Learning to Breathe

Physics v. ML-based Lung Simulations for Control of Medical Ventilators

Nimra Nadeem

2021

Advised by Professor Elad Hazan

Submitted to Princeton University

Department of Computer Science

This thesis represents my own work
in accordance with University Regulations

Date of Submission: April 30, 2021

To Anne Caswell-Klein,
I would not be here without you.

Acknowledgements

I want to begin by thanking the late Professor Steven Gubser, my first adviser at Princeton University, who supported and believed in me when there was very little reason to.

I sincerely thank Professor Elad Hazan, for introducing me to the wonderful world of Control Theory and advising me throughout this project. Thank you to Daniel Suo, for walking me through the entire code base and answering all my amateur questions and to Paula Gradu for her generous time and energy spent explaining basic control theory concepts to me.

My sincerest gratitude to Professor Tom Griffiths, who introduced me to the fascinating field of computational cognitive science which led to my interest in reinforcement learning. My deepest appreciation for Rachit Dubey, who committed so much of his time and effort into explaining the theoretical foundations of RL and genuinely getting me excited about its applications.

I extend my warmest appreciation to Amanda Irwin-Wilkins, for helping me see that the monster was real, that I was exhausted because I was fighting not because I was weak and lazy. I am also indebted to Anita McLean, for teaching me how to fight, how to stand back up and how to keep going.

I cannot thank enough the friends who have been there for me even in the darkest of times. I am so sincerely grateful to Bianca, Sana, Shwe, Kayla, Alya, Colton, Anvay, Sarah, Tilmann, Saif, Furqan, Farhan, Alexis, Marco, Aya, Jana, Blessing, Gautham, Amina, Britt, Arya and Celia for their friendship, love and support through the past four years, I wouldn't have made it through without you guys.

Thank you to Qasim for saving me from my academic downfall, and helping me every step of the way. Thank you to Nayab, for making my life here so much more bearable, for holding me accountable and setting my priorities straight. Thank you to Ahmad Raza, for teaching me lung anatomy and physiology, and providing his insightful feedback to my respiratory mechanics section.

My love and gratitude to Mussa, Romi, Kaddu, Fafa and Shifay, for giving me a reason to keep fighting and dreaming.

I am extremely grateful for Mama, Baba, Nanajan and Nanijan, because of their unconditional love and faith in me, and because they taught me to care about the right things.

And finally, of course, my deepest, profoundest, immeasurable gratitude for Dean Anne Caswell-Klein, to whom I owe all and any of my academic achievements at Princeton University

Contents

Abstract

We compare two different physics-based lung simulations to an ML-based data-driven lung simulation, using two evaluation metrics. The first metric quantifies how well the simulation mimics the pressure state of a human lung. The second metric quantifies how well the simulation serves as a training ground for a PID controller targeting a specific pressure waveform. We find that the physics-based simulation performs worse than the data-driven simulation according to the first metric, but on one of the 9 lung settings, the physics based simulation performs better according to the second metric. This paper contributes to an ongoing project on ventilator control conducted by Princeton University and Google AI, aimed towards building more effective and robust mechanical ventilators for use in clinical settings.

1. Introduction

Invasive Mechanical Ventilation (IMV) is defined as delivering positive pressure to the lungs via an endotracheal or a tracheostomy tube attached to the ventilator. Early recognition of patients who might potentially require ventilatory support is a key goal of critical care outreach programs and an essential skill for all hospital medical staff. Acute hypercapnic respiratory failure due to underlying chronic obstructive pulmonary disease (COPD) and diffuse parenchymal lung disease like Acute Respiratory Distress Syndrome (ARDS) remain the most common indications of IMV. [12, 22]

Lung injury patterns typically vary from a mild pneumonia to full-blown ARDS, which is a severe and rapidly developing respiratory distress syndrome often requiring mechanical ventilatory support. Due to the recent COVID-19 viral pandemic, the incidence of ARDS and consequently the need for mechanical respiratory support has increased dramatically worldwide.

Individual studies report varying rates of ICU admission, frequency of ARDS, and mortality among patients. A meta-analysis of individual studies has indicated that among hospitalized COVID-19 patients, approximately 1/3 (33%) develop ARDS, 1/4 (26%) require transfer to an ICU, 1/6 (16%) receive IMV, and 1/6 (16%) die. [22] For COVID-19 patients transferred to an ICU, nearly 2/3 (63%) receive IMV, and 3/4 (75%) have ARDS. [22] Given the above estimates, the need for mechanical ventilators in managing these critically ill patients is rather obvious.

Despite the crucial role played by ventilators in critical care, the current industry standard for ventilator control has not seen serious improvement in years. [20] In the context of COVID-19, the shortage of ventilators and unavailability of medical staff to monitor ventilator settings in deteriorating patients has made many researchers turn their attention towards developing new, innovative methods of (a) developing low-cost ventilators [10, 11] and (b) improving ventilator control and making it more robust using adaptive control and machine learning techniques. [6, 18, 20]

The latter is the focus of recent work conducted by Suo et al. [20], a collaboration between the Hazan group at the Department of Computer Science at Princeton University and Google AI.

Their project consists of two components. The first, called *real2sim*, is a data-driven simulator of the human lung. The second, called *sim2real*, is a deep neural network controller that is learnt by training on the *real2sim* lung simulator. They show that the neural network controller tracks the target lung pressure waveform better than PID controllers, which are the current industry standard. Their motivation for the first component, i.e. the data-driven lung simulator, is that a physics-based simulation is too idealized and has an underspecification of variability as compared to a real-life human lung.

However, the physics-based model considered by Suo et al. [20] is a simplistic model of the human lungs, which treats the ventilator lung system as two balloons connected with a pipe. While it is certainly true that this *specific* physics-based model is idealized and does not capture variability in lung features, the same is not true of all physics-based lung models. In fact, extensive work has been done in the theory and validation of several physics-based lung models, with varying levels of complexity. [1, 3, 4, 7, 13, 14, 19]

The goal of this paper is to build more sophisticated physics-based lung simulations than the two-balloon model, and compare their performance with the data-driven simulator built by Suo et al. [20] In doing so, we compare the benefits of physics-based versus data-driven simulations of real-world dynamical systems in the context of the ventilator-lung control problem.

This paper acts as an evaluation of the end-to-end pipeline developed by Suo et al. [20] for learning controllers that improve upon PID controllers for tracking pressure waveforms during invasive mechanical ventilation. By adding to this larger project, this paper contributes to efforts towards building more effective and robust mechanical ventilators for use in clinical settings.

2. Background Information and Problem Description

2.1. The Control Problem

In full generality, a control problem involves a dynamical system which is governed by some intrinsic parameters, has a partially observable state at any given time, and an input that we can control to achieve a desired functionality. Formally,

Dynamical system:

$$x_{t+1} = f(x_t, u_t, w_t) \qquad y_t = g(x_t) \tag{1}$$

- x_t = state
- y_t = observation
- u_t = control
- w_t = noise/disturbance

Goal:

$$\text{minimize cost function:} \qquad \sum_t c_t(y_t, u_t) \tag{2}$$

The function $g(x)$ is used to model hidden variables. When the state is fully observable $y_t = x_t$. What distinguishes this formalization from the classical reinforcement learning paradigm is the continuous, infinite state and action space. Note that in this research, time will be treated as a discrete quantity.

2.2. The Ventilator-lung system

An example of a control problem in the real-world is the ventilator-lung system. Due to COVID-19, this control problem has become especially relevant.

The ventilator is attached to an endotracheal tube, which is inserted into the patient's main airway passage, the trachea. The flow of air into the lungs is controlled using a valve in the tube - the wider the valve opens the greater the flow. The output signal is usually the measured airway pressure. The desired behaviour is represented as a pressure waveform, which describes the ideal lung pressure at

different points in time during the respiratory cycle. The goal of the ventilator is to control the valve in such a way that the ideal pressures are achieved.

The simplest mathematical description of the ventilator system is modeled after the two-balloon experiment. [20]

$$v_{t+1} = v_t + u_t * \Delta t$$

$$p_t = p_0 + \left(1 - \left(\frac{r_t}{r_0}\right)^6\right) * \frac{1}{r_t r_0^2} \quad , \quad r_t = \left(\frac{3v_t}{4\pi}\right)^{\frac{1}{3}}$$

- p_t = pressure (measured signal)
- v_t = tidal volume (aux. variable)
- r_t = radius (aux. variable)
- y_t = observation
- u_t = input flow/diameter of ventilator valve (control variable)

In this simple formulation, there is no noise in the system. r_t is a hidden parameter which we don't know as part of the observation, but it contributes to the dynamics of the system.

A general form of this dynamical system would be:

$$p_{t+1} = f(p_t, u_t, w_t)$$

We have several different ways of defining a cost function. One option is to consider the squared difference between measured and target pressure.

$$c_t(p_t, u_t) = |p_t - p_t^*|^2$$

where p_t^* is the target pressure as described by the ideal waveform. The state of the art controller used for ventilator-lung systems is a linear controller, known as the PID controller.

2.3. PID control

A linear controller decides on an input based on a linear combination of the previous k states, i.e.

$$u_t = \sum_{i=0}^{k} \alpha_i x_{t-i} + \alpha_0 \tag{3}$$

Another common formulation is to decide the input based on a linear combination of the previous k *errors*, where error is defined as the difference between the state and the target state.

$$u_t = \sum_{i=0}^{k} \alpha_{i+1} \varepsilon_{t-i} + \alpha_0, \qquad \varepsilon_t = x_t - x_t^* \tag{4}$$

Note that linear functions are unchanged by affine transformations of the variable, therefore this alternative definition makes no difference, i.e. the two formulations above are equivalent.

The PID controller is one of the most widely used controllers in the engineering industry. It is a linear controller with 3 different components:

1. Proportional control: $u_t = \alpha x_t \quad \alpha \in R$
2. Integral control $u_t = \alpha \sum_{i=0}^{t} x_{t-i} \quad \alpha \in R$
3. Differential control $u_t = \alpha(x_t - x_{t-1}) \quad \alpha \in R$

The idea behind these three components is to control the system in different ways. Intuitively, the three components can be understood as follows. Proportional means the greater the error, the greater the input. In the ventilator-lung example, the further we are from the target pressure the higher we want the input flow to be, so the proportional parameter would be high. Integral means we want to take into account the sum of recent errors. For example, if the error is not high in one single step but has been accumulating over time, we want to make sure we realize that the system has been accumulating error for a while. In the ventilator-lung example, let's say our difference from the target pressure is not that high at one time step, so we don't input a large flow, but then the error stays that way for a long time, and it takes much longer to reach the target pressure. Differential means that the input is decided in terms of the difference between the last two errors. The intuition

is that if we start seeing a reduction in error, it means that we are approaching the target and thus should gear down our input. In the ventilator-lung example, if the reduction in error is significant between subsequent time steps, it means we are rapidly approaching the target pressure, so in order to prevent over shooting, we ought to reduce the input flow.

2.4. Previous work done by Suo et al. [20]

Suo et al. [20] learnt a data-driven lung simulator based on data collected on a mechanical lung, in order to use the simulator to train a deep neural-net based controller. In their work they showed that (a) the data-driven lung simulator models the human lung better than the physics-based (two-balloon) model, and (b) the deep neural-net based controller performs better than a PID controller.

In that work, the motive behind the data-driven lung simulator is the limitations of the two-balloon model. They point out that such a model oversimplifies fluid flow dynamics, does not take into account delay in pressure propagation, and underspecifies variations across lungs. They compare their data-driven simulator to this two-balloon model and show that the data-driven model is the better choice.

However, comparison with a two-balloon model is not quite sufficient to show that the data-driven simulator performs better than a physics-based simulator. The two-balloon model is an ~~extreme~~ simplification, but a data-driven simulation is not the only alternative. In fact, a more sophisticated physics-based simulation would serve as a better benchmark for the quality of the data-driven simulation. To build such a physics-based simulation, a basic understanding of respiratory mechanics is necessary.

2.5. Respiratory Mechanics

The primary function of the lung is to allow the oxygenation of blood and the removal of carbon dioxide. There are three essential parts of this function: ventilation, diffusion and blood flow. The lung's anatomy and physiology work together to allow the effective execution of each part.

Anatomically, the lung consists of three components - the airways, the parenchyma and the interstitium, each with a distinct role. The airways are responsible for allowing air to enter and exit

the lungs (ventilation), the parenchyma is where the gas exchange with the blood capillaries takes place (diffusion and blood flow) and the interstitium is the connective tissue which provides support to the lung structure.

The airways consist of a series of conducting tubes which split into smaller and narrower parts as they spread across the lungs. The first and widest airway is the trachea which bifurcates into the two main bronchi, one entering the left lung and the other entering the right one. These two bronchi further bifurcate into primary, secondary and tertiary level bronchi until they become the smallest bronchioles which are the narrowest airways before the alveoli.

The airways enable air to enter and exit the lungs, but for this to happen there needs to be some mechanism to control this breathing. In healthy individuals, this function is performed by the muscles in the thoracic cavity.

Inspiration is an active process. During inhalation, the intercostal muscles and the diaphragm contract increasing the anteroposterior diameter of the thorax. This creates negative pressure in the pleural cavity, which causes the lungs to expand. The increase in lung volume makes the lung pressure drop, which in turn causes atmospheric air to be drawn into the lungs. As air flows in, the lung pressure rises.

Expiration, on the other hand, is passive during normal breathing. During exhalation, the intercostal muscles and the diaphragm relax and the elastic recoil of the lung and chest wall allow the lungs to return to an equilibrium position. The elastic recoil, coupled with the high pressure in the lungs at the end of inspiration, allow air to be released from the lungs. As a result, lung pressure drops back to its equilibrium position, i.e. the same as it was at the start of inspiration.

The two important variables that control breathing are pressure and volume. Figure 1 shows the important lung volumes and Figure 3 shows the important lung pressures during a single breath, i.e. an inspiration followed by an expiration.

As mentioned above, the mechanics of breathing are controlled by the muscles in healthy individuals. In patients with respiratory failure, however, these muscles are unable to perform this function effectively. Thus, mechanical ventilators take up the role of controlling the in-flow and

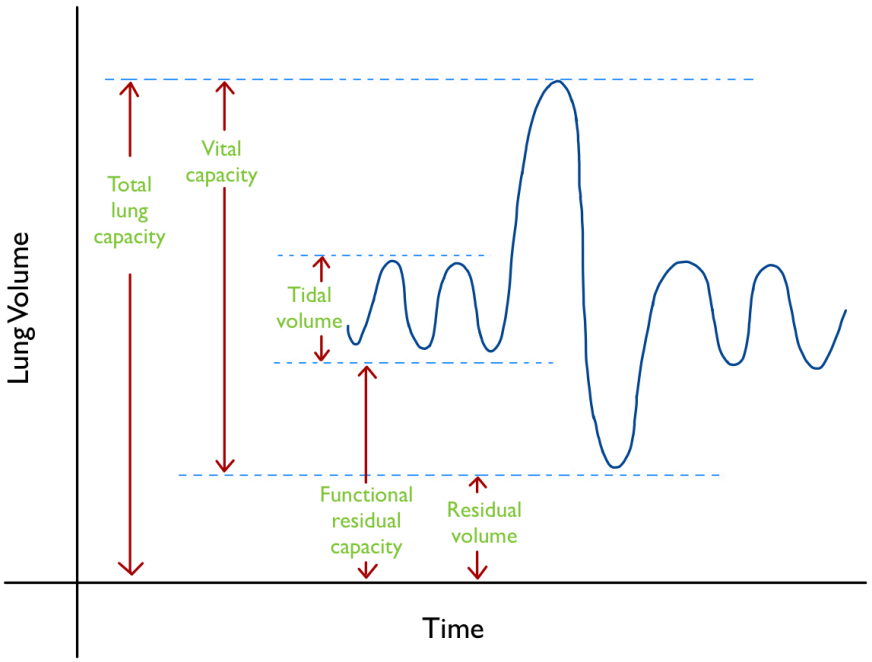

Figure 1: Terminologies for lung volumes at different points in the respiratory cycle.

out-flow of gas in the lungs.

In mechanical ventilation, the two most commonly used modes of ventilation are called pressure-controlled ventilation (PCV) and volume-controlled ventilation (VCV). As the names suggest, the former mode attempts to track the ideal pressure waveform of a lung and the latter attempts to track the ideal volume waveform of a lung. For the purpose of this research, we are concerned with pressure control ventilation. As such, our controllers will attempt to achieve a clinician-specified target pressure trajectory.

The following variables are going to be important for the mathematical lung models I use in this paper.

- **Tidal volume**: The amount of air that moves in or out of the lungs with each respiratory cycle. Its value is approximately 500 ml in an average healthy adult male and approximately 400 ml in a healthy female.
- **Positive end-expiratory pressure (PEEP)**: The positive pressure that remains in the airways at the end of a respiratory cycle (i.e. end of exhalation). This has to be greater than the atmospheric

pressure in the mechanically ventilated patients in order to keep the alveoli from collapsing.

- **Peak inspiratory pressure (PIP)**: The highest level of pressure achieved in the lungs during inhalation.
- **Plateau Pressure**: The pressure inside the lungs at the end of inspiration and before the start of expiration, when there is no airflow.
- **Airway pressure**: The pressure at the airway openings during mechanical ventilation, both during inspiration and expiration.
- **Pleural Pressure**: The pressure inside the pleural space.
- **Transpulmonary Pressure**: The difference between pleural pressure and intrapleural pressure.
- **Flow rate**: The volume of gas delivered by the ventilator per unit time.
- **Expiration time**: The time interval during which expiration happens. This is usually a controlled variable in mechanical ventilators.
- **Inspiration time**: The time interval during which inspiration happens. This is usually a controlled variable in mechanical ventilators.
- **Compliance**: The change in lung volume per unit change in pressure. This is a measure of the lung's ability to stretch. Its inverse is called the lung **elastance**: $\frac{1}{compliance}$
- **Airway resistance**: The resistance of the respiratory tract to airflow during inhalation and exhalation.
- **Respiratory Rate**: The number of breaths per unit time. Normal respiratory rate is between 12 to 16 breaths per minute.
- **Minute ventilation**: The volume of gas inhaled (inhaled minute volume) or exhaled (exhaled minute volume) from a person's lungs per minute. This is the product of tidal volume and respiratory rate.

For further reading on lung physiology and mechanics, see West 2012 [23] and Bates 2009 [3].

3. Approach

3.1. Project aim: Physics-based lung simulation for ventilator control

The goal of this paper is to explore different kinds of physics-based models of ventilation in the human lungs, develop simulations based on these models, and compare the performance of these simulations to the data-driven simulator developed by Suo et al. [20]

The aim is for the simulations to perform well in the following two ways:

(a) The simulation should experience changes in internal state similar to that of an actual human lung. For example, if the same amount of gas is allowed to flow into the simulation and into a human lung, the resulting pressure of the simulated lung ought to be the same as the resulting pressure of the human lung.

(b) A controller trained on the simulation should also perform well on an actual human lung.

Since we do not have access to an actual human lung for experimental purposes, I will be using a mechanical test lung used by Suo et al. as a proxy for the human lung. This poses challenges which I discuss in more detail in Sections 5.2 and 6.2.

3.2. Types of physics-based lung models

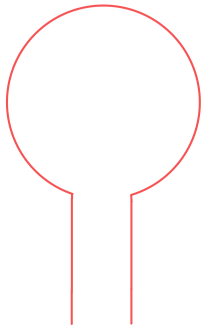

Figure 2: Single Compartment Model

3.2.1. Single compartment

The simplest model of the respiratory system, which is still the most commonly used, is made of two

lumped elements, one representing an elastance (balloon), and the other representing a resistance (pipe). [19]

$$p_t = \dot{v}_t \, R + E \, v_t + p_0 \tag{5}$$

Here, R represents the airway **resistance** of the lungs. Intuitively, this is a measure of how much resistance does airflow face in the respiratory tract during inhalation and exhalation. Mathematically, $R = \frac{\Delta p}{\Delta \dot{v}}$

E represents the **elastance** of the lungs, which is the inverse of the lung **compliance**. Intuitively, compliance is a measure of how hard it is to expand the lungs, that is change in volume per unit pressure change. $E = \frac{\Delta v}{\Delta p}$

A common use of this single compartment model is to use measured pressure, volume and flow data to find values for E and R. This is done using methods like multiple linear regression, electrical subtraction method, or MLE. [17, 19]

Note that this model makes a few assumptions. First, we assume that the system behaves linearly, i.e. R is independent of $\dot{V}$ and E is independent of volume. Second, in this model, inertia is assumed to not play a significant role. [19]

3.2.2. Limitations of single-compartment

Owing to its simplicity, the single compartment model has been extensively used in previous research. [1, 5, 14, 19]

However, past experimental data shows that resistance and elastance are dependent on frequency at low frequencies, particularly in the range 0-2 Hz [19]. Equation 5 does not capture this phenomenon. In this case, two or more compartments with different time constants given by ratios of compartmental resistances to elastances would be needed to model this dependence. [19]

In addition, an examination of the pressure wave after an end-inspiratory occlusion in humans and experimental data from relaxed expiration in dogs show that a single exponential function does not do a great job at modelling lung behaviour. However, a double exponential function fits very well. [19]. This limitation is explained in Figures 3 and 4.

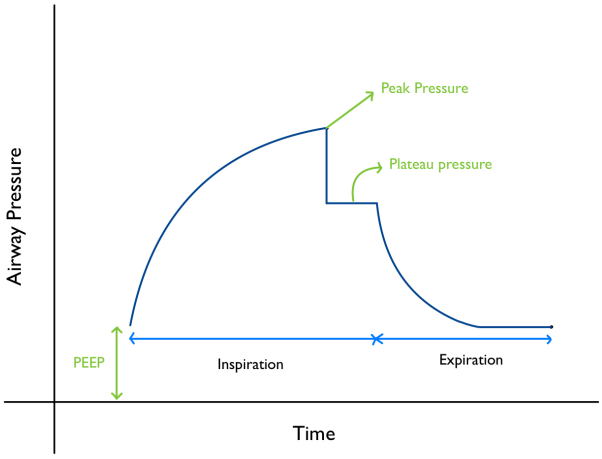

Figure 3: Ideal Wave Form. Notice how inspiration has an exponential increase and expiration has an exponential decrease in pressure.

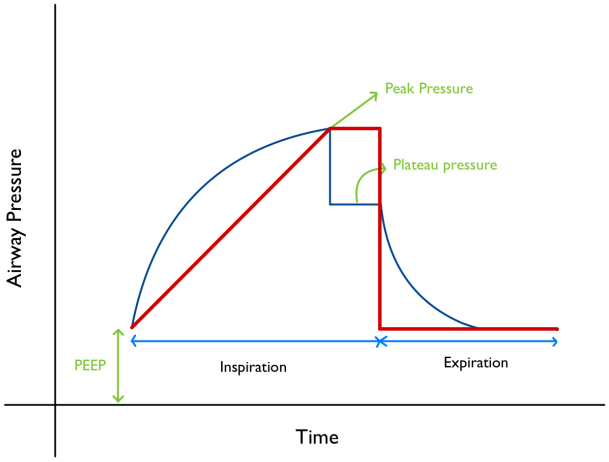

Figure 4: Breathe Wave Form used in single compartment modelling. Note how the linear increase and decrease in pressure approximates but does not accurately model the actual waveform of the human lung.

Thus, a 2-compartment model with one "fast" and one "slow" compartment would do the job.

3.2.3. Double compartment models

There are two broad kinds of double compartment models.

1. Gas Redistribution Models: These try to capture the inhomogeneous distribution of gas during ventilation in lungs
2. Rheologic Models: These try to capture the intrinsic tissue properties of the lungs

Gas Redistribution Models:

There are two further kinds of gas redistribution models: parallel and series. The **parallel model** represents the human lung with one pipe which bifurcates into two branches and each branch leads to one of the two compartments. [3, 19] In this way, the two compartments exist in parallel. Figure 5 shows a diagrammatic representation. The equation governing the dynamics is as follows:

$$\dot{P}(t)[R_1+R_2]+P(t)[E_1+E_2]=\ddot{V}(t)[R_1R_2+R_1R_C] \tag{6}$$

The **series model** represents the human lung with one pipe that opens into the first compartment and another pipe which connects the first compartment to the second compartment. In this way, the two compartments exist in series. [3, 19] Figure 6 shows a diagrammatic representation. The

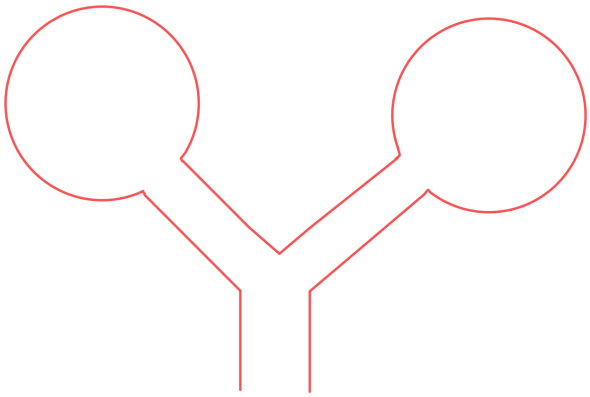

Figure 5: Parallel Double-Compartment Model

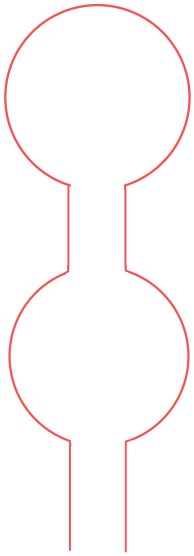

Figure 6: Series Double-Compartment Model

equation governing the dynamics is as follows:

$$\dot{P}(t)[R_2] + P(t)[E_2] = \ddot{V}(t)[R_1R_2] + \dot{V}(t)[E_2R_1 + E_1(R_1 + R_2)] + V(t)[E_1E_2] \tag{7}$$

Rheologic Models:

There are two kinds of rheologic models: viscoelastic models and plastoelastic models.

The **viscoelastic model** represents the stress adaptation properties of the lung tissues. [3, 19] This can account for the slow phase of relaxed expiration, as well as adaptation after end-inspiratory flow occlusion. [19] The equation governing the dynamics is as follows:

$$\dot{P}(t)[R_2] + P(t)[E_2] = \ddot{V}(t)[R_1R_2] + \dot{V}(t)[E_1R_2 + E_2(R_1 + R_2)] + V(t)[E_1E_2] \tag{8}$$

The plastoelastic model tries to model the quasi-static pressure-volume hysteresis in isolated lungs, but experimental results in the past have shown the viscoelastic model to be a superior rheologic model. [19]

It is noteworthy that in past studies attempting to model normal lungs, both gas redistribution and rheologic models have performed similarly. However, in abnormal lungs the two appear to play distinct roles and a combination of them is hypothesized to be the most suitable. [19]

3.3. Evaluation metric

Suo et al. define a black-box metric to evaluate the performance of StitchedSim as compared to the Balloon lung. [20] This metric is called the **open loop distance**, which they define as follows.

Let f_1, f_2 be the transition functions that describe two dynamical systems, where the state-action space of both functions is the same. Let $D = \{u_1, u_2, ..., u_T\}$ be a distribution over sequences of controls. Then the open loop distance with respect to horizon T and a control sequence distribution D is defined as:

$$||f_1 - f_2||_{ol} \equiv \mathbf{E}_{u_1:T \sim D}\left[\sum_{t=1}^{T} ||f_1(x_{t,1}, u_t) - f_2(x_{t,2}, u_t)||\right] \tag{9}$$

Intuitively, this metric can be understood as follows. We take two systems, A and B, and apply the exact same sequence of controls on both systems. At each time step, we record the L1 (or some other) Euclidean norm between the state of system A and state of system B. We sum these L1 norms from each time step. The expected value of this sum as we get closer to the horizon is defined as the open loop distance.

In the second component of their work, they evaluate the performance of their deep neural network controller by getting its average L1 loss on the mechanical test lung and comparing it to the average L1 loss of a PID controller tuned on the mechanical lung itself.

Taking inspiration from these evaluation metrics, I use two distinct metrics to evaluate the performance of my simulations.

1. The open loop average MAE of the simulation against the mechanical test lung
2. The average L1 loss on the mechanical test lung of a controller trained on the simulation

For the first metric, I run the open loop test for each of the 9 lung settings on each of the three simulations against the mechanical lung and record the average MAEs. Table 3 shows the results of this process.

For the second metric, I train a PID controller on each simulation and then test this controller on the mechanical test lung. Suo et al. use their data-driven simulator as the lung module in Figure 8 to

train their controller. They then test the controller by using the mechanical test lung as the lung module.

I use the same approach with my physics-based simulations. For each of the 9 lung settings, I first use the SingleComp as the lung module in Figure 8 to tune a PID controller. Then, I run this tuned PID controller by using the mechanical test lung as the lung module. I record the average L1 loss for each setting and then repeat the entire process for DoubleComp. Table 2 shows the results of this process.

4. Related Research

As mentioned earlier, there has been extensive research into mathematical and physics-based models of the lung. For the purpose of this project, apart from the single and double compartment models that I use in my approach, there are several other kinds of models that have been theorized and tested. [19, 13, 3, 23] However, for my approach I decided to use the single and double compartment models because the former is the most widely studied model and the latter sufficiently tests my hypothesis that the single compartment mechanical test lung used by Suo et al. is not an effective benchmark for the validation of more complex physics-based models. I briefly mention below other physics-based models that I did not use. Future research with better ground truth methods could incorporate these models. For further reading on lung models, refer to Bates, 2009. [3]

4.1. Non-linear models

Examples of non-linear lung models include the **RC Model** [16, 17] and the **Work of Breathing (WOB) based model** [2]. The RC model uses unscented Kalman Filtering (UKF) for parameter estimation. The WOB based model incorporates a non-linear compliance. A viscoelastic structure (Kelvin body) replaces the original pressure-volume characterisation of the lung tissue, and pressure from muscles (P_{mus}) is the driving force instead of pleural pressure (P_{pl}). For further reading on non-linear lung models, see Polak et al. 2006, and Bates, 2009. [2, 3, 13]

4.2. Existing simulations

There are several existing lung simulations, however none of them in its current form can be integrated into the experimental setup used by Suo et al. [20], and thus cannot be useful for the aims of this specific project. The code for these simulations is not open-source so I could not adapt it to be compatible with the experimental setup. I briefly describe these existing simulations below.

SimuVent 1995 is a multi compartment model, where we can have any number of compartments between $1 - 10$. This simulation captures ventilation mechanics, gas transport, gas mixing and gas exchange. [24]

VentSim 1994 is a double compartment model which captures ventilator settings, ventilation mechanics and gas exchange. [15]

The *Web App simulator* developed by Takeuchi et al. in 2004 is a multi-compartment model, with 3 resistances, and 2 compliances. This simulation captures ventilator settings, ventilation mechanics, and spontaneous breathing efforts. [21]

5. Implementation

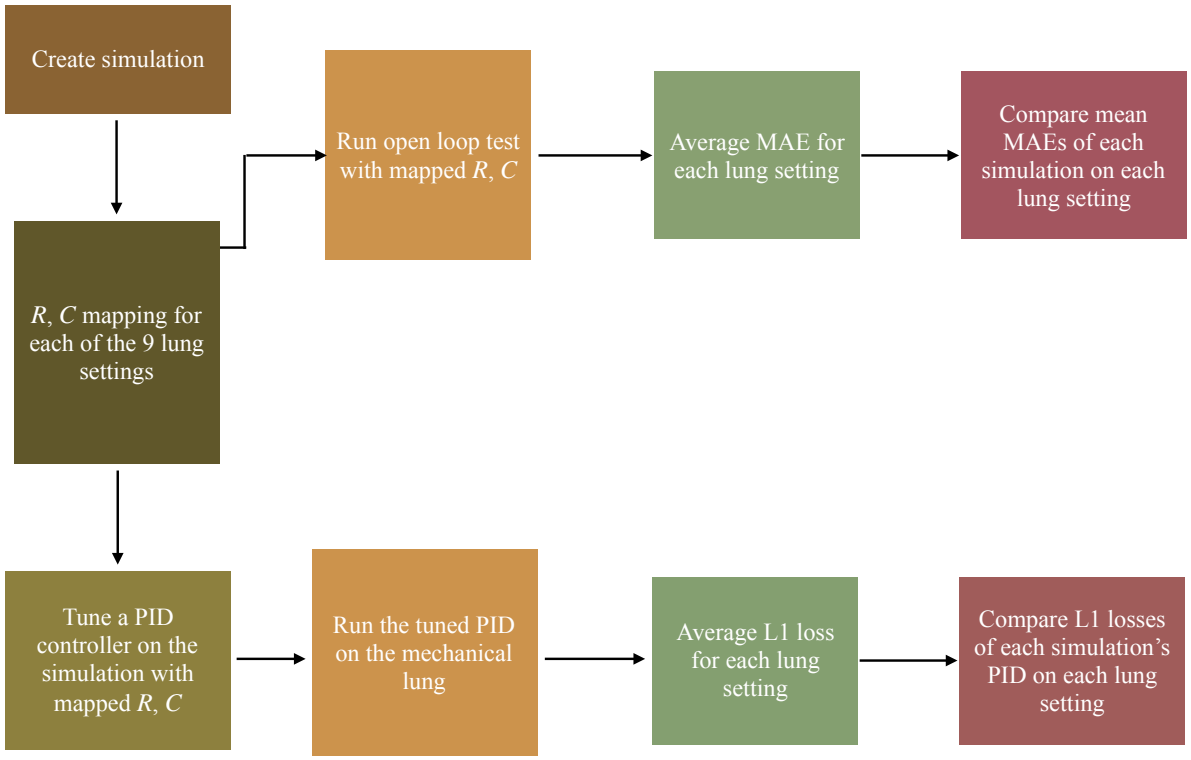

Figure 7: Summary of implementation workflow. For each of the two simulations, I find the best R, C mappings. Then I run the open loop tests for each lung setting and train a PID controller on the simulation at each lung setting. I then use the two metrics to compare the simulations as training grounds for the PID controller: 1) average open loop MAEs and 2) average L1 loss of PID on mechanical lung.

All the code I wrote is designed to work within the venti library, which was implemented by Suo et al. [20] In the following description of my implementation, I indicate explicitly the components which I did not implement myself but used as previously implemented by Suo et al. [20]

Figure 7 summarizes the workflow of my implementation, which is described in more detail below.

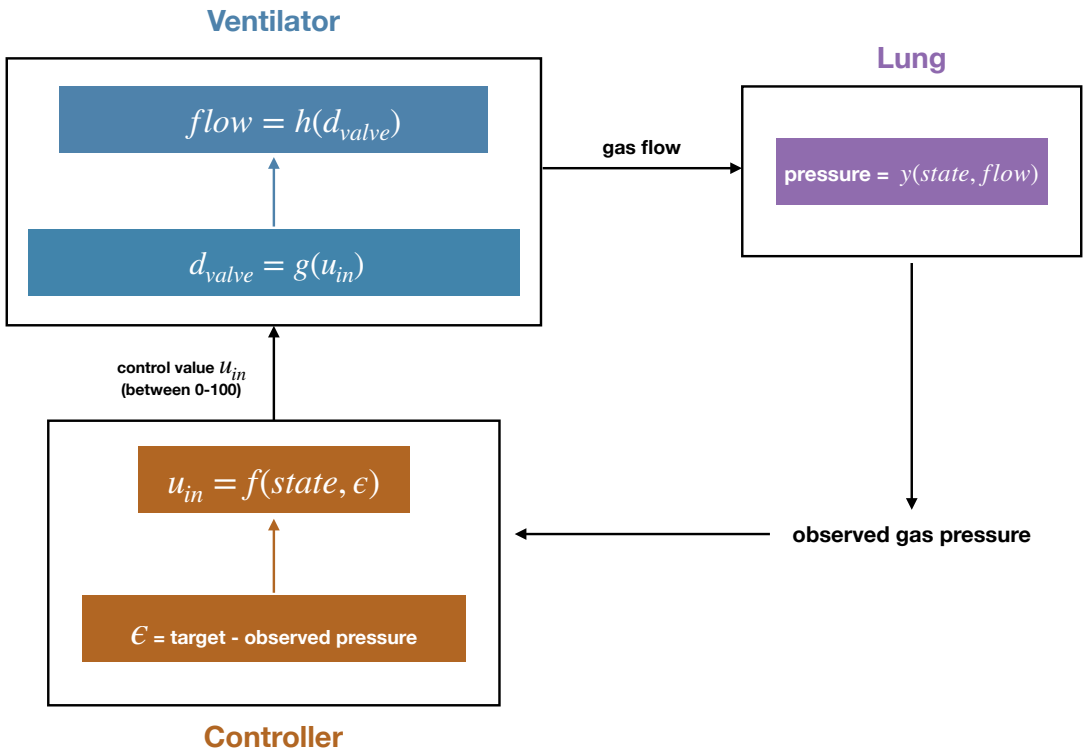

Figure 8: Diagram of the experimental setup. This setup consists of 3 components: 1) the ventilator, 2) the lung and 3) the controller. The controller outputs a scalar value, u_{in}, in the range $[0-100]$. The ventilator takes this value as input and uses it to compute the diameter of the opening in the airway valve, d_{valve}. The ventilator then uses d_{valve} to compute the gas flow which it outputs. The lung takes the gas flow value as input and uses it to update its pressure. It then outputs its pressure which the controller takes as input. The controller uses this observed pressure value to compute the error between the target and observed pressures. The controller then uses the error value to compute the next u_{in}.

5.1. Experimental setup

The experimental setup described in this section is the one used by Suo et al. [20] I explain it here for completeness, but this part of the project was not implemented by me. The set up consists of a mechanical test lung and an open-source ventilator. [9, 11]

The mechanical test lung used in this setup is commercially available under the name *QuickLung* by IngMar Medical. [9] This is a physical device that models the adult human lung. It has three lung compliance settings $C \in \{10, 20, 50\}$ and three airway resistance settings $R \in \{5, 20, 50\}$, i.e. there are a total of 9 possible lung settings for this device. Figure 13 shows a picture of the physical

device.

The mechanical lung is connected to a ventilator using a respiratory circuit. [20] This ventilator is an open source design developed by the People's Ventilator Project [11], and is used to target pressure for a fully unconscious patient, i.e. one with no spontaneous breathing. The ventilator consists of two paths, one for inspiratory airflow and the other for expiratory airflow. For each path, there is a valve that can be adjusted to change the airflow in and out of the lung. The inspiratory valve is a proportional flow valve, which can take a value in the range $[0-100]$, the higher the value, the more air flows through the valve. The expiratory valve is a binary valve, which can take a value of 0 or 1. A value of 0 means the valve is closed and no airflow is permitted. A value of 1 means the valve is open and maximum airflow is permitted.

In their experiments, Suo et al. [20] focus on the data from inspiratory phases to train their simulator, because inspiratory phases are the most relevant to the clinical setting and the ventilator-lung control problem. For this paper, I will also focus on the inspiratory phases since most of the data available to me through the venti library is of the inspiratory phases on the mechanical lung.

Figure 8 shows this experimental setup including a controller. The ventilator takes in an integer value in the range $0-100$, u_{in}, and uses this to calculate the diameter of the opening of the airway valve. This diameter of the opening in turn determines the value of gas flow into the lung. In the lung, this gas flow value and the previous state is used to calculate the resulting pressure of the system. The controller observes the pressure in the lung and calculates the error as compared to the ideal lung pressure at that phase. Depending on this error, the controller determines the u_{in} for the next time step.

5.2. Simulations

For this project, I compared four different simulations: StitchedSim, BalloonLung, SingleComp and DoubleComp.

The **StitchedSim** is the data-driven learned simulator implemented by Suo et al. [20] The **BalloonLung** is based on the two-balloon model of the ventilator-lung system described in Section

2.2. This was also implemented by Suo et al. [20] to which I made minor changes in order to run the open loop test on it. The last two simulations, SingleComp and DoubleComp were implemented by me.

The **SingleComp** is based on the single compartment model described in Section 3.2.1. It defines the following parameters upon initialization:

- $self.R$ = airway resistance
- $self.C$ = lung compliance
- $self.P0$ = initial lung pressure

Its state at time step t is defined by:

- $self.volume$
- $self.pressure$
- $self.flow$

When the simulation runs as part of an experiment, at each time step it takes as input a u_{in} from the controller. Using this, the current flow is calculated, which in turn updates the pressure and volume of the system.

The **DoubleComp** is based on the double compartment model described in Section 3.2.3. It defines the following parameters upon initialization:

- $self.R1$ = airway resistance of first compartment
- $self.R2$ = airway resistance of second compartment
- $self.C1$ = lung compliance of first compartment
- $self.C2$ = lung compliance of second compartment

Its state at time step t is defined by:

- $self.volume$
- $self.pressure$
- $self.pressure\_pvs$
- $self.flow$
- $self.flow\_pvs$

R	C	R_{single}	C_{single}	$R1_{double}$	$R2_{double}$	$C1_{double}$	$C2_{double}$
5	10	6.45	0.1	0.0	0.0	0.24	0.6
5	20	6.45	0.1	0.0	0.0	0.24	0.6
5	50	0.30	0.1	0.6	0.0	0.36	0.6
20	10	11.88	0.1	0.6	0.0	0.24	0.6
20	20	11.88	0.1	0.6	0.0	0.24	0.6
20	50	6.45	0.1	0.6	0.0	0.24	0.6
50	10	22.78	0.1	0.0	0.0	0.36	0.36
50	20	22.78	0.1	0.6	0.0	0.24	0.6
50	50	50	0.1	0.6	0.0	0.24	0.6

Table 1: Mappings from R and C values on the mechanical lung to the R, C values that result in the lowest MAE on the single compartment model and the $R1, R2, C1, C2$ values that result in lowest MAE on the double compartment model. Note that for the single compartment mode, all C have the same value, thus the model fails to capture the variation in compliance settings.

The flow and pressure at timestep $t-1$ are cached in order to calculate $\ddot{V}(t) = \dot{V}(t) - \dot{V}(t-1)$ and $\dot{P}(t) = P(t) - P(t-1)$. This simulation runs similarly to SingleComp, where at each time step it uses the input u_{in} to calculate the flow and update the pressure and volume of the system.

Note that the double compartment model has two parameters for resistance and two for compliance, whereas our mechanical test lung which I use for validation only has a single resistance and single compliance value. This makes it difficult to use the mechanical test lung as a useful ground truth for the double compartment model. I decided to try and run this simulation on the set up anyway and see if the results show something interesting. I discuss this challenge in more detail in Section 6.2.

5.3. R, C mapping

After implementing the environments, I ran the open loop tests to get a sense of the performance. It turned out that the physics-based simulations perform best at R and C input values different from the corresponding setting on the mechanical lung.

For example, when running the open loop test on the mechanical lung data for the setting $R = 5$, $C = 20$, the average MAE on the SingleComp when setting its own R and C to the 5 and 20 was 8.27. On the other hand, when I ran an open loop test on the same mechanical lung data ($R = 5$, $C = 20$) but with the SingleComp set to $R = 6.45$, $C = 0.1$, the average MAE was 5.99.

Thus, I ran an extensive parameter search to find the best R, C mappings for each of the two physics-based simulations I had implemented.

5.3.1. SingleComp mapping

For the SingleComp, I ran a grid search over any combination of R and C values where:

$$R,\ C \ \in \ \{0.0, 0.1, 0.2, \cdots, 0.9, 1.0, 2.0, 3.0, \cdots, 10.0\} \tag{10}$$

Table 1 shows the best R, C mapping for SingleComp. It's worth noting that the mapped C values are all equal to 0.1 which indicates that in modeling the mechanical lung, the resistance setting has a much higher impact on the single compartment model as compared to the compliance. Also, the SingleComp uses the exact same mapping for $R = 5$, $C = 10$ and $R = 5, C = 20$, which means that the difference between these lung settings is not captured well by the SingleComp.

5.3.2. DoubleComp mapping

For the DoubleComp, I ran a grid search over all combinations of $R1$, $R2$, $C1$, $C2$ values, where each of these could be one of 6 values, evenly spaced in the interval $(0.0, 0.6)$.

The parameter search I performed for DoubleComp had a narrower range simply because of the computation time. I was initially searching for all four parameters in DoubleComp, i.e. R_1, R_2, C_1 and C_2, in the range over which I searched for SingleComp parameters, i.e. $\{0.0, 0.1, 0.2, \ldots, 0.9, 1.0, 2.0, 3.0, \ldots, 10.0\}$.

However, this meant that I was looping over $20^4 \times 9 = 1440000$ cases, and that was computationally too expensive. So I ran the full range keeping $R_1 = R_2$ and $C_1 = C_2$, and noticed that the best parameters were all below 0.5, and thus decided to search only over 6 values evenly spaced in the interval $[0.0, 0.6]$.

By doing this I reduced the number of cases to $6^4 \times 9 = 11664$ which is significantly better than before. However, this parameter was not as thorough as it should have been. For most settings of R and C, the best mapped C I found was 0.6 but since that was the upper limit of my search range, I can't be certain that the performance wouldn't be even further improved with a higher value of C.

Table 1 shows the final R, C mapping for DoubleComp. From the parameter search results it seems like the $C1$ parameter is the only one that varies significantly with changes in the lung settings. This is another indication that the DoubleComp model perhaps has superfluous parameters when using the mechanical test lung as ground truth.

5.4. PID tuning

The PID can be tuned on StitchedSim using gradient descent, but that does not work for the physics-based simulations since the physics-based simulations are not differentiable. Specifically, the transformation from the control input value to the gas flow is not a differentiable function. Thus, I decided to use a grid search for PID tuning on the simulations.

For each of the three simulations, SingleComp, DoubleComp and StitchedSim, I ran a PID grid search. My initial plan was to do a wide range PID grid search and then test each of these PIDs on the mechanical lung. However, there were issues with running the mechanical lung remotely, so I decided to use the data we had already collected in past runs on the mechanical lung.

PID parameters in the range $[0.0 - 10.0]$ had been run on the mechanical lung and data on the L1 losses corresponding to each set of PID parameters was stored in the venti library. Thus the grid search that I ran when tuning PIDs on each individual simulation was for parameters in this range, so that I could later get their corresponding performance on the mechanical lung.

In addition to tuning PIDs on each of the three simulations, I also found the best PID on the mechanical lung itself, which I defined as the PID parameters with the lowest L1 loss on the mechanical lung.

Table 4 shows the result of the PID tuning on each of the 3 simulations and the mechanical lung itself.

5.5. PID testing

After tuning these PIDs, I "ran" the mechanical lung with the respective PIDs from each of the three simulations. As mentioned earlier, I didn't have access to the mechanical lung remotely to run

multiple trials, so by "ran" I mean I replayed the data that had been previously collected from the mechanical lung using these PID parameters and was stored in the venti library.

From each of the PIDs, I got the average L1 loss on the mechanical lung. These losses are reported in the results in Section 6.1.

6. Evaluation

6.1. Results

6.1.1. Open Loop Test Performance

Figure 9 shows the average open loop MAE for each of the 4 simulations (the Balloon lung, the SingleComp, the DoubleComp and StitchedSim) at all 9 possible lung settings. The numerical values are detailed in Table 3 in section 9.1 of the Appendix.

Individual Performance of Simulations

The StitchedSim performs worse at higher resistance and higher compliance values. In particular, its performance at $R = 50$ is notably worse than at lower resistances. Over the 9 lung settings, the average MAE on StitchedSim varies between $0.27 - 6.0$. It performs best at $R = 5$, $C = 20$ (lowest resistance) and worst at $R = 50$, $C = 20$ (highest resistance).

Across the 9 lung settings, the average MAE on the SingleComp varies between $2.81 - 10.97$. It performs best at $R = 5, C = 50$, with an average MAE of 2.81. Intuitively, this makes sense as this is the lowest resistance and highest compliance setting, which makes the system closest to an ideal physical system. Thus, the SingleComp performs worse at higher resistances and lower compliance values. Like StitchedSim, it performs worse at $R = 50$ than at lower resistances, and in addition, the average MAE at $C = 10$ (lowest compliance) is notably worse than $C = 50$ at the two lower resistance settings ($R = \{5, 20\}$).

Over the 9 lung settings, the average MAE on the DoubleComp varies between $4.48 - 9.4$. Similarly to the SingleComp, DoubleComp also performs best at $R = 5, C = 50$. In general, the DoubleComp appears to do better at higher compliance settings though the variation in the average MAE for this environment is not as extreme as the others across different lung settings.

Across the 9 lung settings, the average MAE on the Balloon Lung varies between $13.76 - 21.66$. Upon inspection, I observed that the average MAE on the balloon lung is slightly lower at the same R, C settings at which the SingleComp performs better, i.e. higher compliance values yield better performance. I think this is because of the same reason I described above for the SingleComp, i.e.

higher compliance indicates a more ideal physical system thus physics-based models perform better in such settings. Other than this, there are not any clear patterns in the performance of Balloon Lung on varying values of R and C. Intuitively, this makes sense since the Balloon Lung environment does not use the R, C values in computing its state.

Comparing Performance of Simulations

Figure 10 shows the MAE of each of the four simulations in an open loop test as the number of simulated pressures increases. Figure 10 A is the plot at $R = 20$, $C = 20$. It visualizes the following general trend in the performance of the four simulations. Both the SingleComp and the double

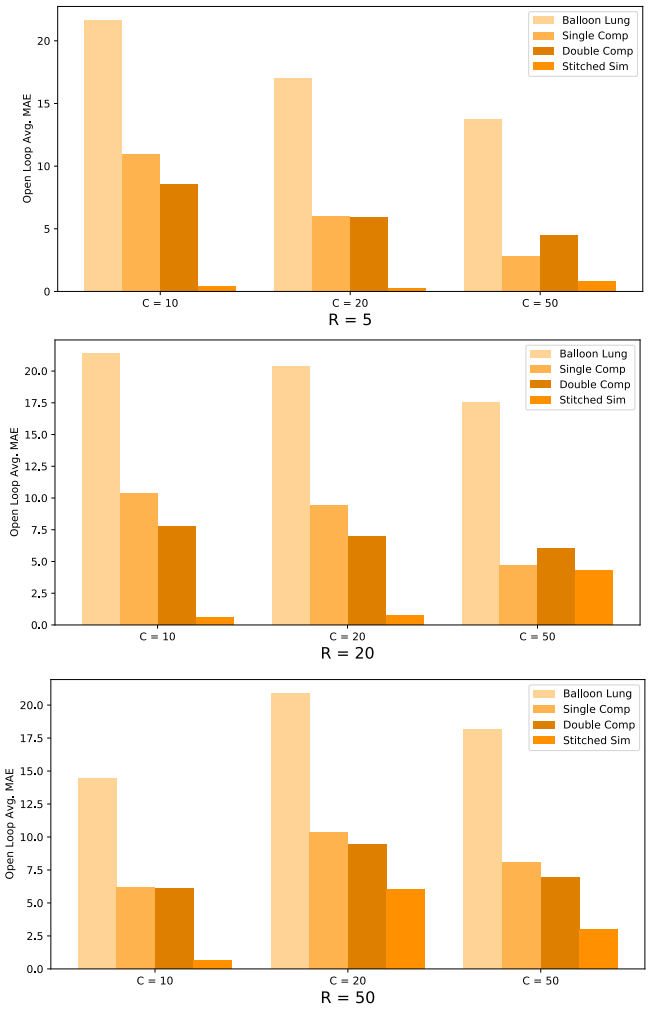

Figure 9: Grouped Bar Chart of the Open Loop test average MAEs at the 9 different lung settings on all four simulations: Balloon Lung, SingleComp, DoubleComp and StitchedSim. The StitchedSim performs the best and the Balloon Lung performs the worst at all lung settings. Both SingleComp and DoubleComp perform noticeably better than the Balloon Lung. Notice that at $R = 20, C = 50$ the SingleComp performs almost just as well as the StitchedSim.

compartment lung simulations perform significantly better than the balloon lung, though they both still perform worse than the StitchedSim. The StitchedSim has the lowest average MAE on each of the 9 lung settings, whereas the Balloon Lung has the highest average MAE on each setting.

There are several interesting points to note. Figure 10 B and C, with ($R = 5$, $C = 50$) and ($R = 20$, $C = 50$) respectively, both show that at certain lung settings the MAE of the SingleComp and the DoubleComp in fact approach that of the StitchedSim when the number of simulated pressures in the open loop tests is high.

Additionally, Figure 9 shows that, for $R = 20$, $C = 50$ the average MAE on the SingleComp (4.68) is very close to that of the StitchedSim (4.31). The same figure also shows that, at both $R = 5$ and $R = 20$, the DoubleComp does better at lower compliances, while the SingleComp does better at $C = 50$. All the simulations act slightly strangely at $R = 50$, i.e. any patterns that are visible at the lower two resistance settings appear to be contradicted at this higher resistance value. My hypothesis is that when the resistance is too high the system strays far from the mechanics of an ideal gas, and thus it is hard to capture in simple models. However, my results do not conclusively support this and further investigation needs to be done to conclude as such.

6.1.2. PID Controller Performance

Refer to Table 4 in section 9.1 of the Appendix for the results of the PID tuning on the 3 simulations (SingleComp, DoubleComp, StitchedSim) and on the mechanical lung itself. The table reports the PID coefficients for each lung setting that results in the lowest loss in that environment.

Table 2 shows the loss on the mechanical lung for each of the four sets of PID parameters (SingleComp, DoubleComp, StitchedSim and the Mechanical Lung itself) on each of the 9 lung settings.

Individual Simulations:

The loss on the PIDs tuned on the SingleComp varies from $0.33 - 1.51$ across the 9 lung settings. These PIDs perform the worst at the highest resistance setting, $R = 50$.

The loss on the PIDs tuned on the StitchedSim varies from $0.29 - 1.16$ over the 9 lung settings. The PID tuned at $R = 50, C = 50$ (highest resistance and compliance) performs the worst.

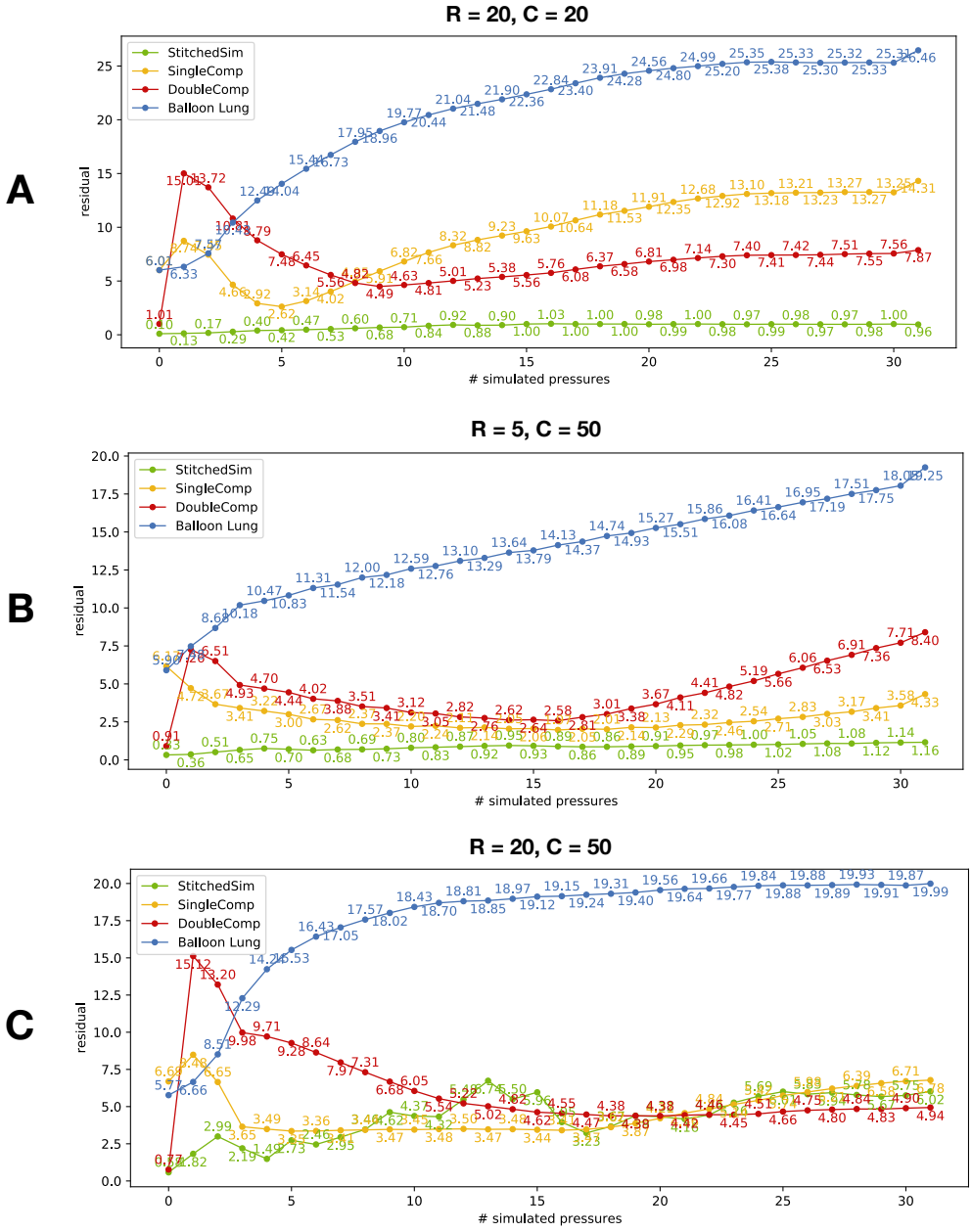

Figure 10: Plots of average MAEs on open loop tests against the number of simulated pressures in each test. **A** shows the plot for $R = 20$, $C = 20$. Here both SingleComp and DoubleComp perform worse than StitchedSim but better than the Balloon Lung. **B** shows the plot for $R = 5, C = 50$. Even though SingleComp and DoubleComp still do worse than StitchedSim, we still see SingleComp's performance is very close to that of StitchedSim. **C** is the plot for $R = 20$, $C = 50$. Here we see a similar trend to that in **B**, only that both DoubleComp and SingleComp end up performing as well as StitchedSim as the number of simulated pressures gets higher. As seen in Table 3, the average MAE for SingleComp (4.68) and StitchedSim (4.31) are very close for this lung setting.

R	C	$Loss_{single}$	$Loss_{double}$	$Loss_{stitched}$	$Loss_{mech}$
5	10	0.33	3.62	0.27	0.26
5	20	0.5	1.08	0.37	0.37
5	50	0.9	0.86	0.82	0.82
20	10	0.39	4.06	0.29	0.24
20	20	0.55	3.59	0.58	0.47
20	50	0.51	1.08	0.49	0.49
50	10	0.61	0.71	0.53	0.39
50	20	1.25	3.13	0.36	0.36
50	50	1.51	3.46	1.16	0.65

Table 2: Loss on mechanical lung with a PID controller set with parameters tuned on the three different simulators and the mechanical lung itself. At some R, C values, the PID tuned on the StitchedSim performs better, for example: $(R = 5, C = 10), (R = 20, C = 20), (R = 50, C = 20)$. However, at other PID parameters, the PID tuned on the Single Compartment Model performs better. For example: $(R = 5, C = 20), (R = 5, C = 50), (R = 20, C = 10)$

The losses on the best PIDs on the mechanical lung itself vary from $0.24 - 0.82$. Like StitchedSim, the best PID at $R = 50, C = 50$ performs worse than the PIDs at most other settings.

The performance of the PIDs tuned on DoubleComp is significantly worse than the other three sets of PIDs, with losses ranging from $0.71 - 4.06$ across the different lung settings.

Comparing Simulations:

For all environments, the PIDs perform worse at high resistance settings.

In general, the PIDs tuned on the StitchedSim perform slightly better than the ones tuned on SingleComp. Notably, at the several lung settings, the PID tuned on the StitchedSim does just as well as the best PID on the mechanical lung itself. This is seen at $(R, C) = \{(5, 20), (5, 50), (20, 50), (50, 20)\}$.

While the PIDs tuned on the SingleComp generally perform slightly worse, there are three important observations to be made here:

Firstly, the loss on the SingleComp PIDs is higher than on the StitchedSim by a very small margin,

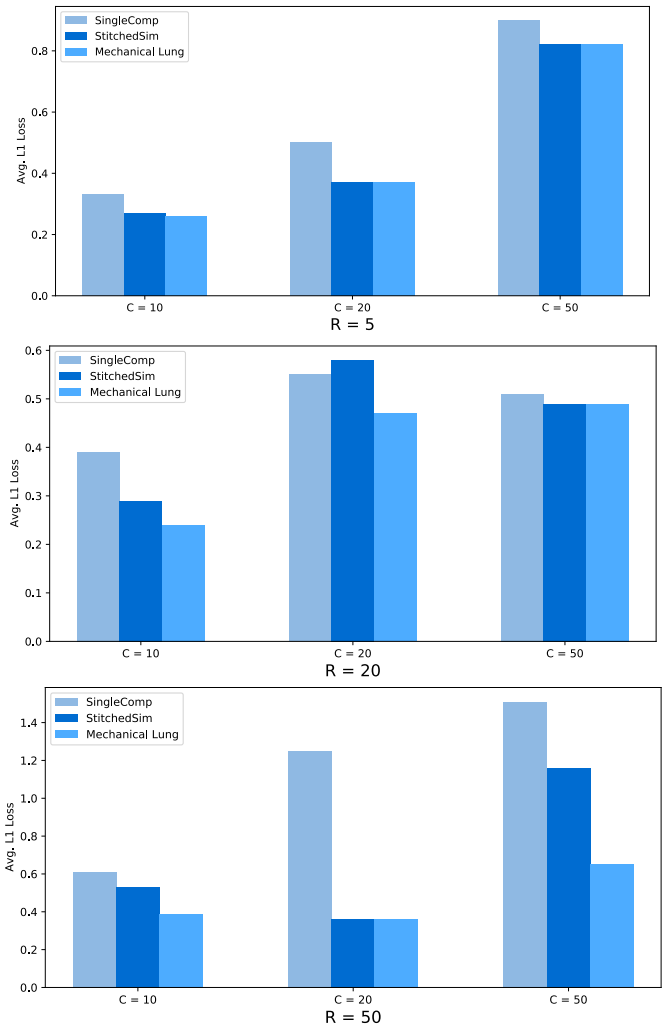

Figure 11: Grouped Bar Chart of the L1 losses for each lung setting on the sets of PIDs tuned on each of the three environments: SingleComp, StitchedSim and Mechanical Lung. Note that the performance of the PID tuned on the SingleComp is *better* than the one tuned on the StitchedSim at $R = 20, C = 20$.

except at $R = 50, C = 20$. Figure 12 shows how this difference in loss might not be so significant. The figure shows plots for $R = 50$ and $C = 10$ for three different PIDs: one tuned on the mechanical lung, one on the StitchedSim and one on the SingleComp. The mean loss at this lung setting for the SingleComp PID is higher than the StitchedSim PID, as shown in Table 2, but the plot shows that the StitchedSim PID results in more ringing and overshooting than the SingleComp PID.

The second interesting thing to notice is that the PID tuned on the SingleComp at $R = 20, C = 20$ performs *better* than the one tuned on the StitchedSim. The difference in loss here is very small though, like the cases in which SingleComp does worse than StitchedSim.

These differences in the PID performance are visualized in Figure 11. In this figure I have not

plotted the PIDs tuned on the DoubleComp since the significant difference in loss made it hard to see the trends between the other 3 sets of PIDs.

6.2. Discussion

SingleComp vs. DoubleComp

The SingleComp does better than the DoubleComp at certain settings and worse at others. Theoretically, DoubleComp should perform better than the SingleComp since it is a more sophisticated model and tries to capture more complexity. There are several possible explanations for why the DoubleComp does worse than the SingleComp at certain settings, and why the controller trained on the DoubleComp does significantly worse than all the other environments.

Firstly, due to computation cost, the parameter search for the DoubleComp model was not as comprehensive as it could have been, as described in the Implementation section. Secondly, the mechanical lung which we are testing on, QuickLung, is physically a single compartment device. This can be seen in Figure 13. Thus the complexity of multiple compartments and regional inhomogeneity in the human lung that the DoubleComp and more sophisticated models [5, 19, 24] try to capture cannot really be tested on the *QuickLung*.

The one reason why the double compartment *does* seem to perform better than the single compartment at certain lung settings is because the DoubleComp has more parameters and thus can be tweaked to capture more complexity than the single compartment model, even if the complexity it captures does not correspond to the physics-based intuition behind the mathematical model.

Comparison with StitchedSim

The two main observations when comparing my simulations with StitchedSim are as follows.

1. Both SingleComp and DoubleComp have worse average open loop MAEs than StitchedSim at all settings, but for at least one setting ($R = 20$, $C = 50$) SingleComp performs almost as well as StitchedSim.
2. PIDs trained on StitchedSim perform slightly better than SingleComp, though for at least one setting ($R = 20$, $C = 20$) the PID on SingleComp does slightly better.

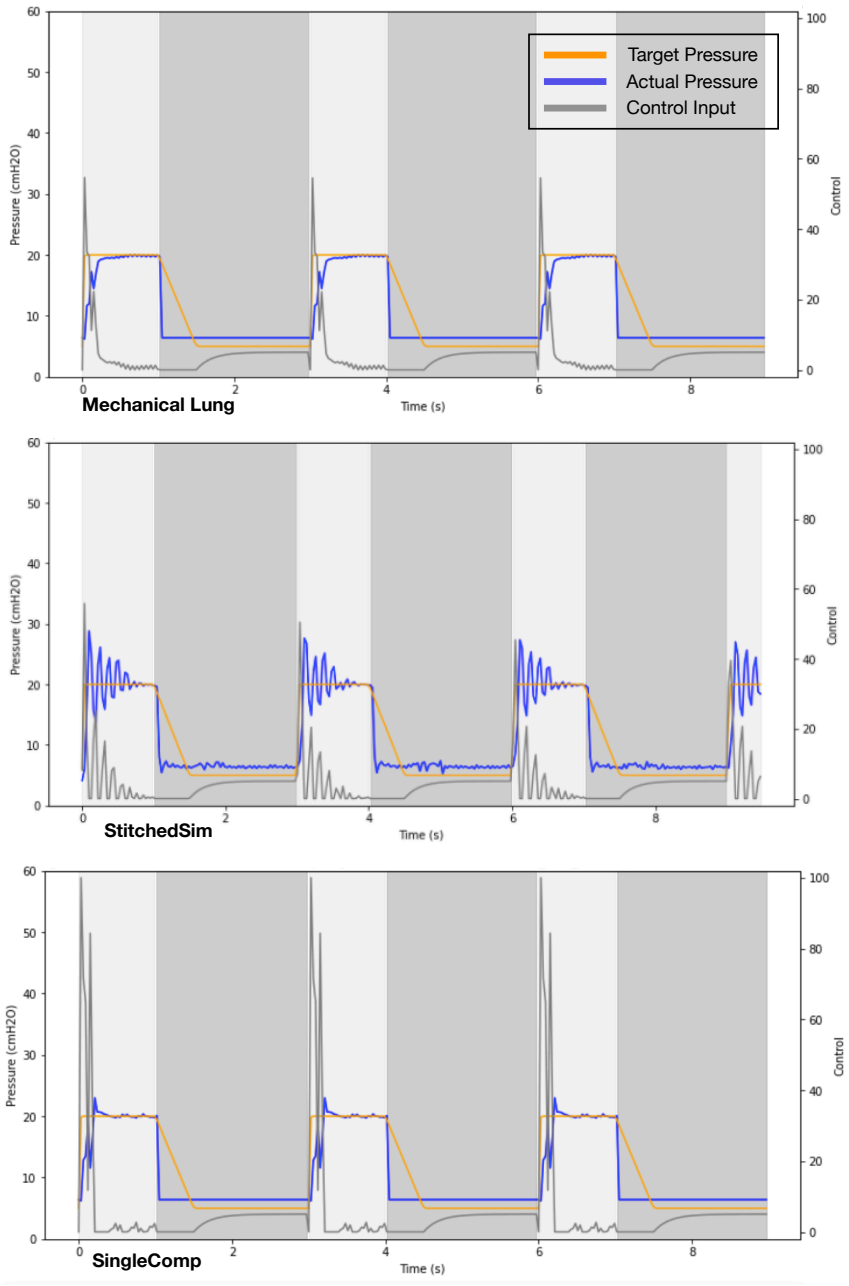

Figure 12: Plot of actual and target lung pressure over time for under PIDs tuned on SingleComp (bottom), StitchedSim (middle) and the mechanical lung itself (top). The plots here are for the lung setting $R = 50$, $C = 10$, with a PEEP value of 5 and a PIP value of 20. Note that while the PID tuned on the SingleComp has a slightly higher mean L1 loss at this lung setting (see Table 2) compared to the PID tuned on StitchedSim, yet the plot shows how the StitchedSim PID results in more ringing than the SingleComp PID. Additionally, the StitchedSim PID overshoots the PIP value several times during inspiration whereas the SingleComp PID only overshoots slightly near the beginning of inspiration, but otherwise maintains the PIP pressure pretty well.

When comparing to StitchedSim, it is crucial to pay attention to our ground truth technique. For both open loop tests and the PID tests we are using the mechanical lung model, *QuickLung*, as ground truth, instead of an actual human lung. In doing so, both physics based models are at a disadvantage because the physics models attempt to model an actual human lung, whereas the StitchedSim is trained directly on data from this specific mechanical lung. While it is true that the mechanical lung models the human lung to some accuracy which is why it is used as the industry standard, but inevitably, there are going to be differences in it from an actual human lung who's structure and physiology the physics-based models are based on.

Additionally, it is important to note that while StitchedSim performs better at all lung settings, StitchedSim does have a lot more parameters than either of the single or double compartment lung, and thus it can be fit to complex data better than the other two.

Given these considerations, the result that the StitchedSim mostly performs better than the physics-based models is unsurprising.

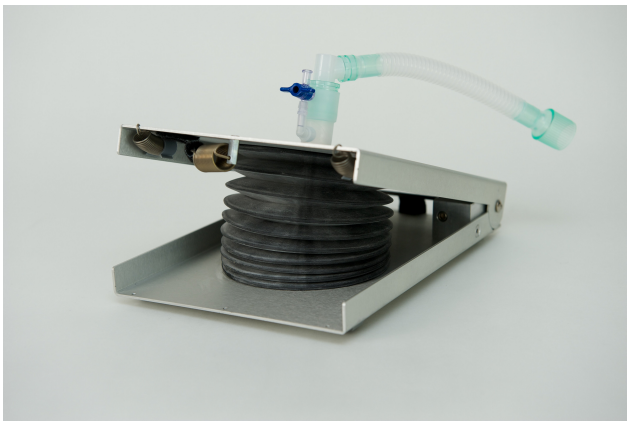

Figure 13: *QuickLung: IngMar Med.* This is the mechanical lung used as a substitute for the human lung to act as ground truth for the different simulations. Image taken from https://www.ingmarmed.com/product/quicklung/

7. Conclusion

7.1. General disadvantages of the physics based model

1. The StitchedSim has mostly better performance according to the results of this study, both in the open loop tests and in the PID tuning. Additionally, unlike the physics-based simulations, StitchedSim is differentiable, which means that we can train a neural net based controller on it directly which has been shown by Suo et al. to show significant improvements over PIDs in the ventilator control problem. [20]

2. The more complicated the lung behavior gets, the worse the linear model will perform, whereas data driven is trained on a neural network so will be able to capture more complexity.

3. The physics based models model a healthy lung. Diseased lungs will introduce non-linearities that the simple linear models can't capture. [13, 3, 19] In this case a more sophisticated, non-linear, multi-compartment model that allows for inhomogeneity in the lungs ought to be used.

Such models have been theorized [2, 13], but will need more sophisticated ground truthing methods than the QuickLung. In the case of a data-driven model like StitchedSim, we would have to train separately on each patient, or collect data from multiple patients with the same respiratory disease and use that for training.

7.2. General strengths of the physics based model

1. The two models I used in this study are relatively simple linear models, in fact the simplest models after the balloon lung, and yet they perform surprisingly well. Both the SingleComp and DoubleComp show significant improvements on the Balloon Lung in the open loop tests. In fact SingleComp performs almost as well as the StitchedSim on at least one lung setting. Additionally, the PIDs tuned on the SingleComp perform almost as well as StitchedSim, and perform even better on at least one lung setting.

This demonstrates the potential of physics-based models, and gives us reason to explore more sophisticated models presented in the literature for use in the ventilator control problem. [24, 19, 3]

The reason I did not use the more sophisticated physics-based models in this simulation is due to the constraint on ground truthing mechanisms.

2. The StitchedSim is in fact 9 separate models, one of which is chosen simply based on the input R and C. So while it looks like one model that captures general lung behavior, it is not truly generalizing across the lung settings. In addition, the greater number of parameters in the StitchedSim means that in some ways it is over-fitted to the data from the *QuickLung* and it will require retraining on a new lung device/human lung.

On the other hand, the physics based model is more generalizable and the same equations determine the behavior of the simulation at all lung settings. The only caveat here is that even for the physics based simulation I ended up using an R, C parameter search. But even with that, the physics-based models would have reasonable behaviour on a mechanical lung with different parameters.

For example, if the ground truth mechanical lung was the *ASL 5000 Breathing Simulator* by IngMar Medical [8], which has a continuous range of values for R and C, then the StitchedSim would have to retrain but the physics based model could theoretically be used out of the box. [1] However, if we could train a data driven model like the StitchedSim on data from actual human lungs, or a variety of mechanical lungs with a broader range of R and C values, then such a data driven model could potentially have both characteristics: better performance and better generalizability. This, however, requires further investigation - this study has only shown that physics-based models show promise as well.

3. The controllers trained on both the simulations perform better than the other at certain lung settings. This means that we can't decisively conclude that StitchedSim is a better simulation to train a controller on. It might be so for certain lung settings, but definitely not all. The physics based models tend to provide some insight into the clinical interpretation of the variables. For details, see

[1] My thought of getting around the R, C parameter search problem that I end up doing with the physics based models is that perhaps we can find a general mapping of the model's R and C to the mechanical lung R and C, which is what Bates talks about as well in [3]. This would be based on data of course, but it is still an empirical question whether such a general mapping is possible and if yes then this would be much more generalizable than the current StitchedSim modeling approach.

[3, 23].

7.3. Weaknesses of the project design

As discussed before, in the current project design, our ground truth is the mechanical QuickLung, which physically has a single compartment, and a total of only 9 possible R, C settings. The sophisticated physics-based lung models that I encountered in the literature search try to capture inhomogeneity in the lungs through non-linear and multi-compartment models (for example the double compartment model I used). [3, 19, 21] However, in this current set up we have no way of testing a double (or multi) compartment model unless we could test it against data from an actual human lung, or a more sophisticated mechanical lung like the *ASL 5000 Breathing Simulator* by IngMar Medical. But because we use the *QuickLung* as the ground truth, the StitchedSim inevitably does better because it is exactly fitted to the data collected from this specific prototype. We have not tested whether it does well when testing on a different human/mechanical lung. Both the physics based models, however, present a general model which a) has insight into the system and b) should perform similarly on a variety of lungs. Whether the physics-based models will perform better or worse than the StitchedSim on a different ground truth lung (either mechanical or human) remains an important empirical question that requires further investigation.

8. Future Work

There are several future directions that this project can take. As mentioned in Section 4, there is an abundance of increasingly complex physics-based mathematical models in the literature. This includes non-linear single and multi-compartment models.[19, 3] However, in order for any of these models to be used as part of the pipeline developed by Suo et al. [20], the most essential step is to obtain a more sophisticated ground truth. Even though the *QuickLung* is used as the ISO standard for ventilatory support equipment [20], the complexity that physics based models of the human lung attempt to model is not captured by this single compartment physical device.

The project team for Suo et al. [20] is currently in the process of securing the *ASL 5000 Breathing Simulator* by IngMar Medical, which is a much more sophisticated physical model of the lungs. This will open up new possibilities of testing more complex physics-based simulations and comparing them to ML-based ones.

Once a satisfactory ground truth is available, the next test for the usefulness of physics-based simulations would be to train a neural-network based controller on the physics based model and compare its performance to that of the neural-network controller trained on an ML-based simulation. This study showed that physics-based and ML-based simulations perform similarly well as training grounds for PID controllers. It remains an open question whether the same is true for the kind of deep neural network controllers developed by Suo et al. [20]. To do so, however, the physics-based environments must be made differentiable, which they currently are not due to the transformation from the control input to the gas flow value.

References

[1] H. Y. Al-Hetari *et al.*, "A mathematical model of lung functionality using pressure signal for volume-controlled ventilation," in *2020 IEEE International Conference on Automatic Control and Intelligent Systems (I2CACIS)*. IEEE, 2020, pp. 135–140.

[2] A. Athanasiades *et al.*, "Energy analysis of a nonlinear model of the normal human lung," *Journal of Biological Systems*, vol. 8, no. 02, pp. 115–139, 2000.

[3] J. H. Bates, *Lung mechanics: an inverse modeling approach*. Cambridge University Press, 2009.

[4] P. Crooke, J. Head, and J. Marini, "A general two-compartment model for mechanical ventilation," *Mathematical and computer modelling*, vol. 24, no. 7, pp. 1–18, 1996.

[5] P. Ghafarian, H. Jamaati, and S. M. Hashemian, "A review on human respiratory modeling," *Tanaffos*, vol. 15, no. 2, p. 61, 2016.

[6] H. Hazarika and A. Swarup, "Improved performance of flow rate tracking in a ventilator using iterative learning control," in *2020 International Conference on Electrical and Electronics Engineering (ICE3)*. IEEE, 2020, pp. 446–451.

[7] B. Huo and R.-R. Fu, "Recent advances in theoretical models of respiratory mechanics," *Acta Mechanica Sinica*, vol. 28, no. 1, pp. 1–7, 2012.

[8] IngMar, "Asl 5000 breathing simulator," https://www.ingmarmed.com/product/asl-5000-breathing-simulator/, 2020.

[9] IngMar, "Quicklung products," https://www.ingmarmed.com/product/quicklung/, 2020.

[10] A. A. J. Bouteloup, E. Vilsbol and F. Branciard, "Covid-19-open-source-ventilators: List of all covid-19 open source ventilator initiatives." https://github.com/bneiluj/covid-19-open-source-ventilators, 2020.

[11] J. LaChance *et al.*, "Pvp1–the people's ventilator project: A fully open, low-cost, pressure-controlled ventilator," *medRxiv*, 2020.

[12] S. H. Norwood and J. M. Civetta, "Ventilatory support in patients with ards," *Surgical Clinics of North America*, vol. 65, no. 4, pp. 895–916, 1985.

[13] A. G. Polak and J. Mroczka, "Nonlinear model for mechanical ventilation of human lungs," *Computers in biology and medicine*, vol. 36, no. 1, pp. 41–58, 2006.

[14] D. P. Redmond *et al.*, "A variable resistance respiratory mechanics model," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 6660–6665, 2017.

[15] G. W. Rutledge, "Ventsim: a simulation model of cardiopulmonary physiology." in *Proceedings of the Annual Symposium on Computer Application in Medical Care*. American Medical Informatics Association, 1994, p. 878.

[16] E. Saatci and A. Akan, "Lung model parameter estimation by unscented kalman filter," in *2007 29th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*. IEEE, 2007, pp. 2556–2559.

[17] E. Saatci and A. Akan, "Inverse modeling of respiratory system during noninvasive ventilation by maximum likelihood estimation," *EURASIP Journal on Advances in Signal Processing*, vol. 2010, pp. 1–12, 2010.

[18] P. Shi *et al.*, "Self-adjusting ventilator control strategy based on pid," 2020.

[19] T. Similowski and J. Bates, "Two-compartment modelling of respiratory system mechanics at low frequencies: gas redistribution or tissue rheology?" *European Respiratory Journal*, vol. 4, no. 3, pp. 353–358, 1991.

[20] D. Suo *et al.*, "Machine learning for mechanical ventilation control," 2021.

[21] A. Takeuchi *et al.*, "Interactive simulation system for artificial ventilation on the internet: virtual ventilator," *Journal of clinical monitoring and computing*, vol. 18, no. 5, pp. 353–363, 2004.

[22] S. J. Tzotzos *et al.*, "Incidence of ards and outcomes in hospitalized patients with covid-19: a global literature survey," *Critical Care*, vol. 24, no. 1, pp. 1–4, 2020.

[23] J. B. West, *Respiratory physiology: the essentials*. Lippincott Williams & Wilkins, 2012.

[24] T. Winkler, A. Krause, and S. Kaiser, "Simulation of mechanical respiration using a multicompartment model for ventilation mechanics and gas exchange," *International journal of clinical monitoring and computing*, vol. 12, no. 4, pp. 231–239, 1995.

9. APPENDIX

9.1. Tables

R	C	MAE_{single}	MAE_{double}	$MAE_{balloon}$	$MAE_{stitched}$
5	10	10.97	8.55	21.66	0.42
5	20	5.99	5.97	17.01	0.27
5	50	2.81	4.48	13.76	0.85
20	10	10.38	7.77	21.38	0.59
20	20	9.41	6.94	20.33	0.78
20	50	4.68	6.05	17.50	4.31
50	10	6.21	6.08	14.47	0.61
50	20	10.35	9.4	20.89	6.00
50	50	8.09	6.92	18.15	2.96

Table 3: Average MAEs on the open loop test of the Single Compartment Simulation in comparison with the Balloon Lung and the Stitched Sim for each of the 9 R,C settings. The average MAE on the Stitched Sim is the lowest, so it is performing the best. However, the single compartment model performs significantly better than the balloon lung. The mapped R and mapped C values are the values for R and C that result in the lowest MAE on the Single Compartment Simulation.

		SingleComp			DoubleComp			StitchedSim			Mechanical Lung		
R	**C**	K_p	K_i	K_d	K_p	K_i	K_d	K_p	K_i	K_d	K_p	K_i	K_d
5	10	10	10	0	0	0	0	7	0	0	9	0.8	0
5	20	10	10	0	0	0	0	8	0	0	10	8	0
5	50	8	10	0	10	10	0	10	0	0	10	9	0
20	10	10	6	0	0	0	0	7	0.5	0	3	7	0
20	20	10	6	0	0	0	0	3	9	0	3	9	0
20	50	10	10	0	0	0	0	8	7	0	5	8	0
50	10	10	0	0	10	10	0	4	0	0	0.7	5	0
50	20	10	0	0	0	0	0	2	0	0	0.7	7	0
50	50	10	0	0	0	0	0	3	3	0	1	8	0

Table 4: Results of the PID tuning step. These are the PID parameters with the lowest loss in each of the three simulations and on the mechanical lung.

9.2. Python Implementation Code

```python
############################################################
##################
################## SingleComp Lung Environment
##################
############################################################

import torch
import numpy as np

from venti.environments.core import Environment

# Sources:
# https://github.com/CohenLabPrinceton/Ventilator-Dev/blob/master/sandbox/HOWTO_RunController.ipynb
# https://github.com/CohenLabPrinceton/Ventilator-Dev/blob/master/vent/controller/control_module.py
# https://github.com/MinRegret/venti
def PropValve(x):  # copied from Controller.__SimulatedPropValve
    y = 3 * x
    y_torch = torch.from_numpy(np.array([0.03 * (y - 130)])).float()
    flow_new = 1.0 * (torch.tanh(y_torch) + 1)

    return torch.clamp(flow_new, 0.0, 1.72)


def Solenoid(x):  # copied from Controller.__SimulatedSolenoid
    if x > 0:
        return x / x
    else:
        return x * 0.

class SingleCompLung(Environment):
    def __init__(self, resistance=15, compliance=50, min_volume=0.2, peep_value=5.0, **kwargs):
        # dynamics hyperparameters
        self.min_volume = min_volume
        self.P0 = 10.0
        self.peep_value = peep_value
        self.R = resistance
        self.C = compliance
        self.flow = 0.0

        # reset states
        self.reset()

```

```python
############################################################
##################
################## DoubleComp Lung Environment
##################
############################################################

import torch
import numpy as np

from venti.environments.core import Environment

# Sources:
# https://github.com/CohenLabPrinceton/Ventilator-Dev/blob/master/sandbox/HOWTO_RunController.ipynb
# https://github.com/CohenLabPrinceton/Ventilator-Dev/blob/master/vent/controller/control_module.py
# https://github.com/MinRegret/venti
def PropValve(x):  # copied from Controller.__SimulatedPropValve
    y = 3 * x
    y_torch = torch.from_numpy(np.array([0.03 * (y - 130)])).float()
    flow_new = 1.0 * (torch.tanh(y_torch) + 1)

    return torch.clamp(flow_new, 0.0, 1.72)


def Solenoid(x):  # copied from Controller.__SimulatedSolenoid
    if x > 0:
        return x / x
    else:
        return x * 0.

class DoubleCompLung(Environment):
    def __init__(self, resistances=(0.1, 0.1), compliances=(0.1,0.1), min_volume=5, peep_value=5, **kwargs):
        # dynamics hyperparameters
        self.min_volume = min_volume
        self.P0 = peep_value
        self.peep_value = peep_value
        self.R1 = resistances[0]
        self.R2 = resistances[1]
        self.C1 = compliances[0]
        self.C2 = compliances[1]
        self.flow_pvs = 0.0
        self.flow = 0.0
```

SingleComp R, C Parameter Search

In [1]:
```
%load_ext autoreload
%autoreload 2
```

In [2]:
```
import warnings
warnings.filterwarnings('ignore')
```

In [3]:
```
import sys
sys.path.append("..")
import venti

import tqdm

import pickle
from glob import glob
import torch
from venti.utils import *
from venti.utils.sim_testing import *

from venti.environments import StitchedSim
from venti.controllers import PID

import matplotlib.pyplot as plt
import numpy as np

from datetime import date
from venti.environments._balloon_lung_nimra import BalloonLungNimra
```

In [12]:
```
from venti.environments._single_comp_lung import SingleCompLung
```

In [13]:
```
def plot_sim_diagnostics_nimra(open_loop_summary=None, forecast_summary=None, run=None):
    plt.rc('figure', figsize=(8, 8))

    plt.subplot(211)  ##### open-loop test
    for preds, truth in open_loop_summary['all_runs']:
        plt.plot(truth - preds, 'gray', alpha=0.1)
    plt.title('open-loop test')
    plt.ylabel('residual')
    plot_spray(open_loop_summary['all_runs'])
    plot_error_metric(open_loop_summary['mae'], label="MAE")


    plt.tight_layout()
    plt.legend()
```

Parameter Search Function

In [25]:
```
def search_parameters_single_comp(R_vals=[], C_vals=[],
                                  munger_R=5.0, munger_C=10.0, munger_PEEP=5, data_type='resid'):

    data_type = 'resid'
    data_runs = sorted(glob(f'{venti.ROOT}/data/R{munger_R}_C{munger_C}_PEEP{munger_PEEP}/{data_
    munger = Munger(data_runs)
    munger_path = f"{venti.ROOT}/pkls/mungers/R{munger_R}_C{munger_C}_PEEP{munger_PEEP}/{today}-
    munger.save(munger_path)

    MAEs = []
    summaries = []
    RC_vals = []

    print("Running grid")
    for R in tqdm.tqdm(R_vals, leave=False):
        for C in tqdm.tqdm(C_vals, leave=False):
            sim = SingleCompLung(resistance=R, compliance=C)
            open_loop_summary = open_loop_test(sim, munger_path)
            MAEs.append(open_loop_summary['mae'].mean())
```

```
            RC_vals.append((R, C))
            summaries.append(open_loop_summary)

    min_mae = min(MAEs)
    idx = MAEs.index(min_mae)
    min_R, min_C = RC_vals[idx]

    print(f"Open Loop Avg. Min MAE: {min_mae}")
    print(f"Min Resistance: {min_R}, Min Compliance: {min_C}")

    return [min_mae, min_R, min_C]
```

Parameter Search - Full range of R and C

In [26]:

```
Rs_search = np.concatenate((np.linspace(0, 0.9, 10), np.linspace(1, 50, 10)))
Cs_search = np.concatenate((np.linspace(0, 0.9, 10), np.linspace(1, 50, 10)))

munger_Rs = [5, 20, 50]
munger_Cs = [10, 20, 50]
```

In [ ]:

```
results = []

for R in munger_Rs:
    for C in munger_Cs:
        print(f"#################### Running parameter search for: R = {R}, C={C}")
        min_mae, min_R, min_C = search_parameters_single_comp(R_vals=Rs_search, C_vals=Cs_search
        print(f"For munger R = {R}, C = {C} ----- Best R = {min_R}, Best C = {min_C}, with MAE =

        result_dict = {
            "R, C": (R, C),
            "min_R": min_R,
            "min_C": min_C,
            "min_mae": min_mae,
        }

        results.append(result_dict)
```

Print R, C mappings

In [37]:

```
for i in range(len(results)):
    R, C = results[i]["R, C"]
    min_R = results[i]["min_R"]
    min_C = results[i]["min_C"]
    min_mae = results[i]["min_mae"]

    print(f"Munger R = {R}, C = {C} ----------> Sim R = {min_R}, Sim C = {min_C} ||| average MAE
```

```
Munger R = 5, C = 10 ----------> Sim R = 6.444444444444445, Sim C = 0.1 ||| average MAE = 10.974
923749077918
Munger R = 5, C = 20 ----------> Sim R = 6.444444444444445, Sim C = 0.1 ||| average MAE = 5.9895
39885597685
Munger R = 5, C = 50 ----------> Sim R = 0.30000000000000004, Sim C = 0.1 ||| average MAE = 2.81
1355288234108
Munger R = 20, C = 10 ----------> Sim R = 11.88888888888889, Sim C = 0.1 ||| average MAE = 10.36
7622391157123
Munger R = 20, C = 20 ----------> Sim R = 11.88888888888889, Sim C = 0.1 ||| average MAE = 9.406
506356788004
Munger R = 20, C = 50 ----------> Sim R = 6.444444444444445, Sim C = 0.1 ||| average MAE = 4.679
01774038625
Munger R = 50, C = 10 ----------> Sim R = 22.77777777777778, Sim C = 0.1 ||| average MAE = 6.207
538048544716
Munger R = 50, C = 20 ----------> Sim R = 22.77777777777778, Sim C = 0.1 ||| average MAE = 10.35
2234595743848
Munger R = 50, C = 50 ----------> Sim R = 50.0, Sim C = 0.1 ||| average MAE = 8.087768698490269
```

Open Loop Tests for Balloon Lung and Stitched Sim

In [1]:
```
%load_ext autoreload
%autoreload 2
```

In [2]:
```
import warnings
warnings.filterwarnings('ignore')
```

In [17]:
```
import sys
sys.path.append("..")
import venti

import pickle
from glob import glob
import torch
from venti.utils import *
from venti.utils.sim_testing import *

from venti.environments import StitchedSim
from venti.controllers import PID

import matplotlib.pyplot as plt
import numpy as np

from datetime import date
from venti.environments._balloon_lung_nimra import BalloonLungNimra
```

In [14]:
```
munger_Rs = [5, 20, 50]
munger_Cs = [10, 20, 50]
PEEP = 5
```

Balloon Lung Average MAEs

In [18]:
```
results = []
summaries = []

for R in munger_Rs:
    for C in munger_Cs:
        print(f"#################### Running open loop test for: R = {R}, C={C}")

        data_type = 'resid'
        data_runs = sorted(glob(f'{venti.ROOT}/data/R{R}_C{C}_PEEP{PEEP}/{data_type}/*.pkl'))
        munger = Munger(data_runs)
        munger_path = f"{venti.ROOT}/pkls/mungers/R{R}_C{C}_PEEP{PEEP}/{today}-{data_type}.pkl"
        munger.save(munger_path)

        sim = BalloonLungNimra()
        open_loop_summary = open_loop_test(sim, munger_path)
        mean_mae = open_loop_summary['mae'].mean()

        print("Open Loop Avg. MAE:", mean_mae)

        result_dict = {
            "R, C": (R, C),
            "summary": open_loop_summary,
            "mean_mae": mean_mae,
        }

        results.append(result_dict)
```

```
#################### Running open loop test for: R = 5, C=10
Added 3120 breaths from 30 paths.
u_in: mean=[10.65289305], std=[16.38068453]
pressure: mean=[21.75751905], std=[10.64563329]
Open Loop Avg. MAE: 21.661304205932097
#################### Running open loop test for: R = 5, C=20
Added 3095 breaths from 30 paths.
u_in: mean=[22.03486641], std=[20.49261697]
pressure: mean=[18.12326043], std=[7.77808338]
```

```
Open Loop Avg. MAE: 17.015197906867474
#################### Running open loop test for: R = 5, C=50
Added 3121 breaths from 30 paths.
u_in: mean=[43.12984948], std=[27.47589867]
pressure: mean=[16.77403692], std=[7.0467063]
Open Loop Avg. MAE: 13.761171325482032
#################### Running open loop test for: R = 20, C=10
Added 3120 breaths from 30 paths.
u_in: mean=[10.48920412], std=[15.49283537]
pressure: mean=[21.6063747], std=[9.99803768]
Open Loop Avg. MAE: 21.381522980616424
#################### Running open loop test for: R = 20, C=20
Added 3120 breaths from 30 paths.
u_in: mean=[13.13677651], std=[12.31302745]
pressure: mean=[20.24810263], std=[9.04639217]
Open Loop Avg. MAE: 20.334621139163204
#################### Running open loop test for: R = 20, C=50
Added 3120 breaths from 30 paths.
u_in: mean=[26.90287047], std=[21.25684765]
pressure: mean=[18.38025257], std=[8.9848058]
Open Loop Avg. MAE: 17.4956432990933
#################### Running open loop test for: R = 50, C=10
Added 3120 breaths from 30 paths.
u_in: mean=[6.25215775], std=[5.46498441]
pressure: mean=[13.90609401], std=[7.36778683]
Open Loop Avg. MAE: 14.467191394444928
#################### Running open loop test for: R = 50, C=20
Added 3120 breaths from 30 paths.
u_in: mean=[10.88460745], std=[13.26222888]
pressure: mean=[21.71666659], std=[14.25470989]
Open Loop Avg. MAE: 20.877972941382836
#################### Running open loop test for: R = 50, C=50
Added 3120 breaths from 30 paths.
u_in: mean=[8.32686543], std=[5.97557228]
pressure: mean=[17.90996284], std=[8.45109544]
Open Loop Avg. MAE: 18.1494473862762
```

Stitched Sim Average MAEs

In [19]:

```
results = []
summaries = []

for R in munger_Rs:
    for C in munger_Cs:
        print(f"#################### Running open loop test for: R = {R}, C={C}")

        data_type = 'resid'
        data_runs = sorted(glob(f'{venti.ROOT}/data/R{R}_C{C}_PEEP{PEEP}/{data_type}/*.pkl'))
        munger = Munger(data_runs)
        munger_path = f"{venti.ROOT}/pkls/mungers/R{R}_C{C}_PEEP{PEEP}/{today}-{data_type}.pkl"
        munger.save(munger_path)

        sim_paths = sorted(glob(f'{venti.ROOT}/pkls/simulators/R{R}_C{C}_PEEP{PEEP}/*.pkl'))
        sim = StitchedSim.load(sim_paths[-1])
        open_loop_summary = open_loop_test(sim, munger_path)
        mean_mae = open_loop_summary['mae'].mean()

        print("Open Loop Avg. MAE:", mean_mae)

        result_dict = {
            "R, C": (R, C),
            "summary": open_loop_summary,
            "mean_mae": mean_mae,
        }

        results.append(result_dict)
```

```
#################### Running open loop test for: R = 5, C=10
Added 3120 breaths from 30 paths.
u_in: mean=[10.65289305], std=[16.38068453]
pressure: mean=[21.75751905], std=[10.64563329]
Open Loop Avg. MAE: 0.41520782881916507
#################### Running open loop test for: R = 5, C=20
Added 3095 breaths from 30 paths.
u_in: mean=[22.03486641], std=[20.49261697]
pressure: mean=[18.12326043], std=[7.77808338]
Open Loop Avg. MAE: 0.2724695607671767
#################### Running open loop test for: R = 5, C=50
Added 3121 breaths from 30 paths.
u_in: mean=[43.12984948], std=[27.47589867]
```

```
pressure: mean=[16.77403692], std=[7.0467063]
Open Loop Avg. MAE: 0.8530337748968541
#################### Running open loop test for: R = 20, C=10
Added 3120 breaths from 30 paths.
u_in: mean=[10.48920412], std=[15.49283537]
pressure: mean=[21.6063747], std=[9.99803768]
Open Loop Avg. MAE: 0.5874404084612336
#################### Running open loop test for: R = 20, C=20
Added 3120 breaths from 30 paths.
u_in: mean=[13.13677651], std=[12.31302745]
pressure: mean=[20.24810263], std=[9.04639217]
Open Loop Avg. MAE: 0.7765486710527449
#################### Running open loop test for: R = 20, C=50
Added 3120 breaths from 30 paths.
u_in: mean=[26.90287047], std=[21.25684765]
pressure: mean=[18.38025257], std=[8.9848058]
Open Loop Avg. MAE: 4.305978587157675
#################### Running open loop test for: R = 50, C=10
Added 3120 breaths from 30 paths.
u_in: mean=[6.25215775], std=[5.46498441]
pressure: mean=[13.90609401], std=[7.36778683]
Open Loop Avg. MAE: 0.6104861703249537
#################### Running open loop test for: R = 50, C=20
Added 3120 breaths from 30 paths.
u_in: mean=[10.88460745], std=[13.26222888]
pressure: mean=[21.71666659], std=[14.25470989]
Open Loop Avg. MAE: 6.002318653647297
#################### Running open loop test for: R = 50, C=50
Added 3120 breaths from 30 paths.
u_in: mean=[8.32686543], std=[5.97557228]
pressure: mean=[17.90996284], std=[8.45109544]
Open Loop Avg. MAE: 2.956697014289328
```

PID grid search on the 3 simulations

In [1]:
```
%load_ext autoreload
%autoreload 2
```

In [3]:
```
import sys
sys.path.append("..")

import venti

from venti.controllers import PID
from venti.environments import PhysicalLung, StitchedSim
from venti.utils import Analyzer, BreathWaveform
from venti.experimental.core import run_calibration

from venti.experimental.pid_grid import pid_grid

import torch
import numpy as np

from glob import glob

from venti.environments._single_comp_lung_pid import SingleCompLung
from venti.environments._double_comp_lung_pid import DoubleCompLung
```

Find best PID function

In [3]:
```
def find_pid_on_sim(R= 6.4, C=0.1, sim_type="single_comp"):

    if sim_type is "single_comp":
        # Create single compartment lung with R, C vals
        sim = SingleCompLung(resistance=R, compliance=C)
    elif sim_type is "stitched_sim":
        # Create stitched sim with R, C vals
        sim_paths = sorted(glob(f'{venti.ROOT}/pkls/simulators/R{R}_C{C}_PEEP{PEEP}/*.pkl'))
        sim = StitchedSim.load(sim_paths[-1])
        sim.to_round = False
    else:
        raise ValueError("Invalid Simulation Type - please choose single_comp or stiched_sim")

    print("Running grid...")
    # Run pid grid search on single compartment lung
    data_dicts = pid_grid(R=R, C=C, PEEP=PEEP, T=300, abort=70,
                          directory=f"{venti.ROOT}/data/R{R}_C{C}_PEEP{PEEP}/pid", vent=sim)

    print("Finished grid")

    print(f"Type of return data from pid_grid = {type(data_dicts)}")
    analyzers = [Analyzer(d) for d in data_dicts]
    print(f"Type of analyzers = {type(analyzers)}")

    # For each analyzer, calculate mean loss per breath
    mean_losses = [a.default_metric() for a in analyzers]
    min_loss = min(mean_losses)
    print(f"Minimum Loss = {min_loss}")

    # Get analyzer with the lowest loss per breath
    idx = mean_losses.index(min_loss)
    best_analyzer = analyzers[idx]

    # Get controller from the best analyzer
    best_pid = best_analyzer.controller
    best_params = best_pid.K

    print(f"Best PID parameters = {best_params}")

    return [best_pid, best_params, min_loss]
```

In [4]:
```
R_map = {
    5: {
```

```
        10: 6.44,
        20: 6.44,
        50: 0.30,
    },
    20: {
        10: 11.88,
        20: 11.88,
        50: 6.44,
    },
    50: {
        10: 22.77,
        20: 22.77,
        50: 50.0,
    },
}

C_map = 0.1
```

In [5]:

```
R_vals = [5, 20, 50]
C_vals = [10, 20, 50]
PEEP = 5
```

In [ ]:

```
results = []

for R in R_vals:
    for C in C_vals:
        pid_A, params_A, loss_A = find_pid_on_sim(R= R_map[R][C], C=C_map, sim_type="single_comp
        pid_B, params_B, loss_B = find_pid_on_sim(R=R, C=C, sim_type="stitched_sim")

        results.append([R, C, pid_A, params_A, loss_A , pid_B, params_B, loss_B])
```

In [7]:

```
for result in results:
    print(f"############ R = {result[0]}, C = {result[1]}:")
    print(f"Best Params Single Comp = {result[3]}, Loss = {result[4]}. ")
    print(f"Best Params Stitched Sim = {result[6]}, Loss = {result[7]}. ")
```

```
############ R = 5, C = 10:
Best Params Single Comp = tensor([10., 10.,  0.], requires_grad=True), Loss = 1.828948532559298
7.
Best Params Stitched Sim = tensor([7., 0., 0.], requires_grad=True), Loss = 0.3170687313793661.
############ R = 5, C = 20:
Best Params Single Comp = tensor([10., 10.,  0.], requires_grad=True), Loss = 1.828948532559298
7.
Best Params Stitched Sim = tensor([8., 0., 0.], requires_grad=True), Loss = 0.36621521190843903.
############ R = 5, C = 50:
Best Params Single Comp = tensor([ 8., 10.,  0.], requires_grad=True), Loss = 0.1709742026272147
4.
Best Params Stitched Sim = tensor([10.,  0.,  0.], requires_grad=True), Loss = 0.819378676985580
2.
############ R = 20, C = 10:
Best Params Single Comp = tensor([10.,  6.,  0.], requires_grad=True), Loss = 1.845227058846326
4.
Best Params Stitched Sim = tensor([7.0000, 0.5000, 0.0000], requires_grad=True), Loss = 0.370475
2990963033.
############ R = 20, C = 20:
Best Params Single Comp = tensor([10.,  6.,  0.], requires_grad=True), Loss = 1.845227058846326
4.
Best Params Stitched Sim = tensor([3., 9., 0.], requires_grad=True), Loss = 0.5783507220164894.
############ R = 20, C = 50:
Best Params Single Comp = tensor([10., 10.,  0.], requires_grad=True), Loss = 1.828948532559298
7.
Best Params Stitched Sim = tensor([8., 7., 0.], requires_grad=True), Loss = 0.48784139391228876.
############ R = 50, C = 10:
Best Params Single Comp = tensor([10.,  0.,  0.], requires_grad=True), Loss = 1.725866233242557
2.
Best Params Stitched Sim = tensor([4., 0., 0.], requires_grad=True), Loss = 0.39172225328388033.
############ R = 50, C = 20:
Best Params Single Comp = tensor([10.,  0.,  0.], requires_grad=True), Loss = 1.725866233242557
2.
Best Params Stitched Sim = tensor([2., 0., 0.], requires_grad=True), Loss = 0.35501044213395233.
############ R = 50, C = 50:
Best Params Single Comp = tensor([10.,  0.,  0.], requires_grad=True), Loss = 2.076162564347665
6.
Best Params Stitched Sim = tensor([3., 3., 0.], requires_grad=True), Loss = 0.6214030085852322.
```

Run for Stitched Sim

In [ ]:
```
results = []

for R in R_vals:
    for C in C_vals:
        pid_B, params_B, loss_B = find_pid_on_sim(R=R, C=C, sim_type="stitched_sim")

        results.append([R, C, pid_B, params_B, loss_B])
```

In [9]:
```
for result in results:
    print(f"############ R = {result[0]}, C = {result[1]}:")
    print(f"Best Params Stitched Sim = {result[3]}, Loss = {result[4]}. ")
```

```
############ R = 5, C = 10:
Best Params Stitched Sim = tensor([7., 0., 0.], requires_grad=True), Loss = 0.3170687313793661.
############ R = 5, C = 20:
Best Params Stitched Sim = tensor([8., 0., 0.], requires_grad=True), Loss = 0.36621521190843903.
############ R = 5, C = 50:
Best Params Stitched Sim = tensor([10.,  0.,  0.], requires_grad=True), Loss = 0.819378676985580
2.
############ R = 20, C = 10:
Best Params Stitched Sim = tensor([7.0000, 0.5000, 0.0000], requires_grad=True), Loss = 0.370475
2990963033.
############ R = 20, C = 20:
Best Params Stitched Sim = tensor([3., 9., 0.], requires_grad=True), Loss = 0.5783507220164894.
############ R = 20, C = 50:
Best Params Stitched Sim = tensor([8., 7., 0.], requires_grad=True), Loss = 0.48784139391228876.
############ R = 50, C = 10:
Best Params Stitched Sim = tensor([4., 0., 0.], requires_grad=True), Loss = 0.39172225328388033.
############ R = 50, C = 20:
Best Params Stitched Sim = tensor([2., 0., 0.], requires_grad=True), Loss = 0.35501044213395233.
############ R = 50, C = 50:
Best Params Stitched Sim = tensor([3., 3., 0.], requires_grad=True), Loss = 0.6214030085852322.
```

Best PID on Double Comp

In [4]:
```
def find_pid_on_double(R1=0.0, R2=0.0, C1=0.1, C2=0.0):
    sim = DoubleCompLung(resistances=(R1,R2), compliances=(C1,C2))


    print("Running grid...")
    # Run pid grid search on double compartment lung
    # here I pass in R=R1 and C=C1 just because these values are not actually used in the PID gr
    # they are just stored in the result analyzer, from which I won't need to use the R and C va
    data_dicts = pid_grid(R=R1, C=C1, PEEP=PEEP, T=300, abort=70,
                          directory=f"{venti.ROOT}/data/R{R}_C{C}_PEEP{PEEP}/pid", vent=sim)

    print("Finished grid")

    print(f"Type of return data from pid_grid = {type(data_dicts)}")
    analyzers = [Analyzer(d) for d in data_dicts]
    print(f"Type of analyzers = {type(analyzers)}")

    # For each analyzer, calculate mean loss per breath
    mean_losses = [a.default_metric() for a in analyzers]
    min_loss = min(mean_losses)
    print(f"Minimum Loss = {min_loss}")

    # Get analyzer with the lowest loss per breath
    idx = mean_losses.index(min_loss)
    best_analyzer = analyzers[idx]

    # Get controller from the best analyzer
    best_pid = best_analyzer.controller
    best_params = best_pid.K

    print(f"Best PID parameters = {best_params}")

    return [best_pid, best_params, min_loss]
```

In [7]:
```
R1_map = {
            5: {
                10: 0.,
                20: 0.,
                50: 0.6,
            },
            20: {
                10: 0.6,
                20: 0.6,
                50: 0.6,
            },
            50: {
                10: 0.,
                20: 0.6,
                50: 0.6,
            },
        }

R2_map = {
            5: {
                10: 0.,
                20: 0.,
                50: 0.,
            },
            20: {
                10: 0.,
                20: 0.,
                50: 0.,
            },
            50: {
                10: 0.,
                20: 0.,
                50: 0.,
            },
        }

C1_map = {
            5: {
                10: 0.24,
                20: 0.24,
                50: 0.36,
            },
            20: {
                10: 0.24,
                20: 0.24,
                50: 0.24,
            },
            50: {
                10: 0.36,
                20: 0.24,
                50: 0.24,
            },
        }

C2_map = {
            5: {
                10: 0.6,
                20: 0.6,
                50: 0.6,
            },
            20: {
                10: 0.6,
                20: 0.6,
                50: 0.6,
            },
            50: {
                10: 0.36,
                20: 0.6,
                50: 0.6,
            },
        }
```

In [8]:
```
R_vals = [5, 20, 50]
C_vals = [10, 20, 50]
PEEP = 5
```

In [ ]:
```
results = []

for R in R_vals:
    for C in C_vals:
        pid, params, loss = find_pid_on_double(R1=R1_map[R][C], R2=R2_map[R][C], C1=C1_map[R][C]

        results.append([R, C, pid, params, loss])
```

In [11]:
```
for result in results:
    print(f"############ R = {result[0]}, C = {result[1]}:")
    print(f"Best Params Double Comp = {result[3]}, Loss = {result[4]}. ")
```

```
############ R = 5, C = 10:
Best Params Double Comp = tensor([0., 0., 0.], requires_grad=True), Loss = 1.0805292962798454.
############ R = 5, C = 20:
Best Params Double Comp = tensor([0., 0., 0.], requires_grad=True), Loss = 1.0805292962798454.
############ R = 5, C = 50:
Best Params Double Comp = tensor([10., 10.,  0.], requires_grad=True), Loss = 1.150542163955951
2.
############ R = 20, C = 10:
Best Params Double Comp = tensor([0., 0., 0.], requires_grad=True), Loss = 1.0822698637479515.
############ R = 20, C = 20:
Best Params Double Comp = tensor([0., 0., 0.], requires_grad=True), Loss = 1.0822698637479515.
############ R = 20, C = 50:
Best Params Double Comp = tensor([0., 0., 0.], requires_grad=True), Loss = 1.0822698637479515.
############ R = 50, C = 10:
Best Params Double Comp = tensor([10., 10.,  0.], requires_grad=True), Loss = 1.148082774518183
9.
############ R = 50, C = 20:
Best Params Double Comp = tensor([0., 0., 0.], requires_grad=True), Loss = 1.0822698637479515.
############ R = 50, C = 50:
Best Params Double Comp = tensor([0., 0., 0.], requires_grad=True), Loss = 1.0822698637479515.
```

Best PID on Mechanical Lung

In [1]:
```
%load_ext autoreload
%autoreload 2

import sys
sys.path.append("..")


import venti
from venti.experimental.best_pid import best_pid, plot_pid, global_best_pid
from venti.utils import Analyzer
from venti.controllers import PID

import matplotlib.pyplot as plt
import numpy as np

from datetime import date

import warnings
import itertools
warnings.filterwarnings('ignore')
```

In [ ]:
```
import os
import torch

DEFAULT_DATA_DIR = f"{venti.ROOT}/data"
directory=DEFAULT_DATA_DIR

p_mask=lambda x: True
i_mask=lambda x: True
d_mask=lambda x: x == 0

loss_fn=np.abs
```

In [3]:
```
today = date.today()
```

Choose lung settings:

In [20]:
```
Rs = [5, 20, 50]
Cs = [10, 20, 50]
PEEP = 5
```

Run loops:

In [24]:
```
results = []

for R in Rs:
    for C in Cs:
        losses = {}

        pid_dir = os.path.join(directory, f"R{R}_C{C}_PEEP{PEEP}", "pid")
        for path in tqdm.tqdm(os.listdir(pid_dir)):
            if path[-4:] == ".pkl":
                try:
                    path = os.path.join(pid_dir, path)
                    analyzer = Analyzer(path)

                    K = analyzer.controller.K
                    if isinstance(K, torch.Tensor):
                        K = K.detach().numpy()
                    K = tuple(K.tolist())
                    if not (p_mask(K[0]) and i_mask(K[1]) and d_mask(K[2])):
                        continue
                    if K not in losses:
                        losses[K] = []
                    loss = analyzer.default_metric(loss_fn=loss_fn)
                    losses[K].append(loss)
                except Exception as e:
                    print(path, e)
```

```
        keys = list(losses.keys())
        losses = [min(losses[key]) for key in keys]
        idxs = np.argsort(losses)

        min_K = keys[idxs[0]]
        min_loss = losses[idxs[0]]

        results.append([("R", R), ("C", C), min_K, min_loss])
```

```
100%|██████████| 3736/3736 [00:07<00:00, 484.69it/s]
100%|██████████| 3651/3651 [00:07<00:00, 491.40it/s]
100%|██████████| 3536/3536 [00:07<00:00, 455.79it/s]
100%|██████████| 6490/6490 [00:13<00:00, 484.77it/s]
100%|██████████| 3582/3582 [00:08<00:00, 413.45it/s]
100%|██████████| 3602/3602 [00:08<00:00, 439.97it/s]
100%|██████████| 3649/3649 [00:09<00:00, 405.29it/s]
100%|██████████| 3607/3607 [00:09<00:00, 371.97it/s]
100%|██████████| 3607/3607 [00:09<00:00, 384.84it/s]
```

Print results:

In [25]:

```
for r in results:
    print(f"R = {r[0][1]}, C = {r[1][1]}:")
    print(f"\t \t K = {r[2]} \t Loss = {r[3]}")
```

```
R = 5, C = 10:
                K = (10.0, 5.0, 0.0)    Loss = 0.26153605291622123
R = 5, C = 20:
                K = (8.0, 0.0, 0.0)     Loss = 0.36621521190843903
R = 5, C = 50:
                K = (10.0, 0.1, 0.0)    Loss = 0.8199037145824501
R = 20, C = 10:
                K = (9.0, 0.9, 0.0)     Loss = 0.24058040698534097
R = 20, C = 20:
                K = (10.0, 0.6000000000000001, 0.0)     Loss = 0.4668655103077592
R = 20, C = 50:
                K = (8.0, 7.0, 0.0)     Loss = 0.48784139391228876
R = 50, C = 10:
                K = (4.0, 0.1, 0.0)     Loss = 0.3924529362028494
R = 50, C = 20:
                K = (2.0, 0.0, 0.0)     Loss = 0.35501044213395233
R = 50, C = 50:
                K = (3.0, 4.0, 0.0)     Loss = 0.6466410698953707
```

Performance of Simulator-tuned PIDs on mechanical lung

In [1]:
```
%load_ext autoreload
%autoreload 2
```

In [2]:
```
import sys
sys.path.append("..")


import venti
from venti.experimental.best_pid import best_pid, plot_pid, global_best_pid
from venti.utils import Analyzer
from venti.controllers import PID

import matplotlib.pyplot as plt
import numpy as np

from datetime import date

import warnings
import itertools
warnings.filterwarnings('ignore')


import os
import tqdm
import torch
```

In [3]:
```
today = date.today()
```

Range of Mechanical Lung Parameters:

In [4]:
```
Rs = [5, 20, 50]
Cs = [10, 20, 50]
PEEP = 5
```

In [5]:
```
RC_vals = [
    (5, 10),
    (5, 20),
    (5, 50),
    (20, 10),
    (20, 20),
    (20, 50),
    (50, 10),
    (50, 20),
    (50, 50),

]
```

Get all PID losses for mech lung

In [6]:
```
DEFAULT_DATA_DIR = f"{venti.ROOT}/data"
directory=DEFAULT_DATA_DIR

p_mask=lambda x: True
i_mask=lambda x: True
d_mask=lambda x: x == 0

loss_fn=np.abs
```

In [7]:
```
results = {}

for R in Rs:
    for C in Cs:
        losses = {}
```

```
        pid_dir = os.path.join(directory, f"R{R}_C{C}_PEEP{PEEP}", "pid")
        for path in tqdm.tqdm(os.listdir(pid_dir)):
            if path[-4:] == ".pkl":
                try:
                    path = os.path.join(pid_dir, path)
                    analyzer = Analyzer(path)

                    K = analyzer.controller.K
                    if isinstance(K, torch.Tensor):
                        K = K.detach().numpy()
                    K = tuple(K.tolist())
                    if not (p_mask(K[0]) and i_mask(K[1]) and d_mask(K[2])):
                        continue
                    if K not in losses:
                        losses[K] = []
                    loss = analyzer.default_metric(loss_fn=loss_fn)
                    losses[K].append(loss)
                except Exception as e:
                    print(path, e)

        keys = list(losses.keys())
        losses = [min(losses[key]) for key in keys]
        idxs = np.argsort(losses)

        min_K = keys[idxs[0]]
        min_loss = losses[idxs[0]]

        results[(R, C)] = {"Ks": keys, "Losses": losses}
```

```
100%|██████████| 7617/7617 [00:19<00:00, 389.89it/s]
100%|██████████| 7751/7751 [00:19<00:00, 400.22it/s]
100%|██████████| 7784/7784 [00:18<00:00, 415.78it/s]
100%|██████████| 10743/10743 [00:27<00:00, 386.10it/s]
100%|██████████| 7804/7804 [00:18<00:00, 412.01it/s]
100%|██████████| 8977/8977 [00:22<00:00, 391.35it/s]
100%|██████████| 8343/8343 [00:19<00:00, 418.72it/s]
100%|██████████| 8186/8186 [00:19<00:00, 425.95it/s]
100%|██████████| 8062/8062 [00:19<00:00, 403.58it/s]
```

In [8]:

```
pids_mech_lung = results

print(type(pids_mech_lung))
print(pids_mech_lung.keys())
```

```
<class 'dict'>
dict_keys([(5, 10), (5, 20), (5, 50), (20, 10), (20, 20), (20, 50), (50, 10), (50, 20), (50, 5
0)])
```

Function to return loss for given PID params

In [9]:

```
def get_pid_loss(R=50, C=10, PEEP=5, K=(2.0, 10.0, 0.0), pids=None):

    pid = pids[(R, C)]
    keys = pid["Ks"]
    losses = pid["Losses"]

    idx = keys.index(K)
    return losses[idx]
```

Function to return dictionary of losses for each R, C setting given a list of PID param tuples

In [10]:

```
def get_all_pid_losses(
    RC_vals=[(5, 10),(5, 20),(5, 50),(20, 10),(20, 20),(20, 50),(50, 10),(50, 20),(50, 50)],
    PEEP=5,
    Ks= [ #default values are from single comp
        (10., 10., 0.),
        (10., 10., 0.),
        (8., 10., 0.),
        (10., 6., 0.),
        (10., 6., 0.),
        (10., 10., 0.),
        (10., 0., 0.),
        (10., 0., 0.),
        (10., 0., 0.),
    ],
```

```
    pids=None,
):
    results = {}
    for i in range(len(RC_vals)):
        R, C = RC_vals[i]
        K = Ks[i]
        loss = get_pid_loss(R=R, C=C, PEEP=PEEP, K=K, pids=pids)
        if loss is None:
            print(f"ERROR IN R = {R}, C = {C}, K = {K}. No PID run with these parameters.")
            continue

        results[(R, C)] = {
            "K": K,
            "loss": loss,
        }

    return results
```

PID parameters from Simulations

Single Compartment Simulation

In [11]:
```
Ks_single_comp = [
    (10., 10., 0.),
    (10., 10., 0.),
    (8., 10., 0.),

    (10., 6., 0.),
    (10., 6., 0.),
    (10., 10., 0.),

    (10., 0., 0.),
    (10., 0., 0.),
    (10., 0., 0.),
]
```

In [12]:
```
results_single_comp = get_all_pid_losses(RC_vals=RC_vals, Ks=Ks_single_comp, pids=pids_mech_lung
```

In [13]:
```
## PRINTING CODE

for R in Rs:
    for C in Cs:
        r = results_single_comp[(R, C)]
        K = r["K"]
        loss = r["loss"]
        print(f"R = {R}, C = {C}, K = {K}, Loss = {loss}")
```

```
R = 5, C = 10, K = (10.0, 10.0, 0.0), Loss = 0.3265919751155273
R = 5, C = 20, K = (10.0, 10.0, 0.0), Loss = 0.49515854121594993
R = 5, C = 50, K = (8.0, 10.0, 0.0), Loss = 0.8998992869479421
R = 20, C = 10, K = (10.0, 6.0, 0.0), Loss = 0.3851909485374241
R = 20, C = 20, K = (10.0, 6.0, 0.0), Loss = 0.5459249687877059
R = 20, C = 50, K = (10.0, 10.0, 0.0), Loss = 0.5137771210037329
R = 50, C = 10, K = (10.0, 0.0, 0.0), Loss = 0.607466637171203
R = 50, C = 20, K = (10.0, 0.0, 0.0), Loss = 1.252930518744265
R = 50, C = 50, K = (10.0, 0.0, 0.0), Loss = 1.5083727956077166
```

Double Compartment Simulation Old Vals

In [14]:
```
Ks_double_comp = [
    (0., 0., 0.),
    (10., 10., 0.),
    (10., 10., 0.),

    (0., 0., 0.),
    (0., 0., 0.),
    (10., 10., 0.),

    (10., 10., 0.),
    (10., 10., 0.),
    (10., 0., 0.),
]
```

In [15]:
```
results_double_comp = get_all_pid_losses(RC_vals=RC_vals, Ks=Ks_double_comp, pids=pids_mech_lung
```

In [16]:
```
## PRINTING CODE

for R in Rs:
    for C in Cs:
        r = results_double_comp[(R, C)]
        K = r["K"]
        loss = r["loss"]
        print(f"R = {R}, C = {C}, K = {K}, Loss = {loss}")
```

```
R = 5, C = 10, K = (0.0, 0.0, 0.0), Loss = 3.622606125066785
R = 5, C = 20, K = (10.0, 10.0, 0.0), Loss = 0.49515854121594993
R = 5, C = 50, K = (10.0, 10.0, 0.0), Loss = 0.8614060581550584
R = 20, C = 10, K = (0.0, 0.0, 0.0), Loss = 4.057501728292121
R = 20, C = 20, K = (0.0, 0.0, 0.0), Loss = 3.594035209726561
R = 20, C = 50, K = (10.0, 10.0, 0.0), Loss = 0.5137771210037329
R = 50, C = 10, K = (10.0, 10.0, 0.0), Loss = 0.7099195201904045
R = 50, C = 20, K = (10.0, 10.0, 0.0), Loss = 1.0787012985570885
R = 50, C = 50, K = (10.0, 0.0, 0.0), Loss = 1.5083727956077166
```

Double Compartment Simulation New Vals

In [17]:
```
Ks_double_comp = [
    (0., 0., 0.),
    (0., 0., 0.),
    (10., 10., 0.),

    (0., 0., 0.),
    (0., 0., 0.),
    (0., 0., 0.),

    (10., 10., 0.),
    (0., 0., 0.),
    (0., 0., 0.),
]
```

In [18]:
```
results_double_comp = get_all_pid_losses(RC_vals=RC_vals, Ks=Ks_double_comp, pids=pids_mech_lung
```

In [19]:
```
## PRINTING CODE

for R in Rs:
    for C in Cs:
        r = results_double_comp[(R, C)]
        K = r["K"]
        loss = r["loss"]
        print(f"R = {R}, C = {C}, K = {K}, Loss = {loss}")
```

```
R = 5, C = 10, K = (0.0, 0.0, 0.0), Loss = 3.622606125066785
R = 5, C = 20, K = (0.0, 0.0, 0.0), Loss = 1.0805292962798454
R = 5, C = 50, K = (10.0, 10.0, 0.0), Loss = 0.8614060581550584
R = 20, C = 10, K = (0.0, 0.0, 0.0), Loss = 4.057501728292121
R = 20, C = 20, K = (0.0, 0.0, 0.0), Loss = 3.594035209726561
R = 20, C = 50, K = (0.0, 0.0, 0.0), Loss = 1.0822698637479515
R = 50, C = 10, K = (10.0, 10.0, 0.0), Loss = 0.7099195201904045
R = 50, C = 20, K = (0.0, 0.0, 0.0), Loss = 3.130421358984061
R = 50, C = 50, K = (0.0, 0.0, 0.0), Loss = 3.463323403417923
```

Stitched Sim

In [20]:
```
Ks_stitched_sim = [
    (7., 0., 0.),
    (8., 0., 0.),
    (10., 0., 0.),
    (7., 0.5, 0.),
    (3., 9., 0.),
    (8., 7., 0.),
    (4., 0., 0.),
    (2., 0., 0.),
    (3., 3., 0.),
]
```

In [21]:
```
results_stitched_sim = get_all_pid_losses(RC_vals=RC_vals, Ks=Ks_stitched_sim, pids=pids_mech_lu
```

Mechanical Lung

In [22]:
```
Ks_mech = [
    (10., 5.0, 0.),
    (8., 0., 0.),
    (10., 0.1, 0.),
    (9., 0.9, 0.),
    (10., 0.6000000000000001, 0.),
    (8., 7., 0.),
    (4., 0.1, 0.),
    (2., 0., 0.),
    (3., 4., 0.),
]
```

In [23]:
```
results_mech = get_all_pid_losses(RC_vals=RC_vals, Ks=Ks_mech, pids=pids_mech_lung)
```

Print performance of both for each R, C

In [24]:
```
## PRINTING CODE

for R in Rs:
    for C in Cs:
        r = results_single_comp[(R, C)]
        K = r["K"]
        loss = r["loss"]
        print(f"Single Comp: R = {R}, C = {C}, K = {K}, Loss = {loss}")

        r = results_double_comp[(R, C)]
        K = r["K"]
        loss = r["loss"]
        print(f"Double Comp: R = {R}, C = {C}, K = {K}, Loss = {loss}")

        r = results_stitched_sim[(R, C)]
        K = r["K"]
        loss = r["loss"]
        print(f"Stitched Sim: R = {R}, C = {C}, K = {K}, Loss = {loss}")


        r = results_mech[(R, C)]
        K = r["K"]
        loss = r["loss"]
        print(f"Mechanical Lung: R = {R}, C = {C}, K = {K}, Loss = {loss}")


        print("-----------------------------------------------------------")
```

```
Single Comp: R = 5, C = 10, K = (10.0, 10.0, 0.0), Loss = 0.3265919751155273
Double Comp: R = 5, C = 10, K = (0.0, 0.0, 0.0), Loss = 3.622606125066785
Stitched Sim: R = 5, C = 10, K = (7.0, 0.0, 0.0), Loss = 0.2725029999498823
Mechanical Lung: R = 5, C = 10, K = (10.0, 5.0, 0.0), Loss = 0.26153605291622123
-----------------------------------------------------------
Single Comp: R = 5, C = 20, K = (10.0, 10.0, 0.0), Loss = 0.49515854121594993
Double Comp: R = 5, C = 20, K = (0.0, 0.0, 0.0), Loss = 1.0805292962798454
Stitched Sim: R = 5, C = 20, K = (8.0, 0.0, 0.0), Loss = 0.36621521190843903
Mechanical Lung: R = 5, C = 20, K = (8.0, 0.0, 0.0), Loss = 0.36621521190843903
-----------------------------------------------------------
Single Comp: R = 5, C = 50, K = (8.0, 10.0, 0.0), Loss = 0.8998992869479421
Double Comp: R = 5, C = 50, K = (10.0, 10.0, 0.0), Loss = 0.8614060581550584
Stitched Sim: R = 5, C = 50, K = (10.0, 0.0, 0.0), Loss = 0.8193786769855802
Mechanical Lung: R = 5, C = 50, K = (10.0, 0.1, 0.0), Loss = 0.8199037145824501
-----------------------------------------------------------
Single Comp: R = 20, C = 10, K = (10.0, 6.0, 0.0), Loss = 0.3851909485374241
Double Comp: R = 20, C = 10, K = (0.0, 0.0, 0.0), Loss = 4.057501728292121
Stitched Sim: R = 20, C = 10, K = (7.0, 0.5, 0.0), Loss = 0.29164536747112363
Mechanical Lung: R = 20, C = 10, K = (9.0, 0.9, 0.0), Loss = 0.24058040698534097
-----------------------------------------------------------
Single Comp: R = 20, C = 20, K = (10.0, 6.0, 0.0), Loss = 0.5459249687877059
Double Comp: R = 20, C = 20, K = (0.0, 0.0, 0.0), Loss = 3.594035209726561
Stitched Sim: R = 20, C = 20, K = (3.0, 9.0, 0.0), Loss = 0.5783507220164894
Mechanical Lung: R = 20, C = 20, K = (10.0, 0.6000000000000001, 0.0), Loss = 0.4668655103077592
-----------------------------------------------------------
```

```
Single Comp: R = 20, C = 50, K = (10.0, 10.0, 0.0), Loss = 0.5137771210037329
Double Comp: R = 20, C = 50, K = (0.0, 0.0, 0.0), Loss = 1.0822698637479515
Stitched Sim: R = 20, C = 50, K = (8.0, 7.0, 0.0), Loss = 0.48784139391228876
Mechanical Lung: R = 20, C = 50, K = (8.0, 7.0, 0.0), Loss = 0.48784139391228876
----------------------------------------------------------
Single Comp: R = 50, C = 10, K = (10.0, 0.0, 0.0), Loss = 0.607466637171203
Double Comp: R = 50, C = 10, K = (10.0, 10.0, 0.0), Loss = 0.7099195201904045
Stitched Sim: R = 50, C = 10, K = (4.0, 0.0, 0.0), Loss = 0.5340001556988414
Mechanical Lung: R = 50, C = 10, K = (4.0, 0.1, 0.0), Loss = 0.3924529362028494
----------------------------------------------------------
Single Comp: R = 50, C = 20, K = (10.0, 0.0, 0.0), Loss = 1.252930518744265
Double Comp: R = 50, C = 20, K = (0.0, 0.0, 0.0), Loss = 3.130421358984061
Stitched Sim: R = 50, C = 20, K = (2.0, 0.0, 0.0), Loss = 0.35501044213395233
Mechanical Lung: R = 50, C = 20, K = (2.0, 0.0, 0.0), Loss = 0.35501044213395233
----------------------------------------------------------
Single Comp: R = 50, C = 50, K = (10.0, 0.0, 0.0), Loss = 1.5083727956077166
Double Comp: R = 50, C = 50, K = (0.0, 0.0, 0.0), Loss = 3.463323403417923
Stitched Sim: R = 50, C = 50, K = (3.0, 3.0, 0.0), Loss = 1.1612034102339202
Mechanical Lung: R = 50, C = 50, K = (3.0, 4.0, 0.0), Loss = 0.6466410698953707
----------------------------------------------------------
```