family"] = "sans-serif"
plt.rcParams["font.sans-serif"] = prop.get_name()
plt.rcParams["font.size"] = 12
```

1.2 Set up energy supply data

- `AVAILABLE_ENERGY` defines the available green energy for each hour.
- `RENEWABLE_PRICE` defines the price of renewable energy for each hour.
- `TRAD_PRICE` defines the price for traditional electricity for each hour.

In [2]:
```
AVAILABLE_ENERGY = [
    0,
    0,
    0,
    0,
    500,
    1400,
    1800,
    2100,
    2400,
    2400,
    2800,
    3200,
    3400,
    3300,
    3100,
    2900,
    2600,
    2500,
    2300,
    1500,
    1000,
    0,
    0,
    0,
]

RENEWABLE_PRICE = [
    0.6,
    0.6,
    0.6,
    0.6,
    0.5,
    0.5,
    0.4,
    0.4,
    0.4,
    0.3,
    0.3,
    0.3,
    0.3,
    0.4,
    0.5,
    0.5,
    0.5,
    0.5,
    0.6,
    0.6,
    0.6,
    0.6,
    0.6,
```

```
    0.6,
]
TRAD_PRICE = [
    0.5,
    0.5,
    0.5,
    0.5,
    0.6,
    0.7,
    0.8,
    0.9,
    1,
    1.2,
    1.3,
    1.3,
    1.2,
    1.1,
    1,
    1,
    1.1,
    1.2,
    1.3,
    1.2,
    1.1,
    1,
    0.8,
    0.6,
]
```

1.3 Define classes

This part defines two classes.

- `Priority` is an `Enum` that stores the priority of the task. The value for each key is the amout of electricity needed (in kWh)
- `Task` is a task. It contains:

  Attributes

  - `start_hr` - Start time of the task (hour of the day)
  - `priority` - Priority of the task
  - `equiv` - Equivalent amount of workload, can be a `float`
  - `delay` - Delayed hours of the task

  Methods

  - `calc_mark` - Property method that calculates the 'mark' of the task. The algrithm is as follows

$$P = \begin{cases} 100\% \cdot 0.6 \cdot e^d, & \text{if } u = \text{HIGH} \\ 100\% \cdot 0.7 \cdot e^d, & \text{if } u = \text{MED} \\ 100\% \cdot 0.8 \cdot e^d, & \text{if } u = \text{LOW} \end{cases}, \text{where } \begin{cases} d & = \text{The delay in hour} \\ u & = \text{The urgency} \end{cases}$$

In [3]:
```
class Priority(enum.Enum):
    HIGH = 80
    MED = 50
    LOW = 30


class Task:
    award: float = 0

    def __init__(self, start_hr: int, priority: Priority, equiv: float, d
        self.start_hr = start_hr
        self.priority = priority
        self.equiv = equiv
        self.delay = delay

    def delay_task(self):
        self.start_hr += 1
        self.delay += 1

    @property
    def calc_mark(self) -> float:
        marks = 1
        if self.priority == Priority.HIGH:
            marks *= 0.6 * np.exp(self.delay)
        if self.priority == Priority.MED:
            marks *= 0.7 * np.exp(self.delay)
        if self.priority == Priority.LOW:
            marks *= 0.8 * np.exp(self.delay)
        marks *= (1 + self.award)
        return marks

    def __repr__(self) -> str:
        return f"<TASK grade={self.calc_mark:0.2f} at {id(self)}>"
```

1.4 Set up variables

The question defined a series of tasks. We are now going to add them to the list `ORIG_TASKS`.

In [118…
```
last_ttl_mark = 0
last_price = None
correct_updated_tasks: List[Dict[int, List[Task]]] = []
energies = {}
green_energies = {}
ORIG_TASKS = []

for i in range(6):
    ORIG_TASKS.append([Task(i, Priority.MED, 6 + 2 / 3), Task(i, Priority

for i in range(6, 8):
    ORIG_TASKS.append([Task(i, Priority.MED, 27.5), Task(i, Priority.LOW,

for i in range(8, 12):
    ORIG_TASKS.append([Task(i, Priority.HIGH, 28.5), Task(i, Priority.MED

for i in range(12, 14):
    ORIG_TASKS.append([Task(i, Priority.HIGH, 27), Task(i, Priority.MED,

for i in range(14, 18):
    ORIG_TASKS.append([Task(i, Priority.HIGH, 38), Task(i, Priority.MED,
```

```
for i in range(18, 22):
    ORIG_TASKS.append(
        [
            Task(i, Priority.HIGH, 12.5),
            Task(i, Priority.MED, 12.5),
            Task(i, Priority.LOW, 10),
        ]
    )

for i in range(22, 24):
    ORIG_TASKS.append(
        [
            Task(i, Priority.MED, 10),
            Task(i, Priority.LOW, 7.5),
        ]
    )
```

2. Try to find a optimal solution

This part is the most time-consuming part. It runs the following over and over again to get a solution.

1. Loop through all the tasks, and randomly deciding whether to delay it.
   - Delaying has the chance of $1/4$
   - Delaying never happens after 23:00, as the question asks for a 24-hour plan.
2. Calculate the energy needed for an entire hour, and, if the renewable energy at that hour is not enough, randomly decide whether to delay it.
   - Delaying has the chance of $1/2$
   - Delaying never happens after 23:00, as the question asks for a 24-hour plan.
   - Tasks will be awarded randomly if there is a delay.
3. Calculate the electricity price
4. Calculate the total marks of all tasks
5. If

   $$\Sigma_{i=1}^{n} T_{mark,i} > (\Sigma T_{mark})_{max}, \text{where } T_{mark} \text{ is the mark of a task}$$

   , and

   $\Sigma_{i=1}^{n} E_i < (\Sigma E_i)_{min}$, where E is the price of the energy consumed in an hour

   , replace $(\Sigma T_{mark})_{max}$ and $(\Sigma E)_{max}$ with the $\Sigma_{i=1}^{n} T_{mark,i}$ and $\Sigma_{i=1}^{n} E_i$.

In [119…
```
for _ in trange(10_000):
    updated_tasks = {x: [] for x in range(24)}
    energies_current = {x: [] for x in range(24)}
    green_energies_current = {x: [] for x in range(24)}
    orig_tasks = copy.deepcopy(ORIG_TASKS)
    ttl_mark = 0
    energy_used = 0
    price = 0

    for idx, hour in enumerate(orig_tasks):
        for task in reversed(hour):
            updated_tasks[idx].append(task)
            delay_prob = random.random() > 0.75
            if delay_prob and idx < 23:
                task.delay_task()
                updated_tasks[idx].remove(task)
                orig_tasks[idx + 1].append(task)
                continue
            energy_used += task.priority.value * task.equiv
            renewable_part = energy_used - AVAILABLE_ENERGY[idx]
            # If there isn't enough renewable energy, attempt to delay
            delay_prob = random.random() > 0.5
            if delay_prob and idx < 23:
                if task.priority == Priority.LOW:
                    task.delay_task()
                    task.award = random.random()
                    renewable_part += Priority.LOW.value * task.equiv
                    updated_tasks[idx].remove(task)
                    updated_tasks[idx + 1].append(task)

        # Process energy used → Find electricity price
        renewable_part = energy_used - AVAILABLE_ENERGY[idx]
        if renewable_part < 0:
            # No worries now, get traditional energy supply!
            trad_part = -renewable_part
            price += TRAD_PRICE[idx] * trad_part
        else:
            price += RENEWABLE_PRICE[idx] * renewable_part
        energies_current[idx] = energy_used
        green_energies_current[idx] = renewable_part

    for key, val in updated_tasks.items():
        for i in val:
            ttl_mark += i.calc_mark

    if not last_price:
        last_price = price

    if ttl_mark > last_ttl_mark and price < last_price:
        last_ttl_mark = ttl_mark
        last_price = price
        correct_updated_tasks.append(updated_tasks)
        energies = energies_current
        green_energies = green_energies_current
```

100% 10000/10000 [00:02<00:00, 3942.66it/s]

3. Statistics

This section does some statistics on the data.

3.1 Sum up all task values

As tasks are broken down into smaller, non-integer parts, we need to sum them up for statistics.

In [120…
```
data = []
for idx, cut_current in enumerate(correct_updated_tasks):
```

```
    data.append([])
    for key, val in cut_current.items():
        high = 0
        med = 0
        low = 0
        for i in val:
            if i.priority == Priority.HIGH:
                high += i.equiv
            elif i.priority == Priority.MED:
                med += i.equiv
            elif i.priority == Priority.LOW:
                low += i.equiv
        data[idx].append([high, med, low])
```

In [167…
```
%%capture
plt.title("Number of tasks for each hour")
plt.rcParams['figure.dpi'] = 100

plt.rcParams['figure.figsize'] = [10, 10]
```

3.2 Curve fitting and output

This part determines the best-fit line in the format

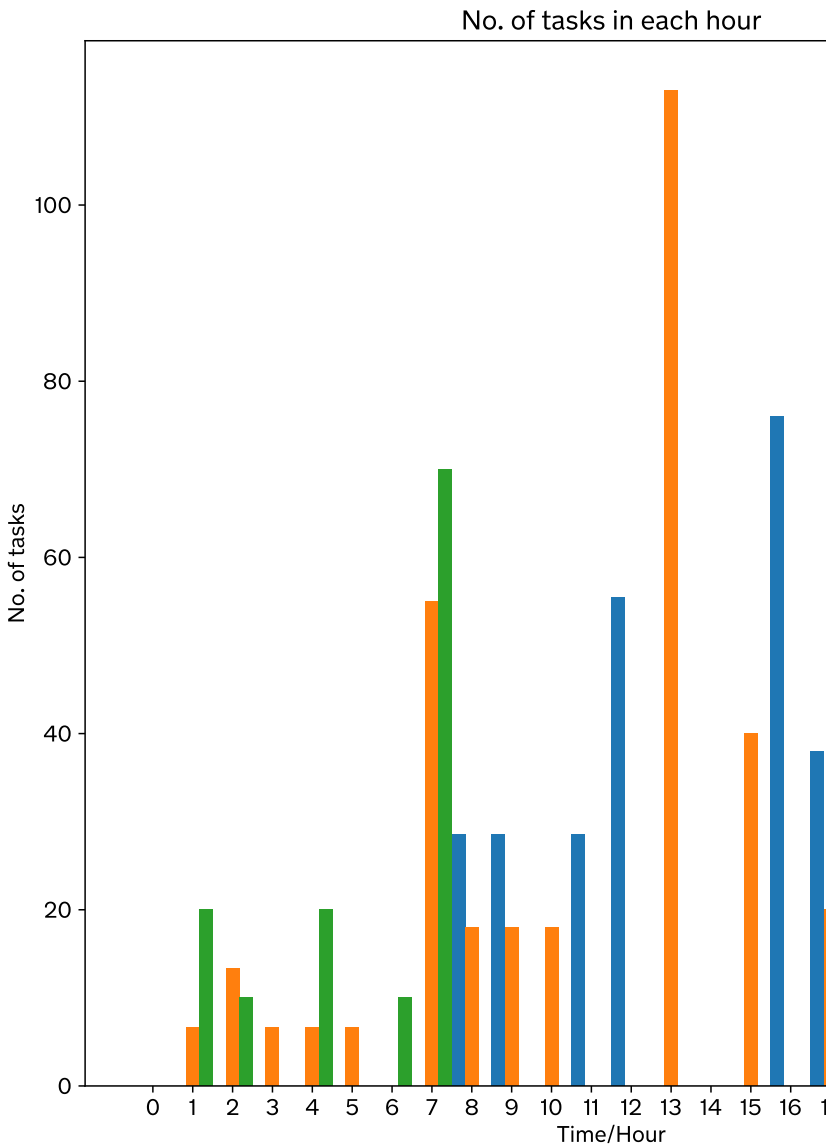
$$N_{\mathbf{TYPE}}(t) = k_9 t^9 + k_8 t^8 + k_7 t^7 + \cdots + k0 \text{ , where}$$
$$\begin{cases} N_{\mathbf{TYPE}} = \text{The number of tasks} \\ k_0 \cdots k_9 \in \mathbb{R} \\ t = \text{Time (hours)} \end{cases}$$

It is a 9^{th} power polynomial.

In [168…
```
result = np.array(data[-1])
np.insert(result, 0, values=np.arange(24), axis=1)
```

Out[168…
```
array([[ 0.        ,  0.        ,  0.        ,  0.        ],
       [ 1.        ,  0.        ,  6.66666667, 20.        ],
       [ 2.        ,  0.        , 13.33333333, 10.        ],
       [ 3.        ,  0.        ,  6.66666667,  0.        ],
       [ 4.        ,  0.        ,  6.66666667, 20.        ],
       [ 5.        ,  0.        ,  6.66666667,  0.        ],
       [ 6.        ,  0.        ,  0.        , 10.        ],
       [ 7.        ,  0.        , 55.        , 70.        ],
       [ 8.        , 28.5       , 18.        ,  0.        ],
       [ 9.        , 28.5       , 18.        ,  0.        ],
       [10.        ,  0.        , 18.        ,  0.        ],
       [11.        , 28.5       ,  0.        ,  0.        ],
       [12.        , 55.5       ,  0.        ,  0.        ],
       [13.        ,  0.        , 113.       ,  0.        ],
       [14.        ,  0.        ,  0.        ,  0.        ],
       [15.        ,  0.        , 40.        ,  0.        ],
       [16.        , 76.        ,  0.        ,  0.        ],
       [17.        , 38.        , 20.        ,  0.        ],
       [18.        , 38.        , 32.5       , 10.        ],
       [19.        ,  0.        , 12.5       , 10.        ],
       [20.        , 12.5       , 12.5       , 10.        ],
       [21.        , 25.        , 12.5       , 10.        ],
       [22.        , 39.5       , 10.        ,  0.        ],
       [23.        ,  0.        , 10.        , 15.        ]])
```

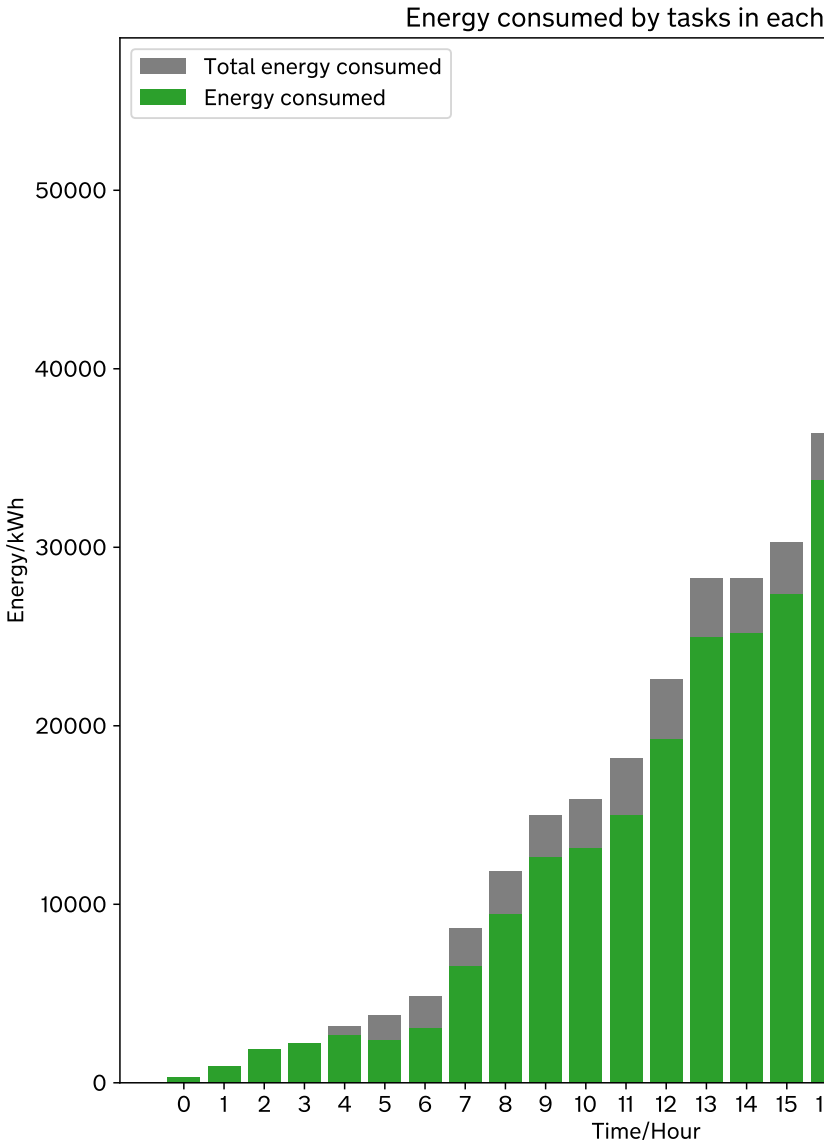
In [173…
```
plt.figure()

plt.bar(X - 1/3, result[:, 0], label="High", width=1/3, align="center")
plt.bar(X, result[:, 1], label="Med", width=1/3, align="center")
plt.bar(X + 1/3, result[:, 2], label="Low", width=1/3, align="center")

plt.title("No. of tasks in each hour")
plt.ylabel("No. of tasks")
plt.xlabel("Time/Hour")
plt.legend()
plt.xticks(X)

plt.savefig("A.png")
plt.show()
```



In [174…

```
plt.figure()
plt.bar(X, energies.values(), label="Total energy consumed", align="cente
plt.bar(X, green_energies.values(), label="Energy consumed", align="cente

plt.title("Energy consumed by tasks in each hour")
plt.ylabel("Energy/kWh")
plt.xlabel("Time/Hour")
plt.xticks(X)
plt.legend()
```

```
plt.savefig("E.png")
plt.show()

import pprint
pprint.pprint(energies)
pprint.pprint(green_energies)
for key, val in energies.items():
    print(green_energies[key] / val * 100)
```

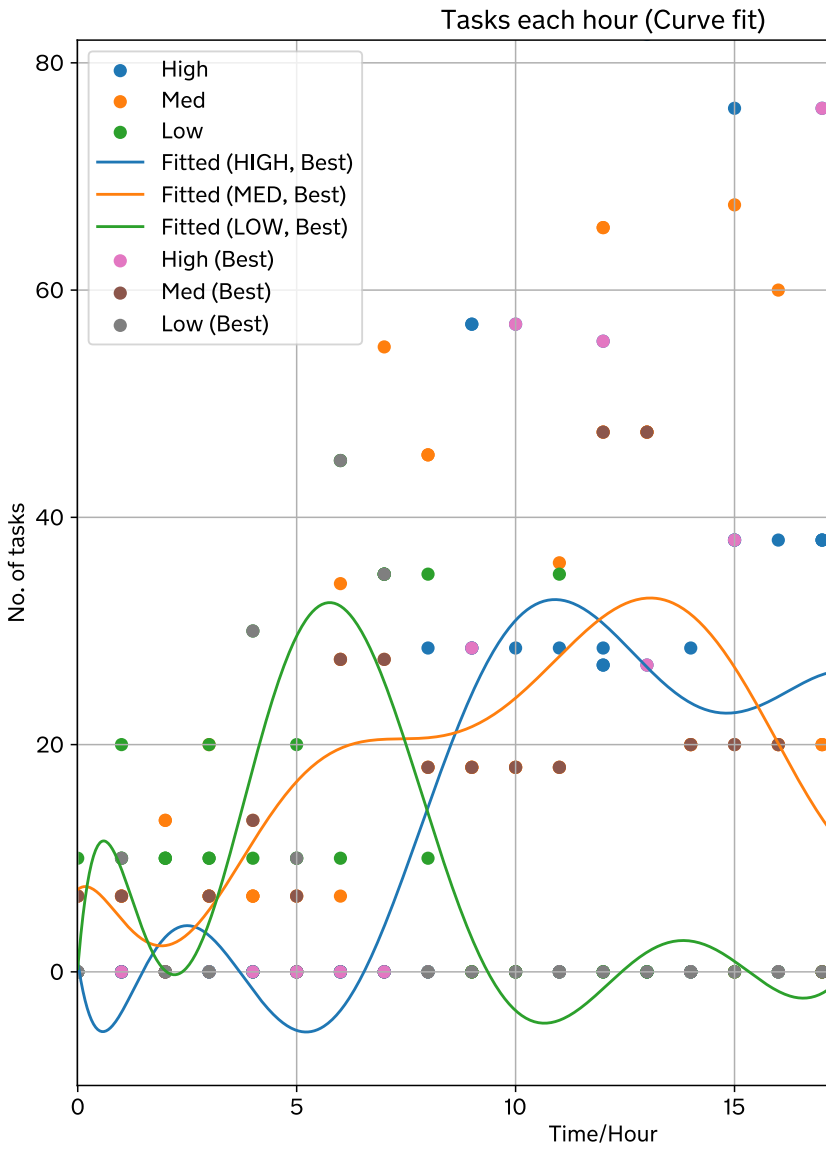


```
{0: 300,
 1: 933.3333333333334,
 2: 1900.0,
 3: 2233.3333333333335,
 4: 3166.666666666667,
 5: 3800.0000000000005,
 6: 4850.0,
 7: 8650.0,
 8: 11830.0,
 9: 15010.0,
 10: 15910.0,
 11: 18190.0,
 12: 22630.0,
 13: 28280.0,
 14: 28280.0,
 15: 30280.0,
 16: 36360.0,
 17: 40400.0,
 18: 45365.0,
 19: 46290.0,
 20: 48215.0,
 21: 51140.0,
 22: 54800.0,
 23: 55750.0}
{0: 300,
 1: 933.3333333333334,
 2: 1900.0,
 3: 2233.3333333333335,
 4: 2666.666666666667,
 5: 2400.0000000000005,
 6: 3050.0,
 7: 6550.0,
 8: 9430.0,
 9: 12610.0,
 10: 13110.0,
 11: 14990.0,
 12: 19230.0,
 13: 24980.0,
 14: 25180.0,
 15: 27380.0,
 16: 33760.0,
 17: 37900.0,
 18: 43065.0,
 19: 44790.0,
 20: 47215.0,
 21: 51140.0,
 22: 54800.0,
 23: 55750.0}
100.0
100.0
100.0
100.0
84.21052631578948
63.15789473684211
62.88659793814433
75.72254335260115
79.71259509721048
84.01065956029313
82.40100565681962
82.40791643760308
84.97569597878922
88.33097595473834
89.03818953323905
90.42272126816381
92.84928492849285
93.8118811881188
94.93001212388405
96.75955930006481
97.92595665249404
100.0
100.0
100.0
```

In [175…

```python
%config InlineBackend.figure_formats = ['svg']
for idx, item in enumerate(data[:-1]):
    item = np.array(item)
    series0 = plt.scatter(np.arange(24), item[:, 0], label="High", color=
    series1 = plt.scatter(np.arange(24), item[:, 1], label="Med", color="
    series2 = plt.scatter(np.arange(24), item[:, 2], label="Low", color="

    print(np.std(item, axis=0))

    X = np.arange(24)
    X_new = np.linspace(0, 24, 1000)
    TIMES = 9

poly1 = np.poly1d(np.polyfit(X, item[:, 0], TIMES))
plt.plot(X_new, poly1(X_new), label="Fitted (HIGH, Best)")

poly2 = np.poly1d(np.polyfit(X, item[:, 1], TIMES))
plt.plot(X_new, poly2(X_new), label="Fitted (MED, Best)")

poly3 = np.poly1d(np.polyfit(X, item[:, 2], TIMES))
plt.plot(X_new, poly3(X_new), label="Fitted (LOW, Best)")


plt.scatter(np.arange(24), item[:, 0], label="High (Best)", color="tab:pi
plt.scatter(np.arange(24), item[:, 1], label="Med (Best)", color="tab:bro
plt.scatter(np.arange(24), item[:, 2], label="Low (Best)", color="tab:gre

# Unify labels
handles, labels = plt.gca().get_legend_handles_labels()
by_label = dict(zip(labels, handles))
plt.legend(by_label.values(), by_label.keys(), loc="upper left")

plt.xlabel("Time/Hour")
plt.ylabel("No. of tasks")
plt.title("Tasks each hour (Curve fit)")
plt.xlim(-0.01, 24.01)
plt.ylim(-10, 82)
plt.grid()

plt.savefig("B.png")
plt.show()

display(sp.Poly(reversed(poly1.coef), t).as_expr())
display(sp.Poly(reversed(poly2.coef), t).as_expr())
display(sp.Poly(reversed(poly3.coef), t).as_expr())
```

```
[22.83485396 19.61699235 11.81270209]
[24.16982738 16.85339838 10.70233295]
[24.46198865 16.7722351  11.08482133]
[22.14331477 12.54805577 12.98871038]
```



$$0.884752168847447t^9 - 25.3923927819019t^8 + 33.4918203784285t^7 - 15.6785852351667t^6 + 3.53 + 2.77020428824635 \cdot 10^{-5}t - 2.52146368538672 \cdot 10^{-7}$$

$$7.24718573883481t^9 + 3.23059528348113t^8 - 11.1504146340085t^7 + 6.80560972459648t^6 - 1.723 - 1.76748679195197 \cdot 10^{-5}t + 1.6908110086406 \cdot 10^{-7}$$

$$0.297808346409592t^9 + 45.6485182325012t^8 - 60.1924792216226t^7 + 29.253711085704t^6 - 6.794 - 6.4463137361545 \cdot 10^{-5}t + 6.24881275450244 \cdot 10^{-7}$$

In [176…

```
best_fit = np.array(data[-1])
errors = [
    poly1(X) - best_fit[:, 0],
    poly2(X) - best_fit[:, 1],
    poly3(X) - best_fit[:, 2]
]

plt.figure()
plt.plot(X, errors[0], label="Error (High)")
plt.plot(X, errors[1], label="Error (Med)")
plt.plot(X, errors[2], label="Error (Low)")

RSS = [
    np.sum(errors[0]**2),
    np.sum(errors[1]**2),
    np.sum(errors[2]**2)
]
STD = [
    np.var(errors[0]),
    np.var(errors[1]),
    np.var(errors[2]),
]

print(f"{RSS=}")
print(f"{STD=}")

plt.title("Curve fitting errors")
plt.ylabel("Error/Tasks")
plt.xlabel("Time/Hours")
plt.legend()

plt.savefig("C.png")
plt.show()
```

```
RSS=[np.float64(9488.170158471446), np.float64(12297.42089022263), np.floa
t64(4053.669801287673)]
STD=[np.float64(395.3404232696436), np.float64(512.3925370926096), np.floa
t64(168.90290838698635)]
```