Autonomous Sustainable Buildings: From Theory to Practice

Mario Bergés

2026-04-07

Table of contents

Preface

Buildings account for a large fraction of our total energy use. Any realistic decarbonization plan needs to consider them. Moreover, though technological upgrades (such as better insulation, newer and more efficient appliances, etc.) can improve their efficiency significantly, there is a relatively untapped potential for autonomous technologies to squeeze out much more out of the input energy sources feeding our building stock.

This book (and website) contains a collection of lecture notes for the Fall 2026 edition of the course (12-770: Autonomous Sustainable Buildings) at Carnegie Mellon University. This course intends to prepare students to become leaders in this nascent field that combines fundamentals of physics and building science with concepts from computer science, statistics and mechanical/electrical engineering, to name a few.

Notably, the lecture notes are —to a great extent— generated through GenAI tools (mostly Claude Code, with Sonnet 4.5) after careful prompting and many rounds of revision. This effort represents an experiment and, as any experiment, should be treated with humility and an open mind.

My hypothesis for the experiment is that it should be possible for me to carefully extract the relevant knowledge that is stored in modern LLMs (through a lossy compression process) in order to generate useful lecture notes that go well beyond what I would be able to write myself in the same time, all while providing additional utility to the students. This additional utility would come in two flavors: (1) the lecture notes can now contain interactive elements based on code snippets that can facilitate building intuition and understanding difficult concepts; (2) the students are encouraged to be consciously and actively suspicious of the material that they are reading, given that it could contain errors.

This hypothesis might be proven correct, or incorrect. I don't know what the answer is but I am curious about it. Obviously, with a small sample size of the students in this class and only self-reported measures of success, it will be hard to assess the hypothesis in general. But it's worth trying. I already tried once in Fall 2025 for a different course and obtained positive results.

To incentivize students to be more critical readers of the material contained in this book, I am awarding each student 1 point (worth 1% of your final grade) for each errors found in the content (submitted to me via e-mail) up to a total of 5 points per student. It is also possible to obtain this same point by contributing written

knowledge to the website/book. You can do so via e-mail or (preferably) by submitting a pull request to the git repository for the book.

If you're wondering about the website design and layout of this book: I'm using Quarto. To learn more about Quarto books visit https://quarto.org/docs/books.

Part I

Logistics

1 Syllabus

12-770 | Sec. A | Spring 2026

TR 12:30PM - 1:50 PM ET Location: GHC 5222

Instructors: Mario Bergés, PH 123L, Phone: x8-4572 **Office Hours:** Thursdays 3:00 PM - 4:00 PM ET in PH 123G and by appointment.

Teaching Assistant: Chenrui Xu (chenrui2@andrew.cmu.edu) **Office Hours: TBD**

Textbooks (optional):

1. Though no official textbooks are required, the instructor will draw from multiple sources and post the relevant references on Canvas.

Prerequisites: ((12-740 + 12-741) | 12-778) + Python

1.1 Official Course Description

We spend a significant portion of our lives inside buildings: working, sleeping, and on leisure activities. Unsurprisingly, buildings are responsible for over 40% of our annual greenhouse gas emissions. The history of buildings goes hand in hand with the history of energy efficiency, as we have moved from non-renewable/inefficient fuel sources and technologies (e.g., firewood, cookstoves) to more renewable/efficient ones (e.g., solar energy, heat pumps). Increasing efficiency has also resulted in tighter integration between buildings, their systems, and the supporting services. All of these trends have led to an explosion in the number of instrumentation systems (for monitoring and/or controlling) installed in buildings, and an associated increase in the number and complexity of the decisions that are being (and can be) made in light of these new instruments. Autonomous technologies (which make decisions on our behalf in order to achieve pre-established goals) are well suited to address the challenges that these information-rich and highly-interconnected buildings pose.

With a focus on real-world deployments, case studies and group projects, this course will cover the theory and emerging practice of retrofitting existing buildings with hardware and software to significantly increase their autonomy and overall sustainability. The focus will be primarily on the operational stage of the life-cycle of buildings, and particularly on HVAC, electrical and water systems within them. In particular, this course will expose the students to recent advances in the quest to endow buildings with the ability to operate autonomously (i.e., with minimal human assistance), in order to provide the services they were designed for, while maintaining quality of service and upholding human values such as privacy, equity and environmental sustainability. The course will be based on lectures, assignments and a final project where the students will have the opportunity to design and implement an autonomous technology and evaluate it through a hardware-in-the-loop experiment in the lab. This course builds upon machine learning, statistics, data acquisition and instrumentation, linear systems and (some) control theory.

It is intended to be a upper-graduate level course (i.e., for Ph.D. students or senior M.S. students) interested in gaining practical exposure to the state-of-the-art in data science for building systems. As such, the format of the course is tailored to that experience and will include reading and critiquing recent publications in the field, learning to implement data analysis techniques described in them, and producing novel results using these newly acquired skills.

The course assumes students are familiar with concepts in instrumentation, linear algebra, probability, statistics and programming, though this is not a strict requirement if the student has previously discussed with the instructor and has received approval.

1.2 Informal Course Description

I have been doing research on buildings and energy efficiency (in the so-called "smart buildings" area) for almost two decades. However, and embarrassingly, it was not until recently that I had the opportunity to carefully study the physics and scientific theories that describe our understanding of how buildings behave and consume energy (i.e., the so-called "building science"). To put it simply, I have largely focused on the *smart* aspects of "smart buildings" and kept my understanding of the rest at the bare minimum required to come up with good solutions. The reason for this gap in my knowledge is not a lack of interest; but the incentives in academia are such that there are virtually only two ways to expand one's knowledge on fundamentals: by taking a sabbatical, or by teaching a course. I chose the latter and this the real reason for why this course exists.

I learn best through practice and by having specific motivations for acquiring knowledge. Given this, when I started the course I thought it would be most interesting if the specific theories and concepts that are covered in the course are directly connected to a smart buildings application that we can build from the ground up. This means that we will not cover all of building physics (far from it!), but it is a trade-off that will serve us well. In time, my plan is to increase my

coverage of fundamental topics by introducing new final projects with each new edition of the course.

I'm splitting the course into three approximately equal-length periods: *concepts*, *case studies* and *implementation*. During the first third, *concepts*, we will learn about the scientific theories that are most relevant for the specific final projects options that are on offer and, during the *case studies* period we will review specific solutions that have been proposed in the (recent) literature to solve the problems that motivate the final project options. The last third of the course will be dedicated to the final project with lecture time being re-purposed for lab sessions (i.e., step-by-step instructions on how to get started with each project), project feedback sessions and hands-on guidance with the specific project implementations by each team.

2 Grading

Component	Weight
Assignments	40%
Interim Project Progress Update	15%
Written Knowledge Contributions	5%
Final Project Report and Demo	40%
- Written Report	30%
- Demo	10%

2.1 Assignments

A total of four assignments are scheduled. The topics covered in each assignment will closely follow the concepts and theory covered during the first third of the course, as well as the academic manuscripts that will be reviewed during the second third.

All assignments are to be solved individually. Discussions and conversations with other students regarding the problem sets are encouraged. However, the final solutions along with the reasoning behind them need to come from you and be clearly explained in the submitted documents.

Each assignment will be worth 10% and is due at the beginning of class on the date that is indicated in each assignment. Assignments that are submitted before this deadline can receive 100% of the available credit. There is a 3 day grace period for late submissions, with the following available credit in each of those five days: 1 Day Late: 90%, 2 Days late: 70%, 3 Days late: 40%. After this time, assignments will not be graded. Of course, if you anticipate not being able to meet this schedule due to a major problem, please talk to the instructor as soon as possible.

2.2 Written Knowledge Contributions

A small pprtion (5%) of the grade for the course will be based on the individual contributions each of you make to the body of knowledge we are collectively capturing on a website/book for the course. The main way you will gain points here is by suggesting corrections for errors, ommissions or other possible improvements to the content. This year I'm running an experiment

to test whether pre-trained LLMs from frontier labs can generate the content with enough prompting and guidance, so I expect there will be multiple opportunities to spot errors. The other way to contribute and earn these points is by submitting your own content: terms, definitions, methods, summaries, worked-out exercises, diagrams, and other contributions are fair game for this purpose. The website is hosted on a private Gitlab instance, and will require that you log in with an account in order to edit it. This will give me the opportunity to evaluate your contributions by simply following the commit log of the repository.

2.3 Project Update

Another small portion (15%) of the grade for the course will be based on a progress update that will take place during the last period of the course. This update will consist of: (1) a written 2-page report, authored by all team members; and (2) a 10 minute individual meeting with the instructor to discuss the project, its goals and the plan forward. The written progress report will confer 10% of the final grade, while the individual meeting discussion will be used to provide the remaining 5%.

2.4 Final Project Report

The final project report and demonstration are worth almost half of the total grade and, in some ways, are the most important assignment. The written report is worth 30% and the demonstration 10% of the total grade. For the written report, we are asking that you submit it using the ACM BuildSys template (either LaTeX or MS Word). You should use your own words when writing it and avoid plagiarism of any kind. In it you will describe, in simple terms, the motivation and specific objectives of your project, the design choices made for the hardware and or software prototype that you put together, the experiments you performed to validate whether or not your solution satisfies the objective, and a discussion about the results and limitations. You should make all code and datasets available as part of your submission.

The rubric that will be used for grading the written report is as follows:

- Formatting and organization (15%)
- Grammar, clarity and accuracy of ideas (15%)
- Display of mastery of concepts covered in class (35%)
- Creativity expressed in the solution (10%)
- Depth of discussion related to the pros/cons of the implemented solution (15%)
- Overall assessment of the project's idea nd execution (10%)

3 Course Policies

Though there are definitely reasons to like anarchism, I still prefer democracy, so here are the rules of the game as they are now (and subject to change if enough of you request that I do).

3.1 Collaboration

Collaboration is expected within the limits of discussing concepts and problems. However, each student must produce his/her own solution to the problems. Copying from another student's assignment is clearly plagiarism. Using information directly from websites, books, papers and other literary sources without appropriate attribution is also plagiarism. Assignments submitted for this class will be reviewed by the instructor and TA and may be scanned through web-based academic integrity software. Occurrences of cheating or plagiarism will be handled according to the university policy on Academic Integrity, https://www.cmu.edu/policies/documents/Academic%20Integrity.htm. Students are expected to have read this policy and conform to the highest standards of academic integrity. For incidents of academic misconduct, the University Academic Disciplinary Actions Policy, found at https://www.cmu.edu/student-affairs/theword/acad_standards/creative/disciplinary.html, will be followed.

3.2 Class Participation

Students are expected to be in class on time and participate in class discussions. If you cannot make class, please inform your instructors and group members ahead of time. In class, students are expected to be courteous and respectful of the views and needs of other students and instructors.

3.3 Students with disabilities

Students requesting classroom accommodation must first register with the Dean of Students Office. The Dean of Students Office will provide documentation to the student who must then provide this documentation to the Instructor when requesting accommodation.

3.4 Posting of course materials

All the material used in the course (syllabus, readings, problem sets, reports) is intended for use in the class only. No unauthorized posting, publication or redistribution is expected. Uploading course materials to Course Hero or other web sites is not an authorized use of the course material.

4 Course Outline

Here is a more detailed overview of how the lectures will unfold over the three periods. Most of this is subject to change. The list of "learning objectives" should be understood as responding to the phrase *By the end of the lecture, students should be able to...*

4.1 First third: Concepts and Tools

This period covers the first six weeks of classes. The goal is to introduce the theory behind the operation of heating, ventilation and air conditioning, which includes the science of heat flows in buildings, occupant thermal comfort and occcupant behavior models, as well as simple control theory concepts, such as Proportional Integral Derivative (PID) control; and even some of the policy, control and institutional context relevant to HVAC systems. We will also use some time to get to know each other better and to set up our learning and tinkering environments for the rest of the semester, as well as to decide the specific project option you will be working on.

ℹ Lecture #1: Course Introduction

Learning Objectives:

- Recognize the relative importance of the assessed components of the course (final project, assignments and written contributions)
- Get to know other students in the course and their motivation for joining it
- Understand the goals of the course and its structure

References:
No references for this week.

ℹ Lecture #2: Building Energy Management: The Good, Bad, and Ugly

Learning Objectives:

- Be able to identify *sense*, *plan* and *act* components in an autonomous system.
- Be familiar with the scope and objectives of the final project
- Be able to recite basic facts about global energy demand and generation.
- Reason about energy demand growth trends and their impact

References:
A good companion paper for this lecture is González-Torres et al. (2022). The three projects from my own lab that were referenced during the lecture were:

- *Contactless sensing of appliance state transitions through variations in electromagnetic fields* Rowe et al. (2010, PDF)
- *Brick: Towards a unified metadata schema for buildings* Balaji et al. (2016, PDF)
- *Gnu-RL: A Precocial Reinforcement Learning Solution for Building HVAC Control Using a Differentiable MPC Policy* Chen et al. (2019, PDF)

Lecture #3: Setting up your learning / tinkering computer environment

Learning Objectives: - Understand how their computers need to be configured for maximum efficiency in working on the Final Project and assignments - Have a working Python installation on their computer, with Jupyter Notebook and uv as the package and project manager - Understand how to use uv and Jupyter Notebooks to: - Create, load, edit and run Jupyter notebook files - Create and manage Python projects and virtual environments - Create Jupyter kernels specific to a uv Python project - Create and access a public git repository on Github - Use git to clone, commit and push changes to a remote repository - Ensure that their computer has a working SSH client and understand the value of this tool
References:

- A cheat sheet for Python: [PDF]
- An animated tutorial into the inner-workings of Git: [URL]
- A simple Git tutorial: [PDF]

Lecture #4: Overview of the Home Assistant Ecosystem

Learning Objectives: - Understand the hardware and software ecosystem components for Home Assistant and its value for the final project - Work with a local Home Assistant container on which to test functionality before deploying things to the Home Assistant Green hub - Set up a Home Assistant Green hub and have it receive voice commands with the Voice Preview - Set up integrations to sensors - Set up one automation routine - Create a simple external controller that interacts with the HA server via WebSockets
References: TBD

Lecture #5: Thermal Dynamics of Building: Part I

Learning Objectives: - Name the main processes by which heat loss takes place in buildings - Recognize the value of thinking about heat lossess as stemming from the product

of building's leakiness and the temperature demand - Be able to mathematically model the different modes of heat transfer (conduction, convection and radiation), understanding the underlying physical laws underpinning them - Have familiarity with different building materials and their thermal conductivity as well as the thermal conductance of a material assembly - Represent wall and window assemblies as networks of thermal resistances and capacitances
References:

- Chapter 2 of Reddy

Lecture #6: Thermal Dynamics of Buildings: Part II

Learning Objectives: - Derive thermal network models from first principles: - Describe the elelments of the Fourier heat equation (the 1-D Fourier transient heat conduction PDE and its steady-state version), including its solution as a sum of steady state plus an infinite sum of exponentials. - Recognize the link between the Fourier heat equation PDE and the Thermal Network Model, understanding it as a finite difference approach using physical construction-based slicing/lumping that leads to a set of N simultaneous first-order differential equations to be solved. - Understand the link between Kirchoff's voltage/current laws and the heat balance equations we can work out in thermal network models. - Recognize that the linearity assumption means with respect to the behavior of network elements (independent of temperature and time), and other implications. - Directly represent radiation heat transfer interactions, lumped building mass components, solar radiation, heat extraction by HVAC, infiltration heat paths. - Solve 1R1C and 2R2C networks to find a building's time constants - Understand the importance of time constants even if they are just approximations - Write down the linear state-space representation of an arbitrary thermal network model
References: TBD

- Chapter 8 of Reddy

Lecture #7: Occupant Thermal Comfort

Learning Objectives: - Recreate the basic equation for the thermal balance of a human body, which gives rise to expressions about heat dissipation to the environment - Understand latent heat (and its distinction from sensible heat, which is what we've covered so far), and Fick's law for moisture transfer (relating it to Fourier's law for conduction) - Be able to calculate shape factors of simple symmetric geometries, and understand their relevance in radiative heat transfer - Derive the Mean Radiant Temperature and its approximation when surfaces have similar temperature - Derive the concept of "Operative Temperature" by starting with convective and radiative heat transfer equations between

the outer clothing and the environment - Be familiar with clothing insulation values (and their units of *clo*) - Calculate total conductive plus radiative sensible heat loss for a person if given their metabolic rate and clothing insulation (as well as the operative temperature). - Use the Predictive Mean Vote method for caluclating a thermal sensation index - Reason about comfort levels using ASHRAE's comfort chart relating operative temperature and humidity ratio.
References:

- Chapter 3 of Reddy

Lecture #8: Basics of AC Power Systems for Buildings

Learning Objectives: - Understand how power is delivered from the grid to the electrical appliances/loads in a building, including: - The mathematical basis for AC power delivery: - Make use Ohm's Law and Kirchhoff's Voltage/Current Laws for DC circuits in order to derive an expression for power provided to an arbitrary RLC load. - Distinguish between power losses and power transmitted in a circuit - Understand the steady-state and transient effects of the R, L, and C compoments for the load in the DC case. - Recreate the expression for power under an AC voltage/current source an an arbitray impedance - Recognize the steady-state and transient effects of the impedance (resistance + reactance) in the circuit for the AC case. - Be able to use time domain (i.e., instantaneous) power expressions, as well as phasor and/or complex representations. - The underlying physical mechanism from which sinusoidal voltage sources arise: - Understand the working principles and components of an AC electrical generator in general, and 3-phase generators in particular - Derive an expression for the voltage at each of the 3 phases of the generator (mathematically showing their phase difference) - Understand how power delivered/consumed and power generated need to be balanced, and the effect of this balance on the grid's frequency - Describe, conceptually, how power from the generator gets transmitted to the distribution system and to the buildings in it. - Understand the role of transformers in the transmission/distribution grids - Understand how split-phase systems to residential buildings in the US work - Analyze AC Power Usage - Starting from a known voltage source $v(t)$ and current $i(t)$, derive expressions for active, reactive, apparent and instantaneous power, as well as power factor - Understand the value of RMS and average quantities for voltage and current over some cycle-aligned period - Be able to leverage the phasor representation of these quantities - Calculate power quantities (active, reactive, etc.) from RMS or sub-cycle measurements of voltage and current - Be familiar with the VI trajectory (a single AC cycle plot of current versus voltage) as a visualization aid to differentiate between different load impedances.
References: TBD - Power in an AC Circuit from the University of Central Florida.

Lecture #9: Basics of Control Theory for Buildings and Their Application

Learning Objectives: - Recognize the different branches of control theory, and where optimal control sits - Identify the components of a Model Predictive Control problem formulation for a given context: cost functional, constraints, dynamics model, decision variables, etc. - Identify how the choice of constraints, cost and variables determines the type of optimization problem and the available solutions - Use linear models of thermal dynamics (e.g., RC thermal networks) to formulate simple MPC problems - Use a solver (e.g, `cvxpy`) to implement an MPC controller for a 1R1C thermal model - Recognize the value of a standardized environment for benchmarking and comparing different control algorithms - Identify the major endpoints of the BOPTEST API - Test a controller against a BOPTEST testcase
References: - Blum, David, Javier Arroyo, Sen Huang, Jan Drgona, Filip Jorissen, Harald Taxt Walnum, Yan Chen, et al. "Building Optimization Testing Framework (BOPTEST) for Simulation-Based Benchmarking of Control Strategies in Buildings." *Journal of Building Performance Simulation* 14, no. 5 (September 3, 2021): 586–610. https://doi.org/10.1080/19401493.2021.1986574 - Drgoňa et al. "All you need to know about model predictive control for buildings" *Annual Reviews in Control*, Volume 50 (2020): 190-232. https://www.sciencedirect.com/science/article/pii/S1367578820300584

Lecture #10: TBD

Learning Objectives:
References: TBD

Lecture #11: TBD

Learning Objectives:
References: TBD

Lecture #12: TBD

Learning Objectives:
References: TBD

4.2 Second third: Case Studies

This period covers the four weeks that follow. The goal during this period is to use the knowledge we gained in the first third and directly tackle recent academic publications describing smart thermostat and smart metering solutions. The "smart" part of these systems will likely need

to be further dissected, as much of that will not have been covered during the first third. We will continue to make heavy use of our collaborative website to jointly acquire the background (and foreground) knowledge that these publications require and convey.

ℹ Lecture #13: Project Proposals

Learning Objectives:
References: None.

ℹ Paper #1: FTM-Sense: Robust Sensor-free Occupancy Sensing Leveraging WiFi Fine Time Measurement

Learning Objectives:
References: Nikseresht, Fateme, and Bradford Campbell. "FTM-Sense: Robust Sensor-free Occupancy Sensing Leveraging WiFi Fine Time Measurement." BuildSys 2023. DOI

ℹ Paper #2: PyDCM: Custom Data Center Models with Reinforcement Learning for Sustainability

Learning Objectives:
References: Naug, Avisek, Antonio Guillen, Ricardo Luna Gutiérrez, Vineet Gundecha, Sahand Ghorbanpour, Lekhapriya Dheeraj Kashyap, Dejan Markovikj, Lorenz Krause, Sajad Mousavi, Ashwin Ramesh Babu, and Soumyendu Sarkar. "PyDCM: Custom Data Center Models with Reinforcement Learning for Sustainability." BuildSys 2023. DOI

ℹ Paper #3: Gnu-RL: A Precocial Reinforcement Learning Solution for Building HVAC Control Using a Differentiable MPC Policy

Learning Objectives:
References: Chen, Bingqing, Zicheng Cai, and Mario Bergés. "Gnu-RL: A Precocial Reinforcement Learning Solution for Building HVAC Control Using a Differentiable MPC Policy." BuildSys 2019. DOI

ℹ Paper #4: Modeling the Impact of Passive Ventilation Systems on Multi-Zone Thermal Dynamics

Learning Objectives:
References: Onyejizu, James, Sandipan Mishra, and Koushik Kar. "Modeling the Impact of Passive Ventilation Systems on Multi-Zone Thermal Dynamics." BuildSys 2024. DOI

Paper #5: Can Attention Improve Sequence-to-Point Load Disaggregation: A Comparative Assessment

Learning Objectives:
References: Bouchur, Mazen, Nan Li, and Andreas Reinhardt. "Can Attention Improve Sequence-to-Point Load Disaggregation? A Comparative Assessment." BuildSys 2025. DOI

Paper #6: RECA: A Multi-Task Deep Reinforcement Learning-Based Recommender System for Co-Optimizing Energy, Comfort and Air Quality in Commercial Buildings

Learning Objectives:
References: Xia, Stephen, Peter Wei, Yanchen Liu, Andrew Sonta, and Xiaofan Jiang. "RECA: A Multi-Task Deep Reinforcement Learning-Based Recommender System for Co-Optimizing Energy, Comfort and Air Quality in Commercial Buildings." BuildSys 2023. DOI

Paper #7: Hypertemporal Imaging of NYC Grid Dynamics

Learning Objectives:
References: Bianco, Federica B., Steven E. Koonin, Charlie Mydlarz, and Mohit S. Sharma. "Hypertemporal Imaging of NYC Grid Dynamics." BuildSys 2016. DOI

Paper #8: Data Predictive Control for Peak Power Reduction

Learning Objectives:
References: Jain, Achin, Rahul Mangharam, and Madhur Behl. "Data Predictive Control for Peak Power Reduction." BuildSys 2016. DOI

Paper #9: BOLT: Energy Disaggregation by Online Binary Matrix Factorization of Current Waveforms

Learning Objectives:
References: Lange, Henning, and Mario Bergés. "BOLT: Energy Disaggregation by Online Binary Matrix Factorization of Current Waveforms." BuildSys 2016. DOI

4.3 Final third: Implementation

This period covers the last four weeks of class, from April 5 to April 28. This is when we finally get our hands dirty and start stepping on the shoulders of giants to create our novel solutions

to the problems represented in the two final project options. This period will begin with considerable hand-holding, starting with a review of the background instrumentation theories required to understand the sensors/actuators in the projects, as well as a tutorial on how to set up the basic systems. Then, each group will have the opportunity to get individualized feedback from the instructors as they prepare for the final project's demonstration and report submission.

i Lecture #21: Temperature and Environmental Measurement Kit

Learning Objectives:
References: TBD

i Lecture #22: Occupancy Monitoring Kit

Learning Objectives:
References: TBD

i Lecture #23: Project Progress Reports

Learning Objectives:
References: None.

i Lecture #24: Project Feedback / Guidance Session

Learning Objectives:
References: None.

i Lecture #25: Project Feedback / Guidance Session

Learning Objectives:
References: None.

i Lecture #26: Project Feedback / Guidance Session

Learning Objectives:
References: None.

i Lecture #27: Guest Lecture: What can I do with these skills in the job market?

Learning Objectives:
References: TBD

5 Schedule

The schedule is subject to change. Here's the most updated version:

Week	Date	Lecture	Notes
1	Jan 13 (Tue)	**Introduction**	
1	Jan 15 (Thu)	**Setting up your computer environment**	
2	Jan 20 (Tue)	**Why Buildings**	Assignment #1 Out
2	Jan 22 (Thu)	**Thermal Dynamics of Buildings: Part I**	
3	Jan 27 (Tue)	**Thermal Dynamics of Buildings: Part I**	
3	Jan 29 (Thu)	**Thermal Dynamics of Buildings: Part II**	
4	Feb 3 (Tue)	**Thermal Dynamics of Buildings: Part II**	
4	Feb 5 (Thu)	**Occupant Thermal Comfort**	Assignment #1 Due, Assignment #2 Out
5	Feb 10 (Tue)	**Setting up your Home Assistant Server**	
5	Feb 12 (Thu)	**Basics of AC Power Systems for Buildings**	
6	Feb 17 (Tue)	**Basics of Control Theory for Buildings and their Applications**	
6	Feb 19 (Thu)	**Basics of Control Theory for Buildings and their Applications**	
7	Feb 24 (Tue)	**Project Presentations**	
7	Feb 26 (Thu)	**Paper #1**: FTM-Sense: Robust Sensor-free Occupancy Sensing Leveraging WiFi Fine Time Measurement	Assignment #2 Due, Assignment #3 Out
8	Mar 3 (Tue)	**SPRING BREAK**	No Class

Week	Date	Lecture	Notes
8	Mar 5 (Thu)	**SPRING BREAK**	No Class
9	Mar 10 (Tue)	**Paper #2**: PyDCM: Custom Data Center Models with Reinforcement Learning for Sustainability	
9	Mar 12 (Thu)	**Paper #3**: Gnu-RL: A Precocial Reinforcement Learning Solution for Building HVAC Control Using a Differentiable MPC Policy	
10	Mar 17 (Tue)	**Paper #4**: Modeling the Impact of Passive Ventilation Systems on Multi-Zone Thermal Dynamics	
10	Mar 19 (Thu)	**Paper #5**: Can Attention Improve Sequence-to-Point Load Disaggregation: A Comparative Assessment	
11	Mar 24 (Tue)	**Paper #6**: RECA: A Multi-Task Deep Reinforcement Learning-Based Recommender System for Co-Optimizing Energy, Comfort and Air Quality in Commercial Buildings	Assignment #3 Due, Assignment #4 Out
11	Mar 26 (Thu)	**Paper #7**: Hypertemporal Imaging of NYC Grid Dynamics	
12	Mar 31 (Tue)	**Paper #8**: Data Predictive Control for Peak Power Reduction	
12	Apr 2 (Thu)	**Paper #9**: BOLT: Energy Disaggregation by Online Binary Matrix Factorization of Current Waveforms	
13	Apr 7 (Tue)	**Project Progress Reports**	Assignment #4 Due

Week	Date	Lecture	Notes
13	Apr 9 (Thu)	**SPRING CARNIVAL**	No Class
14	Apr 14 (Tue)	**Project Feedback / Guidance Session**	
14	Apr 16 (Thu)	**Project Feedback / Guidance Session**	
15	Apr 21 (Tue)	**Project Feedback / Guidance Session**	
15	Apr 23 (Thu)	**Project Feedback / Guidance Session**	

Additional Dates:

- **Friday April 24:** In-person demo session
- **Sunday April 26:** Project demo video submissions due
- **Sunday May 3:** Final project reports due

6 Assignments for the Course

Assignment Number	Due Date	Resources
Assignment #1	2/5	[HTML] [PDF] [Jupyter Notebook]
Assignment #2	2/26	[HTML] [PDF] [Jupyter Notebook]
Assignment #3	3/26	[HTML] [PDF] [Jupyter Notebook]
Assignment #4	4/7	

Note: Solutions for all assignments will be posted on Canvas under Files -> Assignments -> Assignment sub-folder.

7 Useful Links

Throughout the course, we will encounter useful links that may not have an appropriate place within the structure of the website/book. This list will be the default placeholder for those links.

7.1 Shared News / Links

- Does turning the air conditioning off when you're not home actually save energy? Three engineers run the numbers
- Opportunities for students to work for and volunteer at Department of Energy affiliated sites
- Innovation in Buildings Graduate Research Fellowship (IBUILD)
- Atom Energy's solid state circuit breaker for EV charging
- Greta Thunberg's new book
- EV batteries alone could satisfy short-term grid storage demand by 2030
- An interesting learning resource for understanding Heating Ventliation and Air Conditioning systems
- Resources listed by the ACM Special Interest Group on Energy Systems and Informatics
- CMU's Energy Week event has a Student Happy Hour & Networking Reception on 3/21. Sign up here.
- An interesting discussion about heat pumps following the announcement of a Y-Combinator funded start-up trying to sell them directly to customers.
- ACM SIGEnergy's Graduate Student Seminar
- The Continental Automated Buildings Association (CABA) Podcast
- YouTube Channel TechnologyConnections covers thermostats

7.2 Interesting papers for second third

Here are some interesting papers to consider for the second third of the course, where we will be learning about the state-of-the-art in the field with respect to smart meters and smart thermostats.

7.2.1 Papers combining electrical meters and thermostat issues:

- **Non-Intrusive Techniques for Establishing Occupancy Related Energy Savings in Commercial Buildings**
- **Enhancing household-level load forecasts using daily load profile clustering**
- Comparing Gray Box Methods to Derive Building Properties from Smart Thermostat Data
- SMITE: Using Smart Meters to Infer the Thermal Efficiency of Residential Homes
- QUILT: QUantify, Infer and Label the Thermal Efficiency of Heating and Cooling Residential Homes
- Contextually Supervised Source Separation with Application to Energy Disaggregation

7.2.2 Papers mostly related to smart thermostats

- **ThermoCoach: Reducing Home Energy Consumption with Personalized Thermostat Recommendations**
- Operational Characteristics of Residential Air Conditioners with Temporally Granular Remote Thermographic Imaging
- Real-Time Cooling Power Attribution for Co-located Data Center Rooms with Distinct Temperatures
- Heat Reuse Models for Liquid Cooled Data Centers integrated with District Heating
- TEA-bot: A Thermography Enabled Autonomous Robot for Detecting Thermal Leaks of HVAC Systems in Ceilings
- The Impact of Resolution of Occupancy Data on Personal Comfort Model-Based HVAC Control Performance
- Good set-points make good neighbors - User seating and temperature control in uberized workspaces
- Hot or Not: Leveraging Mobile Devices for Ubiquitous Temperature Sensing
- Exploring Fairness in Participatory Thermal Comfort Control in Smart Buildings
- The SPOT* Personal Thermal Comfort System
- A toolkit for low-cost thermal comfort sensing

7.2.3 Papers mostly related to smart meters

- **Analyzing Energy Usage on a City-scale using Utility Smart Meters**
- BOLT: Energy Disaggregation by Online Binary Matrix Factorization of Current Waveforms
- Rimor: Towards Identifying Anomalous Appliances in Buildings

Part II

Preliminaries

8 Why buildings?

You have configured your computing environment, reviewed fundamental concepts in data acquisition and set up your Raspberry Pi devices. You are ready! But ready for what? In the previous chapter we discussed the three words on the title of this course (*Autonomous Sustainable Buildings*) and focused on defining the first two. But why are we targeting buildings in particular? Why not vehicles or financial trading agents? Well, of course, part of the answer is that you and I are both interested in buildings and in some sense that is sufficient. But are there good reasons beyond personal taste to focus our attention on buildings when developing sustainable autonomous technologies? Thankfully, the answer is yes. This last chapter of the preliminaries will provide the necessary context for all of us to answer that question more formally (or at least with hard evidence).

We will be relying on a few sources to understand the energy use of our building stock, namely:

- A paper by Pérez-Lombard et al. (2008) which discusses the building energy landscape as of 2008 using publicly available data at the moment.
- Chapter 1 from a new book by Murphy Jr (2021), which does some fun things to consider the growth of humanity's energy demand.
- Reports by the Energy Information Administration, especially the Commercial Building Energy Consumption Survey (CBECS) and the Residential Energy Consumption Survey (RECS).
- Lawrence Livermore National Laboratory's energy flow charts for energy use in the United States.
- Chapter 4 from Harvey (2010), which discusses more detailed statistics about energy use of bulidings worldwide.

9 Setting up your learning / tinkering computer environment

9.1 Lecture Overview

Learning Objectives

By the end of this module, students will be able to:

- Understand how their computers need to be configured for maximum efficiency in working on the Final Project and assignments
- Have a working Python installation on their computer, with Jupyter Notebook and uv as the package and project manager
- Understand how to use uv and Jupyter Notebooks to:
  - Create, load, edit and run Jupyter notebook files
  - Create and manage Python projects and virtual environments
  - Create Jupyter kernels specific to a uv Python project
- Create and access a public git repository on Github
- Use git to clone, commit and push changes to a remote repository
- Ensure that their computer has a working SSH client and understand the value of this tool

Topics Covered

- Development environment overview for ML-based building energy management
- Python installation and configuration
- uv package and project manager
- Jupyter Notebooks for interactive development
- Git and GitHub for version control
- SSH for secure remote access

9.1.1 Before You Begin

To complete this lecture's activities, you will need:

1. **A GitHub account**: Create one at github.com if you don't already have one
2. **Administrator privileges**: Ensure you have the ability to install software on your computer
3. **A text editor or IDE**: VS Code, Sublime Text, or any editor you're comfortable with (optional but recommended)

9.1.2 Project Milestones

This lecture establishes the foundational development environment for the entire course. By completing this module, you will have achieved the first major project milestone: a fully configured development environment that includes:

- Python and uv for package management
- Jupyter Notebooks for interactive development and assignment submissions
- Git/GitHub for version control and collaboration
- SSH for connecting to the Home Assistant server in the final project

9.2 Setting Up Python and uv

9.2.1 Why Python and uv?

Why Python?

Python has become the de facto language for machine learning and data science applications. For this course, we've chosen Python because:

- It's the primary language used in the ML/AI ecosystem, with libraries like scikit-learn, TensorFlow, and PyTorch
- It allows for rapid prototyping and experimentation - crucial for the iterative nature of building energy management solutions
- It has excellent support for scientific computing (NumPy, SciPy, pandas) and visualization (matplotlib, plotly)
- It integrates well with building automation platforms like Home Assistant

Why uv?

For package management, we're using uv instead of traditional tools like pip or conda. uv is a modern Python package and project manager that offers several advantages:

- **Speed**: Much faster than pip or conda for installing packages
- **Modern dependency resolution**: Better handling of complex dependency trees
- **Project isolation**: Creates lightweight virtual environments without the overhead of conda

- **All-in-one tool**: Manages both Python installation and packages in a single tool
- **Simplicity**: Straightforward commands that are easy to learn and use

Interface Comparison:

Task	pip	conda	uv
Install package	`pip install numpy`	`conda install numpy`	`uv pip install numpy`
Create project	Manual venv setup	`conda create -n myenv`	`uv init myproject`
Manage dependencies	`requirements.txt`	`environment.yml`	`pyproject.toml`
Install Python	Separate download	Bundled with Anaconda	`uv python install 3.11`

Functional Differences:

- **pip**: Package installer only; requires separate Python installation and venv management
- **conda**: Full environment manager with its own package repository; heavier and slower but handles non-Python dependencies
- **uv**: Fast, all-in-one tool that manages Python versions, packages, and projects; uses standard PyPI repository; lightweight and modern

For this course's focus on ML applications for building energy management, uv provides the right balance of simplicity, speed, and functionality.

9.2.2 Installing uv

> ⚠ Security Note
>
> The installation methods below download and execute scripts directly from the internet. While this is the official installation method recommended by the uv developers, you should be aware that running scripts this way carries inherent security risks. Only proceed if you trust the source (Astral, the developers of uv). For production environments, consider reviewing the installation script first or using alternative installation methods.

uv can be installed using the official installer script. Follow the instructions for your operating system:

macOS

Open Terminal and run:

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

After installation, restart your terminal or run:

```
source $HOME/.cargo/env
```

Windows

Open PowerShell and run:

```
powershell -c "irm https://astral.sh/uv/install.ps1 | iex"
```

After installation, restart PowerShell to ensure the `uv` command is available.

Verify Installation

Regardless of your platform, verify that uv is installed correctly by running:

```
uv --version
```

You should see output showing the installed version of uv (e.g., `uv 0.x.x`).

9.2.3 Installing Python

One of uv's key features is that it can install and manage Python versions for you. For this course, we'll use **Python 3.11**.

Check Available Python Versions

First, you can see which Python versions are available:

```
uv python list
```

This will show you all available Python versions that uv can install.

Install Python 3.11

To install Python 3.11, run:

```
uv python install 3.11
```

uv will download and install Python 3.11 for you. This works identically on both macOS and Windows.

Verify Python Installation

Check that Python was installed successfully:

```
uv python list
```

You should see Python 3.11 listed with an indicator showing it's installed.

Managing Multiple Python Versions

uv can manage multiple Python versions simultaneously. You don't need to worry about conflicts - when you create a project (which we'll do next), you'll specify which Python version that project should use. This allows you to work on different projects with different Python versions without any issues.

9.2.4 Learn-by-Doing Activity 1: Python and uv Setup

Objective: Create a simple test project to verify your Python and uv installation is working correctly.

Steps:

1. Create a new directory for your test project:

   ```
   mkdir uv-test
   cd uv-test
   ```

2. Initialize a new uv project with Python 3.11:

   ```
   uv init --python 3.11
   ```

3. This creates a basic project structure. Now add a simple Python package:

   ```
   uv add numpy
   ```

4. Create a simple test script. Create a file called `test.py` with the following content:

   ```
   import numpy as np
   print(f"NumPy version: {np.__version__}")
   print("Installation successful!")
   ```

5. Run your script using uv:

   ```
   uv run test.py
   ```

Expected Result: You should see output showing the NumPy version and the success message. If you see this, congratulations! Your Python and uv setup is working correctly.

9.3 Working with Jupyter Notebooks

9.3.1 What are Jupyter Notebooks?

Jupyter Notebooks are interactive documents that combine live code, visualizations, and explanatory text in a single interface. They run in your web browser and connect to a Python kernel that executes your code. This architecture allows you to write code in "cells," run them individually, see results immediately, and document your work with markdown cells - all in one place.

For this course, Jupyter Notebooks are ideal because they allow you to prototype machine learning solutions iteratively, visualize building energy data alongside your analysis, and create self-contained documents that combine your code, results, and explanations. All course assignments will be submitted as Jupyter Notebook files (`.ipynb`), making it easy for you to show both your work and your reasoning.

9.3.2 Creating and Managing Projects with uv

Creating a New Project

To create a new Python project with uv:

```
uv init myproject --python 3.11
cd myproject
```

This creates a project directory with the following structure: - `pyproject.toml`: Project configuration and dependencies - `.python-version`: Specifies which Python version to use - `README.md`: Project documentation

Managing Dependencies

Add packages to your project:

```
uv add package-name
```

Remove packages:

```
uv remove package-name
```

View your dependency tree:

```
uv tree
```

Syncing Dependencies

When working with an existing project (for example, one you downloaded from a colleague or from GitHub), you'll need to install all the project's dependencies. The `uv sync` command reads the `pyproject.toml` file and installs everything the project needs:

```
cd existing-project
uv sync
```

This is particularly useful when:

- You've just cloned the course repository and want to work with the example notebooks
- You're starting work on a project on a new computer
- Another team member has added new dependencies to the project

The sync command ensures your local environment matches exactly what the project specifies, so the code will run the same way for everyone.

Running Python Code

Execute Python scripts within your project's environment:

```
uv run script.py
```

This automatically uses the correct Python version and has access to all installed dependencies.

9.3.3 Setting Up Jupyter Kernels

What is a Jupyter Kernel?

A Jupyter kernel is the computational engine that executes the code in your notebooks. When you run a code cell, Jupyter sends that code to the kernel, which executes it in a specific Python environment and returns the results. Different kernels can use different Python versions and have access to different installed packages.

Two Approaches to Using Jupyter with uv

There are two ways to set up Jupyter to work with your uv projects:

Approach 1: Simple (Using `uv run`)

The quickest way to start Jupyter with access to your project's environment:

```
uv run --with jupyter jupyter lab
```

> **About the `--with` flag**
>
> The `--with` flag tells uv to temporarily install a package (in this case, `jupyter`) for this command only, without adding it to your project's dependencies. This is useful for tools you need to run but don't want as permanent project dependencies.

This starts Jupyter Lab with access to all your project's packages. Use this approach when you want to quickly work on a single project.

Approach 2: Kernel Registration (Recommended for Course Work)

For more robust setup, especially when working with multiple projects, register your project as a Jupyter kernel:

1. First, add `ipykernel` to your project as a development dependency:

   ```
   uv add --dev ipykernel
   ```

   > **About the `--dev` flag**
   >
   > The `--dev` flag marks a package as a development dependency - something needed for working on the project (like testing tools, linters, or Jupyter kernels) but not required to run the final code. Development dependencies are tracked separately in `pyproject.toml` and won't be installed by users who just want to use your code.

2. Register your project as a kernel:

   ```
   uv run ipython kernel install --user --name=myproject
   ```

3. Start Jupyter Lab:

   ```
   uv run --with jupyter jupyter lab
   ```

4. When creating a new notebook, select your project kernel from the kernel dropdown menu in the top-right corner.

Benefits of Kernel Registration

With a registered kernel, you can:

- Work with multiple projects simultaneously in Jupyter

- Easily switch between different project environments
- Add packages directly from within notebooks using `!uv add package-name`
- Ensure consistent environments across different machines

For this course, we recommend using **Approach 2** (kernel registration) so you can easily work with both the course materials and your own projects.

9.3.4 Running Jupyter Notebooks

Launching Jupyter Lab

Once you have your project set up with a registered kernel, start Jupyter Lab:

```
uv run --with jupyter jupyter lab
```

This opens Jupyter Lab in your default web browser. Jupyter Lab is a modern, feature-rich interface for working with notebooks.

Jupyter Lab vs Jupyter Notebook

There are two interfaces available:

- **Jupyter Lab** (recommended): Modern interface with multiple panels, file browser, and advanced features. Launch with `jupyter lab`
- **Jupyter Notebook**: Classic, simpler interface focused on single notebooks. Launch with `jupyter notebook`

Both work identically for editing notebooks - the difference is in the surrounding interface. Feel free to use whichever you prefer, but we recommend Jupyter Lab for its improved usability.

Getting the Sample Notebook

To practice using Jupyter, you'll work through the sample notebook for this lecture. You can download it directly from the course website or find it in the course repository at:

`lectures/Lecture-3/lecture-3-sample-notebook.ipynb`

If you don't have the course repository yet (we'll cover git cloning shortly), you can download just this file from the course website and place it in your project directory.

Working with Notebooks

Once you open the sample notebook in Jupyter Lab:

1. Select your project kernel from the kernel dropdown in the top-right corner
2. Follow the instructions in the notebook - it will guide you through creating code cells, running them, and working with Python packages
3. The sample notebook covers the basics of the Jupyter interface, so you can learn by doing

9.3.5 Learn-by-Doing Activity 2: Your First Jupyter Notebook

In-Class Exercise: Work through the `lecture-3-sample-notebook.ipynb` file available in the course repository. This hands-on activity will guide you through:

- Creating and running Python code cells
- Using Jupyter magic commands (like `%matplotlib inline`)
- Creating simple visualizations with matplotlib
- Running shell commands from within Jupyter using `!`
- Saving your notebook

Clone the course repository and open this notebook to follow along during the lecture.

9.4 Git and GitHub

9.4.1 Why Version Control?

Version control systems track changes to files over time, allowing you to recall specific versions later. Git, the most widely used version control system, is essential for modern software development and collaborative research.

Problems Version Control Solves:

- **"Final_FINAL_v3_actually_final.docx"**: No more confusing file names - version control tracks all changes with meaningful descriptions
- **Lost work**: Accidentally deleted something important? Version control lets you restore any previous version
- **Collaboration conflicts**: Multiple people editing the same file? Git helps merge changes intelligently
- **Understanding history**: See who changed what, when, and why - crucial for understanding code decisions
- **Experimentation safety**: Try new approaches without fear - you can always revert to a working version

For This Course:

You'll use Git and GitHub in two ways:

1. **Accessing course materials**: The course repository (https://git.inferlab.org/websites/12-770) contains all lecture notebooks, examples, and resources. You'll clone this repository to get updates throughout the semester.

2. **Final project submission**: You'll create your own repository for your final project, making it easy to track your progress, collaborate with team members, and share your work with instructors.

Version control isn't just a technical tool - it's a practice that makes collaborative work possible and individual work more reliable.

9.4.2 Installing Git

Git is the version control system we'll use throughout the course. Installation varies by operating system.

macOS

Git often comes pre-installed on macOS as part of the Xcode Command Line Tools. First, check if it's already installed:

```
git --version
```

If you see a version number, you're all set! If not, macOS will prompt you to install the Command Line Tools. Alternatively, you can install git via Homebrew:

```
brew install git
```

Windows

Download and install Git for Windows from git-scm.com/download/win. The installer will guide you through the setup with recommended defaults.

> **Windows Subsystem for Linux (WSL2)**
>
> While native Git for Windows works well for this course, you may want to consider installing WSL2 (Windows Subsystem for Linux) for your final project work. WSL2 provides a full Linux environment on Windows, which can be helpful for working with tools like Docker and SSH. It's not required now, but worth exploring if you're interested in a more Unix-like development environment. Learn more at docs.microsoft.com/windows/wsl.

Verify Installation

Regardless of your platform, verify that git is installed correctly:

```
git --version
```

You should see output showing the installed version of git (e.g., `git version 2.x.x`).

9.4.3 Git Basics

Git tracks changes to your files through a three-stage workflow. Understanding these core concepts will help you use git effectively.

Core Concepts:

- **Repository (repo)**: A directory where git tracks all changes to your files. Contains the complete history of your project.
- **Commit**: A snapshot of your files at a specific point in time, with a message describing what changed and why.
- **Branch**: A parallel version of your repository, allowing you to work on features independently without affecting the main codebase.

The Three-Stage Workflow:

Git uses three stages to manage changes:

1. **Working Directory**: Where you edit your files normally
2. **Staging Area**: Where you prepare changes for a commit (like a "shopping cart" for changes)
3. **Repository**: Where committed changes are permanently stored

This workflow gives you fine control over what gets saved in each commit.

Basic Git Commands:

Check the status of your repository:

```
git status
```

Add files to the staging area:

```
git add filename.py          # Add a specific file
git add .                    # Add all changed files
```

Commit staged changes:

```
git commit -m "Descriptive message about what changed"
```

View commit history:

```
git log
```

About Branches:

Branches let you experiment or work on features without affecting the main code. For this course, you'll primarily work on the `main` branch for your project. However, branches become valuable when:

- Testing a new approach without breaking working code
- Collaborating with teammates on different features simultaneously
- Keeping experimental work separate from stable code

We won't focus heavily on branching in this course, but it's a powerful feature worth exploring in the Additional Resources.

Visual Learning:

If you're finding these concepts abstract, check out the Explain Git with D3 resource in the Additional Resources section - it provides an interactive visualization of how git works internally.

9.4.4 Common Git Operations

Once you have git installed, you'll use a few core operations repeatedly. These commands connect your local work with remote repositories (like those on GitHub).

Understanding Remotes:

A **remote** is a version of your repository hosted elsewhere (typically on GitHub or another git server). The default remote is usually named `origin`. When you clone a repository, git automatically sets up the remote connection for you.

Clone: Get a Copy of a Repository

Download a complete copy of a repository to your computer:

```
git clone <repository-url>
```

This creates a new directory with all the project files and their complete history.

Commit: Save Your Changes Locally

After making changes to files, save a snapshot:

```
git add .
git commit -m "Description of what changed"
```

This saves changes to your local repository only - they're not yet on the remote server.

Push: Send Your Commits to the Remote

Upload your local commits to the remote repository:

```
git push origin main
```

This makes your changes visible to others and backs them up on the remote server.

Pull: Get Updates from the Remote

Download and merge changes from the remote repository:

```
git pull origin main
```

This updates your local repository with changes others have made. Always pull before starting new work to avoid conflicts.

9.4.4.1 Cloning the Course Repository

One of the first git operations you'll perform is cloning the course repository to access lecture materials and sample notebooks:

```
git clone https://git.inferlab.org/websites/12-770.git
```

This creates a local copy of all course materials on your computer. Navigate into the cloned directory to access the lecture notebooks and other resources.

9.4.5 Creating a GitHub Repository

Before creating repositories, make sure you have a GitHub account. If you haven't already created one as mentioned in the "Before You Begin" section, sign up at github.com.

Creating a New Repository:

1. Log in to GitHub and click the "+" icon in the top-right corner
2. Select "New repository"
3. Choose a repository name (e.g., `12-770-final-project`)
4. Add an optional description
5. Choose repository visibility:

- **Public** (recommended for this course): Anyone can see your code
- **Private**: Only you and collaborators you invite can see the code

For this course, we recommend using **public repositories** for your final project. This makes collaboration easier and allows you to showcase your work. However, be mindful: **never commit secrets, passwords, API keys, or sensitive Home Assistant configuration** to public repositories.

6. Optionally initialize with a README (recommended - it helps document your project)
7. Click "Create repository"

Getting the Repository URL:

After creating the repository, GitHub will show you the repository URL. You'll see options for HTTPS and SSH:

- **HTTPS**: `https://github.com/username/repository.git`
- **SSH**: `git@github.com:username/repository.git`

About Authentication:

When you clone or push to a GitHub repository, you'll need to authenticate. GitHub offers two methods:

- **HTTPS with Personal Access Token**: Works immediately but requires entering credentials
- **SSH Keys**: More convenient after initial setup - no passwords needed

We'll cover SSH key setup in the SSH section later in this lecture. For now, you can use HTTPS to get started, but we recommend setting up SSH keys for easier long-term use.

9.4.6 Learn-by-Doing Activity 3: Your First Git Repository

In-Class Exercise: Practice the complete git workflow:

1. Create a new repository on GitHub
2. Clone it to your local machine using `git clone`
3. Create a new Jupyter notebook in the repository folder
4. Add some Python code or markdown cells to the notebook
5. Stage your changes: `git add yourfile.ipynb`
6. Commit your changes: `git commit -m "Made my first commit"`
7. Push to GitHub: `git push origin main`
8. Verify the file appears in your GitHub repository online

This workflow of clone → modify → commit → push will be used throughout the course for assignment submissions.

9.5 SSH for Remote Access

9.5.1 What is SSH and Why Do We Need It?

SSH (Secure Shell) is a protocol for securely connecting to remote computers over a network. It encrypts all communication between your computer and the remote server, keeping your data safe from eavesdropping.

Why SSH Matters for This Course:

You'll use SSH in two important ways:

1. **GitHub Authentication**: SSH keys provide a convenient, password-free way to authenticate with GitHub. Once set up, you can push and pull code without entering credentials each time.
2. **Home Assistant Server Access**: For your final project, you'll need to connect to the Home Assistant server to deploy and test your solutions. SSH provides secure remote access to the server's command line.

SSH uses key-based authentication instead of passwords, which is both more secure and more convenient for regular use.

9.5.2 Installing and Configuring SSH

Most modern operating systems include an SSH client by default. Let's verify and install if needed.

macOS and Linux

SSH comes pre-installed on macOS and most Linux distributions. Verify it's available:

```
ssh -V
```

You should see output showing the OpenSSH version. If you see this, you're all set!

Windows

Windows 10 and later include OpenSSH by default. Verify it's available by opening PowerShell and running:

```
ssh -V
```

If you see the version information, you're ready to go.

Alternative for Windows: PuTTY

If you prefer a graphical interface or are using an older version of Windows, you can install PuTTY, a popular SSH client for Windows. PuTTY includes both command-line tools and a GUI for managing SSH connections.

However, for this course, we recommend using the built-in OpenSSH client (available in PowerShell) as it provides a consistent experience with macOS/Linux and works seamlessly with the commands we'll use.

9.5.3 SSH Key-Based Authentication

SSH keys work in pairs: a **private key** (kept secret on your computer) and a **public key** (shared with services like GitHub or servers you want to access). When you connect, the server uses your public key to verify you own the matching private key, without the private key ever leaving your computer.

Generating SSH Keys

Create a new SSH key pair:

```
ssh-keygen -t ed25519 -C "your_email@example.com"
```

> **About the Key Algorithm**
>
> The `-t ed25519` flag specifies the Ed25519 algorithm, which is modern, secure, and recommended for new keys. If you omit this flag, ssh-keygen will use RSA by default, which also works but generates larger keys.

When prompted:

1. **File location**: Press Enter to accept the default location (`~/.ssh/id_ed25519`)
2. **Passphrase**: You can optionally enter a passphrase for extra security. This means you'll need to enter the passphrase each time you use the key. For convenience, many users leave this blank, but a passphrase adds an additional layer of protection if your private key is ever compromised.

Viewing Your Public Key

To see your public key (which you'll add to GitHub and other services):

```
cat ~/.ssh/id_ed25519.pub
```

This displays your public key, which starts with `ssh-ed25519` and ends with your email. You can safely share this - it's designed to be public.

Important: Never share your private key (`~/.ssh/id_ed25519` without the `.pub`). Keep it secure on your computer.

9.5.4 Testing Your SSH Connection

Once you've generated SSH keys and added them to services like GitHub, you can test your SSH connection.

Testing GitHub SSH Connection

GitHub provides a convenient test endpoint:

```
ssh -T git@github.com
```

Expected Output:

If successful, you'll see a message like:

```
Hi username! You've successfully authenticated, but GitHub does not provide shell access.
```

This confirms your SSH key is properly configured with GitHub. The "does not provide shell access" part is normal - GitHub only allows git operations via SSH, not shell access.

For Home Assistant Server

Later in the course, when you work with the Home Assistant Green hardware for your final project, you'll use SSH to connect directly to the server:

```
ssh username@server-address
```

This will give you command-line access to deploy and test your solutions on the actual hardware.

Common Issues

If the connection fails, check:

- **Key not added**: Make sure you've added your public key to GitHub (Settings → SSH and GPG keys)
- **Wrong key location**: Ensure your key is in `~/.ssh/id_ed25519` (or update your SSH config if using a different location)

- **Permissions**: SSH keys require specific permissions. If you have issues, run:

```
chmod 600 ~/.ssh/id_ed25519
chmod 644 ~/.ssh/id_ed25519.pub
```

9.5.5 Learn-by-Doing Activity 4: SSH Setup and Testing

Objective: Set up SSH keys for use with GitHub and future Home Assistant server access.

Steps:

- ☐ Verify SSH client is installed (`ssh -V`)
- ☐ Generate SSH key pair using `ssh-keygen -t ed25519 -C "your_email@example.com"`
- ☐ View your public key with `cat ~/.ssh/id_ed25519.pub`
- ☐ Copy your public key to clipboard
- ☐ Add your SSH key to GitHub:
  - Go to GitHub → Settings → SSH and GPG keys → New SSH key
  - Paste your public key and give it a descriptive title (e.g., "My Laptop")
  - For detailed instructions, see GitHub's official SSH key guide

Note: We're setting this up now as a reference for later use. You'll actively use SSH keys when:

- Pushing/pulling from GitHub repositories (more convenient than HTTPS)
- Connecting to the Home Assistant server for your final project

You don't need to verify the connection works right now - we'll do that when you start working with GitHub repositories and the Home Assistant hardware later in the course. This activity ensures you know where to find these instructions when you need them.

9.6 Putting It All Together

9.6.1 Your Complete Development Workflow

Now that you've learned all the individual tools, let's put them together in a complete workflow. This exercise will test your understanding of the entire development environment.

The Challenge:

Complete this workflow on your own to verify your development environment is fully functional:

1. **Clone the course repository** (if you haven't already):

```
git clone https://git.inferlab.org/websites/12-770.git
cd 12-770
```

2. **Find and copy the sample notebook**:

   - Navigate to `lectures/Lecture-3/lecture-3-sample-notebook.ipynb`
   - Copy this file to a new location on your computer (outside the course repo)

3. **Create a new GitHub repository**:

   - Go to GitHub and create a new **public** repository named `12-770-test`
   - Initialize with a README
   - Clone your new repository to your computer

4. **Set up your project with uv**:

   - In your cloned `12-770-test` directory, initialize a uv project
   - Add jupyter and ipykernel as dependencies
   - Register a Jupyter kernel for this project

5. **Work with the notebook**:

   - Copy the sample notebook into your `12-770-test` directory
   - Launch Jupyter Lab
   - Open the notebook and make some modifications (add a cell, change some code, etc.)
   - Save your changes

6. **Commit and push to GitHub**:

```
git add lecture-3-sample-notebook.ipynb
git commit -m "Add modified sample notebook"
git push origin main
```

7. **Verify**: Check your GitHub repository in a web browser - you should see your modified notebook!

What This Tests:

- Git cloning and repository management
- uv project setup and dependency management
- Jupyter Lab with custom kernels
- Git workflow: add, commit, push
- GitHub repository creation and verification

If you can complete this workflow successfully, your development environment is ready for the course!

9.6.2 Best Practices

As you work with these tools throughout the course, keep these key practices in mind:

Git and Version Control:

- Commit frequently with descriptive messages that explain *why* you made changes, not just *what* changed
- Pull from the course repository regularly to get the latest materials and updates
- Never commit sensitive information (passwords, API keys, personal data) to public repositories

Python and uv:

- Create a separate project for each assignment or experiment - don't try to do everything in one environment
- Use `uv sync` when you switch between projects to ensure dependencies are up to date
- Keep your `pyproject.toml` file clean - remove packages you're no longer using

Jupyter Notebooks:

- Restart your kernel and run all cells before submitting work to ensure it runs from a clean state
- Use markdown cells to document your thinking and explain your approach
- Save often - Jupyter's autosave helps, but manual saves are more reliable

General Workflow:

- Keep your development tools updated (run occasional `uv self update` and `git --version` to check)
- When stuck, check the Additional Resources section - the official documentation is often the best help
- Use the course repository as a reference for working examples of these tools in action

9.7 Additional Resources

9.7.1 Python Resources

- Python Cheat Sheet
- Python Official Documentation

- Scientific Python Lectures - Introduction to Python for Scientific Computing
- Introduction to Python Programming Tutorial
- NumPy Tutorial
- Matplotlib Tutorial
- SciPy Lecture Notes - Heavy focus on Python fundamentals
- Quantitative Economics with Python

9.7.2 Online Courses

- Introduction to Scientific Python (Stanford)
- Practical Data Science (CMU)
- Computational Statistics in Python (Duke)

9.7.3 Git Resources

- Git Official Documentation - Especially the first three videos
- Official Git Tutorial - For those familiar with version control
- Explain Git with D3 - Visual, interactive explanation of git internals
- Git from the Inside Out - In-depth discussion of git internals
- Animated Git Tutorial
- Simple Git Tutorial
- Pro Git Book

9.7.4 Jupyter Resources

- Jupyter Documentation

9.7.5 uv Resources

- uv Documentation

9.7.6 SSH Resources

- SSH Academy
- GitHub SSH Guide

Part III

Fundamental Concepts

10 Thermal Dynamics of Buildings: Part I

10.1 Lecture Overview

Learning Objectives

By the end of this module, students will be able to:

- Name the main processes by which heat loss takes place in buildings
- Recognize the value of thinking about heat losses as stemming from the product of building's leakiness and the temperature demand
- Be able to mathematically model the different modes of heat transfer (conduction, convection and radiation), understanding the underlying physical laws underpinning them
- Have familiarity with different building materials and their thermal conductivity as well as the thermal conductance of a material assembly
- Represent wall and window assemblies as networks of thermal resistances and capacitances

Topics Covered

- Building heat loss fundamentals
- Heating Degree-Days (HDD) and Cooling Degree-Days (CDD)
- Fundamental thermal concepts (temperature, heat capacity)
- Conduction heat transfer and Fourier's Law
- Convection heat transfer
- Radiation heat transfer
- Thermal properties of building materials
- Thermal resistance (R-value) and conductance (U-value)
- Thermal resistance networks for building assemblies

Project Milestones

This module provides the theoretical foundation for understanding how buildings exchange heat with their environment. This knowledge is essential for:

- Understanding sensor data from temperature measurements in your final project

- Reasoning about thermal time constants and system dynamics
- Designing control strategies that account for building physics

10.2 Building Thermodynamics

We are now ready to address some of the fundamental technical concepts in building science, which are needed for us to reason about (*plan*) the data that our autonomous technologies will be gathering (i.e., what they *sense*). As we saw in the previous chapter, energy use in buildings is dominated by heating and cooling services. Thus, we will begin by reviewing basic concepts of thermodynamics leading up to the establishment of simple dynamic models of the temperature inside buildings and the energy required to maintain it.

10.2.1 Heat Loss in Buildings: A High-Level View

In a very general sense, buildings lose (and gain) heat through interactions with the outdoor environment, and through internal processes that compensate for that. During a whole season (say, a full year), the energy lost through the building envelope is (again, at a high level) proportional to the leakiness of the building envelope, and to the temperature demand (i.e., the average difference between the indoor and outdoor temperatures). So, if we were to write an equation for the total (say, heating) losses Q that need to be compensated to keep the building at a given internal temperature (T_i), we can write the following high-level formula:

$$Q = \text{leakiness} \times \text{temperature-demand}$$

This leakiness is a function of the properties of the building's envelope. A bit more specifically, it is mostly affected by the way the envelope handles heat conduction and air infiltration and is usually expressed in units of energy per time per degree of temperature difference. The temperature demand is a function of the accumulated difference between the exterior temperature T_o and the average temperature we wish to keep the indoor space T_i over the season in question. This is usually described and tabulated as Heating Degree-Days (HDD) or Cooling Degree-Days (CDD), where Degree-Days refers to the accumulation (e.g., integral) of the difference in temperature ($T_o - T_i$) over the season (expressed as number of days).

The *leakiness* in the above equation represents the rate at which heat is lost through the building enclosure in proportion to the temperature difference (i.e., it's in units of power per degree of temperature difference). This happens through a combination of thermal conduction, radiation and convection as well as through infiltration (air exchange).

> **Tip**
>
> Read Appendix E of MacKay (2008) for more details on this topic. See if you can re-calculate the numbers in Figure E.12 that refer to "my house, before" and "my house, after".

10.2.2 Heating and Cooling Degree-Days

Degree-days provide a practical way to quantify the "temperature demand" in our high-level heat loss equation. A **degree-day** compares the mean outdoor temperature recorded for a location to a standard *base temperature*, typically 65°F (18.3°C) in the United States. This base temperature represents the outdoor temperature at which, on average, buildings require neither heating nor cooling to maintain comfortable indoor conditions.

10.2.2.1 Calculating Degree-Days

The calculation is straightforward. In theory, we would just compute the integral of the difference between outdoor temperature and a base temperature reference. But since we're using discrete measurements we perform a sum. For any given day:

1. Since temperature during the day changes, we need to start with an average value for the day. You may, for example, compute the mean daily temperature from the extreme values recorded: $T_{mean} = \frac{T_{high}+T_{low}}{2}$
2. then compare to the base temperature (65°F):

    - If $T_{mean} < 65°F$: **Heating Degree-Days (HDD)** $= 65 - T_{mean}$
    - If $T_{mean} > 65°F$: **Cooling Degree-Days (CDD)** $= T_{mean} - 65$
    - If $T_{mean} = 65°F$: Both HDD and CDD are zero

For example, a day with a high of 90°F and a low of 66°F has a mean of 78°F, yielding $78 - 65 = 13$ cooling degree-days. Conversely, a day with a high of 33°F and a low of 25°F has a mean of 29°F, yielding $65 - 29 = 36$ heating degree-days.

Finally, annual degree-days are simply the sum of daily values over a year, providing a single number that characterizes how much heating or cooling a location requires.

10.2.2.2 Typical Values Across Climates

The table below shows approximate annual HDD and CDD values for selected U.S. cities, illustrating the dramatic variation across climate zones:

City	Climate	Annual HDD	Annual CDD
Miami, FL	Hot-humid	~200	~4,400
Phoenix, AZ	Hot-dry	~1,200	~4,600
Atlanta, GA	Mixed-humid	~2,800	~1,800
Pittsburgh, PA	Cold	~5,800	~800
Chicago, IL	Cold	~6,200	~900
Minneapolis, MN	Very cold	~7,600	~700

Values are approximate based on 1981-2010 climate normals. For current data, see the NOAA Climate Prediction Center or DegreeDays.net for worldwide locations.

Notice how heating-dominated climates (Pittsburgh, Chicago, Minneapolis) have HDD values 5-10 times larger than their CDD values, while cooling-dominated climates (Miami, Phoenix) show the opposite pattern. This has profound implications for building design: a well-insulated envelope is critical in Minneapolis, while shading and cooling efficiency matter more in Phoenix.

10.2.2.3 Using Degree-Days for Energy Estimation

Returning to our high-level equation, we can now be more precise. If we express the leakiness as an overall heat loss coefficient K (in units of power per degree of temperature difference, such as W/°C), then the seasonal heating energy can be estimated as:

$$Q_{heating} = K \times \text{HDD} \times 24 \text{ hours/day}$$

The factor of 24 converts degree-*days* to degree-*hours*, giving us energy (since K has units of power per degree). This simple model, sometimes called the *degree-day method*, allows quick estimates of heating fuel requirements. For instance, if a building in Pittsburgh ($\approx 5,800$ HDD) has the same envelope as one in Miami (≈ 200 HDD), the Pittsburgh building would require roughly $5800/200 = 29$ times more heating energy annually.

> **Note**
>
> The degree-day method is a useful first approximation, but it has limitations. It assumes a fixed base temperature, ignores internal heat gains from occupants and equipment, and treats the building as if it responds instantaneously to outdoor temperature changes. We will revisit these assumptions later when we discuss dynamic thermal behavior.

10.3 Fundamental Thermal Concepts

Before diving into heat transfer mechanisms, we need to establish some fundamental definitions.

10.3.1 Temperature

Temperature is a measure of the average kinetic energy of the molecules in a substance. When we heat a material, its molecules move faster (or vibrate more intensely in solids), and we perceive this as a higher temperature. Importantly, temperature is an *intensive* property—it does not depend on the amount of material present. A cup of boiling water and a pot of boiling water are at the same temperature, even though the pot contains more thermal energy.

10.3.1.1 Temperature Scales

Four temperature scales are commonly encountered in building science:

Scale	Symbol	Water Freezes	Water Boils	Absolute Zero
Celsius	°C	0	100	-273.15
Fahrenheit	°F	32	212	-459.67
Kelvin	K	273.15	373.15	0
Rankine	°R	491.67	671.67	0

The conversions between these scales are:

$$T_F = \frac{9}{5}T_C + 32 \qquad T_K = T_C + 273.15 \qquad T_R = T_F + 459.67$$

In building science, we typically use Celsius (or Fahrenheit in the U.S.) for everyday temperatures, but Kelvin becomes essential when dealing with radiation heat transfer, where absolute temperature matters.

10.3.1.2 Temperature vs. Heat

A critical distinction that often causes confusion:

- **Temperature** is a *state property*—it describes the current thermal condition of a substance (how "hot" it is).
- **Heat** is *energy in transit*—it refers to the transfer of thermal energy between systems due to a temperature difference.

An analogy: temperature is like the water level in a tank, while heat is like the flow of water between tanks. Water flows from higher to lower levels; heat flows from higher to lower temperatures. Two objects at the same temperature exchange no heat, regardless of how much thermal energy each contains.

This distinction matters practically: when we say a building "loses heat," we mean thermal energy is flowing out of the building because the interior is warmer than the exterior. The rate of this heat loss depends on the temperature *difference*, as we will see repeatedly in the sections that follow.

10.3.2 Heat Capacity

Heat capacity describes a material's ability to store thermal energy. The heat capacity C of an object is defined as:

$$C = cm$$

where c is the specific heat capacity of the material and m is its mass. The specific heat capacity can be understood through the relationship:

$$c = \frac{Q}{m\Delta T}$$

where Q is the heat added and ΔT is the resulting temperature change.

10.3.2.1 Specific Heat of Building Materials

The table below lists specific heat capacities for materials commonly encountered in buildings:

Material	Specific Heat c (J/(kg·K))	Density ρ (kg/m³)
Water	4,186	1,000
Wood (typical)	1,200	400-700
Gypsum board	1,000	800
Air (at 20°C)	1,005	1.2
Concrete	880	2,300
Brick	840	1,700
Glass	840	2,500
Stone/Marble	880	2,600
Steel	420	7,850

Material	Specific Heat c (J/(kg·K))	Density ρ (kg/m³)
Aluminum	900	2,700

Values are approximate and vary with composition and moisture content. Sources: Engineering ToolBox, GreenSpec.

Notice that water has an exceptionally high specific heat—roughly 4-5 times that of most solid building materials. This is why water-based heating and cooling systems are so effective at transporting thermal energy.

10.3.2.2 Thermal Mass in Buildings

While specific heat tells us how much energy is needed to raise a unit *mass* of material by one degree, what often matters in buildings is how much energy is needed to raise a unit *volume*. This is the **volumetric heat capacity**:

$$\rho c = \text{density} \times \text{specific heat} \quad [\text{J/m}^3 \cdot \text{K}]$$

Materials with high volumetric heat capacity can store significant amounts of thermal energy without large temperature swings. This property, called **thermal mass**, has important implications for building design:

- **Temperature stabilization**: High thermal mass materials (concrete, brick, stone) absorb excess heat during warm periods and release it during cool periods, moderating indoor temperature swings.
- **Peak load shifting**: Thermal mass can delay peak cooling loads by several hours, potentially shifting air conditioning demand to off-peak electricity rates.
- **Passive solar design**: In climates with significant day-night temperature differences, thermal mass can capture daytime solar gains and release them at night, reducing heating loads.

For comparison, water has a volumetric heat capacity of approximately 4,186 kJ/m³ · K, while concrete is around 2,000 kJ/m³ · K—meaning water stores about twice as much heat per unit volume. This is why phase-change materials and water tanks are sometimes used for thermal energy storage in buildings.

> **Note**
>
> Thermal mass is most beneficial in climates with significant diurnal (day-night) temperature variation. In consistently hot or consistently cold climates, the benefits are reduced, and insulation typically provides better returns.

10.4 Modes of Heat Transfer

Let's now more formally study thermal conduction, convection and radiation.

10.4.1 Conduction

Joseph Fourier (1768 - 1830) came up with the law of conduction heat transfer, which relates the rate at which heat transfer occurs through a material, to the temperature difference it is subjected to, and the distance through which conduction is occurring. It is actually very similar to Ohm's law relating current (the rate at which electric charge flows through a conductor, to the voltage difference and the resistance). In particular, Fourier's law shows that:

$$\dot{Q} = -kA\frac{dT}{dx}$$

Where k is the thermal conductivity of the material, A is the area through which heat is flowing, and $\frac{dT}{dx}$ is the temperature *gradient* at a specific point x in the material.

10.4.1.1 Conduction Through a Plane Wall

For steady-state conduction through a plane wall:

$$\dot{Q} = kA\frac{T_1 - T_2}{\Delta x}$$

(Assuming many things, including $T_1 > T_2$, no heat sources within the material, constant thermal conductivity, etc.)

This can also be stated as:

$$\dot{Q} = \frac{T_1 - T_2}{\Delta x/(kA)}$$

Which is similar to how one would define Ohm's Law (i.e., $I = V/R$), and shows that the term $\Delta x/(kA)$ can be thought of as the thermal resistance R of the wall.

> **Worked Example: Heat Loss Through a Concrete Wall**
>
> Consider a 200 mm (0.2 m) thick concrete wall with thermal conductivity $k = 1.4$ W/m $\cdot$ K and area $A = 10$ m^2. If the inside surface is at 20°C and the outside surface is at 5°C, what is the rate of heat loss?

$$\dot{Q} = kA\frac{T_1 - T_2}{\Delta x} = 1.4 \times 10 \times \frac{20 - 5}{0.2} = 1{,}050 \text{ W}$$

The thermal resistance of this wall is:

$$R = \frac{\Delta x}{kA} = \frac{0.2}{1.4 \times 10} = 0.0143 \text{ K/W}$$

Or, expressed per unit area (which is more commonly tabulated):

$$R'' = \frac{\Delta x}{k} = \frac{0.2}{1.4} = 0.143 \text{ m}^2 \cdot \text{K/W}$$

This area-normalized thermal resistance is what building codes typically call the **R-value**. Its inverse is the **U-value** (thermal transmittance): $U = 1/R'' = 7.0$ W/m$^2 \cdot$ K. We will explore R-values and U-values in more detail in the next section.

10.4.2 Convection

Convection is heat transfer between a surface and a moving fluid (which, in buildings, is usually air). Unlike conduction, which occurs through stationary material, convection involves bulk fluid motion that carries thermal energy.

10.4.2.1 Newton's Law of Cooling

The rate of convective heat transfer is described by Newton's law of cooling:

$$\dot{Q} = hA(T_s - T_\infty)$$

where:

- h is the **convection heat transfer coefficient** (W/m$^2 \cdot$ K)
- A is the surface area (m^2)
- T_s is the surface temperature
- T_∞ is the bulk fluid temperature far from the surface

This can be rewritten in resistance form:

$$\dot{Q} = \frac{T_s - T_\infty}{1/(hA)}$$

showing that the convective thermal resistance is $R_{conv} = 1/(hA)$, or per unit area: $R''_{conv} = 1/h$.

10.4.2.2 Natural vs. Forced Convection

The convection coefficient h depends strongly on whether the fluid motion is driven by external means or by buoyancy:

- **Natural (free) convection**: Fluid motion is driven by density differences caused by temperature variations. Warm air rises, cool air sinks. This is the dominant mode for interior building surfaces in still air.
- **Forced convection**: Fluid motion is driven by external means such as wind, fans, or pumps. This typically results in much higher heat transfer rates.

10.4.2.3 Typical Values for Buildings

Situation	h (W/m$^2 \cdot$ K)
Still air (natural convection, vertical surface)	5-10
Still air (natural convection, horizontal surface, heat rising)	7-12
Light wind (~2 m/s) on exterior surface	10-15
Moderate wind (~5 m/s) on exterior surface	20-30
Strong wind (~10 m/s) on exterior surface	40-60

Values are approximate and depend on surface geometry, orientation, and temperature difference.

Building codes typically use standardized surface resistances for interior and exterior surfaces. For example, ASHRAE specifies an interior surface resistance of approximately $R''_{si} = 0.12$ m$^2 \cdot$ K/W (corresponding to $h \approx 8$ W/m$^2 \cdot$ K) and an exterior surface resistance of $R''_{se} = 0.03$ m$^2 \cdot$ K/W for winter conditions (corresponding to $h \approx 33$ W/m$^2 \cdot$ K, assuming some wind).

10.4.3 Radiation

Radiation is heat transfer via electromagnetic waves. Unlike conduction and convection, radiation requires no physical medium—it can occur across a vacuum. All objects above absolute zero emit thermal radiation, with the intensity and spectrum depending on their temperature.

10.4.3.1 The Stefan-Boltzmann Law

The maximum rate at which a surface can emit thermal radiation is given by the Stefan-Boltzmann law:

$$\dot{Q}_{emit} = \varepsilon \sigma A T^4$$

where:

- ε is the **emissivity** of the surface (0 to 1, dimensionless)
- $\sigma = 5.67 \times 10^{-8}$ W/(m^2·K^4) is the Stefan-Boltzmann constant
- A is the surface area
- T is the **absolute temperature** in Kelvin

A surface with $\varepsilon = 1$ is called a *blackbody*—it emits the maximum possible radiation for its temperature. Real surfaces have $\varepsilon < 1$.

10.4.3.2 Emissivity of Common Building Materials

Material	Emissivity ε
Brick, concrete, stone	0.90-0.95
Glass	0.90-0.95
Wood	0.85-0.90
White paint	0.85-0.95
Aluminum (polished)	0.04-0.06
Aluminum (anodized)	0.70-0.80
Low-e coating	0.04-0.10

The low emissivity of polished metals and specialized low-e coatings is exploited in window design to reduce radiative heat transfer while maintaining visible light transmission.

10.4.3.3 Radiation Exchange Between Surfaces

When two surfaces "see" each other, they exchange radiation. The net heat transfer from surface 1 to surface 2 depends on their temperatures, emissivities, and geometry:

$$\dot{Q}_{1 \to 2} = \varepsilon_{eff} \sigma A_1 F_{1-2} (T_1^4 - T_2^4)$$

where F_{1-2} is the **view factor**—the fraction of radiation leaving surface 1 that reaches surface 2. View factors depend on geometry and can be complex to calculate, though tabulated values exist for common configurations.

For building applications, a useful simplification for small temperature differences is to linearize the radiation exchange:

$$\dot{Q}_{rad} \approx h_r A(T_1 - T_2)$$

where h_r is a **radiative heat transfer coefficient**:

$$h_r = 4\varepsilon\sigma T_m^3$$

with T_m being the mean absolute temperature of the two surfaces. At typical building temperatures (~300 K), this gives $h_r \approx 5$ W/m$^2 \cdot$ K for high-emissivity surfaces—comparable to natural convection.

> **Tip**
>
> You might want to prove to yourself that the linear approximation is appropriate (and under what conditions). Reading Chapter 2 (page 51) of Reddy et al. (2016) could help with this.

10.4.3.4 Solar Radiation

The sun is a special case of radiative heat transfer. Solar radiation incident on buildings is typically characterized by:

- **Direct (beam) radiation**: Arrives in a straight line from the sun
- **Diffuse radiation**: Scattered by the atmosphere, arrives from all directions
- **Reflected radiation**: Bounced off surrounding surfaces (ground, other buildings)

The total solar radiation on a surface is called **irradiance** and is measured in W/m^2. Peak values on a surface perpendicular to the sun can reach 1,000 W/m^2 on a clear day. The fraction of incident solar radiation that is absorbed depends on the surface's **solar absorptance** α_s, which can differ significantly from its thermal emissivity (especially for selective surfaces like low-e coatings).

> **Learn-by-Doing Activity: Comparing Heat Transfer Modes**
>
> Consider a single-pane glass window with the following properties:
>
> - Glass thickness: 6 mm

- Glass thermal conductivity: $k = 1.0$ W/m $\cdot$ K
- Indoor air temperature: 20°C
- Outdoor air temperature: 0°C
- Interior surface convection coefficient: $h_{in} = 8$ W/m$^2 \cdot$ K
- Exterior surface convection coefficient: $h_{out} = 25$ W/m$^2 \cdot$ K
- Glass emissivity: $\varepsilon = 0.9$

Tasks:

1. Calculate the thermal resistance per unit area for:
   - Interior convection: $R_{conv,in} = 1/h_{in}$
   - Conduction through glass: $R_{glass} = \Delta x/k$
   - Exterior convection: $R_{conv,out} = 1/h_{out}$
2. Calculate the total R-value and U-value for the window (neglecting radiation for now).
3. What is the heat loss per square meter?
4. If the interior glass surface is at approximately 10°C, estimate the radiative heat transfer from the glass to a room at 20°C using $h_r \approx 5$ W/m$^2 \cdot$ K. How does this compare to the convective heat transfer on the interior surface?

Expected insights: You should find that the glass itself contributes very little resistance (R = 0.006 m$^2 \cdot$ K/W). The surface film resistances dominate, and radiation contributes significantly to the interior surface heat transfer—which is why low-e coatings are so effective at improving window performance.

10.5 Thermal Properties of Building Materials

10.5.1 Thermal Conductivity

Thermal conductivity (k or λ) measures how readily a material conducts heat. It is defined as the rate of heat transfer through a unit thickness of material per unit area per unit temperature difference, with units of W/m $\cdot$ K (or equivalently, W/m $\cdot$ °C).

Materials with *low* thermal conductivity are good insulators; materials with *high* thermal conductivity are good conductors. The table below shows representative values spanning several orders of magnitude:

Material	Thermal Conductivity k (W/(m·K))	Category
Still air	0.026	Gas
Expanded polystyrene (EPS)	0.035-0.040	Insulation
Fiberglass batt	0.040-0.045	Insulation
Mineral wool	0.035-0.045	Insulation
Polyurethane foam	0.020-0.028	Insulation
Wood (softwood)	0.12-0.15	Structural
Gypsum board	0.16-0.25	Finish
Brick	0.6-1.0	Masonry
Glass	0.8-1.0	Glazing
Concrete (dense)	1.4-2.0	Structural
Stone (granite)	2.5-3.5	Masonry
Steel	45-50	Metal
Aluminum	200-220	Metal
Copper	380-400	Metal

Values are approximate and vary with density, moisture content, and temperature. Sources: Engineering ToolBox, GreenSpec.

Notice that still air has extremely low thermal conductivity—this is why most insulation materials work by trapping small pockets of air (fiberglass batts, foam cells). The challenge is preventing air movement, which would introduce convective heat transfer.

10.5.1.1 Factors Affecting Thermal Conductivity

- **Density**: For many materials (especially insulation), lower density means more trapped air and lower conductivity—up to a point where the material becomes too sparse to suppress convection.
- **Moisture content**: Water has a thermal conductivity of about 0.6 W/m · K, much higher than air. Wet insulation performs poorly.
- **Temperature**: Conductivity generally increases with temperature, though the effect is small for typical building temperature ranges.

10.5.2 Thermal Resistance (R-value)

The **R-value** (thermal resistance) quantifies how well a material or assembly resists heat flow. For a single homogeneous layer, as we've seen before:

$$R'' = \frac{\Delta x}{k}$$

where Δx is the thickness and k is the thermal conductivity. Higher R-values mean better insulation.

10.5.2.1 Units and Conversion

R-value units differ between SI and Imperial systems, which causes frequent confusion:

System	R-value Units	Symbol
SI (Metric)	$m^2 \cdot K/W$	RSI
Imperial (I-P)	$ft^2 \cdot °F \cdot h/BTU$	R

> **Tip**
>
> I may not be consistent in using R'' to refer to the per-unit-area version of the thermal resistance throughout this material. So instead of relying on the variable name to distinguish between them, just make sure you look at the units that are being used.

The conversion factor is:

$$R_{Imperial} = 5.678 \times R_{SI}$$

For example, an R-19 fiberglass batt (Imperial) has RSI $= 19/5.678 \quad 3.35\ m^2 \cdot K/W$.

> **Warning**
>
> When reading R-values, always check the units! An "R-19" wall in the U.S. is very different from "R-19" in a metric country. U.S. building codes use Imperial R-values; Canadian and European codes use RSI.

10.5.2.2 R-values Add in Series

A key property of thermal resistance: **R-values are additive for layers in series**. For a wall assembly with multiple layers:

$$R_{total} = R_1 + R_2 + R_3 + \cdots$$

This makes R-values convenient for calculating assembly performance. For example, a wall with:

- Interior air film: R-0.68 ($ft^2 \cdot °F \cdot h/BTU$)
- 1/2" gypsum board: R-0.45
- 3.5" fiberglass batt: R-13
- 1/2" plywood sheathing: R-0.62
- Exterior air film: R-0.17

has a total R-value of approximately R-15.

10.5.3 Thermal Conductance (U-value)

The **U-value** (thermal transmittance) is simply the inverse of R-value:

$$U = \frac{1}{R}$$

While R-value measures resistance to heat flow (higher is better), U-value measures how easily heat passes through (lower is better). U-values are typically expressed in $W/m^2 \cdot K$ (SI) or $BTU/h \cdot ft^2 \cdot °F$ (Imperial).

10.5.3.1 Why U-values?

Building energy codes often specify U-values rather than R-values because:

1. **U-values handle parallel paths**: When heat can flow through multiple parallel paths (e.g., through insulation AND through studs), the effective U-value is the area-weighted average, while R-values cannot be directly averaged.
2. **U-values work for complex assemblies**: Windows, for example, have different R-values for the glazing, frame, and spacer. The overall window U-value accounts for all paths.

10.5.3.2 Typical U-values

Assembly	U-value ($W/m^2 \cdot K$)	U-value ($BTU/h \cdot ft^2 \cdot °F$)
Single-pane window	5.5-6.0	0.95-1.05
Double-pane window (air)	2.7-3.0	0.47-0.53
Double-pane, low-e, argon	1.4-1.8	0.25-0.32
Triple-pane, low-e, argon	0.8-1.2	0.14-0.21
Well-insulated wall	0.2-0.3	0.035-0.053
Passive House wall	<0.15	<0.026

Window U-values are typically rated for the entire assembly including frame.

Learn-by-Doing Activity: Calculating Assembly R-values

A residential wall assembly consists of the following layers (from inside to outside):

Layer	Thickness	Thermal Conductivity k
Interior air film	—	$h = 8.3$ W/(m$^2 \cdot$ K)
Gypsum board	12.5 mm	0.16 W/m $\cdot$ K
Fiberglass batt insulation	89 mm	0.043 W/(m $\cdot$ K)
OSB sheathing	11 mm	0.13 W/(m $\cdot$ K)
Air gap (drainage cavity)	25 mm	0.02 W/(m $\cdot$ K)
Brick veneer	90 mm	0.72 W/(m $\cdot$ K)
Exterior air film	—	$h = 34$ W/(m$^2 \cdot$ K)

Tasks:

1. Calculate the R-value of each layer. For the air films, use $R = 1/h$.
2. Sum all R-values to find the total assembly R-value in m$^2 \cdot$ K/W.
3. Convert to Imperial R-value (multiply by 5.678). How does this compare to typical code requirements?
4. Calculate the overall U-value ($U_{overall}$).
5. Calculate the overall U-value excluding the air films and brick veneer ($U_{insulation}$).
6. If wood studs ($k = 0.12$ W/m $\cdot$ K) replace the wall assembly through its thickness (except for the brick veneer), covering 2% of the wall area, calculate the effective U-value of the whole assembly (including exterior air films and brick veneer) using the parallel path method.

Answers (check your work):

- Total R-value: approximately 3.757 m$^2 \cdot$ K/W (R-21 Imperial)
- $U_{overall}$: approximately 0.266 W/(m$^2 \cdot$ K)
- $U_{insulation}$: approximately 0.287 W/(m$^2 \cdot$ K)
- Effective U-value of studs + insulation in parallel (5% framing): approximately 0.316 W/(m$^2 \cdot$ K)
- With 5% framing, after adding back brick and air films: effective U-value increases to approximately 0.291 W/m$^2 \cdot$ K (a 9% degradation)

10.6 Thermal Resistance Networks

The equations for heat transfer through building assemblies can become complex when multiple layers, surfaces, and parallel paths are involved. **Thermal resistance networks** provide a systematic way to analyze these systems by drawing on an analogy with electrical circuits.

10.6.1 The Electrical Analogy

The mathematical similarity between heat flow and electrical current flow is striking:

Electrical Domain	Thermal Domain
Voltage V (Volts)	Temperature T (K or °C)
Current I (Amps)	Heat flow rate $\dot{Q}$ (W)
Electrical resistance R_e (Ohms)	Thermal resistance R (K/W)
Ohm's Law: $I = \frac{V_1 - V_2}{R_e}$	Fourier's Law: $\dot{Q} = \frac{T_1 - T_2}{R}$

This analogy means we can:

1. Draw thermal circuits just like electrical circuits
2. Use the same rules for combining resistances (series and parallel)
3. Apply circuit analysis techniques (Kirchhoff's laws, node analysis)

The key insight is that **temperature difference drives heat flow**, just as voltage difference drives current flow. And just as current is conserved at a junction, heat flow must be conserved (energy balance).

10.6.2 Resistances in Series and Parallel

10.6.2.1 Series Resistances (Layers)

When heat flows through multiple layers in sequence (as through a wall), the resistances add:

$$R_{total} = R_1 + R_2 + R_3 + \cdots$$

The same heat flow $\dot{Q}$ passes through each layer, but the temperature drops across each layer according to its resistance. This is analogous to resistors in series in an electrical circuit.

For a wall with interior air film, multiple material layers, and exterior air film:

$$R_{total} = R_{si} + \sum_i \frac{\Delta x_i}{k_i} + R_{se}$$

where R_{si} and R_{se} are the surface resistances (accounting for convection and radiation at the surfaces).

10.6.2.2 Parallel Resistances (Alternative Paths)

When heat can flow through different paths simultaneously (e.g., through studs AND insulation in a framed wall), the resistances combine in parallel:

$$\frac{1}{R_{total}} = \frac{1}{R_1} + \frac{1}{R_2} + \cdots$$

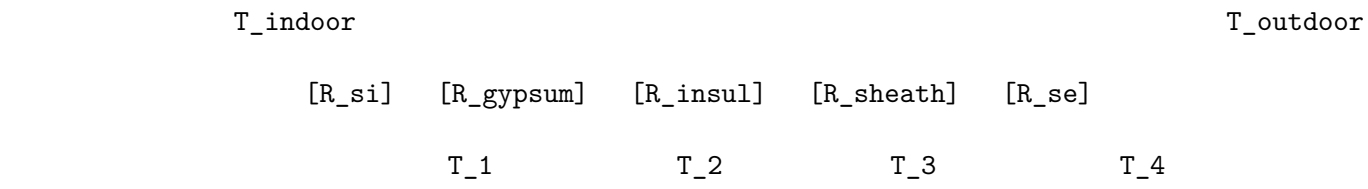
Or equivalently, using U-values (which are more convenient for parallel paths):

$$U_{total} = \frac{A_1}{A_{total}} U_1 + \frac{A_2}{A_{total}} U_2 + \cdots$$

This is an area-weighted average of the U-values, which explains why thermal bridging through studs, lintels, and other structural elements can significantly degrade the overall performance of an otherwise well-insulated wall.

10.6.3 Representing Wall Assemblies

A typical wall assembly can be represented as a series of thermal resistances from the indoor air to the outdoor air:



Where:

- R_{si} = interior surface resistance (~0.12 m² · K/W for still air)
- R_{gypsum} = gypsum board resistance = $\Delta x/k$
- R_{insul} = insulation resistance
- R_{sheath} = sheathing resistance
- R_{se} = exterior surface resistance (~0.03-0.06 m² · K/W depending on wind)

10.6.3.1 Worked Example: Simple Wall Assembly

Consider a wall with:

- Interior surface film: $R_{si} = 0.12$ m² · K/W
- 12.5 mm gypsum board ($k = 0.16$ W/m · K): $R = 0.0125/0.16 = 0.078$ m² · K/W
- 90 mm fiberglass ($k = 0.04$ W/m · K): $R = 0.09/0.04 = 2.25$ m² · K/W
- 12 mm plywood ($k = 0.13$ W/m · K): $R = 0.012/0.13 = 0.092$ m² · K/W
- Exterior surface film: $R_{se} = 0.04$ m² · K/W

Total R-value: $R_{total} = 0.12 + 0.078 + 2.25 + 0.092 + 0.04 = 2.58$ m² · K/W

Overall U-value: $U = 1/2.58 = 0.39$ W/m² · K

If the indoor temperature is 20°C and outdoor is 0°C, the heat loss per square meter is:

$$\dot{q} = U \times \Delta T = 0.39 \times 20 = 7.8 \text{ W/m}^2$$

10.6.3.2 Accounting for Thermal Bridges

In real framed walls, wood or steel studs create thermal bridges—paths of higher conductivity that bypass the insulation. A 38×89 mm (2×4) wood stud has:

$$R_{stud} = 0.089/0.12 = 0.74 \text{ m}^2 \cdot \text{K/W}$$

Compare this to the R-2.25 of the fiberglass it displaces! If studs occupy 15% of the wall area, the effective U-value is:

$$U_{eff} = 0.85 \times U_{insulated} + 0.15 \times U_{stud}$$

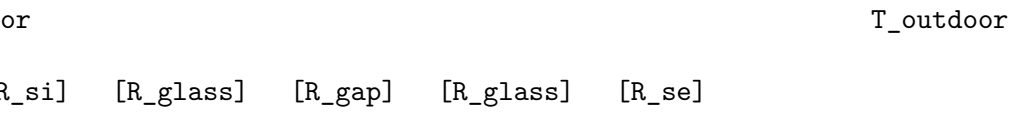
This "parallel path" calculation typically reduces the effective R-value by 10-25% compared to the nominal insulation R-value alone.

10.6.4 Representing Window Assemblies

Windows are more complex because they involve:

1. **Conduction** through the glass
2. **Convection** on both surfaces AND in any air gaps
3. **Radiation** exchange between glass panes and across air gaps

A double-pane window can be represented as:



Where R_{gap} includes both convection and radiation across the air (or gas) space. This is why:

- **Argon or krypton fill** improves performance (lower gas conductivity, suppressed convection)
- **Low-e coatings** dramatically reduce the radiative component of R_{gap}
- **Wider gaps** help—up to about 12-15 mm, beyond which convection currents begin to degrade performance

10.6.5 Introduction to Thermal Capacitance

So far, our thermal networks have been purely resistive—they describe *steady-state* heat flow where temperatures don't change with time. But buildings have **thermal mass**, and temperatures *do* change: outdoor conditions vary, HVAC systems cycle, internal loads fluctuate.

To model dynamic behavior, we add **thermal capacitance** to our networks:

Electrical Domain	Thermal Domain
Capacitance C_e (Farads)	Thermal capacitance C (J/K)
Charge storage: $Q = C_e V$	Heat storage: $Q = C \cdot T$
Current to capacitor: $I = C_e \frac{dV}{dt}$	Heat to thermal mass: $\dot{Q} = C \frac{dT}{dt}$

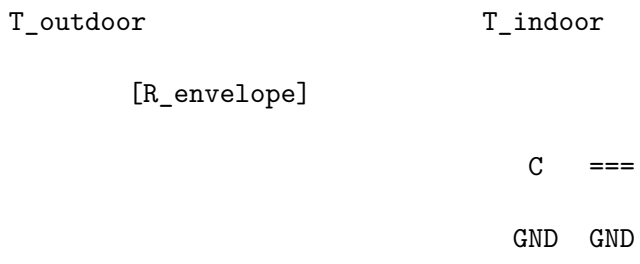
The thermal capacitance of a building element is:

$$C = m \cdot c = \rho \cdot V \cdot c$$

where m is mass, c is specific heat, ρ is density, and V is volume.

10.6.5.1 The RC Circuit Analogy

A simple thermal mass (like a room) connected to the outdoor environment through a resistive envelope can be modeled as an RC circuit:



The temperature response of this system follows the familiar first-order differential equation:

$$C\frac{dT_{indoor}}{dt} = \frac{T_{outdoor} - T_{indoor}}{R_{envelope}} + \dot{Q}_{internal}$$

This equation has a **time constant** $\tau = RC$ that characterizes how quickly the building responds to changes. We will explore this in detail in Lecture 6, deriving thermal network models from first principles and understanding what these time constants mean for building control.

10.7 Where do we go from here?

Throughout this lecture, we have largely treated heat transfer as a **steady-state** phenomenon. Our equations assume that temperatures remain constant over time, that heat flow rates are unchanging, and that the only role of time is in the cumulative accounting of degree-days over a season.

But is this realistic? Consider the following questions:

1. **How long does it take for a building to respond to a sudden change in outdoor temperature?** If the temperature drops from 10°C to 0°C overnight, does the indoor temperature immediately begin falling at a new rate, or is there some delay?
2. **When you turn on a heater, how long until the room warms up?** Our steady-state analysis cannot answer this—it tells us only where the temperature will eventually settle, not how quickly it gets there.
3. **Why do some buildings "coast" through cold snaps better than others?** A massive stone building and a lightweight prefab building might have identical R-values, yet one stays comfortable much longer when the heating fails.

4. **How should an HVAC controller anticipate occupancy?** If people arrive at 8 AM, should the system start heating at 6 AM? 7 AM? This depends on the building's dynamic response—something steady-state analysis cannot capture.
5. **Can thermal mass help reduce peak cooling loads?** We mentioned that thermal mass can shift loads to off-peak hours. But by how much? And under what conditions does this strategy actually help?

These questions all require understanding how building temperatures **evolve over time**. The missing ingredient is the **time constant**—a characteristic that emerges when we combine thermal resistance with thermal capacitance.

10.7.1 Preview of Part II

In Lecture 6, we will extend these steady-state concepts to dynamic thermal behavior by:

- Deriving thermal network models (RC circuits) from first principles using the heat equation
- Solving simple 1R1C and 2R2C networks to find building time constants
- Understanding what time constants physically mean and why they matter for control
- Writing the state-space representation of thermal network models

This dynamic perspective will be essential for the control and optimization problems you will tackle in the final project.

10.8 Additional Resources

10.8.1 Primary Reference

- Chapter 2 of Reddy (as noted in syllabus)

10.8.2 Supplementary Reading

- Appendix E of MacKay (2008) - "Sustainable Energy Without the Hot Air"

10.8.3 Online Resources

10.8.3.1 Degree-Day Data

- U.S. Energy Information Administration - Degree Days Explained - Clear explanation of HDD and CDD with regional U.S. data
- NOAA Climate Prediction Center - Degree Days Statistics - Official U.S. degree-day data and forecasts
- DegreeDays.net - Free worldwide heating and cooling degree-day data
- National Weather Service - Heating and Cooling Degree Days - Definitions and calculation methods

10.8.3.2 Material Properties and Calculators

- Engineering ToolBox - Thermal Conductivity of Materials - Comprehensive database of thermal properties
- GreenSpec - Insulation Materials - Building-focused thermal properties
- Building America Solution Center - Thermal Mass - Guide to thermal mass in buildings
- UpCodes - Building code requirements including insulation R-values by climate zone

11 Thermal Dynamics of Buildings: Part II

11.1 Lecture Overview

Learning Objectives

By the end of this module, students will be able to:

- Derive thermal network models from first principles:
  - Describe the elements of the Fourier heat equation (the 1-D Fourier transient heat conduction PDE and its steady-state version), including its solution as a sum of steady state plus an infinite sum of exponentials
  - Recognize the link between the Fourier heat equation PDE and the Thermal Network Model, understanding it as a finite difference approach using physical construction-based slicing/lumping that leads to a set of N simultaneous first-order differential equations to be solved
  - Understand the link between Kirchhoff's voltage/current laws and the heat balance equations we can work out in thermal network models
  - Recognize that the linearity assumption means with respect to the behavior of network elements (independent of temperature and time), and other implications
  - Directly represent radiation heat transfer interactions, lumped building mass components, solar radiation, heat extraction by HVAC, infiltration heat paths
- Solve 1R1C and 2R2C networks to find a building's time constants
  - Understand the importance of time constants even if they are just approximations
- Write down the linear state-space representation of an arbitrary thermal network model

Topics Covered

- The Fourier heat equation: from steady-state to transient
- Analytical solution of the heat equation
- Finite difference discretization and lumped parameter models

- From PDEs to thermal network models (RC circuits)
- Kirchhoff's laws and heat balance equations
- Linearity assumptions and their implications
- Representing heat transfer elements in networks
- Solving 1R1C networks: the single time constant model
- Solving 2R2C networks: multiple time constants
- Physical interpretation of time constants
- State-space representation of thermal networks

Project Milestones

This module provides the mathematical foundation for building thermal models that can be:

- Used in model predictive control (MPC) strategies
- Identified from data using system identification techniques
- Simulated to predict building response to weather and HVAC inputs
- Analyzed for controllability and observability

Understanding these models is essential for designing intelligent HVAC controllers in your final project.

11.2 From Steady-State to Dynamic: The Need for Time

In Lecture 5, we developed a solid understanding of steady-state heat transfer. We learned to calculate heat loss rates, R-values, and U-values—but we largely ignored time. The questions we posed at the end of that lecture highlighted this gap:

- How long does it take for a building to respond to temperature changes?
- When should we start heating before occupants arrive?
- Why do massive buildings "coast" through cold snaps better than lightweight ones?

To answer these questions, we need to understand how temperatures *evolve* over time. This requires moving from algebraic equations to differential equations.

11.2.1 A Motivating Example: The Office Room Revisited

Let's revisit a concrete example to frame the questions we'll answer in this lecture. Consider a small office room with dimensions 4m $\times$ 4m $\times$ 2.7m. The room has one exterior wall (4m $\times$ 2.7m = 10.8 m^2) containing a 2 m^2 double-pane window. The other walls, floor, and ceiling are adjacent to conditioned spaces (assume adiabatic for this analysis).

Given:

- Exterior wall U-value: 0.35 W/m^2·K
- Window U-value: 2.0 W/m^2·K
- Air density: 1.2 kg/m^3
- Air specific heat: 1,005 J/kg·K
- Indoor temperature: 20°C
- Outdoor temperature: 0°C

From Lecture 5, we can calculate the steady-state heat loss:

- Wall conductance: $U_{wall} \times A_{wall} = 0.35 \times 8.8 = 3.08$ W/K
- Window conductance: $U_{window} \times A_{window} = 2.0 \times 2.0 = 4.00$ W/K
- Total UA: 7.08 W/K
- Heat loss: $\dot{Q} = UA \times \Delta T = 7.08 \times 20 = 142$ W

But now let's ask *dynamic* questions:

1. What is the **thermal capacitance** of the air in this room?

$$C_{air} = \rho \cdot V \cdot c = 1.2 \times 43.2 \times 1005 = 52{,}100 \text{ J/K} \approx 52 \text{ kJ/K}$$

2. What is the **time constant** of this room?

$$\tau = R \times C = \frac{C}{UA} = \frac{52{,}100}{7.08} = 7{,}360 \text{ s} \approx 2 \text{ hours}$$

3. If the heating fails at midnight when $T_{in} = 20$°C and $T_{out} = 0$°C, how does the temperature evolve? How long until it drops below 15°C?

4. If this room had a 50mm thick exposed concrete floor (16 m^2, $\rho = 2300$ kg/m^3, $c = 880$ J/kg·K), the thermal capacitance would jump to over 4,000 kJ/K, and the time constant would become ~73 hours. Why does this matter for building control?

By the end of this lecture, you'll be able to answer all of these questions and understand the mathematical framework behind them. Notice how the window, despite being only 19% of the wall area, contributes 57% of the heat loss—a reminder that thermal networks can reveal non-obvious insights about building performance.

11.3 The Fourier Heat Equation

11.3.1 The 1-D Transient Heat Conduction PDE

In Lecture 5, we derived Fourier's law for steady-state conduction:

$$\dot{Q} = -kA\frac{dT}{dx}$$

But this assumes temperatures don't change with time. To understand *dynamic* behavior, we need to derive the **transient heat equation** from an energy balance.

Consider a thin slab of material with thickness Δx, area A, density ρ, and specific heat c. The rate of change of energy stored in this slab equals the net heat flow into it:

$$\rho c A \Delta x \frac{\partial T}{\partial t} = \dot{Q}_{in} - \dot{Q}_{out}$$

Using Fourier's law for the heat flows at positions x and $x + \Delta x$:

$$\rho c A \Delta x \frac{\partial T}{\partial t} = -kA\frac{\partial T}{\partial x}\bigg|_{x} - \left(-kA\frac{\partial T}{\partial x}\bigg|_{x+\Delta x}\right)$$

Dividing by $A\Delta x$ and taking the limit as $\Delta x \to 0$:

$$\rho c \frac{\partial T}{\partial t} = k\frac{\partial^2 T}{\partial x^2}$$

Convince yourself: Algebraic steps to derive the heat equation

Let's work through the algebra carefully to see how we get from the energy balance to the heat equation.
Starting equation:

$$\rho c A \Delta x \frac{\partial T}{\partial t} = -kA\frac{\partial T}{\partial x}\bigg|_{x} - \left(-kA\frac{\partial T}{\partial x}\bigg|_{x+\Delta x}\right)$$

Step 1: Handle the double negative
The second term has $-(-kA...)$, which becomes positive:

$$\rho c A \Delta x \frac{\partial T}{\partial t} = -kA\frac{\partial T}{\partial x}\bigg|_{x} + kA\frac{\partial T}{\partial x}\bigg|_{x+\Delta x}$$

Step 2: Factor out kA and rearrange

$$\rho c A \Delta x \frac{\partial T}{\partial t} = kA \left(\left. \frac{\partial T}{\partial x} \right|_{x+\Delta x} - \left. \frac{\partial T}{\partial x} \right|_{x} \right)$$

Note the order: we have (gradient at $x + \Delta x$) minus (gradient at x).

Step 3: Divide by $A\Delta x$

$$\rho c \frac{\partial T}{\partial t} = \frac{k}{\Delta x} \left(\left. \frac{\partial T}{\partial x} \right|_{x+\Delta x} - \left. \frac{\partial T}{\partial x} \right|_{x} \right)$$

Step 4: Take the limit as $\Delta x \to 0$

The term in brackets divided by Δx is exactly the definition of the derivative of $\frac{\partial T}{\partial x}$ with respect to x:

$$\lim_{\Delta x \to 0} \frac{1}{\Delta x} \left(\left. \frac{\partial T}{\partial x} \right|_{x+\Delta x} - \left. \frac{\partial T}{\partial x} \right|_{x} \right) = \frac{\partial^2 T}{\partial x^2}$$

Final result:

$$\rho c \frac{\partial T}{\partial t} = k \frac{\partial^2 T}{\partial x^2}$$

Or, in its standard form:

$$\boxed{\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2}}$$

where $\alpha = k/(\rho c)$ is the **thermal diffusivity** (units: m^2/s).

Physical interpretation:

- The left side, $\partial T/\partial t$, is the rate of temperature change at a point
- The right side, $\alpha \partial^2 T/\partial x^2$, involves the *curvature* of the temperature profile
- Where the temperature profile is curved (concave up), heat flows in from both sides, causing temperature to rise
- Where the profile is straight (no curvature), there's no net accumulation of heat

Convince yourself: What does "curvature" mean physically?

The second derivative $\partial^2 T/\partial x^2$ measures how much the temperature gradient changes with position. Consider these three scenarios through a wall:

1. Hot spot in the middle (temperature profile curves downward, concave down):

```
T |      *  ← hot spot
  |     / \
  |    /   \
```

```
  |  /     \
  |---------→ x
```

Heat flows away from the hot spot toward both cooler sides. The gradient $\partial T/\partial x$ is positive (upward slope) on the left and negative (downward slope) on the right. As we move through the hot spot, the gradient *decreases* (becomes less positive, then negative), so $\partial^2 T/\partial x^2 < 0$. Result: temperature at the hot spot *decreases* over time.

2. Cold spot in the middle (temperature profile curves upward, concave up):

```
T |  \     /
  |   \   /
  |    \ /
  |     * ← cold spot
  |---------→ x
```

Heat flows toward the cold spot from both warmer sides. The gradient is negative (downward slope) on the left and positive (upward slope) on the right. As we move through the cold spot, the gradient *increases*, so $\partial^2 T/\partial x^2 > 0$. Result: temperature at the cold spot *increases* over time.

3. Linear profile (no curvature):

```
T |  \
  |   \
  |    \
  |     \
  |---------→ x
```

The gradient is constant throughout. No matter where we are, heat flows through at a steady rate—what comes in equals what goes out. $\partial^2 T/\partial x^2 = 0$. Result: temperature *doesn't change* with time (steady state).

Key insight: The heat equation says temperature changes fastest where curvature is greatest. Heat "diffuses" from convex regions (hot spots) toward concave regions (cold spots), gradually smoothing out the temperature profile until it becomes linear (steady state).

11.3.2 Steady-State as a Special Case

When the system reaches steady state, temperatures stop changing: $\partial T/\partial t = 0$. The heat equation reduces to:

$$\frac{d^2T}{dx^2} = 0$$

This equation says the temperature profile has zero curvature—it must be a straight line! This is exactly what we assumed in Lecture 5 when we wrote:

$$\dot{Q} = kA\frac{T_1 - T_2}{\Delta x}$$

The linear temperature profile through a wall is the steady-state solution to the Fourier heat equation.

Convince yourself

Let's show explicitly how the zero-curvature condition leads to the Lecture 5 formula.
Starting from the steady-state heat equation:

$$\frac{d^2T}{dx^2} = 0$$

Solving this ordinary differential equation:
First integration gives:

$$\frac{dT}{dx} = C_1$$

where C_1 is a constant. Second integration gives:

$$T(x) = C_1 x + C_2$$

This confirms the temperature profile is linear in space.
Applying boundary conditions:
Consider a wall extending from $x = 0$ to $x = \Delta x$, with temperatures $T(0) = T_1$ (hot side) and $T(\Delta x) = T_2$ (cold side).
From $T(0) = T_1$:

$$C_2 = T_1$$

From $T(\Delta x) = T_2$:

$$T_2 = C_1\Delta x + T_1$$

$$C_1 = \frac{T_2 - T_1}{\Delta x}$$

The key insight:
Recall from our first integration that $\frac{dT}{dx} = C_1$ everywhere in the wall (the gradient is constant). We've now determined from the boundary conditions that $C_1 = \frac{T_2 - T_1}{\Delta x}$. Therefore, by substitution:

$$\frac{dT}{dx} = \frac{T_2 - T_1}{\Delta x} \quad \text{(constant throughout the wall)}$$

Applying Fourier's law:

Since the gradient is constant everywhere in the wall, we can apply Fourier's law:

$$\dot{Q} = -kA\frac{dT}{dx} = -kA\frac{T_2 - T_1}{\Delta x} = kA\frac{T_1 - T_2}{\Delta x}$$

This is **exactly** the formula we used in Lecture 5! It's not an approximation or assumption—it's the exact steady-state solution to the Fourier heat equation. The linear temperature profile with constant gradient emerges naturally when $\partial T/\partial t = 0$.

11.3.3 Solution Structure: Steady-State Plus Transients

The general solution to the heat equation (with appropriate boundary conditions) can be found using a technique called **separation of variables**. This method splits the PDE into two ODEs—one for time, one for space. The time component leads to exponential decay, while the spatial component gives rise to eigenfunctions. The result is a beautiful structure:

$$T(x,t) = T_{ss}(x) + \sum_{n=1}^{\infty} A_n \phi_n(x) e^{-t/\tau_n}$$

where:

- $T_{ss}(x)$ is the **steady-state solution**—the linear profile we calculated in Lecture 5
- The summation represents **transient modes**, each with its own spatial shape $\phi_n(x)$ and time constant τ_n
- The coefficients A_n depend on the initial conditions

Key insight: As $t \to \infty$, all the exponential terms decay to zero, leaving only the steady-state solution. The transient terms describe *how* the system approaches steady state, not *where* it ends up.

Explicit form with sinusoidal eigenfunctions

The abstract form $T(x,t) = T_{ss}(x) + \sum_{n=1}^{\infty} A_n \phi_n(x) e^{-t/\tau_n}$ hides the spatial details. Let's connect it to the explicit solution with sinusoidal eigenfunctions that emerges from separation of variables.
The most general solution to the 1-D heat equation can be written as:

$$T(x,t) = A_0 + B_0 x + \sum_{n=1}^{\infty} \left[A_n \cos(\lambda_n x) + B_n \sin(\lambda_n x)\right] e^{-\lambda_n^2 \alpha t}$$

where λ_n are eigenvalues determined by the boundary conditions.
Connecting the two forms:

1. **Steady-state part**: $T_{ss}(x) \leftrightarrow A_0 + B_0 x$
    - For fixed boundary temperatures, this is the linear profile we derived earlier
    - $B_0 = (T_2 - T_1)/L$ gives the constant gradient
    - $A_0 = T_1$ sets the reference temperature
2. **Spatial eigenfunctions**: $\phi_n(x) \leftrightarrow A_n \cos(\lambda_n x) + B_n \sin(\lambda_n x)$
    - These are the sinusoidal "shapes" that the temperature profile can take
    - For a wall with fixed temperatures at $x = 0$ and $x = L$, boundary conditions force $A_n = 0$ (no cosine terms) and $\lambda_n = n\pi/L$
    - This gives $\phi_n(x) = \sin(n\pi x/L)$
3. **Time constants**: $\tau_n = \frac{1}{\lambda_n^2 \alpha}$
    - The exponential e^{-t/τ_n} is exactly the same as $e^{-\lambda_n^2 \alpha t}$
    - For $\lambda_n = n\pi/L$: $\tau_n = \frac{L^2}{n^2 \pi^2 \alpha}$
    - Higher modes ($n = 2, 3, ...$) decay as $1/n^2$, so they vanish quickly

Physical meaning of the eigenfunctions:
Each $\sin(n\pi x/L)$ represents a spatial "mode" with n half-wavelengths across the wall thickness:

- $n = 1$: One smooth arc from $x = 0$ to $x = L$ (fundamental mode, slowest decay)
- $n = 2$: Two half-waves (decays $4\times$ faster)
- $n = 3$: Three half-waves (decays $9\times$ faster)

The specific mix of these modes (the coefficients A_n or B_n) depends on the initial temperature distribution $T(x, 0)$, determined by Fourier analysis.

The time constants τ_n tell us how quickly each mode decays. From the eigenvalue analysis (detailed in the derivation below), for a wall of thickness L:

$$\tau_n \propto \frac{L^2}{\alpha n^2}$$

The first mode ($n = 1$) decays slowest and dominates the long-term response. Higher modes decay much faster (proportional to $1/n^2$).

Convince yourself: Separation of variables derivation

Let's derive this solution structure from first principles using the method of separation of variables.
Step 1: Assume a separable solution

We look for solutions of the form:

$$T(x,t) = X(x) \cdot \Theta(t)$$

where $X(x)$ depends only on position and $\Theta(t)$ depends only on time.

Step 2: Substitute into the heat equation

Starting with $\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2}$ and substituting our assumed form:

$$X(x)\frac{d\Theta}{dt} = \alpha \Theta(t)\frac{d^2X}{dx^2}$$

Step 3: Separate the variables

Divide both sides by $X(x)\Theta(t)$:

$$\frac{1}{\Theta}\frac{d\Theta}{dt} = \frac{\alpha}{X}\frac{d^2X}{dx^2}$$

The left side depends only on t, the right side only on x. For this to be true for all x and t, both sides must equal the same constant. We call this constant $-\lambda$ (the negative sign is chosen for convenience):

$$\frac{1}{\Theta}\frac{d\Theta}{dt} = -\lambda \quad \text{and} \quad \frac{\alpha}{X}\frac{d^2X}{dx^2} = -\lambda$$

Step 4: Solve the time equation

From $\frac{d\Theta}{dt} = -\lambda\Theta$, we get:

$$\Theta(t) = e^{-\lambda t}$$

If we define $\tau = 1/\lambda$, this becomes $\Theta(t) = e^{-t/\tau}$—our exponential decay!

Step 5: Solve the spatial equation

The spatial equation becomes:

$$\frac{d^2X}{dx^2} = -\frac{\lambda}{\alpha}X$$

This is an eigenvalue problem. The solutions are sinusoidal functions whose specific form depends on the boundary conditions. For a wall with fixed boundary temperatures, we get:

$$X_n(x) = \phi_n(x) = \sin\left(\frac{n\pi x}{L}\right)$$

with corresponding eigenvalues:

$$\lambda_n = \frac{n^2\pi^2\alpha}{L^2}$$

This gives time constants:

$$\tau_n = \frac{1}{\lambda_n} = \frac{L^2}{n^2\pi^2\alpha}$$

Step 6: Superposition principle

Because the heat equation is linear, we can sum all possible solutions:

$$T(x,t) = \sum_{n=1}^{\infty} A_n \phi_n(x) e^{-t/\tau_n}$$

Step 7: Add the steady-state solution
The complete solution must also satisfy the boundary conditions at all times. The full solution is:

$$T(x,t) = T_{ss}(x) + \sum_{n=1}^{\infty} A_n \phi_n(x) e^{-t/\tau_n}$$

where $T_{ss}(x)$ is the steady-state profile (linear, as we showed earlier), and the coefficients A_n are determined by matching the initial temperature distribution at $t = 0$.
Physical interpretation:

- Each mode n has its own spatial pattern $\phi_n(x)$ and decay rate $1/\tau_n$
- Higher modes ($n > 1$) decay much faster than the fundamental mode ($n = 1$)
- After a few multiples of τ_1, only the steady-state solution remains
- The separation constant λ emerges as a physical eigenvalue that determines both the spatial pattern and the time constant

11.3.4 Thermal Diffusivity and Its Meaning

Thermal diffusivity $\alpha = k/(\rho c)$ tells us how quickly temperature disturbances propagate through a material. It combines:

- k: how easily heat conducts (higher $k \rightarrow$ faster propagation)
- ρc: how much heat the material stores per unit volume (higher $\rho c \rightarrow$ slower response)

Materials with high diffusivity respond quickly to surface temperature changes; materials with low diffusivity respond slowly.

Material	k (W/m·K)	ρ (kg/m^3)	c (J/kg·K)	α (mm^2/s)
Copper	385	8,940	385	112
Aluminum	205	2,700	900	84
Steel	50	7,850	460	14
Stone (granite)	2.8	2,600	820	1.3
Concrete	1.4	2,300	880	0.69
Brick	0.72	1,700	840	0.50
Glass	0.9	2,500	840	0.43

Material	k (W/m·K)	ρ (kg/m^3)	c (J/kg·K)	α (mm^2/s)
Gypsum board	0.16	800	1,000	0.20
Wood (softwood)	0.12	500	1,200	0.20
Water	0.6	1,000	4,186	0.14

Sources: Engineering ToolBox, Engineers Edge

Notice the huge range: copper's diffusivity is nearly 1000× that of water! Metals respond almost instantaneously to surface heating; water and masonry materials respond very slowly.

11.3.5 Thermal Penetration Depth

A useful concept for building intuition is the **thermal penetration depth**—the distance a temperature disturbance travels into a material in a given time.

Recall from our eigenvalue analysis that the time constant scales as $\tau \sim L^2/\alpha$, where L is a characteristic length. Rearranging this relationship, we can express the length scale that responds over a time t as $L \sim \sqrt{\alpha t}$. This gives us the penetration depth.

For a transient event of duration t:

$$\delta \approx \sqrt{\alpha t}$$

For periodic heating (like the daily outdoor temperature cycle with period T):

$$\delta \approx \sqrt{\frac{\alpha T}{\pi}}$$

This tells us how much of a wall's mass actually "participates" in responding to temperature fluctuations.

Learn-by-Doing Activity: Thermal Penetration Depth

Using the formula $\delta = \sqrt{\alpha t}$, calculate how far a temperature disturbance penetrates into different materials over various time scales.
Given thermal diffusivities:

- Concrete: $\alpha = 0.69$ mm^2/s (6.9×10^{-7} m^2/s)
- Brick: $\alpha = 0.50$ mm^2/s

- Wood: $\alpha = 0.20$ mm^2/s

Tasks:

1. Calculate the penetration depth for each material after:
   - 1 hour (3,600 s)
   - 1 day (86,400 s)
   - 1 week (604,800 s)
2. For daily outdoor temperature cycles, how deep does the temperature wave penetrate into a concrete wall? (Use $\delta = \sqrt{\alpha T/\pi}$ with $T = 86,400$ s)
3. A typical concrete wall is 200 mm thick. Based on your calculation, does the *entire* wall participate in daily temperature swings, or only the outer portion?
4. What does this imply about modeling walls for daily HVAC control versus seasonal energy calculations?

Answers:

Material	1 hour	1 day	1 week
Concrete	50 mm	244 mm	647 mm
Brick	43 mm	209 mm	552 mm
Wood	27 mm	132 mm	348 mm

For daily cycles in concrete: $\delta \approx 138$ mm. This means only the outer ~140 mm of a concrete wall responds significantly to daily temperature swings—the inner core stays at a nearly constant temperature. This is why we can often "lump" the thermal mass rather than modeling the full distributed system.

11.4 From PDEs to Lumped Parameter Models

11.4.1 Why Lump? The Practical Challenge

The Fourier heat equation is a **partial differential equation (PDE)**—temperature varies continuously in both space and time. This is a *distributed parameter system* with infinitely many degrees of freedom.

For practical purposes—simulation, control design, parameter identification—we need models with a **finite number of states**. The solution is to *discretize* the spatial dimension, converting the PDE into a system of ordinary differential equations (ODEs).

The **lumped parameter approach** divides the continuous system into a finite number of *nodes*, each assumed to have a uniform temperature. The temperature gradient *between* nodes drives heat flow; the thermal mass *at* nodes determines how quickly temperatures change.

11.4.2 Finite Difference Discretization

Consider a wall divided into N nodes at positions $x_1, x_2, \ldots, x_N$ with spacing Δx. At each interior node i, we approximate the second derivative using the **central difference formula**:

$$\left.\frac{\partial^2 T}{\partial x^2}\right|_i \approx \frac{T_{i+1} - 2T_i + T_{i-1}}{(\Delta x)^2}$$

Substituting into the heat equation:

$$\frac{\partial T_i}{\partial t} = \alpha \frac{T_{i+1} - 2T_i + T_{i-1}}{(\Delta x)^2}$$

This is now an **ordinary differential equation** for $T_i(t)$—it depends only on time, not on continuous spatial derivatives. We have one such equation for each node, giving us N coupled first-order ODEs.

Rearranging:

$$\frac{dT_i}{dt} = \frac{\alpha}{(\Delta x)^2}(T_{i-1} - 2T_i + T_{i+1})$$

$$= \frac{\alpha}{(\Delta x)^2}(T_{i-1} - T_i) + \frac{\alpha}{(\Delta x)^2}(T_{i+1} - T_i)$$

Each term looks like heat flow driven by a temperature difference—divided by thermal resistance, into a thermal capacitance!

11.4.3 Physical Construction-Based Slicing

Rather than using uniform grid spacing, building scientists typically slice walls at **material boundaries**: the air film, drywall, insulation, sheathing, and exterior surface each become separate regions.

This "physical" discretization has several advantages:

1. **Material properties change at boundaries**: Conductivity, density, and specific heat are constant within each layer but jump at interfaces

2. **Fewer nodes needed**: A 3-node model of a wall (interior surface, insulation midpoint, exterior surface) often captures the essential dynamics
3. **Physical interpretation**: Each node corresponds to something tangible (the air, the wall's interior mass, the exterior surface)

The standard approach assigns:

- **One temperature** to each material layer (or to each "lump" of a thick layer)
- **Resistances** connecting adjacent nodes based on layer thickness and conductivity
- **Capacitances** at nodes based on the mass and specific heat of each lump

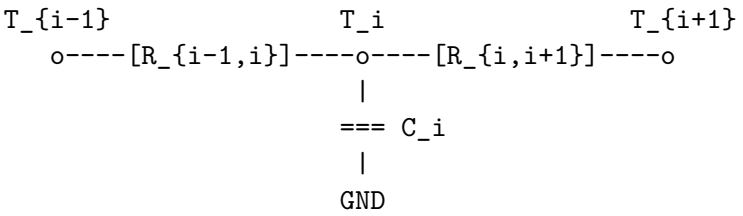
11.4.4 The Emergence of RC Networks

Let's see how the discretized equations map to RC circuits. Consider three adjacent nodes with temperatures T_{i-1}, T_i, and T_{i+1}, connected by thermal resistances $R_{i-1,i}$ and $R_{i,i+1}$, with node i having thermal capacitance C_i.

The heat balance at node i (Kirchhoff's Current Law applied to heat flow):

$$C_i \frac{dT_i}{dt} = \frac{T_{i-1} - T_i}{R_{i-1,i}} + \frac{T_{i+1} - T_i}{R_{i,i+1}}$$

This is **identical in form** to the equation for the voltage at a node in an RC circuit! The heat equation PDE has been transformed into a **thermal network model**.



For a uniform material with spacing Δx:

- $R = \frac{\Delta x}{kA}$ (resistance of each segment)
- $C = \rho c A \Delta x$ (capacitance of each node's mass)

And the product $RC = \frac{\rho c (\Delta x)^2}{k} = \frac{(\Delta x)^2}{\alpha}$, which is exactly the time scale that appears in our discretized equation.

The key insight: By discretizing the Fourier PDE, we've converted a distributed parameter system into a lumped parameter system that can be analyzed using circuit theory. All the tools of electrical network analysis—Kirchhoff's laws, series/parallel combinations, nodal analysis—apply directly to thermal networks.

11.5 Thermal Network Models

11.5.1 The Electrical-Thermal Analogy Revisited

In Lecture 5, we introduced the analogy between thermal and electrical systems for steady-state (resistive) analysis. Now we extend it to include dynamic (capacitive) elements:

Electrical Domain	Thermal Domain	Units
Voltage V	Temperature T	V $\leftrightarrow$ K (or °C)
Current I	Heat flow rate $\dot{Q}$	A $\leftrightarrow$ W
Charge q	Heat (energy) Q	C $\leftrightarrow$ J
Resistance R_e	Thermal resistance R	$\Omega \leftrightarrow$ K/W
Capacitance C_e	Thermal capacitance C	F $\leftrightarrow$ J/K
Ohm's Law: $I = \frac{V_1 - V_2}{R_e}$	Fourier's Law: $\dot{Q} = \frac{T_1 - T_2}{R}$	
Capacitor: $I = C_e \frac{dV}{dt}$	Thermal mass: $\dot{Q} = C\frac{dT}{dt}$	
Time constant: $\tau = R_e C_e$	Time constant: $\tau = RC$	s

The analogy is remarkably complete. Every concept from electrical circuit analysis has a thermal counterpart:

- **Series resistances** add: $R_{total} = R_1 + R_2$ (layers of a wall)
- **Parallel resistances** combine: $\frac{1}{R_{total}} = \frac{1}{R_1} + \frac{1}{R_2}$ (parallel heat paths)
- **RC time constants** characterize dynamic response
- **Kirchhoff's laws** enforce conservation

11.5.2 Kirchhoff's Laws for Heat Balance

The two fundamental laws of circuit analysis translate directly to thermal networks:

11.5.2.1 Kirchhoff's Current Law (KCL) $\rightarrow$ Conservation of Energy

> *The sum of heat flows into any node must equal zero (in steady state) or equal the rate of heat storage (in transient).*

For a node with temperature T_i, capacitance C_i, connected to neighboring nodes and receiving heat input $\dot{Q}_i$:

$$C_i \frac{dT_i}{dt} = \sum_j \frac{T_j - T_i}{R_{ij}} + \dot{Q}_i$$

This is our fundamental nodal equation. It says:

- Rate of energy storage (left side) = net heat flow in (right side)
- Heat flows from higher to lower temperature (like current from higher to lower voltage)

11.5.2.2 Kirchhoff's Voltage Law (KVL) → Temperature Consistency

Around any closed loop, the sum of temperature differences must equal zero.

$$\sum_{\text{loop}} \Delta T = 0$$

This is automatically satisfied if we define temperatures consistently at each node. It's less commonly used directly in thermal analysis but ensures our network is physically meaningful.

11.5.2.3 From KCL to a System of ODEs

Writing KCL at each capacitance node produces one differential equation per node. For a network with n capacitance nodes:

$$\begin{aligned} C_1 \frac{dT_1}{dt} &= (\text{sum of heat flows into node 1}) \\ C_2 \frac{dT_2}{dt} &= (\text{sum of heat flows into node 2}) \\ &\vdots \\ C_n \frac{dT_n}{dt} &= (\text{sum of heat flows into node n}) \end{aligned}$$

This system of n first-order ODEs completely describes the thermal dynamics of the building.

11.5.3 The Linearity Assumption

Our thermal network models are **linear** because we assume:

1. **Resistances are constant**: R doesn't depend on temperature or time
2. **Capacitances are constant**: C doesn't depend on temperature or time
3. **Heat flows are proportional to temperature differences**: $\dot{Q} = \Delta T / R$

When is linearity valid?

- Conduction: Generally linear. Thermal conductivity k varies weakly with temperature for most building materials.
- Convection: Approximately linear for forced convection; natural convection is weakly nonlinear but often linearized.
- Thermal mass: Linear as long as there are no phase changes (melting/freezing).

When does linearity break down?

- **Radiation**: Fundamentally nonlinear ($\dot{Q} \propto T^4$). However, we showed in Lecture 5 that for small temperature differences, radiation can be linearized as $\dot{Q} \approx h_r A \Delta T$ with $h_r = 4\varepsilon\sigma T_m^3$.
- **Phase change materials**: Highly nonlinear at the melting point.
- **Variable convection**: If airflow patterns change significantly with temperature.

Why linearity matters:

Linear systems have the **superposition property**: the response to multiple inputs equals the sum of responses to each input separately. This enables:

- Analytical solutions (eigenvalue methods)
- Efficient simulation
- Well-developed control theory (transfer functions, state-space methods)
- System identification from input-output data

For most building thermal analysis, the linearity assumption is excellent for temperature variations of $\$\pm 10 - 20\hat{}$ \$C around typical operating points.

11.6 Representing Heat Transfer Elements

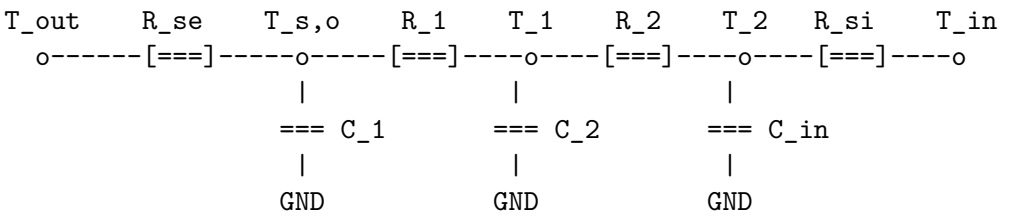
Now that we understand the general framework, let's see how specific heat transfer mechanisms are represented in thermal networks.

11.6.1 Conduction Through Walls

From Lecture 5, we know that conduction through a layer of material is represented by a thermal resistance:

$$R_{cond} = \frac{\Delta x}{kA}$$

For a multi-layer wall, we can represent it as a **chain of R-C-R-C elements**:



Where:

- R_{se}, R_{si} are surface film resistances (convection + radiation)
- R_1, R_2 are conduction resistances of material layers
- C_1, C_2 are thermal capacitances of each layer

How many nodes are needed? The thermal penetration depth gives guidance:

- For **fast dynamics** (hourly control): Only the outer ~50-100 mm of massive materials participates. A 2-3 node model is usually sufficient.
- For **slow dynamics** (daily/seasonal): More of the mass participates. May need 3-5 nodes for thick concrete walls.
- For **lightweight construction** (wood frame with insulation): 1-2 nodes often suffice because there's little thermal mass.

11.6.2 Convection at Surfaces

Surface heat transfer (combined convection and radiation to surroundings) is represented by a **surface film resistance**:

$$R_s = \frac{1}{h_c A}$$

where h_c is the combined surface heat transfer coefficient (convection + linearized radiation).

Standard values from building codes: - **Interior surfaces** (still air): $R_{si} \approx 0.12$ m^2·K/W $\rightarrow$ $h \approx 8$ W/m^2·K - **Exterior surfaces** (with wind): $R_{se} \approx 0.03 - 0.06$ m^2·K/W $\rightarrow$ $h \approx 17 - 33$ W/m^2·K

These resistances connect the solid surface node to the adjacent air node.

11.6.3 Radiation Heat Transfer

Radiation between surfaces is inherently nonlinear ($\dot{Q} \propto T^4$), but for small temperature differences it can be linearized:

$$\dot{Q}_{rad} = h_r A(T_1 - T_2)$$

where the **linearized radiation coefficient** is:

$$h_r = 4\varepsilon\sigma T_m^3$$

with T_m being the mean absolute temperature and $\sigma = 5.67 \times 10^{-8}$ W/m^2·K^4.

At typical building temperatures (~300 K) with high emissivity surfaces ($\varepsilon \approx 0.9$):

$$h_r \approx 4 \times 0.9 \times 5.67 \times 10^{-8} \times 300^3 \approx 5.5 \text{ W/m}^2 \cdot \text{K}$$

The corresponding radiation resistance is:

$$R_{rad} = \frac{1}{h_r A}$$

In networks, radiation often appears **in parallel** with convection at surfaces, since both transfer heat between the same nodes.

11.6.4 Lumped Building Mass (Thermal Capacitance)

The thermal capacitance at a node represents the heat storage capacity of the associated mass:

$$C = mc = \rho V c$$

where m is mass, V is volume, ρ is density, and c is specific heat.

Where to place capacitances matters for accuracy:

- **At the midpoint** of a material layer: Good for symmetric boundary conditions
- **At surfaces**: Good for capturing surface temperature dynamics
- **Split between surface and interior**: The common "3R2C" wall model

11.6.4.1 Common Wall Models

2R1C (simplest): One capacitance at the wall midpoint, resistances on either side

```
T_out ---[R/2]---o---[R/2]--- T_in
                 |
                === C
                 |
                GND
```

3R2C (more accurate): Splits the capacitance between interior and exterior portions

```
T_out ---[R1]---o---[R2]---o---[R3]--- T_in
                |          |
               === C1     === C2
                |          |
               GND        GND
```

The 3R2C model captures the fact that the inner portion of a wall responds differently than the outer portion to temperature changes.

11.6.5 Solar Radiation as a Heat Source

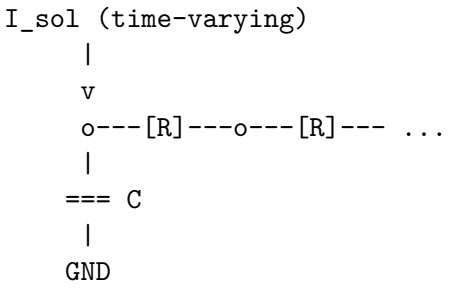
Solar radiation absorbed by a surface enters the network as a **heat source** (current source in the electrical analogy):

$$\dot{Q}_{sol} = \alpha_s \cdot I_{sol} \cdot A$$

where:

- α_s is the solar absorptance of the surface
- I_{sol} is the incident solar irradiance ($\mathrm{W/m^2}$)
- A is the surface area

In the network, this appears as a current injected at the surface node:



Solar gains are **time-varying inputs** that drive the system—they're known disturbances (from weather data) rather than states to be solved for.

11.6.6 HVAC Heat Extraction/Addition

HVAC systems add or remove heat from the building. They appear as **controlled heat sources/sinks**:

$$\dot{Q}_{HVAC}(t)$$

For **air-based systems** (forced air heating/cooling), the heat is typically injected at the indoor air node.

For **radiant systems** (radiant floors, ceiling panels), the heat is injected at the appropriate surface or mass node.

In control terms, $\dot{Q}_{HVAC}$ is the **control input**—it's what we manipulate to achieve desired temperatures.

11.6.7 Infiltration and Ventilation

Air exchange with outdoors brings in outdoor air at outdoor temperature. The heat flow is:

$$\dot{Q}_{inf} = \dot{m} c_p (T_{out} - T_{in})$$

where $\dot{m}$ is the mass flow rate of infiltrating air (kg/s) and $c_p \approx 1005$ J/kg·K.

This can be represented as a **thermal resistance** connecting indoor and outdoor air:

$$R_{inf} = \frac{1}{\dot{m} c_p}$$

Note that $\dot{m}$ depends on wind speed, stack effect, and mechanical ventilation—it may be time-varying.

In the network:

```
T_out ----[R_inf]---- T_in
```

Infiltration acts in parallel with the envelope heat loss path.

Learn-by-Doing Activity: Drawing a Complete Building Network

Draw the thermal resistance-capacitance network for a simple single-zone building with:

- One exterior wall (modeled as 3R2C)
- One window (resistance only, negligible thermal mass)
- Infiltration (at 0.5 air changes per hour for a 100 m^3 room)
- Internal gains from occupants and equipment: 500 W
- HVAC heating/cooling system

Tasks:

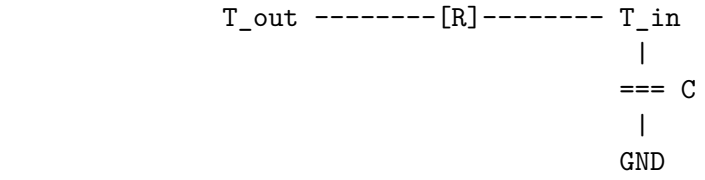
1. Sketch the network showing all R and C elements, heat source inputs, and node temperatures.
2. Label each element with its physical meaning (e.g., "exterior surface resistance," "wall thermal mass").
3. Identify which quantities are:
    - **States** (temperatures at capacitance nodes)
    - **Inputs** (outdoor temperature, solar radiation, HVAC power)
    - **Parameters** (resistances, capacitances)
4. Write the KCL equation for the indoor air node.

Hint: Your network should have at least 3 capacitance nodes (2 in the wall, 1 for indoor air) and at least 5 resistance elements.

11.7 Solving the 1R1C Network

11.7.1 The Simplest Building Model

The **1R1C model** reduces a building to its bare essentials: a single thermal mass (the building's interior and structure) connected to the outdoor environment through a single thermal resistance (the envelope).



We've already seen this model in action in our motivating example. Now let's develop the mathematical framework to understand *why* it behaves as it does.

Despite its simplicity, the 1R1C model captures the essential first-order dynamics of buildings and is widely used for:

- Quick energy estimates
- Understanding fundamental building behavior
- Initial control system design
- Benchmarking more complex models

11.7.2 Deriving the Governing Equation

Applying Kirchhoff's Current Law (energy conservation) at the indoor node:

$$\underbrace{C\frac{dT_{in}}{dt}}_{\text{rate of energy storage}} = \underbrace{\frac{T_{out} - T_{in}}{R}}_{\text{heat flow through envelope}} + \underbrace{\dot{Q}_{gains}}_{\text{internal + HVAC gains}}$$

Rearranging into standard form:

$$\frac{dT_{in}}{dt} = -\frac{1}{RC}T_{in} + \frac{1}{RC}T_{out} + \frac{1}{C}\dot{Q}_{gains}$$

This is a **first-order linear ordinary differential equation (ODE)**. The coefficient $-1/(RC)$ determines the system's dynamics.

11.7.3 The Time Constant

The **time constant** emerges naturally from the equation:

$$\tau = RC$$

It has units of time (check: [K/W] × [J/K] = [J/W] = [s]) and completely characterizes the building's dynamic response.

Physical meaning of τ:

- τ is the time required for the system to complete **63.2%** of its response to a step change (since $1 - e^{-1} \approx 0.632$)
- After 2τ: 86.5% complete
- After 3τ: 95.0% complete
- After 4τ: 98.2% complete (essentially at steady state)

What determines τ?

$$\tau = RC = \frac{1}{UA} \times mc = \frac{mc}{UA}$$

- **Large thermal mass** (m or c) → large τ → slow response
- **High envelope conductance** (UA) → small R → small τ → fast response

11.7.4 Analytical Solution

For constant outdoor temperature T_{out} and constant heat gains $\dot{Q}$, the ODE has an analytical solution:

$$\boxed{T_{in}(t) = T_{in,\infty} + (T_{in,0} - T_{in,\infty})e^{-t/\tau}}$$

where:

- $T_{in,0}$ is the initial indoor temperature at $t = 0$
- $T_{in,\infty}$ is the steady-state temperature as $t \to \infty$

The steady-state temperature is found by setting $dT_{in}/dt = 0$:

$$T_{in,\infty} = T_{out} + R \cdot \dot{Q}_{gains}$$

This makes physical sense: at steady state, the indoor temperature exceeds outdoor by an amount proportional to the heat gains and the thermal resistance.

Connection to the Fourier heat equation: Recall from our earlier analysis (Section 11.3) that the general solution to the transient heat equation has the form:

$$T(x,t) = T_{ss}(x) + \sum_{n=1}^{\infty} A_n \phi_n(x) e^{-t/\tau_n}$$

This was a steady-state solution plus an infinite sum of exponentially decaying modes, each with its own spatial pattern $\phi_n(x)$ and time constant τ_n.

The 1R1C solution above is **exactly this structure**, but with only a single mode ($n = 1$)! By lumping the building into one thermal mass, we've collapsed the infinite series down to a single exponential term:

- $T_{in,\infty}$ plays the role of T_{ss} (the steady-state solution)
- $(T_{in,0} - T_{in,\infty})$ is the coefficient A_1 (determined by initial conditions)
- $e^{-t/\tau}$ is the single exponential decay term
- The spatial variation $\phi_1(x)$ has disappeared—we have only one temperature, not a temperature distribution

We've lost the spatial detail but kept the essential dynamic behavior: exponential decay toward steady state with a characteristic time constant. This is the power of lumped parameter models—they capture the dominant physics while remaining simple enough to analyze and use for control.

Special cases:

1. **Free cooling (no heating, $\dot{Q} = 0$):** $T_{in,\infty} = T_{out}$

$$T_{in}(t) = T_{out} + (T_{in,0} - T_{out})e^{-t/\tau}$$

2. **Constant heating to maintain setpoint:** Solve for required $\dot{Q}$:

$$\dot{Q}_{required} = \frac{T_{setpoint} - T_{out}}{R} = UA \cdot (T_{setpoint} - T_{out})$$

11.7.5 Physical Interpretation

The time constant tells us how a building will behave:

Building Type	Typical τ	Characteristics
Lightweight office (steel frame, curtain wall)	2-5 hours	Responds quickly to heating/cooling; loses heat fast when HVAC stops
Standard residential	5-15 hours	Moderate response; can coast through short HVAC outages
Heavy masonry/concrete	15-50 hours	Very slow response; excellent thermal coasting; slow to warm up
Historic stone building	50-100+ hours	Extremely stable; seasonal rather than daily dynamics dominate

Example: Comparing two buildings

Consider two buildings with the same envelope resistance ($R = 0.01$ K/W, i.e., $UA = 100$ W/K) but different thermal mass:

- **Lightweight:** $C = 500$ kJ/K $\rightarrow \tau = 500,000/100 = 5,000$ s $\approx$ **1.4 hours**
- **Heavyweight:** $C = 5,000$ kJ/K $\rightarrow \tau = 5,000,000/100 = 50,000$ s $\approx$ **14 hours**

If both start at 20°C and outdoor temperature drops to 0°C with no heating:

- After 4 hours: Lightweight at 11.4°C, Heavyweight at 16.7°C
- After 8 hours: Lightweight at 6.5°C, Heavyweight at 14.0°C

The heavyweight building "coasts" much longer—critical for resilience during power outages or for taking advantage of thermal mass for load shifting.

Learn-by-Doing Activity: 1R1C Model Analysis

Return to our motivating example: the office room with $UA = 7.08$ W/K and $C = 52$ kJ/K (air only).
Tasks:

1. Verify that $\tau \approx 2$ hours using $\tau = C/UA$.
2. If heating fails at midnight when $T_{in} = 20$°C and $T_{out} = 0$°C, write the equation for $T_{in}(t)$ and calculate:
   - Temperature at 2 AM (after 2 hours)
   - Temperature at 6 AM (after 6 hours)
3. How long until T_{in} drops to 15°C? Solve $15 = 0 + 20 \cdot e^{-t/\tau}$ for t.

4. What heating power is required to maintain 20°C at steady state when $T_{out} = 0$°C?
5. If we add the concrete floor ($C_{total} = 4,120$ kJ/K), recalculate τ and the time to reach 15°C.

Answers:

1. $\tau = 52,000/7.08 = 7,345$ s ≈ 2.04 hours ✓
2. $T_{in}(t) = 0 + 20 \cdot e^{-t/7345}$
   - At 2 AM ($t = 7200$ s): $T_{in} = 20 \cdot e^{-0.98} = 7.5$°C
   - At 6 AM ($t = 21600$ s): $T_{in} = 20 \cdot e^{-2.94} = 1.1$°C
3. $15 = 20 \cdot e^{-t/\tau} \rightarrow t = -\tau \ln(0.75) = 0.288\tau = 35$ minutes
4. $\dot{Q} = UA \cdot \Delta T = 7.08 \times 20 = 142$ W
5. With concrete: $\tau = 4,120,000/7.08 = 582,000$ s ≈ 162 hours (!), time to 15°C $\approx$ 47 hours

The massive increase in time constant with concrete dramatically changes the building's behavior!

11.8 Solving the 2R2C Network

11.8.1 Why Two Capacitances?

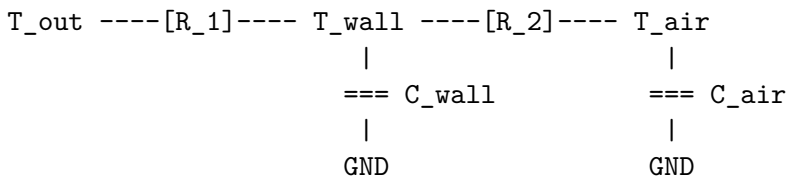
The 1R1C model lumps all thermal mass into a single capacitance. But real buildings have **distinct thermal masses** that respond at different rates:

- **Indoor air**: Low thermal mass, responds in minutes to hours
- **Furnishings and interior surfaces**: Medium mass, responds in hours
- **Building structure (walls, floors)**: High thermal mass, responds in hours to days

A **2R2C model** captures this separation of time scales with two capacitances, enabling more realistic predictions of: - Short-term temperature swings (air heats up fast, but walls lag) - The "warm up" period after turning on heating - How quickly a room "feels" comfortable versus how quickly it reaches thermal equilibrium

11.8.2 Network Configuration

The most common 2R2C configuration for buildings separates the indoor air from the building mass:



Where:

- T_{air} = indoor air temperature (what occupants feel immediately)
- T_{wall} = average temperature of interior surfaces and structure
- C_{air} = thermal capacitance of air + furnishings (small, ~50-200 kJ/K for a room)
- C_{wall} = thermal capacitance of building mass (large, ~500-5000 kJ/K)
- R_1 = resistance from outdoor to wall mass (exterior envelope)
- R_2 = resistance from wall mass to indoor air (interior surface film)

Heat gains (HVAC, occupants, solar) typically enter at the air node or wall node depending on whether they're convective or radiative.

11.8.3 Deriving the System of ODEs

Applying KCL at each capacitance node:

At the wall node (T_w):

$$C_w \frac{dT_w}{dt} = \frac{T_{out} - T_w}{R_1} + \frac{T_a - T_w}{R_2} + \dot{Q}_w$$

At the air node (T_a):

$$C_a \frac{dT_a}{dt} = \frac{T_w - T_a}{R_2} + \dot{Q}_a$$

Where $\dot{Q}_w$ represents heat gains absorbed by the walls (e.g., solar on interior surfaces) and $\dot{Q}_a$ represents gains to the air (e.g., HVAC, occupants).

These are **two coupled first-order ODEs**—the temperature of each node depends on the temperature of the other.

11.8.4 Two Time Constants

Unlike the 1R1C model with its single time constant, the 2R2C system has **two time constants**: τ_{fast} and τ_{slow}.

Intuitively:

- τ_{fast}: Dominated by the smaller capacitance (air). Governs the rapid initial response when heating starts.
- τ_{slow}: Dominated by the larger capacitance (structure). Governs the slow approach to final steady state.

Typically $\tau_{slow}/\tau_{fast} \approx 10 - 100$, meaning the system has two distinct response phases:

1. **Fast phase** (minutes to ~1 hour): Air temperature changes rapidly; walls barely respond
2. **Slow phase** (hours to days): Air and wall temperatures gradually equilibrate

This two-phase behavior explains why:

- A room can feel warm (air heated) even though walls are still cold
- Radiant heating (heats walls directly) feels different than forced-air heating (heats air)
- Buildings need "pre-conditioning" time before occupancy

11.8.5 Solving the System

To find the time constants, we write the system in **matrix form**. Let $\mathbf{x} = [T_w, T_a]^T$ be the state vector. The ODEs become:

$$\frac{d\mathbf{x}}{dt} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$$

where $\mathbf{u}$ contains the inputs (T_{out}, $\dot{Q}_w$, $\dot{Q}_a$) and:

$$\mathbf{A} = \begin{bmatrix} -\frac{1}{C_w}\left(\frac{1}{R_1} + \frac{1}{R_2}\right) & \frac{1}{C_w R_2} \\ \frac{1}{C_a R_2} & -\frac{1}{C_a R_2} \end{bmatrix}$$

The **eigenvalues** of $\mathbf{A}$ determine the time constants:

$$\lambda_1, \lambda_2 = \text{eigenvalues of } \mathbf{A}$$
$$\tau_1 = -\frac{1}{\lambda_1}, \quad \tau_2 = -\frac{1}{\lambda_2}$$

For thermal systems, both eigenvalues are negative (stable system) and real (no oscillations).

The general solution involves the **matrix exponential**:

$$\mathbf{x}(t) = e^{\mathbf{A}t}\mathbf{x}(0) + \int_0^t e^{\mathbf{A}(t-s)}\mathbf{B}\mathbf{u}(s)\,ds$$

For constant inputs, this simplifies to exponential decay toward steady state—but now with *two* exponential terms, each with its own time constant.

11.8.6 Physical Interpretation of Multiple Time Constants

The two-time-constant behavior has important practical implications:

When heating starts:

1. Air temperature rises quickly (fast time constant)
2. Walls remain cold, absorbing heat from the warm air
3. Gradually, walls warm up (slow time constant)
4. Eventually, everything equilibrates

When heating stops:

1. Air temperature drops quickly (fast time constant)
2. But warm walls release stored heat, slowing the air temperature drop
3. Room stays warmer longer than a 1R1C model would predict

This is why occupants might feel cold shortly after heating starts (walls radiating cold toward them) even though the air thermostat shows a comfortable temperature.

Learn-by-Doing Activity: 2R2C Model Simulation

Consider a room with the following 2R2C parameters:

- $C_{air} = 50$ kJ/K (air and light furnishings)
- $C_{wall} = 500$ kJ/K (interior wall mass)
- $R_1 = 0.05$ K/W (exterior envelope resistance)
- $R_2 = 0.01$ K/W (interior surface resistance)
- Initial conditions: $T_{air} = T_{wall} = 15°\text{C}$
- Outdoor temperature: $T_{out} = 0°\text{C}$
- HVAC heating: $\dot{Q} = 1000$ W applied to air node at $t = 0$

Tasks:

1. Write the system of ODEs in matrix form: $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$
2. Calculate the eigenvalues of $\mathbf{A}$ and determine the two time constants.

3. Using Python (or MATLAB), simulate the temperature response for 24 hours. Plot both $T_{air}(t)$ and $T_{wall}(t)$.

4. Create a 1R1C model with equivalent total R ($R_{total} = R_1 + R_2$) and C ($C_{total} = C_{air} + C_{wall}$). Simulate and compare.

5. What's different between the 1R1C and 2R2C responses? Which is more realistic?

Hints:

- The A matrix should be 2×2
- Use `numpy.linalg.eig()` for eigenvalues
- Use `scipy.integrate.solve_ivp()` for simulation
- Time constants: $\tau = -1/\lambda$ for each eigenvalue

Expected results:

- Fast time constant: ~10-20 minutes
- Slow time constant: ~5-10 hours
- Air temperature overshoots initially then settles as walls warm up
- 1R1C model misses this overshoot behavior

11.9 State-Space Representation

11.9.1 Why State-Space?

State-space representation is the **standard language** for describing dynamic systems in modern control theory. Converting our thermal network models to state-space form unlocks powerful tools:

- **Stability analysis**: Are all eigenvalues of A negative? (For thermal systems, yes)
- **Controllability**: Can we drive temperatures to any desired state using available inputs?
- **Observability**: Can we estimate all temperatures from available sensor measurements?
- **State feedback control**: Design controllers that use full state information
- **State estimation**: Kalman filters to estimate unmeasured temperatures
- **Model Predictive Control (MPC)**: Optimize control actions over a prediction horizon

For your final projects involving autonomous building control, state-space models will be essential.

11.9.2 The Standard Form

A **linear time-invariant (LTI) system** in state-space form consists of two equations:

$$\boxed{\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}}$$

(State equation)

$$\boxed{\mathbf{y} = \mathbf{C}\mathbf{x} + \mathbf{D}\mathbf{u}}$$

(Output equation)

Where:

Symbol	Name	Description	Dimensions
$\mathbf{x}$	State vector	Temperatures at capacitance nodes	$n \times 1$
$\mathbf{u}$	Input vector	External inputs (weather, HVAC, gains)	$m \times 1$
$\mathbf{y}$	Output vector	Measured quantities	$p \times 1$
$\mathbf{A}$	System matrix	Describes internal dynamics	$n \times n$
$\mathbf{B}$	Input matrix	How inputs affect states	$n \times m$
$\mathbf{C}$	Output matrix	What we measure	$p \times n$
$\mathbf{D}$	Feedthrough matrix	Direct input-to-output path	$p \times m$

For thermal systems, $\mathbf{D} = \mathbf{0}$ (no instantaneous effect of inputs on outputs).

11.9.3 Converting Thermal Networks to State-Space

Systematic procedure:

1. **Identify state variables**: One state per capacitance node (the temperatures)
2. **Write KCL equations**: Energy balance at each capacitance node
3. **Identify inputs**: Outdoor temperature, solar radiation, HVAC power, internal gains
4. **Rearrange into matrix form**: Isolate $d\mathbf{x}/dt$ on the left
5. **Extract A, B, C, D matrices**: Read off coefficients

Key insight: The structure of $\mathbf{A}$ comes directly from the network topology. Each entry A_{ij} represents how temperature at node j affects the rate of change of temperature at node i.

11.9.4 Worked Example: 1R1C in State-Space

Starting from our 1R1C equation:

$$C\frac{dT_{in}}{dt} = \frac{T_{out} - T_{in}}{R} + \dot{Q}$$

Divide by C and rearrange:

$$\frac{dT_{in}}{dt} = -\frac{1}{RC}T_{in} + \frac{1}{RC}T_{out} + \frac{1}{C}\dot{Q}$$

Define state and inputs:

- State: $x = T_{in}$ (scalar, $n = 1$)
- Inputs: $\mathbf{u} = \begin{bmatrix} T_{out} \\ \dot{Q} \end{bmatrix}$ ($m = 2$)

State-space matrices:

$$A = -\frac{1}{RC} = -\frac{1}{\tau}$$

$$B = \begin{bmatrix} \frac{1}{RC} & \frac{1}{C} \end{bmatrix} = \begin{bmatrix} \frac{1}{\tau} & \frac{1}{C} \end{bmatrix}$$

If we measure indoor temperature directly: $C = 1$, $D = \begin{bmatrix} 0 & 0 \end{bmatrix}$

The single eigenvalue of A is $\lambda = -1/\tau$, confirming our time constant.

11.9.5 Worked Example: 2R2C in State-Space

For the 2R2C network with states $\mathbf{x} = \begin{bmatrix} T_w \\ T_a \end{bmatrix}$ and inputs $\mathbf{u} = \begin{bmatrix} T_{out} \\ \dot{Q}_a \end{bmatrix}$:

Starting from the nodal equations:

$$C_w \frac{dT_w}{dt} = \frac{T_{out} - T_w}{R_1} + \frac{T_a - T_w}{R_2}$$

$$C_a \frac{dT_a}{dt} = \frac{T_w - T_a}{R_2} + \dot{Q}_a$$

Dividing by capacitances and rearranging:

$$\frac{dT_w}{dt} = -\frac{1}{C_w}\left(\frac{1}{R_1} + \frac{1}{R_2}\right)T_w + \frac{1}{C_w R_2}T_a + \frac{1}{C_w R_1}T_{out}$$

$$\frac{dT_a}{dt} = \frac{1}{C_a R_2}T_w - \frac{1}{C_a R_2}T_a + \frac{1}{C_a}\dot{Q}_a$$

State-space matrices:

$$\mathbf{A} = \begin{bmatrix} -\frac{1}{C_w}\left(\frac{1}{R_1} + \frac{1}{R_2}\right) & \frac{1}{C_w R_2} \\ \frac{1}{C_a R_2} & -\frac{1}{C_a R_2} \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} \frac{1}{C_w R_1} & 0 \\ 0 & \frac{1}{C_a} \end{bmatrix}$$

The eigenvalues of $\mathbf{A}$ give the two time constants: $\tau_i = -1/\lambda_i$.

11.9.6 Properties of the System Matrix A

For thermal systems, the system matrix $\mathbf{A}$ has special properties:

1. Stability (all eigenvalues have negative real parts)

Physical systems that dissipate energy are inherently stable. Heat always flows from hot to cold, so temperature differences decay over time. Mathematically, this means all eigenvalues of $\mathbf{A}$ satisfy $\text{Re}(\lambda_i) < 0$.

2. Real eigenvalues (no oscillations)

Pure thermal systems don't oscillate—temperatures decay monotonically toward equilibrium. This is because $\mathbf{A}$ for thermal networks is a special type of matrix (related to graph Laplacians) that guarantees real eigenvalues.

Note: Coupled thermo-fluid systems or systems with feedback control CAN oscillate, but the thermal subsystem alone cannot.

3. Eigenvalues give time constants

Each eigenvalue λ_i corresponds to a time constant:

$$\tau_i = -\frac{1}{\lambda_i}$$

The eigenvalue with smallest magnitude (closest to zero) gives the slowest time constant, which dominates the long-term response.

4. Structure reflects network topology

- Diagonal entries A_{ii} are negative (self-damping)
- Off-diagonal entries A_{ij} are positive if nodes i and j are connected
- Row sums relate to heat loss to external nodes (boundary conditions)

Learn-by-Doing Activity: State-Space Formulation

Consider the following 3-node thermal network representing a room with a wall:

```
T_out ---[R_1]--- T_wall ---[R_2]--- T_air ---[R_3]--- T_adj
                    |                  |
                   === C_w            === C_a
                    |                  |
                   GND                GND
```

Where:

- T_{out} = outdoor temperature (input)
- T_{adj} = adjacent room temperature (input, assume constant at 20°C)
- T_{wall} = wall mass temperature (state)
- T_{air} = room air temperature (state)
- $\dot{Q}_{hvac}$ = HVAC heat input to air node (input)

Parameters:

- $R_1 = 0.05$ K/W, $R_2 = 0.02$ K/W, $R_3 = 0.1$ K/W
- $C_w = 800$ kJ/K, $C_a = 100$ kJ/K

Tasks:

1. Write the KCL equation at each capacitance node.
2. Define $\mathbf{x} = \begin{bmatrix} T_w \\ T_a \end{bmatrix}$ and $\mathbf{u} = \begin{bmatrix} T_{out} \\ T_{adj} \\ \dot{Q}_{hvac} \end{bmatrix}$.
3. Express the system in the form $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$.
4. Calculate the numerical values of $\mathbf{A}$ and $\mathbf{B}$.
5. Find the eigenvalues of $\mathbf{A}$ and the corresponding time constants.
6. If we measure only T_{air}, what is $\mathbf{C}$?

Python verification:

```
import numpy as np

# Parameters (convert kJ to J)
R1, R2, R3 = 0.05, 0.02, 0.1  # K/W
Cw, Ca = 800e3, 100e3  # J/K

# A matrix
A = np.array([
    [-(1/Cw)*(1/R1 + 1/R2), (1/Cw)*(1/R2)],
    [(1/Ca)*(1/R2), -(1/Ca)*(1/R2 + 1/R3)]
])

# Eigenvalues and time constants
eigenvalues = np.linalg.eigvals(A)
time_constants = -1/eigenvalues
print(f"Eigenvalues: {eigenvalues}")
print(f"Time constants: {time_constants/3600} hours")
```

```
Eigenvalues: [-3.24397649e-05 -6.55060235e-04]
Time constants: [8.56287889 0.42404921] hours
```

Expected results:

- Two time constants: one fast, air-dominated, and one slow, wall-dominated.

11.10 Putting It All Together

11.10.1 From Physics to Control

Let's step back and appreciate the journey we've taken:

```
Fourier Heat Equation (PDE)
    ↓ Spatial discretization
Thermal Network (System of ODEs)
    ↓ Matrix formulation
State-Space Model (  = Ax + Bu)
    ↓ Ready for...
Control Design, Simulation, Identification
```

1. Physics → PDE: We started with the fundamental law of heat conduction (Fourier's equation), which describes how temperature varies continuously in space and time.

2. PDE → Network: By discretizing space—either uniformly or at physical boundaries—we converted the infinite-dimensional PDE into a finite-dimensional system of ODEs. The thermal network representation makes the physics intuitive: resistances impede heat flow, capacitances store heat.

3. Network → State-Space: By writing the nodal equations in matrix form, we arrived at the standard state-space representation used throughout control theory. This unlocks decades of tools for analysis and design.

This progression—from first-principles physics to control-ready models—is fundamental not just to buildings, but to most engineered systems. The skills you've learned here apply to thermal management in electronics, chemical process control, vehicle cabin climate, and beyond.

11.10.2 Limitations and Extensions

Our models make simplifying assumptions. It's important to understand when they apply and when more sophisticated approaches are needed:

Linearity: We assumed constant R and C, independent of temperature. This is excellent for typical building temperature ranges ($±20^ $C from nominal) but breaks down for:

- High-temperature processes
- Phase-change materials
- Systems with strong radiation coupling

Single-zone: We modeled single rooms. Real buildings have multiple zones with:

- Inter-zone heat transfer
- Shared HVAC systems
- Varying occupancy patterns

No humidity: We focused on sensible heat (temperature). Latent heat (humidity) matters for:

- Comfort (humid air feels warmer)
- Cooling loads (dehumidification is energy-intensive)
- Condensation risk

Known parameters: We assumed R and C are known. In practice:

- Material properties vary
- Construction details are uncertain
- Parameter identification from data is often needed (a topic for later in this course)

Neglected dynamics: We ignored:

- HVAC system dynamics (heating/cooling isn't instantaneous)
- Sensor and actuator dynamics
- Air mixing and stratification

11.10.3 Preview of What's Next

In the following lectures, we will use these thermal models to:

- Understand occupant thermal comfort (what temperatures should we target?)
- Design and analyze HVAC control strategies
- Apply model predictive control to optimize energy use while maintaining comfort

11.11 Additional Resources

11.11.1 Primary Reference

- Chapter 8 of Reddy (as noted in syllabus)

11.11.2 Supplementary Reading

- Lecture 5 notes (prerequisite steady-state material)
- Any introductory control systems textbook for state-space fundamentals

11.11.3 Online Resources

11.11.3.1 Thermal Modeling

- Engineering ToolBox - Thermal Diffusivity - Material property data
- Thermtest - Heat Penetration Calculator - Online calculator for penetration depth

11.11.3.2 State-Space and Control

- MIT OpenCourseWare - Introduction to State-Space - Free course materials
- Python Control Systems Library - Tools for control system analysis in Python

11.11.3.3 Building Simulation

- EnergyPlus - Detailed building energy simulation (uses thermal networks internally)
- Modelica Buildings Library - Component-based thermal modeling

12 Occupant Thermal Comfort

12.1 Lecture Overview

Learning Objectives

By the end of this module, students will be able to:

- Recreate the basic equation for the thermal balance of a human body, which gives rise to expressions about heat dissipation to the environment
- Understand latent heat (and its distinction from sensible heat, which is what we've covered so far), and Fick's law for moisture transfer (relating it to Fourier's law for conduction)
- Be able to calculate shape factors of simple symmetric geometries, and understand their relevance in radiative heat transfer
- Derive the Mean Radiant Temperature and its approximation when surfaces have similar temperature
- Derive the concept of "Operative Temperature" by starting with convective and radiative heat transfer equations between the outer clothing and the environment
- Be familiar with clothing insulation values (and their units of *clo*)
- Calculate total conductive plus radiative sensible heat loss for a person if given their metabolic rate and clothing insulation (as well as the operative temperature)
- Use the Predictive Mean Vote method for calculating a thermal sensation index
- Reason about comfort levels using ASHRAE's comfort chart relating operative temperature and humidity ratio

Topics Covered

- Human body thermal balance equations
- Heat generation through metabolism
- Heat dissipation mechanisms: conduction, convection, radiation, and evaporation
- Sensible heat vs. latent heat
- Fick's law for moisture transfer
- Shape factors for radiative heat transfer
- Mean Radiant Temperature (MRT) and its calculation
- Operative Temperature concept and derivation

- Clothing insulation and the *clo* unit
- Total sensible heat loss calculations
- Predictive Mean Vote (PMV) thermal sensation index
- ASHRAE comfort charts and their application

Project Milestones

Understanding occupant thermal comfort is critical for designing autonomous HVAC control systems that can:

- Maintain appropriate comfort levels for building occupants
- Balance energy efficiency with occupant satisfaction
- Adapt to different occupancy patterns and preferences
- Account for seasonal variations in clothing insulation
- Respond to both temperature and humidity conditions

This module provides the theoretical foundation for incorporating comfort models into your final project's control strategies.

12.2 From Building Thermodynamics to Human Thermodynamics

In Lectures 5 and 6, we developed an understanding of how buildings exchange heat with the external environment. We studied conduction through walls, convection at surfaces, radiation between surfaces, and thermal network models that describe how building temperatures evolve over time. However, there's a crucial piece missing from this picture: the people inside the buildings.

After all, an empty building does not need to be heated or cooled nearly as much as an occupied one. Buildings are not designed and built merely to shield us from the external environment—they exist to improve the quality of life (or at least the productivity, in the case of commercial buildings) of their occupants. From this perspective, maintaining a proper temperature is but one of many aspects required to achieve this greater objective. The study of the quality of the indoor environment—including acoustic, visual, and thermal aspects and their objective and subjective impact on occupants—is generally referred to as **Indoor Environment Quality (IEQ)**, which includes sub-fields such as Indoor Air Quality (IAQ) that deal with specific aspects of the environment. That said, we will only be discussing a small sliver of the full IEQ field.

In particular, in this lecture, we shift our focus from building thermodynamics to **human thermodynamics**. Just as buildings must balance heat gains and losses to maintain desired indoor temperatures, our bodies must balance the heat they generate (as a byproduct of

metabolizing the chemical energy we ingest in food) by dissipating excess heat to the environment to avoid overheating. This balance is the basis for human thermal comfort.

The good news is that the same fundamental heat transfer modes we studied in previous lectures—conduction, convection, and radiation—still apply. The geometry, materials, and boundary conditions are different (we're now dealing with human skin and clothing rather than concrete walls and windows), but the underlying physics remains the same. The main new element we'll introduce is **latent heat transfer** associated with moisture evaporation, since water mediates a significant portion of the heat exchange between our bodies and the environment. This mechanism, which we haven't covered in detail before, plays a crucial role in thermal comfort.

12.2.1 Why Thermal Comfort Matters for Building Control

Understanding occupant thermal comfort is essential for designing effective autonomous building control systems. The primary goal of HVAC systems is to maintain acceptable thermal comfort for occupants while doing so as efficiently as possible. This creates an inherent tension between energy consumption and occupant satisfaction.

Consider these facts:

- **Thermal complaints dominate building issues**: Studies consistently show that thermal comfort is the most common source of occupant complaints in commercial buildings, far exceeding concerns about air quality, lighting, or acoustics. In a typical office building, 30-50% of occupants may be dissatisfied with their thermal environment at any given time.
- **Impact on productivity**: Research has demonstrated that thermal discomfort significantly affects cognitive performance and productivity. Studies have shown productivity decreases of 5-10% when temperatures deviate from the comfort zone by just 2-3°C. For knowledge workers in office buildings, the cost of this lost productivity can far exceed the building's annual energy costs.
- **Energy implications are enormous**: Space heating and cooling account for approximately 40% of total energy consumption in commercial buildings and over 50% in residential buildings. Even small improvements in how we manage thermal comfort—such as expanding acceptable temperature ranges by 1-2°C or implementing occupancy-based control—can yield energy savings of 10-30%.
- **Comfort drives setpoint battles**: In the absence of sophisticated control, occupants often override thermostats, open windows in air-conditioned buildings, or bring in space heaters—all behaviors that waste energy and indicate failed comfort management. Understanding the physics and perception of thermal comfort allows us to design smarter control strategies that maintain satisfaction while minimizing these inefficiencies.

> ⚠ Fact or hallucination?
>
> Find reliable sources to confirm or deny the numbers claimed in the previous bullet point list.

For autonomous building systems, this means we need more than just temperature sensors and simple on-off control. We need models that predict how occupants will perceive their thermal environment based on multiple factors: air temperature, radiant temperature from surrounding surfaces, humidity, air movement, their activity level, and their clothing. This lecture provides the theoretical foundation for building those models.

12.3 The Human Body as a Thermal System

Like the buildings we've been studying, the human body is a thermal system that must carefully manage heat flows to maintain appropriate temperatures. However, unlike buildings where we can tolerate a relatively wide range of indoor temperatures (say, 15-30°C), the human body must maintain a relatively constant **core temperature** of approximately **37°C (98.6°F)** to function properly. This tight requirement must be met despite widely varying environmental conditions (outdoor temperatures from -40°C to +50°C) and activity levels (from sleeping to intense exercise).

To maintain this critical core temperature, our bodies employ several sophisticated **thermoregulation mechanisms**:

- **Vasodilation and vasoconstriction**: The body can increase or decrease blood flow near the skin surface by dilating or constricting blood vessels. When hot, increased blood flow to the skin enhances heat dissipation to the environment. When cold, reduced skin blood flow conserves heat for the core organs.
- **Sweating**: When the body needs to dissipate large amounts of heat (during exercise or in hot environments), sweat glands secrete moisture onto the skin surface. As this moisture evaporates, it carries away significant heat through the latent heat of vaporization—an extremely effective cooling mechanism.
- **Shivering**: When cold, involuntary muscle contractions (shivering) generate additional metabolic heat to help maintain core temperature.
- **Behavioral adaptations**: Beyond physiological responses, we instinctively adjust our behavior—adding or removing clothing, seeking shade or sun, adjusting posture to change surface area exposure, or simply moving to a more comfortable location.

However, these thermoregulation mechanisms have **limited capacity**. If environmental conditions or activity levels push the body beyond its ability to maintain thermal balance, serious health consequences follow:

- **Hypothermia** occurs when core temperature drops below ~35°C, impairing physical and cognitive function and potentially leading to death
- **Heat exhaustion** and **heat stroke** occur when core temperature rises above ~40°C, causing organ damage and potentially death

The fact that our thermoregulation range is relatively narrow compared to the environments we inhabit explains why humans have developed technologies—clothing, shelter, heating and cooling systems—to extend our effective operating range.

At a high level, the thermal balancing act performed by our bodies can be understood through a simple energy balance: the total rate at which our bodies produce energy (in the form of heat and mechanical work) must equal the total rate at which we dissipate heat to the environment. When these rates are in balance and our core temperature is maintained at ~37°C, we experience **thermal comfort**. When they're out of balance, we feel too hot or too cold, and our bodies must deploy their thermoregulation mechanisms to restore equilibrium—or signal us to take behavioral action.

12.3.1 Metabolic Heat Generation

The energy balance we just described starts with the body's energy production, which comes from **metabolism**—the process of converting chemical energy stored in food into forms the body can use. Through **cellular respiration**, our cells combine glucose and oxygen to produce ATP (adenosine triphosphate), the energy currency of the cell.

This energy is used for two purposes:

1. **Mechanical work** ($\dot{W}$): Moving muscles, pumping blood, breathing, etc.
2. **Heat** ($\dot{Q}$): All metabolic processes ultimately generate heat, either directly or as a byproduct of work (due to inefficiencies)

For most typical indoor activities, the mechanical work component is relatively small compared to the total metabolic rate. For example, when sitting at a desk, virtually all metabolic energy becomes heat. Even during walking, only about 20-25% of the metabolic energy goes into mechanical work; the rest becomes heat.

The total energy production rate is called the **metabolic rate** and is denoted $\dot{M}$. In the field of thermal comfort, metabolic rate is commonly expressed using the **met** unit:

$$\boxed{1 \text{ met} = 58.2 \text{ W/m}^2}$$

Notice that this is expressed in units of power per unit area. The area refers to the **body surface area** (A_{sk}, where "sk" stands for skin). This normalization by body surface area makes sense because larger people generally have higher absolute metabolic rates but similar rates per unit surface area for the same activity.

The typical body surface area for an adult is approximately **1.8 m²** (though this varies with height and weight and can be estimated using the DuBois formula). Thus, 1 met corresponds to an absolute metabolic rate of:

$$\dot{M}_{absolute} = 1 \text{ met} \times 58.2 \text{ W/m}^2 \times 1.8 \text{ m}^2 \approx 105 \text{ W}$$

This value (1 met) was chosen to represent the metabolic rate of a seated, resting adult in thermal comfort—essentially the baseline human energy consumption rate.

The total power generated by our bodies, which must equal the total rate at which we dissipate energy to the environment (in steady state), is:

$$\dot{M} \cdot A_{sk} = \dot{Q} + \dot{W}$$

Or, rearranging to emphasize the heat that must be dissipated:

$$\dot{Q} = \dot{M} \cdot A_{sk} - \dot{W}$$

For most indoor activities where $\dot{W}$ is negligible, we can approximate:

$$\dot{Q} \approx \dot{M} \cdot A_{sk}$$

This heat $\dot{Q}$ is what must be transferred to the environment through the heat dissipation mechanisms we'll discuss shortly.

12.3.1.1 Typical Metabolic Rates

Metabolic rates vary significantly depending on the activity level. The following table, based on ASHRAE Standard 55-2013, shows typical values for common activities:

Activity	Metabolic Rate (met)	Metabolic Rate (W/m²)	Absolute Rate (W)*
Sleeping	0.7	40	72
Reclining	0.8	46	83
Seated, quiet	1.0	58	105
Standing, relaxed	1.2	70	126
Walking, 0.9 m/s (2 mph)	2.0	115	207

Activity	Metabolic Rate (met)	Metabolic Rate (W/m^2)	Absolute Rate (W)*
Office work, typing	1.1	64	115
Cooking	1.6-2.0	95-115	171-207
House cleaning	2.0-3.4	115-200	207-360
Dancing	2.4-4.4	140-255	252-459
Heavy exercise	4.0-6.0	230-350	414-630

*Assuming body surface area $A_{sk} = 1.8$ m^2

Key observations:

- Metabolic rate can vary by nearly an order of magnitude between sleeping and vigorous exercise
- The 1 met baseline (seated, quiet) is close to the lowest sustainable long-term metabolic rate for awake adults
- For HVAC control purposes, typical office occupants can generally be assumed to be at 1.0-1.2 met
- Residential spaces may see wider variation depending on activities (cooking, cleaning, exercising)

Understanding these variations is crucial for comfort modeling: a person exercising at 4 met generates roughly 400 W of heat that must be dissipated, compared to only 100 W when seated. The same environmental conditions (air temperature, humidity) that feel comfortable for a sedentary occupant may feel oppressively hot for someone who is physically active.

12.3.2 Heat Dissipation Mechanisms

The heat generated by metabolism must be dissipated to the environment to maintain thermal balance. The human body uses four main heat transfer mechanisms, three of which we've already encountered in our study of building thermodynamics:

1. **Conduction**: Direct heat transfer through contact (e.g., feet on floor, body on chair)
2. **Convection**: Heat transfer to surrounding air via fluid motion
3. **Radiation**: Electromagnetic heat exchange with surrounding surfaces
4. **Evaporation**: Heat removal through moisture vaporization (new mechanism!)

For most indoor scenarios, **conduction is relatively minor** (typically <5% of total heat loss) unless sitting on a cold surface or submerged in water. The dominant heat transfer paths are **convection, radiation, and evaporation**.

The key insight for modeling human thermal comfort is that these heat transfer mechanisms can be represented as **two parallel thermal network paths**, analogous to the electrical circuits we used for building thermal analysis:

Path 1: Sensible Heat Transfer ($\dot{Q}_{sen}$)

This path handles heat transfer that changes temperature (sensible heat):

$$\dot{Q}_{sen} = \dot{Q}_{conv} + \dot{Q}_{rad}$$

The thermal network for sensible heat looks like this:

```
Skin (T_sk)
    |
    R_cl (clothing thermal resistance)
    |
Clothing surface (T_cl)
    |
    +------ R_conv -----> Air (T_a)
    |
    +------ R_rad ------> Surrounding surfaces (T_mrt)
```

Heat flows from the skin surface (at temperature T_{sk}) through the clothing insulation (thermal resistance R_{cl}) to the outer clothing surface (at temperature T_{cl}). From there, it splits into two parallel paths:

- **Convective path**: Heat flows to the surrounding air (at temperature T_a) through convective resistance
- **Radiative path**: Heat radiates to surrounding surfaces, which we characterize using a single equivalent temperature called the **Mean Radiant Temperature** (T_{mrt})

Path 2: Latent Heat Transfer ($\dot{Q}_{lat}$)

This path handles heat transfer associated with moisture evaporation (latent heat):

$$\dot{Q}_{lat} = \dot{Q}_{evap}$$

The thermal network for latent heat is similar in structure but involves moisture transfer:

```
Skin surface (W_sk)
    |
    R_e,cl (clothing moisture resistance)
    |
Clothing surface (W_cl)
    |
    R_e,evap (evaporative resistance)
    |
Indoor air (W_a)
```

Moisture evaporates from the skin surface (at vapor concentration W_{sk}), diffuses through the clothing (moisture vapor resistance $R_{e,cl}$), and then evaporates from the clothing surface into the surrounding air (at vapor concentration W_a). Each kilogram of water that evaporates carries away approximately **2,430 kJ of heat** (at body temperature)—an extremely efficient cooling mechanism.

Putting the networks together:

These two networks operate **in parallel**—sensible heat and latent heat are transferred simultaneously and independently. The total heat dissipated is:

$$\dot{Q}_{total} = \dot{Q}_{sen} + \dot{Q}_{lat}$$

Which, again, must equal the heat generated by metabolism (minus any mechanical work):

$$\dot{Q}_{total} = \dot{M} \cdot A_{sk} - \dot{W}$$

Important notes:

- The **clothing layer** appears in both networks, acting as thermal resistance for sensible heat and as moisture vapor resistance for latent heat. Different clothing materials have different properties for each path.
- The **Mean Radiant Temperature** (T_{mrt}) is a clever simplification: rather than tracking radiation to every surface in the room (walls, ceiling, floor, windows), we define a single equivalent temperature that gives the same net radiative heat exchange.
- The relative importance of each path depends on environmental conditions and activity level. In cool, dry environments with sedentary occupants, sensible heat dominates. In warm, humid environments or during exercise, evaporative cooling becomes critical.

We'll develop detailed models for each of these components in the sections that follow. For now, keep this parallel network structure in mind—it's the conceptual foundation for all quantitative thermal comfort calculations.

12.3.3 The Thermal Balance Equation

Now we can write down the complete thermal balance equation for the human body. At its core, this equation expresses conservation of energy: the rate at which energy is produced must equal the rate at which it's dissipated (in steady state).

Starting from our earlier observation that the body produces energy at rate $\dot{M} \cdot A_{sk}$, which is split between mechanical work $\dot{W}$ and heat $\dot{Q}$, we can expand the heat dissipation term into its constituent mechanisms:

$$\boxed{\dot{M} \cdot A_{sk} - \dot{W} = \dot{Q}_{conv} + \dot{Q}_{rad} + \dot{Q}_{evap} + \dot{Q}_{cond} + \dot{Q}_{res,sen} + \dot{Q}_{res,lat} + \Delta\dot{Q}_{stored}}$$

where:

- $\dot{M}$ = metabolic rate per unit body surface area (W/m² or met)
- A_{sk} = body surface area (typically ~1.8 m² for adults)
- $\dot{W}$ = rate of mechanical work done by the body (W)
- $\dot{Q}_{conv}$ = convective heat loss rate through skin (W)
- $\dot{Q}_{rad}$ = radiative heat loss rate through skin (W)
- $\dot{Q}_{evap}$ = evaporative heat loss rate through skin (W)
- $\dot{Q}_{cond}$ = conductive heat loss rate (W, usually very small)
- $\dot{Q}_{res,sen}$ = sensible (temperature-based) heat loss through respiration (W)
- $\dot{Q}_{res,lat}$ = latent (moisture-based) heat loss through respiration (W)
- $\Delta\dot{Q}_{stored}$ = rate of change of heat stored in the body (W)

Typical magnitudes and simplifications:

For most thermal comfort analyses, we can make several simplifying assumptions:

1. **Steady-state assumption**: For thermal comfort (as opposed to thermal stress), we assume the body is in steady state, meaning core temperature isn't changing significantly. Therefore: $\Delta\dot{Q}_{stored} \approx 0$
2. **Negligible work**: For typical indoor activities (sitting, standing, light office work), mechanical work is negligible compared to metabolic rate: $\dot{W} \ll \dot{M} \cdot A_{sk}$, so $\dot{W} \approx 0$
3. **Small conduction**: Unless sitting on a very cold or hot surface, or submerged in water, conductive heat transfer is typically <5% of total: $\dot{Q}_{cond} \approx 0$
4. **Small respiration losses**: Respiration accounts for breathing in cool, dry air and exhaling warm, moist air. While non-negligible, it's typically <10% of total heat loss for sedentary activities: $\dot{Q}_{res,sen} + \dot{Q}_{res,lat} \approx 0.1 \times (\dot{Q}_{conv} + \dot{Q}_{rad} + \dot{Q}_{evap})$

With these simplifications, the practical thermal balance equation becomes:

$$\boxed{\dot{M} \cdot A_{sk} \approx \dot{Q}_{conv} + \dot{Q}_{rad} + \dot{Q}_{evap}}$$

Or, using our earlier notation grouping sensible and latent heat:

$$\boxed{\dot{M} \cdot A_{sk} \approx \dot{Q}_{sen} + \dot{Q}_{lat} = (\dot{Q}_{conv} + \dot{Q}_{rad}) + \dot{Q}_{evap}}$$

This is the fundamental equation we'll use for thermal comfort analysis. The left side is determined by the occupant's activity level. The right side depends on environmental conditions (air temperature, mean radiant temperature, humidity, air velocity) and personal factors (clothing insulation). **Thermal comfort occurs when this equation is balanced at the appropriate skin temperature** (typically 33-34°C for comfort).

What happens when it's not balanced?

- If $\dot{M} \cdot A_{sk} > \dot{Q}_{sen} + \dot{Q}_{lat}$: The body is generating more heat than it can dissipate. Core temperature rises, triggering thermoregulation (sweating, vasodilation, behavioral changes). The occupant feels **too warm**.
- If $\dot{M} \cdot A_{sk} < \dot{Q}_{sen} + \dot{Q}_{lat}$: The environment is extracting more heat than the body is generating. Core temperature falls, triggering thermoregulation (shivering, vasoconstriction, behavioral changes). The occupant feels **too cold**.

The goal of HVAC control, from a thermal comfort perspective, is to maintain environmental conditions such that this balance can be achieved at comfortable skin and core temperatures.

A note on what's to come:

This equation introduces several concepts we haven't fully developed yet—particularly **latent heat** and **evaporative cooling**. Before we can make this equation quantitative and calculate actual comfort levels, we need to take a detour to understand these mechanisms in detail. That's what we'll do in the next section.

12.4 Sensible Heat vs. Latent Heat

Up to this point in the course, we've focused exclusively on **sensible heat transfer**—heat transfer that you can "sense" with a thermometer because it changes the temperature of materials. Conduction through walls, convection to air, and radiation between surfaces all fall into this category. However, the thermal balance equation we just derived includes an evaporative heat loss term ($\dot{Q}_{evap}$) that represents a fundamentally different type of heat transfer: **latent heat**.

Sensible heat is heat transfer associated with a change in temperature:

$$Q_{sensible} = m \cdot c \cdot \Delta T$$

When you add sensible heat to a substance, its temperature rises. When you remove it, temperature falls. This is what we've been studying throughout the course.

Latent heat is heat transfer associated with a phase change at constant temperature:

$$Q_{latent} = m \cdot h_{fg}$$

where h_{fg} is the latent heat of vaporization (or fusion). When water evaporates, it absorbs a large amount of energy to break the molecular bonds that hold it in liquid form, but the temperature doesn't change during this process—the water and water vapor both remain at the same temperature during evaporation. This absorbed energy is the latent heat.

Why evaporation is such an effective cooling mechanism:

The latent heat of vaporization for water is **remarkably high**: approximately **2,430 kJ/kg** at human body temperature (~35°C). To put this in perspective:

- Evaporating 1 kg of water absorbs 2,430 kJ of heat
- Cooling 1 kg of water by 100°C (from boiling to near-freezing) releases only $1 \times 4.18 \times 100 = 418$ kJ
- **Evaporation is nearly 6 times more effective per kilogram** than sensible cooling over a 100°C range

For the human body, this means:

- Evaporating just **0.1 kg (100 mL)** of sweat removes **243 kJ** of heat
- For a sedentary person generating ~100 W of metabolic heat, this much evaporation provides **40 minutes** of cooling
- During heavy exercise (~400 W), the body can produce 1-2 liters of sweat per hour, providing cooling power of 650-1,300 W

This is why sweating is the body's primary defense against overheating during exercise or in hot environments.

Why moisture migration matters beyond thermal comfort:

While our focus is on thermal comfort, moisture transfer through building materials and clothing has several other important implications:

1. **Structural integrity**: Moisture accumulation in building envelopes can lead to mold growth, wood rot, corrosion of metal components, and freeze-thaw damage. Understanding moisture migration paths is critical for building durability.

2. **Insulation effectiveness**: Many insulation materials (fiberglass, mineral wool) lose significant thermal resistance when wet. Water has much higher thermal conductivity (~0.6 W/m · K) than air (~0.026 W/m · K), so moisture in insulation creates thermal bridges.
3. **Clothing comfort and performance**: Moisture trapped in clothing feels clammy and reduces the effectiveness of clothing insulation. High-performance athletic fabrics are designed with specific moisture vapor permeability to allow sweat to escape while blocking wind and rain.
4. **Indoor air quality**: Excess indoor humidity promotes mold growth and dust mite proliferation, both of which affect occupant health. Too little humidity causes dry skin, irritated respiratory passages, and static electricity.

Understanding the parallel between heat transfer (governed by Fourier's law) and moisture transfer (governed by Fick's law) allows us to apply similar analytical techniques to both problems. Let's develop that analogy now.

12.4.1 Latent Heat Transfer: Evaporation and Moisture

The human body has two primary pathways for evaporative heat loss: **skin evaporation** and **respiratory evaporation**. Both involve water changing phase from liquid to vapor, each time carrying away the latent heat of vaporization.

The latent heat of vaporization (h_{fg}):

At human body temperature (~35°C / 95°F), the latent heat of vaporization of water is:

$$h_{fg} \approx 2{,}430 \text{ kJ/kg} = 2{,}430{,}000 \text{ J/kg}$$

This value varies slightly with temperature (it's 2,257 kJ/kg at 100°C and 2,501 kJ/kg at 0°C), but 2,430 kJ/kg is appropriate for skin temperature calculations.

Skin moisture evaporation:

The skin continuously loses water through two distinct mechanisms:

1. **Insensible perspiration**: This is passive water diffusion through the skin that occurs even when we're not consciously sweating. The epidermis (outer skin layer) is not perfectly waterproof, so water vapor continually diffuses from the moist inner layers through the skin to the drier environment. This accounts for roughly **20-30 W** of continuous cooling in typical indoor conditions (about 20-30% of basal metabolic rate).

2. **Sensible perspiration (sweating)**: When the body needs additional cooling beyond what insensible perspiration provides, eccrine sweat glands actively secrete liquid water onto the skin surface. This sweat then evaporates, providing substantial cooling. The rate is controlled by the thermoregulatory system:

    - Minimal: 0 (no active sweating in cool conditions)
    - Light sweating: 100-200 W of cooling
    - Moderate sweating: 200-400 W
    - Heavy sweating: 400-600 W
    - Maximum sustainable: ~600-800 W (corresponding to 0.9-1.2 L/hour sweat production)

The total evaporative heat loss from skin is:

$$\dot{Q}_{evap,skin} = \dot{m}_{evap} \cdot h_{fg}$$

where $\dot{m}_{evap}$ is the rate of water evaporation (kg/s).

Respiratory moisture evaporation:

Each breath involves:

- Inhaling relatively cool, dry ambient air
- Warming it to body temperature in the airways (sensible heat loss)
- Saturating it with water vapor in the lungs (latent heat loss)
- Exhaling this warm, moist air

The respiratory latent heat loss can be estimated as:

$$\dot{Q}_{res,lat} = \dot{m}_{air} \cdot (W_{exhaled} - W_{inhaled}) \cdot h_{fg}$$

where W represents the humidity ratio (kg water vapor / kg dry air) and $\dot{m}_{air}$ is the breathing rate.

For typical sedentary conditions:

- Respiratory latent heat loss: ~10-15 W
- Respiratory sensible heat loss: ~5-8 W
- Combined: ~15-23 W (about 15-20% of basal metabolic rate)

During heavy exercise with increased breathing rate, respiratory heat loss increases but remains a smaller fraction of total heat loss compared to skin evaporation.

Total evaporative cooling:

Combining skin and respiratory evaporation:

$$\dot{Q}_{lat} = \dot{Q}_{evap,skin} + \dot{Q}_{res,lat}$$

The body can modulate only the sweating component; insensible perspiration and respiratory evaporation are relatively constant for given environmental conditions.

Environmental factors affecting evaporation:

The effectiveness of evaporative cooling depends critically on the moisture gradient between the skin/clothing surface and the ambient air:

- **Low humidity**: Large moisture gradient → rapid evaporation → effective cooling
- **High humidity**: Small moisture gradient → slow evaporation → reduced cooling

This is why 35°C (95°F) with low humidity can feel more comfortable than 30°C (86°F) with high humidity—evaporative cooling is much more effective in the dry conditions.

In the extreme case of 100% relative humidity at skin temperature, evaporation effectively stops, and the body loses its most powerful cooling mechanism. This is why very humid conditions combined with high temperatures can be deadly—the body cannot dissipate metabolic heat and core temperature rises uncontrollably.

12.4.2 Fick's Law for Moisture Transfer

Just as Fourier's law governs the diffusion of heat through materials, **Fick's law** governs the diffusion of mass (in our case, water vapor) through materials. The mathematical form is strikingly similar, which allows us to use the same analytical framework we developed for thermal networks.

Fick's First Law states that the mass flux (mass transfer rate per unit area) is proportional to the concentration gradient:

$$\boxed{\dot{m}'' = -D\frac{dC}{dx}}$$

where:

- $\dot{m}''$ = mass flux (kg/s · m²) - mass transfer rate per unit area
- D = diffusion coefficient (m²/s) - analogous to thermal diffusivity $\alpha = k/(\rho c)$
- C = concentration (kg/m³) - mass of water vapor per unit volume
- dC/dx = concentration gradient (kg/m)

The negative sign indicates that mass flows from high concentration to low concentration, just as heat flows from high temperature to low temperature.

For a finite layer of thickness L with area A, integrating Fick's law gives:

$$\boxed{\dot{m} = -D \cdot A\frac{\Delta C}{L} = \frac{A \cdot \Delta C}{R_m}}$$

where $R_m = L/D$ is the **moisture diffusion resistance** (analogous to thermal resistance $R = L/k$).

Application to human thermal comfort:

For moisture transfer from skin through clothing to the environment, we need to track the moisture concentration (or more commonly, the water vapor pressure or humidity ratio) at each layer:

1. **At the skin surface** (C_{sk} or W_{sk}): Nearly saturated due to sweat and insensible perspiration
2. **At the clothing surface** (C_{cl} or W_{cl}): Lower than skin but higher than ambient
3. **In the ambient air** (C_a or W_a): Determined by indoor humidity conditions

The moisture flow rate through the clothing is:

$$\dot{m}_{vapor} = \frac{A_{cl}(W_{sk} - W_{cl})}{R_{e,cl}}$$

where:

- W = humidity ratio (kg water vapor / kg dry air) - more common than concentration in HVAC
- $R_{e,cl}$ = clothing moisture vapor resistance ($m^2 \cdot Pa/W$ or clo equivalent for moisture)
- A_{cl} = clothed surface area

Then from the clothing surface to ambient air:

$$\dot{m}_{evap} = h_e \cdot A_{cl}(W_{cl} - W_a)$$

where h_e is the evaporative heat transfer coefficient, analogous to the convective heat transfer coefficient h_c.

Converting mass flow to heat flow:

Once we know the evaporation rate $\dot{m}_{evap}$, we convert it to latent heat loss:

$$\dot{Q}_{lat} = \dot{m}_{evap} \cdot h_{fg}$$

This is how moisture diffusion (governed by Fick's law) connects to the thermal balance equation (energy conservation).

Practical implications:

The moisture vapor resistance of clothing varies dramatically by material:

- **Vapor-permeable fabrics** (cotton, wool, technical athletic wear): Low $R_{e,cl}$, allows moisture to escape
- **Vapor-impermeable layers** (rubber, plastic, waxed fabrics): High $R_{e,cl}$, traps moisture

When moisture cannot escape through clothing, it accumulates as liquid on the skin surface. This feels uncomfortable (clammy) and reduces the effective thermal insulation of the clothing (wet clothing conducts heat much better than dry clothing).

The parallel structure between Fourier's law and Fick's law means we can draw **thermal network diagrams** for moisture transfer that look exactly like those for heat transfer, with concentrations (or vapor pressures) playing the role of temperatures, and moisture resistances playing the role of thermal resistances.

Convince yourself: Fourier's Law vs. Fick's Law

The parallel between heat diffusion and mass diffusion is useful. This table shows the complete analogy:

Aspect	**Heat Transfer (Fourier)**	**Mass Transfer (Fick)**
Driving potential	Temperature difference ΔT	Concentration difference ΔC (or vapor pressure Δp_v)
Flux equation	$\dot{q}'' = -k\frac{dT}{dx}$	$\dot{m}'' = -D\frac{dC}{dx}$
Material property	Thermal conductivity k (W/m $\cdot$ K)	Diffusion coefficient D (m^2/s)
Integrated form	$\dot{Q} = \frac{k \cdot A}{L}\Delta T$	$\dot{m} = \frac{D \cdot A}{L}\Delta C$
Resistance	$R_{thermal} = \frac{L}{k \cdot A}$ (K/W)	$R_{mass} = \frac{L}{D \cdot A}$ (kg/m^3 per kg/s)
Conductance	$U \cdot A = \frac{k \cdot A}{L}$ (W/K)	$G_m \cdot A = \frac{D \cdot A}{L}$ (kg/s per kg/m^3)
Series resistances	$R_{total} = R_1 + R_2 + R_3$	$R_{m,total} = R_{m,1} + R_{m,2} + R_{m,3}$

Parallel resistances	$\frac{1}{R_{total}} = \frac{1}{R_1} + \frac{1}{R_2}$	$\frac{1}{R_{m,total}} = \frac{1}{R_{m,1}} + \frac{1}{R_{m,2}}$

Network diagram analogy:
For heat transfer through clothing:

```
T_sk ----[R_cl]---- T_cl ----[R_conv]---- T_a
```

For moisture transfer through clothing:

```
W_sk ----[R_e,cl]---- W_cl ----[R_e,evap]---- W_a
```

Both use exactly the same circuit analysis techniques!
Key insight: Because the mathematics is identical, every analytical tool we developed for thermal networks (series/parallel combination, time constants for transient problems, etc.) applies equally to moisture transfer networks. This is why understanding building thermodynamics prepares us well for understanding moisture-related comfort issues.
Practical example: Clothing as resistance
A cotton t-shirt has:

- Thermal resistance: $R_{cl} \approx 0.09$ m$^2 \cdot$ K/W (about 0.6 clo)
- Moisture vapor resistance: $R_{e,cl} \approx 0.01$ m$^2 \cdot$ Pa/W

The ratio of these resistances determines whether the clothing "breathes" well. Athletic fabrics are engineered to have low moisture resistance while maintaining thermal insulation—allowing sweat to escape while blocking wind.
A rubber raincoat has:

- Thermal resistance: $R_{cl} \approx 0.08$ m$^2 \cdot$ K/W (similar thermal insulation)
- Moisture vapor resistance: $R_{e,cl} \approx 1000$ m$^2 \cdot$ Pa/W (essentially impermeable!)

This is why raincoats feel clammy during exercise—moisture cannot escape, so sweat accumulates on the skin.

12.5 Radiative Heat Transfer and Shape Factors

In Lecture 5, we introduced radiative heat transfer and the Stefan-Boltzmann law, which describes the total radiation emitted by a surface:

$$\dot{Q}_{rad,emit} = \epsilon \sigma A T^4$$

We also discussed radiation exchange between two surfaces. However, we largely sidestepped a critical geometric question: **of all the radiation leaving one surface, what fraction actually strikes another surface?**

For the human body in an enclosed room, this question becomes essential. A person radiates heat in all directions, but the room has multiple surfaces at different temperatures—walls, ceiling, floor, windows. Each surface intercepts a different fraction of the person's radiation, and each radiates back at a rate determined by its own temperature. To calculate the net radiative heat exchange accurately, we need to account for these geometric relationships.

This is where **shape factors** (also called **view factors** or **configuration factors**) come in. They quantify the geometric relationship between surfaces and allow us to calculate radiative exchange in complex environments without resorting to Monte Carlo ray tracing or other computationally expensive methods.

Understanding shape factors is crucial for thermal comfort because:

1. **Radiative asymmetry** is a major source of local discomfort (e.g., sitting near a cold window in winter)
2. **Mean Radiant Temperature** (MRT), a key parameter in thermal comfort models, is calculated using shape factors
3. **Radiant heating/cooling systems** (floor heating, radiant panels) rely on shape factor calculations for design
4. Even when air temperature is perfectly controlled, poor radiant conditions can make occupants uncomfortable

In this section, we'll develop the concept of shape factors and show how they enable us to calculate the **Mean Radiant Temperature**—a single number that characterizes the radiative environment experienced by a person.

12.5.1 What Are Shape Factors?

The **shape factor** (or view factor) F_{1-2} is defined as **the fraction of radiation leaving surface 1 that directly strikes surface 2**.

Mathematically, if surface 1 emits (or reflects) radiation uniformly in all directions (a "diffuse" surface), then:

$$F_{1-2} = \frac{\text{Radiation leaving surface 1 that strikes surface 2}}{\text{Total radiation leaving surface 1}}$$

Shape factors are purely **geometric properties**—they depend only on the size, shape, and relative position of the surfaces, not on their temperatures or surface properties (emissivity).

Key properties of shape factors:

1. **Range**: $0 \leq F_{1-2} \leq 1$

   - $F_{1-2} = 0$ means surface 1 "sees" none of surface 2 (no line of sight)
   - $F_{1-2} = 1$ means surface 1 "sees" only surface 2 (surface 2 completely surrounds surface 1)

2. **Summation rule**: For a surface in an enclosure, the sum of view factors to all surfaces (including itself, if concave) equals 1:

$$\sum_{j=1}^{n} F_{i-j} = 1$$

   This simply states that all radiation leaving surface i must go somewhere within the enclosure.

3. **Reciprocity relation**: The radiation exchange between two surfaces is symmetric when accounting for their areas:

$$\boxed{A_1 F_{1-2} = A_2 F_{2-1}}$$

   This powerful relation means if we know F_{1-2} and the areas, we can immediately calculate F_{2-1}.

Why shape factors matter for radiation calculations:

The net radiative heat transfer between two diffuse, gray surfaces at temperatures T_1 and T_2 can be written as:

$$\dot{Q}_{1\to 2} = A_1 F_{1-2} \sigma (T_1^4 - T_2^4)$$

(This is a simplified form assuming high emissivity; the full form includes emissivity factors.)

Without the shape factor F_{1-2}, we would vastly overestimate radiative exchange because we'd be assuming all radiation from surface 1 reaches surface 2, which is rarely true in real geometries.

For human thermal comfort:

A person standing in a room needs to exchange radiation with all surrounding surfaces—four walls, ceiling, and floor. Each surface has:

- A different temperature (window might be 10°C, radiator wall might be 25°C, other walls at 20°C)
- A different shape factor from the person (the floor might have $F_{person-floor} = 0.3$, while a distant wall might have $F_{person-wall} = 0.15$)

The challenge is to calculate the total net radiation from the person to this complex environment. Shape factors are the tool that makes this tractable.

12.5.2 Calculating Shape Factors for Simple Geometries

Exact analytical formulas for shape factors exist only for relatively simple geometries. For complex situations (like a person in a real room), numerical methods or empirical correlations are used. However, understanding a few key cases builds intuition.

Case 1: Small surface to a large enclosure

If a small surface (area A_1) is completely surrounded by a much larger surface (area $A_2 \gg A_1$), then:

$$F_{1-2} \approx 1 \quad \text{(small surface sees only the enclosure)}$$

Using reciprocity: $A_1 F_{1-2} = A_2 F_{2-1}$

$$F_{2-1} = \frac{A_1}{A_2} F_{1-2} \approx \frac{A_1}{A_2} \approx 0 \quad \text{(large surface barely sees the small one)}$$

Application to human comfort: A person (surface area ~1.8 m²) in a typical room (surface area ~100 m²) can be approximated this way. The person "sees" essentially the entire room ($F_{person-room} \approx 1$), while any individual room surface barely "sees" the person ($F_{surface-person} \approx 0.02$).

Case 2: Person standing in center of a rectangular room

For a standing person (modeled as a vertical cylinder) in the center of a room with floor, ceiling, and four walls, empirical studies give approximate shape factors:

Surface	Shape Factor $F_{person-surface}$	Typical Value
Floor	$\frac{A_{floor}}{A_{total}}$ weighted by projection	0.25-0.30
Ceiling	$\frac{A_{ceiling}}{A_{total}}$ weighted by projection	0.05-0.10
Walls (total)	Remaining fraction	0.60-0.70

The floor has a larger shape factor than the ceiling because the person is closer to the floor and has better "view" of it. For a standing person in a standard 2.7m ceiling room, the floor is much closer than the ceiling.

Case 3: Person sitting near a window

When a person sits close to a window or wall (distance $<$ 1m), that surface dominates the radiative exchange. Approximate shape factors might be:

Surface	Shape Factor	Notes
Nearby window	0.30-0.40	High due to proximity
Floor	0.20-0.25	Reduced compared to center position
Opposite wall	0.10-0.15	Partially blocked by person's position
Other surfaces	0.25-0.40	Remainder

This is why sitting near a cold window feels so uncomfortable—the high shape factor to the cold surface dominates radiative heat loss.

Analytical formula: Perpendicular rectangles

For completeness, here's one exact analytical case—two perpendicular rectangles sharing a common edge (like a person standing next to a wall):

For rectangles with dimensions such that they share an edge of length l, and have heights/widths h_1 and h_2:

$$F_{1-2} = \frac{1}{\pi h_1}\left[\arctan\left(\frac{h_2}{l}\right) - \text{complicated integral terms}\right]$$

In practice, we use **tabulated values** or **software tools** rather than evaluating these integrals by hand.

Truth or hallucination: Shape Factor Calculation

Can you spot any errors in this calculation?
Problem: A person (modeled as a 1.7m tall, 0.4m diameter cylinder, total surface area $A_p \approx 1.8$ m^2) stands in the center of a 4m $\times$ 5m $\times$ 2.7m room. Estimate the shape factors from the person to each surface.
Solution:
First, calculate room surface areas:

- Floor: $A_{floor} = 4 \times 5 = 20$ m^2
- Ceiling: $A_{ceiling} = 4 \times 5 = 20$ m^2
- Wall 1 (4m wide): $A_{w1} = 4 \times 2.7 = 10.8$ m^2
- Wall 2 (5m wide): $A_{w2} = 5 \times 2.7 = 13.5$ m^2
- Wall 3 (4m wide): $A_{w3} = 10.8$ m^2
- Wall 4 (5m wide): $A_{w4} = 13.5$ m^2
- Total room surface area: $A_{total} = 88.6$ m^2

Step 1: Use empirical correlations for a standing person
For a person centered in a room:

- $F_{p-floor} \approx 0.28$ (from empirical data for cylinder to floor)

- $F_{p-ceiling} \approx 0.08$ (ceiling is far from person at 1.7m height)
- $F_{p-walls} = 1 - F_{p-floor} - F_{p-ceiling} = 1 - 0.28 - 0.08 = 0.64$

Step 2: Distribute wall shape factor by area
Assume the four walls are "seen" in proportion to their areas:

$$F_{p-w1} = F_{p-walls} \times \frac{A_{w1}}{A_{w1} + A_{w2} + A_{w3} + A_{w4}} = 0.64 \times \frac{10.8}{48.6} = 0.142$$

$$F_{p-w2} = 0.64 \times \frac{13.5}{48.6} = 0.178$$

Similarly: $F_{p-w3} = 0.142$ and $F_{p-w4} = 0.178$
Step 3: Verify summation rule

$$F_{p-floor}+F_{p-ceiling}+F_{p-w1}+F_{p-w2}+F_{p-w3}+F_{p-w4} = 0.28+0.08+0.142+0.178+0.142+0.178 = 1.00 \quad \checkmark$$

Step 4: Use reciprocity to find shape factors from room surfaces to person

$$F_{floor-p} = \frac{A_p}{A_{floor}} F_{p-floor} = \frac{1.8}{20} \times 0.28 = 0.025$$

This confirms our earlier observation: room surfaces barely "see" the person (only 2.5% of floor radiation hits the person).

12.5.3 Application to Human Thermal Comfort

With shape factors in hand, we can now calculate the total radiative heat exchange between a person and their surrounding environment. This is critical because **radiation typically accounts for 40-50% of total heat loss** in comfortable indoor conditions (the rest being convection and evaporation).

Net radiative heat loss from a person:

The net radiative heat exchange between a person and surface i is:

$$\dot{Q}_{rad,i} = A_p F_{p-i} \epsilon_p \sigma (T_p^4 - T_i^4)$$

where:

- A_p = person's surface area (~1.8 m² for an adult, but this is actually the clothed area A_{cl})
- F_{p-i} = shape factor from person to surface i

- ϵ_p = effective emissivity of clothing/skin (~0.95 for typical fabrics and skin)
- σ = Stefan-Boltzmann constant (5.67×10^{-8} W/m$^2 \cdot$ K)
- T_p = surface temperature of clothing (typically 28-34°C depending on clothing and conditions)
- T_i = temperature of surface i

The **total** radiative heat loss is the sum over all surfaces:

$$\boxed{\dot{Q}_{rad,total} = A_p \epsilon_p \sigma \sum_{i=1}^{n} F_{p-i}(T_p^4 - T_i^4)}$$

Key insights from this equation:

1. **Shape factors weight the contribution of each surface**: A cold window with $F_{p-window} = 0.35$ has much more impact than a cold wall with $F_{p-wall} = 0.10$, even if both are at the same temperature.
2. **Temperature differences are raised to the fourth power**: This means cold surfaces have a disproportionate effect. A window at 5°C has much more impact than predicted by linear temperature differences.
3. **Position matters**. The view factor (or shape factor) is actually different depending on where the person is located. Imagine it this way: though it is true that we radiate heat to everything outside of us, what matters is their relative proportion in our field of view: we don't radiate as much to stars or deep space, compared to objects that are nearer.

Why "air temperature" isn't enough:

Consider two scenarios:

- **Scenario A**: $T_a = 22°C$, all surfaces at 22°C
- **Scenario B**: $T_a = 22°C$, window at 8°C, opposite wall at 28°C (average still 22°C)

The air temperature is identical, but occupant comfort is very different! In Scenario B, a person near the window will feel cold due to high radiative loss to the cold surface, while a person near the opposite wall will feel uncomfortably warm.

This is why **ASHRAE Standard 55** includes limits on radiant temperature asymmetry:

- Warm ceiling: Max 5°C warmer than other surfaces
- Cool wall (window): Max 10°C cooler than other surfaces
- Cool ceiling: Max 14°C cooler than other surfaces

Exceeding these limits causes local discomfort even if mean radiant temperature and air temperature are in the comfort zone.

Radiant heating/cooling systems:

Understanding shape factors is crucial for designing radiant systems:

- **Radiant floor heating**: High shape factor to floor (0.25-0.30) means floor temperature only needs to be 24-28°C to provide significant heating, allowing lower air temperatures and better comfort.
- **Radiant ceiling panels**: Lower shape factor to ceiling (0.05-0.10) means panels must be at higher/lower temperatures to achieve the same effect, with risk of violating asymmetry limits.
- **Radiant wall panels**: Moderate shape factors (0.15-0.20) offer a compromise, especially effective when positioned near occupant zones (near desks, seating areas).

Convince yourself: Why Shape Factors Matter

Quantitative example: The cold window problem
Let's calculate exactly how much extra heat loss a person experiences when sitting near a cold window compared to sitting in the room center, even though the air temperature is identical.
Setup:

- Room: 4m × 5m × 2.7m, air temperature $T_a = 22°C$
- Person: $A_{cl} = 1.8$ m², clothing surface temperature $T_{cl} = 30°C$, emissivity $\epsilon = 0.95$
- Window: 2m × 1.5m = 3 m², inner surface temperature $T_{window} = 8°C$ (cold winter day)
- Other surfaces: All at $T_i \approx 21°C$ (slightly below air temperature)

Position 1: Person in center of room
Shape factors (from earlier example):

- $F_{p-window} \approx 0.10$ (small fraction of view)
- $F_{p-other} \approx 0.90$ (rest of room)

Radiative heat loss to window:

$$\dot{Q}_{rad,window} = 1.8 \times 0.10 \times 0.95 \times 5.67 \times 10^{-8} \times [(30+273)^4 - (8+273)^4]$$

$$= 1.8 \times 0.10 \times 0.95 \times 5.67 \times 10^{-8} \times [8.468 \times 10^9 - 6.238 \times 10^9]$$

$$= 1.8 \times 0.10 \times 0.95 \times 5.67 \times 10^{-8} \times 2.230 \times 10^9 = 21.6 \text{ W}$$

Radiative heat loss to other surfaces (at 21°C):

$$\dot{Q}_{rad,other} = 1.8 \times 0.90 \times 0.95 \times 5.67 \times 10^{-8} \times [(303)^4 - (294)^4]$$

$$= 1.8 \times 0.90 \times 0.95 \times 5.67 \times 10^{-8} \times 7.51 \times 10^8 = 65.4 \text{ W}$$

Total radiative loss (center position): 21.6 + 65.4 = 87.0 W

Position 2: Person sitting next to window
Shape factors change dramatically: - $F_{p-window} \approx 0.35$ (window dominates view) - $F_{p-other} \approx 0.65$ (reduced view of other surfaces)
Radiative heat loss to window:

$$\dot{Q}_{rad,window} = 1.8 \times 0.35 \times 0.95 \times 5.67 \times 10^{-8} \times 2.230 \times 10^9 = 75.6 \text{ W}$$

Radiative heat loss to other surfaces:

$$\dot{Q}_{rad,other} = 1.8 \times 0.65 \times 0.95 \times 5.67 \times 10^{-8} \times 7.51 \times 10^8 = 47.2 \text{ W}$$

Total radiative loss (window position): 75.6 + 47.2 = 122.8 W

Comparison:

Position	Radiative Loss	Difference
Center of room	87.0 W	Baseline
Next to window	122.8 W	**+35.8 W (41% increase!)**

What does 35.8 W feel like?
To compensate for this extra radiative loss through sensible heat, we'd need to either: - Increase air temperature by approximately **4-5°C** (if convective heat transfer coefficient $h_c \approx 8$ W/m$^2 \cdot$ K) - Increase clothing insulation substantially - Move away from the window!
This is why: - Office workers near windows constantly adjust thermostats or use space heaters - Window seats on airplanes feel drafty even when cabin temperature is comfortable - Radiant floor heating systems can maintain comfort with lower air temperatures (reversing this effect)
Key insight: Shape factors translate geometric position into quantifiable thermal discomfort. A 3x increase in shape factor ($0.10 \rightarrow 0.35$) creates a 40% increase in radiative heat loss—enough to make the difference between comfort and significant discomfort, all while the air temperature remains constant.
This example demonstrates why thermal comfort modeling must account for both convective (air temperature) and radiative (surface temperatures weighted by shape factors) effects. Air temperature alone is insufficient!

12.6 Mean Radiant Temperature (MRT)

Earlier, when discussing heat dissipation mechanisms, we introduced Mean Radiant Temperature as a "clever simplification" that replaces tracking radiation to every surface with a single equivalent temperature. Now we'll formalize this concept and show how to calculate it.

12.6.1 Definition and Physical Meaning

The **Mean Radiant Temperature** (T_{mr}) is defined such that the radiative heat exchange with this uniform-temperature enclosure equals the actual radiative heat exchange with the non-uniform environment:

$$A_p \epsilon_p \sigma (T_p^4 - T_{mr}^4) = A_p \epsilon_p \sigma \sum_{i=1}^{n} F_{p-i} (T_p^4 - T_i^4)$$

Simplifying (canceling $A_p \epsilon_p \sigma T_p^4$ from both sides):

$$T_{mr}^4 = \sum_{i=1}^{n} F_{p-i} T_i^4$$

Taking the fourth root:

$$\boxed{T_{mr} = \left[\sum_{i=1}^{n} F_{p-i} T_i^4 \right]^{1/4}}$$

where:

- F_{p-i} = shape factor from person to surface i
- T_i = absolute temperature of surface i (in Kelvin)
- The summation is over all n surfaces in the enclosure

Physical meaning: MRT is the uniform temperature of an imaginary "black box" enclosure that would exchange the same net radiation with the occupant as the actual room with its non-uniform surface temperatures.

Example calculation: For a person in a room with floor at 21°C ($F_{p-floor} = 0.28$), ceiling at 19°C ($F_{p-ceiling} = 0.08$), and walls at 20°C ($F_{p-walls} = 0.64$):

$$T_{mr} = [(0.28)(294)^4 + (0.08)(292)^4 + (0.64)(293)^4]^{1/4} = 293.0 \text{ K} = 20.0°\text{C}$$

12.6.2 Approximation for Small Temperature Differences

The fourth-power formula is exact but cumbersome for hand calculations. When all surface temperatures are within about 10-15°C of each other (typical for most comfort situations), we can use a **linear approximation**:

$$\boxed{T_{mr} \approx \sum_{i=1}^{n} F_{p-i} T_i}$$

This is simply a **shape-factor-weighted average** of surface temperatures—much easier to calculate.

When is this valid? The approximation works well when $\frac{\Delta T}{T_{avg}} < 0.05$ (i.e., temperature variations <5% in absolute terms, or <15°C around room temperature).

Example (same room as above):

$$T_{mr} \approx (0.28)(21) + (0.08)(19) + (0.64)(20) = 20.2°\text{C}$$

Compared to exact: 20.0°C. Error: only 0.2°C—negligible for comfort calculations!

Why use the approximation?

- Much faster for hand calculations and quick estimates
- Sufficient accuracy for typical comfort scenarios
- Only use exact fourth-power formula when temperatures differ significantly (e.g., person near very cold window or radiant panel)

12.7 Operative Temperature

We now have two temperatures characterizing the thermal environment: air temperature (T_a) and mean radiant temperature (T_{mr}). **Operative temperature** (T_o) combines these into a single metric that characterizes how "warm" or "cold" the environment feels.

Starting from sensible heat loss from the clothing surface:

$$\dot{Q}_{sen} = \dot{Q}_{conv} + \dot{Q}_{rad} = h_c A_{cl}(T_{cl} - T_a) + h_r A_{cl}(T_{cl} - T_{mr})$$

where h_c and h_r are the convective and linearized radiative heat transfer coefficients.

We can define an **operative temperature** T_o such that:

$$\dot{Q}_{sen} = (h_c + h_r) A_{cl}(T_{cl} - T_o)$$

Equating these and solving for T_o:

$$\boxed{T_o = \frac{h_c T_a + h_r T_{mr}}{h_c + h_r}}$$

This is a **weighted average** of air temperature and MRT, with weights determined by the relative heat transfer coefficients.

Special case: For typical indoor conditions with low air velocity, $h_c \approx h_r \approx 4 - 5$ W/m$^2 \cdot$ K, giving:

$$T_o \approx \frac{T_a + T_{mr}}{2}$$

Practical use: Two environments with the same operative temperature should feel thermally similar, even if one has warmer air/cooler surfaces and the other has cooler air/warmer surfaces. This is why radiant floor heating can feel comfortable with lower air temperatures.

12.8 Clothing Insulation

Clothing acts as thermal resistance (R_{cl}) between skin and environment, appearing in the sensible heat network we discussed earlier.

The *clo* unit: Clothing insulation is measured in *clo* units, where:

$$\boxed{1 \text{ clo} = 0.155 \text{ m}^2 \cdot \text{K/W}}$$

This represents typical indoor clothing that maintains comfort at 21°C, 50% RH, with minimal air movement and sedentary activity.

Typical values:

Ensemble	Insulation
Naked	0 clo
Underwear	0.1 clo
Light summer (shorts, t-shirt)	0.3-0.5 clo
Business casual (pants, shirt)	0.6-0.7 clo
Indoor winter (pants, long sleeves, sweater)	0.8-1.0 clo
Business suit	1.0-1.5 clo
Heavy winter outdoor	2.0-3.0 clo

Clothing area factor: Clothing increases effective surface area beyond nude body surface ($A_{body} \approx 1.8$ m²):

$$A_{cl} = f_{cl} \cdot A_{body}$$

where $f_{cl} \approx 1.0 + 0.3 \cdot I_{cl}$ (empirical correlation), so 1 clo clothing gives $f_{cl} \approx 1.3$.

Effect on heat transfer: Clothing adds series thermal resistance in the sensible heat path while also increasing surface area and changing surface temperature. Net effect: higher insulation reduces heat loss, requiring higher operative temperatures for comfort.

12.9 Calculating Total Sensible Heat Loss

Combining all concepts, sensible heat loss from skin through clothing to environment:

$$\boxed{\dot{Q}_{sen} = \frac{A_{body}(T_{sk} - T_o)}{R_{cl} + \frac{1}{h_c + h_r}}}$$

From thermal balance: $\dot{Q}_{sen} = \dot{M} \cdot A_{sk} - \dot{W} - \dot{Q}_{evap}$

Worked example: Sedentary office worker

- Metabolic rate: 1.2 met = 70 W/m², A_{body} = 1.8 m² → 126 W total
- Clothing: 1.0 clo, $f_{cl} = 1.3$
- Evaporative loss: ~25 W (insensible perspiration)
- Required sensible loss: 126 - 25 = 101 W

For comfort with $T_{sk} \approx 33$°C, clothing at 1.0 clo (0.155 m² · K/W), and typical $h_c + h_r \approx 8$ W/m² · K:

$$101 = \frac{1.8(33 - T_o)}{0.155 + \frac{1}{8 \times 1.3}}$$

Solving: $T_o \approx 22$°C required for thermal balance and comfort.

12.10 The Predictive Mean Vote (PMV) Method

So far, we've developed the physics of thermal comfort: heat generation (metabolism), heat dissipation mechanisms (sensible and latent), and the environmental parameters that affect them (air temperature, MRT, humidity, air velocity). But physics alone doesn't tell us when people feel "comfortable" versus "too warm" or "too cold." This is where empirical thermal comfort models come in.

The **Predictive Mean Vote (PMV)** is the most widely used thermal comfort model, combining heat balance calculations with empirical data from human subject studies to predict thermal sensation on a standardized scale.

12.10.1 Historical Context and Development

In the 1960s-70s, Danish researcher **Povl Ole Fanger** conducted extensive laboratory studies where hundreds of subjects were exposed to controlled environmental conditions and asked to rate their thermal sensation. Fanger combined these empirical results with the heat balance equations we've been studying to develop the PMV model.

The model was groundbreaking because it:

- Provided a **quantitative** prediction of thermal sensation (not just "comfortable" or "uncomfortable")
- Was based on **physical principles** (heat balance) validated with **empirical data** (human responses)
- Could be calculated from **measurable parameters** (temperature, humidity, etc.) and **known characteristics** (clothing, activity)

PMV was adopted by:

- **ISO 7730** (International Organization for Standardization, 1984, updated 2005)
- **ASHRAE Standard 55** (Thermal Environmental Conditions for Human Occupancy, 1992 onward)
- **EN 15251** (European standard for indoor environmental criteria)

It remains the dominant model for predicting thermal comfort in mechanically conditioned buildings worldwide.

12.10.2 The PMV Equation

PMV is calculated from **six input parameters**:

Personal factors:

1. **Metabolic rate** ($\dot{M}$, in met or W/m^2) — activity level
2. **Clothing insulation** (I_{cl}, in clo) — what the person is wearing

Environmental factors:

3. **Air temperature** (T_a, in °C)
4. **Mean radiant temperature** (T_{mr}, in °C)
5. **Air velocity** (v_a, in m/s)
6. **Relative humidity** (RH, in %) or water vapor partial pressure (p_a, in Pa)

The PMV equation itself is quite complex. Without going into full detail, it has the form:

$$\text{PMV} = \left[0.303\exp(-0.036\dot{M}) + 0.028\right] \times (\text{thermal load})$$

where the thermal load is the difference between metabolic heat production and total heat loss (sensible + latent). The empirical factors (0.303, 0.036, 0.028) were determined from regression analysis of Fanger's experimental data.

The output is a thermal sensation vote on a **7-point scale**:

PMV Value	Thermal Sensation
+3	Hot
+2	Warm
+1	Slightly warm
0	Neutral (comfortable)
−1	Slightly cool
−2	Cool
−3	Cold

Interpretation:

- **PMV = 0**: Thermal neutrality — the "ideal" thermal environment where heat production equals heat loss at comfortable skin and core temperatures
- **PMV > 0**: Warm sensation — body is retaining more heat than desired, triggering cooling responses (vasodilation, sweating)
- **PMV < 0**: Cool sensation — body is losing more heat than desired, triggering warming responses (vasoconstriction, shivering)

12.10.3 Calculating PMV

The full PMV calculation is iterative because clothing surface temperature (T_{cl}) appears in both the equations for heat loss and in the determination of thermal load. The calculation procedure is:

1. **Assume initial** T_{cl} (often start with $(T_a + T_{mr})/2$ or similar estimate)
2. **Calculate heat losses**: Use T_{cl} to compute convective, radiative, and evaporative heat loss
3. **Calculate thermal load**: Difference between metabolic rate and total heat loss
4. **Calculate PMV**: Apply PMV formula
5. **Update** T_{cl} based on energy balance
6. **Iterate** until T_{cl} converges (typically 3-5 iterations)

In practice, nobody calculates PMV by hand. Instead, use:

- **Online tools**:
  - CBE Thermal Comfort Tool: https://comfort.cbe.berkeley.edu/
  - ASHRAE Standard 55 interactive tool
- **Software libraries**:
  - Python: `pythermalcomfort` package (https://github.com/CenterForTheBuiltEnvironment/pythermalcomfort)
  - MATLAB: Various implementations available
- **Building simulation tools**: EnergyPlus, IDA ICE, and others have built-in PMV calculations

12.10.4 Predicted Percentage Dissatisfied (PPD)

Even when the average person feels thermally neutral (PMV = 0), not everyone is satisfied. Individual differences in metabolism, clothing preferences, acclimatization, and personal preference mean that some people will always be dissatisfied.

Fanger developed the **Predicted Percentage Dissatisfied (PPD)** metric to quantify this:

$$\boxed{\text{PPD} = 100 - 95 \exp\left(-0.03353 \cdot \text{PMV}^4 - 0.2179 \cdot \text{PMV}^2\right)}$$

Key observations:

1. **Minimum dissatisfaction at PMV = 0**: Even under "perfect" neutral conditions, PPD = 5%. This irreducible minimum reflects individual physiological and psychological differences.
2. **Symmetric around neutral**: PPD increases equally whether the environment is too warm (PMV > 0) or too cool (PMV < 0).
3. **Rapid increase away from neutral**: Small deviations from PMV = 0 cause significant increases in dissatisfaction.

PMV	PPD (%)	Interpretation
0	5	Best achievable (5% still dissatisfied)
±0.5	10	ASHRAE 55 Category I (highest quality)
±0.7	15	ASHRAE 55 Category II (normal expectation)
±1.0	26	ASHRAE 55 Category III (acceptable minimum)
±1.5	52	More than half dissatisfied
±2.0	75	Unacceptable for occupied spaces

PMV	PPD (%)	Interpretation

ASHRAE Standard 55 compliance:

- Requires PMV between **−0.5 and +0.5** (PPD $< 10\%$) for acceptable thermal comfort
- Stricter designs may target PMV between **−0.2 and +0.2** (PPD $< 6\%$)

i Learn by doing: PMV Calculation

Let's calculate PMV for a typical office scenario using a computational tool (since hand calculation is impractical).
Scenario:

- **Person**: Office worker, sitting (1.2 met), business casual clothing (0.7 clo)
- **Environment**:
  - Air temperature: $T_a = 23°C$
  - Mean radiant temperature: $T_{mr} = 22°C$ (slightly cooler surfaces)
  - Air velocity: $v_a = 0.1$ m/s (typical for no draft)
  - Relative humidity: 50%

Using the CBE Thermal Comfort Tool or `pythermalcomfort`: calculate the PMV and PPD.

12.10.5 Limitations of PMV

While PMV is the industry standard, it has important limitations:

1. **Steady-state assumption**: PMV assumes the body is in thermal equilibrium. It doesn't capture:
   - Transient thermal discomfort (e.g., entering a cold building from outside)
   - Time-varying conditions (temperature ramping, intermittent heating)
   - Thermal history effects
2. **Population bias**: Developed primarily from studies of young, healthy European and North American subjects. May not generalize well to:
   - Different ethnic/geographic populations
   - Elderly occupants
   - Children
   - People with certain health conditions

3. **Mechanically conditioned buildings**: PMV works best in HVAC-controlled environments. In naturally ventilated buildings, occupants:
    - Adapt their expectations based on outdoor conditions
    - Have higher tolerance for temperature variations
    - Adjust clothing and behavior more dynamically
4. **No adaptation or expectation**: PMV doesn't account for:
    - Psychological adaptation (getting used to conditions)
    - Seasonal acclimatization
    - Cultural/regional preferences
    - Occupant control (having a window or thermostat increases tolerance)
5. **Individual variation**: The 5% minimum PPD at PMV = 0 is somewhat optimistic. In real buildings:
    - Actual dissatisfaction often exceeds predicted values
    - Local discomfort factors (drafts, radiant asymmetry, vertical temperature gradients) not fully captured
    - Metabolic rates and clothing are estimated, not measured

For naturally ventilated buildings: ASHRAE 55 includes an **adaptive comfort model** that better predicts comfort in these contexts, allowing wider acceptable temperature ranges based on outdoor climate.

Despite these limitations, PMV remains extremely useful for:

- **Design**: Sizing HVAC systems and selecting setpoints
- **Commissioning**: Verifying that buildings meet comfort targets
- **Comparison**: Evaluating different design alternatives
- **Research**: Standardized metric for thermal comfort studies

12.11 ASHRAE Comfort Charts

While PMV/PPD provides a quantitative prediction of thermal sensation, designers and building operators often need a quicker way to assess whether environmental conditions meet comfort requirements. **ASHRAE Standard 55** provides graphical **comfort zones** that show acceptable combinations of operative temperature and humidity for typical office occupancies.

These comfort charts are based on the PMV model (specifically, the range where -0.5 PMV $+0.5$, corresponding to PPD 10%), but presented in an easily interpretable format.

12.11.1 Reading the Comfort Chart

The ASHRAE comfort chart has the following structure:

Axes:

- **X-axis**: Operative temperature (T_o, in °C or °F)
- **Y-axis**: Humidity ratio (g water/kg dry air) or relative humidity (%)

Comfort zones (shaded regions) show acceptable combinations. Conditions inside the zone satisfy the PMV criterion (-0.5 PMV $+0.5$). Conditions outside are either too warm, too cool, too humid, or too dry.

Assumptions:

- Typical office activity: 1.0-1.2 met (sedentary)
- Low air velocity: < 0.2 m/s (still air, no draft)
- Separate zones for different clothing levels (summer vs. winter)

Why two comfort zones?

ASHRAE 55 defines separate comfort zones for:

1. **Summer conditions** (0.5 clo clothing): Light, short-sleeved clothing
2. **Winter conditions** (1.0 clo clothing): Long pants, long sleeves, possibly sweater

This reflects **seasonal behavioral adaptation**: people dress differently in summer vs. winter based on outdoor conditions and social norms, even though indoor temperatures could theoretically be the same year-round.

12.11.2 Acceptable Ranges for Thermal Comfort

For **typical office environments** (1.0-1.2 met activity, air velocity < 0.2 m/s):

Summer comfort zone (0.5 clo):

- Operative temperature: **23.5°C to 26.0°C** (74°F to 79°F)
- Humidity ratio: **0.000 to 0.012** kg water/kg dry air
- Equivalent RH at mid-range: approximately 30-60%

Winter comfort zone (1.0 clo):

- Operative temperature: **20.0°C to 23.5°C** (68°F to 74°F)
- Humidity ratio: **0.000 to 0.012** kg water/kg dry air
- Equivalent RH at mid-range: approximately 30-60%

Key observations:

1. **~3.5°C difference between summer and winter**: The winter zone is shifted cooler because occupants wear more clothing. This allows energy savings—heating to 21°C in winter can be just as comfortable as cooling to 25°C in summer, if occupants dress appropriately.
2. **Overlap region** (23.5°C): This temperature is acceptable year-round, regardless of season. It's often used as a year-round setpoint in buildings where seasonal clothing variation is minimal.
3. **Humidity limits**: The upper humidity limit (~0.012 kg/kg 60-70% RH) is set primarily for:

   - Microbial growth prevention
   - Prevention of surface condensation
   - Comfort (high humidity impairs evaporative cooling)

   The lower limit is less strict, but very low humidity (<20% RH) can cause:

   - Dry skin and mucous membranes
   - Increased static electricity
   - Increased respiratory irritation

4. **Air velocity matters**: The standard zones assume low air movement. Higher air velocity (from fans, natural ventilation) extends the upper temperature limit by enhancing convective and evaporative cooling:

   - **Elevated air speed provision**: ASHRAE 55 allows temperatures up to 3°C higher if air velocity increases to 0.8-1.2 m/s (with occupant control)

12.11.3 Using Comfort Charts for HVAC Control

Comfort charts inform HVAC control strategies in several ways:

1. Setpoint selection: Instead of a single fixed setpoint (e.g., "maintain 22°C"), the comfort zone suggests:

- **Heating setpoint**: Lower bound of winter zone (~20°C)
- **Cooling setpoint**: Upper bound of summer zone (~26°C)
- **Deadband**: Region between heating and cooling where no active conditioning occurs

This approach reduces energy consumption compared to tight temperature control around a single setpoint.

2. Seasonal setpoint adjustment: Some strategies adjust setpoints seasonally:

- Winter (Nov-Mar): Heat to 20-21°C, cool above 24°C

- Summer (May-Sep): Heat below 22°C, cool to 24-25°C
- Shoulder seasons: Wider deadband

This acknowledges that occupants naturally adjust clothing seasonally.

3. Humidity control: While temperature is the primary control variable, the charts show that humidity matters:

- **Dehumidification**: When RH > 60-65%, remove moisture even if temperature is acceptable
- **Humidification**: Optional, but beneficial if RH < 30% (especially in winter in cold climates)

4. Free cooling / economizer operation: The comfort zone width allows "free cooling" (using outdoor air when conditions permit) over a wider range of outdoor temperatures, reducing mechanical cooling energy.

12.12 Putting It All Together

This lecture has covered a lot of ground, moving from basic physiology to quantitative comfort prediction. Let's synthesize the key concepts and their implications for autonomous building control.

12.12.1 From Physics to Comfort Prediction

The chain of reasoning we've developed follows this logic:

1. Human body as thermal system:

- Core temperature must be maintained at ~37°C for proper physiological function
- Heat is continuously generated through metabolism (quantified in met units)
- Limited thermoregulation mechanisms (sweating, shivering, vasoconstriction/dilation) have finite capacity

2. Heat dissipation pathways:

- **Sensible heat** (convection + radiation): Accounts for ~60-75% of heat loss in typical conditions
- **Latent heat** (evaporation): Accounts for ~25-40%, becomes dominant during exercise or in warm environments
- These operate through parallel thermal/moisture networks with clothing as series resistance

3. Environmental characterization: Four environmental parameters determine heat exchange rates:

- **Air temperature** (T_a): Drives convective heat transfer
- **Mean Radiant Temperature** (T_{mr}): Drives radiative heat transfer (shape-factor-weighted average of surface temperatures)
- **Humidity** (RH or W): Determines evaporative cooling potential
- **Air velocity** (v_a): Enhances convective and evaporative heat transfer

4. Personal factors: Two personal characteristics modulate heat exchange:

- **Metabolic rate** (met): Determines heat generation based on activity level
- **Clothing insulation** (clo): Provides thermal and moisture resistance

5. Thermal balance and comfort:

- **Thermal neutrality**: Occurs when heat generation equals heat dissipation at comfortable skin temperature (~33°C)
- **PMV model**: Quantifies deviation from neutrality as thermal sensation on -3 to $+3$ scale
- **PPD metric**: Translates PMV into percentage of dissatisfied occupants (minimum 5% at PMV = 0)
- **ASHRAE comfort zones**: Define acceptable operative temperature and humidity ranges (PMV between -0.5 and $+0.5$)

This framework allows us to predict whether a given combination of environmental conditions, personal factors, and clothing will feel comfortable—enabling proactive HVAC control rather than reactive adjustment to complaints.

12.12.2 Implications for Building Control

Understanding occupant thermal comfort fundamentally changes how we think about building environmental control:

1. Multi-dimensional control problem:

Air temperature alone is insufficient. Effective control must address:

- **Temperature**: Both air (T_a) and radiant (T_{mr})—a person near a cold window can feel uncomfortable even if air temperature is perfect
- **Humidity**: Especially important for warm conditions and active occupants (affects evaporative cooling)
- **Air movement**: Can extend upper temperature limits by 2-3°C if occupant-controlled

This has practical implications:

- Thermostats should measure or estimate operative temperature ($(T_a + T_{mr})/2$), not just air temperature
- Humidity sensors should complement temperature sensors, especially in cooling mode
- Radiant system design (floor heating, ceiling panels) must account for shape factors

2. Individual differences and flexibility:

The 5% minimum PPD at PMV = 0 understates real-world variation. In practice:

- **10-30% dissatisfaction** is common even in "well-controlled" buildings
- Sources: individual metabolic differences, clothing preferences, acclimatization, personal thermal history, psychological factors

Control strategies must acknowledge this:

- **Personal environmental control**: Operable windows, desk fans, task lighting with heat generation
- **Spatial diversity**: Provide zones at different temperatures, allowing occupants to self-select
- **Temporal flexibility**: Allow temperature to drift within comfort zone rather than maintaining tight setpoint
- **Feedback mechanisms**: Monitor complaints, occupant behavior (opening windows, using space heaters) to adapt control

3. Seasonal adaptation for energy savings:

Supporting seasonal clothing changes (0.5 clo in summer $\rightarrow$ 1.0 clo in winter) allows:

- **Winter**: Heat to 20-21°C instead of 23°C (10-15% heating energy savings)
- **Summer**: Cool to 24-25°C instead of 23°C (15-25% cooling energy savings)
- **Deadband**: Wider range between heating and cooling setpoints reduces equipment cycling and energy use

Implementation challenges:

- Requires occupant education and buy-in
- Dress codes may constrain clothing adaptation
- Gradual seasonal transitions (1°C every 2-3 weeks) needed to allow acclimatization
- May conflict with expectations that "air conditioning = constant temperature"

4. Integration with occupancy information:

Thermal comfort control can be dramatically improved by knowing:

- **When** spaces are occupied (no need to condition unoccupied spaces to full comfort)
- **Who** is present (typical vs. atypical metabolic rates, clothing preferences)
- **What** activities are occurring (meeting = sedentary 1.2 met, gym = exercise 4+ met)

This links directly to occupancy sensing and modeling (Lecture 8), creating opportunities for:

- Demand-controlled conditioning (precool before occupation, setback when vacant)
- Activity-based control (lower setpoints for gyms, higher for sedentary spaces)
- Personalized comfort profiles (learned preferences per occupant or group)

12.12.3 Local Discomfort Factors

While PMV/PPD provides a good overall assessment of thermal comfort, several **local discomfort factors** can cause complaints even when whole-body thermal balance is neutral:

1. Drafts (air velocity variability):

- Perception of draft depends on temperature, velocity, and **turbulence**
- Higher turbulence intensity increases draft perception
- Particularly noticeable on the back of neck, ankles
- ASHRAE 55 limits: Peak local air velocity < 0.25 m/s in winter, < 0.3 m/s in summer

2. Vertical air temperature gradients:

- Head-to-ankle temperature difference causes discomfort
- Common in poorly designed systems with stratification
- ASHRAE 55 limits: < 3°C difference between 0.1m and 1.1m height (seated), or 0.1m and 1.7m (standing)

3. Warm or cool floors:

- Direct conduction from feet makes floor temperature particularly noticeable
- ASHRAE 55 limits: 19-29°C floor temperature
- Radiant floor heating typically operates at 24-28°C (warm but not uncomfortable)

4. Radiant asymmetry:

- Large temperature differences between opposing surfaces (e.g., cold window vs. warm opposite wall) create directional discomfort
- Our earlier calculations showed 35.8 W extra heat loss near a cold window—equivalent to 4-5°C air temperature reduction
- ASHRAE 55 limits: Cool wall (window) < 10°C cooler than average, warm ceiling < 5°C warmer

Practical considerations:

- PMV assumes **uniform environment**—actual buildings have spatial variations
- Good design minimizes these local factors through proper system selection, placement, and controls

- When local discomfort exists, occupants may report being "too cold" or "too hot" even if PMV predicts neutral

12.13 Additional Resources

12.13.1 Primary Reference

- Reddy, T. Agami. *Heating and Cooling of Buildings: Principles and Practice of Energy Efficient Design*. 3rd edition. CRC Press, 2016. **Chapter 3: Elements of Heat Transfer for Buildings**

12.13.2 Supplementary Reading

- ASHRAE Standard 55-2020: *Thermal Environmental Conditions for Human Occupancy*
- Fanger, P. O. *Thermal Comfort: Analysis and Applications in Environmental Engineering*. McGraw-Hill, 1970. (Classic reference, somewhat dated but foundational)
- Parson, K. C. *Human Thermal Environments: The Effects of Hot, Moderate, and Cold Environments on Human Health, Comfort, and Performance*. 3rd edition. CRC Press, 2014.

12.13.3 Online Resources

12.13.3.1 Thermal Comfort Tools

- CBE Thermal Comfort Tool: https://comfort.cbe.berkeley.edu/ (Interactive PMV calculator and ASHRAE comfort chart visualization)
- ASHRAE Standard 55 Interactive Comfort Tool

12.13.3.2 PMV/PPD Calculations

- Python package `pythermalcomfort`: https://github.com/CenterForTheBuiltEnvironment/pythermalcomfort
- ISO 7730 standard for PMV/PPD calculation

12.13.3.3 ASHRAE Resources

- ASHRAE Handbook—Fundamentals, Chapter 9: Thermal Comfort
- ASHRAE Standard 55 documentation and user's manual

13 Basics of AC Power Systems for Buildings

13.1 Lecture Overview

Learning Objectives

By the end of this module, students will be able to:

- Understand how power is delivered from the grid to the electrical appliances/loads in a building, including:
  - The mathematical basis for AC power delivery:
    - Make use Ohm's Law and Kirchhoff's Voltage/Current Laws for DC circuits in order to derive an expression for power provided to an arbitrary RLC load
    - Distinguish between power losses and power transmitted in a circuit
    - Understand the steady-state and transient effects of the R, L, and C components for the load in the DC case
    - Recreate the expression for power under an AC voltage/current source and an arbitrary impedance
    - Recognize the steady-state and transient effects of the impedance (resistance + reactance) in the circuit for the AC case
    - Be able to use time domain (i.e., instantaneous) power expressions, as well as phasor and/or complex representations
  - The underlying physical mechanism from which sinusoidal voltage sources arise:
    - Understand the working principles and components of an AC electrical generator in general, and 3-phase generators in particular
    - Derive an expression for the voltage at each of the 3 phases of the generator (mathematically showing their phase difference)
    - Understand how power delivered/consumed and power generated need to be balanced, and the effect of this balance on the grid's frequency
    - Describe, conceptually, how power from the generator gets transmitted to the distribution system and to the buildings in it
      - Understand the role of transformers in the transmission/distribution grids
      - Understand how split-phase systems to residential buildings in the US work

- Analyze AC Power Usage
  - Starting from a known voltage source $v(t)$ and current $i(t)$, derive expressions for active, reactive, apparent and instantaneous power, as well as power factor
    - Understand the value of RMS and average quantities for voltage and current over some cycle-aligned period
    - Be able to leverage the phasor representation of these quantities
  - Calculate power quantities (active, reactive, etc.) from RMS or sub-cycle measurements of voltage and current
  - Be familiar with the VI trajectory (a single AC cycle plot of current versus voltage) as a visualization aid to differentiate between different load impedances

Topics Covered

- Review of DC Circuit Fundamentals
  - Ohm's Law and Kirchhoff's Laws
  - Power in DC circuits
  - RLC components and their behavior
- Mathematical Framework for AC Power
  - Time domain representation
  - Phasor and complex notation
  - Impedance and reactance
- AC Electrical Generators
  - Working principles and components
  - Single-phase and 3-phase generation
  - Grid frequency and power balance
- Power Transmission and Distribution
  - Transformers and voltage levels
  - Split-phase residential systems
- AC Power Analysis
  - Active, reactive, apparent, and instantaneous power
  - Power factor
  - RMS and average quantities
  - VI trajectory visualization

Project Milestones

Understanding AC power systems is fundamental for the smart metering aspects of the final project. Students working on energy monitoring and analysis will need to apply concepts from this lecture to: - Interpret voltage and current measurements from smart meters - Calculate power consumption metrics - Understand load characteristics through VI trajectory analysis

13.2 Lecture Notes

13.2.1 Introduction: Why AC Power Matters for Buildings

Throughout this course, we've been building toward the goal of creating autonomous, sustainable buildings—spaces that can sense their environment, make intelligent decisions, and act to optimize energy consumption while maintaining occupant comfort. To achieve this vision, we need to understand not just thermal dynamics and comfort, but also how buildings consume and manage electrical energy.

Buildings account for approximately 40% of total energy consumption in developed countries, and addressing climate change will require deep decarbonization of this sector. A key strategy for decarbonization is **electrification**—transitioning away from fossil fuel combustion (natural gas furnaces, oil boilers) toward electric heat pumps, electric water heaters, and other high-efficiency electric technologies. This shift makes understanding electrical power systems in buildings more critical than ever for anyone working on building automation and energy management.

Today, nearly all electrical power delivered to buildings uses **alternating current (AC)** systems. AC power has dominated for over a century due to compelling advantages: transformers can efficiently step voltages up and down, enabling long-distance transmission at high voltages (minimizing resistive losses) and safe low-voltage distribution within buildings. The infrastructure—from power plants to distribution grids to building panels—is built around AC.

However, an interesting trend is emerging: **DC (direct current) systems are making a comeback in buildings**. Many modern loads (LED lighting, computers, electronics, variable-speed motor drives) internally convert AC to DC anyway, wasting energy in the conversion. Solar panels and batteries naturally produce and store DC power. Recent advances in DC-DC converters, solid-state transformers, and power electronics have made DC distribution systems increasingly viable. Some experts envision future buildings with DC microgrids, potentially offering higher efficiency and better integration with renewable energy and storage.

Despite this potential shift, AC power remains the foundation of building electrical systems today, and will for the foreseeable future. Understanding AC power is essential for:

- Designing and analyzing smart metering systems (a focus of the final project)
- Interpreting measurements from building energy management systems
- Understanding how different loads behave and how to control them
- Evaluating the energy and power quality impacts of building automation strategies

In this lecture, we'll build a comprehensive understanding of AC power systems for buildings, starting from DC circuit fundamentals and progressing through AC circuit analysis, power generation and distribution, and practical measurement techniques.

13.2.2 Review of DC Circuit Fundamentals

Before diving into AC power, we'll briefly review DC (direct current) circuit fundamentals. These concepts form the foundation for understanding AC circuits, and many of the analysis techniques carry over directly.

In DC circuits, voltage and current are constant over time (or vary slowly enough that we can treat them as constant). This simplification makes the mathematics more tractable and helps us build intuition that we'll extend to the AC case.

Key Assumptions and Idealizations:

The circuit models we'll use make several simplifying assumptions:

1. **Linearity**: We assume that circuit elements obey linear relationships between voltage and current. For example, a resistor's resistance doesn't change with voltage or current. This is an approximation—real resistors heat up and change resistance—but it's accurate enough for most practical purposes.
2. **Lumped elements**: We treat resistors, inductors, and capacitors as discrete components with concentrated properties, ignoring distributed effects like the resistance of connecting wires or stray capacitance between traces.
3. **Ideal sources**: We assume voltage and current sources can deliver arbitrary amounts of power without internal losses.

These assumptions enable us to use powerful analysis techniques like **superposition** (the response to multiple sources is the sum of individual responses) and allow us to build **network models** where complex systems are represented as interconnected components.

Connection to Thermal Systems:

If this sounds familiar, it should! In Lectures 5 and 6, we developed thermal network models for buildings using the exact same approach. We created lumped thermal resistances (R) and capacitances (C) to model heat flow and storage, then applied network analysis (analogous to Kirchhoff's laws) to derive building thermal dynamics. The mathematical machinery is identical—only the physical interpretation changes:

Electrical Domain	Thermal Domain
Voltage (V)	Temperature (T)
Current (I)	Heat flow rate ($\dot{Q}$)
Resistance (R)	Thermal resistance (R_{th})
Capacitance (C)	Thermal capacitance (C_{th})

This analogy is powerful and extends to mechanical systems as well (force voltage, velocity current). Once you understand the mathematical structure, you can apply it across multiple physical domains.

13.2.2.1 Ohm's Law and Kirchhoff's Laws

The fundamental laws governing circuit behavior are:

Ohm's Law relates voltage across a resistor to the current through it:

$$V = IR$$

where V is voltage (volts), I is current (amperes), and R is resistance (ohms, Ω).

Kirchhoff's Current Law (KCL) states that the sum of currents entering a node equals the sum leaving:

$$\sum I_{in} = \sum I_{out}$$

This is just conservation of charge—electrons can't accumulate at a node.

Kirchhoff's Voltage Law (KVL) states that the sum of voltage drops around any closed loop is zero:

$$\sum V_{loop} = 0$$

This follows from conservation of energy—no net energy is gained traveling around a closed path.

These laws apply to both DC and AC circuits, though in AC analysis we'll extend them to work with complex quantities (phasors) rather than just real numbers.

13.2.2.2 Power in DC Circuits

Power is the rate of energy transfer or conversion. In electrical circuits, power delivered to a component is the product of voltage across it and current through it:

$$P = VI$$

where P is power in watts (W), V is voltage in volts (V), and I is current in amperes (A).

Derivation: Consider a charge Δq moving through a voltage difference V. The energy transferred is $\Delta E = V\Delta q$. The power is the rate of energy transfer:

$$P = \frac{\Delta E}{\Delta t} = V\frac{\Delta q}{\Delta t} = VI$$

since current is defined as $I = \Delta q/\Delta t$.

For a **resistive load**, we can substitute Ohm's law ($V = IR$) to get alternative expressions:

$$P = VI = (IR)I = I^2R$$

or equivalently,

$$P = VI = V\left(\frac{V}{R}\right) = \frac{V^2}{R}$$

These three equivalent forms are useful in different contexts:

- $P = VI$ when you know voltage and current
- $P = I^2R$ when you know current and resistance
- $P = V^2/R$ when you know voltage and resistance

In a resistor, electrical energy is converted to heat—the power P represents energy dissipation.

13.2.2.3 Resistors, Inductors, and Capacitors (RLC Components)

Electrical circuits are built from three fundamental passive elements: **resistors (R)**, **inductors (L)**, and **capacitors (C)**. Each has a distinct voltage-current relationship and plays a different role in circuit behavior.

These elements have direct analogs in other physical systems, revealing the universal structure of dynamic systems:

Element	Electrical	Thermal	Mechanical (Translational)
Dissipative	Resistor (R)	Thermal resistance	Damper (friction)
Energy storage (type 1)	Inductor (L)	-	Mass (inertia)
Energy storage (type 2)	Capacitor (C)	Thermal capacitance	Spring (compliance)

The mathematical equations governing these elements are structurally identical across domains. This is why we could use RC circuit analysis techniques when modeling building thermal dynamics in earlier lectures.

13.2.2.3.1 Resistors

We've already introduced **resistors** through Ohm's law ($v = iR$) and seen that they dissipate power as heat ($p = i^2R = v^2/R$).

What's important to note about resistors is their **instantaneous response**: they have no memory or transient behavior. The current at any instant depends only on the voltage at that instant, not on past history. Whether in steady-state or during rapid changes, the relationship $v = iR$ always holds immediately. Resistors dissipate energy continuously but do not store it.

13.2.2.3.2 Inductors

An **inductor** stores energy in a magnetic field created by current flowing through a coil of wire.

Voltage-current relationship:

$$v = L\frac{di}{dt}$$

where L is the inductance in henries (H). The voltage across an inductor is proportional to the *rate of change* of current through it, not the current itself.

Energy storage: Energy is stored in the magnetic field:

$$E = \frac{1}{2}LI^2$$

Unlike resistors, inductors don't dissipate energy (in the ideal case)—they store it when current increases and release it when current decreases.

Transient behavior: Inductors oppose *changes* in current. If you try to suddenly change the current through an inductor, it produces a large opposing voltage. This creates time-dependent (transient) behavior. A key result: **current through an inductor cannot change instantaneously**. In steady-state DC conditions (where $di/dt = 0$), an inductor acts like a short circuit (zero voltage drop).

13.2.2.3.3 Capacitors

A **capacitor** stores energy in an electric field between two conductive plates separated by an insulator.

Voltage-current relationship:

$$i = C\frac{dv}{dt}$$

where C is the capacitance in farads (F). The current through a capacitor is proportional to the *rate of change* of voltage across it.

Energy storage: Energy is stored in the electric field:

$$E = \frac{1}{2}CV^2$$

Like inductors, ideal capacitors don't dissipate energy—they store it when voltage increases and release it when voltage decreases.

Transient behavior: Capacitors oppose *changes* in voltage. If you try to suddenly change the voltage across a capacitor, it draws a large current. This creates time-dependent behavior. A key result: **voltage across a capacitor cannot change instantaneously**. In steady-state DC conditions (where $dv/dt = 0$), a capacitor acts like an open circuit (zero current flow).

13.2.2.4 Power Delivered to RLC Loads in DC Circuits

Now we can put these concepts together to understand power flow in circuits containing R, L, and C elements.

Recall that instantaneous power delivered to any circuit element is $p(t) = v(t) \cdot i(t)$.

For a resistor:

$$p_R(t) = v(t) \cdot i(t) = i^2(t)R$$

This power is **dissipated** as heat. The energy is permanently lost from the circuit.

For an inductor:

$$p_L(t) = v(t) \cdot i(t) = L\frac{di}{dt} \cdot i = \frac{d}{dt}\left(\frac{1}{2}Li^2\right)$$

This power represents the rate of change of stored magnetic energy. When $p_L > 0$, energy flows into the inductor (building up the magnetic field). When $p_L < 0$, energy flows back out (as the field collapses). Over a complete cycle of charging and discharging, **net energy dissipation is zero** (in the ideal case).

For a capacitor:

$$p_C(t) = v(t) \cdot i(t) = v \cdot C\frac{dv}{dt} = \frac{d}{dt}\left(\frac{1}{2}Cv^2\right)$$

Similarly, this represents the rate of change of stored electric energy. Energy sloshes in and out, but with **no net dissipation** over a complete cycle (again, ideally).

Key distinction:

- **Power losses** (dissipated permanently): occur in resistive elements
- **Power transmitted** (flowing back and forth): occurs in reactive elements (L and C) during energy storage and release

A note on idealizations: In real circuits, the wires connecting components also have resistance (**line losses**) and reactance (**line impedance**—small amounts of inductance and capacitance). Power is dissipated in these wires, and they affect circuit dynamics. However, for our analysis, we'll assume these effects are negligible compared to the actual circuit components. This is reasonable for short wire runs and moderate currents, but in long-distance power transmission or high-current applications, line losses and reactance become very important.

This distinction between dissipated power and transmitted power becomes crucial when we move to AC circuits, where we'll formalize these concepts as **real power** (dissipated in R) versus **reactive power** (oscillating in L and C).

13.2.3 Transition to AC: Why Alternating Current?

Now that we understand DC circuits, why complicate things by making voltage and current vary sinusoidally over time? The answer lies in the practical challenges of power distribution.

Historical Context: The War of Currents

In the late 1800s, Thomas Edison championed direct current for electrical distribution, building DC power systems across major cities. Nikola Tesla and George Westinghouse advocated for alternating current. This "War of Currents" was ultimately won by AC, not because AC is inherently superior for all applications, but because of one critical advantage: **transformers**.

The Transformer Advantage

Transformers can efficiently change AC voltage levels using electromagnetic induction—the same principle behind inductors. This capability is crucial for power distribution:

1. **High-voltage transmission reduces line losses**: Recall that power loss in a wire is $P_{loss} = I^2 R_{line}$. For a given amount of power $P = VI$ to transmit, we can use high voltage and low current, dramatically reducing resistive losses in transmission lines. AC systems routinely transmit at hundreds of kilovolts, then step down to safe levels for buildings.

2. **Transformers are simple and efficient**: With no moving parts, transformers can convert voltage levels at 95-99% efficiency. Early DC systems had no equivalent technology—Edison's plants had to be located within about a mile of customers due to unacceptable line losses at the low DC voltages used.

3. **Safety and versatility**: AC systems generate at optimal voltages for generators, transmit at high voltages for efficiency, and deliver at safe voltages (120V/240V) for end users, all using transformers.

What Has Changed Since Then?

The War of Currents was decided by the technology available in the 1890s. Today, power electronics have changed the landscape:

- **Modern DC-DC converters** (based on high-frequency switching) can now change DC voltage levels efficiently, something impossible in Edison's era
- **High-voltage DC (HVDC) transmission** is now used for very long distances and undersea cables, where AC's reactive losses become problematic
- **Solar panels, batteries, and LED lighting** all operate on DC internally
- **Modern electronics** (computers, phone chargers, variable-speed drives) convert AC to DC immediately upon receiving power, wasting energy in the conversion

Despite these developments, AC remains dominant because of massive infrastructure investment and the fact that transformers are still simpler and cheaper than power electronics for most applications. However, **DC microgrids** within buildings and **DC distribution systems** are gaining traction, especially where solar and battery storage are involved.

For the foreseeable future, you'll need to understand both: AC for the grid and building mains, DC for electronics and emerging distributed energy systems.

13.2.4 Mathematical Framework for AC Power

In DC circuits, voltage and current are constant, making analysis straightforward. In AC circuits, voltage and current vary sinusoidally with time, which fundamentally changes the circuit dynamics. This time-dependence means that inductors and capacitors—which were relatively simple in DC steady-state—now play active roles, continuously storing and releasing energy.

To analyze AC circuits, we need mathematical tools that handle these time-varying quantities efficiently. We'll develop three representations: **time domain** (using trigonometric functions), **phasor domain** (using rotating vectors), and **complex notation** (using complex numbers). Each has its advantages depending on what we're trying to calculate.

13.2.4.1 Sinusoidal Voltage and Current Sources

AC power systems use **sinusoidal** voltage and current waveforms—signals that vary as sine or cosine functions of time. A sinusoidal voltage can be written as:

$$v(t) = V_m \sin(\omega t + \phi)$$

where:

- V_m is the **amplitude** (peak voltage) in volts
- ω is the **angular frequency** in radians per second
- ϕ is the **phase angle** in radians (or degrees), which determines where the waveform starts at $t = 0$

The **frequency** f (in Hertz, cycles per second) relates to angular frequency by:

$$\omega = 2\pi f$$

In the US, the grid operates at $f = 60$ Hz, so $\omega = 2\pi \cdot 60 \approx 377$ rad/s. In most other countries, $f = 50$ Hz.

Why sinusoids? Real AC sources aren't perfectly sinusoidal—they have some distortion. However, if we assume our circuits are **linear**, then the **superposition principle** applies: the circuit's response to a sum of inputs equals the sum of responses to individual inputs. This means we can decompose any periodic waveform into a sum of sinusoids at different frequencies (Fourier series), analyze the circuit's response to each sinusoid separately, and add up the results. For power system analysis, the fundamental frequency (60 Hz) typically dominates, so we often analyze only that component.

13.2.4.2 Time Domain Representation

In the time domain, we explicitly write voltage and current as functions of time. For an AC circuit with sinusoidal excitation:

Voltage:

$$v(t) = V_m \sin(\omega t + \phi_v)$$

Current:

$$i(t) = I_m \sin(\omega t + \phi_i)$$

The phase angles ϕ_v and ϕ_i are generally different (i.e., in AC circuits, voltage and current don't necessarily reach their peaks at the same time). Why? Well, if the current drawn was all for power being dissipated through resistive loads, then they would be the same. But as we saw before, inductors and capacitors store energy in magnetic and electric fields, thus causing a misalignment between the current and voltage sinusoids. Thus, the **phase difference** $\phi = \phi_v - \phi_i$ is crucial for understanding power flow.

Let's visualize this with an example where voltage and current are out of phase:

```
import numpy as np
import matplotlib.pyplot as plt

# Parameters
f = 60  # Hz (US grid frequency)
omega = 2 * np.pi * f  # Angular frequency
V_m = 170  # Peak voltage (about 120 V RMS)
I_m = 100   # Peak current
phi_v = 0  # Voltage phase angle (reference)
phi_i = -np.pi/4  # Current lags voltage by 45 degrees

# Time vector (3 cycles)
t = np.linspace(0, 3/f, 1000)

# Calculate waveforms
v = V_m * np.sin(omega * t + phi_v)
i = I_m * np.sin(omega * t + phi_i)

# Plot
plt.figure(figsize=(8, 4))
plt.plot(t * 1000, v, label='Voltage $v(t)$', linewidth=2)
plt.plot(t * 1000, i, label='Current $i(t)$', linewidth=2)
plt.xlabel('Time (ms)')
plt.ylabel('Amplitude')
plt.title('AC Voltage and Current (60 Hz, current lagging by 45°)')
plt.grid(True, alpha=0.3)
plt.legend()
plt.axhline(y=0, color='k', linewidth=0.5)
plt.tight_layout()
plt.show()
```

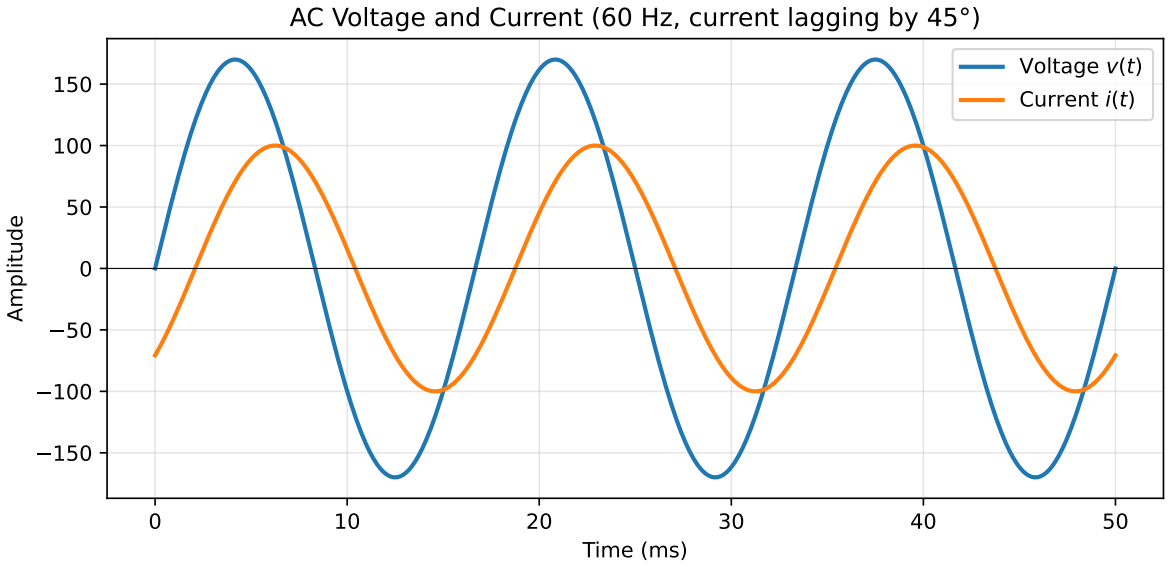


Figure 13.1: Voltage and current waveforms in an AC circuit with phase difference

Notice that the current waveform reaches its peak *after* the voltage waveform—we say the current **lags** the voltage. This happens when the load contains inductance (like motors or transformers). Conversely, capacitive loads cause current to **lead** voltage.

13.2.4.3 Impedance and Reactance

In DC circuits, Ohm's law states that $V = IR$, where resistance R relates voltage to current. In AC circuits, we need a generalization that accounts for the phase difference between voltage and current.

Impedance (Z) is the AC generalization of resistance. It relates the phasor representations of voltage and current:

$$\tilde{V} = \tilde{Z} \cdot \tilde{I}$$

where the tilde ($\tilde{}$) denotes phasor quantities (we'll define phasors formally in the next section).

Impedance is a **complex number** with both magnitude and phase:

$$Z = R + jX$$

where:

- R is the **resistance** (real part), representing energy dissipation
- X is the **reactance** (imaginary part), representing energy storage
- $j = \sqrt{-1}$ is the imaginary unit

The magnitude $|Z| = \sqrt{R^2 + X^2}$ determines how much the impedance opposes current flow. The phase angle $\theta = \arctan(X/R)$ determines the phase shift between voltage and current.

Reactance comes in two flavors:

- **Inductive reactance** ($X_L > 0$): Causes current to lag voltage. Associated with magnetic energy storage in inductors.
- **Capacitive reactance** ($X_C < 0$): Causes current to lead voltage. Associated with electric energy storage in capacitors.

13.2.4.3.1 Impedance of R, L, and C in AC Circuits

Let's derive the impedance of each fundamental circuit element:

Resistor:

$$Z_R = R$$

For a resistor, voltage and current are always in phase. The impedance is purely real with no reactive component. The resistor behaves the same in AC as in DC (at any instant, $v = iR$).

Inductor:

$$Z_L = j\omega L$$

Recall that for an inductor, $v = L\frac{di}{dt}$. When current is sinusoidal, $i(t) = I_m \sin(\omega t)$, the voltage becomes:

$$v(t) = L\frac{d}{dt}[I_m \sin(\omega t)] = \omega L I_m \cos(\omega t) = \omega L I_m \sin(\omega t + 90°)$$

The voltage leads the current by 90°. The magnitude ratio is ωL, which we call the **inductive reactance**: $X_L = \omega L$. In complex notation, this 90° lead is represented by multiplication by j, giving $Z_L = j\omega L$.

Physical interpretation: Higher frequency (ω) or larger inductance (L) creates more opposition to current changes, increasing reactance.

Capacitor:

$$Z_C = \frac{1}{j\omega C} = -\frac{j}{\omega C}$$

For a capacitor, $i = C\frac{dv}{dt}$. When voltage is sinusoidal, $v(t) = V_m \sin(\omega t)$, the current becomes:

$$i(t) = C\frac{d}{dt}[V_m \sin(\omega t)] = \omega C V_m \cos(\omega t) = \omega C V_m \sin(\omega t + 90°)$$

The current leads the voltage by 90°. Equivalently, voltage lags current by 90°. The magnitude ratio is $\frac{1}{\omega C}$, which we call the **capacitive reactance**: $X_C = -\frac{1}{\omega C}$. In complex notation, this 90° lag is represented by division by j (or multiplication by $-j$), giving $Z_C = \frac{1}{j\omega C}$.

Physical interpretation: Higher frequency (ω) or larger capacitance (C) allows more current to flow, decreasing the magnitude of impedance.

13.2.4.4 Phasor Representation

Earlier, we mentioned that phasor representation is one of three mathematical tools for AC circuit analysis. Now let's dive into what phasors actually are and why they're so powerful.

Working with sinusoidal functions directly in the time domain involves messy trigonometric identities and differential equations. **Phasors** provide an elegant shortcut that converts these differential equations into simple algebra.

Formal definition:

A **phasor** is a complex number that represents the amplitude and phase of a sinusoidal signal, with the implicit understanding that the signal oscillates at a known frequency ω.

For a sinusoidal voltage:

$$v(t) = V_m \sin(\omega t + \phi)$$

The corresponding **phasor** is:

$$\tilde{V} = V_m e^{j\phi} = V_m \angle \phi$$

The phasor captures two pieces of information:

- **Magnitude**: $|\tilde{V}| = V_m$ (the amplitude)
- **Phase angle**: $\angle \tilde{V} = \phi$ (the phase at $t = 0$)

The notation $V_m \angle \phi$ is called **polar form**. We can also write it in **rectangular form** as $\tilde{V} = V_m \cos\phi + jV_m \sin\phi$.

Key insight: In steady-state AC analysis, all voltages and currents oscillate at the same frequency ω. The phasor representation "factors out" the common $e^{j\omega t}$ term, leaving only the magnitude and relative phase. This dramatically simplifies analysis.

From Time Domain to Phasor Domain

The magic of phasors is that **differentiation in the time domain becomes multiplication by $j\omega$ in the phasor domain**.

If $v(t) = V_m \sin(\omega t + \phi)$, then:

$$\frac{dv}{dt} = \omega V_m \cos(\omega t + \phi) = \omega V_m \sin(\omega t + \phi + 90°)$$

In phasor notation, this becomes simply:

$$\frac{d\tilde{V}}{dt} \longleftrightarrow j\omega\tilde{V}$$

The 90° phase shift (from differentiation) is captured by multiplying by j (which rotates a complex number by 90°), and the amplitude scaling by ω appears naturally.

This is why the impedances we derived earlier have the form they do:

- Inductor: $v = L\frac{di}{dt}$ becomes $\tilde{V} = j\omega L\tilde{I}$, so $Z_L = j\omega L$
- Capacitor: $i = C\frac{dv}{dt}$ becomes $\tilde{I} = j\omega C\tilde{V}$, so $Z_C = \frac{1}{j\omega C}$

Visualizing Phasors

Phasors can be visualized as rotating vectors in the complex plane:

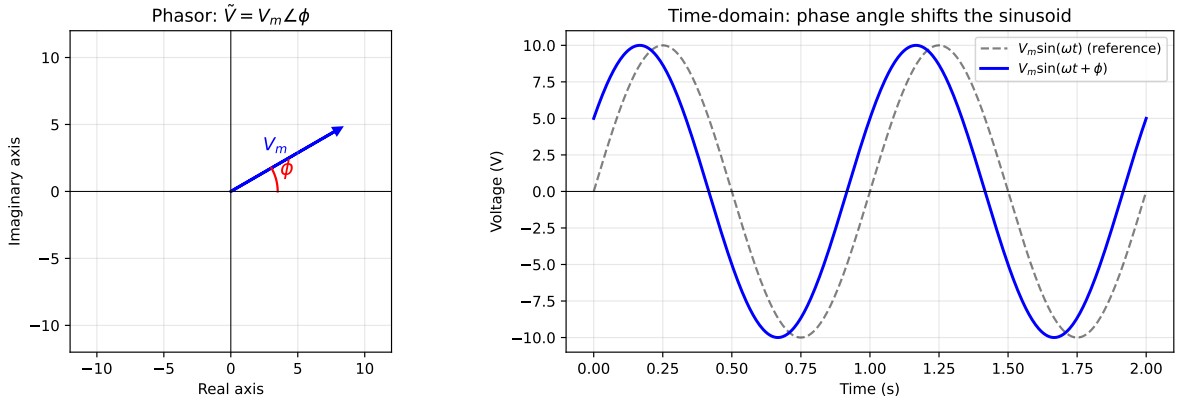


Figure 13.2: Phasor representation: the phasor (left) encodes amplitude and phase angle, which determines the shift of the full sinusoid in the time domain (right)

The phasor captures the amplitude V_m and phase angle ϕ of a sinusoidal signal. In the time domain, a nonzero phase angle ϕ shifts the entire sinusoid earlier (for $\phi > 0$, a "lead") or later (for $\phi < 0$, a "lag") relative to the reference signal $V_m \sin(\omega t)$.

Using Phasors in Circuit Analysis

With phasors, circuit analysis becomes algebraic:

- Ohm's law: $\tilde{V} = \tilde{Z} \cdot \tilde{I}$ (complex multiplication)
- KCL: $\sum \tilde{I}_{in} = \sum \tilde{I}_{out}$ (complex addition)
- KVL: $\sum \tilde{V}_{loop} = 0$ (complex addition)

No differential equations required! We solve for phasor voltages and currents, then convert back to time domain if needed.

13.2.4.5 Complex Representation of AC Quantities

The phasor concept rests on a mathematical foundation: **Euler's formula**, which connects complex exponentials to trigonometric functions.

Euler's Formula:

$$e^{j\theta} = \cos\theta + j\sin\theta$$

This identity allows us to represent sinusoidal functions as the real (or imaginary) part of complex exponentials.

Representing Sinusoids with Complex Exponentials

A sinusoidal voltage $v(t) = V_m \sin(\omega t + \phi)$ can be written as:

$$v(t) = \text{Im}\{V_m e^{j(\omega t+\phi)}\}$$

or equivalently (using $\sin = -\text{Im}\{e^{j\cdot}\}$ or appropriate phase shifts):

$$v(t) = \text{Re}\{V_m e^{j(\omega t+\phi+90°)}\}$$

We can factor this as:

$$v(t) = \text{Im}\{V_m e^{j\phi} \cdot e^{j\omega t}\} = \text{Im}\{\tilde{V} e^{j\omega t}\}$$

where $\tilde{V} = V_m e^{j\phi}$ is the **phasor** we defined earlier!

The Power of Complex Representation

The key insight: when all signals oscillate at the same frequency ω, they all share the common factor $e^{j\omega t}$. We can:

1. **Factor out** the $e^{j\omega t}$ term
2. **Work with** just the complex amplitudes (phasors) $\tilde{V}, \tilde{I}, \tilde{Z}$
3. **Factor back in** the $e^{j\omega t}$ and take the real/imaginary part to get the time-domain result

This transforms differential equations into algebraic equations with complex numbers.

Complex Number Arithmetic

Complex numbers can be expressed in two forms:

Rectangular (Cartesian) form:

$$Z = a + jb$$

where a is the real part and b is the imaginary part.

Polar (magnitude-angle) form:

$$Z = re^{j\theta} = r\angle\theta$$

where $r = |Z| = \sqrt{a^2 + b^2}$ is the magnitude and $\theta = \arctan(b/a)$ is the phase angle.

Conversion between forms:

- Rectangular to polar: $r = \sqrt{a^2 + b^2}$, $\theta = \arctan(b/a)$
- Polar to rectangular: $a = r\cos\theta$, $b = r\sin\theta$

Operations:

- **Addition/subtraction**: Use rectangular form

$$(a_1 + jb_1) + (a_2 + jb_2) = (a_1 + a_2) + j(b_1 + b_2)$$

- **Multiplication/division**: Use polar form

$$r_1\angle\theta_1 \times r_2\angle\theta_2 = (r_1 r_2)\angle(\theta_1 + \theta_2)$$

$$\frac{r_1\angle\theta_1}{r_2\angle\theta_2} = \frac{r_1}{r_2}\angle(\theta_1 - \theta_2)$$

Example: Series RL Circuit

Consider a series RL circuit with $R = 10\,\Omega$, $L = 20$ mH at $f = 60$ Hz.
The total impedance is:

$$Z_{total} = R + Z_L = R + j\omega L$$

With $\omega = 2\pi \cdot 60 \approx 377$ rad/s:

$$Z_{total} = 10 + j(377)(0.02) = 10 + j7.54\,\Omega$$

In polar form:

$$|Z_{total}| = \sqrt{10^2 + 7.54^2} = 12.53\,\Omega$$

$$\angle Z_{total} = \arctan(7.54/10) = 37.0°$$

So $Z_{total} = 12.53\angle 37.0°\,\Omega$.
If we apply a voltage $\tilde{V} = 120\angle 0°$ V, the current is:

$$\tilde{I} = \frac{\tilde{V}}{Z_{total}} = \frac{120\angle 0°}{12.53\angle 37.0°} = 9.58\angle -37.0°\,\text{A}$$

The $-37.0°$ phase means current **lags** voltage by 37°, which makes sense for an inductive load.

13.2.5 Instantaneous Power in AC Circuits

In DC circuits, power is simply $P = VI$, constant over time. In AC circuits, voltage and current vary sinusoidally, so **instantaneous power** $p(t) = v(t) \cdot i(t)$ also varies with time. Understanding this time variation is crucial for comprehending how energy flows in AC systems.

Why instantaneous power matters:

- It reveals the dynamic energy exchange between source and load
- It explains why some power "sloshes back and forth" without being consumed
- It's the foundation for defining active (real) vs. reactive power
- Smart meters measure instantaneous power to calculate energy consumption

Let's visualize how instantaneous power behaves:

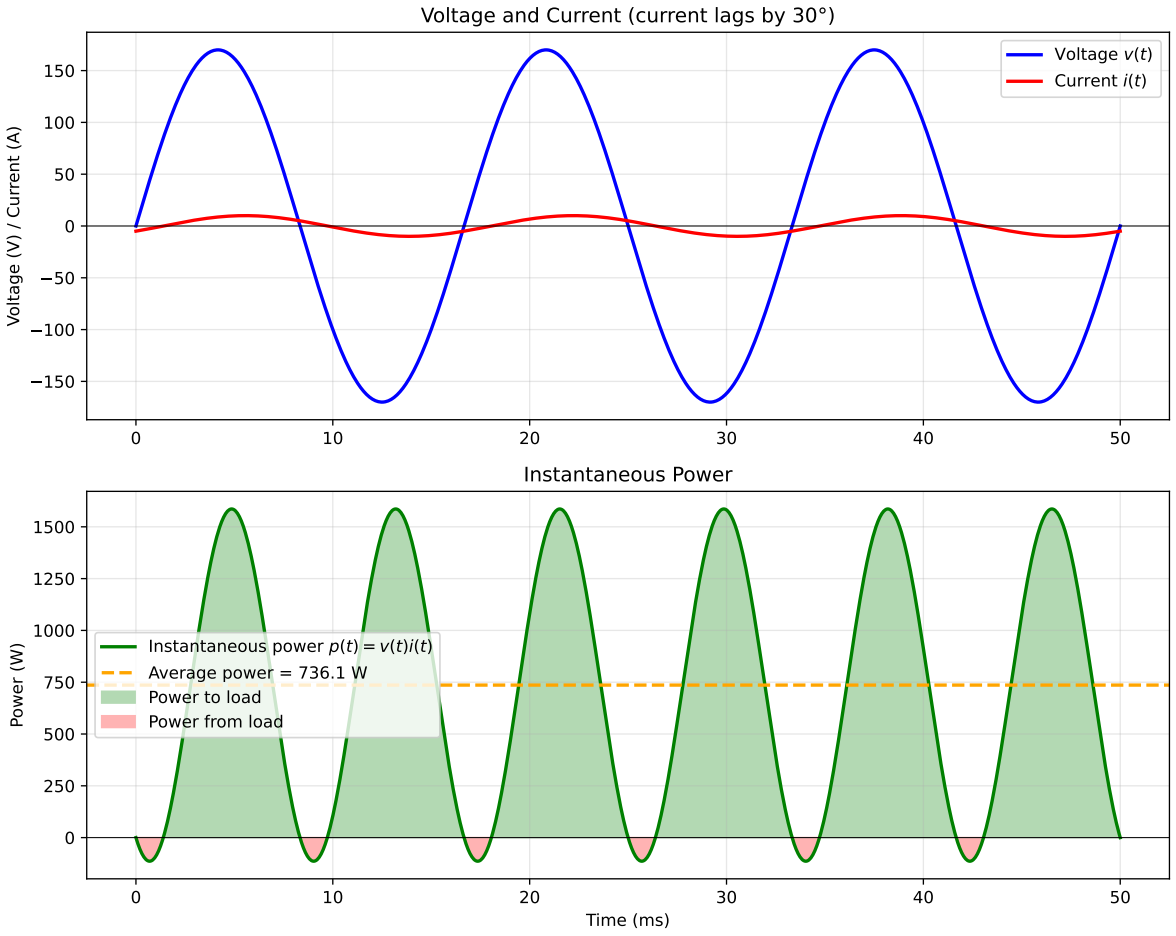


Figure 13.3: Voltage, current, and instantaneous power for an inductive load

Notice that:

1. Power oscillates at **twice** the line frequency (120 Hz for 60 Hz AC)
2. Power can be **negative** (energy flowing back from load to source)
3. The average power is positive (net energy flows to the load)

13.2.5.1 Derivation from Time Domain Expressions

Let's derive the mathematical expression for instantaneous power.

Given:

$$v(t) = V_m \sin(\omega t)$$
$$i(t) = I_m \sin(\omega t - \phi)$$

where ϕ is the phase difference (current lags voltage by ϕ for inductive loads).

The instantaneous power is:

$$p(t) = v(t) \cdot i(t) = V_m \sin(\omega t) \cdot I_m \sin(\omega t - \phi)$$

Using the trigonometric identity:

$$\sin A \sin B = \frac{1}{2}[\cos(A - B) - \cos(A + B)]$$

We get:

$$p(t) = V_m I_m \sin(\omega t) \sin(\omega t - \phi)$$
$$= \frac{V_m I_m}{2}[\cos(\phi) - \cos(2\omega t - \phi)]$$

Rearranging:

$$p(t) = \frac{V_m I_m}{2} \cos(\phi) - \frac{V_m I_m}{2} \cos(2\omega t - \phi)$$

This can be written as:

$$\boxed{p(t) = P_{avg} + P_{osc} \cos(2\omega t - \phi)}$$

where:

- $P_{avg} = \frac{V_m I_m}{2} \cos(\phi)$ is the **average power** (DC component)
- $P_{osc} = \frac{V_m I_m}{2}$ is the **amplitude of oscillation** (AC component at 2ω)

Notice that, of these two, only P_{osc} has a time (t) component.

Expressing instantaneous power in terms of RMS quantities:

Since for sinusoidal waveforms $V_{rms} = \frac{V_m}{\sqrt{2}}$ and $I_{rms} = \frac{I_m}{\sqrt{2}}$, the product $\frac{V_m I_m}{2}$ is simply $V_{rms} I_{rms}$. This lets us rewrite the instantaneous power as:

$$p(t) = V_{rms} I_{rms} \cos(\phi) - V_{rms} I_{rms} \cos(2\omega t - \phi)$$

This form is convenient because RMS values are what meters display and what equipment is rated for. It also makes the connection to the power quantities we define next (active, reactive, apparent) more direct, since those are all expressed in terms of RMS values.

Derivation: RMS of a sinusoidal waveform

The RMS (root-mean-square) value of a periodic signal $x(t)$ with period T is defined as:

$$x_{rms} = \sqrt{\frac{1}{T}\int_0^T x^2(t)\,dt}$$

For a sinusoidal voltage $v(t) = V_m \sin(\omega t)$, we compute each step:

1. Square:

$$v^2(t) = V_m^2 \sin^2(\omega t)$$

2. Mean (average over one period):

Using the trigonometric identity $\sin^2(\theta) = \frac{1-\cos(2\theta)}{2}$:

$$\frac{1}{T}\int_0^T V_m^2 \sin^2(\omega t)\,dt = \frac{V_m^2}{T}\int_0^T \frac{1-\cos(2\omega t)}{2}\,dt$$

$$= \frac{V_m^2}{2T}\left[\int_0^T 1\,dt - \int_0^T \cos(2\omega t)\,dt\right]$$

The first integral evaluates to T. The second integral evaluates to zero because $\cos(2\omega t)$ completes exactly two full cycles over one period $T = \frac{2\pi}{\omega}$, and the positive and negative areas cancel. Therefore:

$$\frac{1}{T}\int_0^T v^2(t)\,dt = \frac{V_m^2}{2}$$

3. Root:

$$V_{rms} = \sqrt{\frac{V_m^2}{2}} = \frac{V_m}{\sqrt{2}}$$

The same result applies to current: $I_{rms} = \frac{I_m}{\sqrt{2}}$. Multiplying:

$$V_{rms} \cdot I_{rms} = \frac{V_m}{\sqrt{2}} \cdot \frac{I_m}{\sqrt{2}} = \frac{V_m I_m}{2}$$

which is exactly the factor that appears in our instantaneous power expression.

13.2.5.2 Power Oscillation at Twice the Line Frequency

Why does power oscillate at 2ω?

Each voltage and current cycle contains both positive and negative half-cycles. When we multiply them:

- Positive voltage × positive current = positive power
- Negative voltage × negative current = positive power (two negatives!)
- Positive voltage × negative current = negative power
- Negative voltage × positive current = negative power

This pattern repeats **twice per AC cycle**, hence the 2ω frequency.

Physical interpretation:

1. **When** $p(t) > 0$: Energy flows from source to load (load absorbs energy)
2. **When** $p(t) < 0$: Energy flows from load back to source (load returns stored energy)
3. **Average over a cycle**: Net energy transfer equals $P_{avg} = \frac{V_m I_m}{2} \cos(\phi)$

Key observations:

- **If** $\phi = 0$ (resistive load): P_{avg} is maximum, power never goes negative, all energy is dissipated
- **If** $\phi = 90°$ (purely reactive load): $P_{avg} = 0$, power oscillates symmetrically around zero, no net energy transfer—energy just sloshes back and forth
- **If** $0 < \phi < 90°$ (typical load): Some energy is dissipated, some oscillates

The oscillating component represents **reactive power**—energy that's temporarily stored in inductors and capacitors, then returned to the source. Notice too that P_{osc}, though time-varying, behaves as a sinusoid which means that over a cycle it has the same amount of positive and negative contributions (i.e., it averages to zero over integer number of cycles).

13.2.6 Active, Reactive, and Apparent Power

Again, the instantaneous power derivation revealed that AC power has two components: a DC (average) component and an oscillating component. This leads us to define three distinct power quantities that characterize AC systems: **active power**, **reactive power**, and **apparent power**.

These three quantities are fundamental to understanding AC power systems, utility billing, and equipment sizing.

13.2.6.1 Active Power (Real Power)

Active power (also called **real power**) is the average power delivered to the load over a complete cycle. This is the power that actually does useful work—it's converted to heat, light, mechanical work, etc.

Definition:

$$P = V_{rms} I_{rms} \cos(\phi)$$

where:

- V_{rms} is the RMS voltage
- I_{rms} is the RMS current
- ϕ is the phase angle between voltage and current
- $\cos(\phi)$ is the **power factor**

Units: Watts (W)

Physical meaning: Active power represents the net energy transfer per unit time from source to load. It's what you pay for on your electric bill (measured in kWh = kilowatt-hours).

From our earlier derivation: We showed that average power is $P_{avg} = \frac{V_m I_m}{2} \cos(\phi)$. Since $V_{rms} = \frac{V_m}{\sqrt{2}}$ and $I_{rms} = \frac{I_m}{\sqrt{2}}$, we have:

$$P = V_{rms} I_{rms} \cos(\phi)$$

Special cases:

- **Resistive load** ($\phi = 0°$): $P = V_{rms} I_{rms}$ (maximum power transfer)
- **Purely reactive load** ($\phi = 90°$): $P = 0$ (no net energy transfer)

13.2.6.2 Reactive Power

Reactive power represents the power that oscillates between source and load without being consumed. It's associated with energy storage in inductors and capacitors.

Definition:

$$Q = V_{rms} I_{rms} \sin(\phi)$$

Units: Volt-Amperes Reactive (VAR)

Physical meaning: Reactive power doesn't do useful work, but it's necessary for:

- Creating magnetic fields in motors and transformers
- Creating electric fields in capacitors
- Maintaining voltage levels in the power system

Sign convention:

- $Q > 0$ (positive): **Inductive reactive power** (current lags voltage, typical for motors, transformers)
- $Q < 0$ (negative): **Capacitive reactive power** (current leads voltage, typical for capacitor banks)

Why it matters: Although reactive power doesn't do work, it requires current to flow in transmission lines, causing resistive losses (I^2R). Utilities often charge industrial customers for excessive reactive power.

13.2.6.3 Apparent Power

Apparent power is the product of RMS voltage and RMS current, without considering the phase angle.

Definition:

$$S = V_{rms} I_{rms}$$

Units: Volt-Amperes (VA)

Physical meaning: Apparent power represents the total power that must be supplied by the source and carried by the wires, regardless of how much actually does useful work. It determines:

- Wire sizing (current carrying capacity)
- Generator/transformer ratings
- Circuit breaker ratings

Equipment is rated in VA (or kVA, MVA) rather than watts because it must handle the total current, not just the portion doing useful work.

13.2.6.4 Power Triangle

The relationship between active, reactive, and apparent power can be visualized geometrically as a **power triangle**:

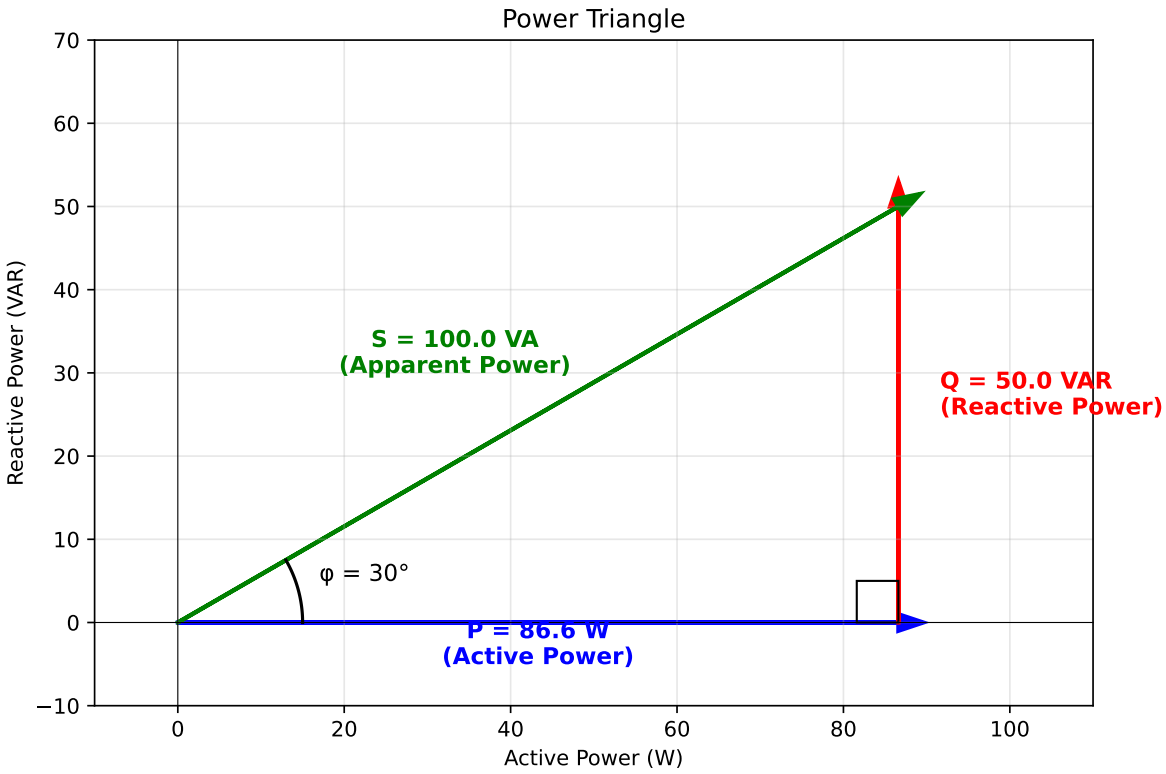


Figure 13.4: Power triangle showing relationship between P, Q, and S

Mathematical relationships:

$$S^2 = P^2 + Q^2$$

$$P = S\cos(\phi)$$

$$Q = S\sin(\phi)$$

$$\phi = \arctan(Q/P)$$

The power triangle is analogous to the impedance triangle ($Z = R + jX$), which is why we use complex notation for power calculations.

13.2.6.5 Power Factor

Power factor (PF) is the ratio of active power to apparent power:

$$\text{PF} = \frac{P}{S} = \cos(\phi)$$

Range: $0 \leq \text{PF} \leq 1$

Interpretation:

- **PF = 1** (unity power factor): Ideal case, all power is active (resistive load)
- **PF = 0**: Worst case, all power is reactive (purely reactive load)
- **PF = 0.7-0.9**: Typical for industrial loads (motors, fluorescent lighting)

Why power factor matters:

1. **Utility billing:** Many utilities charge penalties for low power factor (typically below 0.9) because it forces them to generate and transmit more current for the same useful power
2. **System efficiency:** Low power factor means:
    - Higher current for same active power → higher I^2R losses in wires
    - Larger generators, transformers, and wiring needed
    - Reduced system capacity
3. **Power factor correction:** Capacitor banks are often installed to cancel inductive reactive power, improving power factor

Example: A motor drawing 10 A at 120 V with PF = 0.8:

- Apparent power: $S = 120 \times 10 = 1200$ VA
- Active power: $P = 1200 \times 0.8 = 960$ W (what you pay for)
- Reactive power: $Q = \sqrt{1200^2 - 960^2} = 720$ VAR (wasted current)

Leading vs. Lagging:

- **Lagging power factor** ($Q > 0$): Current lags voltage (inductive load) - most common
- **Leading power factor** ($Q < 0$): Current leads voltage (capacitive load) - less common
- By convention, we often specify PF as "0.8 lagging" or "0.9 leading"

13.2.7 AC Electrical Generators: The Source of Sinusoidal Voltage

We've been working with sinusoidal voltage sources throughout this lecture, treating them as given. But where do these sinusoidal waveforms actually come from? The answer lies in **electromagnetic induction** and the design of AC generators (also called **alternators**).

Understanding generator operation helps explain:

- Why AC power is naturally sinusoidal
- The relationship between mechanical rotation speed and electrical frequency
- How the grid's frequency is maintained
- Why three-phase power is used for generation and transmission

13.2.7.1 Basic Principles of Electromagnetic Induction

The fundamental principle behind all electrical generators is **Faraday's Law of Electromagnetic Induction**:

$$\mathcal{E} = -\frac{d\Phi_B}{dt}$$

where:

- $\mathcal{E}$ is the induced electromotive force (EMF), or voltage
- Φ_B is the magnetic flux through a conductor
- The negative sign indicates the direction (Lenz's law)

In plain language: When the magnetic flux through a conductor changes, a voltage is induced in that conductor.

How generators use this principle:

1. Create a strong magnetic field (using permanent magnets or electromagnets)
2. Rotate a conductor (coil of wire) through this magnetic field
3. As the coil rotates, the magnetic flux through it changes continuously
4. This changing flux induces a voltage in the coil

Why the output is sinusoidal:

When a coil rotates at constant angular velocity ω in a uniform magnetic field, the flux through it varies as:

$$\Phi_B(t) = \Phi_{max} \cos(\omega t)$$

Taking the derivative:

$$\mathcal{E}(t) = -\frac{d}{dt}[\Phi_{max}\cos(\omega t)] = \omega\Phi_{max}\sin(\omega t)$$

The induced voltage is naturally sinusoidal. The mechanical rotation creates an electrical sine wave.

13.2.7.2 Components of an AC Generator

An AC generator consists of these main parts:

1. **Stator** (stationary part):
   - The outer shell with coils of wire (windings) where voltage is induced
   - In large generators, this is the component that outputs electrical power
   - Made of laminated steel to reduce eddy current losses
2. **Rotor** (rotating part):
   - The spinning component that creates the rotating magnetic field
   - Contains electromagnets (field windings) powered by DC current
   - Mechanically coupled to a turbine (steam, hydro, wind, etc.)
3. **Field windings**:
   - Coils on the rotor that create the magnetic field
   - Powered by DC current through slip rings or a separate exciter
   - The strength of this field determines the output voltage magnitude
4. **Armature windings**:
   - Coils in the stator where AC voltage is induced
   - Connected to the output terminals
   - Typically three separate sets of windings for three-phase output
5. **Mechanical drive system**:
   - Steam turbine (coal, nuclear, natural gas plants)
   - Hydro turbine (hydroelectric dams)
   - Wind turbine (wind farms)
   - Diesel/gas engine (backup generators)
6. **Voltage regulator**:
   - Controls the DC current to the field windings
   - Adjusts output voltage to maintain desired level (e.g., 13.8 kV)

13.2.7.3 Single-Phase Generator Operation

Let's trace how mechanical rotation creates electrical voltage in the simplest case: a **single-phase generator**.

Setup: A rectangular coil with N turns rotates in a uniform magnetic field B at angular velocity ω.

The rotating coil:

- As the coil rotates, the angle $\theta = \omega t$ between the coil's normal and the magnetic field changes
- The magnetic flux through the coil is: $\Phi_B = NBA\cos(\omega t)$ where A is the coil area

Induced voltage (from Faraday's law):

$$v(t) = -N\frac{d\Phi_B}{dt} = NBA\omega\sin(\omega t) = V_m\sin(\omega t)$$

where $V_m = NBA\omega$ is the peak voltage.

Key insights:

- The amplitude V_m depends on: magnetic field strength (B), number of turns (N), coil area (A), and rotation speed (ω)
- The frequency is $f = \omega/(2\pi)$, directly tied to rotation speed
- One complete mechanical rotation = one complete electrical cycle

Typical values: For a 60 Hz generator:

- Rotation speed: 3600 RPM (for 2-pole machine) or 1800 RPM (for 4-pole machine)
- Output voltage: Depends on design, typically 13.8 kV or 18 kV for utility generators

13.2.7.4 Three-Phase Generators

While single-phase generation is conceptually simple, virtually all large-scale electrical power generation uses **three-phase systems**. A three-phase generator has three separate sets of windings in the stator, positioned 120° apart mechanically.

Physical arrangement:

- Three identical coils (windings) placed around the stator
- Each coil is offset by 120° (one-third of a full rotation)
- The same rotating magnetic field passes through all three coils
- Each coil "sees" the magnetic field at a different time in the rotation cycle

Why 120° spacing?

- Dividing 360° by 3 gives 120°
- This ensures balanced, symmetric power generation
- The three phases are identical in magnitude but shifted in time

13.2.7.4.1 Voltage Expression for Each Phase

As the rotor spins, it induces voltage in each of the three windings. Because the windings are spaced 120° apart, the voltage waveforms are also phase-shifted by 120°:

Phase A (reference phase):

$$v_A(t) = V_m \sin(\omega t)$$

Phase B (lags Phase A by 120°):

$$v_B(t) = V_m \sin(\omega t - 120°)$$

Phase C (lags Phase A by 240°, or equivalently, leads by 120°):

$$v_C(t) = V_m \sin(\omega t - 240°) = V_m \sin(\omega t + 120°)$$

Key property: At any instant, the three phase voltages sum to zero:

$$v_A(t) + v_B(t) + v_C(t) = 0$$

This can be verified using trigonometric identities, or visualized using phasor diagrams where the three phasors form an equilateral triangle.

13.2.7.4.2 Advantages of Three-Phase Power

Three-phase systems offer significant advantages over single-phase:

1. Constant power delivery:

- In single-phase AC, instantaneous power oscillates (as we saw earlier)
- In three-phase with balanced loads, instantaneous total power is constant!

For a balanced three-phase system:

$$p_{total}(t) = p_A(t) + p_B(t) + p_C(t) = 3V_{rms}I_{rms}\cos(\phi) = \text{constant}$$

This means no pulsating torque in motors, smoother operation of equipment.

2. More efficient transmission:

- Three-phase transmits more power with less conductor material
- For the same power delivered, three-phase requires less copper/aluminum than single-phase
- Mathematically, three-phase power: $P_{3\phi} = \sqrt{3} V_L I_L \cos(\phi)$ where V_L is line voltage (where the $\sqrt{3}$ comes from $\sin(120°) = \frac{\sqrt{3}}{2}$)

3. **Enables rotating magnetic fields:**

- Motors can start and run without additional components
- Three-phase motors are simpler, more reliable, and more efficient
- The three phase-shifted currents naturally create a rotating magnetic field

4. **Voltage options:**

- Provides both line-to-line voltage ($\sqrt{3} \times V_{phase}$) and line-to-neutral voltage
- Example: 208Y/120V system provides 208V for large loads and 120V for small loads

5. **Better voltage regulation:**

- Easier to maintain voltage stability under varying loads
- Load balancing across three phases reduces voltage drops

Applications:

- **Power generation**: All utility-scale generators are three-phase
- **Transmission**: Long-distance power lines use three-phase
- **Industrial motors**: Almost always three-phase for >1 HP
- **Data centers**: Three-phase for efficient high-power distribution
- **Residential**: Typically stepped down to single-phase at the transformer, though some larger homes have three-phase service

⚠ 3-Phase Voltage Visualization

Let's now visualize three-phase voltages to understand their relationship and verify that they sum to zero.

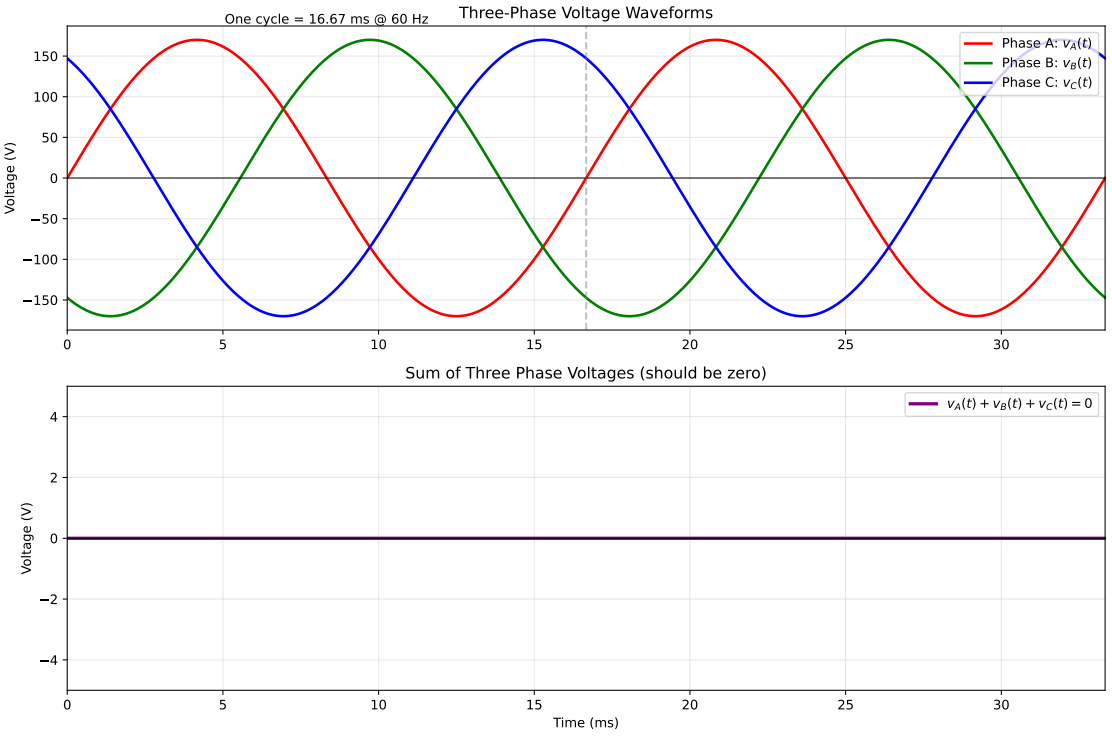


Figure 13.5: Three-phase voltage waveforms and their sum

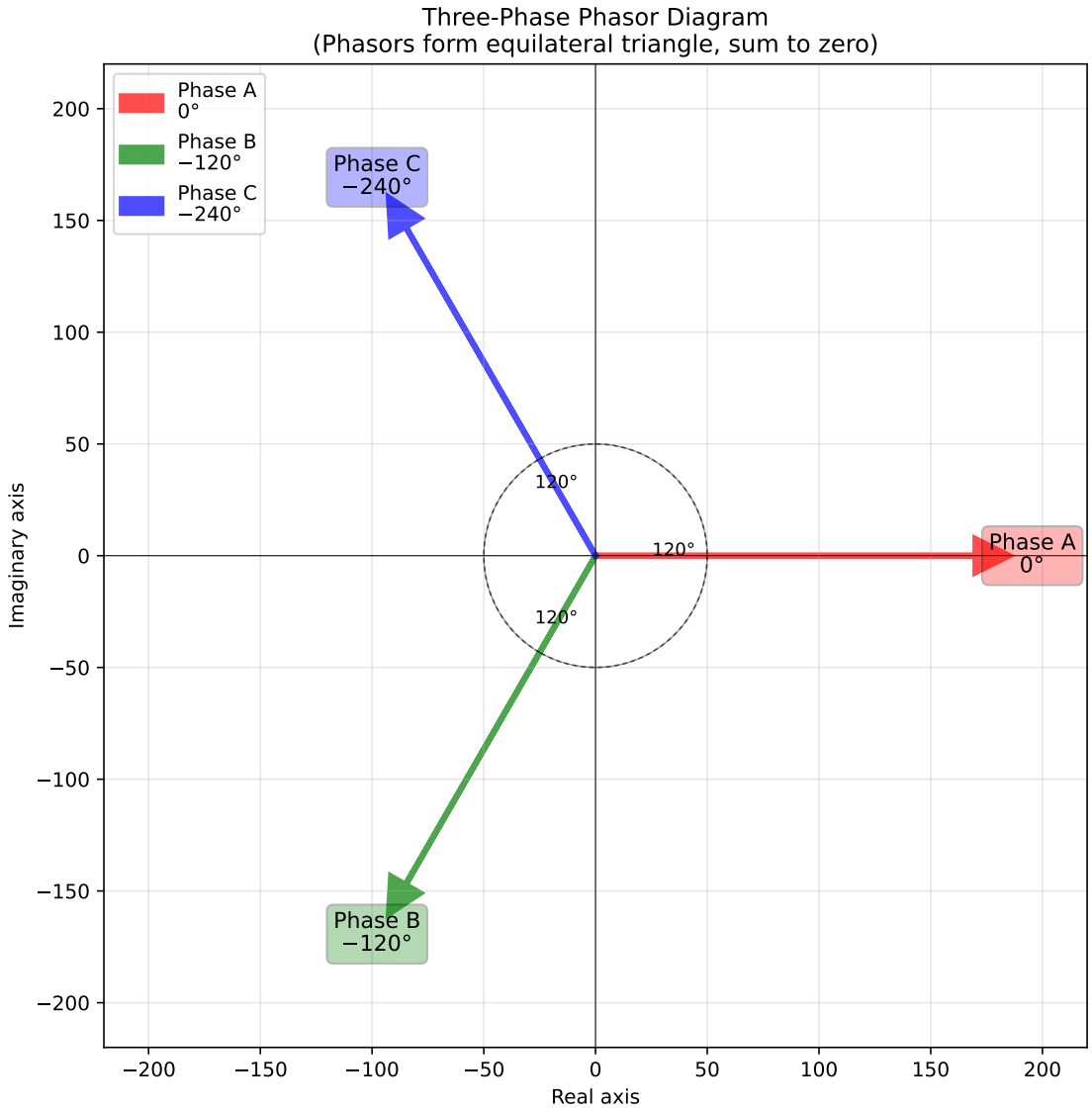


Figure 13.6: Phasor diagram showing 120° phase relationships

Observations:

1. **Waveforms**: The three sinusoids are identical in shape and amplitude, but shifted by 120° in time
2. **Sum is zero**: The bottom plot shows that at every instant, $v_A + v_B + v_C = 0$ (within numerical precision)

3. **Phasor diagram**: The three phasors form an equilateral triangle, with their vector sum equal to zero
4. **Physical meaning**: This zero-sum property means that in a balanced three-phase system, no return (neutral) conductor is theoretically needed—the currents balance out

13.2.8 Power Balance and Grid Frequency

One of the most fascinating aspects of AC power systems is the tight coupling between mechanical rotation, electrical frequency, and power balance. Understanding this relationship is crucial for grid stability and building automation that interacts with the grid.

13.2.8.1 Generator Frequency and Rotation Speed

The electrical frequency of an AC generator is directly tied to its mechanical rotation speed through this relationship:

$$f = \frac{p \cdot n}{120}$$

where:

- f is the electrical frequency in Hz
- p is the number of **magnetic poles** in the generator
- n is the rotation speed in RPM (revolutions per minute)
- The factor 120 converts from RPM to Hz for a 60-cycle basis

Why this matters:

For a **2-pole** generator at 60 Hz:

$$n = \frac{120 \cdot f}{p} = \frac{120 \cdot 60}{2} = 3600 \text{ RPM}$$

For a **4-pole** generator at 60 Hz:

$$n = \frac{120 \cdot 60}{4} = 1800 \text{ RPM}$$

Practical implications:

- **High-speed turbines** (gas turbines, small hydro): Use 2-pole generators (3600 RPM @ 60 Hz)

- **Medium-speed turbines** (large steam turbines): Use 4-pole generators (1800 RPM @ 60 Hz)
- **Low-speed turbines** (hydroelectric): Use many poles (e.g., 40 poles → 180 RPM @ 60 Hz)

The more poles, the slower the required rotation speed for a given frequency. This is why massive hydroelectric generators can spin slowly while still producing 60 Hz power.

13.2.8.2 Power-Frequency Coupling

Here's a critical insight: **Grid frequency is a real-time indicator of the balance between power generation and power consumption.**

The physics:

Generators are massive rotating machines with significant rotational inertia (J). When there's a power imbalance:

1. **Load > Generation** (more demand than supply):
   - Electrical load acts as a brake on the generator
   - Rotor slows down slightly
   - Frequency drops (e.g., from 60.00 Hz to 59.95 Hz)
2. **Generation > Load** (more supply than demand):
   - Less load means less braking force
   - Rotor speeds up slightly
   - Frequency rises (e.g., from 60.00 Hz to 60.05 Hz)

The equation (simplified):

The rate of frequency change is proportional to the power imbalance:

$$\frac{df}{dt} \propto \frac{P_{gen} - P_{load}}{J_{total}}$$

where J_{total} is the total rotational inertia of all connected generators.

What you can observe:

Modern smart meters and frequency monitors can detect these tiny frequency fluctuations. A sudden large load (like a steel mill starting up) causes a measurable frequency dip across the entire interconnected grid.

13.2.8.3 Frequency Regulation

Grid operators work continuously to maintain frequency at exactly 60.00 Hz (in the US) or 50.00 Hz (in Europe). This is called **frequency regulation**.

Why precise frequency control matters:

1. **Equipment timing**: Electric clocks, some motors, and industrial processes depend on frequency
2. **Generator synchronization**: All generators on the grid must stay synchronized
3. **System stability**: Large frequency deviations can trigger automatic disconnections and blackouts
4. **Power quality**: Sensitive electronics can malfunction with frequency variations

How frequency is regulated:

The grid uses a hierarchy of control mechanisms:

1. **Primary control (seconds):** Automatic governor response

   - Governors on generators automatically adjust steam/water flow when frequency changes
   - No human intervention needed
   - Responds within seconds
   - Example: Frequency drops → governor opens steam valve → more power → frequency recovers

2. **Secondary control (minutes):** Automatic Generation Control (AGC)

   - Centralized computer systems monitor frequency
   - Send set-point adjustments to generators
   - Balances load across generators economically
   - Responds within minutes

3. **Tertiary control (minutes to hours):** Economic dispatch

   - Bring additional generators online or take them offline
   - Coordinate with power markets
   - Plan for predicted demand changes
   - Human operators involved

Typical operating range:

- Normal: 59.98 - 60.02 Hz (±0.02 Hz)
- Acceptable: 59.95 - 60.05 Hz (±0.05 Hz)
- Alarm: 59.90 - 60.10 Hz (±0.10 Hz)
- Emergency actions: Outside ±0.10 Hz

Implications for building automation:

- **Demand response**: Buildings can help stabilize the grid by reducing load when frequency drops
- **Frequency-based load shedding**: Some building systems automatically shed non-critical loads during frequency events
- **Battery storage**: Can inject or absorb power to help regulate frequency (very fast response)
- **Real-time monitoring**: Smart building systems can monitor grid frequency and adjust operations accordingly

This tight coupling between mechanical rotation, electrical frequency, and power balance is unique to synchronous AC systems and is one reason why grid operation is so complex and fascinating!

13.2.9 Power Transmission and Distribution

Power generated at a power plant must travel potentially hundreds of miles to reach buildings. The electrical grid that accomplishes this is one of humanity's greatest engineering achievements, and understanding it is essential for anyone working on building energy systems.

13.2.9.1 Overview of the Power System

The electrical grid operates at multiple voltage levels, each optimized for a specific purpose. Power flows through these stages:

1. **Generation** (typically 13.8 kV to 25 kV)
   - Power plants generate three-phase AC at medium voltages
   - Generator output: typically 13.8 kV, 18 kV, or 22 kV
   - These voltages are manageable for the generator equipment
2. **Step-up Transmission** (115 kV to 765 kV)
   - Transformers boost voltage to very high levels for long-distance transmission
   - Common transmission voltages in the US:
     - 115 kV (local transmission)
     - 230 kV, 345 kV, 500 kV (regional transmission)
     - 765 kV (extra-high voltage for long distances)
   - High voltage dramatically reduces transmission losses
3. **Step-down to Distribution** (4 kV to 35 kV)

- Substations near cities step voltage down to medium levels
- Common distribution voltages: 12.47 kV, 13.8 kV, 34.5 kV
- This voltage level can be safely routed on utility poles through neighborhoods

4. **Final Step-down to Service** (120 V / 240 V residential, 208 V / 480 V commercial)

- Pole-mounted or pad-mounted transformers provide final voltage reduction
- Residential: 240V/120V split-phase service
- Commercial/Industrial: 208V/120V or 480V/277V three-phase service

Why multiple voltage levels?

The fundamental reason is **minimizing transmission losses** while maintaining safety:

For a given power $P = VI$, we can use:

- High voltage, low current → minimal I^2R losses in transmission lines
- Low voltage, higher current → safe for buildings and equipment

Example calculation

Transmit 100 MW over 100 miles with wire resistance of 0.1 Ω/mile (total $R = 10\,\Omega$):
At **10 kV**: $I = P/V = 100 \times 10^6/10 \times 10^3 = 10{,}000$ A

- Line loss: $I^2R = (10{,}000)^2 \times 10 = 1{,}000{,}000{,}000$ W = 1000 MW (more power lost than transmitted!)

At **500 kV**: $I = 100 \times 10^6/500 \times 10^3 = 200$ A

- Line loss: $I^2R = (200)^2 \times 10 = 400{,}000$ W = 0.4 MW (0.4% loss - acceptable)

This dramatic difference is why high-voltage transmission is essential.

13.2.9.2 The Role of Transformers

Transformers are the key technology that makes AC power distribution practical. They efficiently change voltage levels using electromagnetic induction, with no moving parts and typical efficiencies of 95-99%.

How transformers work:

A transformer consists of two coils (windings) wrapped around a common iron core:

- **Primary winding**: Connected to the input (source)
- **Secondary winding**: Connected to the output (load)

- **Iron core**: Provides a path for magnetic flux to link the two windings

When AC current flows in the primary, it creates a changing magnetic flux in the core. This changing flux induces voltage in the secondary winding (Faraday's law). The voltage transformation depends on the ratio of turns in each winding.

13.2.9.2.1 Transformer Basics

For an **ideal transformer** (no losses):

Voltage transformation:

$$\frac{V_2}{V_1} = \frac{N_2}{N_1}$$

where:

- V_1, V_2 are primary and secondary voltages
- N_1, N_2 are number of turns in primary and secondary windings

Power conservation (ideal case):

$$P_1 = P_2$$
$$V_1 I_1 = V_2 I_2$$

Therefore, **current transformation:**

$$\frac{I_2}{I_1} = \frac{N_1}{N_2} = \frac{V_1}{V_2}$$

Key insights:

- **Step-up transformer** ($N_2 > N_1$): Increases voltage, decreases current
- **Step-down transformer** ($N_2 < N_1$): Decreases voltage, increases current
- Power (approximately) conserved: high voltage × low current = low voltage × high current
- Impedance transformation: $Z_2 = Z_1(N_2/N_1)^2$

Example

A transformer steps 13.8 kV down to 240 V with $N_1 = 5750$ turns.
Turns ratio: $N_2/N_1 = V_2/V_1 = 240/13{,}800 = 1/57.5$
So $N_2 = 5750/57.5 = 100$ turns
If the secondary supplies 100 A to a load:

$$I_1 = I_2 \cdot \frac{N_2}{N_1} = 100 \cdot \frac{1}{57.5} = 1.74 \text{ A}$$

Real transformers:

- Have losses (copper losses in windings, core losses in iron)
- Typical efficiency: 95-99% depending on size and load
- Larger transformers are more efficient
- Operate only on AC (the changing current is essential)

13.2.9.2.2 Voltage Levels in the Grid

Here's a typical voltage cascade from power plant to your wall outlet:

Stage	Voltage	Current (for 100 MW)	Purpose
Generator output	18 kV	5,556 A	Generation
Transmission (step-up)	500 kV	200 A	Long-distance transport
Subtransmission	115 kV	870 A	Regional distribution
Distribution	12.47 kV	8,020 A	Neighborhood distribution
Service (residential)	240 V	417 kA	Individual buildings

Notice how current decreases dramatically as voltage increases, minimizing I^2R losses in the transmission lines.

Why these specific voltages?

The voltage levels have evolved over time based on:

- **Technical constraints**: Insulation capabilities, transformer design
- **Economic optimization**: Balance between equipment cost and transmission efficiency
- **Safety considerations**: Lower voltages are inherently safer
- **Historical standards**: Once established, voltages are difficult to change due to infrastructure investment

13.2.9.3 Split-Phase Systems for Residential Buildings

Most residential buildings in North America receive power through a **split-phase** service. This clever system provides both 120V for small loads (lights, outlets) and 240V for large loads (electric dryers, ovens, water heaters, HVAC) using a single transformer.

Understanding split-phase service is essential for building energy monitoring and automation.

13.2.9.3.1 Center-Tapped Transformer

The split-phase system uses a **center-tapped transformer** on the utility pole or pad near your home.

Configuration:

- **Primary side**: Connected to medium-voltage distribution line (e.g., 12.47 kV)
- **Secondary side**: 240V total, with a **center tap** creating two 120V "legs"
  - Hot leg 1 (L1): +120V relative to neutral
  - Neutral (N): 0V (center tap, grounded)
  - Hot leg 2 (L2): +120V relative to neutral
  - Between L1 and L2: 240V

The mathematics:

Each leg of the secondary provides 120 V RMS relative to neutral, which corresponds to a peak voltage of $V_m = 120\sqrt{2} \approx 170$ V. The two legs are 180° out of phase:

- $v_{L1}(t) = +170\sin(\omega t)$ (120 V RMS relative to neutral)
- $v_{L2}(t) = -170\sin(\omega t)$ (120 V RMS relative to neutral, **opposite phase**)
- $v_{L1-L2}(t) = v_{L1}(t) - v_{L2}(t) = 340\sin(\omega t)$ (240 V RMS between L1 and L2)

The key: **L1 and L2 are 180° out of phase**. When L1 is at +120V, L2 is at -120V, so the voltage between them is 240V.

Why this configuration?

- **Efficiency**: Provides two voltages with a single transformer
- **Safety**: Most circuits at safer 120V, only large loads at 240V
- **Flexibility**: Can power both small and large appliances
- **Balance**: Loads can be split between L1 and L2 to balance current

13.2.9.3.2 120V and 240V Circuits

120V circuits (between one hot leg and neutral):

- Standard outlets
- Lighting
- Small appliances (toasters, coffee makers, TVs)
- Electronics
- Most HVAC controls
- Uses: L1-N or L2-N

240V circuits (between both hot legs):

- Electric range/oven
- Electric dryer
- Electric water heater
- Central air conditioning
- Electric vehicle chargers
- Heat pumps
- Uses: L1-L2 (no neutral needed for pure 240V loads)

240V circuits with neutral (L1-L2-N):

- Electric ranges (240V for heating elements, 120V for controls/lights)
- Electric dryers (240V for heating, 120V for drum motor/controls)
- Provides both voltages to a single appliance

13.2.9.3.3 Panel Configuration

The **electrical panel** (breaker box) distributes power from the service entrance to individual circuits.

Physical layout:

- **Main breaker**: 200A typical for modern homes (or 100A, 150A for older/smaller homes)
- **Bus bars**: Two vertical bars carrying L1 and L2
- **Branch circuit breakers**: Snap onto bus bars in alternating pattern
  - Odd positions connect to L1
  - Even positions connect to L2
  - This alternating pattern helps balance loads

Breaker types:

- **Single-pole (120V)**: Takes one slot, connects one hot leg to neutral
  - Width: 1 slot
  - Rating: typically 15A or 20A
  - Example: bedroom outlets, lighting
- **Double-pole (240V)**: Takes two adjacent slots, connects both legs
  - Width: 2 slots
  - Rating: typically 30A, 40A, or 50A
  - Example: electric dryer (30A), range (40-50A)

Load balancing:

Ideally, power drawn from L1 and L2 should be approximately equal. If unbalanced:

- One leg could be overloaded while the other is underutilized
- Neutral wire carries the **difference** between L1 and L2 currents
- Excessive neutral current can cause voltage imbalances

Example:

- L1 supplies: 50A
- L2 supplies: 30A
- Neutral current: $|50 - 30| = 20$A (the imbalance)

For a perfectly balanced load (L1 = L2), neutral current is zero—this is why three-phase systems often don't need a neutral wire.

13.2.10 Analyzing AC Power Usage: Measurement and Calculation

The mathematical framework we developed — phasors, impedance, active/reactive/apparent power — assumes perfectly sinusoidal signals in steady state. In practice, the voltages and currents we measure in real buildings deviate from this ideal in several ways. Non-linear loads such as switch-mode power supplies, variable-frequency drives, and LED drivers inject **harmonic distortion**, making waveforms non-sinusoidal. Load switching, motor startups, and grid events introduce **transients** that violate the steady-state assumption. Even the grid voltage itself fluctuates due to sag, swell, and frequency drift.

Beyond the signals themselves, the measurement process introduces its own challenges. Analog-to-digital converters (ADCs) impose **quantization error** — the continuous signal is rounded to discrete levels. If the **sampling rate** is not high enough relative to the signal's frequency content, aliasing can corrupt the measurement. Voltage and current sensors add noise, offset, and calibration drift. All of these factors mean that computing quantities like V_{rms}, power factor, or active power from real data requires care.

The mathematical definitions are still the right foundation — they tell us *what* to compute and *why*. But applying them to measured data requires understanding *how* to compute them robustly. The following subsections walk through the practical approaches, starting from raw high-frequency waveform samples and then moving to the simpler case where cycle-level statistics (like RMS values) are already available.

13.2.10.1 Measuring Voltage and Current

In most commercial energy monitors and smart meters, the power calculations we described earlier are handled by dedicated **power measurement integrated circuits** (ICs). Chips such as the ADE9000 (Analog Devices) or ATM90E26 (Microchip) accept raw analog voltage and current inputs from sensors, digitize them internally, and output pre-computed quantities: V_{rms}, I_{rms}, active power, reactive power, apparent power, power factor, and line frequency.

The engineer configures the IC and reads registers — no manual FFT or zero-crossing detection required.

That said, understanding how these quantities are derived from raw measurements remains valuable. It helps you interpret IC datasheets critically (e.g., what does "0.1% accuracy on active power" mean, and under what conditions?), diagnose anomalies when readings don't match expectations, and choose the right sensor and sampling configuration for a given application. It is also essential for research applications — such as non-intrusive load monitoring (NILM) — where you work directly from high-frequency waveform samples and need full control over the signal processing pipeline.

13.2.10.2 Calculating Power from Measurements

We now review how to compute the power quantities defined earlier — active, reactive, and apparent power — but starting from actual measured data rather than idealized mathematical expressions. We consider two scenarios that represent different "entry points" into the same power triangle. First, we start from **raw voltage and current samples** acquired at a rate high enough to resolve the within-cycle waveform shape. This is the more general (and more involved) case, typical of research instrumentation and custom data acquisition. Second, we consider the simpler case where **cycle-level statistics** such as V_{rms}, I_{rms}, and phase angle are already available — the common situation when using a metering IC or commercial power analyzer.

13.2.10.2.1 From Sub-Cycle Measurements

Suppose we have discrete samples of voltage and current, $v[n]$ and $i[n]$, acquired at a sampling rate f_s (e.g., several kHz — high enough to capture the waveform shape within each AC cycle). Our goal is to extract the power triangle quantities from these raw samples.

Step 1: Estimate line frequency from voltage zero-crossings

The first task is to identify cycle boundaries. We do this by detecting **upward zero-crossings** of the voltage signal — instants where $v[n]$ transitions from negative to positive. Voltage zero-crossings are preferred over current zero-crossings because the voltage waveform is typically much closer to a pure sinusoid (it is set by the grid, not distorted by individual loads), making the crossings sharper and less susceptible to noise.

The time between two consecutive upward zero-crossings gives the period T, and hence the line frequency $f = 1/T$. In the US grid, we expect $T \approx 16.67$ ms ($f \approx 60$ Hz), though the actual value fluctuates slightly. These zero-crossings also define the boundaries of each cycle for the calculations that follow.

Step 2: Compute instantaneous and average (active) power

Instantaneous power at each sample is simply the product:

$$p[n] = v[n] \cdot i[n]$$

The **active power** (average power) over one complete cycle of N samples is:

$$P = \frac{1}{N} \sum_{n=0}^{N-1} p[n]$$

It is important to sum over **complete cycles** — starting and ending at the zero-crossings identified in Step 1. Averaging over a fractional cycle would allow the oscillating component to leak into the estimate, biasing the result.

Step 3: Compute RMS values and apparent power

The RMS voltage and current over the same complete cycle are:

$$V_{rms} = \sqrt{\frac{1}{N} \sum_{n=0}^{N-1} v[n]^2} \qquad I_{rms} = \sqrt{\frac{1}{N} \sum_{n=0}^{N-1} i[n]^2}$$

From these, the **apparent power** is:

$$S = V_{rms} \cdot I_{rms}$$

Step 4: Estimate the phase difference

The phase angle ϕ between voltage and current can be estimated from the **time delay between their respective zero-crossings**. If the voltage upward zero-crossing occurs at time t_v and the nearest current upward zero-crossing occurs at time t_i, then:

$$\phi \approx \omega \cdot (t_v - t_i) = 2\pi f \cdot (t_v - t_i)$$

A positive ϕ (current zero-crossing *after* voltage) indicates current lagging voltage (inductive load); a negative ϕ indicates current leading (capacitive load). Note that current zero-crossings can be noisier than voltage zero-crossings, especially for loads with significant harmonic distortion, so this estimate may need filtering or averaging over several cycles.

Step 5: Close the power triangle

With active power P from Step 2 and apparent power S from Step 3, we can compute the remaining quantities:

$$Q = \sqrt{S^2 - P^2}$$

The **sign** of Q is determined by the phase difference from Step 4: positive for inductive (lagging), negative for capacitive (leading). The power factor follows directly:

$$\text{PF} = \frac{P}{S}$$

We now have the complete power triangle — P, Q, S, and PF — derived entirely from raw voltage and current samples.

A note on redundancy and uncertainty: Strictly speaking, the power triangle is defined by three quantities (P, Q, S) linked by the constraint $S^2 = P^2 + Q^2$, so only **two independent estimates** are needed to determine the third. In the steps above, we computed P (from instantaneous power), S (from RMS values), and ϕ (from zero-crossings) — more information than the triangle requires. In principle, any pair would suffice: P and S, or P and ϕ, or S and ϕ. However, each of these estimates carries its own measurement uncertainty — from quantization, sensor noise, waveform distortion, and finite sample windows. "Closing" the triangle from different pairs of quantities will generally yield slightly different results for the third. Formally, if we could quantify the uncertainty in each estimate, we could choose the pair that minimizes the propagated uncertainty in the derived quantity. In practice, P (from the average of $v[n] \cdot i[n]$) tends to be the most robust estimate because the sample-by-sample product and averaging operation is relatively insensitive to noise, while the phase angle ϕ from zero-crossing differences tends to be the least reliable, especially for distorted current waveforms.

Practical considerations

All of the above assumes approximately **steady-state conditions** over the analysis window. In a real building, loads switch on and off, motors ramp up, and the grid voltage fluctuates. Keeping the analysis window short — typically a few cycles (e.g., 3–10 cycles, or roughly 50–170 ms at 60 Hz) — helps ensure that the steady-state assumption holds reasonably well within each window. Longer windows improve noise averaging but risk blurring transient events.

13.2.10.2.2 From RMS Measurements

When a metering IC or power analyzer has already computed V_{rms}, I_{rms}, and the phase angle ϕ for us, obtaining the power triangle quantities is straightforward — we simply apply the definitions directly:

$$P = V_{rms}\, I_{rms} \cos(\phi) \qquad Q = V_{rms}\, I_{rms} \sin(\phi) \qquad S = V_{rms}\, I_{rms}$$

$$\text{PF} = \frac{P}{S} = \cos(\phi)$$

Worked example

A smart meter reports $V_{rms} = 121.3$ V, $I_{rms} = 8.4$ A, and $\phi = 22°$ (current lagging) for a residential circuit. Then:

- $S = 121.3 \times 8.4 = 1018.9$ VA
- $P = 1018.9 \times \cos(22°) = 944.7$ W
- $Q = 1018.9 \times \sin(22°) = 381.5$ VAR (inductive)
- PF $= 0.927$ lagging

The same steady-state caveat applies here: these RMS values are only meaningful if the underlying voltage and current were approximately constant over the measurement window. Most metering ICs update their registers every cycle or every few cycles (roughly 16–50 ms at 60 Hz), which is short enough for typical steady-state loads. However, if loads are switching rapidly — for example, a compressor cycling on — the reported RMS values may represent a blend of two different operating states. When precise transient analysis is needed, the sub-cycle approach described above gives finer temporal resolution.

13.2.11 VI Trajectory: Visualizing Load Characteristics

So far we have analyzed AC power using time-series plots — voltage and current as functions of time. But there is another way to visualize a load's electrical behavior that is often more revealing: plotting current *against* voltage over one complete AC cycle. The resulting curve is called a **VI trajectory** (or VI signature), and its shape encodes the load's impedance characteristics at a glance.

Use the interactive tool below to build intuition. Adjust the resistance R and reactance X of a load and observe how the VI trajectory changes shape.

```
//| echo: false

viewof Vrms = Inputs.range([80, 140], {value: 120, step: 1, label: "V_rms (V)"})
viewof R = Inputs.range([1, 200], {value: 50, step: 1, label: "Resistance R (Ω)"})
viewof X = Inputs.range([-200, 200], {value: 0, step: 1, label: "Reactance X (Ω)"})
```

```
//| echo: false

// Compute derived quantities
Z_mag = Math.sqrt(R * R + X * X)
phi_rad = Math.atan2(X, R)
phi_deg = phi_rad * 180 / Math.PI
Irms = Vrms / Z_mag
```

```
PF = Math.cos(phi_rad)
P_active = Vrms * Irms * PF
Q_reactive = Vrms * Irms * Math.sin(phi_rad)
S_apparent = Vrms * Irms
```

```
//| echo: false

html`<p><strong>Computed quantities:</strong> |<em>Z</em>| = ${Z_mag.toFixed(1)} Ω,   <e
<p><strong>Power:</strong> <em>P</em> = ${P_active.toFixed(1)} W,   <em>Q</em> = ${Q_rea
```

```
//| echo: false
//| label: fig-vi-trajectory-interactive
//| fig-cap: "Interactive VI trajectory - adjust R and X to see how the load signature change

{
  const N = 500;
  const Vm = Vrms * Math.SQRT2;
  const Im = Irms * Math.SQRT2;

  // Generate one cycle of v(t) and i(t)
  const data = Array.from({length: N}, (_, k) => {
    const theta = 2 * Math.PI * k / N;
    const v = Vm * Math.sin(theta);
    const i = Im * Math.sin(theta - phi_rad);
    return {v, i};
  });

  // Build plot
  const width = 500;
  const height = 400;
  const margin = {top: 20, right: 30, bottom: 50, left: 60};

  const svg = d3.create("svg")
    .attr("width", width)
    .attr("height", height)
    .attr("viewBox", [0, 0, width, height]);

  const xExtent = [-Vm * 1.1, Vm * 1.1];
  const yExtent = [-Im * 1.1, Im * 1.1];

  const xScale = d3.scaleLinear()
    .domain(xExtent)
```

```
  .range([margin.left, width - margin.right]);

const yScale = d3.scaleLinear()
  .domain(yExtent)
  .range([height - margin.bottom, margin.top]);

// Grid lines
svg.append("line")
  .attr("x1", xScale(0)).attr("x2", xScale(0))
  .attr("y1", margin.top).attr("y2", height - margin.bottom)
  .attr("stroke", "#999").attr("stroke-width", 0.5);

svg.append("line")
  .attr("x1", margin.left).attr("x2", width - margin.right)
  .attr("y1", yScale(0)).attr("y2", yScale(0))
  .attr("stroke", "#999").attr("stroke-width", 0.5);

// Axes
svg.append("g")
  .attr("transform", `translate(0,${height - margin.bottom})`)
  .call(d3.axisBottom(xScale).ticks(6));

svg.append("g")
  .attr("transform", `translate(${margin.left},0)`)
  .call(d3.axisLeft(yScale).ticks(6));

// Axis labels
svg.append("text")
  .attr("x", width / 2).attr("y", height - 5)
  .attr("text-anchor", "middle").attr("font-size", 13)
  .text("Voltage (V)");

svg.append("text")
  .attr("x", -height / 2).attr("y", 15)
  .attr("text-anchor", "middle").attr("font-size", 13)
  .attr("transform", "rotate(-90)")
  .text("Current (A)");

// VI trajectory path
const line = d3.line()
  .x(d => xScale(d.v))
  .y(d => yScale(d.i));
```

```
  svg.append("path")
    .datum(data)
    .attr("fill", "none")
    .attr("stroke", "steelblue")
    .attr("stroke-width", 2)
    .attr("d", line);

  // Starting point marker
  svg.append("circle")
    .attr("cx", xScale(data[0].v))
    .attr("cy", yScale(data[0].i))
    .attr("r", 4)
    .attr("fill", "red");

  return svg.node();
}
```

What to look for:

- **Straight line** (when $X = 0$): Voltage and current are in phase — a purely resistive load. The slope of the line is $1/R$ (steeper = lower resistance = more current).
- **Ellipse** (when $X \neq 0$): The phase shift between voltage and current opens the trajectory into an ellipse. A wider ellipse means a larger phase angle and more reactive power.
- **Tilt direction**: For inductive loads ($X > 0$), current lags voltage and the ellipse traverses counterclockwise. For capacitive loads ($X < 0$), current leads and the ellipse traverses clockwise.
- **Ellipse area**: Proportional to the reactive power Q — a wider ellipse means more energy "sloshing" back and forth without doing useful work.

13.2.11.1 What is a VI Trajectory?

A **VI trajectory** is a parametric plot of current $i(t)$ versus voltage $v(t)$ over one complete AC cycle, with time as the implicit parameter. Unlike time-series plots — which show voltage and current separately as functions of time — the VI trajectory captures their *relationship* directly in a single curve.

For a linear load with impedance $Z = R + jX$, the trajectory is described by the parametric equations:

$$v(\theta) = V_m \sin(\theta), \qquad i(\theta) = I_m \sin(\theta - \phi)$$

where $\theta = \omega t$ sweeps from 0 to 2π over one cycle. This parametric curve is always an **ellipse** (you can verify this by eliminating θ from the two equations). The degenerate case $\phi = 0$ collapses the ellipse into a straight line — as you can observe in the interactive tool above by setting $X = 0$.

The power of VI trajectories lies in their compactness: a single curve encodes the load's impedance magnitude (through the current amplitude), phase relationship (through the ellipse shape), and power characteristics (through the ellipse area). This makes them particularly useful as "fingerprints" for identifying different types of electrical loads, as we discuss next.

13.2.11.2 VI Trajectories for Different Load Types

The shape of a VI trajectory is determined by the type of load. For linear loads, the trajectory is always an ellipse (or a line), but non-linear loads produce more complex shapes. The figure below shows representative trajectories for four common load types.

Resistive load (top left): The trajectory is a straight line through the origin. Voltage and current are perfectly in phase — when voltage is positive, current is positive, and vice versa. The slope of the line equals $1/R$. Examples: incandescent light bulbs, resistive space heaters, electric water heater elements.

Inductive load (top right): The trajectory opens into an ellipse. Current lags voltage, so the ellipse is traversed **counterclockwise** (indicated by the arrow). The wider the ellipse, the larger the phase angle and the more reactive power the load draws. Examples: induction motors, fans, transformers, fluorescent lighting ballasts.

Capacitive load (bottom left): Also an ellipse, but now current leads voltage and the trajectory is traversed **clockwise**. Pure capacitive loads are less common in buildings, but capacitor banks used for power factor correction produce this signature. Some electronic power supplies also exhibit a leading power factor.

Non-linear load (bottom right): The trajectory departs from the smooth elliptical shape entirely. The example shown simulates a rectifier with a capacitor filter — the type of front-end found in most computers, phone chargers, and LED drivers. Current flows only in brief pulses near the voltage peaks (when the supply voltage exceeds the capacitor voltage), producing a distinctive "pinched" shape with sharp features. For these loads, the simple $Z = R + jX$ model no longer applies, and a full harmonic analysis is needed to characterize their behavior.

13.2.11.3 Using VI Trajectories for Load Identification

Since different load types produce distinct VI trajectory shapes, these trajectories can serve as **electrical fingerprints** for identifying what appliance is running. This insight is the foundation of **Non-Intrusive Load Monitoring (NILM)** — a technique where a single sensor installed at the electrical panel monitors the aggregate voltage and current, and algorithms decompose

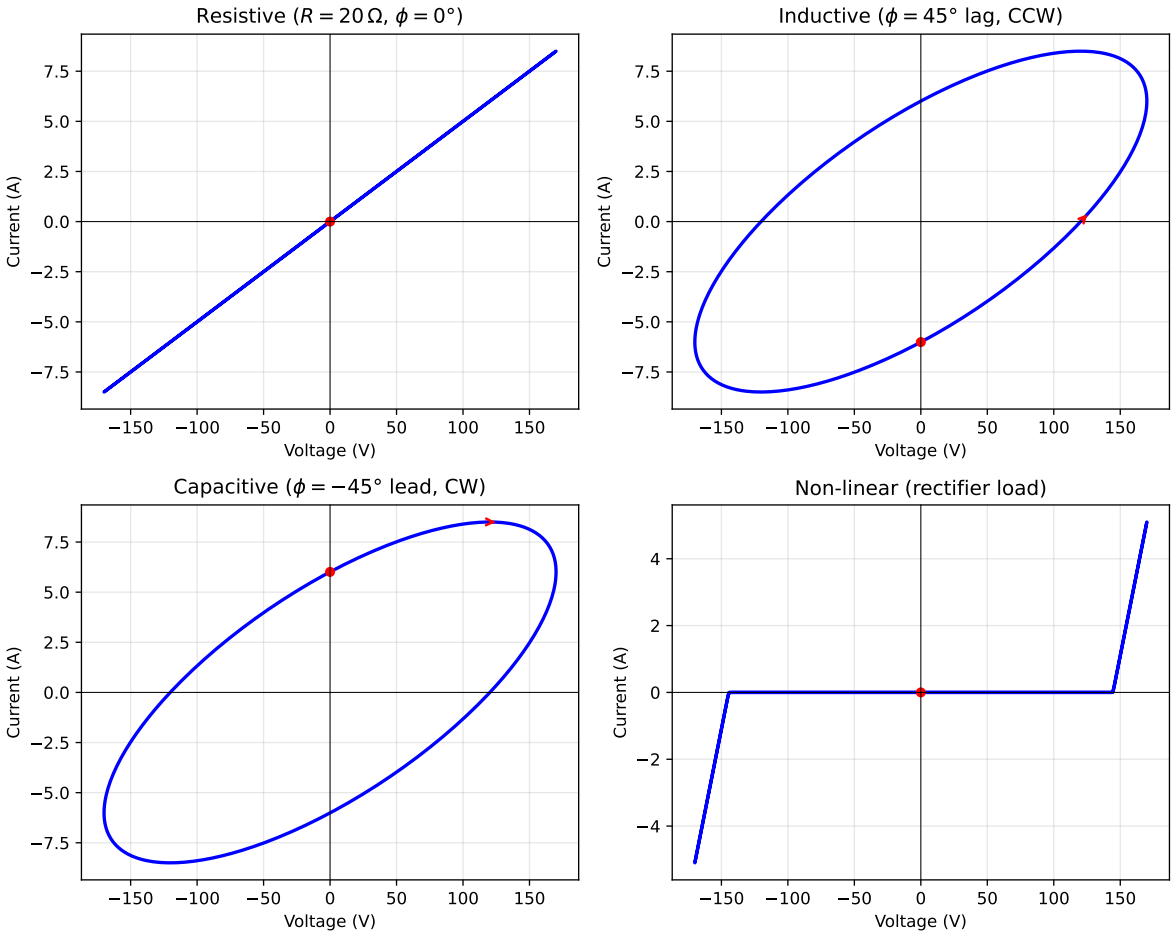


Figure 13.7: VI trajectories for four load types: resistive (straight line), inductive (ellipse, CCW), capacitive (ellipse, CW), and non-linear (complex shape)

the combined signal to identify and track individual loads without requiring a sensor on each appliance.

In practice, a NILM system works by building a library of reference VI trajectories for known appliance types (e.g., a refrigerator compressor, a microwave, a laptop charger). When the system detects a change in the aggregate signal — such as a new load turning on — it extracts the VI trajectory of the change and compares it against the library using pattern matching or machine learning classifiers. Variations of this approach have been applied to residential energy disaggregation, fault detection in commercial buildings, and industrial equipment monitoring. NILM remains an active research area, with ongoing work on improving accuracy, handling multiple simultaneous events, and scaling to diverse building types.

That said, VI-based load identification has practical limitations. When multiple loads operate simultaneously, the measured trajectory is a **superposition** of individual trajectories, making decomposition challenging. Loads of the same type (e.g., two different induction motors) may produce similar trajectories that are hard to distinguish. And loads that operate in variable or transient modes — such as a washing machine cycling through different stages — produce time-varying trajectories that don't fit neatly into a single-cycle fingerprint. Despite these challenges, VI trajectories remain one of the most intuitive single-cycle representations of electrical load behavior.

13.3 Additional Resources

13.3.1 References

- Power in an AC Circuit - University of Central Florida Physics Textbook

14 Basics of Control Theory for Buildings and Their Application

14.1 Lecture Overview

Learning Objectives

By the end of this module, students will be able to:

- Recognize the different branches of control theory, and where optimal control sits
- Identify the components of a Model Predictive Control problem formulation for a given context: cost functional, constraints, dynamics model, decision variables, etc.
- Identify how the choice of constraints, cost and variables determines the type of optimization problem and the available solutions
- Use linear models of thermal dynamics (e.g., RC thermal networks) to formulate simple MPC problems
- Use a solver (e.g, `cvxpy`) to implement an MPC controller for a 1R1C thermal model
- Recognize the value of a standardized environment for benchmarking and comparing different control algorithms
- Identify the major endpoints of the BOPTEST API
- Test a controller against a BOPTEST testcase

Topics Covered

- Overview of control theory branches and optimal control
- Model Predictive Control (MPC) fundamentals and formulation
- MPC for building thermal dynamics using RC network models
- Implementation of MPC controllers using cvxpy
- Building Optimization Testing Framework (BOPTEST)
- BOPTEST API and controller testing

Project Milestones

This lecture introduces MPC concepts and the BOPTEST framework, both of which may be relevant for teams pursuing HVAC control as part of their final project. Understanding

MPC formulation and implementation will be valuable for designing autonomous building control strategies.

14.2 Introduction to Control Theory

Every physical system in a building — thermal, lighting, acoustic, hydraulic — has its own natural dynamics that tend toward an equilibrium determined by the environment and the laws of physics. Without intervention, indoor air temperature drifts toward the outdoor temperature, light levels follow the sun's position and cloud cover, CO_2 concentration rises with occupancy, and domestic hot water cools to ambient. In virtually every case, these natural equilibria are not where we want the building to operate. Occupants need temperatures around 21°C regardless of whether it's $-10°C$ or $35°C$ outside, consistent illumination for visual tasks, acceptable air quality, and hot water on demand. Bridging the gap between where physics takes the system and where we need it to be requires **actively expending energy** — and doing so intelligently is the domain of control theory.

The challenge is that the objectives we care about often compete. Maintaining tight temperature control requires more energy than allowing wider fluctuations. Pre-cooling a building to reduce peak demand may temporarily sacrifice comfort. Ventilating aggressively for air quality increases heating or cooling loads. These trade-offs mean that building control is not simply a matter of turning equipment on and off — it requires balancing multiple, sometimes conflicting goals:

- **Thermal comfort**: Maintain indoor air temperature (and humidity, radiant temperature) within acceptable ranges
- **Energy efficiency**: Minimize total energy consumption or cost
- **Indoor air quality**: Keep CO_2, particulate matter, and VOC concentrations below thresholds
- **Visual comfort**: Provide adequate and glare-free illumination
- **Acoustic comfort**: Manage noise from HVAC equipment and other sources
- **Peak demand management**: Reduce electricity demand during expensive or grid-stressed periods
- **Equipment longevity**: Avoid excessive cycling or operation outside design conditions

In the preceding lectures, we built up the physical understanding needed to model these systems — thermal dynamics via RC networks, electrical power fundamentals, and sensor/actuator technologies. Now we turn to the question: given a model of how the building behaves and a set of objectives, **how do we make optimal control decisions?**

14.2.1 Branches of Control Theory

At the broadest level, control strategies for buildings fall into two categories: **passive** and **active**. Passive control requires no energy input or real-time decision-making — it is designed into the building itself. Examples include high-performance insulation, thermal mass that buffers temperature swings, fixed shading devices sized for the sun's seasonal path, and natural ventilation driven by stack effect or wind pressure. Passive strategies are powerful and should always be the first line of defense, but they have limits: they cannot adapt to changing conditions or occupant preferences, and they alone are rarely sufficient to maintain the precise indoor conditions modern buildings demand.

Active control bridges the gap by using energy and information to manipulate building systems in real time. Within active control, the simplest approach is **open-loop control**, where the system follows a pre-determined schedule with no knowledge of the current state. A time-clock that turns the HVAC system on at 6 AM and off at 6 PM is open-loop: it executes the same plan regardless of whether the building is already warm, whether it's occupied, or whether the weather has changed. Open-loop control is easy to implement but inherently blind to what is actually happening in the building.

Sensor-based control improves on this by using measurements to inform decisions. Here the controller has access to some observation of the system's state — a thermostat reading, a CO_2 sensor, an occupancy detector — and adjusts its actions accordingly. Sensor-based control can be further divided by *what information* the controller uses. **Feedback control** reacts to the current (or recent) state of the system: the classic thermostat measures indoor temperature and turns heating on or off to correct deviations from the setpoint. **Feedforward control**, by contrast, anticipates disturbances *before* they affect the system: a controller that pre-heats a building based on a weather forecast or pre-cools before an anticipated occupancy surge is using feedforward information. In practice, most effective building controllers combine both — using feedback to correct for current errors and feedforward to anticipate known future disturbances.

Within sensor-based control, the question of *how* to design the control law — what action to take given the available information — has given rise to a rich landscape of approaches. Classical control (PID controllers, on-off thermostats) relies on simple rules and tuning. Modern control theory introduces state-space models and systematic design methods. **Optimal control**, which we focus on next, takes the approach of explicitly defining what "good" performance means through a cost function and then finding the control strategy that minimizes that cost — subject to the system's dynamics and constraints.

14.2.2 Optimal Control and Its Place in Control Theory

The approaches described above — PID controllers, on-off thermostats, scheduled setbacks — are designed by intuition and tuning. They work well for simple cases, but as objectives

multiply and constraints tighten, hand-crafted rules become difficult to design and impossible to guarantee as optimal. **Optimal control** takes a fundamentally different approach: instead of prescribing *how* the controller should behave, we define *what we want* (through a cost function) and *what's possible* (through a model and constraints), and then let mathematical optimization find the best control strategy.

This requires formalizing the control problem with four ingredients:

1. Dynamical system — The physical process we are controlling, described by how its state evolves over time:

$$\dot{x}(t) = f(x(t), u(t), w_d(t))$$

where $x(t)$ is the **state vector** (e.g., zone temperatures, wall temperatures, stored energy), $u(t)$ is the **control input vector** (e.g., heating power, valve positions, fan speeds), and $w_d(t)$ is a **disturbance vector** (e.g., outdoor temperature, solar radiation, internal heat gains from occupants and equipment). The function f encodes the physics — for thermal systems, these are the RC network equations we developed in earlier lectures.

2. Observations — What the controller can actually measure:

$$y(t) = g(x(t), u(t), w_n(t))$$

where $y(t)$ is the **observation vector** (sensor readings) and $w_n(t)$ is **measurement noise**. In general, the controller does not have direct access to the full state x — it only sees y, which may be a noisy, incomplete view. For instance, we might measure indoor air temperature at one location but not the temperature distribution throughout the zone or within the walls.

3. Control law — The rule that maps available information to control actions:

$$u(t) = k(y(t), w_r(t))$$

where $w_r(t)$ is a **reference signal** (e.g., the desired setpoint trajectory or an occupancy schedule). The function k is what we are designing — it is the controller itself.

4. Cost functional — A scalar measure of how well the controller performs over time:

$$J = \int_0^T \ell(x(t), u(t), w_r(t))\, dt$$

where ℓ is a **stage cost** that penalizes undesirable states (discomfort, high energy use, constraint violations) at each instant. The optimal control problem is then: find the control law k that minimizes J subject to the dynamics, observations, and constraints.

This framework is a natural fit for buildings. The competing objectives we identified — comfort, energy, air quality, cost — are encoded in J. Physical limitations (maximum heating capacity, temperature bounds, ventilation minimums) appear as constraints. The building's thermal, electrical, and air quality dynamics appear in f. And the imperfect information available from sensors appears in g and w_n. Among the various methods for solving optimal control problems, **Model Predictive Control (MPC)** has emerged as the most practical and widely used approach for buildings, and it is the focus of the remainder of this lecture.

14.3 Model Predictive Control Fundamentals

The optimal control framework above is elegant but, in its full generality, difficult to solve. Finding the globally optimal control law k over an infinite (or very long) time horizon, under uncertainty, for a constrained nonlinear system is computationally intractable for most practical problems. **Model Predictive Control (MPC)** offers a powerful and practical approximation that has become the dominant approach for optimal building control.

14.3.1 What is Model Predictive Control?

The core idea of MPC is to replace the intractable infinite-horizon problem with a sequence of tractable **finite-horizon** problems. At each control time step, the controller uses a model of the system to predict how the state will evolve over a fixed prediction horizon (e.g., the next 2–6 hours). It then solves an optimization problem to find the sequence of control inputs over that horizon that minimizes the cost function while satisfying all constraints. Crucially, only the **first** control action from the optimal sequence is actually applied to the system. At the next time step, the controller takes a new measurement, shifts the horizon forward by one step, and solves the optimization again. This strategy is called the **receding horizon** (or moving horizon) principle.

The receding horizon approach is what makes MPC robust in practice. Because the controller re-solves with fresh measurements at every step, it naturally corrects for model inaccuracies, unmeasured disturbances, and forecast errors. The model does not need to be perfect — it only needs to be good enough to make useful predictions over the horizon length. If the outdoor temperature turns out to be 2°C colder than forecasted, the controller will see the resulting indoor temperature deviation at the next measurement and adjust its plan accordingly.

MPC is particularly well suited to building control for several reasons. First, buildings have relatively **slow dynamics** — thermal time constants range from minutes to hours — so the optimization does not need to run at millisecond speeds; solving once every 5–15 minutes is typically sufficient. Second, **forecast data** that MPC needs is readily available: weather forecasts, occupancy schedules, and energy price signals can all be incorporated as predicted disturbances. Third, buildings have hard **constraints** that must be respected (equipment capacity limits, comfort bounds, ventilation minimums), and MPC handles these naturally as

part of the optimization. Finally, the **multi-objective** nature of building control — balancing comfort, energy, cost, and emissions — maps directly onto MPC's cost function, where each objective becomes a weighted term.

14.3.2 Components of an MPC Problem Formulation

Every MPC problem is defined by four ingredients: a **cost functional** that encodes what we want to achieve, **constraints** that encode what is physically and operationally possible, a **dynamics model** that predicts how the system will respond to our actions, and **decision variables** that define what the optimizer is free to choose. The choices we make for each of these determine the mathematical structure of the resulting optimization problem — and therefore what solution methods are available and how computationally expensive the problem is to solve.

14.3.2.1 Cost Functional (Objective Function)

The cost functional (also called the objective function) is the mathematical expression of "what good performance looks like." It assigns a scalar cost to any candidate trajectory of states and control inputs over the prediction horizon, and the optimizer seeks the trajectory that minimizes this cost.

For building control, common cost functional forms include:

- **Quadratic setpoint tracking**: $\sum_k w_{comfort}(T_i^k - T_{set})^2$ — penalizes deviations from a desired setpoint, with squared terms ensuring that large deviations are penalized disproportionately more than small ones
- **Energy minimization**: $\sum_k w_{energy}(P_{hea}^k)^2$ or $\sum_k w_{energy} P_{hea}^k$ — penalizes control effort; the quadratic form discourages large spikes in power while the linear form penalizes total energy equally
- **Economic cost**: $\sum_k c_k \cdot P_{hea}^k \cdot \Delta t$ — where c_k is a time-varying electricity price, encouraging load shifting to off-peak hours
- **Peak demand reduction**: $w_{peak} \cdot \max_k P_{hea}^k$ — penalizes the maximum power draw, which is relevant for demand charge reduction

In practice, the cost functional is usually a weighted sum of several of these terms. The **weights** ($w_{comfort}$, w_{energy}, etc.) are design parameters that set the trade-off between competing objectives. Choosing these weights is one of the most consequential decisions in MPC design — they determine whether the controller prioritizes tight comfort, low energy, or economic savings.

14.3.2.2 Constraints

Constraints define the boundaries of what is feasible — both physically and operationally. They come in two forms. **Equality constraints** express relationships that must hold exactly, the most important being the dynamics equations themselves: $x_{k+1} = A_d x_k + B_d u_k + E_d w_k$. These are structural — they ensure the optimizer's predicted trajectory is consistent with the physics of the building. The initial condition ($x_0 = x_{measured}$) is also an equality constraint that anchors the prediction to the current state.

Inequality constraints express bounds that must not be violated. These include **actuator limits** (heating power between 0 and P_{max}, valve positions between 0% and 100%), **comfort bounds** (indoor temperature between T_{min} and T_{max}), and **operational limits** (minimum ventilation rates, maximum ramp rates for equipment). In some formulations, comfort bounds are treated as **soft constraints** — meaning small violations are allowed but penalized in the cost function — rather than hard constraints. This is useful because hard comfort constraints can make the problem infeasible (e.g., if the building is far from the comfort range and the heating system lacks the capacity to reach it within one time step). Soft constraints maintain feasibility while still discouraging violations.

14.3.2.3 Dynamics Model

The dynamics model is what gives MPC its predictive power — it is what distinguishes MPC from reactive controllers like thermostats. The model allows the controller to answer "if I apply this sequence of control inputs over the next N time steps, what will the indoor temperature trajectory look like?" This predictive capability is what enables anticipatory behavior: pre-heating before a cold front, pre-cooling before peak pricing, or coasting on thermal mass when outdoor conditions are favorable.

For building thermal control, the RC network models we developed in Lectures 5 and 6 provide exactly the kind of model MPC needs. Expressed in discrete-time state-space form ($x_{k+1} = A_d x_k + B_d u_k + E_d w_k$), they are linear, computationally cheap to evaluate, and have parameters with physical meaning. The trade-off is between model fidelity and computational cost: a 1R1C model is fast to solve but may not capture multi-zone interactions, while a higher-order model (3R3C, or even a full-building EnergyPlus model) is more accurate but makes the optimization harder. For MPC, a simpler model that captures the dominant dynamics is often preferable to a detailed model that slows down the optimization, since MPC's receding horizon naturally compensates for model errors at each time step.

14.3.2.4 Decision Variables

The decision variables are the quantities the optimizer is free to choose in order to minimize the cost function. In a standard MPC formulation, these are the **future control inputs**

$u_0, u_1, \ldots, u_{N-1}$ over the prediction horizon. In many formulations (including the one we'll use), the **future states** $x_1, x_2, \ldots, x_N$ are also treated as decision variables, with the dynamics equations imposed as equality constraints linking them to the inputs. This "simultaneous" approach makes the optimization problem larger but keeps it structured in a way that modern solvers handle efficiently.

The **prediction horizon** N (expressed in number of time steps) determines how far into the future the controller plans. A longer horizon allows the controller to anticipate disturbances further ahead — for example, pre-heating overnight in anticipation of a cold morning — but increases the number of decision variables and thus the computational cost. For building thermal control with 15-minute time steps, horizons of 4–24 hours (16–96 steps) are typical. Beyond a certain point, extending the horizon yields diminishing returns because forecast accuracy degrades and the receding horizon re-solves the problem at every step anyway.

14.3.3 From Problem Formulation to Optimization Type

The choices made in the four components above — cost functional, constraints, dynamics, and decision variables — collectively determine the mathematical class of the resulting optimization problem. This matters enormously in practice because the problem class dictates what solution algorithms are available, how fast they run, and what guarantees they provide about the quality of the solution.

14.3.3.1 Linear vs. Nonlinear Problems

If the dynamics model is linear (as with our RC thermal networks), the constraints are linear inequalities (box bounds on temperatures and control inputs), and the cost function is linear, the MPC problem is a **Linear Program (LP)**. If the cost is quadratic (e.g., squared deviation from setpoint), we get a **Quadratic Program (QP)**. Both LPs and QPs can be solved reliably and efficiently by mature solvers — even for problems with thousands of variables, solutions typically take milliseconds to seconds.

When the dynamics are nonlinear (e.g., a detailed model that includes radiative heat exchange, humidity-dependent properties, or variable-speed equipment curves), the resulting optimization is a **Nonlinear Program (NLP)**. NLPs are harder to solve: they require iterative algorithms (e.g., interior point methods, sequential quadratic programming), may take longer to converge, and — as we discuss next — may not guarantee a globally optimal solution. For building MPC, this is a key motivation for using linearized RC models even when more detailed nonlinear models are available: keeping the dynamics linear keeps the optimization tractable.

14.3.3.2 Convex vs. Non-Convex Problems

A problem is **convex** if the cost function is convex (bowl-shaped) and the feasible region defined by the constraints is a convex set. The critical property of convex problems is that **any local minimum is also the global minimum** — so if a solver finds a solution, it is guaranteed to be the best one. There are no hidden, better solutions lurking elsewhere in the feasible space.

QPs with linear constraints — which is what we get from linear RC models with quadratic comfort and energy costs — are convex. This is the practical sweet spot for building MPC: expressive enough to capture meaningful trade-offs (quadratic costs penalize large deviations more than small ones), yet convex enough that solvers can find the global optimum quickly and reliably. Tools like `cvxpy` (which we'll use shortly) are specifically designed for convex problems and will even verify that your problem formulation is convex before attempting to solve it. Non-convex problems, by contrast, may have multiple local minima, and solvers can get trapped in a suboptimal one with no way to know whether a better solution exists.

14.3.3.3 Continuous vs. Discrete Decision Variables

In the formulations above, all decision variables are continuous — heating power can take any value between 0 and P_{max}. But some building systems have inherently **discrete** decisions: a boiler is either on or off, a chiller plant has 1, 2, or 3 units running, a window is open or closed, and lighting may only be dimmable in fixed steps. When some decision variables are restricted to integer or binary values while others remain continuous, the result is a **Mixed-Integer Program (MIP)** — either a Mixed-Integer Linear Program (MILP) or Mixed-Integer Quadratic Program (MIQP) depending on the cost function.

MIPs are significantly harder to solve than their continuous counterparts. The discrete variables create a combinatorial explosion: with n binary variables, there are 2^n possible combinations to consider. While modern MIP solvers (e.g., Gurobi, CPLEX) use branch-and-bound algorithms that avoid enumerating all combinations, solve times can still be orders of magnitude longer than for continuous QPs. For building MPC, this means that formulations with many on/off decisions (e.g., coordinating a large number of discrete HVAC units) may push against real-time computation limits. A common workaround is to solve the continuous relaxation first (allowing "fractional" on/off values) and then round the result, accepting a small loss of optimality in exchange for computational tractability.

14.4 MPC for Building Thermal Dynamics

In Lectures 5 and 6, we developed thermal network models to represent how heat flows through buildings. We learned how to model walls, windows, and thermal mass using networks of resistances (R) and capacitances (C), deriving equations that describe the building's thermal

dynamics. Those models allowed us to *predict* what would happen to indoor temperature given certain inputs (outdoor temperature, solar gains, heating power). Now, we take the next step: using those same models not just for prediction, but for making *optimal control decisions*.

This is where Model Predictive Control comes in. MPC leverages a dynamics model (like our RC thermal networks) to predict the future behavior of the building over some time horizon. It then uses optimization to find the sequence of control inputs (e.g., heating/cooling power) that best achieves our objectives—such as maintaining comfort while minimizing energy consumption—subject to physical and operational constraints.

The beauty of MPC for building control is that it naturally handles several challenges that simple reactive controllers (like proportional controllers) struggle with:

- **Anticipatory behavior**: MPC can "see" upcoming disturbances (weather forecasts, occupancy schedules) and act proactively rather than just reacting to current conditions
- **Multi-objective optimization**: We can explicitly balance competing goals like comfort, energy cost, and peak demand reduction in a single cost function
- **Constraint handling**: Physical limits (maximum heating power, comfort temperature bounds) are directly incorporated as constraints in the optimization problem
- **Thermal mass utilization**: MPC can strategically use the building's thermal mass to shift energy consumption to off-peak hours while maintaining comfort

The key insight is this: the RC thermal network models we've studied provide exactly the kind of predictive model MPC needs. Because these models are often linear (or can be linearized), we can formulate the resulting MPC problem as a convex optimization problem that can be solved efficiently and reliably at each control timestep.

14.4.1 Using RC Thermal Networks in MPC

To use RC thermal networks in an MPC framework, we need to express the building's thermal dynamics in a form that's compatible with optimization algorithms. Let's recall that RC thermal networks give us differential equations describing how temperatures evolve over time. For example, a simple 1R1C network yields:

$$C\frac{dT_i}{dt} = \frac{1}{R}(T_a - T_i) + P_{hea}$$

where T_i is the indoor temperature, T_a is the ambient (outdoor) temperature, P_{hea} is the heating power, C is the thermal capacitance, and R is the thermal resistance.

For MPC, we need to transform this continuous-time differential equation into a form that:

1. **Separates states, inputs, and disturbances**: We need to clearly distinguish what we control (inputs) from what we observe (states) and what affects us but we can't control (disturbances)

2. **Works with discrete time steps**: Digital controllers operate at discrete time intervals (e.g., every 15 minutes), so we need a discrete-time representation
3. **Enables prediction over a horizon**: The model needs to let us predict future states based on sequences of future inputs and disturbances

The standard approach is to use **state-space representation**, which expresses the system dynamics in matrix form. This representation is ideal for MPC because:

- It provides a compact, general framework that works for networks of any complexity (1R1C, 2R2C, etc.)
- It clearly separates the system's state evolution from the outputs we measure
- It interfaces naturally with optimization solvers that expect linear dynamics
- It makes it straightforward to incorporate forecasts (weather, occupancy, etc.)

14.4.2 State-Space Representation Review

The **state-space representation** is a mathematical framework for modeling dynamical systems. For linear systems, the continuous-time state-space form is:

$$\dot{x}(t) = Ax(t) + Bu(t) + Ew(t)$$

where:

- $x(t)$ is the **state vector** (temperatures in our case, like T_i, T_e)
- $u(t)$ is the **control input vector** (what we can manipulate, like P_{hea})
- $w(t)$ is the **disturbance vector** (external factors we can't control, like T_a, solar irradiance)
- A is the **state matrix** (describes system dynamics)
- B is the **input matrix** (describes how control inputs affect states)
- E is the **disturbance matrix** (describes how disturbances affect states)

In Lecture 6, you saw how thermal network models could be written in this form. For example, the 2R2C model equations can be expressed as a 2×2 state matrix operating on the state vector $[T_i, T_e]^T$.

14.4.2.1 Discrete-Time State-Space Form

Since digital controllers operate at discrete time steps (not continuously), we need the **discrete-time** version. Using explicit Euler discretization with timestep Δt, we get:

$$x_{k+1} = A_d x_k + B_d u_k + E_d w_k$$

where the subscript k denotes the time index (i.e., $x_k = x(k \cdot \Delta t)$), and the discrete-time matrices are related to the continuous-time ones by:

$$A_d = I + \Delta t \cdot A$$
$$B_d = \Delta t \cdot B$$
$$E_d = \Delta t \cdot E$$

This discrete-time form is exactly what we need for MPC: given the current state x_k, future control inputs $u_k, u_{k+1}, ...$, and forecasts of disturbances $w_k, w_{k+1}, ...$, we can recursively predict future states $x_{k+1}, x_{k+2}, ...$ over the MPC horizon.

> **Why Explicit Euler?**
>
> More sophisticated discretization methods (e.g., implicit Euler, Runge-Kutta) can provide better accuracy, especially for stiff systems or large timesteps. However, explicit Euler is simple, maintains linearity (which keeps the MPC problem convex), and is sufficiently accurate for typical building control timesteps (15-60 minutes). For the purposes of this course, we'll use explicit Euler discretization.

14.4.3 Example: 1R1C Model for MPC

Let's work through a complete example of formulating an MPC problem for a building represented by a simple 1R1C thermal network. This will illustrate all four components of MPC: the dynamic model, cost function, constraints, and optimization.

14.4.3.1 Model Setup

Consider a building modeled as a single thermal zone with:

- **State**: Indoor temperature T_i (the temperature we want to control)
- **Control input**: Heating power P_{hea} (what we manipulate)
- **Disturbance**: Ambient temperature T_a (outdoor temperature, which we forecast but can't control)

The continuous-time thermal dynamics are:

$$C\frac{dT_i}{dt} = \frac{1}{R}(T_a - T_i) + P_{hea}$$

where C is the thermal capacitance [J/K] and R is the thermal resistance [K/W].

Rearranging into state-space form:

$$\frac{dT_i}{dt} = -\frac{1}{RC}T_i + \frac{1}{C}P_{hea} + \frac{1}{RC}T_a$$

This gives us:

- State matrix: $A = -\frac{1}{RC}$
- Input matrix: $B = \frac{1}{C}$
- Disturbance matrix: $E = \frac{1}{RC}$

Discrete-time version (using explicit Euler with timestep Δt):

$$T_i^{k+1} = T_i^k + \frac{\Delta t}{C}\left[\frac{1}{R}(T_a^k - T_i^k) + P_{hea}^k\right]$$

Rearranging to isolate T_i^{k+1}:

$$T_i^{k+1} = \underbrace{\left(1 - \frac{\Delta t}{RC}\right)}_{a} T_i^k + \underbrace{\frac{\Delta t}{C}}_{b} P_{hea}^k + \underbrace{\frac{\Delta t}{RC}}_{e} T_a^k$$

or more compactly:

$$T_i^{k+1} = a \cdot T_i^k + b \cdot P_{hea}^k + e \cdot T_a^k$$

where $a = 1 - \frac{\Delta t}{RC}$, $b = \frac{\Delta t}{C}$, and $e = \frac{\Delta t}{RC}$.

14.4.3.2 Cost Function Design

The MPC controller needs to balance two competing objectives:

1. **Comfort**: Keep indoor temperature close to a setpoint T_{set}
2. **Energy efficiency**: Minimize heating energy consumption

A typical cost function that balances these objectives is:

$$J = \sum_{k=0}^{N-1} \left[w_{comfort}(T_i^k - T_{set})^2 + w_{energy}(P_{hea}^k)^2\right]$$

where:

- N is the prediction horizon (number of timesteps into the future)

- $w_{comfort}$ is the weight on comfort violations (large value = prioritize comfort)
- w_{energy} is the weight on energy use (large value = prioritize energy savings)
- Squared terms penalize large deviations more heavily than small ones

The ratio $\frac{w_{comfort}}{w_{energy}}$ determines the trade-off:

- High ratio → tight temperature control, higher energy use
- Low ratio → looser temperature control, lower energy use

i Terminal Cost

In addition to the stage costs above, MPC formulations often include a **terminal cost** on the final state:

$$J = \sum_{k=0}^{N-1} \left[w_{comfort}(T_i^k - T_{set})^2 + w_{energy}(P_{hea}^k)^2 \right] + w_{terminal}(T_i^N - T_{set})^2$$

The terminal cost serves several purposes:

- **Stability**: For certain choices of terminal cost and terminal constraints, the terminal cost can guarantee closed-loop stability of the MPC controller
- **Long-term behavior**: It encourages the controller to leave the system in a good state at the end of the horizon, which helps performance beyond the prediction horizon
- **Approximation of infinite horizon**: It approximates the cost-to-go from the terminal state onward when we can't afford to optimize over an infinite horizon

For practical building control, the terminal cost is often set equal to or slightly higher than the stage comfort cost weight (i.e., $w_{terminal} \geq w_{comfort}$).

14.4.3.3 Constraint Specification

Real systems have physical and operational limits that must be respected:

Temperature comfort bounds:

$$T_{min} \leq T_i^k \leq T_{max} \quad \forall k = 1, \ldots, N$$

For example, $T_{min} = 20°C$ and $T_{max} = 24°C$ define an acceptable comfort range.

Heating power limits:

$$0 \leq P_{hea}^k \leq P_{max} \quad \forall k = 0, \ldots, N-1$$

where P_{max} is the maximum capacity of the heating system (e.g., 5000 W). Note that we typically don't allow negative heating power (cooling would be a separate control input).

Initial condition:

$$T_i^0 = T_i^{measured}$$

The optimization must start from the current measured temperature.

14.4.3.4 Complete Problem Formulation

Putting it all together, the MPC optimization problem at each timestep is:

! MPC Optimization Problem (1R1C Example)

Decision variables: $T_i^0, T_i^1, \ldots, T_i^N$ and $P_{hea}^0, P_{hea}^1, \ldots, P_{hea}^{N-1}$
Minimize:

$$J = \sum_{k=0}^{N-1} \left[w_{comfort}(T_i^k - T_{set})^2 + w_{energy}(P_{hea}^k)^2 \right] + w_{terminal}(T_i^N - T_{set})^2$$

Subject to:
Dynamics (for $k = 0, \ldots, N-1$):

$$T_i^{k+1} = a \cdot T_i^k + b \cdot P_{hea}^k + e \cdot T_a^k$$

Initial condition:

$$T_i^0 = T_i^{measured}$$

Temperature bounds (for $k = 1, \ldots, N$):

$$T_{min} \leq T_i^k \leq T_{max}$$

Heating power bounds (for $k = 0, \ldots, N-1$):

$$0 \leq P_{hea}^k \leq P_{max}$$

Key observations:

1. This is a **convex quadratic program** (QP) because the cost is quadratic and all constraints are linear
2. We have $N + 1$ temperature variables and N control variables (total: $2N + 1$ decision variables)

3. Once solved, we implement only P^0_{hea} (the first control action) and discard the rest
4. At the next timestep, we resolve the entire problem with updated measurements and forecasts—this is the **receding horizon** principle of MPC

14.5 Implementation with cvxpy

Now that we've formulated the MPC problem mathematically, we need a way to actually solve it. While we could implement our own optimization algorithms (gradient descent, interior point methods, etc.), this would be time-consuming and error-prone. Instead, we'll use **CVXPY**, a Python library specifically designed for formulating and solving convex optimization problems.

14.5.1 What is cvxpy?

CVXPY (pronounced "cvx-pie") is a domain-specific language for convex optimization embedded in Python. It allows you to express optimization problems in a natural mathematical syntax, then automatically transforms them into a standard form and dispatches them to appropriate solvers.

Key features that make cvxpy ideal for MPC:

- **Declarative syntax**: You write code that looks like the mathematical problem formulation
- **Automatic problem classification**: cvxpy determines whether your problem is linear, quadratic, convex, etc.
- **Solver interface**: Automatically selects and interfaces with appropriate solvers (OSQP, ECOS, SCS, etc.)
- **Disciplined Convex Programming (DCP)**: Enforces rules that guarantee your problem is actually convex
- **Fast prototyping**: Quickly experiment with different cost functions and constraints

How cvxpy works:

1. You define **variables** using `cp.Variable()`
2. You express **constraints** using Python comparison operators (`<=`, `>=`, `==`)
3. You define the **objective** using cvxpy functions like `cp.Minimize()`, `cp.sum_squares()`, etc.
4. You create a **problem** object: `cp.Problem(objective, constraints)`
5. You call `problem.solve()` which invokes the appropriate solver
6. You extract the solution from the `.value` attribute of your variables

For our MPC application, cvxpy will handle all the complexity of transforming our problem into the standard form expected by quadratic programming (QP) solvers, and will efficiently solve the optimization at each control timestep.

14.5.2 Setting Up an MPC Problem in cvxpy

Let's translate the 1R1C MPC problem from the previous section into cvxpy code. We'll go step-by-step, mapping each mathematical element to its cvxpy equivalent.

Step 1: Define optimization variables

In our problem, the decision variables are the temperature trajectory and the control inputs:

```
import cvxpy as cp
import numpy as np

# Problem parameters
N = 24  # Prediction horizon (number of timesteps)

# Define optimization variables
T = cp.Variable(N + 1)    # Temperature at each timestep [T_0, T_1, ..., T_N]
u_kW = cp.Variable(N)    # Control inputs in kW [u_0, u_1, ..., u_{N-1}]
```

Step 2: Set up constraint list

We'll build a Python list to hold all constraints:

```
constraints = []
```

Step 3: Add initial condition constraint

$$T_i^0 = T_i^{measured}$$

```
T_current = 20.0  # Current measured temperature [°C]
constraints += [T[0] == T_current]
```

Step 4: Add dynamics constraints

For each timestep $k = 0, \ldots, N-1$:

$$T_i^{k+1} = a \cdot T_i^k + b \cdot P_{hea}^k + e \cdot T_a^k$$

```
# Model parameters (example values for a 1R1C model)
R = 0.005     # Thermal resistance [K/W]
C = 5e6       # Thermal capacitance [J/K]
dt = 900      # Timestep [seconds] (15 minutes)

# Compute discrete-time coefficients
a = 1 - dt/(R*C)
b = dt/C * 1000  # Multiplies kW, so scale by 1000
e = dt/(R*C)

# Ambient temperature forecast (assume constant for simplicity)
T_amb = np.full(N, 5.0)  # [°C]

# Add dynamics constraints for each timestep
for k in range(N):
    constraints += [T[k+1] == a*T[k] + b*u_kW[k] + e*T_amb[k]]
```

Step 5: Add temperature bounds

$$T_{min} \leq T_i^k \leq T_{max} \quad \forall k = 1, \ldots, N$$

```
T_min = 20.0  # Minimum comfort temperature [°C]
T_max = 24.0  # Maximum comfort temperature [°C]

constraints += [T[1:] >= T_min]  # Apply to T[1], T[2], ..., T[N]
constraints += [T[1:] <= T_max]
```

Step 6: Add control input bounds

$$0 \leq P_{hea}^k \leq P_{max} \quad \forall k = 0, \ldots, N-1$$

```
P_max_kW = 5.0  # Maximum heating power [kW]

constraints += [u_kW >= 0]
constraints += [u_kW <= P_max_kW]
```

Step 7: Define the cost function

$$J = \sum_{k=0}^{N-1} \left[w_{comfort}(T_i^k - T_{set})^2 + w_{energy}(P_{hea}^k)^2 \right] + w_{terminal}(T_i^N - T_{set})^2$$

```
T_set = 21.0      # Temperature setpoint [°C]
w_comfort = 1.0   # Weight on comfort
w_energy = 0.01   # Weight on energy
w_terminal = 1.0  # Terminal cost weight

# Stage costs (over horizon)
comfort_cost = w_comfort * cp.sum_squares(T[:-1] - T_set)  # T[0] through T[N-1]
energy_cost = w_energy * cp.sum_squares(u_kW / P_max_kW)

# Terminal cost
terminal_cost = w_terminal * cp.sum_squares(T[-1] - T_set)  # T[N]

# Total cost
total_cost = comfort_cost + energy_cost + terminal_cost
```

Step 8: Create and solve the problem

```
# Define the optimization problem
problem = cp.Problem(cp.Minimize(total_cost), constraints)

# Solve the problem
problem.solve()

# Check if solution was found
if problem.status in ["optimal", "optimal_inaccurate"]:
    print(f"Optimal solution found!")
    print(f"Optimal cost: {problem.value:.2f}")
else:
    print(f"Optimization failed: {problem.status}")
```

```
Optimal solution found!
Optimal cost: 1.61
```

14.5.3 Solving and Extracting Results

Once `problem.solve()` completes, the solution is stored in the `.value` attribute of each variable:

```python
# Extract optimal temperature trajectory
T_optimal = T.value
print(f"Temperature trajectory: {T_optimal}")

# Extract optimal control sequence
u_optimal = u_kW.value
print(f"Control sequence (kW): {u_optimal}")

# MPC receding horizon: apply only the first control action
u_apply = u_optimal[0]
print(f"\nControl to apply now: {u_apply:.2f} kW")
```

```
Temperature trajectory: [20.         20.36       20.70704    20.99634105 20.99970444 20.9997
 20.999744   20.999744   20.999744   20.999744   20.999744   20.999744
 20.999744   20.999744   20.999744   20.999744   20.999744   20.999744
 20.999744   20.999744   20.999744   20.99974399 20.99974309 20.99966527
 20.99297167]
Control sequence (kW): [5.         5.         4.74863604 3.21795373 3.20015812 3.19995123
 3.19994883 3.1999488  3.1999488  3.1999488  3.1999488  3.1999488
 3.1999488  3.1999488  3.1999488  3.1999488  3.1999488  3.1999488
 3.1999488  3.1999488  3.19994874 3.19994377 3.19951629 3.16274642]

Control to apply now: 5.00 kW
```

Interpreting the results:

- `problem.value`: The optimal value of the cost function
- `problem.status`: Solution status (e.g., "optimal", "infeasible", "unbounded")
- `T.value`: Array of optimal temperatures $[T_0, T_1, \ldots, T_N]$
- `u_kW.value`: Array of optimal control inputs in kW $[u_0, u_1, \ldots, u_{N-1}]$

Important notes:

1. **Check the status**: Always verify that `problem.status` is "optimal" before using the solution
2. **Receding horizon**: In practice, you only implement `u_optimal[0]`, then re-solve the problem at the next timestep with new measurements
3. **Solver selection**: cvxpy automatically selects a solver, but you can specify one: `problem.solve(solver=cp.OSQP)`

14.5.4 Complete Example: MPC Controller for 1R1C Model

Here's a complete, runnable example that brings together everything we've discussed. This implements an MPC controller as a Python function that can be called repeatedly at each timestep:

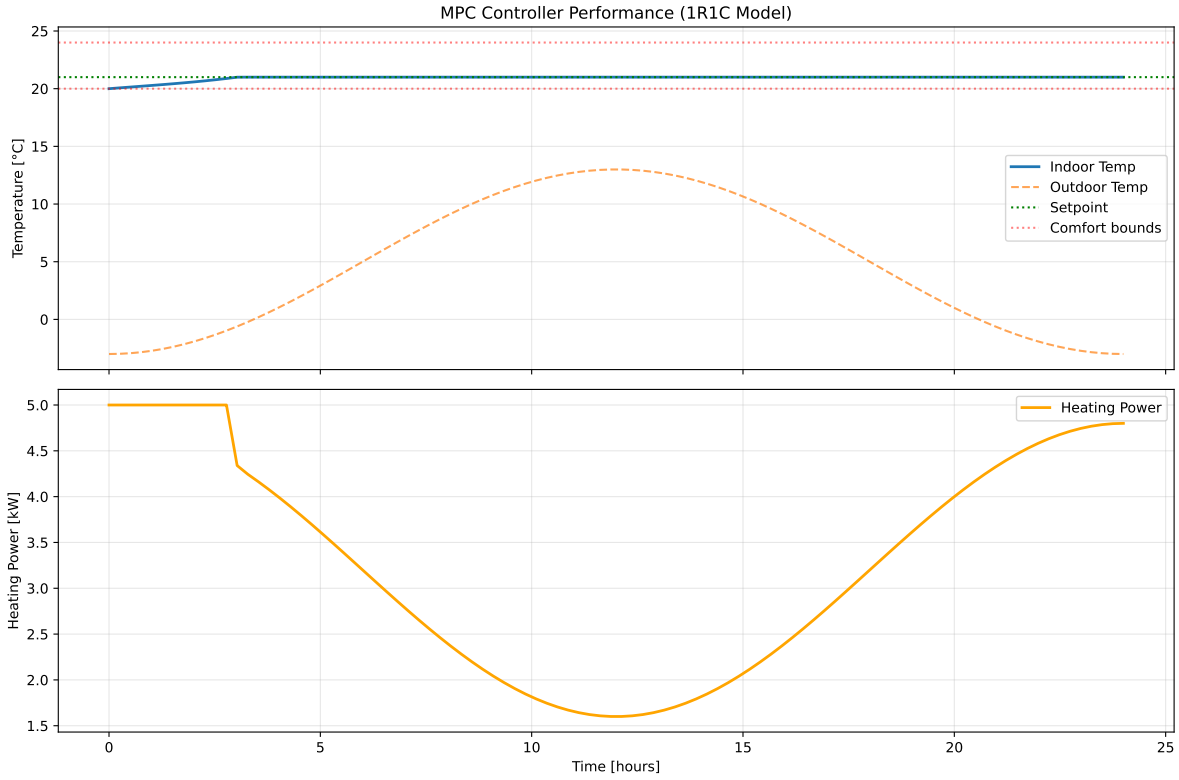
```
import cvxpy as cp
import numpy as np
import matplotlib.pyplot as plt

def mpc_controller_1R1C(T_current, T_amb_forecast, params):
    """
    MPC controller for a 1R1C thermal model.

    Parameters
    ----------
    T_current : float
        Current measured indoor temperature [°C]
    T_amb_forecast : array
        Forecast of ambient temperature over horizon [°C]
    params : dict
        Dictionary containing:
        - 'R': Thermal resistance [K/W]
        - 'C': Thermal capacitance [J/K]
        - 'dt': Timestep [seconds]
        - 'N': Prediction horizon
        - 'T_set': Temperature setpoint [°C]
        - 'T_min': Minimum comfort temperature [°C]
        - 'T_max': Maximum comfort temperature [°C]
        - 'P_max_kW': Maximum heating power [kW]
        - 'w_comfort': Weight on comfort cost
        - 'w_energy': Weight on energy cost
        - 'w_terminal': Weight on terminal cost

    Returns
    -------
    u_optimal : array
        Optimal control sequence [kW]
    T_optimal : array
        Predicted temperature trajectory [°C]
    """
    # Extract parameters
    R = params['R']
```

```
    C = params['C']
    dt = params['dt']
    N = params['N']
    T_set = params['T_set']
    T_min = params['T_min']
    T_max = params['T_max']
    P_max_kW = params['P_max_kW']
    w_comfort = params['w_comfort']
    w_energy = params['w_energy']
    w_terminal = params['w_terminal']

    # Compute discrete-time model coefficients
    a = 1 - dt/(R*C)
    b = dt/C * 1000  # Multiplies kW, so scale by 1000
    e = dt/(R*C)

    # Define optimization variables
    T = cp.Variable(N + 1)
    u_kW = cp.Variable(N)

    # Build constraints
    constraints = []

    # Initial condition
    constraints += [T[0] == T_current]

    # Dynamics
    for k in range(N):
        constraints += [T[k+1] == a*T[k] + b*u_kW[k] + e*T_amb_forecast[k]]

    # Temperature bounds
    constraints += [T[1:] >= T_min]
    constraints += [T[1:] <= T_max]

    # Control bounds
    constraints += [u_kW >= 0]
    constraints += [u_kW <= P_max_kW]

    # Define cost function
    comfort_cost = w_comfort * cp.sum_squares(T[:-1] - T_set)
    energy_cost = w_energy * cp.sum_squares(u_kW / P_max_kW)
    terminal_cost = w_terminal * cp.sum_squares(T[-1] - T_set)
```

```
    total_cost = comfort_cost + energy_cost + terminal_cost

    # Solve
    problem = cp.Problem(cp.Minimize(total_cost), constraints)
    problem.solve(verbose=False)

    # Check solution status
    if problem.status not in ["optimal", "optimal_inaccurate"]:
        print(f"Warning: Optimization status = {problem.status}")
        return None, None

    return u_kW.value, T.value


# Model parameters (example values)
params = {
    'R': 0.005,         # [K/W]
    'C': 5e6,           # [J/K]
    'dt': 900,          # [s] = 15 minutes
    'N': 24,            # 24 steps = 6 hours ahead
    'T_set': 21.0,      # [°C]
    'T_min': 20.0,      # [°C]
    'T_max': 24.0,      # [°C]
    'P_max_kW': 5.0,    # [kW]
    'w_comfort': 1.0,
    'w_energy': 0.01,
    'w_terminal': 1.0
}

# Simulation setup
sim_hours = 24
sim_steps = int(sim_hours * 3600 / params['dt'])

# Generate ambient temperature profile (cold morning, warmer afternoon)
time_hours = np.linspace(0, sim_hours, sim_steps)
T_amb_actual = 5 + 8*np.sin(2*np.pi*(time_hours - 6)/24)  # [°C]

# Initialize state
T_current = 20.0  # Start at 20°C

# Storage for results
T_history = [T_current]
```

```
u_history = []

# MPC simulation loop
for step in range(sim_steps):
    # Create forecast (here we assume perfect forecast)
    forecast_end = min(step + params['N'], sim_steps)
    forecast_length = forecast_end - step
    T_amb_forecast = T_amb_actual[step:forecast_end]

    # Pad forecast if needed
    if forecast_length < params['N']:
        T_amb_forecast = np.concatenate([
            T_amb_forecast,
            np.full(params['N'] - forecast_length, T_amb_forecast[-1])
        ])

    # Solve MPC problem
    u_opt, T_pred = mpc_controller_1R1C(T_current, T_amb_forecast, params)

    if u_opt is None:
        print(f"MPC failed at step {step}")
        break

    # Apply first control action (receding horizon)
    u_apply = u_opt[0]
    u_history.append(u_apply)

    # Simulate actual system response (using same 1R1C model)
    a = 1 - params['dt']/(params['R']*params['C'])
    b = params['dt']/params['C'] * 1000  # u_apply is in kW
    e = params['dt']/(params['R']*params['C'])

    T_current = a*T_current + b*u_apply + e*T_amb_actual[step]
    T_history.append(T_current)

# Plot results
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 8), sharex=True)

# Temperature plot
ax1.plot(time_hours, T_history[:-1], label='Indoor Temp', linewidth=2)
ax1.plot(time_hours, T_amb_actual, label='Outdoor Temp', linestyle='--', alpha=0.7)
ax1.axhline(params['T_set'], color='green', linestyle=':', label='Setpoint')
```

```
ax1.axhline(params['T_min'], color='red', linestyle=':', alpha=0.5, label='Comfort bounds')
ax1.axhline(params['T_max'], color='red', linestyle=':', alpha=0.5)
ax1.set_ylabel('Temperature [°C]')
ax1.legend()
ax1.grid(True, alpha=0.3)
ax1.set_title('MPC Controller Performance (1R1C Model)')

# Control input plot
ax2.plot(time_hours, u_history, label='Heating Power', linewidth=2, color='orange')
ax2.set_xlabel('Time [hours]')
ax2.set_ylabel('Heating Power [kW]')
ax2.legend()
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Print summary statistics
print(f"\nSimulation Summary:")
print(f"  Mean indoor temperature: {np.mean(T_history[:-1]):.2f} °C")
print(f"  Temperature std dev: {np.std(T_history[:-1]):.2f} °C")
print(f"  Total energy used: {np.sum(u_history)*params['dt']/3600:.2f} kWh")
print(f"  Average power: {np.mean(u_history):.2f} kW")
```




```
Simulation Summary:
  Mean indoor temperature: 20.93 °C
  Temperature std dev: 0.22 °C
  Total energy used: 78.23 kWh
  Average power: 3.26 kW
```

You can now re-run this to experiment with a few different things, including:

- Cost function weights (`w_comfort`, `w_energy`)
- Prediction horizon (`N`)
- Comfort bounds (`T_min`, `T_max`)
- Model parameters (`R`, `C`)

14.6 Building Optimization Testing Framework (BOPTEST)

The MPC controller we developed in the previous section works well in simulation with our simple 1R1C model. But how would it perform in a real building with more complex dynamics,

weather variations, and occupancy patterns? More importantly, how can we objectively compare different control strategies to determine which is best?

This is where **BOPTEST** (Building Optimization Testing Framework) comes in. BOPTEST provides a standardized platform for testing and benchmarking building control algorithms using realistic building simulations. Think of it as a "digital twin" testbed where you can evaluate your controller's performance before deploying it to an actual building.

14.6.1 The Need for Standardized Benchmarking

Evaluating building control algorithms is challenging for several reasons:

Problem 1: Lack of reproducibility

- Different researchers test on different buildings with different characteristics
- Weather conditions, occupancy patterns, and baseline systems vary
- Results from one study can't be directly compared to another
- "My controller saved 30% energy" is meaningless without context

Problem 2: Real-world deployment barriers

- Testing experimental controllers on real buildings is risky and expensive
- Building owners are reluctant to allow untested algorithms to control their HVAC
- Seasonal variations mean you might wait months to evaluate performance
- Safety and comfort constraints limit what experiments you can run

Problem 3: Fair comparison difficulties

- Even with the same building, comparing controllers requires identical conditions
- You can't replay the exact same weather twice in a real building
- Baseline controller performance varies with tuning
- Implementation details (sampling rates, sensor noise) affect results

BOPTEST's solution:

BOPTEST addresses these challenges by providing:

1. **Standardized test cases**: Realistic building models representing different archetypes (residential, commercial, etc.)
2. **Reproducible conditions**: Exact same weather, occupancy, and disturbances every time
3. **Fair comparison**: All controllers tested on identical virtual buildings
4. **Comprehensive metrics**: Standardized KPIs (energy, comfort, cost, emissions) for evaluation
5. **Open access**: Free, open-source framework anyone can use

This enables **apples-to-apples comparisons** where the only variable is the control algorithm itself.

14.6.2 What is BOPTEST?

BOPTEST is an open-source software framework developed through an international collaboration (IBPSA Project 1) that provides:

Core Components:

1. **Building Emulators**: High-fidelity building models implemented in Modelica (a physical modeling language)
    - Models include detailed HVAC systems, envelope dynamics, occupancy, and weather
    - Simulate at sub-minute timesteps for realistic response
    - Include measurement noise and actuator dynamics
2. **REST API**: Standard web-based interface for controller interaction
    - Controllers send control signals via HTTP requests
    - Receive measurements and forecasts in response
    - Language-agnostic (works with Python, MATLAB, Julia, etc.)
3. **KPI Calculation Engine**: Automatic computation of performance metrics
    - Energy consumption (heating, cooling, total)
    - Thermal comfort violations (PMV, operative temperature)
    - Economic cost (based on time-of-use pricing)
    - CO emissions

How it works:

```
                HTTP/REST API
Your Control                      BOPTEST Service
 Algorithm       - Send control inputs         (Building Model)
(Python/etc.)    - Get measurements             + KPI tracker
         - Get forecasts
```

The controller you develop runs independently and communicates with the BOPTEST service through simple HTTP requests. This means you can develop in any language and test against the same standardized buildings.

14.6.3 BOPTEST Test Cases

BOPTEST includes multiple standardized test cases representing different building types and HVAC systems. Each test case is a complete building model with:

- Defined geometry and thermal properties
- Specified HVAC system type and sizing
- Weather data (typically a full year)
- Occupancy schedules
- Baseline controller for comparison

Example Test Cases:

1. `bestest_hydronic_heat_pump`
    - Single-zone residential building
    - Heat pump with radiant floor heating
    - Well-insulated envelope
    - Located in Belgium (temperate climate)
    - Good for: Testing basic heating control strategies
2. `bestest_air`
    - Single-zone residential building
    - Forced air heating and cooling
    - Variable air volume (VAV) system
    - Good for: Testing air-based HVAC control
3. `multizone_residential_hydronic`
    - Multi-zone residential building
    - Multiple rooms with different usage patterns
    - Hydronic distribution system
    - Good for: Testing zone-level control coordination
4. `multizone_office_simple_air`
    - Office building with multiple zones
    - Simple air handling unit
    - Occupancy-driven loads
    - Good for: Testing commercial building control

Each test case comes with: - Detailed documentation of the building and systems - List of available measurements (temperatures, power, etc.) - List of controllable inputs (setpoints, valve positions, etc.) - Example baseline controller implementation

14.6.4 BOPTEST API Overview

Controllers interact with BOPTEST through a RESTful HTTP API. All communication happens via standard HTTP requests using the `requests` library in Python (or equivalent in other languages).

Base workflow:

1. **Select** a test case and get a unique test ID
2. **Initialize** the simulation to starting conditions
3. **Loop**: Get measurements → Compute control → Send control → Advance simulation
4. **Retrieve** KPIs to evaluate performance

All API endpoints follow the pattern: `http://<server>/<endpoint>/<testid>`

14.6.4.1 Major API Endpoints

14.6.4.1.1 Select Test Case: `/testcases/{testcase}/select`

Purpose: Start a new test instance

Method: POST

Example:

```
import requests

# BOPTEST web service URL (course server)
url = 'http://13.217.71.86'
testcase = 'bestest_hydronic_heat_pump'

response = requests.post(f'{url}/testcases/{testcase}/select')
testid = response.json()['testid']
print(f"Test ID: {testid}")
```

```
Test ID: 945294a6-2ac7-4239-9dbf-96d33a6de1ee
```

Returns: A unique `testid` for this test instance

14.6.4.1.2 Initialize: `/initialize/{testid}`

Purpose: Reset simulation to starting conditions (typically January 1, midnight)

Method: PUT (JSON body)

Parameters: - `start_time`: Simulation start time [seconds] - `warmup_period`: Warmup period before test starts [seconds]

Example:

```
# Initialize with default start time
requests.put(f'{url}/initialize/{testid}', json={'start_time': 0, 'warmup_period': 0})

# Or specify start time (e.g., start in summer)
#start_june = 15*24*3600  # 15 days into year
#requests.put(f'{url}/initialize/{testid}', json={'start_time': start_june, 'warmup_period':
```

```
<Response [200]>
```

14.6.4.1.3 Get Measurements: Included in `/advance` response

Purpose: Retrieve current sensor readings

Note: Measurements are automatically returned with each `/advance` call

Available measurements depend on test case, typically include: - Zone temperatures (`reaTZon_y`) - Outdoor temperature (`weaSta_reaWeaTDryBul_y`) - HVAC power consumption (`reaPHeaPum_y`, `reaPCoo_y`) - CO concentration, humidity, etc.

14.6.4.1.4 Get Inputs: `/inputs/{testid}`

Purpose: Query what control inputs are available

Method: GET

Example:

```
inputs = requests.get(f'{url}/inputs/{testid}').json()['payload']
for name, details in inputs.items():
    print(f"{name}: {details['Description']} [{details['Unit']}]")
```

```
oveFan_activate: Activation for Integer signal to control the heat pump evaporator fan either
oveFan_u: Integer signal to control the heat pump evaporator fan either on or off [1]
oveHeaPumY_activate: Activation for Heat pump modulating signal for compressor speed between
oveHeaPumY_u: Heat pump modulating signal for compressor speed between 0 (not working) and 1
ovePum_activate: Activation for Integer signal to control the emission circuit pump either or
ovePum_u: Integer signal to control the emission circuit pump either on or off [1]
oveTSet_activate: Activation for Zone operative temperature setpoint [None]
oveTSet_u: Zone operative temperature setpoint [K]
```

Returns: Dictionary of available control inputs with descriptions and units

14.6.4.1.5 Advance: `/advance/{testid}`

Purpose: Send control inputs and step simulation forward

Method: POST

Parameters: Control input values (see `/inputs` for available controls)

Example:

```python
# Set heat pump modulation to 50%
control_inputs = {
    'oveHeaPumY_u': 0.5,          # Control value
    'oveHeaPumY_activate': 1      # Activate override (1=on, 0=off)
}

response = requests.post(f'{url}/advance/{testid}', json=control_inputs)
measurements = response.json()['payload']

# Extract specific measurements
T_zone = measurements['reaTZon_y']
P_heating = measurements['reaPHeaPum_y']
```

Returns: Dictionary of all current measurements

> **Control Input Format**
>
> BOPTEST uses an "override" pattern for control inputs:
>
> - `<signal>_u`: The control value you want to set
> - `<signal>_activate`: Set to 1 to override, 0 to use default controller
>
> This allows you to control only specific aspects while leaving others to the baseline controller.

14.6.4.1.6 Get Forecast: `/forecast/{testid}`

Purpose: Retrieve predictions of future disturbances (weather, occupancy)

Method: PUT (JSON body)

Parameters:

- `point_names`: List of forecast variable names (e.g., `['TDryBul']`)
- `horizon`: Forecast horizon [seconds]
- `interval`: Time between forecast points [seconds]

Example:

```
# Get 6-hour weather forecast at hourly intervals
forecast_response = requests.put(
    f'{url}/forecast/{testid}',
    json={
        'point_names': ['TDryBul'],
        'horizon': 6*3600,       # 6 hours
        'interval': 3600          # 1 hour intervals
    }
)

forecast = forecast_response.json()['payload']
T_amb_forecast = forecast['TDryBul']  # Future outdoor temps [K]
```

Returns: Dictionary of forecasted variables with time series values

14.6.4.1.7 Get KPIs: `/kpi/{testid}`

Purpose: Retrieve performance metrics for the test run

Method: GET

Example:

```
kpis = requests.get(f'{url}/kpi/{testid}').json()['payload']

print(f"Total Energy: {kpis['ener_tot']:.2f} kWh")
print(f"Thermal Discomfort: {kpis['tdis_tot']:.2f} Kh")
print(f"Operating Cost: {kpis['cost_tot']:.2f} EUR")
print(f"CO2 Emissions: {kpis['emis_tot']:.2f} kg")
```

```
Total Energy: 0.01 kWh
Thermal Discomfort: 1.76 Kh
Operating Cost: 0.00 EUR
CO2 Emissions: 0.00 kg
```

Returns: Dictionary of KPIs accumulated since initialization

Common KPIs:

- `ener_tot`: Total energy [kWh]
- `tdis_tot`: Thermal discomfort [Kh] (degree-hours outside comfort bounds)
- `cost_tot`: Operating cost [currency]
- `emis_tot`: CO emissions [kg]
- Individual components: `ener_heat`, `ener_cool`, etc.

14.6.5 Testing a Controller Against a BOPTEST Test Case

Now let's walk through the complete process of testing a controller on BOPTEST, from setup to evaluation.

14.6.5.1 Setting Up BOPTEST

For this course, we're using a **web-based BOPTEST service** hosted at `http://13.217.71.86`. This means:

- **No local installation required** - the server is already running
- **No Docker setup needed** - just use HTTP requests
- **Accessible from any machine** - as long as you have internet access

What you need:

1. Python with the `requests` library:

```
# Install if needed
# pip install requests numpy matplotlib cvxpy
import requests
```

2. The server URL:

```
url = 'http://13.217.71.86'
```

That's it! You're ready to start testing controllers.

Local BOPTEST Installation (Optional)

If you want to run BOPTEST locally (for offline work or custom test cases), you can install it via Docker:

```
git clone https://github.com/ibpsa/project1-boptest.git
cd project1-boptest
docker-compose up
```

Then use `url = 'http://localhost:5000'` instead of the course server.

14.6.5.2 Implementing a Controller Interface

To test a controller on BOPTEST, you need to structure your code to:

1. **Initialize** the test
2. **Loop** over the simulation period
3. At each timestep:
    - Get current measurements
    - Compute control action
    - Send control and advance simulation
4. **Retrieve** final KPIs

Here's a template function that wraps this logic:

```
def test_controller_on_boptest(url, testcase, controller, simulation_days=1, step_size=300):
    """
    Test a controller on a BOPTEST test case.

    Parameters
    ----------
    url : str
        BOPTEST service URL
    testcase : str
        Test case name (e.g., 'bestest_hydronic_heat_pump')
    controller : callable
        Controller function with signature: u = controller(y, url, testid)
        where y is current measurements dict, u is control inputs dict
    simulation_days : float
        Number of days to simulate
```

```
    step_size : int
        Control timestep [seconds]

    Returns
    -------
    results : dict
        Dictionary containing time series data and KPIs
    """
    import numpy as np

    # Select test case
    response = requests.post(f'{url}/testcases/{testcase}/select')
    testid = response.json()['testid']
    print(f"Test ID: {testid}")

    # Initialize simulation
    requests.put(f'{url}/initialize/{testid}', json={'start_time': 0, 'warmup_period': 0})

    # Set simulation step size
    requests.put(f'{url}/step/{testid}', json={'step': step_size})

    # Storage for results
    time_data = []
    temp_data = []
    control_data = []
    power_data = []

    # Calculate number of steps
    total_steps = int(simulation_days * 24 * 3600 / step_size)

    # Get initial measurements
    y = requests.post(f'{url}/advance/{testid}', json={}).json()['payload']

    # Simulation loop
    for step in range(total_steps):
        # Compute control action
        u = controller(y, url, testid)

        # Apply control and advance
        y = requests.post(f'{url}/advance/{testid}', json=u).json()['payload']

        # Store data
```

```
        time_data.append(step * step_size / 3600)  # hours
        temp_data.append(y['reaTZon_y'] - 273.15)  # Convert K to C
        control_data.append(u.get('oveHeaPumY_u', 0))
        power_data.append(y.get('reaPHeaPum_y', 0))

        if step % 100 == 0:
            print(f"Progress: {step}/{total_steps} steps")

    # Get final KPIs
    kpis = requests.get(f'{url}/kpi/{testid}').json()['payload']

    return {
        'time': np.array(time_data),
        'temperature': np.array(temp_data),
        'control': np.array(control_data),
        'power': np.array(power_data),
        'kpis': kpis,
        'testid': testid
    }
```

Key points:

- Controller function receives measurements `y` and must return control dict `u`
- The `url` and `testid` are passed to allow controllers to query forecasts
- Results include both time series data and final KPIs

14.6.5.3 Running a Test

With the template above, running a test is straightforward. First, define your controller:

```
def simple_proportional_controller(y, url=None, testid=None):
    """
    Simple proportional controller for BOPTEST.
    """
    T_current = y['reaTZon_y']  # Current zone temp [K]
    T_set = 273.15 + 21.0       # Setpoint [K]
    K_p = 2.0                   # Proportional gain [1/K]

    # Compute control (clamp to 0-1 range for BOPTEST)
    error = T_set - T_current
    u_heat = max(0.0, min(K_p * error, 1.0))
```

```
    # Return in BOPTEST format
    return {
        'oveHeaPumY_u': u_heat,
        'oveHeaPumY_activate': 1
    }
```

Then run the test:

```
url = 'http://13.217.71.86'
testcase = 'bestest_hydronic_heat_pump'

results = test_controller_on_boptest(
    url=url,
    testcase=testcase,
    controller=simple_proportional_controller,
    simulation_days=0.25,
    step_size=3600  # 1 hour
)

print("\nTest Complete!")
print(f"Total Energy: {results['kpis']['ener_tot']:.2f} kWh")
print(f"Thermal Discomfort: {results['kpis']['tdis_tot']:.2f} Kh")
```

```
Test ID: 2c0a5ce5-f8b0-43f5-a5f7-7f733741bb07
Progress: 0/6 steps

Test Complete!
Total Energy: 0.13 kWh
Thermal Discomfort: 7.02 Kh
```

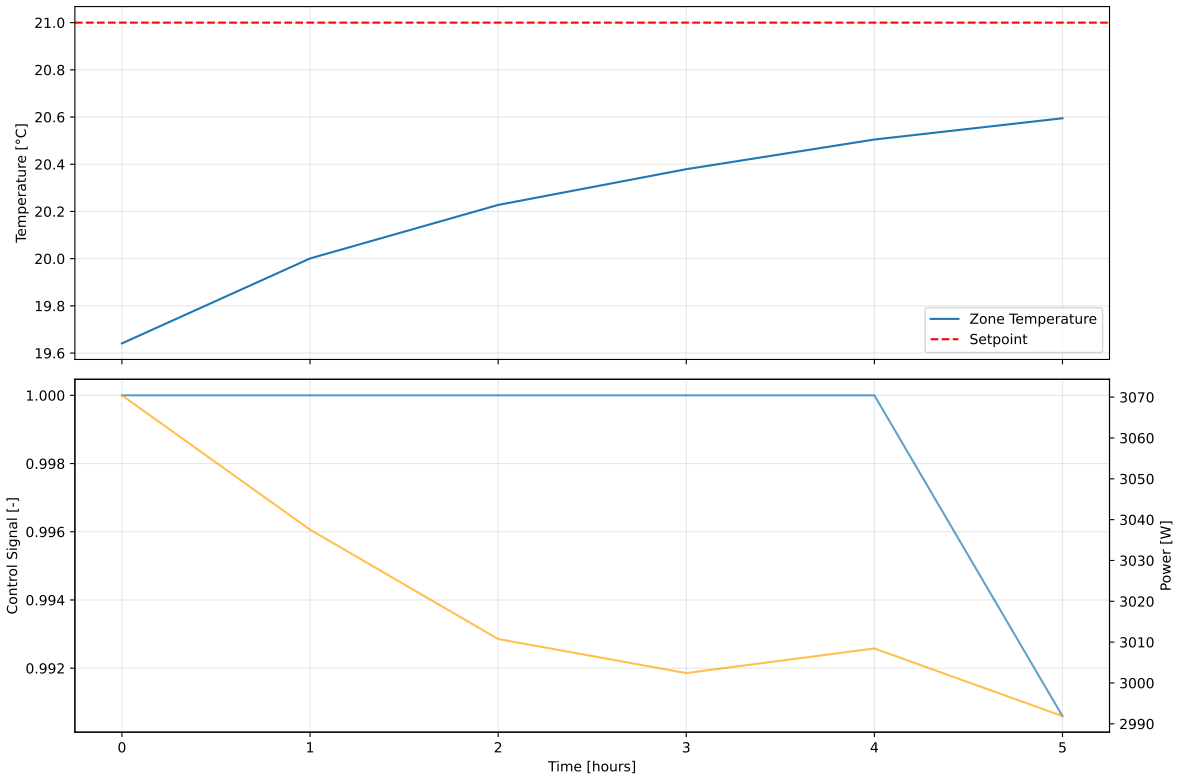
14.6.5.4 Evaluating Performance

After running a test, you can analyze the results in multiple ways:

1. Visualize time series data:

```
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 8), sharex=True)

# Temperature
```

```
ax1.plot(results['time'], results['temperature'], label='Zone Temperature')
ax1.axhline(21, color='red', linestyle='--', label='Setpoint')
ax1.set_ylabel('Temperature [°C]')
ax1.legend()
ax1.grid(True, alpha=0.3)

# Control and power
ax2.plot(results['time'], results['control'], label='Control Signal', alpha=0.7)
ax2_twin = ax2.twinx()
ax2_twin.plot(results['time'], results['power'], 'orange', label='Power', alpha=0.7)
ax2.set_xlabel('Time [hours]')
ax2.set_ylabel('Control Signal [-]')
ax2_twin.set_ylabel('Power [W]')
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```



2. Compare KPIs across controllers:

```
# Run multiple controllers
controllers = {
    'Proportional': simple_proportional_controller,
    'MPC': mpc_controller_for_boptest,  # Your MPC implementation
}

comparison = {}
for name, controller in controllers.items():
    print(f"\nTesting {name} controller...")
    results = test_controller_on_boptest(url, testcase, controller)
    comparison[name] = results['kpis']

# Print comparison table
print("\n" + "="*70)
print(f"{'Controller':<20} {'Energy [kWh]':<15} {'Discomfort [Kh]':<15} {'Cost':<10}")
print("="*70)
for name, kpis in comparison.items():
    print(f"{name:<20} {kpis['ener_tot']:<15.2f} {kpis['tdis_tot']:<15.2f} {kpis['cost_tot']
```

3. Interpret KPI meanings:

- **`ener_tot`**: Total energy consumption [kWh] - lower is better
- **`tdis_tot`**: Thermal discomfort [Kh] - time-weighted temperature deviation outside comfort bounds (20-24°C) - lower is better
- **`cost_tot`**: Operating cost based on time-of-use pricing - lower is better
- **`emis_tot`**: CO emissions - lower is better

Trade-offs to consider:

- Minimizing energy may increase discomfort
- Tight temperature control may increase energy use
- Cost depends on shifting loads to off-peak hours
- Best controller balances multiple objectives

14.6.6 Example: MPC Controller on BOPTEST

Now let's adapt our MPC controller to work with BOPTEST. The key difference from our earlier example is that we need to:

1. Work with BOPTEST's actual building model (not our simple 1R1C approximation)
2. Use BOPTEST's forecast API to get weather predictions
3. Return control inputs in BOPTEST format

Here's a complete implementation:

```
import cvxpy as cp
import numpy as np
import requests

def mpc_controller_for_boptest(y, url, testid,
                               horizon=6,
                               T_set=273.15+21,
                               w_comfort=10.0,
                               w_energy=0.1):
    """
    MPC controller for BOPTEST using simplified 1R1C thermal model.

    Parameters
    ----------
    y : dict
        Current measurements from BOPTEST
    url : str
        BOPTEST service URL
    testid : str
        Test case ID
    horizon : int
        Prediction horizon [timesteps of 2 hours each]
    T_set : float
        Temperature setpoint [K]
    w_comfort : float
        Weight on comfort in cost function
    w_energy : float
        Weight on energy in cost function

    Returns
    -------
    u : dict
        Control inputs for BOPTEST
    """

    # Get current state
    T_current = y['reaTZon_y']  # Zone temperature [K]

    # Get weather forecast from BOPTEST
    try:
        forecast_response = requests.put(
```

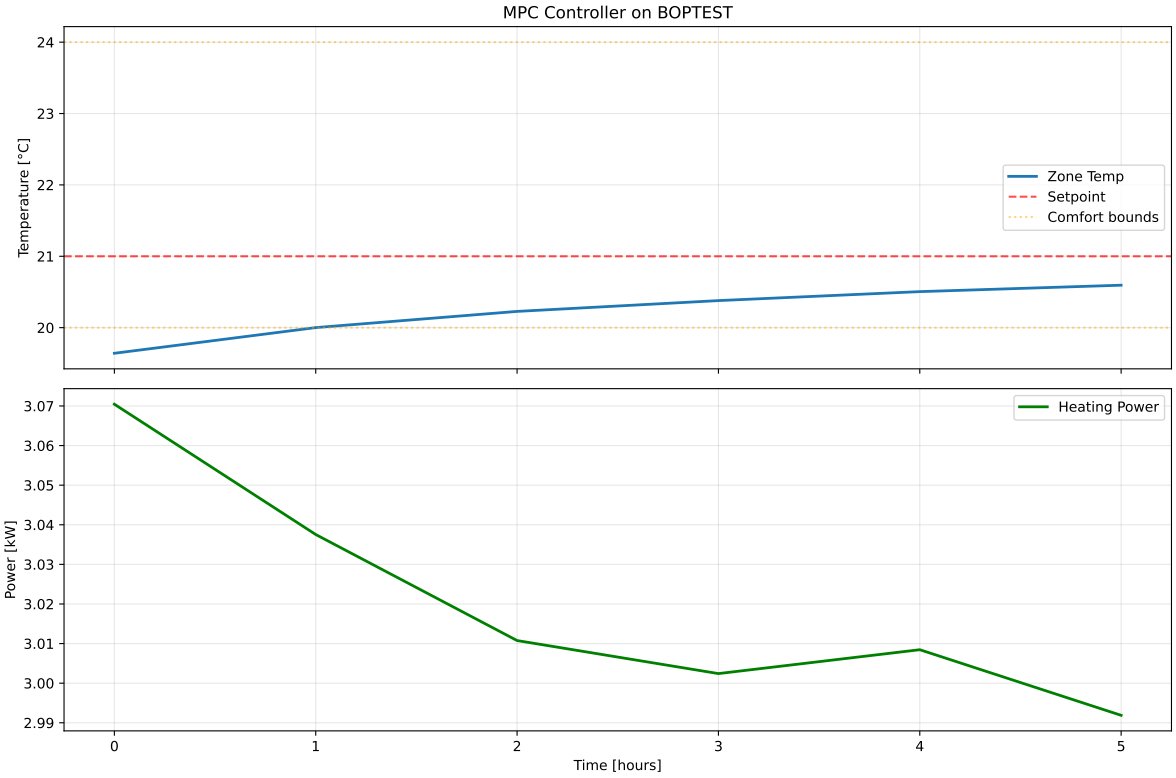
```
            f'{url}/forecast/{testid}',
            json={
                'point_names': ['TDryBul'],
                'horizon': horizon * 3600,
                'interval': 3600
            }
        )
        if forecast_response.status_code == 200:
            forecast_data = forecast_response.json()['payload']
            T_amb_forecast = np.array(forecast_data['TDryBul'][:horizon])
        else:
            T_amb_forecast = np.full(horizon, 273.15 + 1.0)
    except:
        T_amb_forecast = np.full(horizon, 273.15 + 1.0)

    # Simplified building model parameters (approximation)
    # These would ideally be identified from BOPTEST data
    R = 0.002     # Thermal resistance [K/W]
    C = 2e7       # Thermal capacitance [J/K]
    dt = 3600     # Timestep [s] (1 hour)
    P_max_kW = 10.0  # Max heating power [kW]

    # Discrete model coefficients
    a = 1 - dt/(R*C)
    b = dt/C * 1000  # Multiplies kW, so scale by 1000
    e = dt/(R*C)

    # Define optimization variables
    T = cp.Variable(horizon + 1)
    u_kW = cp.Variable(horizon)

    # Constraints
    constraints = []
    constraints += [T[0] == T_current]  # Initial condition

    # Dynamics
    for k in range(horizon):
        constraints += [T[k+1] == a*T[k] + b*u_kW[k] + e*T_amb_forecast[k]]

    # Temperature bounds (comfort)
    T_min = 273.15 + 19  # 19°C (relaxed for feasibility)
    T_max = 273.15 + 24  # 24°C
```

```
    constraints += [T[1:] >= T_min]
    constraints += [T[1:] <= T_max]

    # Control bounds
    constraints += [u_kW >= 0]
    constraints += [u_kW <= P_max_kW]

    # Cost function
    comfort_cost = w_comfort * cp.sum_squares(T[:-1] - T_set)
    energy_cost = w_energy * cp.sum_squares(u_kW / P_max_kW)
    terminal_cost = w_comfort * cp.sum_squares(T[-1] - T_set)
    total_cost = comfort_cost + energy_cost + terminal_cost

    # Solve
    problem = cp.Problem(cp.Minimize(total_cost), constraints)
    try:
        problem.solve(verbose=False)

        if problem.status in ["optimal", "optimal_inaccurate"]:
            u_opt_kW = u_kW.value[0]  # Take first control action
            # Normalize to 0-1 range for BOPTEST
            u_normalized = float(np.clip(u_opt_kW / P_max_kW, 0.0, 1.0))
        else:
            # Fallback to proportional control
            error = max(0, T_set - T_current)
            u_normalized = min(2.0 * error, 1.0)
    except:
        # Fallback to proportional control
        error = max(0, T_set - T_current)
        u_normalized = min(2.0 * error, 1.0)

    # Return in BOPTEST format
    return {
        'oveHeaPumY_u': u_normalized,
        'oveHeaPumY_activate': 1
    }


# Run MPC controller on BOPTEST
url = 'http://13.217.71.86'
testcase = 'bestest_hydronic_heat_pump'
```

```
print("Testing MPC controller on BOPTEST...")
mpc_results = test_controller_on_boptest(
    url=url,
    testcase=testcase,
    controller=mpc_controller_for_boptest,
    simulation_days=0.25,
    step_size=3600  # 1 hour
)

print("\n" + "="*50)
print("MPC Controller Results:")
print("="*50)
print(f"Total Energy: {mpc_results['kpis']['ener_tot']:.2f} kWh")
print(f"Thermal Discomfort: {mpc_results['kpis']['tdis_tot']:.2f} Kh")
print(f"Operating Cost: {mpc_results['kpis']['cost_tot']:.2f} EUR")
print(f"CO2 Emissions: {mpc_results['kpis']['emis_tot']:.2f} kg")

# Plot results
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 8), sharex=True)

ax1.plot(mpc_results['time'], mpc_results['temperature'], label='Zone Temp', linewidth=2)
ax1.axhline(21, color='red', linestyle='--', label='Setpoint', alpha=0.7)
ax1.axhline(20, color='orange', linestyle=':', label='Comfort bounds', alpha=0.5)
ax1.axhline(24, color='orange', linestyle=':', alpha=0.5)
ax1.set_ylabel('Temperature [°C]')
ax1.legend()
ax1.grid(True, alpha=0.3)
ax1.set_title('MPC Controller on BOPTEST')

ax2.plot(mpc_results['time'], mpc_results['power']/1000, label='Heating Power',
         linewidth=2, color='green')
ax2.set_xlabel('Time [hours]')
ax2.set_ylabel('Power [kW]')
ax2.legend()
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

```
Testing MPC controller on BOPTEST...
```

```
Test ID: 7494c3df-476e-405e-b3d5-6471bb4be2d4
Progress: 0/6 steps

================================================
MPC Controller Results:
================================================
Total Energy: 0.13 kWh
Thermal Discomfort: 7.02 Kh
Operating Cost: 0.03 EUR
CO2 Emissions: 0.02 kg
```



Key implementation notes:

1. **Model mismatch**: We use a simple 1R1C model even though BOPTEST has a more complex building. This is intentional - it demonstrates MPC's robustness to model mismatch.

2. **Forecast integration**: The controller queries BOPTEST's forecast API to get future weather predictions, which is crucial for MPC's anticipatory behavior.

3. **Fallback strategy**: If the optimization fails, the controller falls back to simple proportional control. This ensures robustness.
4. **Normalization**: BOPTEST expects control signals in the 0-1 range, so we normalize the power output.
5. **Error handling**: The code includes try-except blocks to handle API failures gracefully.

14.7 Additional Resources

14.7.1 Key References

- Blum, David, Javier Arroyo, Sen Huang, Jan Drgona, Filip Jorissen, Harald Taxt Walnum, Yan Chen, et al. "Building Optimization Testing Framework (BOPTEST) for Simulation-Based Benchmarking of Control Strategies in Buildings." *Journal of Building Performance Simulation* 14, no. 5 (September 3, 2021): 586–610. https://doi.org/10.1080/19401493.2021.1986574
- Drgoňa et al. "All you need to know about model predictive control for buildings" *Annual Reviews in Control*, Volume 50 (2020): 190-232. https://www.sciencedirect.com/science/article/pii/S1367578820300584

14.7.2 Software Tools

- **cvxpy**: https://www.cvxpy.org/
- **BOPTEST**: https://github.com/ibpsa/project1-boptest

Part IV

State-of-the-Art Applications

15 Paper Discussions

During this part of the course, we shift from foundational concepts to examining how researchers and practitioners apply those concepts to real-world problems. Each week, we will read and discuss one or two papers from the recent literature on smart buildings, energy systems, and related topics. The goal is not just to understand what each paper does, but to critically evaluate the methods, identify connections to what we have learned, and think about what questions remain open.

The papers below are drawn primarily from BuildSys (ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation), one of the leading venues for research at the intersection of computing, sensing, and the built environment. They span topics including occupancy sensing, energy disaggregation, thermal modeling, control, and grid interaction.

Papers marked with a checkmark will be discussed in class. The rest are provided as additional reading for interested students.

15.0.1 Papers Under Consideration

15.0.1.1 Occupancy and Human Sensing

1. **Nonintrusive Occupant Identification by Sensing Body Shape and Movement** — Nacer Khalil, Driss Benhaddou, Omprakash Gnawali, Jaspal Subhlok. *BuildSys 2016.* DOI
2. **Sonicdoor** — Nacer Khalil, Driss Benhaddou, Omprakash Gnawali, Jaspal Subhlok. *BuildSys 2017.* DOI
3. **The Impact of Occupancy Resolution on the Accuracy of Building Energy Performance Simulation** — Fisayo Caleb Sangogboye, Krzysztof Arendt, Muhyiddine Jradi, Christian Veje, Mikkel Baun Kjærgaard, Bo Nørregaard Jørgensen. *BuildSys 2018.* DOI
4. **Robust and Practical WiFi Human Sensing Using On-device Learning with a Domain Adaptive Model** — Elahe Soltanaghaei, Rahul Anand Sharma, Zehao Wang, Adarsh Chittilappilly, Anh Luong, Eric Giler, Katie Hall, Steve Elias, Anthony Rowe. *BuildSys 2020.* DOI

5. **FTM-Sense: Robust Sensor-free Occupancy Sensing Leveraging WiFi Fine Time Measurement** — Fateme Nikseresht, Bradford Campbell. *BuildSys 2023.* DOI

6. **TODOS: Thermal Sensor Data-driven Occupancy Estimation System for Smart Buildings** — Hamid Rajabi, Xianzhong Ding, Wan Du, Alberto Cerpa. *BuildSys 2023.* DOI

7. **ScreenSense: Screen Activity Detection in Real-World Environments with Indoor Light Sensors** — Tushar Routh, Nurani Saoda, Fateme Nikseresht, Md Fazlay Rabbi Masum Billah, Jiechao Gao, Viswajith Govinda Rajan, Bradford Campbell. *BuildSys 2024.* DOI

15.0.1.2 Energy Disaggregation and Monitoring

8. **BOLT: Energy Disaggregation by Online Binary Matrix Factorization of Current Waveforms** — Henning Lange, Mario Bergés. *BuildSys 2016.* DOI

9. **Towards Reproducible State-of-the-Art Energy Disaggregation** — Nipun Batra, Rithwik Kukunuri, Ayush Pandey, Raktim Malakar, Rajat Kumar, Odysseas Krystalakos, Mingjun Zhong, Paulo Meira, Oliver Parson. *BuildSys 2019.* DOI

10. **EffiSenseSee: towards classifying light bulb types and energy efficiency with camera-based sensing** — Alex Yen, Zeal Shah, Benjamin Ochoa, Pat Pannuto, Jay Taneja. *BuildSys 2022.* DOI

11. **Can Attention Improve Sequence-to-Point Load Disaggregation? A Comparative Assessment** — Mazen Bouchur, Nan Li, Andreas Reinhardt. *BuildSys 2025.* DOI

15.0.1.3 Grid and Infrastructure

12. **Hypertemporal Imaging of NYC Grid Dynamics** — Federica B. Bianco, Steven E. Koonin, Charlie Mydlarz, Mohit S. Sharma. *BuildSys 2016.* DOI

13. **Observability: A Principled Approach to Provisioning Sensors in Buildings** — Anshul Agarwal, Vitobha Munigala, Krithi Ramamritham. *BuildSys 2016.* DOI

14. **GridInSight: Monitoring Electricity Using Visible Lights** — Zeal Shah, Alex Yen, Ajey Pandey, Jay Taneja. *BuildSys 2019.* DOI

15. **SolarWalk: smart home occupant identification using unobtrusive indoor photovoltaic harvesters** — Md Fazlay Rabbi Masum Billah, Nurani Saoda, Victor Ariel Leal Sobral, Tushar Routh, Wenpeng Wang, Bradford Campbell. *BuildSys 2022.* DOI

15.0.1.4 Control and Optimization

16. **Data Predictive Control for Peak Power Reduction** — Achin Jain, Rahul Mangharam, Madhur Behl. *BuildSys 2016.* DOI

17. **Gnu-RL: A Precocial Reinforcement Learning Solution for Building HVAC Control Using a Differentiable MPC Policy** — Bingqing Chen, Zicheng Cai, Mario Bergés. *BuildSys 2019.* DOI

18. **SMITE: Using Smart Meters to Infer the Thermal Efficiency of Residential Homes** — Joe Brown, Jonathan Chambers, Alessandro Abate, Alex Rogers. *BuildSys 2020.* DOI

19. **Adversarial Poisoning Attacks on Reinforcement Learning-Driven Energy Pricing** — Sam Gunn, Doseok Jang, Orr Paradise, Lucas Spangher, Costas J. Spanos. *BuildSys 2022.* DOI

20. **RECA: A Multi-Task Deep Reinforcement Learning-Based Recommender System for Co-Optimizing Energy, Comfort and Air Quality in Commercial Buildings** — Stephen Xia, Peter Wei, Yanchen Liu, Andrew Sonta, Xiaofan Jiang. *BuildSys 2023.* DOI

15.0.1.5 Thermal Modeling and Simulation

21. **Unmasking the Thermal Behavior of Single-Zone Multi-Room Houses: An Empirical Study** — Ozan Baris Mulayim, Mario Bergés. *BuildSys 2023.* DOI

22. **Modeling the Impact of Passive Ventilation Systems on Multi-Zone Thermal Dynamics** — James Onyejizu, Sandipan Mishra, Koushik Kar. *BuildSys 2024.* DOI

23. **OmniFlow: A Framework for Generalizable Surrogates for Real-Time Airflow Simulation and Control in Unseen Indoor Environments** — M Tanjid Hasan Tonmoy, Upal Mahbub, Tauhidur Rahman. *BuildSys 2025.* DOI

15.0.1.6 Indoor Environment and Comfort

24. **TEA-bot: a thermography enabled autonomous robot for detecting thermal leaks of HVAC systems in ceilings** — Weijia Cai, Le Zhang, Lei Huang, Xinran Yu, Zhengbo Zou. *BuildSys 2022.* DOI

15.0.1.7 Data Centers

25. **PyDCM: Custom Data Center Models with Reinforcement Learning for Sustainability** — Avisek Naug, Antonio Guillen, Ricardo Luna Gutiérrez, Vineet Gundecha, Sahand Ghorbanpour, Lekhapriya Dheeraj Kashyap, Dejan Markovikj, Lorenz Krause, Sajad Mousavi, Ashwin Ramesh Babu, Soumyendu Sarkar. *BuildSys 2023.* DOI

16 FTM-Sense: Robust Sensor-free Occupancy Sensing Leveraging WiFi Fine Time Measurement

16.1 Overview

Citation

Fateme Nikseresht and Bradford Campbell. 2023. FTM-Sense: Robust Sensor-free Occupancy Sensing Leveraging WiFi Fine Time Measurement. In *The 10th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation (BuildSys '23)*, November 15–16, 2023, Istanbul, Turkey. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3600100.3623741

Sense, Plan, Act Mapping and Objective

Application domain: Smart building (energy) management such as automatic HVAC control, lighting systems, security.

- **Sense**: This is the primary innovation of the paper. FTM-Sense introduces a novel sensing approach that uses WiFi Fine Time Measurement (FTM) protocol to detect human presence through variations in round-trip time (RTT) measurements caused by multipath reflections from the human body. Unlike traditional occupancy sensors (PIR, cameras, IAQ sensors), this approach:
  - Detects static occupants (including those with minimal motion like breathing or typing)
  - Works through walls (NLOS scenarios)
  - Adapts to new environments without retraining
  - Uses existing WiFi infrastructure with low-cost ESP32S2 devices
  - Provides privacy-preserving sensing without cameras or personal device tracking
- **Plan**: Though the occupancy estimates are not used to plan anything leading to the application domains described above, we could think of the specific plan/learn process needed to come up with the occupancy estimates as a proxy. In this sense, the paper employs machine learning classifiers (Random Forest, XGBoost, KNN,

Logistic Regression) to process extracted statistical features (variance, standard deviation, root mean square) from FTM data bundles. The models are trained once on baseline data and are shown to generalize to new environments.

- **Act**: While not the focus of this paper, the accurate occupancy detection enables downstream building automation actions such as automated HVAC control, lighting adjustments, and energy-efficient building operations.

16.2 Review of the paper

16.2.1 Summary

FTM-Sense presents a sensor-free occupancy detection system that leverages WiFi Fine Time Measurement (FTM), a protocol standardized in IEEE 802.11mc. The core innovation is demonstrating empirically, using larger samples than previously presented, how using variations in RTT measurements between two WiFi devices (caused by multipath reflections from human presence) can be used to detect occupancy with high accuracy and the overall performance of this approach (97.7%).

Key contributions:

1. **Static occupant detection**: Unlike PIR sensors that fail to detect motionless occupants, FTM-Sense achieves 97.68% accuracy detecting occupants with only fine motions (typing, reading) and acceptable performance (70.09%) even with very fine motions (breathing only).
2. **Adaptability**: The system adapts to new indoor environments within 80 seconds without requiring retraining. Tested across 7 different rooms (office and residential) with varying configurations, achieving >87.83% accuracy.
3. **NLOS capability**: Can detect occupants through walls and doors (>92% accuracy through wood/drywall, >87% through aluminum sheet).
4. **Low-cost implementation**: Uses standard ESP32S2 boards (~$5 each) with built-in WiFi, no specialized hardware required.

Technical approach:

- Collects FTM measurements across 10 WiFi channels (2.4 GHz band) to avoid interference
- Extracts statistical features (variance, std, rms) from sliding time windows of FTM data
- Trains baseline models (Random Forest and XGBoost perform best) on 3 hours of data from single room
- Generalizes to new environments without retraining

16.2.2 What do we know already?

This paper connects to several concepts covered in earlier lectures:

From Lecture 2 (Why Buildings):

- The energy management motivation: buildings account for 40% of greenhouse gas emissions, and occupancy-based control can significantly reduce energy waste from heating/cooling/lighting unoccupied spaces
- The Sense/Plan/Act framework for autonomous building systems - FTM-Sense provides the "Sense" component

From Lecture 7 (Thermal Comfort):

- We know that occupant needs are the main reason why we cool/heat buildings. So clearly, understanding their behavior and presence is very useful.

General building automation concepts:

- The need for real-time occupancy information to enable demand-based HVAC and lighting control
- Trade-offs between accuracy, cost, privacy, and energy consumption in sensing systems

Things to learn more about

To fully understand this paper's contributions and methods, students may need background on:

1. **WiFi protocols and the physical (PHY) layer**:
    - IEEE 802.11 standards and amendments (specifically 802.11mc)
    - How FTM protocol works: burst exchanges, timestamp recording at PHY layer
    - Round-trip time (RTT) measurement and time-of-flight ranging
    - Channel State Information (CSI) vs. FTM for occupancy sensing
2. **RF signal propagation**:
    - Multipath propagation in indoor environments
    - How human body presence affects RF signals
    - Line-of-sight (LOS) vs. non-line-of-sight (NLOS) scenarios
    - Signal attenuation through different materials (wood, drywall, metal)
3. **Machine learning for time series classification**:
    - Feature extraction from time series data
    - Sliding window techniques
    - Supervised classification tasks:

    - Random Forest and XGBoost ensemble methods
    - K-Nearest Neighbors (KNN)
  - Cross-validation and generalization to new environments

4. **Understanding related sensing technologies**:
   - Bluetooth Low Energy (BLE) RSSI-based methods
   - WiFi RSS (Received Signal Strength) methods
   - Channel State Information (CSI) approaches

5. **Occupancy detection metrics and evaluation**:
   - Accuracy, precision, recall, F1-score
   - Static vs. dynamic occupancy scenarios
   - Adaptation latency as a performance metric

16.3 Methods

The paper employs several technical methods that should be expanded upon in class discussion:

16.3.1 1. Time Series Feature Extraction

- **Variance**: Captures spread in RTT measurements - higher when occupant present due to multipath variations
- **Standard deviation**: Measures RTT variability
- **Root mean square (RMS)**: Quantifies magnitude of RTT fluctuations

16.3.2 2. Machine Learning Classification

- **Random Forest**: Ensemble of decision trees, best performer (97.72% accuracy)
- **XGBoost**: Gradient boosted trees (96.58% accuracy)
- **K-Nearest Neighbors**: Instance-based learning (83.48% accuracy)
- **Logistic Regression**: Linear classifier (74.86% accuracy)

16.3.3 3. Hardware Implementation

- ESP32S2 development boards and their use for WiFi diagnostics

17 PyDCM: Custom Data Center Models with Reinforcement Learning for Sustainability

17.1 Overview

Citation

Primary paper:
Avisek Naug, Antonio Guillen, Ricardo Luna Gutiérrez, Vineet Gundecha, Dejan Markovikj, Lekhapriya Dheeraj Kashyap, Lorenz Krause, Sahand Ghorbanpour, Sajad Mousavi, Ashwin Ramesh Babu, and Soumyendu Sarkar. 2023. PyDCM: Custom Data Center Models with Reinforcement Learning for Sustainability. In *The 10th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation (BuildSys '23)*, November 15–16, 2023, Istanbul, Turkey. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3600100.3623732
Companion paper (extended version):
Avisek Naug, Antonio Guillen, Ricardo Luna, Vineet Gundecha, Cullen Bash, Sahand Ghorbanpour, Sajad Mousavi, Ashwin Ramesh Babu, Dejan Markovikj, Lekhapriya D Kashyap, Desik Rengarajan, and Soumyendu Sarkar. 2024. SustainDC: Benchmarking for Sustainable Data Center Control. In *38th Conference on Neural Information Processing Systems (NeurIPS 2024) Track on Datasets and Benchmarks*.

Sense, Plan, Act Mapping and Objective

Application domain: Sustainable data center energy management and cooling control. Data centers are among the fastest-growing energy consumers globally, driven by the exponential increase in AI/ML workloads. Optimizing their operation—workload scheduling, cooling setpoints, and battery storage—can significantly reduce energy consumption and carbon footprint.

- **Sense**: PyDCM and SustainDC model the sensing of key data center state variables: IT room temperatures (supply and return air), server CPU utilization and power consumption, HVAC component energy usage, external weather conditions, grid carbon intensity (CI), and battery state of charge. These measurements form the observation/state space for the RL agents. The models incorporate precomputed Computational Fluid Dynamics (CFD) results to capture temperature distributions.

- **Plan**: This is the primary innovation area. PyDCM provides a fast, customizable Python-based simulation environment for data center thermal and electrical dynamics, enabling rapid prototyping of RL-based control strategies. SustainDC extends this into a full multi-agent reinforcement learning (MARL) benchmark with three coordinated agents: a workload scheduler ($Agent_{LS}$), a cooling optimizer ($Agent_{DC}$), and a battery manager ($Agent_{BAT}$). The agents learn policies to jointly minimize carbon footprint (CFP) over a planning horizon.
- **Act**: The control actions include: (1) scheduling delay-tolerant workloads to periods of lower grid carbon intensity, (2) adjusting CRAH cooling setpoints to optimize the trade-off between cooling energy and server efficiency, and (3) charging/discharging on-site battery banks to shift grid energy consumption away from high-CI periods.

17.2 Review of the paper

17.2.1 Summary

PyDCM (the primary paper) introduces a modular, Python-native data center simulation model that replaces the traditional reliance on EnergyPlus or Modelica for data center thermal modeling. Key advantages include:

1. **Speed**: PyDCM is 30–40x faster than EnergyPlus for data center simulations due to vectorized thermal calculations and in-place operations, with sub-linear scaling as CPUs increase.
2. **Customizability**: Users can configure individual server specifications, cabinet arrangements (rows, racks, CPUs per rack), HVAC component parameters, and airflow containment strategies via a simple JSON configuration file (`dc_config.json`).
3. **RL integration**: PyDCM wraps the simulation as an OpenAI Gymnasium environment, enabling direct use with standard RL libraries without cross-platform communication overhead.
4. **Reduced resource usage**: Lower memory (16.84 GB vs. 18.20 GB) and CPU utilization (18.21% vs. 20.64%) compared to EnergyPlus.

SustainDC (the companion paper, NeurIPS 2024) builds on PyDCM to create a comprehensive multi-agent RL benchmarking platform for data center operations. It defines three interconnected Gymnasium environments:

- **Workload Environment** (Env_{LS}): Manages scheduling of delay-tolerant workloads using real traces from Alibaba and Google data centers.
- **Data Center Environment** (Env_{DC}): Simulates the thermo-fluid dynamics of the IT room and HVAC cooling system, with automatic chiller "sizing" based on workload and configuration.

- **Battery Environment** (Env_{BAT}): Models on-site battery charging/discharging to shift energy consumption away from high grid carbon intensity periods.

SustainDC benchmarks several MARL algorithms (IPPO, MAPPO, HAPPO, HAA2C, HAD3QN, HASAC) across multiple US locations, demonstrating that multi-agent approaches outperform single-agent baselines on carbon footprint, energy consumption, and water usage metrics.

17.2.2 What do we know already?

This paper connects to several concepts covered in earlier lectures:

From Lecture 2 (Why Buildings):

- The energy and carbon motivation: data centers consume massive amounts of energy (a moderately sized DC can use up to 100x the energy of a similarly sized office space), making them prime targets for energy efficiency improvements
- The Sense/Plan/Act framework—PyDCM/SustainDC provides the simulation environment needed to develop and test the "Plan" component (RL-based controllers)

From Lectures 3–5 (Thermal Dynamics of Buildings):

- The heat transfer and thermodynamic modeling concepts directly apply here: data center thermal models capture conduction, convection, and heat exchange between IT equipment, CRAH units, chillers, and cooling towers
- The concept of thermal zones maps to the IT room zones, cabinet-level temperature distributions, and supply/return air temperatures
- CFD-derived approach temperatures are used to simplify the complex 3D thermal dynamics into a computationally tractable model

From Lecture 6 (Thermal Comfort):

- While data centers don't have human occupants to keep comfortable, the servers have strict thermal operating envelopes. The cooling setpoint optimization problem is analogous to comfort-constrained HVAC control—instead of PMV/PPD bounds, we have maximum allowable server inlet temperatures

From Lecture 7 (AC Power):

- Understanding of electrical power consumption models for IT equipment and HVAC components
- The grid carbon intensity concept—how the carbon content of electricity varies with time, location, and generation mix

From Lectures 8–9 (Control Theory):

- The fundamentals of feedback control apply directly to the CRAH setpoint control problem
- The paper compares RL-based controllers against rule-based baselines (e.g., ASHRAE Guideline 36), connecting to the classical control approaches covered in class
- The multi-agent formulation extends single-loop control to coordinated multi-objective optimization

Things to learn more about

To fully understand these papers' contributions, students may need background on:

1. **Data center architecture and cooling systems**:
    - IT room layout: rows, cabinets, servers, and airflow containment strategies (cold aisle, hot aisle, open)
    - HVAC components specific to data centers: Computer Room Air Handlers (CRAH), chillers, pumps, cooling towers
    - Heat rejection chain: server → CRAH → chiller → cooling tower → external environment
    - Power Usage Effectiveness (PUE) and other data center efficiency metrics
2. **Reinforcement learning fundamentals**:
    - Markov Decision Processes (MDPs): states, actions, rewards, transitions
    - Policy gradient methods, specifically Proximal Policy Optimization (PPO)
    - The OpenAI Gymnasium interface (`reset`, `step`, `init`) and how simulation environments are wrapped for RL
    - Reward shaping and its impact on learned behavior
3. **Multi-agent reinforcement learning (MARL)**:
    - Independent vs. centralized training paradigms
    - Independent PPO (IPPO) vs. Multi-Agent PPO (MAPPO) vs. heterogeneous methods (HAPPO, HAA2C, HAD3QN, HASAC)
    - Collaborative reward structures and the α weighting parameter for reward sharing
    - Challenges of heterogeneous action and observation spaces
4. **Grid carbon intensity and carbon-aware computing**:
    - How grid carbon intensity varies by location, time of day, and season
    - Carbon-aware workload scheduling: shifting computation to low-CI periods
    - Battery storage for energy arbitrage and carbon footprint reduction
5. **Computational Fluid Dynamics (CFD) for data centers**:

- How CFD simulations are used to precompute approach temperatures (difference between CRAH supply temperature and server inlet temperature)
- Simplifying 3D thermal dynamics into reduced-order models suitable for real-time control

17.3 Methods

The papers employ several technical methods that should be expanded upon in class discussion:

17.3.1 1. Data Center Thermal Modeling

- **IT power model**: Computes power consumption for each CPU and fan based on utilization and inlet temperature, with configurable power curves (idle power, rated full load power, rated full load frequency)
- **HVAC model**: Hierarchical model of CRAH, chiller, pump, and cooling tower, each with configurable parameters. Energy consumption is computed based on thermal load and component characteristics
- **CFD-derived approach temperatures**: Precomputed temperature offsets between CRAH supply air and actual server inlet temperatures, capturing the 3D airflow patterns without running CFD at every timestep
- **Automatic chiller sizing**: HVAC cooling capacities are automatically adjusted based on workload demands and IT room configurations

17.3.2 2. Reinforcement Learning Formulation

- **State spaces**: Each agent observes different variables—e.g., the cooling agent sees time of day, dry-bulb temperature, room temperature, previous energy usage, and forecasted grid CI
- **Action spaces**: All three agents use `Discrete(3)`. The specific mappings are:

Agent	Action 0	Action 1	Action 2	Do-Nothing
$Agent_{LS}$	Defer shiftable tasks	Process normally	Process deferred queue	**1**
$Agent_{DC}$	Decrease CRAH setpoint	Maintain setpoint	Increase setpoint	**1**

Agent	Action 0	Action 1	Action 2	Do-Nothing
$Agent_{BAT}$	Charge battery	Discharge battery	Idle (no charge/discharge)	**2**

Note that the do-nothing action is **1** for the workload and cooling agents but **2** for the battery agent—easy to get wrong when writing baselines.

- **Reward design**: Default reward is negative carbon footprint ($CFP_t = (E_{hvac} + E_{it} + E_{bat}) \times CI_t$), with customizable alternatives including energy consumption, operating costs, and water usage
- **Collaborative reward sharing**: Weighted combination where each agent receives α of its own reward plus $(1 - \alpha)/2$ from each of the other two agents

17.3.3 3. Multi-Agent Coordination

- **Heterogeneous MARL**: Different agents have different observation spaces, action spaces, and reward structures, requiring heterogeneous multi-agent methods
- **Sequential environment stepping**: At each timestep, the workload agent acts first (adjusting the compute load), then the DC cooling agent (setting the CRAH setpoint given the adjusted workload), then the battery agent (deciding charge/discharge given total energy consumption)
- **Benchmarked algorithms**: IPPO, MAPPO (centralized critic), HAPPO, HAA2C, HAD3QN, HASAC—spanning on-policy and off-policy, homogeneous and heterogeneous approaches

17.3.4 4. Simulation Performance Optimization

- **Vectorized computations**: NumPy-based vectorized thermal and power calculations instead of EnergyPlus's sequential approach
- **In-place operations**: Minimizing memory allocation overhead during simulation steps
- **Efficient reset**: Fast environment reset for RL training loops (99.99% reduction in reset time vs. EnergyPlus)
- **Sub-linear scaling**: Simulation time scales sub-linearly with the number of CPUs, enabling hyper-scale DC modeling

17.4 Data Center Cooling Primer

Before diving into the codebase, it helps to understand the physical system being modeled.

17.4.1 The Heat Rejection Chain

A data center's cooling system removes heat generated by IT equipment through a series of stages:

```
CPU/GPU → Server Fans → Rack (hot aisle) → CRAH Unit → Chilled Water Loop → Chiller → Coolin
```

Each stage has an associated energy cost:

1. **IT equipment** generates heat proportional to its power consumption. A server drawing 300W converts nearly all of that to heat.
2. **Server fans** push air through the rack, moving heat from the chip to the hot aisle. Fan power increases with temperature (more cooling needed → faster fans).
3. **Computer Room Air Handlers (CRAHs)** draw hot air from the hot aisle, cool it via a chilled water coil, and supply cold air to the cold aisle. The CRAH fan consumes significant power.
4. **Chillers** cool the water circulating through the CRAHs. Chiller efficiency is measured by the **Coefficient of Performance (COP)**—the ratio of cooling provided to electricity consumed. A COP of 5.0 means 5 kW of cooling per 1 kW of electricity.
5. **Cooling towers** reject heat from the chiller condenser loop to the atmosphere via evaporative cooling, consuming water and pump energy.

17.4.2 Key Metrics

Power Usage Effectiveness (PUE) is the standard efficiency metric for data centers:

$$PUE = \frac{E_{total}}{E_{IT}} = \frac{E_{IT} + E_{cooling} + E_{other}}{E_{IT}}$$

A PUE of 1.0 is the theoretical ideal (all energy goes to computation). Typical values range from 1.2 (efficient) to 2.0+ (inefficient). The cooling system is the largest contributor to the gap between PUE and 1.0.

ASHRAE Thermal Guidelines define allowable server inlet temperature ranges. The recommended range is 18–27°C, with an allowable range extending to 15–32°C for short periods. Operating at higher setpoints saves cooling energy but risks thermal throttling or hardware damage.

17.4.3 Approach Temperatures and CFD

In a real data center, the temperature at each server's inlet differs from the CRAH supply temperature due to airflow mixing, recirculation, and containment effectiveness. The **approach temperature** captures this offset:

$$T_{inlet,rack} = T_{CRAH,supply} + \Delta T_{approach,rack}$$

PyDCM uses precomputed CFD results to set these approach temperatures per rack, avoiding the need to run expensive 3D fluid simulations at every timestep while still capturing spatial non-uniformity.

17.5 Reinforcement Learning Foundations

17.5.1 Markov Decision Processes (MDPs)

An RL problem is formalized as an MDP defined by the tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$:

- $\mathcal{S}$: **State space** — the set of all possible observations the agent can see
- $\mathcal{A}$: **Action space** — the set of all actions the agent can take
- $P(s'|s,a)$: **Transition function** — the probability of reaching state s' after taking action a in state s
- $R(s,a)$: **Reward function** — the immediate reward for taking action a in state s
- $\gamma \in [0,1]$: **Discount factor** — how much the agent values future vs. immediate rewards

The agent's goal is to learn a **policy** $\pi(a|s)$ that maximizes the expected cumulative discounted reward:

$$J(\pi) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)\right]$$

17.5.2 Mapping MDPs to Data Center Cooling

In the SustainDC cooling environment (`dc_gym.py`), the MDP components are:

MDP Component	Data Center Cooling
State s_t	Ambient temperature, CRAH setpoint, zone air temperature, HVAC power, IT power (5-dim vector, normalized)

MDP Component	Data Center Cooling
Action a_t	Discrete: {decrease setpoint, maintain setpoint, increase setpoint}
Reward r_t	$-CFP_t = -(E_{hvac} + E_{it}) \times CI_t$ (negative carbon footprint)
Transition	PyDCM thermal model steps forward one timestep
Episode	One year of operation (8,760 hourly steps)

17.5.3 The OpenAI Gymnasium Interface

The individual sub-environments (`dc_gym.py`, `ls_gym.py`, `bat_gym.py`) follow the standard Gymnasium API:

```
import gymnasium as gym

# Create the environment
env = gym.make("dc_gym-v0", config=config)

# Reset to initial state
obs, info = env.reset()

# Run one episode
done = False
while not done:
    action = agent.select_action(obs)  # your policy
    obs, reward, terminated, truncated, info = env.step(action)
    done = terminated or truncated
```

The key methods:

- `reset()` → returns initial observation and info dict. In PyDCM this is ~100x faster than EnergyPlus because there are no IDF files to recompile.
- `step(action)` → advances simulation by one timestep, returns `(observation, reward, terminated, truncated, info)`. PyDCM's vectorized computation makes this ~30x faster than EnergyPlus.
- `observation_space` and `action_space` → define the valid ranges for observations and actions.

⚠ SustainDC Wrapper API Quirks

The `SustainDC` multi-agent wrapper (`sustaindc_env.py`) was designed for the HARL training framework and deviates from the standard Gymnasium API in several ways:

- `reset()` returns **only** the observation dict—not the `(obs, info)` tuple you would expect. The info dict is stored internally on `self.infos`.
- `observation_space` **and** `action_space` are **lists** (one entry per active agent), not dicts keyed by agent name. The ordering follows the `"agents"` config list.
- `step()` returns the standard 5-tuple `(obs, rewards, terminated, truncated, info)`, but all values are dicts keyed by agent name (e.g., `"agent_dc"`).
- **Episode end uses truncation, not termination.** The wrapper sets `truncateds["__all__"] = True` but leaves `terminateds["__all__"] = False`. Your simulation loop **must** check both: `done = terminated.get("__all__", False) or truncated.get("__all__", False)`. If you only check `terminated`, the loop will never exit and will eventually crash when data managers run past the end of their arrays.
- **The** `month` **parameter is required.** It is not in `EnvConfig`'s defaults, but if omitted, `self.month` is `None` and the environment crashes. The value is 0-indexed: `0` = January, `11` = December.

See the SustainDC hands-on tutorial for working code examples.

17.5.4 Policy Gradient Methods and PPO

Policy gradient methods directly optimize the policy $\pi_\theta(a|s)$ (parameterized by θ) by computing gradients of the expected reward with respect to θ:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot A_t\right]$$

where A_t is the **advantage function** — how much better action a_t is compared to the average action in state s_t.

Proximal Policy Optimization (PPO) is the most widely used policy gradient algorithm. Its key idea is to prevent destructively large policy updates by clipping the objective:

$$L^{CLIP}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)A_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t\right)\right]$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ is the probability ratio between the new and old policies, and ϵ (typically 0.2) limits how far the new policy can deviate from the old one in a single update.

PPO is the base algorithm for most of SustainDC's multi-agent methods (IPPO, MAPPO, HAPPO all build on PPO).

17.5.5 Reward Shaping

The choice of reward function fundamentally determines what the agent learns. SustainDC provides several options via `utils/reward_creator.py`:

Reward Function	What It Optimizes
`default_dc_reward`	Negative carbon footprint: $-(E_{hvac} + E_{it}) \times CI_t$
`tou_reward`	Time-of-use electricity cost
`energy_PUE_reward`	Power Usage Effectiveness
`water_usage_efficiency_reward`	Cooling tower water consumption
`temperature_efficiency_reward`	Thermal constraint satisfaction
`custom_agent_reward`	User-defined (template provided)

Reward Design Matters

Using carbon footprint as the reward means the agent will learn to cool *less aggressively* during high-CI periods (when the grid is dirty) and more aggressively during low-CI periods. This is desirable for sustainability but may conflict with thermal safety. In practice, you often need to combine multiple objectives or add constraint penalties.

17.6 Hands-On Tutorials

The concepts covered above — data center thermal modeling, the Gymnasium interface, PPO, and multi-agent coordination — are explored in depth through two companion tutorials:

1. **Getting Started with PyDCM**: Uses the `pydcm` branch (BuildSys 2023). Covers environment setup, the Gymnasium simulation loop, benchmarking PyDCM's speed advantage, and training a single-agent PPO controller for the cooling task.
2. **Getting Started with SustainDC**: Uses the `main` branch (NeurIPS 2024). Covers the full multi-agent framework with three coordinated agents, rule-based baseline controllers, HARL training (HAPPO, MAPPO, etc.), and evaluation using the five SustainDC metrics.

Start with the PyDCM tutorial (simpler, single-agent) before moving to SustainDC (multi-agent with more configuration options).

17.7 Connecting to Gnu-RL (Paper 3 Preview)

Assignment 3 asks you to apply the **Gnu-RL** algorithm (Paper 3) to the SustainDC environment. Gnu-RL uses a fundamentally different policy architecture from the standard neural network policies in MAPPO/HAPPO.

17.7.1 The Differentiable MPC Policy

Instead of a neural network mapping observations to actions, Gnu-RL uses a **Differentiable Model Predictive Control (MPC)** layer as the policy. At each timestep, the policy solves an optimization problem:

$$\min_{u_t,\ldots,u_{t+T-1}} \sum_{k=t}^{t+T-1} \left[\frac{\eta}{2}\|x_k - x_{setpoint}\|^2 + \|u_k\|\right]$$

subject to:

$$x_{k+1} = Ax_k + B_u u_k + B_d d_k, \quad \underline{u} \leq u_k \leq \overline{u}$$

where:

- x_k: state (zone temperatures)
- u_k: control action (cooling setpoint adjustment)
- d_k: disturbances (weather, workload)
- T: planning horizon (e.g., 3 hours ahead)
- η: weight balancing comfort vs. energy
- A, B_u, B_d: **learnable** system dynamics parameters

The key insight is that this optimization problem is **differentiable** with respect to A, B_u, B_d, and η. This means we can backpropagate through the MPC solver to learn the dynamics model end-to-end.

17.7.2 Why This Matters

Compared to standard neural network policies (PPO, SAC, etc.), the Differentiable MPC policy:

Property	Neural Network Policy	Differentiable MPC Policy
Parameters	Thousands–millions of weights	Only A, B_u, B_d (a few dozen)

Property	Neural Network Policy	Differentiable MPC Policy
Sample efficiency	Needs many episodes	Learns from limited data
Interpretability	Black box	Learned dynamics are inspectable
Domain knowledge	None encoded	Planning horizon, constraints, cost structure
Pre-training	Random initialization	Imitation learning from baseline controller

17.7.3 Adapting for Data Center Cooling

To apply Gnu-RL to the SustainDC DC cooling environment, you need to define:

State vector x_t (what temperatures to track):

- Zone air temperature (IT room average)
- CRAH supply air temperature

Control action u_t:

- CRAH setpoint adjustment (maps to the discrete actions in `dc_gym.py`, or can be relaxed to continuous)

Disturbance vector d_t (uncontrollable external inputs):

- Outdoor dry-bulb temperature (from weather data)
- IT workload / CPU utilization (from workload traces)
- Grid carbon intensity (if included in the cost function)

Linear dynamics model:

$$\underbrace{\begin{bmatrix} T_{zone} \\ T_{CRAH} \end{bmatrix}}_{x_{t+1}} = \underbrace{A}_{2\times 2} \underbrace{\begin{bmatrix} T_{zone} \\ T_{CRAH} \end{bmatrix}}_{x_t} + \underbrace{B_u}_{2\times 1} \underbrace{[\Delta T_{setpoint}]}_{u_t} + \underbrace{B_d}_{2\times 3} \underbrace{\begin{bmatrix} T_{outdoor} \\ W_{load} \\ CI \end{bmatrix}}_{d_t}$$

17.7.4 Two-Phase Training

Gnu-RL trains in two phases:

Phase 1: Imitation Learning (offline)

1. Run the baseline controller for 3+ months of simulated time to collect state-action pairs (x_t, u_t, d_t)
2. Initialize A, B_u, B_d randomly
3. Minimize a combined loss:

$$\mathcal{L} = \lambda \sum_t \|x_{t+1} - \hat{x}_{t+1}\|^2 + (1 - \lambda) \sum_t \|u_t - \hat{u}_t\|^2$$

where $\hat{x}_{t+1}$ and $\hat{u}_t$ are the model's predictions, and λ balances state prediction accuracy vs. action matching.

Phase 2: Online Learning (policy gradient refinement)

1. Deploy the pre-trained agent in the SustainDC environment
2. Continue training with PPO, backpropagating through the MPC solver
3. The agent fine-tunes A, B_u, B_d to improve actual performance (not just imitation)

> **!** Key Adaptation Consideration
>
> The original Gnu-RL was designed for building HVAC (slow dynamics, 5–15 min timesteps, 3-hour planning horizon). Data center thermal dynamics are **faster** (1–5 min timesteps) because server rooms have less thermal mass than buildings. You may need to adjust the planning horizon and timestep accordingly. Additionally, data centers have better instrumentation than typical buildings, so the state observations tend to be more reliable.

18 Hands-On: Getting Started with PyDCM

18.1 Overview

Learning Objectives

By the end of this activity, students will be able to:

- Set up and configure a PyDCM data center simulation environment
- Understand the relationship between the JSON configuration and the physical data center being modeled
- Run a simulation loop using the OpenAI Gymnasium interface
- Benchmark PyDCM step times and understand the source of its speed advantage
- Implement and evaluate a baseline controller
- Train a PPO agent and compare its performance against the baseline

Prerequisites

This tutorial assumes you have read the preceding chapter, Paper 2: PyDCM & SustainDC, which covers:

- The PyDCM architecture and its advantages over EnergyPlus
- Data center cooling fundamentals (heat rejection chain, PUE, approach temperatures)
- Reinforcement learning basics (MDPs, PPO, the Gymnasium interface)
- The SustainDC multi-agent framework

Here we put those concepts into practice.

! Run This on Your Own Machine (or Colab)

The code in this notebook requires the `dc-rl` repository and its dependencies. It is **not** executed during the book build. Follow along by running each cell in your own Python environment.
The original authors provide a Colab notebook — this tutorial is a cleaned-up and expanded version of that notebook.

18.2 Environment Setup

18.2.1 Cloning the Repository

The PyDCM code lives on the `pydcm` branch of the `dc-rl` repository. The `main` branch contains the full SustainDC multi-agent framework (NeurIPS 2024); the `pydcm` branch has the original BuildSys 2023 implementation along with the `DCRL` wrapper that we will use here.

```
git clone -b pydcm https://github.com/HewlettPackard/dc-rl.git
cd dc-rl
```

18.2.2 Installing Dependencies

Initialize a `uv` project in the cloned repository and install the dependencies:

```
uv init --python 3.10
```

Before installing packages, edit `pyproject.toml` to restrict Python to 3.10 only. Change the line `requires-python = ">=3.10"` to:

```
requires-python = ">=3.10, <3.11"
```

This is necessary because Ray 2.4.0 has conflicting dependency versions across Python 3.10 vs. 3.11+, and `uv` resolves for all supported versions by default. Now install:

```
uv add numpy==1.23.5 pandas==1.5.3 scipy==1.13.0 "ray[rllib]==2.4.0" torch==2.0.0 "setuptools
```

Ray 2.4.0 will pull in `gymnasium==0.26.3` automatically as a dependency. You also need `setuptools<72` because Ray 2.4.0 uses `pkg_resources`, which was removed from setuptools in version 72.

> macOS Apple Silicon Troubleshooting
>
> If `uv sync` fails with a `grpcio` build error (`ModuleNotFoundError: No module named 'pkg_resources'`), add this to your `pyproject.toml` and re-run:
>
> ```
> [tool.uv]
> override-dependencies = ["grpcio>=1.60.0"]
> ```
>
> This forces `uv` to use a newer `grpcio` that has pre-built Apple Silicon wheels, bypassing the source build of the older version Ray 2.4.0 would otherwise pull in.

Why Is Ray Required?

The `DCRL` wrapper (`dcrl_env.py`) inherits from Ray's `MultiAgentEnv` at the module level, so `from dcrl_env import DCRL` will fail with a `ModuleNotFoundError` if Ray is not installed. Ray and PyTorch are heavyweight packages (~2 GB together), but they are **not optional** for this codebase.

Colab Users

If you are running in Google Colab (which does not have `uv`), you can install packages directly with pip:

```
pip install numpy==1.23.5 pandas==1.5.3 scipy==1.13.0 "ray[rllib]==2.4.0" torch==2.0.0 xlsx
```

18.2.3 Verifying the Setup

A quick sanity check — import the key modules and confirm the environment registers:

```
from dcrl_env import DCRL
import numpy as np

env = DCRL({"agents": ["agent_dc"]})
obs, info = env.reset()

print("Observation space:", env.observation_space["agent_dc"])
print("Action space:     ", env.action_space["agent_dc"])
```

You should see output like:

```
Observation space: Box(-2000000000.0, 5000000000.0, (12,), float32)
Action space:      Discrete(9)
```

The observation is a 12-dimensional vector (normalized state variables), and the agent chooses from 9 discrete actions (combinations of setpoint adjustments).

18.3 Understanding the Configuration

Before running any simulation, it is worth understanding what physical system you are modeling. The data center layout and equipment parameters are defined in `utils/dc_config.json`.

18.3.1 Key Configuration Parameters


```
{
  "data_center_configuration": {
    "NUM_ROWS": 2,
    "NUM_RACKS_PER_ROW": 1,
    "CPUS_PER_RACK": 150,
    "RACK_SUPPLY_APPROACH_TEMP_LIST": [5.3, 5.3],
    "RACK_RETURN_APPROACH_TEMP_LIST": [-3.7, -3.7]
  },
  "hvac_configuration": {
    "C_AIR": 1006,
    "RHO_AIR": 1.225,
    "CHILLER_COP": 6.0,
    "CRAC_SUPPLY_AIR_FLOW_RATE_pu": 0.00005663,
    "CT_FAN_REF_P": 1000
  },
  "server_characteristics": {
    "CPU_POWER_RATIO_LB": [0.22, 1.00],
    "CPU_POWER_RATIO_UB": [0.24, 1.02],
    "IT_FAN_AIRFLOW_RATIO_LB": [0.0, 0.6],
    "IT_FAN_AIRFLOW_RATIO_UB": [0.7, 1.3],
    "INLET_TEMP_RANGE": [18, 27],
    "DEFAULT_SERVER_POWER_CHARACTERISTICS": [[170, 110], [120, 60]],
    "HP_PROLIANT": [110, 170]
  }
}
```


18.3.2 What Each Section Controls

Section	Parameter	Physical Meaning
`data_center_configuration`	`NUM_ROWS`, `NUM_RACKS_PER_ROW`	IT room geometry: how many rack rows and racks per row
	`CPUS_PER_RACK`	Number of servers in each rack
	`RACK_SUPPLY_APPROACH_TEMP_LIST`	CFD-derived offsets (ΔT) between CRAH supply air and actual rack inlet (one per rack)

Section	Parameter	Physical Meaning
`hvac_configuration`	`C_AIR`, `RHO_AIR`	Thermophysical properties of air ($c_p = 1006$ J/kg·K, $\rho = 1.225$ kg/m^3)
	`CHILLER_COP`	Chiller coefficient of performance (6.0 = 6 kW cooling per 1 kW electricity)
	`CRAC_SUPPLY_AIR_FLOW_RATE_pu`	Volumetric airflow through the CRAH unit
`server_characteristics`	`CPU_POWER_RATIO_LB/UB`	Linear power curve coefficients: how CPU power scales with temperature and load
	`IT_FAN_AIRFLOW_RATIO_LB/UB`	Fan speed curve: how server fan airflow scales with temperature
	`INLET_TEMP_RANGE`	ASHRAE-compliant operating range for server inlet temperatures (18–27°C)
	`DEFAULT_SERVER_POWER_CHARACTERISTICS`	`[max_power, idle_power]` pairs for server types (Watts)

> **The Vectorization Connection**
>
> Recall from our discussion: each rack's `CPUS_PER_RACK` servers have their power curve parameters (`CPU_POWER_RATIO_LB/UB`) collected into NumPy arrays of shape `(num_CPUs,)`. The simulation then computes power for all servers in a rack simultaneously via NumPy broadcasting, rather than looping over individual server objects. The `RACK_SUPPLY_APPROACH_TEMP_LIST` entries determine each rack's inlet temperature offset from the CRAH supply.

18.4 Running a Simulation

18.4.1 The Gymnasium Loop

The `DCRL` wrapper exposes the standard Gymnasium API. Here is the minimal simulation loop:

```python
from dcrl_env import DCRL
import numpy as np

# Configure environment: only the cooling agent is active
env_config = {
    "agents": ["agent_dc"],
    "location": "ny",
    "cintensity_file": "NYIS_NG_&_avgCI.csv",
    "weather_file": "USA_NY_New.York-Kennedy.epw",
    "workload_file": "Alibaba_CPU_Data_Hourly_1.csv",
    "max_bat_cap_Mw": 2,
    "individual_reward_weight": 0.8,
    "flexible_load": 0.1,
    "ls_reward": "default_ls_reward",
    "dc_reward": "default_dc_reward",
    "bat_reward": "default_bat_reward",
}

env = DCRL(env_config)
obs, info = env.reset()

# Run one full episode (default: 30 days at 15-min timesteps  2,977 steps)
done = False
total_reward = 0.0
step_count = 0

while not done:
    # Fixed action: "maintain current setpoint"
    action = {"agent_dc": 4}
    obs, reward, terminated, truncated, info = env.step(action)
    total_reward += reward["agent_dc"]
    step_count += 1
    done = terminated.get("__all__", False)

print(f"Episode finished after {step_count} steps")
print(f"Total reward: {total_reward:.2f}")
```

18.4.2 Understanding the Environment Config

A few things to note about `env_config`:

- **agents**: Setting ["agent_dc"] means only the cooling agent is learning; the workload and battery agents use default baselines.
- **location**: Determines which weather and carbon intensity data to load. Options include "ny", "az", "wa".
- **individual_reward_weight**: The α parameter for collaborative reward sharing. At 0.8, the cooling agent receives 80% of its own reward and 10% from each of the other two agents.
- **flexible_load**: Fraction of workload that is delay-tolerant (10% here).

18.5 Benchmarking PyDCM

The paper claims PyDCM is 30–40× faster than EnergyPlus. Let's measure the three key RL methods: **init**, **reset**, and **step**.

```
import time
from statistics import mean, stdev
from dcrl_env import DCRL

env_config = {
    "agents": ["agent_dc"],
    "location": "ny",
    "cintensity_file": "NYIS_NG_&_avgCI.csv",
    "weather_file": "USA_NY_New.York-Kennedy.epw",
    "workload_file": "Alibaba_CPU_Data_Hourly_1.csv",
    "max_bat_cap_Mw": 2,
    "individual_reward_weight": 0.8,
    "flexible_load": 0.1,
    "ls_reward": "default_ls_reward",
    "dc_reward": "default_dc_reward",
    "bat_reward": "default_bat_reward",
}

N_RUNS = 10
N_STEPS = 1000

init_times = []
reset_times = []
step_times = []

for run in range(N_RUNS):
    # Time: environment creation
```

```
    t0 = time.perf_counter()
    env = DCRL(env_config)
    init_times.append(time.perf_counter() - t0)

    # Time: reset
    t0 = time.perf_counter()
    obs, info = env.reset()
    reset_times.append(time.perf_counter() - t0)

    # Time: step (average over N_STEPS)
    action = env.dc_env.action_space.sample()
    t0 = time.perf_counter()
    valid_steps = 0
    while valid_steps < N_STEPS:
        obs, rew, terminated, truncated, info = env.step({"agent_dc": action})
        valid_steps += 1
        if terminated.get("__all__", False) or truncated.get("__all__", False):
            env.reset()
    step_times.append((time.perf_counter() - t0) / N_STEPS)

print(f"init  : {mean(init_times):.4f} ± {stdev(init_times):.4f} s")
print(f"reset : {mean(reset_times):.6f} ± {stdev(reset_times):.6f} s")
print(f"step  : {mean(step_times):.6f} ± {stdev(step_times):.6f} s")
```

18.5.1 Expected Results

For comparison, here are the numbers from the paper (Table 2) alongside what the Colab notebook produces:

Method	EnergyPlus	PyDCM (paper)	PyDCM (Colab)
`init`	1.05 s ± 23.6 ms	1.57 ms ± 60.4 μs	~0.20 s
`reset`	2.67 s ± 23.8 ms	0.03 ms ± 0.25 μs	~0.013 s
`step`	0.46 ms ± 98 μs	0.13 ms ± 15.8 μs	~1.18 ms

> ⚠ **Why Are the Colab Numbers Slower?**
>
> The paper's benchmarks were run on a 48-core Intel Xeon 6248 server. Colab provides a shared VM with fewer resources, so the absolute times are higher. The key takeaway is the **relative speedup** over EnergyPlus, not the absolute times. On the authors' server, `step` was ~8,300 iterations/s; on Colab, ~850 iterations/s.

18.5.2 Episode-Level Timing

You can also measure cumulative simulation time for realistic episode lengths:

```
def run_episode_timing(num_steps, n_runs=10):
    """Time a full episode of num_steps steps, averaged over n_runs."""
    episode_times = []
    for _ in range(n_runs):
        env = DCRL(env_config)
        env.reset()
        action = env.dc_env.action_space.sample()

        t0 = time.perf_counter()
        for _ in range(num_steps):
            obs, rew, terminated, truncated, info = env.step({"agent_dc": action})
            if terminated.get("__all__", False) or truncated.get("__all__", False):
                break
        episode_times.append(time.perf_counter() - t0)
    return mean(episode_times), stdev(episode_times)

# 7 days at 15-min timesteps = 7 * 24 * 4 = 672 steps
mean_7d, std_7d = run_episode_timing(7 * 24 * 4)
# 30 days at 15-min timesteps = 30 * 24 * 4 = 2880 steps
mean_30d, std_30d = run_episode_timing(30 * 24 * 4)

print(f"7-day episode  : {mean_7d:.3f} ± {std_7d:.3f} s")
print(f"30-day episode : {mean_30d:.3f} ± {std_30d:.3f} s")
```

18.6 Evaluating a Baseline Controller

Before training an RL agent, you need a baseline to compare against. Here we run a simple **fixed-action controller** that always takes the same action (maintain current setpoint):

```
import numpy as np
from dcrl_env import DCRL

env_config = {
    "agents": ["agent_dc"],
    "location": "ny",
    "cintensity_file": "NYIS_NG_&_avgCI.csv",
    "weather_file": "USA_NY_New.York-Kennedy.epw",
```

```
    "workload_file": "Alibaba_CPU_Data_Hourly_1.csv",
    "max_bat_cap_Mw": 2,
    "individual_reward_weight": 0.8,
    "flexible_load": 0.1,
    "ls_reward": "default_ls_reward",
    "dc_reward": "default_dc_reward",
    "bat_reward": "default_bat_reward",
    "evaluation": True,  # enables evaluation mode (used by the load-shifting agent)
}

env = DCRL(env_config)
obs, info = env.reset()

done = False
energy_trace = []
carbon_trace = []
baseline_action = 4  # "maintain current setpoint"

while not done:
    obs, rew, terminated, truncated, info = env.step({"agent_dc": baseline_action})
    dc_info = info["agent_dc"]
    energy_trace.append(dc_info["bat_total_energy_with_battery_KWh"])
    carbon_trace.append(dc_info["bat_CO2_footprint"])
    done = terminated["__all__"]

print(f"Baseline final energy: {energy_trace[-1]:.2f} kWh")
print(f"Baseline final carbon: {carbon_trace[-1]:.2f} gCO2")
```

What the `info` Dict Contains

The `info["agent_dc"]` dictionary returned at each step contains detailed metrics including total energy consumption (IT + HVAC + battery), carbon footprint, and individual component breakdowns. This is how you track KPIs without modifying the environment code.

18.7 Training a PPO Agent

The repository includes a training script `train_ppo.py` that uses Ray RLlib to train a PPO agent on the cooling control task.

18.7.1 Running Training

From the repository root:

```
uv run python train_ppo.py
```

This will train for the default number of iterations and save checkpoints to the `pydcm/` directory. You can monitor training progress with TensorBoard:

```
uv run tensorboard --logdir pydcm/
```

18.7.2 What Is the Agent Learning?

During training, the PPO agent interacts with the PyDCM simulation thousands of times per iteration. At each timestep it:

1. **Observes** the current state (ambient temperature, CRAH setpoint, zone temperature, HVAC power, IT power, etc.)
2. **Selects an action** (decrease, maintain, or increase the CRAH supply temperature setpoint)
3. **Receives a reward** equal to the negative carbon footprint: $r_t = -(E_{hvac,t} + E_{it,t}) \times CI_t$

The agent learns to adjust the cooling setpoint dynamically — cooling less aggressively when the grid is clean (low CI_t) or the workload is light, and more aggressively when temperatures approach unsafe limits.

18.8 Comparing Trained Agent vs. Baseline

After training completes, evaluate the trained policy against the fixed-action baseline:

```
import numpy as np
from pathlib import Path
from ray.rllib.algorithms.algorithm import Algorithm
from dcrl_env import DCRL

# --- Load the trained agent ---
trial_dir = next(Path("pydcm/pydcm_hvac_ppo").glob("PPO_*"))
checkpoint_dir = sorted(trial_dir.glob("checkpoint_*"))[-1]
algo = Algorithm.from_checkpoint(str(checkpoint_dir))
```

```
# --- Environment config (evaluation mode) ---
env_config = {
    "agents": ["agent_dc"],
    "location": "ny",
    "cintensity_file": "NYIS_NG_&_avgCI.csv",
    "weather_file": "USA_NY_New.York-Kennedy.epw",
    "workload_file": "Alibaba_CPU_Data_Hourly_1.csv",
    "max_bat_cap_Mw": 2,
    "individual_reward_weight": 0.8,
    "flexible_load": 0.1,
    "ls_reward": "default_ls_reward",
    "dc_reward": "default_dc_reward",
    "bat_reward": "default_bat_reward",
    "evaluation": True,
}

def rollout(policy_fn):
    """Run one full episode using the given policy function."""
    env = DCRL(env_config)
    obs, info = env.reset()
    done = False
    rewards = []
    final_energy, final_carbon = None, None

    while not done:
        action = policy_fn(obs, env)
        obs, rew, terminated, truncated, info = env.step({"agent_dc": action})
        rewards.append(rew["agent_dc"])
        final_energy = info["agent_dc"]["bat_total_energy_with_battery_KWh"]
        final_carbon = info["agent_dc"]["bat_CO2_footprint"]
        done = terminated["__all__"]

    return {
        "total_reward": float(np.sum(rewards)),
        "final_energy_kwh": float(final_energy),
        "final_carbon": float(final_carbon),
    }

# --- Define policies ---
def trained_policy(obs, env):
    return algo.compute_single_action(
        obs["agent_dc"], policy_id="agent_dc", explore=False
```

```
    )

def baseline_policy(obs, env):
    return 4  # fixed: maintain setpoint

# --- Run comparison ---
trained_results = rollout(trained_policy)
baseline_results = rollout(baseline_policy)

print(f"{'Metric':<25} {'Baseline':>12} {'Trained':>12} {'Reduction':>10}")
print("-" * 62)
for key, label in [("final_energy_kwh", "Energy (kWh)"),
                   ("final_carbon", "Carbon (gCO2)")]:
    b = baseline_results[key]
    t = trained_results[key]
    pct = 100 * (b - t) / b
    print(f"{label:<25} {b:>12.1f} {t:>12.1f} {pct:>9.1f}%")
```

18.8.1 Expected Results

The Colab notebook reports approximately:

Metric	Baseline	Trained	Reduction
Energy (kWh)	554.4	443.2	~20%
Carbon (gCO_2)	315,128	252,067	~20%

A ~20% reduction in both energy and carbon footprint from a relatively short PPO training run is a strong result, demonstrating that even simple RL agents can significantly outperform fixed-setpoint controllers.

> **Why Does This Work?**
>
> The fixed-action baseline maintains a constant CRAH setpoint regardless of conditions. The RL agent learns to **modulate the setpoint** based on current weather, workload, and grid carbon intensity. For example, it may allow the data center to run slightly warmer during periods of high carbon intensity (reducing HVAC energy when the grid is dirty) and cool more aggressively when the grid is clean.

18.9 Customizing the Data Center

One of PyDCM's main advantages is configurability. You can modify `utils/dc_config.json` to model different data center designs.

18.9.1 Example: Scaling Up

To model a larger data center (e.g., 4 rows × 5 racks × 200 CPUs per rack = 4,000 servers):

```
{
  "data_center_configuration": {
    "NUM_ROWS": 4,
    "NUM_RACKS_PER_ROW": 5,
    "CPUS_PER_RACK": 200,
    "RACK_SUPPLY_APPROACH_TEMP_LIST": [
      5.3, 5.3, 5.5, 5.5, 5.7,
      5.3, 5.3, 5.5, 5.5, 5.7,
      5.3, 5.3, 5.5, 5.5, 5.7,
      5.3, 5.3, 5.5, 5.5, 5.7
    ]
  }
}
```

Note that `RACK_SUPPLY_APPROACH_TEMP_LIST` must have exactly `NUM_ROWS * NUM_RACKS_PER_ROW` entries — one approach temperature offset per rack, derived from CFD analysis.

18.9.2 Example: Different Server Types

To model heterogeneous servers with different power characteristics, modify the server power curves:

```
{
  "server_characteristics": {
    "DEFAULT_SERVER_POWER_CHARACTERISTICS": [
      [300, 150],
      [170, 110],
      [120, 60]
    ]
  }
}
```

Each `[full_load_power, idle_power]` pair (in Watts) defines a server type. PyDCM will distribute these across the racks.

> Experiment Ideas
>
> 1. **Location comparison**: Change `location` to `"az"` (hot climate, Arizona) vs. `"wa"` (mild climate, Washington) and compare baseline energy consumption. How does climate affect cooling costs?
> 2. **Chiller efficiency**: Try `CHILLER_COP` values of 4.0, 6.0, and 8.0. How sensitive is the total energy consumption to chiller efficiency?
> 3. **Data center scale**: Compare simulation speed for 300, 3,000, and 30,000 CPUs. Does it match the sub-linear scaling claim from Figure 3 in the paper?

18.10 Next Steps

This tutorial covers the `pydcm` branch — the original BuildSys 2023 implementation. For Assignment 3, you will use the **`main` branch** which contains the full SustainDC multi-agent framework and apply the Gnu-RL algorithm (Paper 3) to this environment. The concepts and API patterns are the same; the main difference is the multi-agent coordination layer described in the paper notes.

19 Hands-On: Getting Started with SustainDC

19.1 Overview

Learning Objectives

By the end of this activity, students will be able to:

- Set up the full SustainDC multi-agent environment from the `main` branch
- Understand how the three sub-environments (Workload, Data Center, Battery) interact sequentially
- Run simulations with different agent configurations (single-agent, multi-agent)
- Use the built-in baseline and rule-based controllers
- Train multi-agent RL policies using the HARL framework (HAPPO, MAPPO, etc.)
- Evaluate trained agents against baselines using the five SustainDC metrics
- Customize reward functions and experiment with collaborative reward sharing (α)

Prerequisites

This tutorial assumes familiarity with:

- The preceding PyDCM hands-on tutorial, which covers the single-agent cooling control problem
- The Paper 2: PyDCM & SustainDC chapter, which explains the multi-agent formulation, reward structure, and MARL algorithms

SustainDC extends PyDCM from a single cooling agent to **three coordinated agents**. If you are comfortable with the PyDCM Gymnasium loop, the jump to SustainDC is mostly about understanding the multi-agent wrapper and the information flow between environments.

! Run This on Your Own Machine

The code in this notebook requires the `dc-rl` repository (`main` branch) and its dependencies. It is **not** executed during the book build. Follow along by running each cell in your own Python environment.

19.2 Environment Setup

19.2.1 Cloning the Repository

SustainDC lives on the **`main` branch** of the same repository as PyDCM:

```
git clone https://github.com/HewlettPackard/dc-rl.git sustaindc
cd sustaindc
```

> **Branch Differences**
>
> The `pydcm` branch (BuildSys 2023) has a simpler single-agent wrapper. The `main` branch (NeurIPS 2024) adds the multi-agent orchestration layer (`sustaindc_env.py`), the HARL training framework, rule-based baseline agents, and expanded data for 8 US locations.

19.2.2 Installing Dependencies

Initialize a `uv` project in the cloned repository and install the core simulation dependencies:

```
uv init --python 3.10
uv add numpy==1.23.5 pandas==1.5.3 gymnasium==0.29.1 scipy==1.13.0 PyYAML==6.0.1 PsychroLib==
```

This gives you everything needed to run simulations. For training with the HARL framework, also add:

```
uv add torch==2.0.0 tensorboard==2.12.3 tensorboardX==2.6.2.2 setproctitle "supersuit==3.7.0
```

> **Why Not `uv add -r requirements.txt`?**
>
> The full `requirements.txt` pulls in PyTorch, TensorFlow, Ray, JAX, and many other heavy libraries — some of which may conflict or fail to install on your platform. Installing the core dependencies explicitly gives you a working simulation environment without the headaches. Add training dependencies only when you need them.

19.3 The SustainDC Architecture

Before writing any code, it helps to understand the key architectural difference from PyDCM: SustainDC orchestrates **three interconnected Gymnasium environments** through a single wrapper class.

19.3.1 Sequential Execution Within Each Timestep

At every timestep, the three agents act **sequentially**, with information flowing downstream:

```
1. Agent_LS  (Workload Scheduler)
   Observes: time of day, carbon intensity forecast, pending workload
   Acts: store delayable tasks / compute all / maximize throughput
        ↓  adjusted workload (B_t)
2. Agent_DC  (Cooling Optimizer)
   Observes: ambient temp, room temp, HVAC power, IT power, CI forecast
   Acts: decrease / maintain / increase CRAH setpoint
        ↓  total energy consumption (E_hvac + E_it)
3. Agent_BAT (Battery Manager)
   Observes: battery SoC, DC energy consumption, CI forecast
   Acts: charge / hold / discharge
```

This ordering matters: the workload agent's decision changes how much heat the servers generate, which affects the cooling agent's problem, which in turn determines the total energy that the battery agent must manage.

19.3.2 The `SustainDC` Wrapper

The `sustaindc_env.py` class handles all of this coordination. It:

1. Creates the three sub-environments via factory functions
2. Manages four data managers (`Time_Manager`, `Workload_Manager`, `Weather_Manager`, `CI_Manager`)
3. Routes information between sub-environments at each step
4. Constructs per-agent observation vectors (including CI trend features like slopes and percentiles)
5. Computes collaborative rewards based on the α weighting parameter

19.4 Verifying the Setup

Let's confirm everything works by creating the environment and inspecting its structure:

```python
from sustaindc_env import SustainDC, EnvConfig

# Default config: all three agents active, New York location
env_config = EnvConfig({
    "agents": ["agent_ls", "agent_dc", "agent_bat"],
    "month": 2,  # March (0-indexed: 0=Jan, 11=Dec)
    "location": "NY",
    "cintensity_file": "NYIS_NG_&_avgCI.csv",
    "weather_file": "USA_NY_New.York-Kennedy.Intl.AP.744860_TMY3.epw",
    "workload_file": "Alibaba_CPU_Data_Hourly_1.csv",
    "datacenter_capacity_mw": 1.0,
    "battery_capacity_mwh": 0.5,
    "flexible_load": 0.1,
    "individual_reward_weight": 0.8,
    "ls_reward": "default_ls_reward",
    "dc_reward": "default_dc_reward",
    "bat_reward": "default_bat_reward",
})

env = SustainDC(env_config)
obs = env.reset()  # returns only the states dict (no info tuple)

print("=== Observation Spaces ===")
for i, space in enumerate(env.observation_space):
    print(f"  agent {i}: {space.shape}")

print("\n=== Action Spaces ===")
for i, space in enumerate(env.action_space):
    print(f"  agent {i}: {space}")

print("\n=== Initial Observations (first 5 values) ===")
for agent_id, ob in obs.items():
    print(f"  {agent_id}: {ob[:5]}...")
```

Ignore the "Warning" Messages

You will see messages like `"Warning: Please check if the do nothing action for Battery is the '2' action"` during init, and `"Warning, using base agent for agent_bat: 2"` on every step. These are harmless `print()` statements from the SustainDC code confirming that baseline agents are active for agents not in your `"agents"` list. They cannot be suppressed without editing the source.

The `month` Parameter is Required

The `month` key is **not** in `EnvConfig`'s defaults, but it must be provided — otherwise `self.month` is `None` and the environment crashes when creating sub-environments. The value is 0-indexed: `0` = January, `11` = December. During training, the HARL framework sets `month` automatically via the worker index (cycling through all 12 months across parallel workers). For manual experimentation, you must set it yourself.

You should see three agents, each with their own observation and action spaces. The observation dimensions differ because each agent sees different state variables.

API Quirks

`SustainDC` was designed for the HARL multi-agent training framework, not as a standard Gymnasium environment. Two things to watch for:

- **`reset()`** returns only the observation dict — not the `(obs, info)` tuple you would expect from Gymnasium. The info dict is stored internally on `self.infos`.
- **`observation_space` and `action_space`** are **lists** (one entry per active agent in order), not dicts keyed by agent name. The ordering follows the order agents appear in the `"agents"` config list.
- **`step()`** does return the standard 5-tuple `(obs, rewards, terminated, truncated, info)`, all as dicts keyed by agent name.
- **Episode end uses truncation, not termination.** `_handle_terminal()` sets `truncateds["__all__"] = True` but leaves `terminateds["__all__"] = False`. Your loop must check **both**: `done = terminated.get("__all__", False) or truncated.get("__all__", False)`. If you only check `terminated`, the loop will never exit and the simulation will eventually crash when data managers run past the end of their arrays.

19.4.1 Understanding the Action Spaces

All three agents use `Discrete(3)` action spaces, but the actions mean different things:

Agent	Action 0	Action 1	Action 2	Do-Nothing
$Agent_{LS}$	Defer shiftable tasks	Do nothing (process normally)	Process deferred task queue	**1**
$Agent_{DC}$	Decrease CRAH setpoint	Maintain setpoint	Increase setpoint	**1**

Agent	Action 0	Action 1	Action 2	Do-Nothing
$Agent_{BAT}$	Charge battery	Discharge battery	Idle (no charge/discharge)	**2**

> **⚠ Do-Nothing Actions Are Not Uniform**
>
> Note that the "do nothing" action is **1** for the workload and cooling agents but **2** for the battery agent. This is easy to get wrong when writing baseline policies — always use the values from `utils/base_agents.py` as the reference.

19.5 Running a Multi-Agent Simulation

19.5.1 Full Three-Agent Loop

Here is a complete simulation with all three agents taking fixed baseline actions:

```
from sustaindc_env import SustainDC, EnvConfig
import numpy as np

env_config = EnvConfig({
    "agents": ["agent_ls", "agent_dc", "agent_bat"],
    "month": 2,
    "location": "NY",
    "cintensity_file": "NYIS_NG_&_avgCI.csv",
    "weather_file": "USA_NY_New.York-Kennedy.Intl.AP.744860_TMY3.epw",
    "workload_file": "Alibaba_CPU_Data_Hourly_1.csv",
    "datacenter_capacity_mw": 1.0,
    "battery_capacity_mwh": 0.5,
    "flexible_load": 0.1,
    "individual_reward_weight": 0.8,
    "ls_reward": "default_ls_reward",
    "dc_reward": "default_dc_reward",
    "bat_reward": "default_bat_reward",
})

env = SustainDC(env_config)
obs = env.reset()
```

```
# Fixed baseline actions (do-nothing for each agent)
actions = {
    "agent_ls": 1,   # process normally (no load shifting)
    "agent_dc": 1,   # maintain current setpoint
    "agent_bat": 2,   # idle (no charge/discharge) - note: NOT 1
}

done = False
step_count = 0
total_rewards = {"agent_ls": 0.0, "agent_dc": 0.0, "agent_bat": 0.0}

# Track key metrics over time
carbon_trace = []
hvac_energy_trace = []
it_energy_trace = []

while not done:
    obs, rewards, terminated, truncated, info = env.step(actions)

    for agent_id in total_rewards:
        total_rewards[agent_id] += rewards[agent_id]

    # Extract metrics from the DC agent's info dict
    dc_info = info["agent_dc"]
    carbon_trace.append(dc_info.get("bat_CO2_footprint", 0))
    hvac_energy_trace.append(dc_info.get("dc_HVAC_total_power_kW", 0))
    it_energy_trace.append(dc_info.get("dc_ITE_total_power_kW", 0))

    step_count += 1
    done = terminated.get("__all__", False) or truncated.get("__all__", False)

print(f"Episode finished after {step_count} steps")
print(f"\nTotal rewards:")
for agent_id, r in total_rewards.items():
    print(f"  {agent_id}: {r:.2f}")
print(f"\nFinal carbon footprint: {carbon_trace[-1]:.2f} gCO2")
print(f"Avg HVAC power: {np.mean(hvac_energy_trace):.2f} kW")
print(f"Avg IT power:   {np.mean(it_energy_trace):.2f} kW")
```

19.5.2 Single-Agent Mode (Cooling Only)

For simpler experiments, you can activate only the cooling agent. The other two agents will automatically use built-in baseline controllers:

```
env_config_single = EnvConfig({
    "agents": ["agent_dc"],  # only cooling agent is learning
    "month": 2,
    "location": "NY",
    "cintensity_file": "NYIS_NG_&_avgCI.csv",
    "weather_file": "USA_NY_New.York-Kennedy.Intl.AP.744860_TMY3.epw",
    "workload_file": "Alibaba_CPU_Data_Hourly_1.csv",
    "datacenter_capacity_mw": 1.0,
    "battery_capacity_mwh": 0.5,
    "flexible_load": 0.1,
})

env_single = SustainDC(env_config_single)
obs = env_single.reset()

# Now only agent_dc needs an action
done = False
while not done:
    obs, rew, terminated, truncated, info = env_single.step({"agent_dc": 1})
    done = terminated.get("__all__", False) or truncated.get("__all__", False)
```

> **Baseline Agents**
>
> When an agent is not in the `"agents"` list, SustainDC uses a do-nothing baseline from `utils/base_agents.py`. Specifically: the workload agent takes action 1 (process normally, no load shifting), the cooling agent takes action 1 (maintain setpoint), and the battery agent takes action 2 (idle). You will see warnings like `"Warning, using base agent for agent_bat: 2"` — these are informational and confirm the base agent is active.

19.6 Using Rule-Based Controllers

Before jumping to RL, it is useful to understand the rule-based battery controller in `utils/rbc_agents.py`. The repository currently ships one RBC agent — for the battery — while the cooling and workload baselines are the do-nothing agents from `utils/base_agents.py`.

19.6.1 Carbon-Aware Battery Controller

`RBCBatteryAgent` implements a simple heuristic: it compares the current carbon intensity against a smoothed look-ahead forecast. If the forecast CI is higher than current (grid is getting dirtier), it **charges** now while the grid is cleaner; otherwise it **discharges** stored energy:

```
from utils.rbc_agents import RBCBatteryAgent

# look_ahead: how many future CI steps to consider
# smooth_window: moving average window for the forecast
rbc_battery = RBCBatteryAgent(look_ahead=3, smooth_window=1)

# In your simulation loop, pass the CI forecast and current battery SoC:
# action = rbc_battery.act(ci_forecast_values, current_soc)
# Returns: 0 (charge), 1 (discharge), or 2 (idle)
```

> **No RBC Cooling Agent (Yet)**
>
> The paper's benchmarks (Section 6) reference an ASHRAE Guideline 36 cooling baseline, but this is implemented separately in `ashrae36_evalpydcm.py`, not as a reusable agent class. For the cooling agent, the do-nothing baseline (action 1 = maintain setpoint) is what `base_agents.py` provides. Writing a proportional or rule-based cooling controller is a good exercise — see the examples in the PyDCM tutorial.

19.7 Training with the HARL Framework

SustainDC includes the **Heterogeneous Agent RL (HARL)** framework, which supports a wide range of multi-agent algorithms. Training is done through `train_sustaindc.py`.

19.7.1 Quick Start: HAPPO

HAPPO (Heterogeneous Agent PPO) is the recommended starting algorithm. It handles agents with different observation and action spaces natively:

```
uv run python train_sustaindc.py --algo happo --exp_name happo_ny
```

19.7.2 Other Supported Algorithms

Algorithm	Type	Key Idea
IPPO	On-policy, independent	Each agent trains its own PPO; no shared information
MAPPO	On-policy, centralized critic	Shared critic sees all agents' observations
HAPPO	On-policy, heterogeneous	Like MAPPO but handles different obs/action spaces
HAA2C	On-policy, heterogeneous	Advantage Actor-Critic variant of HAPPO
HAD3QN	Off-policy, heterogeneous	Dueling Double DQN for discrete actions
HASAC	Off-policy, heterogeneous	Soft Actor-Critic with entropy regularization
MADDPG	Off-policy, centralized critic	Multi-Agent DDPG (continuous actions)

To try a different algorithm:

```
uv run python train_sustaindc.py --algo mappo --exp_name mappo_ny
uv run python train_sustaindc.py --algo ippo --exp_name ippo_ny
```

19.7.3 Configuration Files

The HARL framework loads configuration from YAML files in `harl/configs/`:

- **Algorithm config**: `harl/configs/algos_cfgs/<algo>.yaml` — learning rate, batch size, number of epochs, etc.
- **Environment config**: `harl/configs/envs_cfgs/sustaindc.yaml` — location, data files, reward settings, agent list

You can override any config parameter from the command line:

```
# Train in Arizona with a different reward weight
uv run python train_sustaindc.py --algo happo --exp_name happo_az \
    --location AZ \
    --cintensity_file "AZPS_NG_&_avgCI.csv" \
    --weather_file USA_AZ_Davis-Monthan.AFB.722745_TMY3.epw \
    --individual_reward_weight 0.5
```

19.7.4 Monitoring Training

Training logs are written to TensorBoard format:

```
uv run tensorboard --logdir results/sustaindc/
```

Key metrics to monitor:

- **Episode reward** (per agent): should increase over training
- **Carbon footprint**: should decrease
- **Task queue length**: for the LS agent, should stay low (tasks being completed on time)

19.8 Evaluation

19.8.1 The Five SustainDC Metrics

The paper defines five evaluation metrics (Section 5):

Metric	Description	Lower is Better?
CO_2 Footprint (CFP)	Cumulative carbon emissions: $\sum_t (E_{hvac} + E_{it} + E_{bat}) \times CI_t$	Yes
HVAC Energy	Total cooling energy (chiller + pumps + cooling tower + CRAH fan)	Yes
IT Energy	Total server energy consumption	Yes
Water Usage	Cooling tower water consumption	Yes
Task Queue	Accumulated deferred workload not completed within the horizon	Yes

19.8.2 Running Evaluation

Use the `eval_sustaindc.py` script to evaluate a trained checkpoint:

```
uv run python eval_sustaindc.py --algo happo --exp_name happo_ny
```

Or evaluate programmatically within a notebook:

```python
from sustaindc_env import SustainDC, EnvConfig
import numpy as np

def evaluate_episode(env, policy_fn):
    """Run one full episode and collect metrics."""
    obs = env.reset()
    done = False
    metrics = {
        "carbon": [], "hvac_energy": [], "it_energy": [],
        "water": [], "rewards": {"agent_ls": 0, "agent_dc": 0, "agent_bat": 0},
    }

    while not done:
        actions = policy_fn(obs)
        obs, rewards, terminated, truncated, info = env.step(actions)

        dc_info = info["agent_dc"]
        metrics["carbon"].append(dc_info.get("bat_CO2_footprint", 0))
        metrics["hvac_energy"].append(dc_info.get("dc_HVAC_total_power_kW", 0))
        metrics["it_energy"].append(dc_info.get("dc_ITE_total_power_kW", 0))
        metrics["water"].append(dc_info.get("dc_water_usage", 0))

        for agent_id in metrics["rewards"]:
            if agent_id in rewards:
                metrics["rewards"][agent_id] += rewards[agent_id]

        done = terminated.get("__all__", False) or truncated.get("__all__", False)

    return {
        "total_carbon": metrics["carbon"][-1] if metrics["carbon"] else 0,
        "avg_hvac_kw": np.mean(metrics["hvac_energy"]),
        "avg_it_kw": np.mean(metrics["it_energy"]),
        "total_water": np.sum(metrics["water"]),
        "rewards": metrics["rewards"],
    }

# --- Baseline policy: do-nothing for each agent ---
DO_NOTHING = {"agent_ls": 1, "agent_dc": 1, "agent_bat": 2}

def baseline_policy(obs):
    return {agent: DO_NOTHING[agent] for agent in obs}
```

```
# --- Create environment ---
env_config = EnvConfig({
    "agents": ["agent_ls", "agent_dc", "agent_bat"],
    "month": 2,
    "location": "NY",
    "cintensity_file": "NYIS_NG_&_avgCI.csv",
    "weather_file": "USA_NY_New.York-Kennedy.Intl.AP.744860_TMY3.epw",
    "workload_file": "Alibaba_CPU_Data_Hourly_1.csv",
    "datacenter_capacity_mw": 1.0,
    "battery_capacity_mwh": 0.5,
    "flexible_load": 0.1,
    "individual_reward_weight": 0.8,
    "ls_reward": "default_ls_reward",
    "dc_reward": "default_dc_reward",
    "bat_reward": "default_bat_reward",
})

env = SustainDC(env_config)

baseline_results = evaluate_episode(env, baseline_policy)

print("=== Baseline Results ===")
print(f"  Carbon footprint: {baseline_results['total_carbon']:.2f} gCO2")
print(f"  Avg HVAC power:   {baseline_results['avg_hvac_kw']:.2f} kW")
print(f"  Avg IT power:     {baseline_results['avg_it_kw']:.2f} kW")
print(f"  Total water:      {baseline_results['total_water']:.2f} L")
```

19.9 Reward Customization

One of SustainDC's strengths is its flexible reward system. All reward functions are defined in `utils/reward_creator.py`.

19.9.1 Available Reward Functions

Reward Function	Target Metric
`default_dc_reward`	$-(E_{hvac} + E_{it}) \times CI_t$ (negative carbon footprint)
`default_ls_reward`	$-(CFP_t + LS_{penalty})$ (carbon + task completion penalty)

Reward Function	Target Metric
`default_bat_reward`	$-CFP_t$ (total carbon including battery)
`tou_reward`	Time-of-use electricity cost
`energy_PUE_reward`	Power Usage Effectiveness
`water_usage_efficiency_reward`	Cooling water consumption
`temperature_efficiency_reward`	Thermal constraint satisfaction

19.9.2 Changing Reward Functions

To train with a different objective, simply change the reward strings in the config:

```
env_config_custom = EnvConfig({
    "agents": ["agent_ls", "agent_dc", "agent_bat"],
    "month": 6,  # July - try a summer month for higher cooling loads
    "location": "NY",
    "cintensity_file": "NYIS_NG_&_avgCI.csv",
    "weather_file": "USA_NY_New.York-Kennedy.Intl.AP.744860_TMY3.epw",
    "workload_file": "Alibaba_CPU_Data_Hourly_1.csv",
    "datacenter_capacity_mw": 1.0,
    "battery_capacity_mwh": 0.5,
    "flexible_load": 0.1,
    # Use water usage as the DC reward instead of carbon
    "dc_reward": "water_usage_efficiency_reward",
    # Keep defaults for the other agents
    "ls_reward": "default_ls_reward",
    "bat_reward": "default_bat_reward",
})
```

19.9.3 Collaborative Reward Sharing (α)

The `individual_reward_weight` parameter controls α, the balance between an agent's own reward and the rewards from other agents:

$$R_{DC} = \frac{(1-\alpha)}{2} \cdot r_{LS} + \alpha \cdot r_{DC} + \frac{(1-\alpha)}{2} \cdot r_{BAT}$$

- $\alpha = 1.0$: fully independent — each agent only sees its own reward
- $\alpha = 0.8$: default — 80% own reward, 10% from each other agent
- $\alpha = 0.1$: highly collaborative — each agent mostly optimizes for the group

The paper (Section 6.2) shows that $\alpha = 0.8$ outperforms both extremes, especially in partially observable settings where agents benefit from indirect feedback about how their actions affect other subsystems.

```
# Ablation: try different alpha values
uv run python train_sustaindc.py --algo ippo --exp_name ippo_alpha10 --individual_reward_wei
uv run python train_sustaindc.py --algo ippo --exp_name ippo_alpha08 --individual_reward_wei
uv run python train_sustaindc.py --algo ippo --exp_name ippo_alpha01 --individual_reward_wei
```

19.10 Multi-Location Experiments

SustainDC ships with data for 8 US locations, chosen for their high data center density and diverse climates/grid mixes:

Location	Climate	Grid Characteristics
AZ (Arizona)	Hot, arid	High solar penetration
CA (California)	Mediterranean	Mixed renewables, moderate CI
GA (Georgia)	Hot, humid	Coal + nuclear, higher CI
IL (Illinois)	Continental	Nuclear-heavy, lower CI
NY (New York)	Continental	Gas + nuclear, moderate CI
TX (Texas)	Hot, variable	Wind + gas, volatile CI
VA (Virginia)	Humid subtropical	Highest DC density in the US
WA (Washington)	Marine	Hydro-heavy, very low CI

The data files are located in:

- `data/Weather/` — EPW files (Typical Meteorological Year format)
- `data/CarbonIntensity/` — hourly grid carbon intensity from the US EIA

Experiment Idea: Climate Impact on Control Strategy

Train the same HAPPO agent in Arizona (hot, high cooling load) and Washington (mild, low CI). Compare:

- Does the Arizona agent learn more aggressive cooling strategies?
- Does the Washington agent rely less on the battery (since the grid is already clean)?
- How do carbon footprints compare across locations?

```
uv run python train_sustaindc.py --algo happo --exp_name happo_az \
    --location AZ \
    --cintensity_file "AZPS_NG_&_avgCI.csv" \
    --weather_file USA_AZ_Davis-Monthan.AFB.722745_TMY3.epw

uv run python train_sustaindc.py --algo happo --exp_name happo_wa \
    --location WA \
    --cintensity_file "WAAT_NG_&_avgCI.csv" \
    --weather_file USA_WA_Port.Angeles-Fairchild.Intl.AP.727885_TMY3.epw
```

19.11 From PyDCM to SustainDC: What Changed?

If you completed the PyDCM tutorial, here is a summary of what SustainDC adds:

Aspect	PyDCM (`pydcm` branch)	SustainDC (`main` branch)
Agents	Single (cooling only)	Three (workload + cooling + battery)
Training	Ray RLlib PPO	HARL framework (10+ algorithms)
Baselines	Fixed action	Rule-based controllers (ASHRAE 36, etc.)
Reward	Single carbon footprint	Per-agent rewards with collaborative sharing
Locations	3 (NY, AZ, WA)	8 US locations
Workload data	Alibaba only	Alibaba + Google traces
Observation space	5–12 dimensions	26+ dimensions (includes CI trends, forecasts)
Evaluation	Energy + carbon	5 metrics (CFP, HVAC, IT, water, task queue)
Config	`DCRL` wrapper	`SustainDC` + `EnvConfig` with full YAML support

The underlying thermal and HVAC models (the vectorized NumPy code we examined) are the same. The difference is in the orchestration layer and the breadth of the experimental framework.

19.12 Next Steps

1. **Assignment 3**: Apply the Gnu-RL differentiable MPC policy (Paper 3) to the SustainDC DC cooling environment. Start in single-agent mode (`agents: ["agent_dc"]`) before attempting multi-agent.
2. **Algorithm comparison**: Reproduce the paper's radar charts (Figure 5) by training IPPO, MAPPO, and HAPPO on the same location and comparing the five metrics.
3. **Custom rewards**: Define a reward function in `utils/reward_creator.py` that balances carbon footprint and water usage, and observe how the learned policy changes.

20 Gnu-RL: A Precocial Reinforcement Learning Solution for Building HVAC Control Using a Differentiable MPC Policy

20.1 Overview

Citation

Bingqing Chen, Zicheng Cai, and Mario Berges. 2019. Gnu-RL: A Precocial Reinforcement Learning Solution for Building HVAC Control Using a Differentiable MPC Policy. In *The 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation (BuildSys '19)*, November 13–14, 2019, New York, NY, USA. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3360322.3360849

Sense, Plan, Act Mapping and Objective

Application domain: Building HVAC energy management and thermal comfort. Most HVAC systems are still operated by simple rule-based or PID controllers that do not leverage predictive information (weather forecasts, occupancy schedules), resulting in sub-optimal energy performance. Gnu-RL aims to make RL-based HVAC control practical and scalable by eliminating the need for high-fidelity simulation models.

- **Sense**: Not the primary innovation. The agent uses zone temperatures as state variables and weather data (outdoor temperature, humidity, solar radiation, wind speed) plus occupancy as disturbance inputs. In the real-world deployment, a depth-image-based occupancy sensor and weather API provide these inputs.
- **Plan**: **This is the primary innovation.** Gnu-RL replaces the standard neural network policy with a Differentiable Model Predictive Control (MPC) policy. This encodes domain knowledge about planning and system dynamics directly into the policy architecture. The learnable parameters are the linear state-space dynamics matrices (A, B_u, B_d) rather than thousands of neural network weights. Training proceeds in two phases: (1) offline imitation learning from an existing controller's historical data, and (2) online refinement via Proximal Policy Optimization (PPO), with gradients backpropagated through the MPC solver. The "precocial" nature

means the agent is competent from its first interaction with the environment—no millions of simulation steps required.

- **Act**: Controls HVAC setpoints. In the simulation experiment, the agent controls supply water temperature (20–65°C) for a radiant heating system. In the real-world deployment, it controls supply airflow (10–200 CFM) for a Variable Air Volume (VAV) box in a conference room.

20.2 Review of the paper

20.2.1 Summary

Gnu-RL (named after the precocial African herbivore) proposes a reinforcement learning agent that is practical to deploy for building HVAC control. The key insight is that replacing the neural network policy with a **Differentiable MPC policy** yields an agent that is sample-efficient, interpretable, and capable of good performance from the moment of deployment.

Key contributions:

1. **Precocial RL agent**: Unlike standard RL agents that require millions of interaction steps (47.5 simulated years in one comparison), Gnu-RL is pre-trained on historical data from existing controllers via imitation learning and performs well immediately upon deployment.
2. **Differentiable MPC policy**: The policy solves an MPC optimization problem in its forward pass and backpropagates through the KKT conditions of that optimization in its backward pass. The learnable parameters are the linear dynamics matrices $\theta = \{A, B_u, B_d\}$—far fewer parameters than a neural network, with clear physical interpretation.
3. **Two-phase training**:
   - **Phase 1 (Imitation Learning)**: The agent learns from historical state-action pairs logged by an existing controller, jointly fitting both the dynamics model and the control behavior by minimizing a combined loss: $\mathcal{L}_{imit}(\theta) = \sum_t \lambda(x_t - \hat{x}_t)^2 + (u_t - \hat{u}_t)^2$, where λ balances state prediction vs. action matching.
   - **Phase 2 (Online PPO)**: The pre-trained agent is deployed and continues to improve by maximizing expected reward via policy gradient updates.
4. **Imitation learning outperforms system identification**: The paper compares two initialization schemes and shows that imitation learning produces agents that track setpoints well from day one, while system identification (PEM) yields good prediction

error but poor control performance—demonstrating that small prediction error does not imply good control.

5. **Simulation results**: On an EnergyPlus model of a 600 m^2 office building (Carnegie Mellon's Intelligent Workspace), Gnu-RL saved 6.6% energy compared to the best published RL agent while maintaining better occupant comfort (lower PPD).
6. **Real-world deployment**: Deployed for three weeks in a 20 m^2 conference room on CMU's campus, controlling a VAV box. Gnu-RL saved 16.7% of cooling demand compared to the existing fixed-schedule controller while achieving significantly better temperature setpoint tracking (RMSE of 1.02°F vs. 2.4°F).

20.2.2 What do we know already?

This paper connects deeply to several concepts covered in the course:

From Lectures 3–5 (Thermal Dynamics of Buildings):

- The linear state-space model $x_{t+1} = Ax_t + B_u u_t + B_d d_t$ used by Gnu-RL is exactly the type of discrete-time thermal dynamics model we studied. The matrices A, B_u, B_d capture how zone temperatures evolve based on current state, control actions, and external disturbances.
- The assumption of local linearity around operating points—building dynamics are non-linear globally but approximately linear in the region of normal operation—is the same simplification used in our thermal RC network models.

From Lectures 8–9 (Control Theory, PID and MPC):

- Gnu-RL's Differentiable MPC policy directly builds on the MPC formulation covered in class: minimizing a cost function over a receding planning horizon subject to dynamics constraints and input bounds.
- The cost function $C_t(\tau_t) = \frac{\eta}{2}\sum(x_{t,i} - x_{i,setpoint})^2 + \sum|u_{t,i}|$ balances comfort (state deviation from setpoint) against energy (control effort), with η weighting their relative importance—the same trade-off we discussed in the MPC lecture.
- The comparison against P-controllers in both experiments connects to the PID control concepts from Lecture 8.

From Lecture 7 (Thermal Comfort):

- The paper uses Predicted Percentage Dissatisfied (PPD) as a comfort metric in the simulation experiment, directly connecting to the thermal comfort models we studied.
- The use of different η values for occupied ($\eta = 3$) vs. unoccupied ($\eta = 0.1$) periods reflects the occupancy-aware comfort management concepts from Lecture 7.

From Paper 2 (PyDCM/SustainDC):

- Paper 2 already introduced RL fundamentals (MDPs, PPO, policy gradient methods) and the Gymnasium interface. Those concepts apply directly here.
- Paper 2's "Connecting to Gnu-RL" section previewed the Differentiable MPC policy, the two-phase training, and how to adapt Gnu-RL for data center cooling. This paper provides the full technical details behind that preview.
- The key contrast: Paper 2's SustainDC uses standard neural network policies (MAPPO, HAPPO, etc.) that require extensive simulation, while Gnu-RL uses a structured MPC policy that needs only historical data from an existing controller.

Things to learn more about

To fully understand this paper's contributions, students may need background on:

1. **Differentiable optimization**:
   - How to differentiate through the solution of an optimization problem
   - Karush-Kuhn-Tucker (KKT) conditions for constrained optimization
   - The OptNet framework for embedding optimization as a neural network layer
   - Automatic differentiation (backpropagation through the MPC solver)
2. **Imitation learning**:
   - Learning from expert demonstrations (behavioral cloning)
   - The distinction between imitation learning and system identification: imitation learning matches both dynamics *and* actions, while system identification only matches dynamics
   - Why good prediction error does not guarantee good control performance
3. **Model Predictive Control details**:
   - Receding horizon control: solve over a planning horizon T, apply only the first action, re-plan at the next step
   - The role of the planning horizon length (12 steps = 3 hours in the simulation)
   - How input constraints ($\underline{u} \leq u \leq \overline{u}$) are handled in the optimization
   - Re-parameterization of the policy output as a Gaussian distribution around the MPC solution for use with policy gradient methods
4. **System identification vs. imitation learning**:
   - Prediction Error Methods (PEM) for estimating dynamic system parameters
   - Why system identification requires excitation signals that may disturb normal operation
   - The paper's key finding that imitation learning is superior for control initialization
5. **Real-world deployment challenges**:

- Communication delays with the Building Automation System (BAS)
- Sensor noise and equipment issues (e.g., the reheat coil leakage discovered during deployment)
- Adapting to discrepancies between expected and actual disturbances (e.g., occupancy sensor counting errors)

20.3 Methods

The paper employs several technical methods that should be expanded upon in class discussion:

20.3.1 1. Differentiable MPC Policy

The core innovation replaces the neural network policy $\pi_\theta(u|x)$ with an MPC optimization layer:

- **Forward pass**: Solves a constrained quadratic program to find the optimal control trajectory $\tau^*_{1:T} = \{x^*_t, u^*_t\}_{1:T}$ that minimizes the cost function subject to linear dynamics and input constraints
- **Backward pass**: Computes gradients $\nabla_\theta \mathcal{L}$ by differentiating through the KKT conditions of the optimization problem, using techniques from OptNet
- **Learnable parameters**: $\theta = \{A, B_u, B_d\}$—the state-space dynamics matrices. The cost function parameters (e.g., η) can also be learned but were fixed in the experiments
- **Action selection**: Only the first optimal action u^*_t is applied (receding horizon), and the policy is re-parameterized as $\hat{u}_t \sim \mathcal{N}(u^*_t, \sigma^2)$ for compatibility with policy gradient methods

20.3.2 2. Linear State-Space Dynamics Model

$$x_{t+1} = f_\theta(\tau_t) = Ax_t + B_u u_t + B_d d_t$$

- Assumes local linearity of building thermal dynamics around operating conditions
- Disturbance terms d_t include weather variables and occupancy, provided over the planning horizon $d_{t:t+T-1}$
- The small number of free parameters makes the model interpretable—engineers can inspect A, B_u, B_d to verify physical plausibility

20.3.3 3. Cost Function Design

$$C_t(\tau_t) = \frac{\eta}{2} \sum_{i=1}^{\#states} (x_{t,i} - x_{i,setpoint})^2 + \sum_{i=1}^{\#actions} |u_{t,i}|$$

- η balances comfort (L2-norm of state deviation from setpoint) against energy (L1-norm of control actions)
- Different η values for occupied ($\eta = 3$) vs. unoccupied ($\eta = 0.1$) periods encode the insight that comfort matters more when people are present
- The cost function is specified by the engineer, not learned—this is where domain knowledge enters beyond the dynamics

20.3.4 4. Imitation Learning (Algorithm 1)

- Collects state-action demonstrations (X, U) from an existing controller's historical data
- Jointly minimizes state prediction loss and action matching loss: $\mathcal{L}_{imit}(\theta) = \sum_t \lambda(x_t - \hat{x}_t)^2 + (u_t - \hat{u}_t)^2$
- Hyperparameter λ balances dynamics learning vs. behavioral cloning; can be increased if the existing controller's actions are of low quality
- Uses RMSprop optimizer with learning rate 1×10^{-4}

20.3.5 5. Online Learning with PPO (Algorithm 2)

- After pre-training, deploys the agent and continues training with Proximal Policy Optimization
- PPO loss: $\mathcal{L}_{PPO}(\theta) = -\hat{\mathbb{E}}_t[\min(w_t(\theta)\hat{A}_t, \text{clip}(w_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$
- Gradients flow through the MPC solver to update the dynamics parameters θ
- The paper demonstrates that maximizing expected reward (PPO objective) produces better controllers than minimizing prediction error (adaptive MPC objective), even though the latter yields smaller state prediction errors

20.3.6 6. Comparison: Policy Gradient vs. Prediction Error Minimization

- **Adaptive MPC** ($\mathcal{L}_{PEM}$): Updates parameters online to minimize $\sum_t (x_t - \hat{x}_t)^2$—i.e., improve the model's predictive accuracy
- **Gnu-RL** ($\mathcal{L}_{PPO}$): Updates parameters to maximize expected reward—i.e., improve actual control performance
- Key finding: $\mathcal{L}_{PPO}$ consistently yields larger residual reward (better control) despite $\mathcal{L}_{PEM}$ achieving smaller prediction error. This demonstrates that prediction accuracy is only a proxy for control quality, and directly optimizing the task objective is more effective.

20.4 Hands-On Tutorial

The concepts covered above — differentiable MPC, imitation learning, and online PPO refinement — are explored in depth through a companion tutorial:

- **Hands-On: Getting Started with Gnu-RL**: Walks through the Gnu-RL codebase and its Differentiable MPC module, imitation learning and online PPO training pipelines, and interpretation of the original authors' pre-computed results.

21 Hands-On: Getting Started with Gnu-RL

21.1 Overview

Learning Objectives

By the end of this activity, students will be able to:

- Set up the Gnu-RL codebase with modern PyTorch and understand its repository structure
- Run the Differentiable MPC solver on a simplified state-space environment
- Explain how the MPC policy works as a PyTorch module and how gradients flow backward through the solver
- Trace the imitation learning pipeline: how expert data trains the dynamics model (F, B_d) through backpropagation through the MPC
- Trace the online PPO pipeline: how the agent refines its dynamics model through environment interaction
- Interpret the pre-computed results from the original authors' experiments
- Identify the environment interface calls that must be replaced to apply Gnu-RL to a different environment (e.g., SustainDC for Assignment 3)

Prerequisites

This tutorial assumes familiarity with:

- The Paper 3: Gnu-RL chapter, which covers the paper's contributions, the Differentiable MPC policy, and the two-phase training approach
- The SustainDC hands-on tutorial, which covers the data center environment you will use in Assignment 3
- The Assignment 3 instructions, so you understand what you are building toward
- Basic PyTorch: tensors, `requires_grad`, optimizers, and `loss.backward()`
- Model Predictive Control (MPC) concepts from Lectures 8–9

! Run This on Your Own Machine

The code in this notebook requires the `Gnu-RL` repository and its dependencies. It is **not** executed during the book build. Follow along by running each cell in your own Python environment.
Important: The original Gnu-RL code requires **EnergyPlus** (a building energy simulation engine) for its full training pipeline. We will **not** use EnergyPlus in this tutorial. Instead, we work with:

1. The **Differentiable MPC module** (`diff_mpc/`) — pure PyTorch, no external dependencies
2. A **simplified state-space environment** (`ssM_env/`) — a lightweight linear system simulator
3. **Pre-computed results** from the original authors' experiments

The original codebase was written for PyTorch 1.0 (2018). A `pytorch2-compat` branch with all necessary patches is provided so you don't need to modify anything manually.

21.2 Environment Setup

21.2.1 Cloning the Repository

Clone the `pytorch2-compat` branch, which contains the original code with compatibility patches for modern PyTorch (≥ 2.0):

```
git clone -b pytorch2-compat https://github.com/INFERLab/Gnu-RL.git gnu-rl
cd gnu-rl
```

What changed in the `pytorch2-compat` branch?

The `diff_mpc/` module required four categories of updates to work with PyTorch 2.x:

1. **Autograd Function API** (`lqr_step.py`): PyTorch 2.x requires `torch.autograd.Function` subclasses to use `@staticmethod` for `forward` and `backward`. The old instance-method style was removed. The patched code splits `LQRStep` into a wrapper class (holds config) and `_LQRStepFunction` (static autograd methods that receive config via `ctx`).

2. **Removed linear algebra ops** (`pnqp.py`, `lqr_step.py`): `torch.btrifact()` and `torch.btrisolve()` were removed in favor of `torch.linalg.lu_factor()` and `torch.linalg.lu_solve()`.

3. **Boolean tensor handling** (`util.py`, `pnqp.py`, `lqr_step.py`): `.byte()` masks replaced with proper `torch.bool` tensors; arithmetic negation `1 - mask` replaced with bitwise `~mask`.

4. **Variable wrapping** (`mpc.py`): `torch.autograd.Variable` was a no-op wrapper since PyTorch 0.4 and has been fully removed. All `Variable(...)` calls replaced with direct tensor operations.

None of these changes affect the mathematical behavior of the solver — they are purely API migrations.

21.2.2 Installing Dependencies

Initialize a `uv` project and install the required packages:

```
uv init --python 3.10
uv add torch numpy pandas matplotlib requests
```

21.2.3 Verifying the Setup

A quick sanity check — import the MPC solver and run a trivial optimization:

```
import torch
from diff_mpc.mpc import MPC, QuadCost, LinDx

# 1D system: x_{t+1} = 0.9*x + 0.1*u, drive state toward 24
n_state, n_ctrl, T = 1, 1, 5
F = torch.zeros(T-1, 1, n_state, n_state + n_ctrl)
F[:, :, 0, 0] = 0.9  # state dynamics
F[:, :, 0, 1] = 0.1  # control influence
f = torch.zeros(T-1, 1, n_state)

C = torch.zeros(T, 1, n_state + n_ctrl, n_state + n_ctrl)
C[:, :, 0, 0] = 1.0   # penalize state deviation
C[:, :, 1, 1] = 0.01  # penalize control effort
c = torch.zeros(T, 1, n_state + n_ctrl)
c[:, :, 0] = -24.0     # target state = 24

x_init = torch.tensor([[20.0]])
```

```
solver = MPC(n_state, n_ctrl, T,
             u_lower=0.0, u_upper=50.0,
             lqr_iter=20, verbose=0,
             exit_unconverged=False)
x, u, costs = solver(x_init, QuadCost(C, c), LinDx(F, f))

print(f"Initial state: {x_init[0, 0]:.1f}")
print(f"Planned states:   {x[:, 0, 0].detach().numpy()}")
print(f"Planned controls: {u[:, 0, 0].detach().numpy()}")
```

You should see the MPC driving the state from 20 toward 24 by applying large control actions early, then tapering off.

21.3 Repository Walkthrough

The repository has three main parts: the **environment-agnostic MPC solver** (`diff_mpc/`), the **environment-specific training scripts** (`agent/`), and a **lightweight test environment** (`ssM_env/`). For Assignment 3, you will reuse `diff_mpc/` as-is and adapt the training logic from `agent/` to work with SustainDC instead of EnergyPlus.

21.3.1 Key Files

```
gnu-rl/
  diff_mpc/                 # Differentiable MPC solver (pure PyTorch) ← KEEP AS-
IS
      mpc.py              # MPC class: orchestrates iLQR iterations
      lqr_step.py         # Single LQR backward/forward sweep (autograd Function)
      pnqp.py             # Box-constrained QP solver for control bounds
      dynamics.py         # Dynamics model wrappers (LinDx, etc.)
      util.py             # Batch matrix utilities (bmv, bger, bdiag)

  ssM_env/                  # Simplified state-space environment (no EnergyPlus)
      env.py              # Linear system: x_{t+1} = Ax + Bu*u + Bd*d

  agent/                    # Training scripts ← ADAPT FOR SUSTAINDC
      simulate.py         # Step 0: Run baseline controller, collect data
      Imit_EP.py          # Step 1: Offline imitation learning (Learner class)
      PPO_MPC_EP.py       # Step 2: Online PPO refinement
      utils.py            # Reward function, data utilities
      results/            # Pre-computed results from the authors
```

```
eplus_env/              # EnergyPlus Gym wrapper (not used)
Demo.ipynb              # Results visualization notebook
```

21.3.2 The Two-Phase Pipeline in Code

The two training phases from the paper map directly to files:

Phase	File	What it does
0. Data collection	`agent/simulate.py`	Runs the existing controller in EnergyPlus, logs (x, u, d) trajectories
1. Imitation learning	`agent/Imit_EP.py`	`Learner` class with learnable F, B_d; trains by backpropagating $\mathcal{L}_{imit}$ through the MPC solver
2. Online PPO	`agent/PPO_MPC_EP.py`	Wraps MPC output in a Gaussian policy; updates F, B_d via PPO to maximize reward

The `diff_mpc/` module is called by both phases but is never modified during training — only the dynamics parameters (F, B_d) that are *passed into* it are updated.

21.4 Understanding the Pre-Computed Results

The authors provide pre-computed results in `agent/results/` so you can visualize the full training pipeline without running EnergyPlus. The code below is adapted from `Demo.ipynb`. Run these cells to reproduce the four key plots from the paper.

21.4.1 Setup

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

filePath = "agent/results/"
lam = 10  # weight on state loss in the imitation objective
```

21.4.2 Imitation Learning Loss Curves

This plot shows how $\mathcal{L}_{state}$, $\mathcal{L}_{action}$, and the combined $\mathcal{L}_{imit} = \lambda\mathcal{L}_{state} + \mathcal{L}_{action}$ evolve over 20 epochs. Look for the train/validation gap — a large gap would indicate overfitting of the dynamics parameters.

```
imit_loss = pd.read_pickle(filePath + "Imit_loss_rl.pkl")

fig, axes = plt.subplots(1, 3, figsize=(16, 4))
for ax, title, train_key, val_key in [
    (axes[0], r"$\mathcal{L}_{state}$", "train_state_loss", "val_state_loss"),
    (axes[1], r"$\mathcal{L}_{action}$", "train_action_loss", "val_action_loss"),
]:
    ax.plot(imit_loss[train_key], label="train")
    ax.plot(imit_loss[val_key], label="val")
    ax.set_title(title)
    ax.set_xlabel("Epoch")
    ax.set_xlim((0, 19))
    ax.legend()

# Combined loss
axes[2].plot(lam * imit_loss["train_state_loss"] + imit_loss["train_action_loss"], label="tra
axes[2].plot(lam * imit_loss["val_state_loss"] + imit_loss["val_action_loss"], label="val")
axes[2].set_title(r"$\mathcal{L}_{imit}$")
axes[2].set_xlabel("Epoch")
axes[2].set_xlim((0, 19))
axes[2].legend()
plt.tight_layout()
plt.show()
```

21.4.3 Agent Behavior After Pretraining

After imitation learning, the agent should track the expert controller's states and actions closely. Each epoch's results are saved as `Imit_rl_{epoch}.pkl` — try different epochs to see how tracking improves during training.

```
epoch = 19  # best epoch (last one); try 0 vs. 19 to see improvement
imit_record = pd.read_pickle(filePath + f"Imit_rl_{epoch}.pkl")

# Setpoint and occupancy live in the simulation dataset, not the imitation record
sim_data = pd.read_pickle(filePath + "Sim-TMY2.pkl").loc[imit_record.index]
```

```
# Reset to integer index so set_xlim works with integer bounds
imit_record = imit_record.reset_index(drop=True)
sim_data = sim_data.reset_index(drop=True)

start_time, end_time = 0, 500  # adjust to zoom in/out

fig, axes = plt.subplots(2, 1, figsize=(20, 5), sharex=True)
axes[0].plot(imit_record["Expert nState"], label="Expert")
axes[0].plot(imit_record["Learner nState"], label="Learner")
axes[0].plot(sim_data["Indoor Temp. Setpoint"], "k--", label="Setpoint")
axes[0].set_ylabel("Next State (Temperature)")
axes[0].set_xlim((start_time, end_time))
axes[0].legend()

axes[1].plot(imit_record["Expert action"], label="Expert")
axes[1].plot(imit_record["Learner action"], label="Learner")
axes[1].plot(sim_data["Occupancy Flag"] * 5, "k--", label="Occupancy")
axes[1].set_ylabel("Action (Supply Temp.)")
axes[1].set_xlim((start_time, end_time))
axes[1].legend()
plt.tight_layout()
plt.show()
```

21.4.4 Online Learning Performance

After PPO refinement, compare Gnu-RL against the EnergyPlus baseline controller. Notice how Gnu-RL learns to **pre-heat** before occupancy periods — the agent anticipates the setpoint change rather than reacting to it.

```
baseline = pd.read_pickle(filePath + "Sim-TMY3.pkl").reset_index(drop=True)
rl = pd.read_pickle(filePath + "perf_rl_obs.pkl").reset_index(drop=True)

start_time, end_time = 2000, 2500  # a representative week

fig, axes = plt.subplots(2, 1, figsize=(20, 6), sharex=True)
axes[0].plot(baseline["Indoor Temp."], "b-", label="Baseline")
axes[0].plot(rl["Indoor Temp."], "r-", label="Gnu-RL")
axes[0].plot(rl["Indoor Temp. Setpoint"], "k--", label="Setpoint")
axes[0].set_ylabel("Indoor Temperature")
axes[0].set_xlim([start_time, end_time])
axes[0].legend()
```

```
axes[1].plot(baseline["Sys Out Temp."], "b", label="Baseline")
axes[1].plot(rl["Sys Out Temp."], "r", label="Gnu-RL")
axes[1].plot(baseline["Occupancy Flag"] * 30, "k--", label="Occupancy")
axes[1].set_ylabel("Supply Air Temp. (Action)")
axes[1].set_xlim([start_time, end_time])
axes[1].legend()
plt.tight_layout()
plt.show()
```

21.4.5 Residue Reward Over Time

The residue reward is the difference between Gnu-RL's reward and the baseline's reward at each timestep. A positive trend confirms that PPO is improving the policy over the 90-day online learning period.

```
rl_perf = np.load(filePath + "perf_rl.npy", allow_pickle=True)
baseline_perf = pd.read_pickle(filePath + "Sim-TMY3.pkl")

# Compute residue reward (moving average with confidence band)
residue = pd.Series(rl_perf[:, 0])  # reward column (one value per episode/day)
RewardStats = residue.rolling(window=7).agg(["mean", "std"]).dropna()  # 7-day moving averag
RewardStats.columns = ["Mean", "Std"]

fig = plt.figure(figsize=(10, 4))
plt.plot(RewardStats["Mean"])
plt.fill_between(
    RewardStats.index,
    RewardStats["Mean"] - 1.645 * RewardStats["Std"],
    RewardStats["Mean"] + 1.645 * RewardStats["Std"],
    alpha=0.2,
)
plt.axhline(y=0, color="k", linestyle="--")
plt.ylabel("Residue Reward")
plt.xlabel("Episode (day)")
plt.tight_layout()
plt.show()
```

21.5 Code Deep-Dive: The Differentiable MPC Policy

You do not need to modify anything inside `diff_mpc/`. What you *do* need is a clear understanding of the **interface** — what you pass in and what you get back — so you can wire it into your own training loop for SustainDC.

21.5.1 The MPC Interface

The solver is called as:

```
x, u, _ = mpc.MPC(n_state, n_ctrl, T, **options)(x_init, QuadCost(C, c), LinDx(F, f))
```

The "Shape" column below gives the PyTorch tensor dimensions. These are 3D or 4D tensors because the solver is vectorized over a batch dimension (B) and a time dimension (T). Let $n = n_{state}$, $m = n_{ctrl}$, $B = n_{batch}$:

Argument	Shape	Dimensions	Description
`x_init`	`[n_batch, n_state]`	$\in \mathbb{R}^{B \times n}$	Current state (e.g., zone temperatures)
`C`	`[T, n_batch, n_state+n_ctrl, n_state+n_ctrl]`	$\in \mathbb{R}^{T \times B \times (n+m) \times (n+m)}$	Quadratic cost matrix (diagonal: weights on states and controls)
`c`	`[T, n_batch, n_state+n_ctrl]`	$\in \mathbb{R}^{T \times B \times (n+m)}$	Linear cost vector (encodes target setpoints)
`F`	`[T-1, n_batch, n_state, n_state+n_ctrl]`	$\in \mathbb{R}^{(T-1) \times B \times n \times (n+m)}$	Dynamics matrix: $x_{t+1} = F \begin{bmatrix} x_t \\ u_t \end{bmatrix} + f_t$
`f`	`[T-1, n_batch, n_state]`	$\in \mathbb{R}^{(T-1) \times B \times n}$	Affine dynamics term (disturbance contribution: $B_d d_t$)

Return	Shape	Dimensions	Description
`x`	`[T, n_batch, n_state]`	$\in \mathbb{R}^{T \times B \times n}$	Optimal state trajectory

Return	Shape	Dimensions	Description
`u`	`[T, n_batch, n_ctrl]`	$\in \mathbb{R}^{T \times B \times m}$	Optimal control trajectory (apply `u[0]` only — receding horizon)

Key solver options: `u_lower`/`u_upper` (control bounds), `lqr_iter` (number of iLQR iterations, typically 20), `exit_unconverged=False` (don't error if solver doesn't fully converge).

21.5.2 How F and f Encode the Dynamics

The dynamics $x_{t+1} = Ax_t + B_u u_t + B_d d_t$ are split into two parts for the solver:

- F: The concatenation $[A \mid B_u]$, shape `[n_state, n_state + n_ctrl]`. This is **learnable**.
- f: The disturbance term $B_d d_t$, pre-computed for the planning horizon. B_d is **learnable**; d_t comes from weather/workload forecasts.

In code (from `Imit_EP.py`):

```
# F_hat is [n_state, n_state + n_ctrl] --- the learnable dynamics
F_hat = torch.ones((n_state, n_state + n_ctrl)).double().requires_grad_()

# Bd_hat is [n_state, n_dist] --- the learnable disturbance matrix
Bd_hat = torch.tensor(np.random.rand(n_state, n_dist)).requires_grad_()

# f is computed by multiplying Bd_hat with the disturbance forecast
dt = ...  # [n_dist, T-1] disturbance values over the horizon
ft = torch.mm(Bd_hat, dt).transpose(0, 1).unsqueeze(1)  # [T-1, 1, n_state]
```

For Assignment 3, you will define what x, u, and d mean for SustainDC's cooling environment, then initialize `F_hat` and `Bd_hat` with appropriate shapes.

21.5.3 How C and c Encode the Cost Function

The MPC minimizes $\sum_t \frac{1}{2}\tau_t^T C_t \tau_t + c_t^T \tau_t$ where $\tau_t = [x_t; u_t]$. In Gnu-RL, the cost balances comfort against energy:

```
# Diagonal of C: [eta, eta, ..., 0.001, 0.001, ...]
#                 ^-- n_state --^  ^-- n_ctrl --^
# eta depends on occupancy: high when occupied, low when not
diag[:, :n_state] = eta_w_flag   # comfort weight (varies by timestep)
diag[:, n_state:] = 0.001        # energy weight (constant)

# Linear cost c encodes the setpoint target
c[:, :n_state] = -eta * x_target  # drives state toward setpoint
c[:, n_state:] = 1                # L1 penalty on control effort
```

For Assignment 3, you will design a cost function appropriate for data center cooling — balancing temperature constraint satisfaction against energy consumption (and potentially carbon intensity).

21.5.4 Differentiability

The solver's backward pass (in `lqr_step.py`) computes $\partial u^*/\partial F$ and $\partial u^*/\partial B_d$ by differentiating through the KKT conditions of the optimization. You don't need to understand the internals — just know that if `F_hat` and `Bd_hat` have `requires_grad=True`, then calling `loss.backward()` on any loss computed from the MPC output will produce gradients for these parameters.

21.6 Code Deep-Dive: Imitation Learning

`agent/Imit_EP.py` contains the `Learner` class that implements Phase 1 of the training pipeline. This is the file you will adapt most directly for Assignment 3.

21.6.1 The `Learner` Class

The class has four key methods:

Method	What it does	Assignment 3 adaptation
`__init__`	Initializes `F_hat`, `Bd_hat` with `requires_grad=True`; sets up Adam optimizer	Change shapes to match SustainDC's state/control/disturbance dimensions
`Cost_function(cur_time)`	Builds C, c matrices from occupancy schedule and setpoint targets	Replace with SustainDC's cost structure (temperature bounds, energy, carbon)

Method	What it does	Assignment 3 adaptation
`forward(x_init, C, c, cur_time)`	Calls `mpc.MPC(...)` and returns next predicted state + first control action	Replace disturbance lookup (EnergyPlus timestamps → SustainDC observations)
`predict(x_init, action, cur_time)`	One-step dynamics: $\hat{x}_{t+1} = F \cdot [x_t; u_t] + B_d \cdot d_t$	Same timestamp adaptation as `forward`
`update_parameters(loss)`	Computes $\mathcal{L}_{imit}$, calls `backward()`, steps optimizer	Reusable as-is (loss structure doesn't depend on environment)

21.6.2 The Training Loop

The loop in `main()` follows this structure per epoch:

1. **Sample** a random timestep from the training set
2. **Get expert data**: the existing controller's state x_t and action u_t at that timestep
3. **Run MPC** (`learner.forward`): given x_t, produce the learner's predicted action $\hat{u}_t$
4. **Predict next state** (`learner.predict`): given x_t and the expert's u_t, produce $\hat{x}_{t+1}$
5. **Every 256 samples**: compute loss and backpropagate

The key detail: step 3 uses the MPC to generate $\hat{u}$, while step 4 uses one-step dynamics to generate $\hat{x}$. Both pathways produce gradients for `F_hat` and `Bd_hat`.

For Assignment 3, you will collect expert demonstrations by running your baseline controller in SustainDC, then adapt this loop to sample from that collected data.

21.6.3 The Imitation Loss

```
state_loss  = mean((x_true - x_pred)^2)
action_loss = mean((u_true - u_pred)^2)
traj_loss   = lambda * state_loss + action_loss  # lambda = eta = 5
```

The λ parameter balances two objectives:

- **State loss** (dynamics accuracy): does the learned model predict the right next state?
- **Action loss** (behavioral cloning): does the MPC produce the same action as the expert?

Both matter. Good dynamics alone don't guarantee good control (this is the paper's key finding vs. system identification).

21.7 Code Deep-Dive: Online PPO

`agent/PPO_MPC_EP.py` contains the `PPO` class that implements Phase 2. It reuses the same `F_hat`/`Bd_hat` parameters (loaded from imitation learning weights) and continues updating them via policy gradient.

21.7.1 From MPC Solution to Stochastic Policy

The MPC produces a deterministic action u^*. PPO needs a stochastic policy for exploration. The solution is simple — wrap u^* in a Gaussian:

```
# In PPO.select_action():
dist = Normal(mu=u_mpc, sigma=sigma)  # sigma decays over episodes
action = dist.sample()
log_prob = dist.log_prob(action)
```

σ starts at 1.0 and decays linearly to 0.1 over 90 episodes (`sigma = 1 - 0.9 * i_episode / tol_eps`), so the agent explores broadly early and exploits later.

21.7.2 The PPO Update

The `update_parameters` method implements the clipped surrogate objective:

```
# Recompute MPC action under current parameters
opt_states, opt_actions = self.forward(states, f, C, c, current=True)
log_probs, _ = self.evaluate_action(opt_actions[0], actions, sigma)

# PPO clipped surrogate
ratio = exp(log_probs - old_log_probs)
surr1 = ratio * advantage
surr2 = clamp(ratio, 1 - epsilon, 1 + epsilon) * advantage
loss  = -min(surr1, surr2).mean()

loss.backward()  # gradients flow through MPC to F_hat, Bd_hat
optimizer.step()
```

The critical point: `loss.backward()` differentiates through the MPC solver, so `F_hat` and `Bd_hat` are updated to maximize expected reward, not just to minimize prediction error.

21.7.3 Environment Interaction Loop

The `main()` function contains the EnergyPlus-specific interaction loop. For Assignment 3, this is the main part you replace:

Gnu-RL (EnergyPlus)	Your code (SustainDC)
`env = gym.make('5Zone-control_TMY3-v0')`	`env = gym.make('sustaindc/SustainDC-v0', ...)`
`timeStep, obs, isTerminal = env.reset()`	`obs, info = env.reset()`
`timeStep, obs, isTerminal = env.step([SAT_stpt])`	`obs, reward, terminated, truncated, info = env.step(action)`
Disturbances from pre-loaded weather CSV	Disturbances from SustainDC's observation space
Reward via `R_func(obs_dict, action, eta)`	Reward from `env.step()` or custom reward function
Episode = 1 natural day (96 steps @ 15 min)	Episode defined by SustainDC config

The `PPO` class itself (the MPC call, action selection, parameter update) is largely reusable — it's the environment interaction in `main()` that needs rewriting.

21.8 Hands-On: End-to-End Example on `ssM_env`

This section walks through the complete Gnu-RL pipeline on `ssM_env` — a lightweight linear system that requires no EnergyPlus. The pattern here is exactly what you will replicate with SustainDC in Assignment 3.

21.8.1 Step 1: Define the Environment and Collect Expert Data

```
import numpy as np
import torch
import torch.optim as optim
import matplotlib.pyplot as plt
from ssM_env.env import ssModel
from diff_mpc.mpc import MPC, QuadCost, LinDx

# --- Environment: single-zone thermal model ---
```

```
A  = np.array([[0.95]])          # thermal inertia
Bu = np.array([[0.05]])          # heating effectiveness
Bd = np.array([[0.02, 0.01]])    # weather sensitivity [outdoor_temp, solar]
C_obs = np.array([[1.0]])        # observe temperature directly

# 1 day of disturbance data (96 steps @ 15 min)
n_steps = 96
hours = np.arange(n_steps) / 4
outdoor_temp = 5 + 10 * np.sin(2 * np.pi * (hours - 6) / 24)
solar = np.maximum(0, 3 * np.sin(2 * np.pi * (hours - 6) / 24))
d = np.stack([outdoor_temp, solar])  # [2, 96]

ssMatrix = {'A': A, 'Bu': Bu, 'Bd': Bd, 'C': C_obs, 'd': d}
env = ssModel(ssMatrix, x_init=18, x_target=22, eta=5)

# --- Collect expert data with a proportional controller ---
Kp = 2.0
expert_states, expert_actions, expert_disturbances = [], [], []

obs = env.reset()
for t in range(n_steps - 1):
    temp = obs[0]
    action = np.array([Kp * max(0, 22 - temp)])  # P-controller toward setpoint
    expert_states.append(temp)
    expert_actions.append(action[0])
    expert_disturbances.append(d[:, t])
    obs, reward = env.step(action)

expert_states.append(obs[0])  # final state
print(f"Collected {len(expert_actions)} expert transitions")
```

21.8.2 Step 2: Learn Dynamics from Expert Data

We initialize `F_hat` and `Bd_hat` with wrong values, then learn them by minimizing the one-step state prediction error $\|F[x_t; u_t] + B_d d_t - x_{t+1}\|^2$. This is the state prediction component of Gnu-RL's imitation loss (the action matching component is added in the full pipeline with more data).

```
n_state, n_ctrl, n_dist = 1, 1, 2

# Vectorize training data
```

```
X = torch.tensor(
    np.column_stack([expert_states[:-1], expert_actions]), dtype=torch.float64)  # [N, 2]
D = torch.tensor(d[:, :len(expert_actions)].T, dtype=torch.float64)              # [N, 2]
Y = torch.tensor(expert_states[1:], dtype=torch.float64)                         # [N]

# --- Learnable parameters (initialized wrong) ---
F_hat  = torch.tensor([[0.5, 0.2]], dtype=torch.float64, requires_grad=True)   # true: [0.95
Bd_hat = torch.tensor([[0.1, 0.1]], dtype=torch.float64, requires_grad=True)   # true: [0.02
optimizer = optim.Adam([F_hat, Bd_hat], lr=0.05)

losses = []
for epoch in range(500):
    optimizer.zero_grad()
    pred = (X @ F_hat.T + D @ Bd_hat.T).squeeze()  # vectorized one-step prediction
    loss = ((pred - Y)**2).mean()
    loss.backward()
    optimizer.step()
    losses.append(loss.item())
    if (epoch + 1) % 100 == 0:
        print(f"Epoch {epoch+1:3d}  loss={losses[-1]:.6f}  "
              f"F_hat={F_hat.detach().numpy().round(3)}  "
              f"Bd_hat={Bd_hat.detach().numpy().round(3)}")

print(f"\nTrue F:  [0.95, 0.05]   Learned: {F_hat.detach().numpy().round(3)}")
print(f"True Bd: [0.02, 0.01]   Learned: {Bd_hat.detach().numpy().round(3)}")
```

```
plt.plot(losses)
plt.xlabel("Epoch")
plt.ylabel("State Prediction Loss")
plt.title("Learning Dynamics Parameters")
plt.show()
```

You should see `F_hat` converge toward `[0.95, 0.05]` and `Bd_hat` toward `[0.02, 0.01]`.

21.8.3 Step 3: Run the MPC with Learned Dynamics

Now verify that the learned dynamics produce a useful MPC controller — this closes the loop between learning and control:

```
T = 12
C_mpc = torch.zeros(T, 1, n_state + n_ctrl, n_state + n_ctrl, dtype=torch.float64)
C_mpc[:, :, 0, 0] = 1.0      # comfort weight
C_mpc[:, :, 1, 1] = 0.001    # energy weight
c_mpc = torch.zeros(T, 1, n_state + n_ctrl, dtype=torch.float64)
c_mpc[:, :, 0] = -22.0       # target = 22

env2 = ssModel(ssMatrix, x_init=18, x_target=22, eta=5)
obs = env2.reset()
mpc_temps, mpc_actions = [obs[0]], []

with torch.no_grad():
    for t in range(n_steps - 1):
        x_init = torch.tensor([[obs[0]]], dtype=torch.float64)
        dt = torch.tensor(d[:, t:t+min(T-1, n_steps-1-t)], dtype=torch.float64)
        if dt.shape[1] < T - 1:
            dt = torch.cat([dt, dt[:, -1:].repeat(1, T-1-dt.shape[1])], dim=1)
        ft = torch.mm(Bd_hat.detach(), dt).T.unsqueeze(1)

        x_opt, u_opt, _ = MPC(
            n_state, n_ctrl, T,
            u_lower=0.0, u_upper=50.0,
            lqr_iter=20, verbose=-1,
            exit_unconverged=False,
        )(x_init, QuadCost(C_mpc, c_mpc),
          LinDx(F_hat.detach().repeat(T-1, 1, 1, 1), ft))

        action = np.array([max(0, u_opt[0, 0, 0].item())])
        mpc_actions.append(action[0])
        obs, reward = env2.step(action)
        mpc_temps.append(obs[0])

hours = np.arange(len(mpc_temps)) / 4
fig, axes = plt.subplots(2, 1, figsize=(12, 5), sharex=True)
axes[0].plot(hours, mpc_temps, label="MPC (learned dynamics)")
axes[0].axhline(y=22, color='k', linestyle='--', label="Setpoint")
axes[0].set_ylabel("Temperature")
axes[0].legend()
axes[1].step(hours[:-1], mpc_actions, where='post')
axes[1].set_ylabel("Control Action")
axes[1].set_xlabel("Hours")
plt.tight_layout()
```

```
plt.show()
```

21.8.4 Step 4: Online Learning

Steps 2–3 used a two-stage approach: learn dynamics *offline* from expert data, then deploy with *fixed* parameters. But the deployed controller visits different states than the expert did, so the learned dynamics may be inaccurate in the regions that matter most. **Online learning** refines F and B_d from the controller's own experience.

We show two approaches that mirror the paper's two phases:

- **4a — Online dynamics refinement**: update F, B_d to minimize prediction error on transitions the controller actually generates. Simple, but optimizes for *accuracy*, not *control*.
- **4b — PPO through the MPC**: update F, B_d via policy gradient to maximize reward. This is Gnu-RL's key contribution — the dynamics converge to values that produce good *control*, even if they don't match the true system exactly.

21.8.4.1 Gradient flow check

Before doing online learning, verify that gradients actually flow through the MPC solver back to the dynamics parameters:

```
F_test = F_hat.detach().clone().requires_grad_(True)
Bd_test = Bd_hat.detach().clone().requires_grad_(True)

x_init = torch.tensor([[21.0]], dtype=torch.float64)
dt = torch.tensor(d[:, 0:T-1], dtype=torch.float64)
ft = torch.mm(Bd_test, dt).T.unsqueeze(1)

x_opt, u_opt, _ = MPC(
    n_state, n_ctrl, T,
    lqr_iter=20, verbose=-1,
    exit_unconverged=False,
)(x_init, QuadCost(C_mpc, c_mpc), LinDx(F_test.repeat(T-1, 1, 1, 1), ft))

target_loss = (x_opt[1, 0, 0] - 22.0)**2
target_loss.backward()

print(f"dL/dF  = {F_test.grad.numpy().round(4)}")
```

```
print(f"dL/dBd = {Bd_test.grad.numpy().round(4)}")
print("Non-zero gradients confirm: the MPC solver is differentiable!")
```

21.8.4.2 4a: Online dynamics refinement

The simplest approach: run the MPC controller for one episode, collect the transitions (x_t, u_t, d_t, x_{t+1}), then do a batch gradient step on the prediction error $\|\hat{x}_{t+1} - x_{t+1}\|^2$. Repeat for several episodes — each episode uses improved dynamics, visits better states, and generates better training data.

```
# Start from Step 2's learned dynamics
F_online  = F_hat.detach().clone().requires_grad_(True)
Bd_online = Bd_hat.detach().clone().requires_grad_(True)
opt_online = optim.Adam([F_online, Bd_online], lr=0.01)

n_episodes = 10
online_rewards = []

for ep in range(n_episodes):
    env_ep = ssModel(ssMatrix, x_init=18, x_target=22, eta=5)
    obs = env_ep.reset()

    # --- Rollout: control with current dynamics, collect data ---
    ep_states, ep_actions, ep_dist, ep_next = [], [], [], []
    ep_reward = []
    for t in range(n_steps - 1):
        x_t = obs[0]
        x_init = torch.tensor([[x_t]], dtype=torch.float64)
        dt = torch.tensor(d[:, t:t+min(T-1, n_steps-1-t)], dtype=torch.float64)
        if dt.shape[1] < T - 1:
            dt = torch.cat([dt, dt[:, -1:].repeat(1, T-1-dt.shape[1])], dim=1)
        ft = torch.mm(Bd_online.detach(), dt).T.unsqueeze(1)

        with torch.no_grad():
            _, u_opt, _ = MPC(n_state, n_ctrl, T,
                u_lower=0.0, u_upper=50.0, lqr_iter=20, verbose=-1,
                exit_unconverged=False,
            )(x_init, QuadCost(C_mpc, c_mpc),
              LinDx(F_online.detach().repeat(T-1, 1, 1, 1), ft))

        action = max(0, u_opt[0, 0, 0].item())
```

```
        ep_states.append(x_t)
        ep_actions.append(action)
        ep_dist.append(d[:, t])
        obs, reward = env_ep.step(np.array([action]))
        ep_next.append(obs[0])
        ep_reward.append(reward)

    # --- Batch update on collected episode data ---
    X = torch.tensor(np.column_stack([ep_states, ep_actions]), dtype=torch.float64)
    D = torch.tensor(np.array(ep_dist), dtype=torch.float64)
    Y = torch.tensor(ep_next, dtype=torch.float64)

    opt_online.zero_grad()
    pred = (X @ F_online.T + D @ Bd_online.T).squeeze()
    loss = ((pred - Y)**2).mean()
    loss.backward()
    opt_online.step()

    avg_r = np.mean(ep_reward)
    online_rewards.append(avg_r)
    print(f"Episode {ep+1}: avg_reward={avg_r:.2f}  pred_loss={loss.item():.4f}  "
          f"F={F_online.detach().numpy().round(4)}  "
          f"Bd={Bd_online.detach().numpy().round(4)}")
```

You should see both the prediction loss and average reward improve over episodes as the dynamics become more accurate in the regions the controller actually visits.

21.8.4.3 4b: PPO through the Differentiable MPC

The PPO approach is more powerful: instead of minimizing prediction error, it directly maximizes the environment reward by backpropagating policy gradients *through* the MPC solver. This is the core of Gnu-RL's Phase 2.

The key steps per episode:

1. **Rollout**: run the MPC, wrap its output u^* in a Gaussian $\mathcal{N}(u^*, \sigma)$ for exploration, record $(s, a, r, \log \pi_{old})$
2. **Advantages**: normalize rewards as a simple baseline
3. **PPO update**: re-run the MPC (with gradients), compute the new log-probability of each recorded action, and apply the clipped surrogate objective

```
# Start fresh from Step 2's learned dynamics
F_ppo  = F_hat.detach().clone().requires_grad_(True)
Bd_ppo = Bd_hat.detach().clone().requires_grad_(True)
opt_ppo = optim.Adam([F_ppo, Bd_ppo], lr=0.003)

n_episodes = 10
sigma_start, sigma_end = 0.5, 0.05  # exploration noise (decays)
eps_clip = 0.2
ppo_rewards = []

for ep in range(n_episodes):
    sigma = sigma_start - (sigma_start - sigma_end) * ep / max(n_episodes - 1, 1)
    env_ep = ssModel(ssMatrix, x_init=18, x_target=22, eta=5)
    obs = env_ep.reset()

    # --- Rollout with stochastic policy ---
    rollout = []  # list of (x_t, timestep, action, log_prob_old, reward)
    for t in range(n_steps - 1):
        x_t = obs[0]
        x_init = torch.tensor([[x_t]], dtype=torch.float64)
        dt = torch.tensor(d[:, t:t+min(T-1, n_steps-1-t)], dtype=torch.float64)
        if dt.shape[1] < T - 1:
            dt = torch.cat([dt, dt[:, -1:].repeat(1, T-1-dt.shape[1])], dim=1)
        ft = torch.mm(Bd_ppo.detach(), dt).T.unsqueeze(1)

        with torch.no_grad():
            _, u_opt, _ = MPC(n_state, n_ctrl, T,
                u_lower=0.0, u_upper=50.0, lqr_iter=20, verbose=-1,
                exit_unconverged=False,
            )(x_init, QuadCost(C_mpc, c_mpc),
              LinDx(F_ppo.detach().repeat(T-1, 1, 1, 1), ft))

        mu = u_opt[0, 0, 0].item()
        dist = torch.distributions.Normal(mu, sigma)
        a_sampled = dist.sample()
        action = max(0.0, a_sampled.item())
        log_prob_old = dist.log_prob(torch.tensor(action)).item()

        obs, reward = env_ep.step(np.array([action]))
        rollout.append((x_t, t, action, log_prob_old, reward))

    rewards_arr = np.array([r[4] for r in rollout])
```

```
    advantages = (rewards_arr - rewards_arr.mean()) / (rewards_arr.std() + 1e-8)

    # --- PPO update: re-run MPC with gradients ---
    ppo_loss = torch.tensor(0.0, dtype=torch.float64)
    for idx, (x_t, t, action, lp_old, _) in enumerate(rollout):
        x_init = torch.tensor([[x_t]], dtype=torch.float64)
        dt = torch.tensor(d[:, t:t+min(T-1, n_steps-1-t)], dtype=torch.float64)
        if dt.shape[1] < T - 1:
            dt = torch.cat([dt, dt[:, -1:].repeat(1, T-1-dt.shape[1])], dim=1)
        ft = torch.mm(Bd_ppo, dt).T.unsqueeze(1)  # gradients flow here

        _, u_opt, _ = MPC(n_state, n_ctrl, T,
            u_lower=0.0, u_upper=50.0, lqr_iter=20, verbose=-1,
            exit_unconverged=False,
        )(x_init, QuadCost(C_mpc, c_mpc),
          LinDx(F_ppo.repeat(T-1, 1, 1, 1), ft))  # and here

        mu_new = u_opt[0, 0, 0]
        dist_new = torch.distributions.Normal(mu_new, sigma)
        log_prob_new = dist_new.log_prob(torch.tensor(action))

        ratio = torch.exp(log_prob_new - lp_old)
        adv = torch.tensor(advantages[idx], dtype=torch.float64)
        surr1 = ratio * adv
        surr2 = torch.clamp(ratio, 1 - eps_clip, 1 + eps_clip) * adv
        ppo_loss = ppo_loss - torch.min(surr1, surr2)

    ppo_loss = ppo_loss / len(rollout)
    opt_ppo.zero_grad()
    ppo_loss.backward()
    opt_ppo.step()

    avg_r = rewards_arr.mean()
    ppo_rewards.append(avg_r)
    print(f"Episode {ep+1:2d}/{n_episodes}  sigma={sigma:.2f}  "
          f"avg_reward={avg_r:.2f}  "
          f"F={F_ppo.detach().numpy().round(4)}  "
          f"Bd={Bd_ppo.detach().numpy().round(4)}")
```

```
fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(range(1, len(online_rewards)+1), online_rewards, 'o-', label="Online dynamics (4a)")
ax.plot(range(1, len(ppo_rewards)+1), ppo_rewards, 's-', label="PPO (4b)")
```

```
ax.set_xlabel("Episode")
ax.set_ylabel("Average Reward")
ax.set_title("Online Learning: Prediction-Based vs. Reward-Based")
ax.legend()
plt.tight_layout()
plt.show()
```

> **Note**
>
> Both approaches improve the controller, but they optimize for different things. The dynamics refinement (4a) makes the model more *accurate* at predicting states — which indirectly improves control. PPO (4b) makes the model produce better *actions* — the dynamics may not match the true system exactly, but they converge to values that maximize reward. This is the paper's central claim: optimizing dynamics for control performance (through the differentiable MPC) outperforms pure system identification.

21.9 Gnu-RL on BOPTEST

The `ssM_env` example above uses a toy linear system where we know the true dynamics. Before jumping to SustainDC (a data center), let's apply Gnu-RL to a **realistic building** via the BOPTEST framework you used in Lecture 9. This is a natural stepping stone: BOPTEST is a building environment with continuous actions — exactly what Gnu-RL was designed for.

21.9.1 The Mapping

For the `bestest_hydronic_heat_pump` test case:

Gnu-RL concept	BOPTEST equivalent	Variable
State x	Zone temperature	`reaTZon_y` (in K)
Action u	Heat pump modulation	`oveHeaPumY_u` ($\in [0, 1]$)
Disturbance d	Outdoor temp, solar radiation	`TDryBul`, `HGloHor` via `/forecast`
Target	Heating setpoint	`reaTSetHea_y` (in K)
Step size	15 min (900 s)	Same as Gnu-RL's original setup

21.9.2 Step 1: Collect Expert Data from BOPTEST

We run a simple proportional controller for 2 days and record (x, u, d) tuples — the same data collection step as `agent/simulate.py` in Gnu-RL:

```
import requests
import numpy as np
import torch

url = 'http://13.217.71.86'
testcase = 'bestest_hydronic_heat_pump'

# Select and initialize
testid = requests.post(f'{url}/testcases/{testcase}/select').json()['testid']
requests.put(f'{url}/initialize/{testid}', json={'start_time': 0, 'warmup_period': 0})
requests.put(f'{url}/step/{testid}', json={'step': 900})  # 15-min steps

# --- Collect expert data with a proportional controller ---
K_p = 2.0
T_set_C = 21.0
T_set_K = 273.15 + T_set_C
n_steps = 2 * 96  # 2 days

states, actions, disturbances = [], [], []
y = requests.post(f'{url}/advance/{testid}', json={}).json()['payload']

for t in range(n_steps):
    T_zone = y['reaTZon_y']
    T_outdoor = y['weaSta_reaWeaTDryBul_y']
    solar = y['weaSta_reaWeaHGloHor_y']

    # Proportional controller (expert)
    error = T_set_K - T_zone
    u_heat = max(0.0, min(K_p * error, 1.0))

    states.append(T_zone)
    actions.append(u_heat)
    disturbances.append([T_outdoor, solar])

    y = requests.post(f'{url}/advance/{testid}', json={
        'oveHeaPumY_u': u_heat,
        'oveHeaPumY_activate': 1
    }).json()['payload']
```

```
states.append(y['reaTZon_y'])  # final state

states = np.array(states)
actions = np.array(actions)
disturbances = np.array(disturbances)  # [n_steps, 2]
print(f"Collected {len(actions)} transitions, temp range: "
      f"{(states.min()-273.15):.1f}--{(states.max()-273.15):.1f} °C")
```

21.9.3 Step 2: Learn Dynamics from Expert Data

Same approach as the `ssM_env` example — learn F and B_d by minimizing one-step state prediction error on the collected data:

```
from diff_mpc.mpc import MPC, QuadCost, LinDx

n_state, n_ctrl, n_dist = 1, 1, 2
T = 5  # planning horizon

# Normalize disturbances for numerical stability
d_mean = disturbances.mean(axis=0)
d_std  = disturbances.std(axis=0) + 1e-8
d_norm = (disturbances - d_mean) / d_std

# Vectorize training data
X = torch.tensor(np.column_stack([states[:-1], actions]), dtype=torch.float64)  # [N, 2]
D = torch.tensor(d_norm, dtype=torch.float64)                                   # [N, 2]
Y = torch.tensor(states[1:], dtype=torch.float64)                               # [N]

# Learnable parameters (initialized wrong)
F_hat  = torch.tensor([[0.5, 0.3]], dtype=torch.float64, requires_grad=True)
Bd_hat = torch.tensor([[0.1, 0.1]], dtype=torch.float64, requires_grad=True)
optimizer = torch.optim.Adam([F_hat, Bd_hat], lr=0.05)

losses = []
for epoch in range(300):
    optimizer.zero_grad()
    pred = (X @ F_hat.T + D @ Bd_hat.T).squeeze()
    loss = ((pred - Y)**2).mean()
    loss.backward()
    optimizer.step()
    losses.append(loss.item())
```

```
    if (epoch + 1) % 100 == 0:
        print(f"Epoch {epoch+1:3d}  loss={losses[-1]:.6f}  "
              f"F={F_hat.detach().numpy().round(4)}  "
              f"Bd={Bd_hat.detach().numpy().round(4)}")
```

21.9.4 Step 3: Deploy the Learned Controller

After training, use the learned dynamics to run the MPC as a controller on a fresh BOPTEST episode:

```
import matplotlib.pyplot as plt

# Cost matrices for MPC
C_mpc = torch.zeros(T, 1, n_state + n_ctrl, n_state + n_ctrl, dtype=torch.float64)
C_mpc[:, :, 0, 0] = 1.0    # comfort weight
C_mpc[:, :, 1, 1] = 0.01   # energy weight
c_mpc = torch.zeros(T, 1, n_state + n_ctrl, dtype=torch.float64)
c_mpc[:, :, 0] = -T_set_K  # target setpoint

# Fresh test instance
testid2 = requests.post(f'{url}/testcases/{testcase}/select').json()['testid']
requests.put(f'{url}/initialize/{testid2}', json={'start_time': 0, 'warmup_period': 0})
requests.put(f'{url}/step/{testid2}', json={'step': 900})

mpc_temps, mpc_actions = [], []
y = requests.post(f'{url}/advance/{testid2}', json={}).json()['payload']

for t in range(96):  # 1 day
    T_zone = y['reaTZon_y']

    # Get disturbance forecast for the planning horizon
    fc = requests.put(f'{url}/forecast/{testid2}', json={
        'point_names': ['TDryBul', 'HGloHor'],
        'horizon': (T - 1) * 900,
        'interval': 900
    }).json()['payload']
    d_fc = np.column_stack([fc['TDryBul'], fc['HGloHor']])  # forecast returns lists
    if len(d_fc) < T - 1:
        d_fc = np.pad(d_fc, ((0, T - 1 - len(d_fc)), (0, 0)), 'edge')
    d_fc_norm = (d_fc[:T-1] - d_mean) / d_std
```

```
    # Run MPC with learned dynamics
    x_init = torch.tensor([[T_zone]], dtype=torch.float64)
    dt = torch.tensor(d_fc_norm.T, dtype=torch.float64)
    ft = torch.mm(Bd_hat.detach(), dt).T.unsqueeze(1)

    with torch.no_grad():
        x_opt, u_opt, _ = MPC(
            n_state, n_ctrl, T,
            u_lower=0.0, u_upper=1.0,
            lqr_iter=20, verbose=-1,
            exit_unconverged=False,
        )(x_init, QuadCost(C_mpc, c_mpc),
          LinDx(F_hat.detach().repeat(T-1, 1, 1, 1), ft))

    u_mpc = u_opt[0, 0, 0].item()
    mpc_temps.append(T_zone - 273.15)
    mpc_actions.append(u_mpc)

    y = requests.post(f'{url}/advance/{testid2}', json={
        'oveHeaPumY_u': u_mpc,
        'oveHeaPumY_activate': 1
    }).json()['payload']

mpc_temps.append(y['reaTZon_y'] - 273.15)
kpis = requests.get(f'{url}/kpi/{testid2}').json()['payload']

# Plot
hours = np.arange(len(mpc_temps)) / 4
fig, axes = plt.subplots(2, 1, figsize=(14, 5), sharex=True)
axes[0].plot(hours, mpc_temps, label="Gnu-RL (learned MPC)")
axes[0].axhline(y=T_set_C, color='k', linestyle='--', label="Setpoint")
axes[0].set_ylabel("Zone Temp (°C)")
axes[0].legend()

axes[1].step(hours[:-1], mpc_actions, where='post')
axes[1].set_ylabel("Heat Pump Signal")
axes[1].set_xlabel("Hours")
plt.tight_layout()
plt.show()

print(f"Energy: {kpis['ener_tot']:.2f} kWh, Discomfort: {kpis['tdis_tot']:.4f} Kh")
```

21.9.5 Step 4: Online PPO Learning

Step 3 deployed the MPC with *fixed* dynamics. Now we close the loop: run a single continuous simulation over several days, and at the end of each day use that day's trajectory to update F and B_d via PPO. This mirrors how Gnu-RL would operate in a real building — the controller runs continuously and refines its model overnight.

Each day has two phases:

1. **Rollout** (96 steps, interacts with BOPTEST): run the MPC, add Gaussian noise for exploration, record $(x_t, d_t, a_t, \log \pi_{old}, r_t)$
2. **PPO update** (local only, no API calls): re-run the MPC with gradients on the stored inputs, compute the clipped surrogate loss, backpropagate through the MPC to update F and B_d

```
import torch.optim as optim

F_ppo  = F_hat.detach().clone().requires_grad_(True)
Bd_ppo = Bd_hat.detach().clone().requires_grad_(True)
opt_ppo = optim.Adam([F_ppo, Bd_ppo], lr=0.001)

n_days = 5
sigma_start, sigma_end = 0.3, 0.05  # exploration noise (decays)
eps_clip = 0.2

def boptest_reward(T_zone_K, u):
    """Reward: penalize deviation from setpoint + energy use."""
    return -(5.0 * (T_zone_K - T_set_K)**2 + u**2)

# Single continuous simulation (no reset between days)
tid = requests.post(f'{url}/testcases/{testcase}/select').json()['testid']
requests.put(f'{url}/initialize/{tid}', json={'start_time': 0, 'warmup_period': 0})
requests.put(f'{url}/step/{tid}', json={'step': 900})
y = requests.post(f'{url}/advance/{tid}', json={}).json()['payload']

ppo_rewards = []
for day in range(n_days):
    sigma = sigma_start - (sigma_start - sigma_end) * day / max(n_days - 1, 1)

    # --- Phase 1: Rollout for one day (continues from where we left off) ---
    rollout = []
    for t in range(96):
        T_zone = y['reaTZon_y']
```

```
        fc = requests.put(f'{url}/forecast/{tid}', json={
            'point_names': ['TDryBul', 'HGloHor'],
            'horizon': (T - 1) * 900, 'interval': 900
        }).json()['payload']
        d_fc = np.column_stack([fc['TDryBul'], fc['HGloHor']])
        if len(d_fc) < T - 1:
            d_fc = np.pad(d_fc, ((0, T - 1 - len(d_fc)), (0, 0)), 'edge')
        d_fc_norm = (d_fc[:T-1] - d_mean) / d_std

        # MPC action (no grad during rollout)
        x_init = torch.tensor([[T_zone]], dtype=torch.float64)
        dt = torch.tensor(d_fc_norm.T, dtype=torch.float64)
        ft = torch.mm(Bd_ppo.detach(), dt).T.unsqueeze(1)
        with torch.no_grad():
            _, u_opt, _ = MPC(n_state, n_ctrl, T,
                u_lower=0.0, u_upper=1.0, lqr_iter=20, verbose=-1,
                exit_unconverged=False,
            )(x_init, QuadCost(C_mpc, c_mpc),
              LinDx(F_ppo.detach().repeat(T-1, 1, 1, 1), ft))

        # Stochastic policy: Gaussian around MPC output
        mu = u_opt[0, 0, 0].item()
        dist = torch.distributions.Normal(mu, sigma)
        action = float(max(0.0, min(1.0, dist.sample().item())))
        log_prob_old = dist.log_prob(torch.tensor(action)).item()

        y = requests.post(f'{url}/advance/{tid}', json={
            'oveHeaPumY_u': action, 'oveHeaPumY_activate': 1
        }).json()['payload']

        rollout.append((T_zone, d_fc_norm.copy(), action, log_prob_old,
                        boptest_reward(T_zone, action)))

rewards_arr = np.array([r[4] for r in rollout])
advantages = (rewards_arr - rewards_arr.mean()) / (rewards_arr.std() + 1e-8)

# --- Phase 2: PPO update (local only, no API calls) ---
ppo_loss = torch.tensor(0.0, dtype=torch.float64)
for idx, (T_zone, d_fc_n, action, lp_old, _) in enumerate(rollout):
    x_init = torch.tensor([[T_zone]], dtype=torch.float64)
    dt = torch.tensor(d_fc_n.T, dtype=torch.float64)
    ft = torch.mm(Bd_ppo, dt).T.unsqueeze(1)        # ← gradients flow
```

```
        _, u_opt, _ = MPC(n_state, n_ctrl, T,
            u_lower=0.0, u_upper=1.0, lqr_iter=20, verbose=-1,
            exit_unconverged=False,
        )(x_init, QuadCost(C_mpc, c_mpc),
          LinDx(F_ppo.repeat(T-1, 1, 1, 1), ft))        # ← gradients flow

        mu_new = u_opt[0, 0, 0]
        dist_new = torch.distributions.Normal(mu_new, sigma)
        log_prob_new = dist_new.log_prob(torch.tensor(action))

        ratio = torch.exp(log_prob_new - lp_old)
        adv = torch.tensor(advantages[idx], dtype=torch.float64)
        surr1 = ratio * adv
        surr2 = torch.clamp(ratio, 1 - eps_clip, 1 + eps_clip) * adv
        ppo_loss = ppo_loss - torch.min(surr1, surr2)

    ppo_loss = ppo_loss / len(rollout)
    opt_ppo.zero_grad()
    ppo_loss.backward()
    opt_ppo.step()

    ppo_rewards.append(rewards_arr.mean())
    print(f"Day {day+1}/{n_days}  sigma={sigma:.2f}  "
          f"avg_reward={rewards_arr.mean():.2f}  "
          f"F={F_ppo.detach().numpy().round(4)}  "
          f"Bd={Bd_ppo.detach().numpy().round(4)}")

kpis = requests.get(f'{url}/kpi/{tid}').json()['payload']
print(f"\nCumulative KPIs over {n_days} days: "
      f"energy={kpis['ener_tot']:.2f} kWh, discomfort={kpis['tdis_tot']:.4f} Kh")
```

```
fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(range(1, len(ppo_rewards) + 1), ppo_rewards, 's-')
ax.set_xlabel("Day")
ax.set_ylabel("Average Reward")
ax.set_title("BOPTEST PPO: Average Reward per Day")
ax.axhline(y=ppo_rewards[0], color='k', linestyle='--', alpha=0.4, label="Day 1 baseline")
ax.legend()
plt.tight_layout()
plt.show()
```

The simulation runs continuously — each day picks up where the previous day ended, so

the controller experiences changing weather conditions across days. The PPO update at the end of each day adjusts F and B_d to improve control for the conditions the agent actually encounters.

The progression is now: toy system (`ssM_env`) → realistic building (BOPTEST) → data center (SustainDC). The code structure is identical at each step — only the environment interface changes.

21.10 Connecting to Assignment 3

The environment interface mapping and the keep-vs-replace breakdown are already covered in sections above. This section addresses the two non-obvious design decisions you will face when adapting Gnu-RL for SustainDC.

21.10.1 State vs. Disturbance Decomposition

SustainDC's `Agent_DC` (cooling optimizer) observes 26+ dimensions. You need to decide which are **states** (x, what the controller affects), which are **disturbances** (d, exogenous inputs), and which to ignore:

Role	Candidate variables	Notes
State x	Room temperature	The variable you are trying to control — analogous to "Indoor Temp." in Gnu-RL
Disturbance d	Ambient temperature, IT power/workload, carbon intensity forecast	Things the cooling controller cannot change but needs to anticipate
Ignore (initially)	Battery SoC, workload scheduling variables	Belong to the other two agents; not relevant if you fix them to baselines

This gives you a small state-space model ($n_{state} = 1$, $n_{dist} = 3$–5) similar in size to Gnu-RL's original formulation. You can always add more state dimensions later if the model is too simple.

21.10.2 Continuous vs. Discrete Actions

Gnu-RL's MPC produces **continuous** control actions (e.g., supply air temperature in $[20, 65]°C$). SustainDC's `Agent_DC` uses a `Discrete(3)` action space: decrease / maintain / increase CRAH setpoint.

You have two options:

1. **Discretize the MPC output**: Run the MPC with continuous bounds, then map the result to the closest discrete action (e.g., $u^* >$ threshold $\rightarrow$ increase, $u^* < -$threshold $\rightarrow$ decrease, else maintain). Simple, but the discretization is not differentiable.
2. **Bypass the discrete wrapper**: Access SustainDC's underlying continuous setpoint directly (the discrete actions just increment/decrement an internal setpoint variable). This preserves differentiability but requires modifying how you interact with the environment.

Either approach is acceptable for the assignment. Document your choice and its trade-offs.

21.10.3 Checklist

Before starting Assignment 3, confirm you can:

- ☐ Run the `ssM_env` imitation learning example from this tutorial (Section 8) and see `F_hat` converge
- ☐ Instantiate a SustainDC environment with a simplified configuration (from the SustainDC tutorial)
- ☐ Run a baseline controller in SustainDC and collect (x, u, d) trajectories
- ☐ Define the shapes of `F_hat` and `Bd_hat` for your chosen state/disturbance decomposition
- ☐ Construct `C` and `c` matrices that encode your cost function

22 Modeling the Impact of Passive Ventilation Systems on Multi-Zone Thermal Dynamics

22.1 Overview

Citation

Onyejizu, J., Mishra, S., & Kar, K. 2024. Modeling the Impact of Passive Ventilation Systems on Multi-Zone Thermal Dynamics. In *The 11th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation (BuildSys '24)*, November 7–8, 2024, Hangzhou, China. ACM, New York, NY, USA. https://doi.org/10.1145/3671127.3698178

Sense, Plan, Act Mapping and Objective

Application domain: Multi-zone commercial building energy management with passive ventilation. Traditional HVAC-only models cannot capture the thermal effects of opening windows and doors, limiting the potential for energy-saving natural ventilation strategies. This paper provides a modeling framework that enables predictive control over both HVAC and passive ventilation elements.

- **Sense**: Zone temperatures $T_i(t)$, ambient temperature $T_\infty(t)$, HVAC power inputs, and wind/weather conditions. The model assumes these measurements are available for each zone. The paper does not innovate on the sensing side but requires standard BAS instrumentation.
- **Plan**: **This is the primary innovation.** The Locally-interactive Bilinear Flow (LiBF) model provides a computationally tractable representation of multi-zone thermal dynamics that includes the effect of passive ventilation elements (windows and doors). Because the model is differentiable and structured (bilinear in state and input), it is suitable for MPC-style optimization over window/door opening schedules—enabling predictive control that coordinates HVAC and natural ventilation.
- **Act**: Motorized windows and doors with continuously adjustable opening factors $\theta \in [0, 1]$. Each passive element can be set to any position between fully closed

($\theta = 0$) and fully open ($\theta = 1$), providing fine-grained control over inter-zone and zone-to-ambient airflow paths.

22.2 Review of the paper

22.2.1 Summary

The paper proposes the **Locally-interactive Bilinear Flow (LiBF) model**—an extension of standard linear RC thermal network models where thermal conductances K_{ij} become functions of passive element opening factors θ. This introduces *bilinearity*: the system dynamics depend on the product of state variables (temperatures) and control inputs (opening factors).

Key contributions:

1. **LiBF model formulation**: Extends the standard multi-zone RC thermal model by replacing fixed thermal conductances K_{ij} with opening-factor-dependent functions $K_{ij}(\theta_{ij})$. The "locally interactive" assumption means each passive element (window or door) only affects the thermal conductance between its two directly connected zones, keeping the model sparse and physically interpretable.
2. **Two-step parameter estimation**:
   - **Step 1**: For each fixed opening configuration, use Non-Negative Least Squares (NNLS) to identify the thermal parameters (K_{ij}, $K_{i\infty}$, C_i).
   - **Step 2**: Fit constrained 2nd-order polynomials $K(\theta) = a_0 + a_1\theta + a_2\theta^2$ to the parameter estimates across configurations, with monotonicity constraints ensuring that conductance increases with opening factor.
3. **Measurement efficiency**: Only $O(N+M)$ training configurations are needed to predict $O(NM)$ unseen configurations, where N is the number of zones and M is the number of passive elements. This is a significant reduction in the experimental burden.
4. **Validation**: On a 3-zone EnergyPlus commercial building model with 3 passive elements (2 windows, 1 door), the LiBF model achieves RMSE of 0.21–0.53°C on 5-day validation data after training on 30 days of data.

22.2.2 What do we know already?

This paper connects directly and deeply to the thermal modeling content from Lecture 5 and Lecture 6. The table below maps each key concept from the paper to its origin in our course material:

Paper Concept	Lecture Source	Connection
Thermal resistance R and conductance $K = 1/R$	Lecture 5: Fourier's law, thermal resistance networks	The paper's K_{ij} parameters are the same thermal conductances we derived from Fourier's law for conduction through walls
Thermal capacitance C_i per zone	Lecture 5: $C = \rho V c$, RC circuit analogy	Each zone's lumped thermal mass stores energy; the paper estimates C_i for each zone just as we did for single-zone models
Energy balance at each node (Kirchhoff's laws)	Lecture 6: KCL applied to thermal circuits	The LiBF governing equation is an energy balance at each zone node—heat flows in from adjacent zones, ambient, and HVAC, exactly as in our RC network analysis
Multi-node RC network → coupled ODEs	Lecture 6: lumped parameter models, finite difference discretization	The paper extends the multi-zone coupled ODE system from Lecture 6 by making some conductances state-dependent
State-space form $\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u}$	Lecture 6: state-space representation of thermal networks	When all opening factors are fixed, the LiBF model reduces exactly to the linear state-space form from Lecture 6
Infiltration/ventilation as parallel thermal resistance	Lecture 6: infiltration heat transfer paths	The paper's passive ventilation conductances $K_{ij}(\theta)$ generalize the fixed infiltration resistance $R_{inf} = 1/(\dot{m}c_p)$ from Lecture 6 to a controllable, opening-dependent path

The key extension beyond Lectures 5–6: In our lecture material, all thermal conductances K_{ij} are fixed constants determined by material properties and geometry. The paper's central innovation is making these conductances *functions* of controllable opening factors θ_{ij}, so:

$$C_i \dot{T}_i = \sum_{j \in \mathcal{N}(i)} K_{ij}(\theta_{ij})(T_j - T_i) + K_{i\infty}(\theta_{i\infty})(T_\infty - T_i) + Q_{HVAC,i}$$

This introduces **bilinearity**: the term $K_{ij}(\theta_{ij}) \cdot (T_j - T_i)$ is a product of a function of one input (θ) and a function of the state ($T_j - T_i$). The system is no longer linear in the classical sense, but it retains enough structure to be tractable for identification and control.

Things to learn more about

To fully understand this paper's contributions, students may need background on:

1. **Bilinear dynamical systems**: Systems where the dynamics include products of state variables and inputs, i.e., $\dot{x} = Ax + Bu + \sum_k N_k x \cdot u_k$. These are more general than linear systems but more structured than fully nonlinear ones, enabling specialized identification and control methods.

2. **Non-Negative Least Squares (NNLS)**: A constrained least squares method that enforces $x \geq 0$ on the solution. Physical thermal parameters (conductances, capacitances) must be non-negative, so NNLS naturally encodes this physical constraint during estimation.

3. **Constrained polynomial fitting with monotonicity guarantees**: Fitting polynomials $K(\theta) = a_0 + a_1\theta + a_2\theta^2$ subject to the constraint that K is monotonically non-decreasing in θ (opening a window more should not *decrease* thermal conductance). This requires constrained optimization during the curve fitting step.

4. **EnergyPlus simulation methodology and EMS scripting**: EnergyPlus is the reference building energy simulation tool used to generate training and validation data. The Energy Management System (EMS) scripting interface allows custom control logic (e.g., setting window opening factors on a schedule) within the simulation.

5. **Stack effect and wind-driven ventilation physics**: The physical mechanisms behind natural ventilation—buoyancy-driven flow (stack effect) due to indoor-outdoor temperature differences and wind-induced pressure differences across openings. These are the underlying phenomena that the opening-factor-dependent conductances $K(\theta)$ are capturing in a lumped fashion.

6. **Graph-based representations of multi-zone buildings**: The paper represents the building as a graph where nodes are zones and edges are thermal connections (walls, windows, doors). This representation enables systematic enumeration of heat flow paths and is the basis for the "locally interactive" assumption.

22.3 Methods

22.3.1 The LiBF Model

The governing equation for zone i in the LiBF model is:

$$C_i\dot{T}_i(t) = \sum_{j\in\mathcal{N}(i)} K_{ij}(\theta_{ij})[T_j(t) - T_i(t)] + K_{i\infty}(\theta_{i\infty})[T_\infty(t) - T_i(t)] + Q_{HVAC,i}(t)$$

where:

- C_i is the thermal capacitance of zone i
- $K_{ij}(\theta_{ij})$ is the thermal conductance between zones i and j, which depends on the opening factor $\theta_{ij} \in [0, 1]$ of any passive element (window or door) connecting them
- $K_{i\infty}(\theta_{i\infty})$ is the conductance between zone i and the ambient, similarly dependent on any exterior window's opening factor
- $T_\infty(t)$ is the ambient temperature
- $Q_{HVAC,i}(t)$ is the HVAC heat input to zone i

Connection to Lecture 6: When all opening factors θ are held constant, every $K_{ij}(\theta)$ becomes a fixed scalar, and the equation above reduces exactly to the standard linear multi-zone RC model:

$$C_i\dot{T}_i = \sum_j K_{ij}(T_j - T_i) + K_{i\infty}(T_\infty - T_i) + Q_{HVAC,i}$$

which can be written in state-space form $\dot{\mathbf{T}} = A\mathbf{T} + B\mathbf{u}$, as covered in Lecture 6.

The "locally interactive" assumption: Each passive element affects only the conductance between its two directly connected zones. A window between zones 1 and 2 changes $K_{12}(\theta_{12})$ but does not affect K_{13} or K_{23}. This keeps the number of parameters linear in the number of passive elements rather than combinatorial.

22.3.2 Two-Step Parameter Estimation

Step 1: NNLS for fixed configurations

For each training configuration (a specific set of fixed opening factors $\theta^{(k)}$), the continuous-time ODE is discretized and the parameters (K_{ij}, $K_{i\infty}$, C_i) are estimated using Non-Negative Least Squares:

$$\min_{\mathbf{p}\geq 0} \|\Phi\mathbf{p} - \mathbf{b}\|_2^2$$

where $\mathbf{p}$ is the vector of thermal parameters and Φ is the regression matrix constructed from temperature measurements. The non-negativity constraint $\mathbf{p} \geq 0$ ensures physical plausibility (no negative conductances or capacitances).

Step 2: Constrained polynomial fit

Once parameters are estimated for multiple fixed configurations, the conductance K_{ij} as a function of θ_{ij} is fit with a constrained 2nd-order polynomial:

$$K_{ij}(\theta) = a_0 + a_1\theta + a_2\theta^2$$

subject to the monotonicity constraint:

$$\frac{dK_{ij}}{d\theta} = a_1 + 2a_2\theta \geq 0, \quad \forall\theta \in [0, 1]$$

This ensures that increasing the opening factor never decreases the thermal conductance—a physically sensible requirement.

Measurement efficiency: The key insight is that training configurations can be chosen to vary one passive element at a time (plus a few joint configurations). This requires only $O(N + M)$ experiments to predict the behavior of all $O(NM)$ possible configurations.

22.3.3 Experimental Validation

Setup:

- 3-zone EnergyPlus model of a commercial building
- 3 passive elements: 2 exterior windows (zones 1 and 3) and 1 interior door (between zones 1 and 2)
- Opening factors discretized to 6 levels: $\theta \in \{0, 0.2, 0.4, 0.6, 0.8, 1.0\}$
- 30-day training period, 5-day validation period

Results:

- RMSE across zones: 0.21–0.53°C on validation data
- The model successfully predicts temperature dynamics for opening factor combinations not seen during training
- Sensitivity analysis shows that exterior windows (DOF/WOF) have a larger impact on zone temperatures than interior doors, consistent with the larger temperature differential between indoor and ambient vs. between adjacent zones

23 Can Attention Improve Sequence-to-Point Load Disaggregation?

23.1 Overview

Citation

Bouchur, M., Li, N., & Reinhardt, A. 2025. Can Attention Improve Sequence-to-Point Load Disaggregation? A Comparative Assessment. In *The 12th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation (BuildSys '25)*, November 19–21, 2025, Golden, CO, USA. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3736425.3772099

Sense, Plan, Act Mapping and Objective

Application domain: Residential building energy management—specifically disaggregating per-appliance power consumption from a single whole-house meter (Non-Intrusive Load Monitoring, or NILM).

- **Sense**: **This is, perhaps, the primary innovation.** The paper improves the *sensing (and perception)* pipeline by adding attention modules to the Sequence-to-Point (S2P) neural network used in processing power measurements from a building, enabling better extraction of appliance-specific power signatures from the aggregate mains signal. The input is aggregate active power measured at a single point (the mains); the output is appliance-level power estimates. One could consider this a **plan/learn** process, but that's just a matter of taste.
- **Plan**: Not a focus. Once appliance-level consumption is known, planning (e.g., demand response scheduling, energy audits) is enabled but not addressed here.
- **Act**: Not addressed. No actuation or control loop is proposed.

23.2 Review of the paper

23.2.1 Summary

The paper addresses Non-Intrusive Load Monitoring (NILM) using the **Sequence-to-Point (S2P)** model—a CNN that takes a window of aggregate power readings and predicts the midpoint power of a single target appliance. Its central contribution is a **systematic comparison of seven attention-based extensions** from three families, inserted between the last convolutional layer and the dense head of the S2P architecture.

Key contributions and findings:

1. **Seven attention variants from three families**: Channel Attention (CA), Feed-Forward Attention (FFA), and Self-Attention (SA), each with 1–3 sub-variants.
2. **Context-dependent performance**: The best attention variant depends on both **appliance type** and **dataset**—there is no one-size-fits-all winner.
3. **Appliance-specific findings on UK-DALE**:
    - For **transient, high-power loads** (microwave, kettle): SA+MLP+PE (transformer-style self-attention with positional encoding) works best, achieving -15% MAE on average and -45% on microwave.
    - For **steady/periodic loads** (fridge): SA+MLP or CA variants perform better.
4. **Dataset-specific findings**: On REDD, channel attention (CA+SpA) generalizes best on average (-11% MAE).
5. **Cross-dataset transfer**: FFA generalizes best UK-DALE→REDD; CA+SpA best REDD→UK-DALE.
6. **Resource tradeoffs**: SA variants require $\sim 2\times$ GFLOPs and $\sim 4\times$ training time; CA adds only $\sim 4\%$ overhead.
7. **Practical recommendation**: Attention selection should be treated as a **domain-sensitive hyperparameter**, not a universal improvement.

23.2.2 What do we know already?

This paper connects directly to the AC power systems content from Lecture 8, particularly the sections on power measurement, VI trajectories, and load identification. The table below maps each key concept from the paper to its origin in our course material:

Paper Concept	Lecture 8 Source	Connection
Aggregate power signal (mains)	Instantaneous power: $p[n] = v[n] \cdot i[n]$	The S2P model's input is exactly the aggregate active power time series that you learned to compute from raw V/I samples
Per-appliance power signatures	VI trajectories as load fingerprints	The paper's goal (disaggregation) is the ML generalization of the VI-based load identification concept from Lecture 8
Appliance characteristics (transient vs. steady-state)	RLC transient behavior and load types	The paper's finding that attention type depends on appliance dynamics (sharp transients vs. long plateaus) maps directly to the resistive vs. reactive vs. non-linear load taxonomy
Sampling rate and window size	Sub-cycle vs. cycle-level measurement	The paper uses 6s sampling / 599-sample windows (~ 1 hour); Lecture 8 discussed tradeoffs between temporal resolution and noise averaging
NILM concept	VI trajectory section on load identification	Lecture 8 introduced NILM as a motivation for understanding VI trajectories; this paper is a state-of-the-art NILM method

Key extension beyond Lecture 8: Lecture 8 introduced NILM via physics-based VI trajectories—single-cycle fingerprints compared via pattern matching. This paper shows how deep learning (specifically attention-augmented CNNs) can learn to disaggregate directly from **low-rate active power time series** *without* sub-cycle V/I waveforms, capturing temporal patterns across minutes rather than within a single 16 ms cycle. Where VI trajectories require high-frequency sampling (kHz), S2P operates on 6-second intervals and relies on learned temporal features rather than physics-derived signatures.

Things to learn more about

To fully understand this paper's contributions, you may need background on:

1. **The Transformer architecture and attention mechanisms**—the core ML concept underlying the SA variants (covered in Section 23.3.2 below)
2. **Convolutional Neural Networks (CNNs) for time series**—the baseline S2P model uses 1D convolutions to extract features from the aggregate power sequence
3. **Sequence-to-Point (S2P) learning**—the specific framing where a window of input maps to a single midpoint output value
4. **Squeeze-and-Excitation (SE) networks**—the basis for Channel Attention, where convolutional filter outputs are reweighted by learned importance scores
5. **Positional encoding**—how sequence order information is injected into attention models that are otherwise permutation-invariant
6. **UK-DALE and REDD datasets**—standard NILM benchmarks used for evaluation

23.3 Methods

23.3.1 The Sequence-to-Point (S2P) Baseline

The S2P model (Zhang et al. 2018) is a CNN designed specifically for NILM. Its architecture is straightforward:

1. **Input**: A window of $W = 599$ aggregate power samples at 6-second resolution ($\sim$ 1 hour of data)
2. **Feature extraction**: 5 successive 1D convolutional layers with increasing filter counts (30, 30, 40, 50, 50) and kernel size 10
3. **Prediction head**: Flatten $\rightarrow$ Dense(1024) $\rightarrow$ Dense(1) $\rightarrow$ single scalar output
4. **Output**: The predicted power consumption of a single target appliance at the **midpoint** of the input window

Connection to Lecture 8: The input sequence is exactly the active power time series $p[n]$ from the power measurement section—except measured at the whole-house mains rather than at individual appliances, and sampled at low rate (6s) rather than sub-cycle.

A separate S2P model is trained for each target appliance. The convolutional layers learn to detect temporal patterns in the aggregate signal that correspond to the target appliance's operation—effectively learning a data-driven version of the "load fingerprinting" concept from Lecture 8's VI trajectory discussion.

23.3.2 Attention Mechanisms: Intuition

The core idea behind attention is simple: **not all parts of the input are equally important for the prediction**. Attention lets the model learn *where to focus*.

Consider disaggregating a microwave from the aggregate signal. A microwave has a distinctive sharp on/off pattern—a sudden jump to $\sim$ 1200W followed by a plateau and then a sharp drop. The S2P model should focus on these transitions and ignore the background steady-state. Without attention, the dense layers must learn this selectivity implicitly from the flat feature vector. With attention, the model can explicitly learn to weight the relevant time steps (or feature channels) more heavily before the features reach the dense head.

This intuition maps directly to a finding in the paper: SA+MLP+PE (which can learn to attend to specific temporal positions) gives the largest improvement for **transient-dominated appliances** like the microwave (-45% MAE on UK-DALE).

23.3.3 The Transformer Architecture and Self-Attention

The Self-Attention (SA) variants in this paper are derived from the **Transformer** architecture (Vaswani et al. 2017), originally developed for machine translation and now the foundation of modern large language models. Understanding how self-attention works is key to understanding the paper's most powerful (and most expensive) attention family.

23.3.3.1 Query, Key, Value: The Core Mechanism

Self-attention operates on a sequence of input vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$. For each position i, it computes three vectors via learned linear projections:

- **Query** $\mathbf{q}_i = W_Q \mathbf{x}_i$: "What am I looking for?"
- **Key** $\mathbf{k}_j = W_K \mathbf{x}_j$: "What do I contain?"
- **Value** $\mathbf{v}_j = W_V \mathbf{x}_j$: "What information do I provide?"

The output for position i is a weighted sum of all value vectors, where the weights are determined by how well each key matches the query:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V$$

where:

- QK^T computes the dot product between every query-key pair (an $n \times n$ attention matrix)
- $\sqrt{d_k}$ is a scaling factor to prevent the dot products from becoming too large (which would push softmax into saturation)

- softmax normalizes each row so the attention weights sum to 1

Intuition for NILM: In the S2P context, each position in the sequence corresponds to a time step's feature representation after the convolutional layers. Self-attention allows every time step to "look at" every other time step and decide which ones are relevant. For a microwave disaggregation, the time steps near the on/off transitions would learn to attend to each other, reinforcing the sharp-edge pattern.

23.3.3.2 Multi-Head Attention

Rather than computing a single attention function, the Transformer runs **multiple attention heads in parallel**, each with its own W_Q, W_K, W_V projections:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W_O$$

where each $\text{head}_i = \text{Attention}(QW_Q^i, KW_K^i, VW_V^i)$.

Each head can learn to attend to different aspects of the input—one head might focus on sharp transients, another on periodic patterns, and another on baseline power levels.

23.3.3.3 Positional Encoding

Self-attention is **permutation-invariant**: shuffling the input sequence would produce the same attention weights (since it only compares vectors, not positions). For time-series data where ordering matters, this is a problem. Positional encoding solves it by adding position-dependent signals to the input:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right), \quad PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

These sinusoidal functions give each position a unique signature and allow the model to learn relative position relationships.

Connection to the paper: The SA+MLP+PE variant includes positional encoding, while SA and SA+MLP do not. The paper finds that PE provides a significant boost for **transient appliances** (microwave, kettle), which makes sense: knowing *when* in the window a sharp transition occurs is critical for identifying these appliances.

23.3.3.4 The Transformer Block

A complete Transformer encoder block combines these elements:

1. Multi-Head Self-Attention
2. Add & Layer Normalize (residual connection)
3. Feed-Forward Network (two dense layers with nonlinearity)
4. Add & Layer Normalize (residual connection)

Connection to the paper: The SA+MLP+PE variant (Fig. 2g in the paper) is essentially one Transformer encoder block applied to the S2P feature maps after the convolutional layers. It is the most powerful but also the most expensive variant ($\sim 2\times$ GFLOPs, $\sim 4\times$ training time compared to the baseline).

23.3.4 The Three Attention Families in the Paper

The paper evaluates seven attention variants organized into three families. All are inserted at the same point in the S2P architecture: **between the last convolutional layer and the dense head**.

To build intuition for how each family works, it helps to think about the **shape of the tensor** entering the attention module. After 5 convolutional layers, the original 599-sample input has been transformed into a 2D feature map with two axes:

- C **channels** ($= 50$, one per convolutional filter in the last layer): each channel captures a different learned pattern (e.g., sharp edges, slow ramps, periodic oscillations)
- T **time positions** (roughly the length of the input, reduced slightly by each conv layer): each position corresponds to a local region of the original power time series

So the attention input is a $(C \times T)$ matrix per sample—50 feature channels, each containing a time series of activations. The three attention families differ in *which axis* they attend to and *how* they compute importance weights.

23.3.4.1 Channel Attention (CA)

Based on **Squeeze-and-Excitation (SE) networks** (Hu et al. 2018). CA asks: **"Which of the C filters are most important?"** It collapses the time axis entirely and produces a per-channel weight:

1. **Squeeze** (global average pool over time): Each channel's T-long activation is averaged to a single scalar, reducing the $(C \times T)$ feature map to a C-dimensional vector—a compact "channel descriptor."

2. **Excitation** (MLP + sigmoid): A small bottleneck MLP ($C \to C/r \to C$ with reduction ratio r) learns inter-channel dependencies and outputs C importance weights in $[0, 1]$ via sigmoid.
3. **Scale** (broadcast multiply): Each channel's entire time series is multiplied by its scalar weight. Output shape is the same ($C \times T$), but channels have been reweighted.

The key property: CA is **blind to temporal position**. It can amplify a channel that captures "sharp on/off edges" and suppress one that captures "slow drift," but it cannot say "focus on time step 300." This is also why it's so cheap—the MLP operates on a 50-dimensional vector, not the full $C \times T$ tensor.

Two variants:

Variant	Description	Key Feature
CA	Standard SE block	Reweights filters globally
CA+SpA	SE + spatial (position-wise) attention	Also reweights time positions

CA+SpA adds a second stage that asks **"Which time positions matter?"** After the channel reweighting, it pools across channels (not time) at each position to get a T-dimensional descriptor, applies a 1D convolution, and produces a per-position weight in $[0, 1]$. The result is a tensor reweighted along *both* axes: important channels and important time positions are amplified.

Overhead: Minimal—only $\sim$ 4% additional parameters and GFLOPs. This makes CA attractive when computational budget is limited.

23.3.4.2 Feed-Forward Attention (FFA)

Based on **Bahdanau attention** (Bahdanau et al. 2015). Where CA attends along the channel axis, FFA attends along the **time axis**: it asks **"How important is each time position?"**

1. **Position-wise MLP**: A two-layer MLP is applied independently to each time position's C-dimensional feature vector, producing a single scalar score per position. Think of it as sliding the same small network across all T positions.
2. **tanh activation**: The scores pass through tanh, giving values in $[-1, 1]$ (so FFA can actually flip the sign of a position's contribution, not just suppress it).
3. **Reweight** (broadcast multiply): Each position's entire C-dimensional feature vector is scaled by its score. Output shape is again ($C \times T$).

The critical difference from CA: FFA can focus on "the moment the microwave turns on" (a specific time position), while CA cannot. However, each position is scored **independently**—FFA does not model interactions *between* positions. It can say "time step 300 is important" but not "time step 300 is important *because of what happened at* time step 250."

Two variants:

Variant	Description	Key Feature
FFA	MLP + tanh → position weights	Lightweight position weighting
FFA+LSTM	FFA preceded by bidirectional LSTM	Adds sequential memory before attention

The LSTM variant addresses FFA's independence limitation by first passing the features through a bidirectional LSTM, which injects sequential context before the attention scoring.

23.3.4.3 Self-Attention (SA)

Based on the **Transformer** architecture (Vaswani et al. 2017). SA fills the gap that FFA leaves: it computes **pairwise** interactions between all T time positions, producing a $T \times T$ attention matrix where entry (i, j) represents "how much should position i attend to position j." This means SA can learn relationships like "this time step matters *because of its relationship to* that other time step"—but at the cost of computing and storing that full $T \times T$ matrix.

Three variants:

Variant	Description	Key Feature
SA	Raw self-attention only	Minimal SA, no feed-forward
SA+MLP	SA + feed-forward network	Adds nonlinear transformation
SA+MLP+PE	SA + MLP + positional encoding	Full Transformer encoder block

SA is the most powerful family (it can model arbitrary position-to-position dependencies) but also the most expensive ($\sim 2\times$ GFLOPs, $\sim 4\times$ training time). The $T \times T$ attention matrix is the main source of this cost.

Summary of what each family can "see":

Family	Attends along	Can distinguish time positions?	Models position-to-position interactions?
CA	Channels	No	No
FFA	Time	Yes (independently)	No

Family	Attends along	Can distinguish time positions?	Models position-to-position interactions?
SA	Time	Yes (jointly)	Yes ($T \times T$ attention matrix)

23.3.5 Key Results Summary

The paper's main finding is that **no single attention variant dominates across all settings**. Performance depends on both the appliance and the dataset:

Best variants by appliance type (UK-DALE):

Appliance	Best Attention	MAE Change	Why It Works
Microwave	SA+MLP+PE	−45%	Sharp on/off transients benefit from positional and pairwise attention
Kettle	SA+MLP+PE	−22%	Similar transient pattern to microwave
Washing machine	CA+SpA	−15%	Multi-phase operation benefits from spatial + channel reweighting
Fridge	SA+MLP	−8%	Periodic steady-state pattern; PE not needed
Dish washer	FFA	−5%	Long, complex cycles; lightweight attention suffices

Cross-dataset generalization:

Transfer Direction	Best Attention	Average MAE Change
UK-DALE → REDD	FFA	Best generalization
REDD → UK-DALE	CA+SpA	Best generalization
Within REDD	CA+SpA	−11% average

Resource tradeoffs:

Family	Additional GFLOPs	Training Time Increase	Parameters Added
CA	$\sim 4\%$	Minimal	$\sim 4\%$
FFA	$\sim 10\%$	$\sim 1.5\times$	$\sim 8\%$

Family	Additional GFLOPs	Training Time Increase	Parameters Added
SA	$\sim 100\%(2\times)$	$\sim 4\times$	~ 30 to 50%

Practical takeaway: Attention type should be treated as a **domain-sensitive hyperparameter**. If computational budget is limited, CA variants offer the best cost-performance ratio. If maximum accuracy on transient appliances is the goal, SA+MLP+PE is worth the compute cost.

23.4 Additional Resources

For those of you who want to go deeper into the ML concepts underlying this paper:

- Vaswani, A., Shazeer, N., Parmar, N., et al. 2017. "Attention Is All You Need." *Advances in Neural Information Processing Systems (NeurIPS)*. (The original Transformer paper)
- Bahdanau, D., Cho, K., & Bengio, Y. 2015. "Neural Machine Translation by Jointly Learning to Align and Translate." *Proceedings of the International Conference on Learning Representations (ICLR)*. (Bahdanau attention / FFA origin)
- Hu, J., Shen, L., & Sun, G. 2018. "Squeeze-and-Excitation Networks." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (SE / Channel Attention origin)
- Chaudhari, S., Mithal, V., Polatkan, G., & Ramanath, R. 2021. "An Attentive Survey of Attention Models." *ACM Transactions on Intelligent Systems and Technology (TIST)*. (Comprehensive survey of attention mechanisms)
- Zhang, C., Zhong, M., Wang, Z., Goddard, N., & Sutton, C. 2018. "Sequence-to-Point Learning with Neural Networks for Non-Intrusive Load Monitoring." *Proceedings of the AAAI Conference on Artificial Intelligence*. (The original S2P paper)
- 3Blue1Brown. "But what is a GPT? Visual intro to Transformers." Deep Learning Chapter 5. See also Chapter 6: "Attention in Transformers, step-by-step". (Excellent visual explanations of the Transformer architecture for visual learners)

Part V

Implementation

24 Building a Novel Smart Electrical Meter

This document describes the motivation, objectives and initial ideas for Project #2 of the Spring 2023 edition of the course. The projects are the cornerstone of the course, as they are what motivates all of the material that is discussed during the semester.

The project's goal is to develop a computer system (hardware and software) that can monitor the electrical power usage of a section of the electrical power distribution system inside a building (e.g., at the main feed, sub-panel or individual circuit) and to leverage this information to support specific decisions by the building occupants or managers. These decisions need to be identified ahead of time by the project team, and can range from simple ones such as identifying base-load power consumption values, to more complex ones such as suggesting actionable steps to improve power quality or provide estimates of future power consumption under different scenarios.

There are various examples of projects (and commercially avaiable products) that can be used as a starting point. If you are curious to know more about them, a good place to start would be this (growing) list of links:

- EmonPi
- Split Single-Phase Energy Meter
  - Using the Split Single-Phase Energy Meter with RPi

25 Building a Novel Smart Thermostat

This document describes the motivation, objectives and initial ideas for Project #1 of the Spring 2023 edition of the course. The projects are the cornerstone of the course, as they are what motivates all of the material that is discussed during the semester.

The project's goal is to develop a computer system (hardware and software) that can interact with heating, ventilation and/or air conditioning (HVAC) equipment in a residential or commercial building, in order to enact "smarter" control of these, where "smart" here means that the new controller provides some cost, energy and/or productivity gains over the older one. One of the lowest-level controllers in modern HVAC systems consists of a (digital) thermostat and a Proportional Integral Derivative (PID) controller. Thus, the goal of this project is to create a computer system that can be used to replace an existing digital thermostat and increase the efficiency (energy, cost or productivity) of the building.

There are various examples of projects (and commercially avaiable products) that do just this. If you are curious to know more about them, a good place to start would be this (growing) list of links:

- ThermOS, with code here.
- DIY Thermostat with a Raspberry Pi
- Raspberry Pi as a touchscreen thermostat
- Building a Thermostat with the Raspberry Pi
- Programmable Thermostat with the Raspberry Pi
- PiTherm
- Raspberry Pi as a thermostat
- ThermTerm

26 Final Projects

Below are links to the websites for past years' final projects.

26.1 Spring 2026

- Self-Adaptive Occupancy Forecasting in CMU Classrooms (and Beyond!)
- Thermal Digital Twin
- CEE-12770-Team5
- Disag-With-Light

References

Balaji, Bharathan, Arka Bhattacharya, Gabriel Fierro, et al. 2016. "Brick: Towards a Unified Metadata Schema for Buildings." *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments*, 41–50.

Chen, Bingqing, Zicheng Cai, and Mario Bergés. 2019. "Gnu-Rl: A Precocial Reinforcement Learning Solution for Building Hvac Control Using a Differentiable Mpc Policy." *Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*, 316–25.

González-Torres, M., L. Pérez-Lombard, Juan F. Coronel, Ismael R. Maestre, and Da Yan. 2022. "A Review on Buildings Energy Information: Trends, End-Uses, Fuels and Drivers." *Energy Reports* 8: 626–37. https://doi.org/10.1016/j.egyr.2021.11.280.

Harvey, Danny. 2010. *Energy and the New Reality 1: Energy Efficiency and the Demand for Energy Services*. Routledge. https://doi.org/10.4324/9781849774918.

MacKay, David JC. 2008. *Sustainable Energy–Without the Hot Air*. UIT cambridge. https://www.repository.cam.ac.uk/handle/1810/217849.

Murphy Jr, Thomas W. 2021. *Energy and Human Ambitions on a Finite Planet*. https://escholarship.org/uc/item/9js5291m.

Pérez-Lombard, Luis, José Ortiz, and Christine Pout. 2008. "A Review on Buildings Energy Consumption Information." *Energy and Buildings* 40 (3): 394–98.

Reddy, T, Jan F Kreider, Peter S Curtiss, and Ari Rabl. 2016. *Heating and Cooling of Buildings: Principles and Practice of Energy Efficient Design*. CRC press.

Rowe, Anthony, Mario Berges, and Raj Rajkumar. 2010. "Contactless Sensing of Appliance State Transitions Through Variations in Electromagnetic Fields." *Proceedings of the 2nd ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Building*, 19–24.